PERIYAR UNIVERSITY

(NAAC 'A++' Grade with CGPA 3.61 (Cycle - 3)
State University - NIRF Rank 56 - State Public University Rank 25
SALEM - 636 011

CENTRE FOR DISTANCE AND ONLINE EDUCATION
(CDOE)

MASTER OF COMPUTER APPLICATION
SEMESTER - II

CORE VII : DATA STRUCTURES AND ALGORITHMS

(Candidates admitted from 2024 onwards)

PERIYAR UNIVERSITY

CENTRE FOR DISTANCE AND ONLINE EDUCATION (CDOE)

M.C.A 2024 admission onwards

CORE – VII
Data Structures and Algorithms

Prepared by:

Centre for Distance and Online Education
Periyar University
Salem - 636011

SYLLABUS

DATA STRUCTURES AND ALGORITHMS

Unit I: **Abstract Data Types:** Introduction-Date Abstract Data Type – Bags-Iterators. **Arrays**: Array Structure-Python List-Two Dimensional Arrays-Matrix Abstract Data Type. **Sets, Maps:** Sets – Maps – Multi-Dimensional Arrays.
Unit II: **Algorithm Analysis:** Experimental Studies-Seven Functions-Asymptotic Analysis. **Recursion:** Illustrative Examples-Analyzing Recursive Algorithms-Linear Recursion- Binary Recursion- Multiple Recursion.
Unit III: **Stacks, Queues, and Deques:** Stacks – Queues – Double-Ended Queues Linked. **Lists:** Singly Linked Lists-Circularly Linked Lists-Doubly Linked Lists. **Trees:** General Trees-Binary Trees- Implementing Trees-Tree Traversal Algorithms
Unit IV: **Priority Queues:** Priority Queue Abstract Data Type- Implementing a Priority Queue – Heaps- Sorting with a Priority Queue. **Maps, Hash Tables, and Skip Lists:** Maps and Dictionaries-Hash Tables- Sorted Maps-Skip Lists-Sets, Multi Sets, and Multi Maps.
Unit V: Search Trees: Binary Search Trees-Balanced Search Trees-AVL Trees-Splay Trees. **Sorting and Selection:** Merge sort-Quick sort-Sorting through an Algorithmic Lens- Comparing Sorting Algorithms-Selection. **Graph Algorithms:** Graphs-Data Structures for Graphs-Graph Traversals- Shortest Paths-Minimum Spanning Trees.

TABLE OF CONTENTS

DATA STRUCTURES AND ALGORITHMS

UNIT 1 – INTRODUCTION
Abstract Data Types: Introduction-Date Abstract Data Type – Bags-Iterators. **Arrays**: Array Structure-Python List-Two Dimensional Arrays-Matrix Abstract Data Type. **Sets, Maps:** Sets – Maps – Multi-Dimensional Arrays.

Abstract Data Types

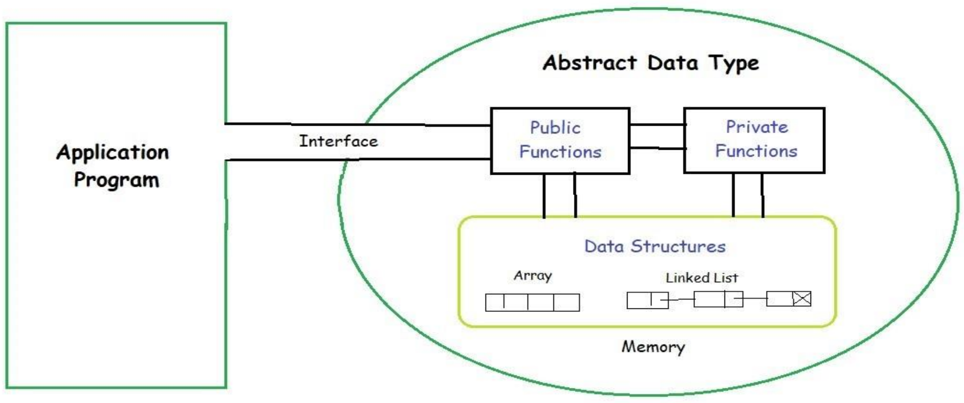
UNIT OBJECTIVES

In this course segment, we delve into foundational abstract data types (ADTs) and their practical implementations in Python. Beginning with the Date ADT, we explore methods for representing and manipulating dates, followed by an in-depth examination of Bags, an ADT allowing for element storage with possible repetitions. We then introduce the concept of iterators for efficient data traversal. Moving forward, we explore Arrays, focusing on their structure, dynamic resizing, and Python's list implementation. Additionally, we discuss Two-Dimensional Arrays, particularly matrices, and their significance. Finally, we cover Sets and Maps, understanding their properties, operations, and applications, alongside Multi-Dimensional Arrays, extending array concepts to higher dimensions. Through practical examples and exercises, students gain proficiency in using these fundamental data structures in Python.

SECTION 1.1: ABSTRACT DATA TYPE

An abstract data type (ADT) is a mathematical model for data types, defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented) It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view. The user of data type does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented) .



Abstract Data Types (ADTs) are a theoretical concept used in computer science to define data structures purely in terms of their behavior (operations) rather than their implementation. An ADT specifies:

- ✓ The types of data it can hold.
- ✓ The operations that can be performed on the data.
- ✓ The behavior of these operations (what they do, not how they do it).

Key Characteristics of ADTs:

- ✓ **Encapsulation:** ADTs hide the implementation details and expose only the operations necessary to interact with the data structure. This allows the implementation to change without affecting the code that uses the ADT.
- ✓ **Modularity:** ADTs promote a modular design where the implementation of data structures is separate from their usage. This enhances code readability and maintainability.

Examples of ADTs:

- ✓ **List ADT:** Defines operations such as insert, delete, and retrieve, which can be implemented using arrays, linked lists, or other structures.
- ✓ **Stack ADT:** Defines push and pop operations, typically implemented using arrays or linked lists.
- ✓ **Queue ADT:** Defines enqueue and dequeue operations, often implemented with circular arrays or linked lists.

Benefits of ADTs:

- ✓ **Improved Code Quality:** By defining clear interfaces and separating implementation details, ADTs help in creating robust and error-free code.
- ✓ **Reusability:** ADTs allow for the reuse of the same interface across different implementations, facilitating code reuse.
- ✓ **Ease of Maintenance:** Changes in implementation do not affect the overall system as long as the interface remains unchanged, simplifying maintenance.

Abstract Data Types in Python:

- ✓ Python, as a high-level programming language, supports the implementation of various ADTs through its built-in data structures and classes.
- ✓ Python lists, dictionaries, sets, and other collections can be used to implement ADTs efficiently.

ADTs provide a framework for defining and working with data structures by focusing on what operations are possible and how they behave, rather than on how these operations are implemented. This abstraction leads to better software design, enabling encapsulation, modularity, and ease of maintenance.

ABSTRACT DATA TYPES (ADTS) HAVE SEVERAL ADVANTAGES AND DISADVANTAGES

ADVANTAGES:

- **Encapsulation**: ADTs provide a way to encapsulate data and operations into a single unit, making it easier to manage and modify the data structure.
- **Abstraction**: ADTs allow users to work with data structures without having to know the implementation details, which can simplify programming and reduce errors.
- **Modularity**: ADTs can be combined with other ADTs to form more complex data structures, which can increase flexibility and modularity in programming.

DISADVANTAGES:

- **Overhead**: Implementing ADTs can add overhead in terms of memory and processing, which can affect performance.

- **Complexity**: ADTs can be complex to implement, especially for large and complex data structures.
- **Learning** Curve: Using ADTs requires knowledge of their implementation and usage, which can take time and effort to learn.

1.1.1 – DATE ABSTRACT DATATYPE

The Date ADT represents a single day in the proleptic Gregorian calendar, starting from November 24, 4713 BC.

Definition of Date ADT:

- ✓ Date(month, day, year): Creates a new Date instance for the given Gregorian date.
- ✓ day(): Returns the day of the month.
- ✓ month(): Returns the month number.
- ✓ year(): Returns the year.
- ✓ monthName(): Returns the month's name.
- ✓ dayOfWeek(): Returns the day of the week (0 = Monday, 6 = Sunday).
- ✓ numDays(otherDate): Returns the number of days between this date and otherDate.
- ✓ isLeapYear(): Checks if the year is a leap year.
- ✓ advanceBy(days): Advances or decrements the date by a specified number of days.
- ✓ comparable(otherDate): Compares this date to another date.
- ✓ toString(): Returns a string representation of the date in the format mm/dd/yyyy.

Operations as Python Operators:

- ✓ comparable() can use logical operators like <, <=, >, >=, ==, !=.
- ✓ toString() is used as str().

Example Usage

```
from datetime import datetime
class Date:
   def __init__(self, month, day, year):
      self.date = datetime(year, month, day)
```

```
    def day(self):
        return self.date.day
      def month(self):
        return self.date.month
       def year(self):
        return self.date.year
       def monthName(self):
        return self.date.strftime('%B')
       def dayOfWeek(self):
        return self.date.weekday()
       def numDays(self, otherDate):
        delta = self.date - otherDate.date
        return abs(delta.days)
       def isLeapYear(self):
        year = self.date.year
        return year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)
       def advanceBy(self, days):
        from datetime import timedelta
        self.date += timedelta(days=days)
       def __lt__(self, other):
        return self.date < other.date
       def __le__(self, other):
        return self.date <= other.date
       def __gt__(self, other):
        return self.date > other.date
       def __ge__(self, other):
        return self.date >= other.date
       def __eq__(self, other):
        return self.date == other.date
      def __ne__(self, other):
        return self.date != other.date
    def __str__(self):
        return self.date.strftime('%m/%d/%Y')
```

Example Usage: Processing Birth Dates

The program checkdates.py processes a collection of birth dates to determine if individuals are at least 21 years old.

```
from date import Date
def main():
    bornBefore = Date(6, 1, 1988)
      def promptAndExtractDate():
        print("Enter a birth date.")
        month = int(input("month (0 to quit): "))
        if month == 0:
            return None
        day = int(input("day: "))
        year = int(input("year: "))
        return Date(month, day, year)
     date = promptAndExtractDate()
    while date is not None:
        if date <= bornBefore:
            print("Is at least 21 years of age:", date)
        date = promptAndExtractDate()
main()
```

Implementing the Date ADT

The Date class stores the date as a Julian day number. The constructor converts a Gregorian date to a Julian day number. Conversion back to Gregorian is done via a helper method.

Partial Implementation of the Date Class:

```
class Date:
    def __init__(self, month, day, year):
        assert self._isValidGregorian(month, day, year), "Invalid Gregorian
date."
        tmp = -1 if month < 3 else 0
        self._julianDay = day - 32075 + (1461 * (year + 4800 + tmp) // 4) + \
                          (367 * (month - 2 - tmp * 12) // 12) - \
                          (3 * ((year + 4900 + tmp) // 100) // 4)
    def month(self):
        return self._toGregorian()[0]
        def day(self):
        return self._toGregorian()[1]
        def year(self):
        return self._toGregorian()[2]
        def dayOfWeek(self):
        month, day, year = self._toGregorian()
        if month < 3:
            month += 12
            year -= 1
        return (13 * month + 3) // 5 + day + year + year // 4 - year // 100 + year
// 400 % 7
        def __str__(self):
        month, day, year = self._toGregorian()
        return "%02d/%02d/%04d" % (month, day, year)
```

```
  def __eq__(self, otherDate):
    return self._julianDay == otherDate._julianDay
  def __lt__(self, otherDate):
    return self._julianDay < otherDate._julianDay
  def __le__(self, otherDate):
    return self._julianDay <= otherDate._julianDay
  def _toGregorian(self):
    A = self._julianDay + 68569
    B = 4 * A // 146097
    A -= (146097 * B + 3) // 4
    year = 4000 * (A + 1) // 1461001
    A -= (1461 * year // 4) + 31
    month = 80 * A // 2447
    day = A - (2447 * month // 80)
    month = month + 2 - 12 * (month // 11)
    year = 100 * (B - 49) + year + month // 11
    return month, day, year
```

1.1.2 - BAGS DATATYPE , ITERATORS DATATYPE

The Bag ADT is a container that stores a collection of items where duplicates are allowed. The items do not have a specific order, but they must be comparable. The Bag ADT includes the following operations:

Initialization

Bag(): Creates an empty bag.

Operations

- ✓ **__len__():** Returns the number of items in the bag, accessible via the len() function.
- ✓ **__contains__(item):** Checks if a given item is in the bag, accessible via the in operator.
- ✓ **add(item):** Adds an item to the bag.
- ✓ **remove(item):** Removes and returns an occurrence of an item from the bag. Raises an exception if the item is not found.
- ✓ **__iter__():** Returns an iterator for traversing the items in the bag.

Examples of Using Bag ADT

Guessing Game Example:

```
myBag = Bag()
```

```
myBag.add(19)
myBag.add(74)
myBag.add(23)
myBag.add(19)
myBag.add(12)
value = int(input("Guess a value contained in the bag: "))
if value in myBag:
   print("The bag contains the value", value)
else:
   print("The bag does not contain the value", value)
```

Storing Birth Dates Example:

```
from linearbag import Bag
from date import Date
def main():
   bornBefore = Date(6, 1, 1988)
   bag = Bag()
   date = promptAndExtractDate()
   while date is not None:
      bag.add(date)
      date = promptAndExtractDate()
      for date in bag:
      if date <= bornBefore:
         print("Is at least 21 years of age:", date)
```

Advantages of Using Bag ADT

- ✓ Abstraction: Focus on problem-solving without worrying about the underlying implementation.
- ✓ Error Reduction: Less chance of misuse compared to using a list directly.
- ✓ Coordination: Improved coordination between different modules and designers.
- ✓ Efficiency: Potential to swap out implementations for more efficient versions.

Selecting a Data Structure

When selecting a data structure for implementing the Bag ADT, consider:

- ✓ Storage Requirements: Must store all possible values within the domain.
- ✓ Functionality: Must provide the necessary operations without exposing implementation details.
- ✓ Efficiency: Some data structures offer more efficient implementations than others.

List-Based Implementation

The list is a suitable choice for implementing a simple Bag ADT:

- ✓ Storage: Can store any type of comparable object, including duplicates.

Operations:

- ✓ An empty bag is represented by an empty list.
- ✓ Size is determined by the length of the list.
- ✓ Containment is checked using the in operator.
- ✓ Adding an item appends it to the end of the list.
- ✓ Removing an item uses the pop method.

Example Implementation in Python

```
class Bag:
  # Constructs an empty bag.
  def __init__(self):
    self._theItems = list()
  # Returns the number of items in the bag.
  def __len__(self):
    return len(self._theItems)
  # Determines if an item is contained in the bag.
  def __contains__(self, item):
    return item in self._theItems
  # Adds a new item to the bag.
  def add(self, item):
    self._theItems.append(item)
  # Removes and returns an instance of the item from the bag.
  def remove(self, item):
    assert item in self._theItems, "The item must be in the bag."
    ndx = self._theItems.index(item)
    return self._theItems.pop(ndx)
  # Returns an iterator for traversing the list of items.
  def __iter__(self):
    return iter(self._theItems)
```

This implementation satisfies the Bag ADT requirements using the list's functionality. Each operation is implemented using methods provided by the list, ensuring simplicity and efficiency for typical use cases.

ITERATORS DATA TYPE

Iterators are essential tools for performing traversals over collections in Python. Traversals involve iterating over each element in a collection, enabling operations such as searching, filtering, or simply accessing each item for various purposes. Python's built-in container types (strings, tuples, lists, and dictionaries) inherently support traversal through the for loop construct. However, when dealing with user-defined abstract data types (ADTs), specific methods need to be added to facilitate such traversals.

An iterator is a powerful construct in Python that allows generic traversals through a container without exposing its underlying implementation. This abstraction is crucial as it maintains the integrity and encapsulation of the data structure. Implementing an iterator involves defining a class with two special methods: __iter__() and __next__(). These methods enable the use of Python’s for loop to traverse both built-in and user-defined containers seamlessly.

Student Records Application:

In many real-world applications, managing and processing data stored externally is a common task. Consider a scenario where we need to process student records stored on disk to produce a sorted report. Each student record contains details like ID number, name, classification (freshman, sophomore, junior, or senior), and GPA.

To handle this, we can define a Student File Reader ADT that abstracts the process of reading records from an external file. This ADT includes methods for opening and closing the file, fetching individual records, and fetching all records at once. The fetched data is stored in a structured format using a StudentRecord class.

By combining the iterator pattern with the Student File Reader ADT, we can create a robust solution for reading, processing, and reporting student records. This approach leverages the power of abstraction, allowing us to design a flexible and reusable system without being tied to specific data formats or storage methods.

Traversals and Iterators:

- ✓ Traversals iterate over collections to access individual elements.
- ✓ Python's built-in types (strings, tuples, lists, dictionaries) can be traversed using for loops.

- ✓ User-defined ADTs can implement traversal methods for specific operations, but a generic traversal mechanism is more flexible.

Iterator Design:

Python uses an iterator construct for traversals, preserving abstraction without exposing the underlying data structure.

An iterator class should define __iter__() and __next__() methods.

Example - BagIterator Class:

Initialization:

```
class _BagIterator:
    def __init__(self, theList):
        self._bagItems = theList
        self._curItem = 0
```

Iteration Methods:

```
def __iter__(self):
    return self
def __next__(self):
    if self._curItem < len(self._bagItems):
        item = self._bagItems[self._curItem]
        self._curItem += 1
        return item
    else:
        raise StopIteration
```

Integrating with Bag Class:

```
def __iter__(self):
    return _BagIterator(self._theItems)
```

Using Iterators:

Example with for loop:

```
for item in bag:
    print(item)
```

Under the hood:

```
iterator = myBag.__iter__()
while True:
    try:
        item = iterator.__next__()
        print(item)
    except StopIteration:
        break
```

Application: Student Records

Student File Reader ADT:

Manages data extraction from external sources.

Functions:

- ✓ open(): Opens the data source.
- ✓ close(): Closes the data source.
- ✓ fetchRecord(): Extracts and returns the next record.
- ✓ fetchAll(): Extracts and returns all records in a list.

StudentRecord Class:

Stores individual student data fields:

```
class StudentRecord:
    def __init__(self):
        self.idNum = 0
        self.firstName = None
        self.lastName = None
        self.classCode = 0
        self.gpa = 0.0
```

Generating Student Report:

```
from studentfile import StudentFileReader
FILE_NAME = "students.txt"
def main():
    reader = StudentFileReader(FILE_NAME)
    reader.open()
    studentList = reader.fetchAll()
   reader.close()
    studentList.sort(key=lambda rec: rec.idNum)
    printReport(studentList)
def printReport(theList):
    classNames = (None, "Freshman", "Sophomore", "Junior", "Senior")
    print("LIST OF STUDENTS".center(50))
    print("%-5s %-25s %-10s %-4s" % ('ID', 'NAME', 'CLASS', 'GPA'))
    print("%5s %25s %10s %4s" % ('-' * 5, '-' * 25, '-' * 10, '-' * 4))
    for record in theList:
        print("%5d %-25s %-10s %4.2f" % \
              (record.idNum, \
               record.lastName + ', ' + record.firstName,
               classNames[record.classCode], record.gpa))
    print("-" * 50)
    print("Number of students:", len(theList))
main()
```

Student File Reader Implementation:

Reading Data from Text File:

```
class StudentFileReader:
    def __init__(self, inputSrc):
        self._inputSrc = inputSrc
        self._inputFile = None
    def open(self):
        self._inputFile = open(self._inputSrc, "r")
    def close(self):
        self._inputFile.close()
        self._inputFile = None
    def fetchAll(self):
        theRecords = list()
        student = self.fetchRecord()
        while student:
            theRecords.append(student)
            student = self.fetchRecord()
        return theRecords
    def fetchRecord(self):
        line = self._inputFile.readline()
        if line == "":
            return None
        student = StudentRecord()
        student.idNum = int(line)
        student.firstName = self._inputFile.readline().rstrip()
        student.lastName = self._inputFile.readline().rstrip()
        student.classCode = int(self._inputFile.readline())
        student.gpa = float(self._inputFile.readline())
        return student
```

This implementation allows traversing custom data structures and generating a report by leveraging Python's iterator protocol and the abstraction of data extraction.

Let Us Sum Up

In this segment, we delve into the foundational concepts of Abstract Data Types (ADTs). We start with the Date ADT, exploring methods for representing and manipulating dates. Moving forward, we delve into Bags, an ADT allowing for flexible element storage with possible repetitions. Additionally, we introduce iterators for efficient data traversal. Through practical examples and exercises, students gain proficiency in using these essential ADTs, laying a strong foundation for further exploration in data structures and algorithms.

Check Your Progress

1. What is an Abstract Data Type (ADT)?
    A) A specific implementation of a data structure
    B) A mathematical model for representing data and its associated operations
    C) An algorithm for data manipulation
    D) A programming language feature for dynamic typing
2. Which of the following is an example of a Date ADT operation?
    A) Adding two dates together
    B) Sorting a list of dates
    C) Calculating the average of a set of dates
    D) Printing the current date and time
3. Bags, as an ADT, allow for:
    A) Storing elements with possible repetitions
    B) Storing only unique elements
    C) Storing elements in sorted order
    D) Storing elements in a fixed-size array
4. What is the purpose of an iterator in Python?
    A) To perform mathematical calculations
    B) To efficiently traverse and manipulate data structures
    C) To convert data types from one form to another
    D) To execute database queries efficiently
5. In the context of Bags, what does it mean when an element is stored with possible repetitions?
    A) Each element is stored exactly once
    B) Elements are stored in sorted order
    C) Elements may occur more than once in the collection
    D) Elements are stored in a fixed-size array
6. Which of the following operations is typically associated with iterators?
    A) Adding elements to a data structure
    B) Removing elements from a data structure
    C) Traversing elements in a data structure

   D) Sorting elements in a data structure

7. What is the primary advantage of using iterators in Python?
   A) They simplify syntax in loop structures
   B) They provide additional memory allocation for data structures
   C) They enable parallel processing of data
   D) They improve performance by efficiently accessing elements
8. How does the Python list data structure differ from a Bag ADT?
   A) Lists allow only unique elements
   B) Bags store elements in a fixed-size array
   C) Bags allow for possible repetitions of elements
   D) Lists are only used for numerical data
9. Which of the following is a common use case for a Bag ADT?
   A) Representing a collection of unique user IDs
   B) Storing sorted data for quick retrieval
   C) Tracking inventory items in a warehouse
   D) Managing database connections
10. What role do ADTs play in software development?
   A) They provide specific implementations of data structures
   B) They define the structure and behavior of data without specifying its implementation
   C) They are used exclusively in low-level programming languages
   D) They are primarily used for graphical user interface (GUI) development

2 SECTION 1.2: ARRAY

Arrays and Their Importance

At the hardware level, arrays are contiguous memory blocks accessed using an index. They provide random access to elements and are efficient for operations where the number of elements is known beforehand. Unlike lists, arrays have a fixed size and limited operations: creation, reading, writing, and traversal.

Why Study Arrays?

Arrays and Python's list structures share similarities, but they serve different purposes. Key differences include:

- ✓ Limited Operations: Arrays support basic operations like creation, reading, and writing values. Lists offer a broader set of operations, including insertion, deletion, and searching.
- ✓ Fixed Size: Arrays have a fixed size defined at creation, while lists can dynamically grow and shrink during execution.

Arrays are ideal when the number of elements is known beforehand, providing memory efficiency. Lists, however, are better suited for situations where the size of the sequence needs to change.

1.2.1 – Array Abstract Data Type

Array Abstract Data Type (ADT):

An Array ADT represents a one-dimensional array with fixed size and basic operations:

- ✓ Array(size): Creates an array of specified size with elements initialized to None.
- ✓ length(): Returns the number of elements in the array.
- ✓ getitem(index): Retrieves the value at the specified index.
- ✓ setitem(index, value): Sets the value at the specified index.
- ✓ clear(value): Sets all elements to the specified value.
- ✓ iterator(): Returns an iterator for the array.

Implementation of the Array ADT

Using the ctypes Module:

Python’s ctypes module allows creating low-level arrays similar to those in C. These arrays can store references to Python objects and are accessed using integer indices. The ctypes module provides the necessary functionality for array creation and manipulation while keeping the implementation details hidden within a class.

Class Definition:

The Array class, implemented using ctypes, includes methods for creating and managing a fixed-size array:

```
import ctypes
class Array:
    def __init__(self, size):
        assert size > 0, "Array size must be > 0"
        self._size = size
        PyArrayType = ctypes.py_object * size
        self._elements = PyArrayType()
        self.clear(None)
    def __len__(self):
        return self._size
    def __getitem__(self, index):
        assert 0 <= index < self._size, "Array subscript out of range"
        return self._elements[index]
    def __setitem__(self, index, value):
        assert 0 <= index < self._size, "Array subscript out of range"
        self._elements[index] = value
    def clear(self, value):
        for i in range(self._size):
            self._elements[i] = value
    def __iter__(self):
        return _ArrayIterator(self._elements)
class _ArrayIterator:
    def __init__(self, theArray):
        self._arrayRef = theArray
        self._curNdx = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self._curNdx < len(self._arrayRef):
            entry = self._arrayRef[self._curNdx]
            self._curNdx += 1
            return entry
        else:
            raise StopIteration
```

This implementation defines an Array class with essential methods for managing array elements, ensuring a fixed-size, efficient, and straightforward data structure for scenarios where dynamic resizing is unnecessary.

Use Cases for the Array Class

Creating and Using Arrays:

```
# Create an array of size 100
valueList = Array(100)
# Fill the array with random values
import random
for i in range(len(valueList)):
    valueList[i] = random.random()
# Print the array values
for value in valueList:
    print(value)
```

Counting Character Occurrences in a Text File:

```
# Create an array for counting ASCII character occurrences
theCounters = Array(127)
theCounters.clear(0)
# Open and read a text file
with open('atextfile.txt', 'r') as theFile:
    for line in theFile:
        for letter in line:
            code = ord(letter)
            theCounters[code] += 1
# Print the count of each letter
for i in range(26):
    print(f"{chr(65+i)} - {theCounters[65+i]} {chr(97+i)} -
{theCounters[97+i]}")
```

Arrays provide a fixed-size, memory-efficient data structure ideal for scenarios where the sequence size is known upfront. The Array ADT in Python, implemented using the ctypes module, offers basic operations for creating, accessing, and managing arrays, bridging the gap between Python's flexible list structure and the more rigid, efficient array structure found in lower-level languages.

1.2.2 - THE PYTHON LIST

Creating a Python List

- ✓ When a list is created in Python, it internally uses an array structure to store the elements.
- ✓ Here's how it works:
- ✓ The list() constructor is called with the specified values, creating a list object.
- ✓ Internally, an array structure is initialized, initially larger than needed to accommodate future expansions.
- ✓ The array stores the actual contents of the list, with unused slots for potential future elements.

List Initialization in Python:

1. Using Square Brackets [ ]:

The most common way to initialize a list is by enclosing elements within square brackets.

my_list = [1, 2, 3, 4, 5]

2. Using the list() Constructor:

The list() constructor can be used to create a list from an iterable object like a tuple, string, or another list.

my_list = list((1, 2, 3, 4, 5)) # Convert tuple to list

3. Using List Comprehensions:

List comprehensions offer a concise way to create lists based on existing lists or other iterables.

squares = [x ** 2 for x in range(1, 6)] # Generate a list of squares

4. Using range() Function:

The range() function can be used to generate a sequence of numbers, which can then be converted into a list.

numbers = list(range(1, 6)) # Create a list of numbers from 1 to 5

5. Using Repeat Operator (*):

The repeat operator * can initialize a list with repeated elements.

repeated_list = [0] * 5 # Create a list with five zeros

6. Using Nested Lists:

Lists can contain other lists, allowing for the creation of multidimensional arrays.

nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] # Create a 2D list

7. Using List Methods:

List methods like append(), extend(), and insert() can be used to initialize or modify lists dynamically.

my_list = [] # Initialize an empty list

my_list.append(10) # Add an element to the list

8. Using zip() Function:

The zip() function can be used to combine multiple iterables into tuples, which can then be converted into a list.

list_of_tuples = list(zip([1, 2, 3], ['a', 'b', 'c'])) # Zip two lists into a list of tuples

9. Using Dict Comprehensions:

Dict comprehensions can be used to create lists from dictionaries based on specific conditions.

even_values = [value for key, value in my_dict.items() if key % 2 == 0] # Create a list of even values from a dictionary

10. Using Generators:

Generators can be used to lazily initialize large lists without consuming memory upfront.

large_list = [x for x in range(1000000)] # This consumes memory upfront

large_list_lazy = (x for x in range(1000000)) # This is a generator expression

These are some of the common ways to initialize lists in Python, each offering flexibility and suitability for different scenarios. Choose the method that best fits your requirements in terms of readability, efficiency, and ease of use.

In Python, a list is a built-in data structure used to store a collection of items. Lists are versatile and can hold elements of different types, including other lists (i.e., nested lists). They are mutable, meaning their contents can be changed after creation. Here's a basic overview:

Example :

Creating Lists:

Create a list by enclosing comma-separated values within square brackets [].

my_list = [1, 2, 3, 4, 5]

Accessing Elements:

Access individual elements of a list using indexing. Indexing starts at 0.

print(my_list[0]) # Output: 1

print(my_list[2]) # Output: 3

Modifying Lists:

Lists are mutable, so you can change their elements.

my_list[0] = 10

print(my_list) # Output: [10, 2, 3, 4, 5]

List Operations:

It can perform various operations on lists, like concatenation and repetition.

list1 = [1, 2, 3]

list2 = [4, 5, 6]

concatenated_list = list1 + list2

print(concatenated_list) # Output: [1, 2, 3, 4, 5, 6]

repeated_list = list1 * 3

```
print(repeated_list)  # Output: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

List Methods:

Python provides several built-in methods for manipulating lists, such as append(), extend(), insert(), remove(), pop(), sort(), and reverse().

```
my_list.append(6)  # Appends 6 to the end of the list
my_list.extend([7, 8, 9])  # Extends the list with elements from another list
my_list.insert(2, 20)  # Inserts 20 at index 2
my_list.remove(3)  # Removes the first occurrence of 3 from the list
my_list.pop()  # Removes and returns the last element of the list
my_list.sort()  # Sorts the list
my_list.reverse()  # Reverses the elements of the list
```

Lists are incredibly versatile and are used extensively in Python programming for a wide range of tasks, from storing data to implementing complex algorithms.

Appending Items

- ✓ Appending an item to the end of a list is a common operation. Here's what happens:
- ✓ If there's room in the array, the item is stored in the next available slot, and the length field is incremented.
- ✓ If the array becomes full, it needs to be expanded:
- ✓ A new array, typically double the size of the original, is created.
- ✓ Values from the original array are copied to the new larger array.
- ✓ The new array replaces the original in the list, and the original is destroyed.

Extending a List

- ✓ Extending a list with another list involves adding all elements of the second list to the first. Here's how it's done:
- ✓ If the destination list has enough capacity, elements are copied directly.
- ✓ If there's not enough capacity, the array is expanded, similar to the append operation.
- ✓ Elements from both lists are copied to the new larger array, and the destination list is updated.

Inserting Items

- ✓ Inserting an item at any position within the list involves shifting elements to make room for the new item. Here's the process:
- ✓ Items following the insertion point are shifted down, creating space for the new item.
- ✓ If the array becomes full during insertion, it's expanded following the same steps as the append operation.

Removing Items

- ✓ Removing an item from a list typically involves shifting elements to close the gap. Here's how it works:
- ✓ When an item is removed, the elements following it are shifted down, filling the gap.
- ✓ If removal results in a significant number of empty slots, the array may be resized to conserve memory.

List Slicing

Slicing creates a new list containing a subset of elements from the original list

Basic List Slicing:

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# Extract elements from index 2 to 5 (exclusive)
subset = my_list[2:5]
print(subset)  # Output: [3, 4, 5]
```

Using Negative Indices:

Negative indices count from the end of the list.

```
# Extract last three elements
subset = my_list[-3:]
print(subset)  # Output: [8, 9, 10]
```

Specifying Step Size:

Can specify a step size to skip elements during slicing.

```
# Extract every second element
subset = my_list[::2]
print(subset)  # Output: [1, 3, 5, 7, 9]
```

List Iteration:

Iterating over a list allows you to access each element one by one.

Using a for Loop:

```
for item in my_list:
    print(item)
```

Using enumerate():

Can use enumerate() to get both the index and the element.

```
for index, item in enumerate(my_list):
    print(f"Index: {index}, Value: {item}")
```

List Comprehension:

List comprehensions offer a concise way to iterate over lists and perform operations on elements.

```
squared_values = [x ** 2 for x in my_list]
print(squared_values)
```

Using zip():

zip() allows to iterate over multiple lists simultaneously.

```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']
for item1, item2 in zip(list1, list2):
    print(item1, item2)
```

Using While Loop:

Can also iterate over a list using a while loop.

```
index = 0
while index < len(my_list):
    print(my_list[index])
    index += 1
```

List Slicing and Iteration Combined:

List slicing and iteration can be combined to perform complex operations efficiently.

```
# Print every second element starting from index 1
for item in my_list[1::2]:
    print(item)
```

These examples demonstrate various techniques for list slicing and iteration in Python, offering flexibility and efficiency for different use cases.

Here's what happens:

- ✓ A new list is created with enough capacity to store the subset.
- ✓ Elements within the specified range are copied to the new list.
- ✓ The original list remains unchanged.

Example

```
# Creating a Python List
pyList = [4, 12, 2, 34, 17]
# Appending Items
pyList.append(50)  # Appending 50 to the end of the list
print("After appending 50:", pyList)
# Extending a List
pyListA = [34, 12]
pyListB = [4, 6, 31, 9]
pyListA.extend(pyListB)  # Extending pyListA with elements from
pyListB
print("After extending pyListA:", pyListA)
# Inserting Items
pyList.insert(3, 79)  # Inserting 79 at index position 3
print("After inserting 79 at index 3:", pyList)
# Removing Items
pyList.pop(0)  # Removing the first item
pyList.pop()  # Removing the last item
print("After removing first and last items:", pyList)
# List Slicing
aSlice = pyList[2:4]  # Creating a slice from index 2 to 4 (exclusive)
print("Slice from index 2 to 4:", aSlice)
```

This code demonstrates various list operations:

- ✓ Creating a Python List: Initializes a list with some initial values.
- ✓ Appending Items: Appends a new item (50) to the end of the list.
- ✓ Extending a List: Extends one list with elements from another list.
- ✓ Inserting Items: Inserts an item (79) at a specific index position.
- ✓ Removing Items: Removes the first and last items from the list.
- ✓ List Slicing: Creates a slice of the list containing elements from index 2 to 4 (exclusive).

1.2.3 TWO DIMENSIONAL ARRAYS IN PYTHON

A two-dimensional (2D) array organizes data into rows and columns, similar to a table or grid. Each element in the array is accessed by specifying its row and column indices, e.g., array[row, col].

Array2D Abstract Data Type (ADT)

Operations:

- ✓ Array2D(nrows, ncols): Creates a 2D array with nrows rows and ncols columns. Elements are initialized to None.
- ✓ numRows(): Returns the number of rows.
- ✓ numCols(): Returns the number of columns.
- ✓ clear(value): Sets all elements to the given value.
- ✓ getitem(i, j): Returns the element at row i and column j.
- ✓ setitem(i, j, value): Sets the element at row i and column j to the given value.

Example Use Case

Consider a text file storing exam grades for students. The first line specifies the number of students, the second line specifies the number of exams, and the subsequent lines contain the grades for each student.

Text File (grades.txt):

```
3
2
90 85
88 76
92 95
```

Python Code to Read and Process Grades:

```
from array import Array2D
filename = "grades.txt"
gradeFile = open(filename, "r")
# Read number of students and exams
numStudents = int(gradeFile.readline())
numExams = int(gradeFile.readline())
# Create the 2D array
examGrades = Array2D(numStudents, numExams)
# Read and store the grades
```

```
for i in range(numStudents):
    grades = gradeFile.readline().split()
    for j in range(numExams):
        examGrades[i, j] = int(grades[j])
gradeFile.close()
# Calculate and display average grade for each student
for i in range(numStudents):
    total = sum(examGrades[i, j] for j in range(numExams))
    average = total / numExams
    print(f"Student {i+1}: Average Grade: {average:.2f}")
```

Implementation of Array2D

Using an Array of Arrays:

A common approach to implement a 2D array is to use an array of arrays, where each row is a 1D array.

```
class Array2D:
    def __init__(self, numRows, numCols):
        self._theRows = [Array(numCols) for _ in range(numRows)]
    def numRows(self):
        return len(self._theRows)
    def numCols(self):
        return len(self._theRows[0])
    def clear(self, value):
        for row in self._theRows:
            row.clear(value)
    def __getitem__(self, ndxTuple):
        assert len(ndxTuple) == 2, "Invalid number of array subscripts."
        row, col = ndxTuple
        assert 0 <= row < self.numRows() and 0 <= col < self.numCols(),
"Array subscript out of range."
        return self._theRows[row][col]
    def __setitem__(self, ndxTuple, value):
        assert len(ndxTuple) == 2, "Invalid number of array subscripts."
        row, col = ndxTuple
        assert 0 <= row < self.numRows() and 0 <= col < self.numCols(),
"Array subscript out of range."
        self._theRows[row][col] = value
```

Array Class for 1D Arrays:

```
class Array:
    def __init__(self, size):
        self._elements = [None] * size
    def __len__(self):
        return len(self._elements)
```

```
def __getitem__(self, index):
   return self._elements[index]
def __setitem__(self, index, value):
   self._elements[index] = value
def clear(self, value):
   for i in range(len(self._elements)):
      self._elements[i] = value
```

Key Concepts and Operations

- ✓ Initialization: The Array2D constructor creates a list of Array objects, each representing a row.
- ✓ Accessing Elements: The __getitem__ and __setitem__ methods use tuple subscripts (row, col) to access elements.
- ✓ Clearing the Array: The clear method sets all elements in the 2D array to a specified value by iterating over each row.
- ✓ Index Validation: Always check that indices are within valid ranges to prevent errors.
- ✓ Efficiency Considerations: Operations on 2D arrays can be complex; ensure efficient handling of row and column accesses.
- ✓ Flexibility: The Array2D implementation provides a flexible way to handle multi-dimensional data structures.

Two-dimensional arrays are a fundamental data structure in programming, useful for organizing data in a tabular format. The Array2D ADT provides a robust framework for creating and manipulating 2D arrays in Python, offering essential operations such as element access, modification, and array-wide clearing. Implementing 2D arrays using an array of arrays ensures a structured and efficient way to handle multi-dimensional data.

In Python, a two-dimensional array is typically represented using a list of lists. It's essentially a list where each element is itself a list, forming rows and columns. Here's how you can work with two-dimensional arrays:

CREATING A TWO-DIMENSIONAL ARRAY:

You can create a 2D array by nesting lists within lists. Each inner list represents a row, and the outer list contains all the rows.

```
# Creating a 2D array with 3 rows and 4 columns filled with zeros
two_d_array = [[0 for _ in range(4)] for _ in range(3)]
```

ACCESSING ELEMENTS:

You can access elements by specifying both the row index and the column index.

```
print(two_d_array[0][0])  # Accessing the element at row 0, column 0
```

MODIFYING ELEMENTS:

You can modify elements just like accessing them.

```
two_d_array[1][2] = 5  # Setting the element at row 1, column 2 to 5
```

ITERATING THROUGH A 2D ARRAY:

You can iterate through each element of a 2D array using nested loops.

```
For row in two_d_array:
   for element in row:
      print(element, end=' ')
   print()  # Move to the next line after printing each row
```

WORKING WITH LIBRARIES:

If you're dealing with numerical data and computations, you might want to consider using libraries like NumPy, which provides efficient implementations for multi-dimensional arrays and a wide range of mathematical functions.

```
import numpy as np
# Creating a 2D NumPy array
two_d_array_np = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Accessing elements
print(two_d_array_np[0, 0])  # Accessing the element at row 0, column 0
```

Using NumPy can offer significant performance improvements and additional functionality compared to using nested lists for numerical computations involving multi-dimensional arrays.

EXAMPLE:

Here's an example code demonstrating how to work with a two-dimensional array in Python using nested lists:

```
# Define a 2D array (3x3)
two_d_array = [[1, 2, 3],
               [4, 5, 6],
               [7, 8, 9]]
# Accessing elements
print("Element at row 0, column 0:", two_d_array[0][0])  # Output: 1
print("Element at row 1, column 2:", two_d_array[1][2])  # Output: 6
# Modifying elements
two_d_array[1][1] = 10
```

```
print("Modified 2D array:")
for row in two_d_array:
    print(row)
# Iterating through the array
print("Iterating through the 2D array:")
for row in two_d_array:
    for element in row:
        print(element, end=' ')
    print()  # Move to the next line after printing each row
```

This code creates a 3x3 two-dimensional array, accesses and modifies elements, and then iterates through the array to print its contents.

1.2.4 MATRIX ABSTRACT DATATYPE

A matrix is a mathematical construct that consists of a rectangular grid of numerical values, organized into rows and columns. In an m × n matrix, there are m rows and n columns. Matrices are fundamental in various fields such as linear algebra and computer graphics. They are used extensively to represent and solve systems of linear equations, among other applications.

Definition of the Matrix ADT

The Matrix ADT is a formal structure that allows for the creation and manipulation of matrices. It includes a set of operations that can be performed on matrices. Here's a detailed look at the operations defined in the Matrix ADT:

- ✓ Matrix(nrows, ncols): Initializes a new matrix with the given number of rows (nrows) and columns (ncols), with all elements set to 0.
- ✓ numRows(): Returns the number of rows in the matrix.
- ✓ numCols(): Returns the number of columns in the matrix.
- ✓ getitem(row, col): Retrieves the value stored at the specified row and column indices. Both indices must be within valid ranges.
- ✓ setitem(row, col, scalar): Sets the value at the specified row and column indices to the given scalar. The indices must be within valid ranges.
- ✓ scaleBy(scalar): Multiplies each element of the matrix by the given scalar value. This operation modifies the matrix in place.
- ✓ transpose(): Returns a new matrix that is the transpose of the current matrix.

- ✓ add(rhsMatrix): Creates and returns a new matrix that is the result of adding the current matrix to the given rhsMatrix. Both matrices must be of the same size.
- ✓ subtract(rhsMatrix): Similar to the add operation but performs subtraction instead.
- ✓ multiply(rhsMatrix): Creates and returns a new matrix that is the result of multiplying the current matrix with the rhsMatrix. The number of columns in the current matrix must equal the number of rows in rhsMatrix.

Addition and Subtraction

Addition: Two matrices of the same dimensions can be added together by summing their corresponding elements.

- Example:

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix} + \begin{bmatrix} 6 & 7 \\ 8 & 9 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \\ 5 & 5 \end{bmatrix}$$

Subtraction: Subtraction is performed similarly, with corresponding elements being subtracted.

- Example:

$$\begin{bmatrix} 6 & 8 \\ 10 & 12 \\ 5 & 5 \end{bmatrix} - \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 7 \\ 9 & 11 \\ 4 & 4 \end{bmatrix}$$

Scaling

Scaling: Each element of the matrix is multiplied by a scalar value.

- Example:

$$3 \cdot \begin{bmatrix} 6 & 7 \\ 8 & 9 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 18 & 21 \\ 24 & 27 \\ 3 & 0 \end{bmatrix}$$

Multiplication

Matrix Multiplication: Matrix multiplication is only defined when the number of columns in the first matrix equals the number of rows in the second matrix. The resulting matrix will have the same number of rows as the first matrix and the same number of columns as the second matrix.

Transpose: Transpose of a matrix is obtained by swapping rows and columns.

- Example:

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix} \cdot \begin{bmatrix} 6 & 7 & 8 \\ 9 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 9 & 1 & 0 \\ 39 & 17 & 16 \\ 69 & 33 & 32 \end{bmatrix}$$

Transpose: Transpose of a matrix is obtained by swapping rows and columns.

- Example:

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}^T = \begin{bmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix}$$

Implementing the Matrix

The Matrix ADT can be implemented using a two-dimensional array. Here is a basic implementation in Python:

```
from array import Array2D
class Matrix:
    def __init__(self, numRows, numCols):
        self._theGrid = Array2D(numRows, numCols)
        self._theGrid.clear(0)
    def numRows(self):
        return self._theGrid.numRows()
    def numCols(self):
        return self._theGrid.numCols()
    def __getitem__(self, ndxTuple):
        return self._theGrid[ndxTuple[0], ndxTuple[1]]
    def __setitem__(self, ndxTuple, scalar):
        self._theGrid[ndxTuple[0], ndxTuple[1]] = scalar
    def scaleBy(self, scalar):
        for r in range(self.numRows()):
            for c in range(self.numCols()):
                self[r, c] *= scalar
    def transpose(self):
        newMatrix = Matrix(self.numCols(), self.numRows())
        for r in range(self.numRows()):
            for c in range(self.numCols()):
                newMatrix[c, r] = self[r, c]
        return newMatrix
    def __add__(self, rhsMatrix):
        assert rhsMatrix.numRows() == self.numRows() and
rhsMatrix.numCols() == self.numCols(), "Matrix sizes not compatible
for the add operation."
        newMatrix = Matrix(self.numRows(), self.numCols())
        for r in range(self.numRows()):
            for c in range(self.numCols()):
                newMatrix[r, c] = self[r, c] + rhsMatrix[r, c]
        return newMatrix
    def __sub__(self, rhsMatrix):
        assert rhsMatrix.numRows() == self.numRows() and
rhsMatrix.numCols() == self.numCols(), "Matrix sizes not compatible
for the subtract operation."
        newMatrix = Matrix(self.numRows(), self.numCols())
        for r in range(self.numRows()):
            for c in range(self.numCols()):
                newMatrix[r, c] = self[r, c] - rhsMatrix[r, c]
        return newMatrix
    def __mul__(self, rhsMatrix):
        assert self.numCols() == rhsMatrix.numRows(), "Matrix sizes not
compatible for the multiply operation."
        newMatrix = Matrix(self.numRows(), rhsMatrix.numCols())
        for r in range(self.numRows()):
            for c in range(rhsMatrix.numCols()):
                sum = 0
                for k in range(self.numCols()):
                    sum += self[r, k] * rhsMatrix[k, c]
                newMatrix[r, c] = sum
        return newMatrix
```

This implementation provides the basic operations defined in the Matrix ADT, allowing for matrix creation, element access, scaling, addition, subtraction, multiplication, and transposition.

Let Us Sum Up

In this segment, we explore the foundational concept of arrays and their implementations in Python. Beginning with an understanding of array structure, we delve into Python's versatile list data type, which serves as a dynamic array capable of storing various data types. We then extend our discussion to two-dimensional arrays, particularly focusing on matrices as a prominent example, and explore their applications in mathematical operations and data manipulation tasks. Finally, we introduce the concept of a Matrix Abstract Data Type, emphasizing its role in representing and performing operations on matrices efficiently. Through practical examples and exercises, students gain proficiency in utilizing arrays and matrices to solve diverse problems in programming and computational mathematics**.**

Check Your Progress

1. What is the primary purpose of an array?
   A) Storing data of different types
   B) Providing a dynamic data structure
   C) Storing a collection of elements of the same type
   D) Sorting data efficiently
2. In Python, which data structure is commonly used to implement arrays?
   A) List
   B) Tuple
   C) Set
   D) Dictionary
3. What is the key characteristic of a two-dimensional array?
   A) It stores elements in a single row
   B) It stores elements in multiple rows and columns
   C) It allows storing elements of different types
   D) It dynamically resizes based on the number of elements
4. Which operation is commonly performed on matrices?
   A) Searching
   B) Sorting
   C) Transposition
   D) Concatenation

5. What role does the Matrix Abstract Data Type play in programming?
    A) It defines the structure of a matrix but not its operations
    B) It provides specific implementations of matrix operations
    C) It is used only for storing numerical data
    D) It is primarily used in low-level programming languages
6. How is a two-dimensional array accessed in Python?
    A) Using only one index
    B) Using two indices (row and column)
    C) Using a loop structure
    D) Using a built-in function
7. What is a key advantage of using arrays for data storage?
    A) Flexibility in storing elements of different types
    B) Dynamic resizing based on the number of elements
    C) Efficient access to elements using indices
    D) Ability to perform complex mathematical operations
8. Which of the following operations is NOT typically performed on arrays?
    A) Insertion
    B) Deletion
    C) Searching
    D) Traversal
9. What is the time complexity of accessing an element in an array by its index?
    A) O(1)
    B) O(log n)
    C) O(n)
    D) O(n^2)
10. What is the advantage of using a Python list over a traditional array in other programming languages?
    A) Lists can store elements of different types
    B) Lists can dynamically resize themselves
    C) Lists are more efficient in terms of memory usage
    D) Lists provide better support for mathematical operations

3 SECTION 1.3: SETS, MAPS

1.3.1 – Sets

A set in computer science is a collection of unique values, mirroring the mathematical concept. It is used to store unique elements without concern for order and supports operations common in set theory.

Core Operations of the Set ADT:

- ✓ Set(): Initializes an empty set.
- ✓ length(): Returns the number of elements (cardinality) in the set, accessed via len().
- ✓ contains(element): Checks if an element is in the set, accessed via the in operator.
- ✓ add(element): Adds an element to the set if it is not already present.
- ✓ remove(element): Removes an element from the set if it exists, raises an exception otherwise.
- ✓ equals(setB): Checks if two sets are equal, i.e., they have the same elements. Accessed via == or !=.
- ✓ isSubsetOf(setB): Checks if the set is a subset of another set.
- ✓ union(setB): Returns a new set that is the union of the current set and setB.
- ✓ intersect(setB): Returns a new set that is the intersection of the current set and setB.
- ✓ difference(setB): Returns a new set that is the difference of the current set and setB.
- ✓ iterator(): Returns an iterator for traversing the set.

Example Usage

Consider two sets of courses for two students, Smith and Roberts. You can:

- ✓ Add courses to their respective sets.
- ✓ Check if they are taking the same courses.
- ✓ Find common courses using the intersection operation.
- ✓ Identify courses that one student is taking and the other is not using the difference operation.

- ✓ Implementation Details

Choosing a Data Structure:

The list structure is chosen to implement the Set ADT because it dynamically grows and allows for easy element addition and removal. Unlike dictionaries, lists do not inherently enforce uniqueness, so duplicate checks are necessary.

List-Based Implementation:

- ✓ Class Definition: The Set class uses a list to store elements.
- ✓ Adding Elements: The add() method ensures uniqueness by checking if an element is already present before adding it.
- ✓ Equality Check: The __eq__() method compares two sets by first checking their sizes and then verifying if one set is a subset of the other.
- ✓ Subset Check: The isSubsetOf() method iterates through the elements of the set and checks for their presence in another set.
- ✓ Union Operation: The union() method creates a new set, copies elements from both sets, and ensures no duplicates.

Key Methods in Implementation

Adding Elements: Ensures no duplicates are added.

```
def add(self, element):
    if element not in self:
        self._theElements.append(element)
```

Equality Check:

Uses subset check to determine equality.

```
def __eq__(self, setB):
    if len(self) != len(setB):
        return False
    else:
        return self.isSubsetOf(setB)
```

Subset Check:

Iterates over elements to confirm if all are present in another set.

```
def isSubsetOf(self, setB):
    for element in self:
        if element not in setB:
            return False
    return True
```

Union Operation:

Combines elements from both sets, ensuring uniqueness.

```
def union(self, setB):
    newSet = Set()
    newSet._theElements.extend(self._theElements)
    for element in setB:
        if element not in self:
            newSet._theElements.append(element)
    return newSet
```

The Set ADT, implemented using a list, provides essential functionality for managing collections of unique values and performing standard set operations. The choice of a list allows dynamic resizing and easy manipulation, though care must be taken to handle uniqueness explicitly. This abstraction simplifies working with sets by focusing on the provided operations rather than the underlying implementation details.

Here's an overview of working with sets in Python:

CREATING SETS:

Can create a set using curly braces {} or the set() constructor.

```
my_set = {1, 2, 3, 4, 5}
another_set = set([4, 5, 6, 7, 8])
```

ADDING AND REMOVING ELEMENTS:

Can add elements to a set using the add() method, and remove elements using the remove() or discard() methoD)

```
my_set.add(6)
my_set.remove(3)
```

SET OPERATIONS:

Python sets support various mathematical operations, including union, intersection, difference, and symmetric difference.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1.union(set2)  # Union of two sets
intersection_set = set1.intersection(set2)  # Intersection of two sets
```

```
difference_set = set1.difference(set2)  # Elements in set1 but not in set2
symmetric_difference_set = set1.symmetric_difference(set2)   # Elements in either set1 or set2, but not both
```

SET MEMBERSHIP:

Can check if an element is present in a set using the in keyworD)

```
if 3 in my_set:
    print("3 is present in the set.")
```

ITERATING THROUGH A SET:

Can iterate through the elements of a set using a for loop.

```
for item in my_set:
   print(item)
```

Python sets are incredibly useful for various tasks, such as removing duplicates from a list, performing set operations, and checking for membership. They provide efficient implementations of these operations, making them valuable in many programming scenarios.

EXAMPLE:

An example code demonstrating the use of sets in Python:

```
# Define two sets
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}
# Set operations
union_set = set1.union(set2)  # Union of two sets
print("Union:", union_set)
intersection_set = set1.intersection(set2)  # Intersection of two sets
print("Intersection:", intersection_set)
difference_set = set1.difference(set2)  # Elements in set1 but not in set2
print("Difference (set1 - set2):", difference_set)
symmetric_difference_set = set1.symmetric_difference(set2)    #
Elements in either set1 or set2, but not both
print("Symmetric Difference:", symmetric_difference_set)
# Set membership
print("Is 3 present in set1?", 3 in set1)  # Output: True
print("Is 9 present in set1?", 9 in set1)  # Output: False
# Adding and removing elements
set1.add(6)  # Add an element to the set
print("Set1 after adding 6:", set1)
set2.remove(8)  # Remove an element from the set
print("Set2 after removing 8:", set2)
# Iterating through a set
print("Elements in set1:")
for element in set1:
    print(element)
```

In this example:

- Define two sets set1 and set2.
- Perform set operations such as union, intersection, difference, and symmetric difference.
- Check for set membership using the in operator.
- Add and remove elements from sets using the add() and remove() methods.
- Iterate through a set using a for loop.

1.3.2 - MAP DATATYPE

Maps, also known as dictionaries, are Abstract Data Types (ADTs) that map unique keys to corresponding values. This is useful for scenarios where quick and efficient retrieval of information is necessary. For example, a university registrar might use student identification numbers to quickly access student records, avoiding issues with non-unique or incorrectly entered names.

Map ADT Definition

The Map ADT is defined by the following operations:

- ✓ Map(): Creates a new empty map.
- ✓ length(): Returns the number of key/value pairs in the map.
- ✓ contains(key): Determines if the given key is in the map and returns True if found, False otherwise.
- ✓ add(key, value): Adds a new key/value pair to the map or replaces the existing value if the key is already present. Returns True if a new key is added and False if the value is replaced.
- ✓ remove(key): Removes the key/value pair for the given key if it exists in the map, otherwise raises an exception.
- ✓ valueOf(key): Returns the value associated with the given key, or raises an exception if the key does not exist.
- ✓ iterator(): Creates and returns an iterator to traverse the keys in the map.

In Python, the term "maps" is commonly used to refer to the dictionary data structure. A dictionary is an unordered collection of key-value pairs, where each key is associated with a value. It's also known as a "mapping" because it maps keys to values. Here's an overview of dictionaries in Python:

CREATING A DICTIONARY:

Can create a dictionary by enclosing key-value pairs within curly braces {}.

```
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}
```

ACCESSING ELEMENTS:

Can access the value associated with a key using square brackets [].

```
print(my_dict['name'])  # Output: John
```

MODIFYING ELEMENTS:

Can modify the value associated with a key.

```
my_dict['age'] = 35
```

ADDING ELEMENTS:

Can add new key-value pairs to the dictionary.

```
my_dict['gender'] = 'Male'
```

REMOVING ELEMENTS:

Can remove key-value pairs using the del keyword or the pop() methoD)

```
del my_dict['city']  # Removes the 'city' key-value pair
my_dict.pop('age')  # Removes the 'age' key-value pair and returns the value
```

DICTIONARY METHODS:

Python dictionaries provide several built-in methods for performing various operations, such as keys(), values(), items(), update(), clear(), copy(), etC)

```
keys_list = my_dict.keys()  # Returns a list of all keys
values_list = my_dict.values()  # Returns a list of all values
items_list = my_dict.items()  # Returns a list of key-value tuples
```

ITERATING THROUGH A DICTIONARY:

You can iterate through the keys, values, or key-value pairs of a dictionary using loops.

```
for key in my_dict:
    print(key, my_dict[key])
# Or you can use the items() method for both key and value
for key, value in my_dict.items():
    print(key, value)
```

Dictionaries are widely used in Python due to their flexibility, efficiency, and ability to represent key-value mappings.

List-Based Implementation

The Map ADT can be implemented using a list, allowing dynamic resizing and avoiding the fixed size limitation of arrays. The implementation involves storing both keys and values together, ensuring their association is maintained.

Implementation Overview

- ✓ Initialization: The map is initialized with an empty list to hold the key/value pairs.
- ✓ Length: The length method returns the number of entries in the map.
- ✓ Containment Check: The containment check method determines if a key is present by searching through the list.
- ✓ Add Operation: Adds a new key/value pair or updates the value if the key already exists.
- ✓ Value Retrieval: Retrieves the value associated with a given key.
- ✓ Remove Operation: Removes the key/value pair for the specified key.
- ✓ Iterator: Provides an iterator to traverse the keys.

Detailed Implementation

The Map class uses a single list to store entries, with each entry being an instance of the _MapEntry class, which holds a key and a value.

```
class Map:
    # Creates an empty map instance.
    def __init__(self):
        self._entryList = list()
    # Returns the number of entries in the map.
    def __len__(self):
        return len(self._entryList)
    # Determines if the map contains the given key.
    def __contains__(self, key):
        ndx = self._findPosition(key)
        return ndx is not None
    # Adds a new entry to the map if the key does not exist. Otherwise,
the new value replaces the current value associated with the key.
    def add(self, key, value):
        ndx = self._findPosition(key)
        if ndx is not None:  # if the key was found
```

```
            self._entryList[ndx].value = value
            return False
        else:  # otherwise add a new entry
            entry = _MapEntry(key, value)
            self._entryList.append(entry)
            return True
    # Returns the value associated with the key.
    def valueOf(self, key):
        ndx = self._findPosition(key)
        assert ndx is not None, "Invalid map key."
        return self._entryList[ndx].value
    # Removes the entry associated with the key.
    def remove(self, key):
        ndx = self._findPosition(key)
        assert ndx is not None, "Invalid map key."
        self._entryList.pop(ndx)
    # Returns an iterator for traversing the keys in the map.
    def __iter__(self):
        return _MapIterator(self._entryList)
    # Helper method used to find the index position of a key. If the key
is not found, None is returned.
    def _findPosition(self, key):
        for i in range(len(self)):
            if self._entryList[i].key == key:
                return i
        return None
    # Storage class for holding the key/value pairs.
    class _MapEntry:
        def __init__(self, key, value):
            self.key = key
            self.value = value
```

Helper Methods

- ✓ _findPosition: Searches the list for a given key and returns its index if found, or None otherwise. This method is used by other methods to determine key presence and location efficiently.

The above implementation ensures that all key/value pairs maintain their association and provides a clear and structured way to manage and manipulate map entries. The use of a single list to store MapEntry objects encapsulates the key/value pairs and ensures operations like addition, removal, and searching are performed efficiently.

1.3.3 - MULTIDIMENSIONAL ARRAYS

Multi-dimensional arrays extend the concept of one-dimensional and two-dimensional arrays to more dimensions, enabling storage and access to data in a more complex structure. These arrays are useful for problems that require data organization beyond two dimensions, such as simulations, image processing, and scientific computations.

Abstract View of Multi-Dimensional Arrays

Multi-dimensional arrays store data in multiple dimensions and access elements using multi-component subscripts, such as $x_{i,j}$for three dimensions $y_{i,j,k}$. For example:

- ✓ A two-dimensional array is viewed as a table with rows and columns.
- ✓ A three-dimensional array can be visualized as a box of tables, with each table divided into rows and columns.

MULTIARRAY ABSTRACT DATA TYPE (ADT)

To handle multi-dimensional arrays in a programming language that does not natively support them (like Python), we define the MultiArray ADT. This ADT includes essential operations for managing multi-dimensional arrays.

Multi-dimensional arrays are arrays that have more than one dimension. They are often used to represent data in multiple dimensions, such as matrices or higher-dimensional tensors. In Python, multi-dimensional arrays can be implemented using nested lists or arrays provided by libraries like NumPy.

NESTED LISTS:

One way to represent multi-dimensional arrays in Python is by using nested lists. Each inner list represents a row, and the outer list contains all the rows. This creates a 2D array.

```
# Creating a 2D array (3x3) using nested lists
two_d_array = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

NUMPY ARRAYS:

NumPy is a powerful library in Python for numerical computing. It provides an efficient implementation of multi-dimensional arrays, known as numpy.ndarray.

```
import numpy as np
# Creating a 2D NumPy array
two_d_array_np = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

NumPy arrays offer several advantages over nested lists, including better performance, support for mathematical operations, and a wide range of functions for array manipulation.

ACCESSING ELEMENTS:

Can access elements of multi-dimensional arrays using indexing. For a 2D array, you specify both the row and column index.

```
element = two_d_array[1][2]  # Accessing element at row 1, column 2
```

ITERATING THROUGH A MULTI-DIMENSIONAL ARRAY:

Can iterate through each element of a multi-dimensional array using nested loops.

```
for row in two_d_array:
    for element in row:
        print(element, end=' ')
    print()  # Move to the next line after printing each row
```

Whether using nested lists or NumPy arrays, multi-dimensional arrays are essential for representing and working with data in multiple dimensions efficiently.

EXAMPLE:

Create an example with a three-dimensional array. We'll use NumPy for this purpose:

```
  import numpy as np
 # Define a 3D array (2x3x4)
three_d_array = np.array([[[1, 2, 3, 4],
   [5, 6, 7, 8],
   [9, 10, 11, 12]],
  [[13, 14, 15, 16],
   [17, 18, 19, 20],
   [21, 22, 23, 24]]])
# Accessing elements
print("Element at index (0, 1, 2):", three_d_array[0, 1, 2])  # Output: 7
print("Element at index (1, 2, 3):", three_d_array[1, 2, 3])  # Output: 24
# Modifying elements
three_d_array[0, 1, 2] = 100
print("Modified 3D array:")
print(three_d_array)
# Iterating through the array
print("Iterating through the 3D array:")
for matrix in three_d_array:
    for row in matrix:
        for element in row:
            print(element, end=' ')
        print()  # Move to the next line after printing each row
    print()  # Move to the next line after printing each matrix
```

In this example, we've created a 3D array with dimensions 2x3x4. We access and modify elements, and then iterate through the array to print its contents. This demonstrates working with multi-dimensional arrays in Python, specifically a three-dimensional array in this case.

MultiArray ADT Operations:

- ✓ MultiArray(d1, d2, ..., dn): Creates a multi-dimensional array with n dimensions, initializing all elements to None. The number of dimensions must be greater than 1, and each dimension size must be positive.
- ✓ dims(): Returns the number of dimensions in the array.
- ✓ length(dim): Returns the size of the specified dimension, with dimensions numbered starting from 1.
- ✓ clear(value): Sets every element in the array to the specified value.
- ✓ getitem(i1, i2, ..., in): Returns the value at the specified n-tuple index. All indices must be within the valid range.
- ✓ setitem(i1, i2, ..., in, value): Sets the value at the specified n-tuple index. All indices must be within the valid range.

Data Organization in Memory

Although multi-dimensional arrays are conceptually organized in multiple dimensions, they are stored as one-dimensional arrays in memory. This approach simplifies memory management but requires a method to convert multi-dimensional indices to one-dimensional offsets.

Storage Order:

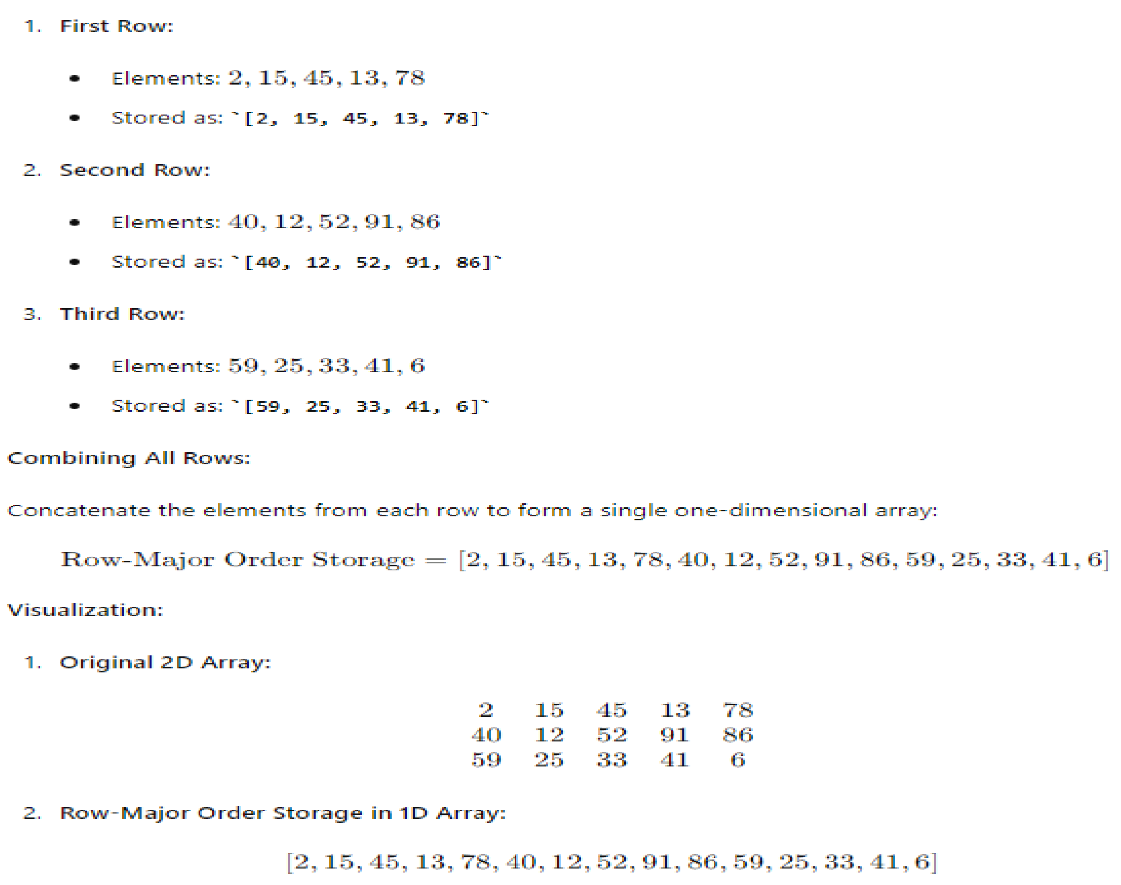
- ✓ Row-Major Order: Rows are stored sequentially. This is the default for most programming languages.
- ✓ Column-Major Order: Columns are stored sequentially. FORTRAN is a notable language using this method.

Example of Row-Major Order Storage

Given 2D Array:

Let's consider a 2D array with 3 rows and 5 columns:

$$\begin{matrix} 2 & 15 & 45 & 13 & 78 \\ 40 & 12 & 52 & 91 & 86 \\ 59 & 25 & 33 & 41 & 6 \end{matrix}$$



1. **First Row:**
   - Elements: $2, 15, 45, 13, 78$
   - Stored as: `[2, 15, 45, 13, 78]`
2. **Second Row:**
   - Elements: $40, 12, 52, 91, 86$
   - Stored as: `[40, 12, 52, 91, 86]`
3. **Third Row:**
   - Elements: $59, 25, 33, 41, 6$
   - Stored as: `[59, 25, 33, 41, 6]`

Combining All Rows:

Concatenate the elements from each row to form a single one-dimensional array:

$$\text{Row-Major Order Storage} = [2, 15, 45, 13, 78, 40, 12, 52, 91, 86, 59, 25, 33, 41, 6]$$

Visualization:

1. **Original 2D Array:**

$$\begin{matrix} 2 & 15 & 45 & 13 & 78 \\ 40 & 12 & 52 & 91 & 86 \\ 59 & 25 & 33 & 41 & 6 \end{matrix}$$

2. **Row-Major Order Storage in 1D Array:**

$$[2, 15, 45, 13, 78, 40, 12, 52, 91, 86, 59, 25, 33, 41, 6]$$

Let Us Sum Up

In this segment, we explore two fundamental data structures: Sets and Maps, along with their applications, and introduce the concept of Multi-Dimensional Arrays. Sets provide a collection of unique elements, allowing efficient membership testing and set operations such as union, intersection, and difference. Maps, also known as dictionaries in Python, associate keys with values, enabling fast retrieval and manipulation of data based on keys. Additionally, we delve into Multi-Dimensional Arrays, extending the concept of arrays to higher dimensions, facilitating storage and manipulation of structured data. Through practical examples and exercises, students gain proficiency in utilizing Sets, Maps, and Multi-Dimensional Arrays to address various data management and computation challenges efficiently.

Check Your Progress

1. What is the primary characteristic of a set data structure?
   - A) It allows duplicate elements
   - B) It stores elements in a sorted order
   - C) It allows only unique elements
   - D) It can store elements of different types
2. Which data structure in Python is commonly used to implement sets?
   - A) List
   - B) Tuple
   - C) Set
   - D) Dictionary
3. What operation allows combining the elements of two sets into a single set?
   - A) Union
   - B) Intersection
   - C) Difference
   - D) Symmetric difference
4. In a map data structure, what is a key-value pair?
   - A) Two elements stored adjacent to each other
   - B) A unique identifier associated with a value
   - C) A sorted collection of elements
   - D) An element stored in a set

5. Which of the following is NOT a characteristic of a map data structure?
    - A) Fast retrieval of values based on keys
    - B) Ability to store duplicate keys
    - C) Association of unique keys with values
    - D) Efficient manipulation of key-value pairs
6. How are multi-dimensional arrays represented in Python?
    - A) Using a nested list structure
    - B) Using a tuple of tuples
    - C) Using a dictionary of lists
    - D) Using a set of sets
7. What operation is commonly performed on sets to find common elements between two sets?
    - A) Union
    - B) Intersection
    - C) Difference
    - D) Symmetric difference
8. In a map data structure, what happens if a key-value pair with an existing key is added?
    - A) The new value replaces the existing value
    - B) The new value is appended to the existing values
    - C) The operation fails with an error
    - D) The new key-value pair is ignored
9. What is the primary advantage of using a map over a set?
    - A) Maps allow duplicate elements
    - B) Maps provide faster access to elements
    - C) Maps store elements in a sorted order
    - D) Maps are more memory-efficient
10. How are multi-dimensional arrays accessed in Python?
    - A) Using a single index
    - B) Using two indices (row and column)
    - C) Using a loop structure
    - D) Using built-in functions

Unit Summary

Abstract Data Types (ADTs) provide a theoretical framework for data structures, focusing on what operations are supported without specifying implementation details. Encapsulation in ADTs hides internal complexities, promoting easier maintenance and flexibility. Lists, Queues, Stacks, Sets, and Dictionaries are common ADTs, each with unique operations and applications. Lists allow element access and modification by position, while Queues and Stacks follow FIFO and LIFO principles, respectively. Sets enforce unique elements, contrasting with Bags that permit duplicates. Dictionaries store key-value pairs, facilitating fast lookups. Multi-dimensional arrays, like those in NumPy, efficiently handle numerical data. Iterators enable sequential access without exposing data representation. Python's built-in list methods, such as append, extend and remove enhance list functionality. Understanding these ADTs and their operations is crucial for efficient data management and algorithm implementation.

Glossary

- **Abstract Data Type (ADT):** A model for data types defined by their behavior from the user's perspective, focusing on the operations that can be performed on the data.
- **Encapsulation:** A principle where the internal details of a data structure are hidden, exposing only necessary functionalities to the user.
- **List:** An ordered collection of elements, allowing access, insertion, and deletion of elements at specific positions.
- **Set:** A collection of unique elements without any specific order, used for membership testing and set operations like union and intersection.
- **Dictionary (or Map):** A collection of key-value pairs, allowing fast retrieval, insertion, and deletion of values based on their keys.
- **Multi-dimensional Array:** An array containing more than one dimension, used to represent data in matrices or higher-dimensional spaces.
- **Iterator:** An object that provides a way to access elements of a collection sequentially without exposing the underlying structure.

Self – Assessment Questions

1. Analyze the concept of Abstract Data Types (ADTs) and explain their significance in data structure design.
2. Write about the principle of encapsulation and its role in maintaining data structure integrity.
3. Explain the operations and functionalities associated with the List Abstract Data Type (ADT).
4. Compare and contrast the operations of the Queue and Stack ADTs, highlighting their distinct characteristics.
5. Evaluate the advantages and disadvantages of using Sets as compared to other data structures like Lists or Dictionaries.
6. Discuss the key features and operations provided by the Dictionary (Map) ADT and its practical applications.
7. Analyze the use cases and benefits of using multi-dimensional arrays, particularly in numerical computations.
8. Explore the significance of iterators in providing sequential access to collection elements and their impact on data structure efficiency.
9. Examine the role of pointers in data structures and their importance in managing memory addresses.
10. Evaluate the efficiency and practical implications of using NumPy arrays compared to nested lists for multi-dimensional data representation.
11. Discuss the importance of algorithms in solving complex problems and their relationship with data structures.
12. Analyze the characteristics and applications of primitive data types in programming languages.
13. Examine the principle of FIFO and its relevance in data structures like queues.
14. Evaluate the role of encapsulation in maintaining data structure integrity and security.
15. Discuss the significance of abstraction in data structure design and its impact on code maintainability and readability.

Activities / Exercises / Case Studies

1. Implementation Practice: Assign students tasks to implement basic abstract data types (such as lists, queues, or stacks) in a programming language of their choice. They can then test and compare the efficiency of their implementations.
2. Data Structure Design Challenge: Divide students into groups and give them a real-world problem to solve using abstract data types. For example, they could design a system to manage a library's book inventory using dictionaries, lists, and queues.
3. Algorithm Analysis Exercise: Provide students with different algorithms for common operations on abstract data types (e.g., searching, sorting). Have them analyze the time and space complexity of each algorithm and compare their performance.
4. Debugging and Optimization Task: Give students code snippets implementing abstract data types with intentional bugs or inefficiencies. Challenge them to debug and optimize the code to improve its performance and functionality.
5. Problem-Solving Scenarios: Present students with problem-solving scenarios where they must choose the most appropriate abstract data type and algorithm to solve a given problem efficiently. Encourage them to justify their choices.
6. Coding Challenges: Organize coding challenges where students must solve programming problems using abstract data types. Provide a set of problems that require the use of lists, queues, stacks, or dictionaries to solve.
7. Case Study: Real-World Data Structures: Present students with case studies of real-world systems and applications (e.g., social media networks, online shopping platforms) and discuss the data structures and algorithms used to optimize performance and scalability.

Answers for check your progress

Modules	S. No.	Answers
Module 1	1.	B) A mathematical model for representing data and its associated operations
	2.	A) Adding two dates together
	3.	A) Storing elements with possible repetitions
	4.	B) To efficiently traverse and manipulate data structures

	5.	C) Elements may occur more than once in the collection
	6.	C) Traversing elements in a data structure
	7.	D) They improve performance by efficiently accessing elements
	8.	C) Bags allow for possible repetitions of elements
	9.	C) Tracking inventory items in a warehouse
	10.	B) They define the structure and behavior of data without specifying its implementation
Module 2	**1.**	C) Storing a collection of elements of the same type
	2.	A) List
	3.	B) It stores elements in multiple rows and columns
	4.	C) Transposition
	5.	B) It provides specific implementations of matrix operations
	6.	B) Using two indices (row and column)
	7.	C) Efficient access to elements using indices
	8.	A) Insertion
	9.	A) O(1)
	10.	A) Lists can store elements of different types
Module 3	**1.**	C) It allows only unique elements
	2.	C) Set
	3.	A) Union
	4.	B) A unique identifier associated with a value
	5.	B) Ability to store duplicate keys
	6.	A) Using a nested list structure
	7.	B) Intersection
	8.	A) The new value replaces the existing value

	9.	B) Maps provide faster access to elements
	10.	B) Using two indices (row and column)

Suggested Readings

1. Zelle, J. M. (2004). *Python programming: an introduction to computer science*. Franklin, Beedle & Associates, Inc..
2. Beazley, D., & Jones, B. K. (2013). *Python cookbook: Recipes for mastering Python 3*. " O'Reilly Media, Inc.".
3. Ascher, D., & Lutz, M. (1999). *Learning Python*. O'Reilly.

Open-Source E-Content Links

1. GeeksforGeeks - Data Structures
2. Khan Academy - Algorithms
3. Coursera - Data Structures and Algorithms Specialization
4. GeeksforGeeks - Arrays
5. W3Schools - Python Arrays
6. Khan Academy - Intro to Arrays
7. GeeksforGeeks - Sets
8. GeeksforGeeks - Maps
9. Real Python - Python Sets
10. Real Python - Dictionaries

References

1. B Downey, A. (2012). Think Python: How to Think Like a Computer Scientist-2e.
2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.

UNIT II – INTRODUCTION

Algorithm Analysis: Experimental Studies-Seven Functions-Asymptotic Analysis. **Recursion:** Illustrative Examples-Analyzing Recursive Algorithms-Linear Recursion-Binary Recursion- Multiple Recursion.

Introduction : Algorithm Analysis

UNIT OBJECTIVES

In this course segment, students will learn the fundamental concepts of algorithm analysis through experimental studies and asymptotic analysis. They will explore various types of functions and understand their behavior through empirical studies. Additionally, students will delve into recursion, including linear, binary, and multiple recursion, and analyze recursive algorithms to understand their efficiency and performance characteristics. Through practical examples and theoretical analysis, students will gain proficiency in evaluating algorithmic complexity and designing efficient algorithms for solving computational problems.

SECTION 2.1: ALGORITHM ANALYSIS

2.1.1 – EXPERIMENT STUDIES

Algorithm analysis is the process of determining the computational complexity of algorithms, specifically their time and space requirements. This analysis helps in understanding the efficiency of algorithms and choosing the most suitable one for a given problem.

Challenges of Experimental Analysis

Comparison Difficulty:

- ✓ Direct comparison of experimental running times of two algorithms is challenging unless experiments are conducted under identical hardware and software environments.
- ✓ Variation in hardware, software, and other concurrent processes on the computer can influence experimental results.

Limited Test Inputs:

- ✓ Experimental studies can only cover a limited set of test inputs, potentially excluding crucial input scenarios.
- ✓ The running times of inputs not included in the experiment may be significant but are not accounted for.

Full Implementation Requirement:

- ✓ Conducting experimental studies requires full implementation of the algorithm, which may be time-consuming.
- ✓ Implementing multiple algorithms for comparative analysis at the early design stages might not be feasible or efficient.

Moving Beyond Experimental Analysis

Objectives:

Relative Efficiency Evaluation:

- ✓ Evaluate the relative efficiency of algorithms independently of hardware and software environments.

High-Level Description Study:

- ✓ Analyze algorithm efficiency based on high-level descriptions, eliminating the need for full implementation.

Consider All Inputs:

- ✓ Account for the running times of all possible inputs, not just those included in experimental studies.

Counting Primitive Operations

Definition:

- ✓ Primitive operations are defined as basic operations with constant execution times.
- ✓ Examples include assigning identifiers, arithmetic operations, comparisons, list element accesses, function calls, and returns.

Operation Counting:

- ✓ Instead of measuring specific execution times, count the number of primitive operations executed by the algorithm.
- ✓ The count of primitive operations correlates with the algorithm's actual running time, assuming similar execution times for different primitive operations.

Measuring Operations as a Function of Input Size

- ✓ Associate each algorithm with a function $f(n)$ representing the number of primitive operations performed as a function of input size n

- ✓ Different input sizes n lead to varying numbers of primitive operations, forming the basis for analyzing algorithm efficiency.

Focusing on the Worst-Case Input

Worst-Case Analysis:

- ✓ Express algorithm running time as a function of input size based on the worst-case scenario.
- ✓ Identifying the worst-case input is typically simpler and does not require complex probability distributions.
- ✓ Designing algorithms for the worst-case scenario often results in stronger and more robust solutions.
- ✓ By understanding and implementing these approaches, we can analyze algorithm efficiency effectively without the constraints and limitations of experimental studies.

2.1.2– SEVEN FUNCTIONS

1. Constant Function

- ✓ Defined as $f(n) = c$, where c is a fixed constant.
- ✓ Represents a constant number of steps needed for basic operations on a computer.
- ✓ Often represented as g(n)=1 for simplicity.

2. Logarithm Function

- ✓ Defined as $f(n)=\log_b n$ where b>1 is a constant.
- ✓ Commonly used in computer science with base b=2 due to binary representation.
- ✓ Useful for analyzing algorithms with repeatedly halving inputs.

3. Linear Function

- ✓ Defined as f(n)=n.
- ✓ Represents a linear relationship between input size and number of operations.
- ✓ Occurs when a single basic operation is performed for each element in the input.

4. N-Log-N Function

- ✓ Defined as f(n)=nlogn.
- ✓ Grows slightly faster than linear but significantly slower than quadratic.

- ✓ Common in algorithms like sorting, where optimal performance is achieved in O(nlogn) time.

5. Quadratic Function

- ✓ Defined as $f(n)=n^2$
- ✓ Arises from nested loops where each iteration of the outer loop triggers a linear number of operations in the inner loop.
- ✓ Represents a substantial increase in operations compared to linear time.

6. Cubic Function

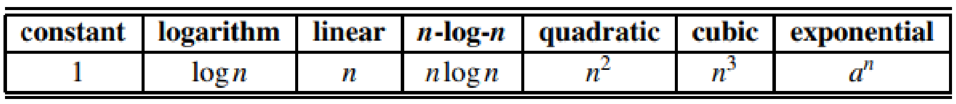
- ✓ Defined as $f(n)=n^3$
- ✓ Less common in algorithm analysis but appears occasionally.
- ✓ Represents an even greater increase in operations compared to quadratic time.

7. Exponential Function

- ✓ Defined as $f(n)=b^n$, where b is a positive constant.
- ✓ Commonly used with b=2 in algorithm analysis.
- ✓ Represents rapid growth in operations with increasing input size.

Comparing Growth Rates

- ✓ Ideally, algorithms should run in constant or logarithmic time for efficient performance.
- ✓ O(nlogn) time complexities are preferable for practical algorithms.
- ✓ Quadratic and cubic time complexities are less practical, while exponential time complexities are generally infeasible for large inputs.



constant	logarithm	linear	n-log-n	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

Table 3.1: Classes of functions. Here we assume that $a > 1$ is a constant.

Ceiling and Floor Functions

- ✓ Used in algorithm analysis to convert real-valued results (like logarithms) into integers.
- ✓ Ceiling function ($\lceil x \rceil$) returns the smallest integer greater than or equal to x

- ✓ Floor function (⌊x⌋) returns the largest integer less than or equal to x

2.1.3– ASYMPTOTIC ANALYSIS

Asymptotic analysis is a method used in computer science to describe the efficiency of algorithms, focusing on the growth rate of the running time as a function of the input size n. This method uses mathematical notation that disregards constant factors and lower-order terms, allowing us to concentrate on the primary factor affecting the algorithm's growth rate.

The "Big-Oh" Notation

- ✓ Definition: A function f(n) is O(g(n)) if there exist constants c>0 and $n_0 \geq 1$ such that f(n)≤cg(n) for all $n \geq n_0$.
- ✓ Purpose: Big-Oh notation provides an upper bound on the growth rate of a function, indicating that f(n) grows at most as fast as g(n) up to a constant factor.
- ✓ Usage: We say " f(n) is O(g(n))" rather than using inequalities or equalities with big-Oh notation.

Properties of Big-Oh Notation

- ✓ Ignoring Constant Factors and Lower-Order Terms: Big-Oh notation focuses on the dominant term, ignoring constants and less significant terms.
- ✓ Simplest Characterization: When using big-Oh notation, we strive to describe the function as accurately and simply as possible.

Algorithm Analysis Using Big-Oh Notation

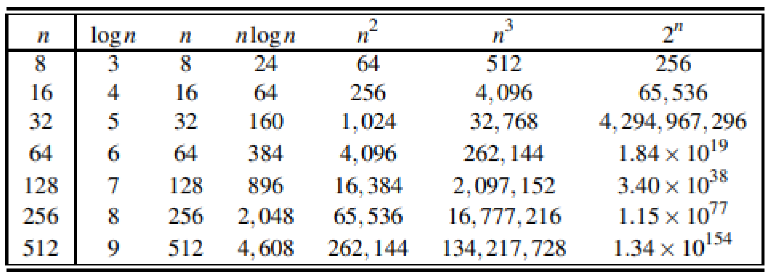
Example (Finding Maximum in a List):

```
def find_max(data):
    biggest = data[0]
    for val in data:
        if val > biggest:
            biggest = val
    return biggest
```

Running Time Analysis:

- ✓ Initialization (constant time): A fixed number of operations.
- ✓ Loop (linear time): Executes n times, each iteration taking constant time.
- ✓ Conclusion: The running time of find_max is O(n).

We illustrate the growth rates of the seven functions in Table 3.2. (See also Figure 3.4 from Section 3.2.1.)



n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Table 3.2: Selected values of fundamental functions in algorithm analysis.

- ✓ Algorithm analysis using big-Oh notation provides insights into the efficiency of algorithms.
- ✓ Python implementations are used to illustrate various algorithmic concepts, leveraging the list class for array representation.
- ✓ The efficiency of different operations is discussed in the context of Python's list class behaviors.

Constant-Time Operations:

- ✓ Operations like len(data) and accessing elements via data[j] in Python lists are analyzed.
- ✓ These operations are shown to execute in constant time, denoted by O(1), due to efficient list implementations.

Revisiting the Problem of Finding the Maximum of a Sequence:

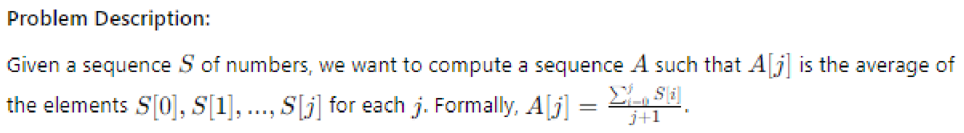
- ✓ The find_max algorithm for identifying the maximum value in a sequence is examined.
- ✓ Its time complexity is confirmed to be O(n) based on the number of iterations and comparisons involved.

Further Analysis of the Maximum-Finding Algorithm:

- ✓ The expected number of updates to the maximum value in find_max for randomly ordered sequences is explored.]
- ✓ Probability theory is applied to derive the expected number of updates, revealing it to be O(log n).

Prefix Averages:

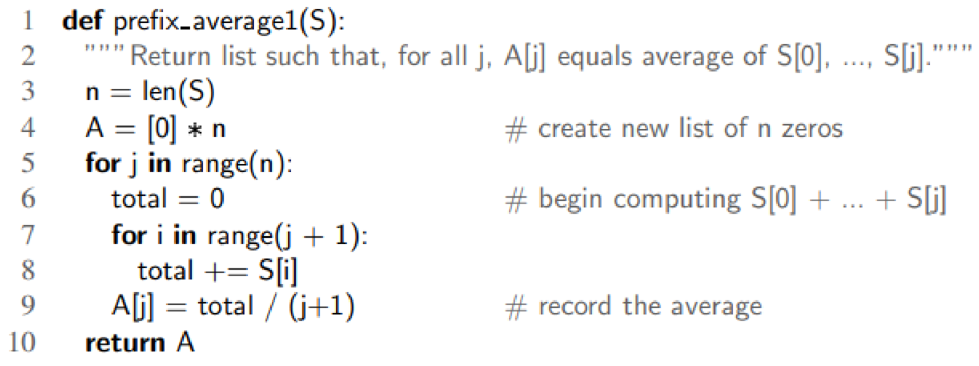
- ✓ The problem of computing prefix averages of sequences is introduced, along with its applications.



Problem Description:

Given a sequence S of numbers, we want to compute a sequence A such that $A[j]$ is the average of the elements $S[0], S[1], \ldots, S[j]$ for each j. Formally, $A[j] = \frac{\sum_{i=0}^{j} S[i]}{j+1}$.

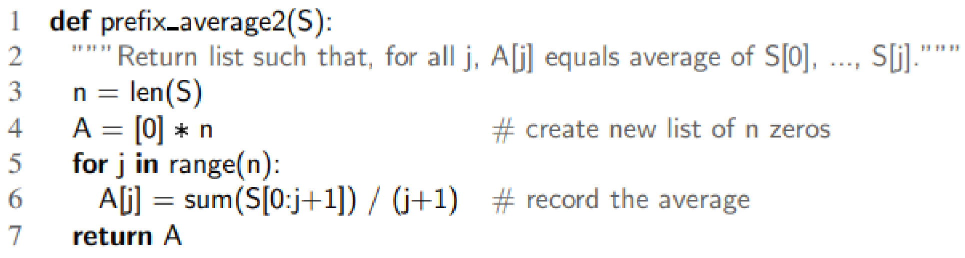
- ✓ Three different implementations, each with varying time complexities, are analyzed:
- ✓ Quadratic-time algorithm (prefix_average1)



```
def prefix_average1(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                    # create new list of n zeros
  for j in range(n):
    total = 0                    # begin computing S[0] + ... + S[j]
    for i in range(j + 1):
      total += S[i]
    A[j] = total / (j+1)         # record the average
  return A
```

Code Fragment 3.2: Algorithm prefix_average1.

- ✓ **Description:** This algorithm computes each prefix average independently by iterating over all elements up to *j* for each *j*.
- ✓ **Time Complexity:** O(n^2) because for each *j* (from 0 to n-1), it computes the sum of *j* +1 elements. This results in 1 + 2 + 3 + ... + n operations, which sums to O(n^2).
- ✓ Quadratic-time algorithm with simplified expression (prefix_average2)



```
def prefix_average2(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                    # create new list of n zeros
  for j in range(n):
    A[j] = sum(S[0:j+1]) / (j+1) # record the average
  return A
```

Code Fragment 3.3: Algorithm prefix_average2.

- ✓ **Description**: This algorithm simplifies the computation by using Python's built-in sum function to compute the sum of elements up to *j* for each *j* .

- ✓ **Time Complexity**: O(n^2) because although it looks simpler, the sum(S[0:j+1]) operation is O(j), and it's performed for each *j* j from 0 to n-1, resulting in the same number of operations as prefix_average1.
- ✓ Linear-time algorithm (prefix_average3)

```
def prefix_average3(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                      # create new list of n zeros
  total = 0                        # compute prefix sum as S[0] + S[1] + ...
  for j in range(n):
    total += S[j]                  # update prefix sum to include S[j]
    A[j] = total / (j+1)           # compute average based on current sum
  return A
```

Code Fragment 3.4: Algorithm prefix_average3.

- ✓ **Description:** This algorithm improves efficiency by maintaining a running total of the sum of elements seen so far. For each *j*, it updates the running total and computes the average in constant time.
- ✓ **Time Complexity:** O(n) because it processes each element of *S* exactly once, maintaining a running sum and computing each prefix average in constant time.
- ✓ prefix_average1 and prefix_average2 both have a time complexity of O(n^2) because they involve summing elements for each prefix separately.
- ✓ prefix_average3 has an optimal time complexity of O(n) as it leverages a running total to avoid redundant summation operations.

Algorithm analysis plays a crucial role in understanding the efficiency of algorithms. Python implementations serve as practical examples for demonstrating algorithmic concepts.

Three-Way Set Disjointness:

Problem Description: Given three sequences A, B, and C, the task is to determine if there are any common elements present in all three sequences.

Algorithm 1 (disjoint1):

```
def disjoint1(A, B, C):
  """Return True if there is no element common to all three lists."""
  for a in A:
    for b in B:
      for c in C:
        if a == b == c:
          return False        # we found a common value
  return True                 # if we reach this, sets are disjoint
```

Code Fragment 3.5: Algorithm disjoint1 for testing three-way set disjointness.

Description: Uses nested loops to check all possible triples of elements from A, B, and C.

Time Complexity: O(n^3) where n is the size of each sequence.

Algorithm 2 (disjoint2):

```
def disjoint2(A, B, C):
  """Return True if there is no element common to all three lists."""
  for a in A:
    for b in B:
      if a == b:              # only check C if we found match from A and B
        for c in C:
          if a == c           # (and thus a == b == c)
            return False      # we found a common value
  return True                 # if we reach this, sets are disjoint
```

Code Fragment 3.6: Algorithm disjoint2 for testing three-way set disjointness.

Description: Improves upon disjoint1 by stopping the iteration over C when a match is found between elements from A and B.

Time Complexity: O(n^2) where n is the size of each sequence.

Element Uniqueness:

```
def unique1(S):
  """Return True if there are no duplicate elements in sequence S."""
  for j in range(len(S)):
    for k in range(j+1, len(S)):
      if S[j] == S[k]:
        return False              # found duplicate pair
  return True                     # if we reach this, elements were unique
```

Code Fragment 3.7: Algorithm unique1 for testing element uniqueness.

Problem Description: Given a single sequence S, determine if all elements in S are distinct.

Algorithm 1 (unique1):

```
def unique1(S):
  """Return True if there are no duplicate elements in sequence S."""
  for j in range(len(S)):
    for k in range(j+1, len(S)):
      if S[j] == S[k]:
        return False               # found duplicate pair
  return True                      # if we reach this, elements were unique
```

Code Fragment 3.7: Algorithm unique1 for testing element uniqueness.

Description: Uses nested loops to check all distinct pairs of indices in S for duplicate elements.

Time Complexity: O(n^2) where n is the length of sequence S.

Algorithm 2 (unique2):

```
def unique2(S):
  """Return True if there are no duplicate elements in sequence S."""
  temp = sorted(S)                 # create a sorted copy of S
  for j in range(1, len(temp)):
    if S[j−1] == S[j]:
      return False                 # found duplicate pair
  return True                      # if we reach this, elements were unique
```

Code Fragment 3.8: Algorithm unique2 for testing element uniqueness.

Description: Utilizes sorting to solve the element uniqueness problem. Sorts the sequence and checks for consecutive duplicates.

Time Complexity: O(n log n) where n is the length of sequence S, considering the sorting operation.

Let Us Sum Up

In algorithm analysis, experimental studies involve measuring the actual running time or space usage of algorithms on various inputs to empirically evaluate their performance. Asymptotic analysis, on the other hand, provides a theoretical framework to describe an algorithm's efficiency in terms of input size, using functions such as constant , logarithmic , linear , linearithmic quadratic , cubic and exponential . By characterizing algorithms with these seven functions, we can predict their scalability and identify potential inefficiencies, complementing empirical observations with rigorous theoretical bounds.

Check Your Progress

1. What is the purpose of asymptotic analysis in algorithm analysis?
    A) To measure the exact running time of an algorithm.
    B) To predict the performance of an algorithm as the input size grows.
    C) To optimize the space complexity of an algorithm.
    D) To compare algorithms based on their constant factors.
2. In the context of experimental studies, what is typically measured to evaluate an algorithm?
    A) Theoretical performance.
    B) Worst-case complexity.
    C) Actual running time or space usage.
    D) Best-case complexity.
3. Which of the following complexities is the most efficient for large input sizes?
    A) O(1)
    B) O(nlogn)
    C) O(n ^{2})
    D) O(2 n)
4. In asymptotic analysis, what does the notation O(n) represent?
    A) Constant time complexity.
    B) Linear time complexity.
    C) Quadratic time complexity.
    D) Exponential time complexity.
5. Which term best describes the behavior of an algorithm with O(nlogn) complexity?
    A) Linear
    B) Linearithmic
    C) Quadratic
    D) Logarithmic
6. When conducting experimental studies on algorithms, which factors are important to consider?
    A) Input size and implementation details.
    B) Only the theoretical time complexity.
    C) Only the hardware specifications.

D) None of the above.

SECTION 2.2: RECURSION

2.2.1– ILLUSTRATIVE EXAMPLES

1. The Factorial Function

The factorial of a positive integer n!, is the product of the integers from 1 to n

Special case: 0!=1 by convention.

Formal definition:

- ✓ n!=1 if n=0
- ✓ n!=n·(n−1)! if n≥1

Example:

5!=5·4·3·2·1=120

The factorial function represents the number of permutations of n distinct items.

Recursive Definition:

- ✓ n!=n·(n−1)!
- ✓ Base case: n=0
- ✓ Recursive case: n≥1

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

- ✓ Recursion Trace: Shows the sequence of function calls and returns.

2. Drawing an English Ruler

- ✓ An English ruler consists of major and minor ticks.
- ✓ Major ticks are at whole inches, and minor ticks are between the major ticks at intervals of 1/2, 1/4, etc.
- ✓ The length of ticks decreases as the interval decreases.

Recursive Approach:

- ✓ A central tick of length L divides the ruler into two intervals each with central tick length L−1.

Recursive definition:

- ✓ Draw an interval of length L−1
- ✓ Draw the central tick of length L
- ✓ Draw another interval of length L−1

Python Implementation:

```
def draw_line(tick_length, tick_label=''):
    line = '-' * tick_length
    if tick_label:
        line += ' ' + tick_label
    print(line)
def draw_interval(center_length):
    if center_length > 0:
        draw_interval(center_length - 1)
        draw_line(center_length)
        draw_interval(center_length - 1)
def draw_ruler(num_inches, major_length):
    draw_line(major_length, '0')
    for j in range(1, 1 + num_inches):
        draw_interval(major_length - 1)
        draw_line(major_length, str(j))
```

Recursion Trace:

- ✓ Shows the nested function calls for drawing the intervals.

Binary Search

- ✓ Efficiently locates a target value within a sorted sequence.
- ✓ Maintains two parameters, low and high, to track the search range.
- ✓ Compares the target value to the middle element:
    - o If equal, the target is found.
    - o If less, search in the left half.
    - o If greater, search in the right half.

```
def binary_search(data, target, low, high):
    if low > high:
        return False
    else:
        mid = (low + high) // 2
        if target == data[mid]:
            return True
                elif target < data[mid]:
```

```
            return binary_search(data, target, low, mid - 1)
        else:
            return binary_search(data, target, mid + 1, high)
```

File systems are recursive, with directories containing files and other directories. Operations like computing disk usage are naturally implemented recursively.

Disk Usage Algorithm:

- ✓ Calculates the total disk space used by a directory and its contents.
- ✓ Adds the immediate disk space used by each entry and recursively sums the space used by nested entries.

```
import os
def disk_usage(path):
    total = os.path.getsize(path)
    if os.path.isdir(path):
        for filename in os.listdir(path):
            childpath = os.path.join(path, filename)
            total += disk_usage(childpath)
    print(f'{total:<7} {path}')
    return total
```

Recursion Trace:

The output shows the disk usage for each directory and its nested contents. The recursive calls are completed in order, reflecting the cumulative disk usage. Recursion is a technique where a function calls itself to solve smaller instances of the same problem. Analyzing recursive algorithms involves understanding their recurrence relations and solving them to get the time complexity.

Example of Recursive Function: Factorial

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

<u>Recurrence Relation:</u>

$T(n) = T(n-1) + O(1)$

Solution:

This solves to $O(n)$ because the function makes n recursive calls.

Example of Recursive Function: Fibonacci

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Recurrence Relation:

T(n) = T(n-1) + T(n-2) + O(1)

Solution:

This solves to O(2^n) because the number of calls grows exponentially.

2.2.2– ANALYZING RECURSION ALGORITHMS

Examples of Recursive Algorithms

Computing Factorials

The factorial computation is straightforward to analyze. For factorial(n), there are n+1 activations as the parameter decreases from n to 0. Each activation executes a constant number of operations, leading to an overall time complexity of O(n).

Drawing an English Ruler

To analyze the English ruler drawing algorithm, we consider the number of lines of output generated by draw_interval(c). Each call to draw_interval(c) results in 2^c - 1 lines of output, established by induction. The algorithm runs in O(2^c) time due to the exponential growth of output lines.

Performing a Binary Search

The binary search algorithm executes a constant number of operations per recursive call. The number of recursive calls is proportional to the logarithm of the input size n, leading to a time complexity of O(log n). Each call reduces the search space by half, ensuring efficient performance.

Computing Disk Space Usage

For computing disk space usage, the problem size n is the number of entries in the file system. There are exactly n recursive calls, one for each entry. The overall computation time is O(n) because there are O(n) recursive calls, each performing a constant amount of work outside a loop that runs in O(n) time across all calls. This leads to an overall linear time complexity due to the amortization of the for-loop iterations.

Recursion Run Amok

Recursion can be inefficient if poorly implemented. The unique3 function demonstrates inefficiency by checking uniqueness through a recursive approach that results in exponential time complexity O(2^n). This inefficiency arises because each recursive call leads to two more recursive calls, creating an exponential growth in the number of invocations.

Fibonacci Numbers

Inefficient Fibonacci Calculation

The bad_fibonacci function calculates Fibonacci numbers using a direct recursive approach, resulting in an exponential number of calls. The time complexity is O(2^n) due to redundant calculations.

Efficient Fibonacci Calculation

The good_fibonacci function avoids redundant calculations by returning a pair of consecutive Fibonacci numbers, resulting in a linear time complexity O(n). Each recursive call computes the next pair in constant time.

Maximum Recursive Depth in Python

Python imposes a limit on the maximum recursion depth to prevent infinite recursion, which would otherwise consume resources rapidly. The default limit is around 1000 nested calls. For algorithms requiring deeper recursion, this limit can be adjusted using the sys module with sys.setrecursionlimit.

Recursive Analysis: Each activation is analyzed individually, summing the operations across all activations.

Factorials: O(n) time due to n+1 activations with constant operations.

English Ruler: O(2^c) time due to exponential growth in output lines.

Binary Search: O(log n) time as each call reduces the search space by half.

Disk Space Usage: O(n) time by considering cumulative loop iterations.

Inefficient Recursion: Demonstrated by unique3 and bad_fibonacci with exponential time complexities.

Efficient Recursion: Achieved with good_fibonacci using linear recursion.

Python Recursion Limit: Can be adjusted to accommodate deeper recursive needs using sys module functions.

2.2.3 LINEAR RECURSION

Linear recursion involves a recursive algorithm where each recursive call leads to only one further recursive call until a base case is reached. This type of recursion forms a linear chain of function calls. In a linear recursion, the recursive call reduces the problem size by a constant amount with each recursion.

Characteristics:

- ✓ Each recursive call leads to only one further recursive call.
- ✓ Problem size decreases linearly with each recursive call until it reaches a base case.
- ✓ Typically used for problems where the solution can be derived by repeatedly reducing the problem size by a fixed amount.
- ✓ Example: Factorial calculation, where each recursive call decrements the input value by 1 until reaching the base case of 0.

```
def linear_recursion(n):
    """A linear recursive function."""
    if n == 0:
        return
    else:
        print(n)
        linear_recursion(n - 1)
# Testing the linear recursion function
print("Linear Recursion:")
linear_recursion(5)
```

OUTPUT

```
Linear Recursion:
5
4
3
2
1
```

2.2.4 – BINARY RECURSION

Binary recursion involves a recursive algorithm where each recursive call leads to two further recursive calls until a base case is reached. This type of recursion results in a binary tree structure of function calls. In binary recursion, the problem size typically reduces by half with each recursion.

Characteristics:

- ✓ Each recursive call leads to two further recursive calls.
- ✓ Problem size decreases exponentially with each recursive call until it reaches a base case.
- ✓ Commonly used for problems where the solution can be derived by dividing the problem into two equal or nearly equal parts.
- ✓ Example: Binary search algorithm, where each recursive call divides the search space in half.

```
def binary_recursion(n):
    """A binary recursive function."""
    if n == 0:
        return
    else:
        print(n)
        binary_recursion(n // 2)
        binary_recursion(n // 2)
# Testing the binary recursion function
print("Binary Recursion:")
binary_recursion(8)
```

OUTPUT

```
Binary Recursion:
8
4
2
1
1
2
1
1
```

2.2.5– MULTIPLE RECURSION

Multiple recursion involves a recursive algorithm where each recursive call leads to multiple further recursive calls until a base case is reached. This type of recursion results in branching structures with more than two recursive calls per level. In multiple recursion, the problem size reduction may vary with each recursive call.

Characteristics:

- ✓ Each recursive call leads to multiple further recursive calls, often with varying parameters.
- ✓ Problem size reduction pattern may vary with each recursive call.
- ✓ Suitable for problems where the solution involves multiple subproblems, each of which may need further subdivision.

```
def multiple_recursion(n):
    """A multiple recursive function."""
    if n == 0:
        return
    else:
        print(n)
        multiple_recursion(n - 1)
        multiple_recursion(n - 2)
# Testing the multiple recursion function
print("Multiple Recursion:")
multiple_recursion(5)
```

OUTPUT

```
Multiple Recursion:
5
4
3
2
1
3
2
1
```

Example: Generating permutations or combinations, where each recursive call involves considering multiple possibilities for the next element.

```
# Linear Recursion
def linear_recursion(n):
    """A linear recursive function."""
    if n == 0:
        return
    else:
        print(n)
        linear_recursion(n - 1)
# Binary Recursion
def binary_recursion(n):
    """A binary recursive function."""
    if n == 0:
        return
    else:
        print(n)
        binary_recursion(n // 2)
        binary_recursion(n // 2)
# Multiple Recursion
def multiple_recursion(n):
    """A multiple recursive function."""
    if n == 0:
        return
    else:
        print(n)
        multiple_recursion(n - 1)
        multiple_recursion(n - 2)
# Testing the functions
print("Linear Recursion:")
linear_recursion(5)
print("\nBinary Recursion:")
binary_recursion(8)
print("\nMultiple Recursion:")
multiple_recursion(5)
```

- ✓ Linear Recursion: The function linear_recursion recursively decrements n until it reaches 0, printing each value along the way. It demonstrates a simple linear recursion where each recursive call results in only one further recursive call.
- ✓ Binary Recursion: The function binary_recursion recursively divides n by 2 until it reaches 0, printing each value along the way. It demonstrates binary recursion where each recursive call results in two further recursive calls.
- ✓ Multiple Recursion: The function multiple_recursion recursively decrements n by 1 and 2 alternatively until it reaches 0, printing each value along the way. It demonstrates multiple recursion where each recursive call results in multiple further recursive calls.

Let Us Sum Up

Recursion is a powerful technique in algorithm design, where functions call themselves to solve smaller instances of the same problem. Linear recursion involves functions making a single recursive call, such as in computing factorials, resulting in linear time complexity. Binary recursion, seen in algorithms like binary search, involves two recursive calls and typically has logarithmic complexity. Multiple recursion involves more than two calls, as illustrated by the inefficient Fibonacci algorithm, often leading to exponential complexity. Optimizing these algorithms, such as using memoization for Fibonacci, can significantly improve performance. Analyzing these recursive patterns is crucial for designing efficient algorithms.

Check Your Progress

1. Which of the following algorithms uses binary recursion?
   A) Factorial computation
   B) Linear search
   C) Binary search
   D) Bubble sort
2. How many recursive calls are made in binary recursion?
   A) One
   B) Two
   C) Three
   D) Four
3. In the context of recursion, what does O(log n) represent?
   A) Exponential time
   B) Constant time
   C) Linear time
   D) Logarithmic time
4. Which recursion type involves more than two recursive calls?
   A) Linear recursion
   B) Binary recursion
   C) Multiple recursion
   D) Exponential recursion
5. What is the main drawback of multiple recursion?
   A) It is difficult to implement
   B) It always leads to infinite loops
   C) It has exponential time complexity
   D) It cannot solve complex problems

6. How can the efficiency of the Fibonacci algorithm be improved?
    A) By increasing the number of recursive calls
    B) By using linear recursion instead of binary recursion
    C) By implementing memoization
    D) By removing recursion entirely

7. Which technique can be used to analyze the efficiency of recursive algorithms?
    A) Memoization
    B) Binary search
    C) Dynamic programming
    D) Recurrence equations
8. What is the purpose of memoization in recursive algorithms?
    A) To decrease memory usage
    B) To improve time complexity
    C) To store previously computed results
    D) To increase the number of recursive calls
9. Which of the following is an example of linear recursion?
    A) Binary search
    B) Tower of Hanoi
    C) Factorial computation
    D) Quicksort
10. What is the primary advantage of binary recursion over linear recursion?
    A) It requires fewer recursive calls
    B) It has a lower time complexity
    C) It can solve a wider range of problems
    D) It is easier to implement
11. In the context of recursion, what does exponential time complexity imply?
    A) The time taken doubles with each recursive call
    B) The time taken increases linearly with input size
    C) The time taken increases exponentially with input size
    D) The time taken remains constant regardless of input size
12. Which recursion type is commonly used in algorithms like factorial computation?

A) Linear recursion
B) Binary recursion
C) Multiple recursion
D) Exponential recursion

13. What is the primary focus of analyzing recursive algorithms?
A) Increasing the number of recursive calls
B) Reducing memory usage
C) Optimizing time complexity
D) Simplifying the implementation process

Unit Summary

In Unit II, we explored the fundamentals of algorithm analysis and recursion. We covered experimental studies for practical performance insights and learned about the seven common growth functions to classify algorithm complexity. Asymptotic analysis was introduced to theoretically compare algorithms using Big-O, Big-Theta, and Big-Omega notations. We examined recursion through illustrative examples, including factorials and the Fibonacci sequence, and discussed how to analyze recursive algorithms using recurrence relations. Different types of recursion—linear, binary, and multiple—were explained, highlighting their respective time complexities. Understanding these concepts aids in evaluating and designing efficient algorithms.

Glossary

- **Algorithm Analysis**: The process of determining the computational complexity of algorithms, which includes time complexity (how fast an algorithm runs) and space complexity (how much memory it uses).
- **Asymptotic Analysis**: A method of describing the behavior of algorithms as the input size grows. It helps in comparing the efficiency of different algorithms.
- **Big-O Notation (O):** A mathematical notation used to describe the upper bound of an algorithm's running time or space requirements in terms of the input size. It represents the worst-case scenario.
- **Big-Omega Notation (Ω):** A notation used to describe the lower bound of an algorithm's running time or space requirements. It represents the best-case scenario.

- **Big-Theta Notation (Θ):** A notation used to describe the exact bound of an algorithm's running time or space requirements. It represents both the upper and lower bounds.
- **Constant Time (O(1)):** An algorithm is said to run in constant time if its running time does not change with the size of the input.
- **Logarithmic Time (O(log n)):** An algorithm is said to run in logarithmic time if its running time grows logarithmically as the input size increases.
- **Linear Time (O(n)):** An algorithm is said to run in linear time if its running time grows linearly with the size of the input.
- **Log-linear Time (O(n log n)):** An algorithm is said to run in log-linear time if its running time grows in proportion to n log n, where n is the input size.
- **Quadratic Time (O(n^2)):** An algorithm is said to run in quadratic time if its running time grows proportionally to the square of the input size.
- **Cubic Time (O(n^3)):** An algorithm is said to run in cubic time if its running time grows proportionally to the cube of the input size.
- **Exponential Time (O(2^n)):** An algorithm is said to run in exponential time if its running time doubles with each addition to the input size.
- **Recursive Algorithm:** An algorithm that calls itself in order to solve a problem.
- **Recursive Case:** The part of a recursive function that includes the recursive call.
- **Recurrence Relation:** An equation or inequality that describes the running time of a recursive algorithm in terms of the running time of smaller instances.
- **Linear Recursion:** A type of recursion where the function makes a single recursive call.
- **Binary Recursion:** A type of recursion where the function makes two recursive calls.
- **Multiple Recursion**: A type of recursion where the function makes more than two recursive calls.

Self – Assessment Questions

1. Analyze the time complexity of the binary search algorithm.
2. Analyze how the depth of recursion impacts the space complexity of a recursive algorithm.

3. Explain the difference between Big-O, Big-Theta, and Big-Omega notations.
4. Explain why divide-and-conquer algorithms often use binary recursion.
5. Compare the time complexity of merge sort and bubble sort.
6. Compare linear and binary recursion in terms of their use cases and efficiency.
7. Discuss the advantages and disadvantages of using experimental studies for algorithm analysis.
8. Discuss why tail recursion can be optimized by compilers and how it affects performance.
9. Evaluate the impact of input size on the performance of quadratic time complexity algorithms.
10. Evaluate the benefits and limitations of using recurrence relations to analyze recursive algorithms.
11. Illustrate how the Master Theorem is used to solve recurrence relations in divide-and-conquer algorithms.
12. Illustrate the concept of multiple recursion with the naive Fibonacci sequence algorithm.
13. Justify why merge sort is preferred over quick sort in worst-case scenarios.
14. Justify the use of Big-O notation in the analysis of algorithms.
15. Summarize the key steps in solving a problem using a divide-and-conquer approach.

Activities / Exercises / Case Studies

1.Experiment with Sorting Algorithms:

- Implement different sorting algorithms (e.g., bubble sort, merge sort, quick sort) in your preferred programming language.
- Measure their execution time with varying input sizes and plot the results to visualize their time complexities.

2. Recursion Visualization:

- Write a recursive function for calculating the factorial of a number.
- Trace the function calls on paper or using a visualization tool to understand how recursion works.

3.Asymptotic Analysis Practice:

- Given a set of functions, practice deriving their Big-O, Big-Theta, and Big-Omega notations.
- Compare and rank these functions based on their growth rates.

4. Divide-and-Conquer Algorithm Implementation:

- Implement a divide-and-conquer algorithm, such as the merge sort.
- Break down each step (divide, conquer, and combine) and explain how they contribute to the overall time complexity.

Exercises

1. Analyze Recursive Algorithms:

- Given the recursive definition of the Fibonacci sequence, derive its time complexity using a recurrence relation.
- Solve the recurrence relation using the Master Theorem.

2.Algorithm Optimization:

- Optimize a naive recursive solution for the Fibonacci sequence using memoization or dynamic programming.
- Compare the time and space complexity of the naive and optimized versions.

3.Experimental Analysis:

- Write a program that implements linear search and binary search.
- Test both algorithms with various input sizes and measure their performance.
- Analyze and compare the results with their theoretical time complexities.

4. Complexity Classification:

- Classify the following algorithms based on their time complexities:
    - Finding the maximum element in an array.
    - Matrix multiplication.
    - Binary search in a sorted array.
    - Computing the nth Fibonacci number using recursion.

Case Studies

1. Case Study: Sorting Algorithm Performance:

Objective: Compare the performance of different sorting algorithms on large datasets.

Description: Collect large datasets from various domains (e.g., numerical data, text strings). Implement and run different sorting algorithms on these datasets. Measure execution time, memory usage, and other performance metrics.

Analysis: Determine which sorting algorithm performs best under different conditions and why.

2.Case Study: Recursive vs. Iterative Approaches:

Objective: Compare the efficiency of recursive and iterative approaches for solving the same problem.

Description: Choose a problem that can be solved both recursively and iteratively (e.g., calculating factorials, Fibonacci sequence). Implement both versions and analyze their performance in terms of time complexity, space complexity, and readability.

Analysis: Discuss the trade-offs between recursion and iteration based on the results.

3.Case Study: Real-World Application of Divide-and-Conquer:

Objective: Understand the application of divide-and-conquer strategies in real-world scenarios.

Description: Investigate a real-world problem that uses a divide-and-conquer approach (e.g., database query optimization, image processing). Study how the problem is divided into subproblems, how the solutions to subproblems are combined, and the overall efficiency of the approach.

Analysis: Explain why divide-and-conquer is an effective strategy for this problem and discuss any potential improvements.

Answers for Check Your Progress

Modules	S. No.	Answers
Module 1	1.	B) To predict the performance of an algorithm as the input size grows.
	2.	C) Actual running time or space usage.
	3.	A) $O(1)$
	4.	B) Linear time complexity.
	5.	B) Linearithmic
	6.	A) Input size and implementation details.
Module 2	1.	C) Binary search
	2.	B) Two

	3.	D) Logarithmic time
	4.	C) Multiple recursion
	5.	C) It has exponential time complexity
	6.	C) By implementing memoization
	7.	D) Recurrence equations
	8.	C) To store previously computed results
	9.	C) Factorial computation
	10.	A) It requires fewer recursive calls
	11.	C) The time taken increases exponentially with input size
	12.	A) Linear recursion
	13.	C) Optimizing time complexity

Suggested Readings

1. Miller, B., & Ranum, D. (2013). Problem solving with algorithms and data structures. *URL: https://www. cs. auckland. ac. nz/... /Problem Solving with Algorithms and Data Structures. pdf (Last accessed: 30.03. 2018)*.
2. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). *Data structures and algorithms in Python*. John Wiley & Sons Ltd.
3. Sedgewick, R., & Wayne, K. (2014). *Algorithms: Part I*. Addison-Wesley Professional.

Open-Source E-Content Links

1. GeeksforGeeks – Stacks
2. GeeksforGeeks – Queues
3. GeeksforGeeks – Deques
4. Khan Academy - Stacks and Queues
5. GeeksforGeeks - Linked Lists
6. Khan Academy - Linked Lists
7. GeeksforGeeks – Trees
8. Khan Academy – Trees
9. Coursera - Data Structures

References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.
2. Allen, W. M. (2007). *Data structures and algorithm analysis in C++*. Pearson Education India..

UNIT III – INTRODUCTION

Stacks, Queues, and Deques: Stacks – Queues – Double-Ended Queues Linked. **Lists:** Singly Linked Lists-Circularly Linked Lists-Doubly Linked Lists. **Trees:** General Trees-Binary Trees- Implementing Trees-Tree Traversal Algorithms.

Stacks, Queues and Deques

UNIT OBJECTIVES

Unit III of the course aims to provide students with a comprehensive understanding of fundamental data structures like Stacks, Queues, and Deques, as well as Linked Lists and Trees. Through this unit, students will delve into the conceptual understanding, implementation techniques, and practical applications of these data structures. By the end of the unit, students will be equipped with the knowledge and skills necessary to utilize these data structures effectively in solving various computational problems and developing efficient algorithms.

SECTION 3.1: STACKS, QUEUES, AND DEQUES

Stacks, queues, and double-ended queues (deques) are fundamental data structures in computer science, each with distinct characteristics and use cases.

3.1.1 – Introduction and Definition of Management

STACKS:

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. This means that the last element added to the stack will be the first one to be removed. Stacks are used extensively in various algorithms and systems due to their simplicity and utility in managing data.

Basic Operations:

- Push: Add an element to the top of the stack.
- Pop: Remove the element from the top of the stack.
- Peek/Top: Retrieve the element at the top of the stack without removing it.
- isEmpty: Check if the stack is empty.

Applications of Stacks:

Function Call Management: The call stack in programming languages helps manage function calls and local variables.

Expression Evaluation: Stacks are used to evaluate arithmetic expressions and handle parentheses in expressions.

Backtracking: Stacks help in backtracking algorithms, such as finding paths in mazes or solving puzzles.

Undo Mechanism: Many applications use stacks to implement undo functionality, where the most recent action can be reversed.

Stack Implementation

Stacks can be implemented using arrays or linked lists. Below are implementations of stacks using both approaches.

Stack Implementation Using Array

In this implementation, the stack is represented as a list, and the end of the list represents the top of the stack.

```
class ArrayStack:
    def __init__(self):
        self.stack = []
    def push(self, value):
        self.stack.append(value)
    def pop(self):
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self.stack.pop()
    def top(self):
        if self.is_empty():
            raise IndexError("top from empty stack")
        return self.stack[-1]
    def is_empty(self):
        return len(self.stack) == 0
    def size(self):
        return len(self.stack)
```

Stack Implementation Using Linked List:

In this implementation, each node contains a value and a reference to the next node. The top of the stack is the head of the linked list.

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None
class LinkedListStack:
    def __init__(self):
        self.top = None
    def push(self, value):
        new_node = Node(value)
        new_node.next = self.top
        self.top = new_node
    def pop(self):
        if self.is_empty():
            raise IndexError("pop from empty stack")
        value = self.top.value
        self.top = self.top.next
        return value
    def top(self):
        if self.is_empty():
            raise IndexError("top from empty stack")
        return self.top.value
    def is_empty(self):
        return self.top is None
    def size(self):
        count = 0
        current = self.top
        while current:
            count += 1
            current = current.next
   return count
```

Example: Usage of Stack

Here are some practical examples demonstrating how stacks can be used in different scenarios:

BALANCED PARENTHESES

Checking if an expression has balanced parentheses using a stack.

```
def is_balanced(expression):
    stack = ArrayStack()
    for char in expression:
        if char in "({[":
            stack.push(char)
        elif char in ")}]":
            if stack.is_empty():
                return False
            top_char = stack.pop()
            if not ((top_char == '(' and char == ')') or
                    (top_char == '{' and char == '}') or
                    (top_char == '[' and char == ']')):
                return False
    return stack.is_empty()
# Example usage
```

```
print(is_balanced("(a + b) * (c + d)"))  # True
print(is_balanced("((a + b) * (c + d)"))  # False
```

EXPRESSION EVALUATION

Evaluating a postfix (Reverse Polish Notation) expression using a stack.

```
def evaluate_postfix(expression):
    stack = ArrayStack()
    for char in expression:
        if char.isdigit():
            stack.push(int(char))
        else:
            operand2 = stack.pop()
            operand1 = stack.pop()
            if char == '+':
                stack.push(operand1 + operand2)
            elif char == '-':
                stack.push(operand1 - operand2)
            elif char == '*':
                stack.push(operand1 * operand2)
            elif char == '/':
                stack.push(operand1 / operand2)
    return stack.pop()
# Example usage
print(evaluate_postfix("231*+9-"))  # (2 + (3 * 1)) - 9 = -4
```

- Stacks follow the LIFO principle.
- Common operations include push, pop, peek, and is_empty.
- Applications include function call management, expression evaluation, backtracking, and undo mechanisms.
- Stacks can be implemented using arrays or linked lists.

3.1.2 QUEUES

A queue is a linear data structure that follows the First In, First Out (FIFO) principle. This means that the first element added to the queue will be the first one to be removed. Queues are essential in various applications where the order of processing is critical.

Basic Operations:

- Enqueue: Add an element to the end of the queue.
- Dequeue: Remove the element from the front of the queue.

- Peek/Front: Retrieve the element at the front of the queue without removing it.
- isEmpty: Check if the queue is empty.

Applications of Queues:

- Task Scheduling: Managing tasks in operating systems, printers, and other scheduling systems.
- Breadth-First Search (BFS): Traversing or searching tree or graph data structures.
- Buffering: Implementing buffers in streaming and data processing.
- Producer-Consumer Problems: Coordinating producer and consumer threads in concurrent programming.

Queue Implementation:

Queues can be implemented using arrays or linked lists. Below are implementations of queues using both approaches.

Queue Implementation Using Array:

In this implementation, the queue is represented as a list. The end of the list is the rear of the queue, and the start of the list is the front of the queue.

```
class ArrayQueue:
    def __init__(self):
        self.queue = []
    def enqueue(self, value):
        self.queue.append(value)
    def dequeue(self):
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self.queue.pop(0)
    def front(self):
        if self.is_empty():
            raise IndexError("front from empty queue")
        return self.queue[0]
    def is_empty(self):
        return len(self.queue) == 0
    def size(self):
        return len(self.queue)
```

Queue Implementation Using Linked List:

In this implementation, each node contains a value and a reference to the next node. The front of the queue is the head of the linked list, and the rear of the queue is the tail of the linked list.

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None
class LinkedListQueue:
    def __init__(self):
        self.front = None
        self.rear = None
    def enqueue(self, value):
        new_node = Node(value)
        if self.rear is None:
            self.front = self.rear = new_node
        else:
            self.rear.next = new_node
            self.rear = new_node
    def dequeue(self):
        if self.front is None:
            raise IndexError("dequeue from empty queue")
        value = self.front.value
        self.front = self.front.next
        if self.front is None:
            self.rear = None
        return value
    def front(self):
        if self.front is None:
            raise IndexError("front from empty queue")
        return self.front.value
    def is_empty(self):
        return self.front is None
    def size(self):
        count = 0
        current = self.front
        while current:
            count += 1
            current = current.next
        return count
```

Example :Usage of Queue

Here are some practical examples demonstrating how queues can be used in different scenarios:

Task Scheduling:

Simulating a simple task scheduler using a queue.

```
def task_scheduler(tasks):
    queue = ArrayQueue()
    for task in tasks:
        queue.enqueue(task)
    while not queue.is_empty():
        current_task = queue.dequeue()
        print(f"Processing task: {current_task}")
# Example usage
tasks = ["task1", "task2", "task3"]
task_scheduler(tasks)
# Output:
# Processing task: task1
# Processing task: task2
# Processing task: task3
```

BREADTH-FIRST SEARCH (BFS):

Performing BFS on a graph represented using an adjacency list.

```
from collections import defaultdict, deque
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)
    def bfs(self, start_vertex):
        visited = set()
        queue = deque([start_vertex])
        visited.add(start_vertex)
        while queue:
            vertex = queue.popleft()
            print(vertex, end=" ")
            for neighbor in self.graph[vertex]:
                if neighbor not in visited:
                    queue.append(neighbor)
                    visited.add(neighbor)
# Example usage
g = Graph()
g.add_edge(0, 1)
```

```
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 0)
g.add_edge(2, 3)
g.add_edge(3, 3)
print("Breadth-First Search starting from vertex 2:")
g.bfs(2)  # Output: 2 0 3 1
```

- Queues follow the FIFO principle.
- Common operations include enqueue, dequeue, peek, and is_empty.
- Applications include task scheduling, BFS, buffering, and solving producer-consumer problems.
- Queues can be implemented using arrays or linked lists.

3.1.3 DOUBLE ENDED QUEUE

A double-ended queue (deque) is a data structure that allows insertion and deletion of elements from both ends, making it more versatile than a standard queue. Deques can function as both queues and stacks due to their ability to handle elements at both ends.

Basic Operations:

- Add First (Unshift): Add an element to the front of the deque.
- Add Last (Push): Add an element to the end of the deque.
- Remove First (Shift): Remove the element from the front of the deque.
- Remove Last (Pop): Remove the element from the end of the deque.
- Peek First: Retrieve the element at the front without removing it.
- Peek Last: Retrieve the element at the end without removing it.
- isEmpty: Check if the deque is empty.

Applications of Deques:

<u>Palindrome Checking:</u> Deques can be used to check if a string is a palindrome by comparing characters from both ends.

<u>Sliding Window Algorithms:</u> Deques efficiently support operations needed in sliding window techniques.

<u>Job Scheduling:</u> Deques can be used in systems where both enqueue and dequeue operations are required at both ends.

Deque Implementation:

Deques can be implemented using arrays or linked lists. Below is an implementation of a deque using a doubly linked list, which supports efficient operations at both ends.

Deque Implementation Using Doubly Linked List

In this implementation, each node contains a value and references to both the next and previous nodes. The deque maintains references to both the front and rear nodes.

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None
        self.prev = None
class Deque:
    def __init__(self):
        self.front = None
        self.rear = None
    def add_first(self, value):
        new_node = Node(value)
        if self.front is None:
            self.front = self.rear = new_node
        else:
            new_node.next = self.front
            self.front.prev = new_node
            self.front = new_node
    def add_last(self, value):
        new_node = Node(value)
        if self.rear is None:
            self.front = self.rear = new_node
        else:
            new_node.prev = self.rear
            self.rear.next = new_node
            self.rear = new_node
    def remove_first(self):
        if self.front is None:
            raise IndexError("remove from empty deque")
        value = self.front.value
        self.front = self.front.next
        if self.front is None:
            self.rear = None
        else:
            self.front.prev = None
        return value
    def remove_last(self):
        if self.rear is None:
            raise IndexError("remove from empty deque")
        value = self.rear.value
        self.rear = self.rear.prev
        if self.rear is None:
            self.front = None
        else:
            self.rear.next = None
        return value
    def peek_first(self):
        if self.front is None:
            raise IndexError("peek from empty deque")
        return self.front.value
    def peek_last(self):
        if self.rear is None:
            raise IndexError("peek from empty deque")
        return self.rear.value
    def is_empty(self):
        return self.front is None
    def size(self):
        count = 0
        current = self.front
        while current:
            count += 1
            current = current.next
        return count
```

Example Usage of Deque

Here are some practical examples demonstrating how deques can be used in different scenarios:

Palindrome Checking

Using a deque to check if a string is a palindrome.

```
def is_palindrome(s):
    deque = Deque()
    for char in s:
        deque.add_last(char)
    while deque.size() > 1:
        if deque.remove_first() != deque.remove_last():
            return False
    return True
```

```
# Example usage
print(is_palindrome("radar"))  # True
print(is_palindrome("hello"))  # False
```

Sliding Window Maximum

Using a deque to find the maximum in each sliding window of size k.

```
from collections import deque
def sliding_window_max(nums, k):
    if not nums:
        return []

    result = []
    dq = deque()
    for i in range(len(nums)):
        # Remove elements not within the sliding window
        if dq and dq[0] < i - k + 1:
            dq.popleft()
        # Remove smaller elements as they are not useful
        while dq and nums[dq[-1]] < nums[i]:
            dq.pop()
        dq.append(i)
        # The first element in the deque is the largest element in the window
        if i >= k - 1:
            result.append(nums[dq[0]])
    return result
# Example usage
print(sliding_window_max([1,3,-1,-3,5,3,6,7], 3))  # [3, 3, 5, 5, 6, 7]
```

- Deques support insertion and deletion at both ends.
- Common operations include add_first, add_last, remove_first, remove_last, peek_first, peek_last, and is_empty.
- Applications include palindrome checking, sliding window algorithms, and job scheduling.
- Deques can be implemented using arrays or linked lists, with doubly linked lists being efficient for operations at both ends.

Understanding deques is essential for solving problems that require flexible data structures with efficient operations at both ends. They provide a robust foundation for various algorithms and applications in computer science.

Ended Queues Linked:

A double-ended queue (deque) implemented using a linked list combines the flexibility of a deque with the dynamic memory allocation of a linked list. This allows for efficient insertions and deletions at both ends.

Characteristics of Deques:

- Double-Ended Operations: You can insert and remove elements from both the front and the rear.
- Bidirectional Traversal: Elements can be traversed from both ends.
- Deque Operations
- Add First (Unshift): Add an element to the front.
- Add Last (Push): Add an element to the end.
- Remove First (Shift): Remove an element from the front.
- Remove Last (Pop): Remove an element from the end.
- Peek First: Get the element at the front without removing it.
- Peek Last: Get the element at the end without removing it.
- isEmpty: Check if the deque is empty.

Implementation Using Doubly Linked List:

In a doubly linked list, each node contains a value, a reference to the next node, and a reference to the previous node. This allows for efficient operations at both ends of the deque.

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None
        self.prev = None
class Deque:
    def __init__(self):
        self.front = None
        self.rear = None
    def add_first(self, value):
        new_node = Node(value)
        if self.front is None:
            self.front = self.rear = new_node
        else:
            new_node.next = self.front
            self.front.prev = new_node
            self.front = new_node
```

```
def add_last(self, value):
    new_node = Node(value)
    if self.rear is None:
        self.front = self.rear = new_node
    else:
        new_node.prev = self.rear
        self.rear.next = new_node
        self.rear = new_node
def remove_first(self):
    if self.front is None:
        raise IndexError("remove from empty deque")
    value = self.front.value
    self.front = self.front.next
    if self.front is None:
        self.rear = None
    else:
        self.front.prev = None
    return value
def remove_last(self):
    if self.rear is None:
        raise IndexError("remove from empty deque")
    value = self.rear.value
    self.rear = self.rear.prev
    if self.rear is None:
        self.front = None
    else:
        self.rear.next = None
    return value
def peek_first(self):
    if self.front is None:
        raise IndexError("peek from empty deque")
    return self.front.value
def peek_last(self):
    if self.rear is None:
        raise IndexError("peek from empty deque")
    return self.rear.value
def is_empty(self):
    return self.front is None
def size(self):
    count = 0
    current = self.front
    while current:
        count += 1
```

```
            current = current.next
        return count
```

Example Usage of Deque

Here are some practical examples demonstrating how deques can be used in different scenarios:

Palindrome Checking

Using a deque to check if a string is a palindrome.

```
def is_palindrome(s):
    deque = Deque()
    for char in s:
        deque.add_last(char)

    while deque.size() > 1:
        if deque.remove_first() != deque.remove_last():
            return False
    return True
# Example usage
print(is_palindrome("radar"))  # True
print(is_palindrome("hello"))  # False
```

Sliding Window Maximum

Using a deque to find the maximum in each sliding window of size k.

```
from collections import deque
def sliding_window_max(nums, k):
    if not nums:
        return []
    result = []
    dq = deque()
    for i in range(len(nums)):
        # Remove elements not within the sliding window
        if dq and dq[0] < i - k + 1:
            dq.popleft()
        # Remove smaller elements as they are not useful
        while dq and nums[dq[-1]] < nums[i]:
            dq.pop()
        dq.append(i)
        # The first element in the deque is the largest element in the window
        if i >= k - 1:
            result.append(nums[dq[0]])
    return result
# Example usage
print(sliding_window_max([1,3,-1,-3,5,3,6,7], 3))  # [3, 3, 5, 5, 6, 7]
```

➢ Deques allow for efficient insertion and deletion at both ends.

- Common operations include add_first, add_last, remove_first, remove_last, peek_first, peek_last, and is_empty.
- Applications include palindrome checking, sliding window algorithms, and job scheduling.
- Deques can be efficiently implemented using doubly linked lists, supporting operations at both ends.

Understanding deques and their implementations using linked lists is essential for solving a variety of problems that require flexible data structures.

Let us Sum Up

Stacks, queues, and deques are fundamental data structures in computer science. A stack operates on the Last-In-First-Out (LIFO) principle, where elements are added and removed from the top. Queues, on the other hand, follow the First-In-First-Out (FIFO) principle, with elements added at the rear and removed from the front. Deques, or double-ended queues, are versatile structures that allow insertion and deletion from both ends. These data structures find wide applications in algorithm design, system implementations, and various other areas, offering efficient ways to manage data in a variety of scenarios.

Check Your Progress

1. Which data structure follows the Last-In-First-Out (LIFO) principle?
   A) Stack
   B) Queue
   C) Deque
   D) Heap
2. In a queue, where are new elements typically added?
   A) Front
   B) Rear
   C) Middle
   D) Random positions
3. Which data structure allows insertion and deletion operations at both ends?
   A) Stack
   B) Queue
   C) Deque
   D) Linked list
4. What is the operation called when an element is removed from a stack?
   A) Push
   B) Pop
   C) Peek
   D) Enqueue
5. Which of the following is NOT a valid implementation of a deque?

A) Doubly linked list
B) Array
C) Singly linked list
D) Circular buffer

6. In a stack, which operation retrieves the top element without removing it?
   A) Push
   B) Pop
   C) Peek
   D) Remove
7. Which of the following is an application of a queue?
   A) Undo mechanism in text editors
   B) Browser history
   C) Recursion
   D) Expression evaluation
8. What is the time complexity of the push operation in a stack implemented using an array?
   A) O(1)
   B) O(log n)
   C) O(n)
   D) O(n log n)
9. Which of the following is a disadvantage of using a linked list to implement a stack?
   A) Limited size
   B) No dynamic resizing
   C) Requires extra space for pointers
   D) Slower insertion and deletion
10. Which data structure combines the features of both stacks and queues?
    A) Stack
    B) Queue
    C) Deque
    D) Priority queue

SECTION 3.2: LISTS

3.2.1 – Linked Lists

A linked list is a linear data structure where elements, known as nodes, are stored in a sequence. Each node contains data and a reference (or link) to the next node in the sequence. Linked lists are dynamic and can grow or shrink in size, making them flexible for various applications.

Types of Linked Lists:

- Singly Linked List: Each node points to the next node in the sequence.
- Doubly Linked List: Each node points to both the next and the previous nodes.
- Circular Linked List: The last node points back to the first node, forming a circle.

Singly Linked List:

In a singly linked list, each node contains a value and a reference to the next node.

Operations on Singly Linked List

- Insertion: Add a node at the beginning, end, or any position in the list.
- Deletion: Remove a node from the beginning, end, or any position in the list.
- Traversal: Access each node sequentially.
- Search: Find a node with a specific value.

Implementation of Singly Linked List

```
class Node:
   def __init__(self, value):
      self.value = value
      self.next = None
class SinglyLinkedList:
   def __init__(self):
      self.head = None
   def insert_at_beginning(self, value):
      new_node = Node(value)
      new_node.next = self.head
      self.head = new_node
   def insert_at_end(self, value):
      new_node = Node(value)
      if self.head is None:
         self.head = new_node
         return
      last = self.head
      while last.next:
         last = last.next
      last.next = new_node
   def delete_node(self, key):
        temp = self.head
        if temp is not None:
            if temp.value == key:
                self.head = temp.next
                temp = None
                return
      while temp is not None:
            if temp.value == key:
                break
            prev = temp
            temp = temp.next
        if temp is None:
            return
        prev.next = temp.next
        temp = None
    def search(self, key):
        current = self.head
        while current:
            if current.value == key:
                return True
            current = current.next
        return False
    def traverse(self):
        current = self.head
        while current:
            print(current.value, end=" -> ")
            current = current.next
        print("None")
# Example usage
sll = SinglyLinkedList()
sll.insert_at_end(1)
sll.insert_at_end(2)
sll.insert_at_end(3)
sll.traverse()  # Output: 1 -> 2 -> 3 -> None
print(sll.search(2))  # Output: True
sll.delete_node(2)
sll.traverse()  # Output: 1 -> 3 -> None
```

3.2.2 DOUBLY LINKED LIST

In a doubly linked list, each node contains a value, a reference to the next node, and a reference to the previous node.

Operations on Doubly Linked List:

- Insertion: Add a node at the beginning, end, or any position in the list.
- Deletion: Remove a node from the beginning, end, or any position in the list.
- Traversal: Access each node sequentially, forwards or backwards.
- Search: Find a node with a specific value.

Implementation of Doubly Linked List

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None
        self.prev = None
class DoublyLinkedList:
    def __init__(self):
        self.head = None
    def insert_at_beginning(self, value):
        new_node = Node(value)
        new_node.next = self.head
        if self.head is not None:
            self.head.prev = new_node
        self.head = new_node
    def insert_at_end(self, value):
        new_node = Node(value)
        if self.head is None:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node
        new_node.prev = last
    def delete_node(self, key):
        temp = self.head
        while temp is not None:
            if temp.value == key:
                if temp.prev:
                    temp.prev.next = temp.next
                if temp.next:
                    temp.next.prev = temp.prev
                if temp == self.head:
                    self.head = temp.next
                temp = None
                return
            temp = temp.next
    def search(self, key):
        current = self.head
        while current:
            if current.value == key:
                return True
            current = current.next
        return False
    def traverse_forward(self):
        current = self.head
        while current:
            print(current.value, end=" -> ")
            current = current.next
        print("None")
    def traverse_backward(self):
        current = self.head
        if not current:
            return
        while current.next:
            current = current.next
        while current:
            print(current.value, end=" -> ")
            current = current.prev
        print("None")
# Example usage
dll = DoublyLinkedList()
dll.insert_at_end(1)
dll.insert_at_end(2)
dll.insert_at_end(3)
dll.traverse_forward()  # Output: 1 -> 2 -> 3 -> None
dll.traverse_backward ()  # Output: 3 -> 2 -> 1 -> None
print(dll.search(2))  # Output: True
dll.delete_node(2)
dll.traverse_forward()  # Output: 1 -> 3 -> None
```

Circular Linked List:

In a circular linked list, the last node points back to the first node, forming a circle.

Operations on Circular Linked List

- Insertion: Add a node at the beginning, end, or any position in the list.
- Deletion: Remove a node from the beginning, end, or any position in the list.

- Traversal: Access each node starting from any node.
- Search: Find a node with a specific value.

Implementation of Circular Linked List

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None
class CircularLinkedList:
    def __init__(self):
        self.head = None
    def insert_at_end(self, value):
        new_node = Node(value)
        if self.head is None:
            self.head = new_node
            new_node.next = self.head
            return
        last = self.head
        while last.next != self.head:
            last = last.next
        last.next = new_node
        new_node.next = self.head
    def delete_node(self, key):
        if self.head is None:
            return
        temp = self.head
        prev = None
        while True:
            if temp.value == key:
                if prev is None:
                    last = self.head
                    while last.next != self.head:
                        last = last.next
                    self.head = temp.next
                    last.next = self.head
                else:
                    prev.next = temp.next
                temp = None
                return
            prev = temp
            temp = temp.next
            if temp == self.head:
                break
    def search(self, key):
        if self.head is None:
            return False
        current = self.head
        while True:
            if current.value == key:
                return True
            current = current.next
            if current == self.head:
                break
        return False
    def traverse(self):
        if self.head is None:
            return
        current = self.head
        while True:
            print (current.value, end=" -> ")
            current = current.next
            if current == self.head:
                break
        print()
# Example usage
cll = CircularLinkedList()
cll.insert_at_end(1)
cll.insert_at_end(2)
cll.insert_at_end(3)
cll.traverse()  # Output: 1 -> 2 -> 3 ->
print(cll.search(2))  # Output: True
cll.delete_node(2)
cll.traverse()  # Output: 1 -> 3 ->
```

- Singly Linked Lists: Simple structure with each node pointing to the next.
- Doubly Linked Lists: Each node points to both the next and the previous nodes, allowing bidirectional traversal.
- Circular Linked Lists: The last node points back to the first node, forming a circular structure.
- Linked lists are fundamental data structures that provide flexibility and efficient dynamic memory management, making them suitable for various applications and algorithms in computer science.

Let us Sum Up

Singly linked lists, circularly linked lists, and doubly linked lists are fundamental data structures in computer science. Singly linked lists consist of nodes where each node contains a data element and a reference (or pointer) to the next node in the sequence. Circularly linked lists are similar to singly linked lists, except that the last node points back to the first node, forming a circular structure. Doubly linked lists extend singly linked lists by adding an additional pointer to each node, pointing to the previous node, enabling traversal in both forward and backward directions. These data structures are commonly used for dynamic memory allocation, implementation of data structures like stacks and queues, and in various algorithms for efficient data manipulation.

Check Your Progress

1. Which of the following data structures allows traversal only in one direction?
   A) Singly linked list
   B) Circularly linked list
   C) Doubly linked list
   D) Stack
2. In a circularly linked list, what does the last node point to?
   A) Null
   B) The first node
   C) The previous node
   D) Itself
3. Which of the following operations can be performed more efficiently in a doubly linked list compared to a singly linked list?
   A) Insertion at the beginning
   B) Deletion at the end
   C) Traversal from end to beginning
   D) Searching for an element
4. What is the time complexity for inserting a node at the beginning of a singly linked list?
   A) O(1)
   B) O(n)
   C) O(log n)
   D) O(n^2)
5. Which of the following is true about a circularly linked list?
   A) It has a fixed size.
   B) It does not have a beginning or an end.
   C) It allows traversal in both forward and backward directions.
   D) It cannot contain duplicate elements.
6. In a doubly linked list, each node contains how many pointers?

A) One
B) Two
C) Three
D) Four

7. What is the space complexity of a circularly linked list with n nodes?
   A) O(1)
   B) O(n)
   C) O(n^2)
   D) O(log n)
8. Which of the following operations can be performed in constant time (O(1)) in a doubly linked list?
   A) Insertion at the end
   B) Deletion at the beginning
   C) Traversal from beginning to end
   D) Searching for an element
9. What is the main advantage of using a circularly linked list over a singly linked list?
   A) Easier implementation
   B) Better memory utilization
   C) Faster traversal
   D) None of the above
10. In a singly linked list, what is the time complexity for accessing the kth element from the end?
    A) O(k)
    B) O(n)
    C) O(log n)
    D) O(n/k)
11. Which of the following is true about insertion and deletion operations in a doubly linked list?
    A) Both insertion and deletion can be performed in O(1) time.
    B) Insertion can be performed in O(1) time, but deletion requires O(n) time.
    C) Insertion requires O(n) time, but deletion can be performed in O(1) time.
    D) Both insertion and deletion require O(n) time.
12. In a circularly linked list, how is the end of the list identified?
    A) By a special flag set in the last node
    B) By the first node pointing to null
    C) By a sentinel node at the end
    D) By the last node pointing to the first node
13. Which of the following statements is true about the memory overhead in a doubly linked list compared to a singly linked list?
    A) Doubly linked lists have lower memory overhead due to fewer pointers.
    B) Doubly linked lists have higher memory overhead due to additional pointers.

   C) Doubly linked lists have the same memory overhead as singly linked lists.
   D) Memory overhead depends on the implementation details and cannot be generalized.

14. What is the time complexity for deleting a node at a given position in a doubly linked list?
   A) O(1)
   B) O(n)
   C) O(log n)
   D) O(n^2)
15. Which of the following operations cannot be performed efficiently in a circularly linked list?
   A) Traversal from beginning to end
   B) Insertion at the end
   C) Deletion at the beginning
   D) Traversal from end to beginning

SECTION 3.3: TREES

A tree is a data structure composed of nodes. It is used to represent hierarchical relationships, such as organizational structures, file systems, or abstract syntax trees in compilers. A tree has the following properties:

Root: The top node of the tree.

Parent and Child: Nodes connected by edges, where the parent node is one level above the child node.

Leaf: A node with no children.

Subtree: A tree consisting of a node and its descendants.

Tree Terminology

- **Node:** Each element in a tree data structure is called a node. A node contains a value or data and may also contain references (pointers or links) to other nodes.
- **Root:** The topmost node in a tree is called the root. It is the starting point of the tree.
- **Parent:** A node that has child nodes directly connected below it is called a parent node.
- **Child:** Nodes directly connected to another node when moving away from the root are called child nodes.
- **Siblings:** Nodes that share the same parent are called siblings.

- **Leaf:** Nodes that do not have any child nodes are called leaf nodes or terminal nodes.
- **Internal Node:** Any node of a tree that has at least one child node is called an internal node.
- **Depth:** The depth of a node in a tree is the length of the path from the root to that node. The root node has a depth of 0.
- **Height:** The height of a tree is the length of the longest path from the root to a leaf node. Alternatively, it can be defined as the maximum depth of any node in the tree.
- **Subtree:** A subtree of a tree is a tree formed by selecting a node and all its descendants, including the node itself.
- **Binary Tree:** A binary tree is a tree data structure in which each node has at most two children, referred to as the left child and the right child.
- **Binary Search Tree (BST):** A binary search tree is a binary tree in which the value of each node in the left subtree is less than or equal to the value of the node, and the value of each node in the right subtree is greater than the value of the node.
- **Traversal:** Traversal refers to visiting all the nodes in a tree in a specific order. Common tree traversal methods include in-order, pre-order, post-order, and level-order traversal.
- **Balanced Tree:** A balanced tree is a tree in which the height of the left and right subtrees of any node differ by at most one.
- **Complete Binary Tree:** A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
- **Full Binary Tree:** A full binary tree is a binary tree in which every node other than the leaves has two children.
-

3.3.1 – Tree Abstract Datatype

The Tree Abstract Data Type (ADT) is introduced. The Tree ADT defines a hierarchical structure composed of nodes. Each node contains a value and a set of references (pointers or links) to other nodes. Unlike linear data structures like arrays or linked lists, trees organize data in a hierarchical manner, with a single root node at the top and child nodes branching out from it. The Tree ADT provides operations for creating, manipulating, and traversing trees.

Computing Depth and Height:

The depth of a node in a tree is the length of the path from the root to that node. It is typically measured by counting the number of edges or levels traversed from the root to the node. The height of a tree is the length of the longest path from the root to a leaf node. Alternatively, it can be defined as the maximum depth of any node in the tree.

3.3.2 –General Trees

In a general tree, each node can have an arbitrary number of children. There are no restrictions on the number of children each node can have, making it versatile for various applications.

Properties:

- Degree of a node: The number of children a node has.
- Depth/Level: The distance from the root to the node.
- Height of a tree: The length of the longest path from the root to a leaf.

Binary Trees:

A binary tree is a specialized form of a tree in which each node has at most two children, commonly referred to as the left child and the right child.

Types of Binary Trees:

- Full Binary Tree: Every node other than the leaves has two children.
- Complete Binary Tree: All levels are completely filled except possibly the last level, which is filled from left to right.
- Perfect Binary Tree: All internal nodes have two children and all leaves are at the same level.
- Balanced Binary Tree: The height of the left and right subtrees of any node differ by at most one.

- Binary Search Tree (BST): A binary tree where the left subtree contains only nodes with values less than the parent node, and the right subtree contains only nodes with values greater than the parent node.

Linked Structure for Binary Trees:

In a linked structure, each node contains a data element and references (pointers or links) to its left child and right child nodes. The linked structure for binary trees allows for dynamic memory allocation and flexible tree manipulation. It typically involves defining a node structure with data and pointers to left and right children. Operations such as insertion, deletion, traversal, and searching are implemented using recursive or iterative algorithms based on the linked structure.

Array-Based Representation of a Binary Tree:

In the array-based representation, the binary tree is stored in a one-dimensional array. The relationship between parent and child nodes is determined based on the position of elements in the array. Specifically, for a node at index i in the array, its left child is stored at index 2i+1, and its right child is stored at index 2i+2. This representation offers efficient memory usage and faster access to elements compared to linked structures, especially for complete binary trees.

Linked Structure for General Trees:

Unlike binary trees, general trees allow nodes to have any number of children. In a linked structure for general trees, each node contains a data element and references to its children nodes, typically stored in a linked list or an array. The linked structure for general trees supports various operations such as insertion, deletion, traversal, and searching, with algorithms tailored to handle the variable number of children for each node.

```
class TreeNode:
   def __init__(self, data):
      self.data = data
      self.left = None
      self.right = None
class BinaryTree:
   def __init__(self):
      self.root = None
   def insert(self, data):
      if self.root is None:
         self.root = TreeNode(data)
```

```
        else:
            self._insert_recursive(self.root, data)
    def _insert_recursive(self, node, data):
        if data < node.data:
            if node.left is None:
                node.left = TreeNode(data)
            else:
                self._insert_recursive(node.left, data)
        elif data > node.data:
            if node.right is None:
                node.right = TreeNode(data)
            else:
                self._insert_recursive(node.right, data)
    def inorder_traversal(self, node):
        if node:
            self.inorder_traversal(node.left)
            print(node.data, end=" ")
            self.inorder_traversal(node.right)
# Example usage:
tree = BinaryTree()
tree.insert(5)
tree.insert(3)
tree.insert(7)
tree.insert(2)
tree.insert(4)
tree.insert(6)
tree.insert(8)
print("Inorder Traversal:")
tree.inorder_traversal(tree.root)
```

3.3.3 IMPLEMENTING TREES

General Tree Implementation

A node in a general tree can be implemented using a class in various programming languages. Here's an example in Python:

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []
    def add_child(self, child_node):
        self.children.append(child_node)
# Usage
root = TreeNode(1)
```

```
child1 = TreeNode(2)
child2 = TreeNode(3)
root.add_child(child1)
root.add_child(child2)
```

Binary Tree Implementation

A binary tree node can also be implemented using a class. Here's an example in Python:

```
class BinaryTreeNode:
   def __init__(self, value):
      self.value = value
      self.left = None
      self.right = None
   def insert_left(self, child_value):
      self.left = BinaryTreeNode(child_value)
      return self.left
   def insert_right(self, child_value):
      self.right = BinaryTreeNode(child_value)
      return self.right
# Usage
root = BinaryTreeNode(1)
left_child = root.insert_left(2)
right_child = root.insert_right(3)
```

3.3.4 – TREE TRAVERSAL ALGORITHM

Tree traversal refers to the process of visiting each node in a tree in a systematic way. There are several traversal methods:

Preorder Traversal:

- ✓ Preorder traversal is a tree traversal algorithm that visits nodes in a specific order.
- ✓ In preorder traversal, the algorithm starts at the root node and recursively visits each node in the following order:
  - Visit the current node.
  - Recursively traverse the left subtree.
  - Recursively traverse the right subtree.
- ✓ This traversal method is called "preorder" because it visits the current node before traversing its children nodes.

- ✓ Preorder traversal is useful for various tree-related tasks such as creating a prefix expression from an expression tree, copying a tree, or constructing a tree from its preorder traversal.

Postorder Traversal:

- ✓ Postorder traversal is another tree traversal algorithm that visits nodes in a specific order.
- ✓ In postorder traversal, the algorithm recursively visits each node in the following order:
  - o Recursively traverse the left subtree.
  - o Recursively traverse the right subtree.
  - o Visit the current node.
- ✓ This traversal method is called "postorder" because it visits the current node after traversing its children nodes.
- ✓ Postorder traversal is useful for tasks such as deleting a tree, evaluating postfix expressions stored in expression trees, or producing a Reverse Polish Notation (RPN) from an expression tree.

Inorder traversal

Inorder traversal is a tree traversal algorithm that visits nodes in a specific order:

- ✓ Visit the Left Subtree: In inorder traversal, the algorithm first recursively traverses the left subtree of the current node.
- ✓ Visit the Current Node: After visiting all nodes in the left subtree, the algorithm visits the current node.
- ✓ Visit the Right Subtree: Finally, the algorithm recursively traverses the right subtree of the current node.

The order of visiting nodes in inorder traversal is as follows: left subtree, current node, right subtree

Depth-First Traversal (DFS)

Preorder Traversal: Visit the root node, then recursively visit the left subtree, followed by the right subtree.

```
def preorder(node):
    if node:
            print(node.value)
            preorder(node.left)
            preorder(node.right)
```

Inorder Traversal: Recursively visit the left subtree, visit the root node, then recursively visit the right subtree.

```
def inorder(node):
    if node:
        inorder(node.left)
        print(node.value)
        inorder(node.right)
```

Postorder Traversal: Recursively visit the left subtree, then the right subtree, and finally visit the root node.

```
def postorder(node):
    if node:
        postorder(node.left)
        postorder(node.right)
        print(node.value)
```

Breadth-First Traversal (BFS)

Also known as Level Order Traversal, it visits nodes level by level from left to right.

```
from collections import deque
def level_order(root):
    if not root:
        return
    queue = deque([root])
    while queue:
        node = queue.popleft()
        print(node.value)
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
```

Trees are fundamental data structures that support hierarchical data representation. General trees allow for multiple children per node, while binary trees restrict each node to at most two children.

Implementing trees involves creating nodes and connecting them appropriately. Traversal algorithms such as DFS (preorder, inorder, postorder) and BFS (level order) are essential for accessing and manipulating tree data.

Implementation Preorder Traversal:

```
class TreeNode:
    def __init__(self, value):
        self.val = value
        self.left = None
        self.right = None
def preorder_traversal(root):
    if root:
        print(root.val, end=" ")
        preorder_traversal(root.left)
        preorder_traversal(root.right)
# Example usage:
# Construct a binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
# Preorder traversal
print("Preorder Traversal:")
preorder_traversal(root)
```

Postorder Traversal:

```
def postorder_traversal(root):
    if root:
        postorder_traversal(root.left)
        postorder_traversal(root.right)
        print(root.val, end=" ")
# Example usage:
# Using the same tree as above
# Postorder traversal
print("\nPostorder Traversal:")
postorder_traversal(root)
```

Breadth-First Traversal:

```
from collections import deque
def breadth_first_traversal(root):
    if not root:
        return
    queue = deque([root])
    while queue:
        node = queue.popleft()
        print(node.val, end=" ")
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
# Example usage:
# Using the same tree as above
# Breadth-first traversal
print("\nBreadth-First Traversal:")
breadth_first_traversal(root)
```

Inorder Traversal:

```
def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.val, end=" ")
        inorder_traversal(root.right)
# Example usage:
# Using the same tree as above
# Inorder traversal
print("\nInorder Traversal:")
inorder_traversal(root)
```

Applications of Tree Traversals

1. **Table of Contents Generation:**
   - ✓ Preorder traversal of a tree can be used to generate a table of contents for a document.
   - ✓ This involves visiting each node in the tree and printing its content in a specified format.
   - ✓ Indentation based on the depth of the node within the tree can improve the presentation of the table of contents.
2. **Efficient Tree Traversal for Table of Contents:**
   - ✓ Preorder traversal with indentation can be efficiently implemented using a top-down recursion approach.
   - ✓ This approach includes the current depth as an additional parameter, avoiding the inefficiency of calculating depth at each step.
3. **Labeling Tree Elements:**
   - ✓ Preorder traversal can be used to display tree structures with explicit numbering, even if the numbering is not present in the tree itself.
   - ✓ This involves assigning labels to tree elements based on their positions within the tree.

4. **Parenthetic Representations of Trees:**
   - ✓ The parenthetic string representation of a tree provides a concise and computer-friendly way to represent its structure.
   - ✓ It is essentially a preorder traversal with additional punctuation to denote tree structure.
5. **Computing Disk Space:**
   - ✓ Postorder traversal is suitable for computing total disk space used by a directory structure represented as a tree.
   - ✓ Each level of recursion provides a return value to the parent caller, allowing for an efficient computation of disk space.

Tree traversals are fundamental operations in various domains and are used for a wide range of tasks, including:

1. Searching: Tree traversal algorithms are used in searching algorithms such as depth-first search (DFS) and breadth-first search (BFS) to explore and find elements in a tree data structure efficiently.
2. Sorting: Tree traversal algorithms, particularly inorder traversal, are used in sorting algorithms such as binary tree sort, where elements are inserted into a binary search tree and then traversed in sorted order.
3. Expression Evaluation: Tree traversals are used in evaluating mathematical expressions stored in expression trees. For example, inorder traversal of an expression tree can be used to convert an infix expression into postfix or prefix notation, which can then be evaluated using stack-based algorithms.
4. Syntax Tree Construction: Tree traversal algorithms are used in constructing syntax trees for parsing and analyzing programming languages. Traversals help in building the hierarchical structure of the syntax tree by visiting and processing nodes in a specific order.
5. Network Routing Algorithms: Tree traversal algorithms are used in network routing algorithms to discover and maintain paths between nodes in a network topology. Traversals help in exploring the network graph and finding the most efficient routes for data transmission.

Euler Tours:

An Euler tour is a systematic way of traversing a tree or graph that visits every edge exactly once. An Euler tour involves traversing the tree in a recursive manner, visiting each edge twice: once when descending into a subtree and once when returning from it. Euler tours are useful for various tree-related tasks, such as computing subtree sums, finding LCA (Lowest Common Ancestor) of two nodes, and detecting cycles in undirected graphs. The specific implementations of preorder, postorder, and inorder traversals in the Tree and BinaryTree classes are not always sufficient for capturing the range of computations needed. Custom implementations were developed for various applications, but this approach lacks adaptability and reusability.

Euler Tour Traversal:

- ✓ Euler tour traversal of a general tree involves a "walk" around the tree, starting from the root and proceeding toward its leftmost child while keeping the edges to the left.
- ✓ The traversal progresses exactly two times along each edge of the tree, once downward and later upward.
- ✓ Euler tour traversal unifies the concepts of preorder and postorder traversals by defining "pre visit" and "post visit" actions for each position in the tree.

Recursive Nature of Euler Tour:

- ✓ The Euler tour traversal process can be viewed recursively, with each position having a "pre visit" and a "post visit" action.
- ✓ Between the "pre visit" and "post visit" of a position, there is a recursive tour of each of its subtrees.

Pseudo-code for Euler Tour Traversal:

- ✓ The pseudo-code for performing an Euler tour traversal of a subtree rooted at a position p is provided.
- ✓ It involves performing the "pre visit" action for the current position, recursively touring each child subtree, and then performing the "post visit" action for the current position.

Template Method Pattern:

The Template Method Pattern is a design pattern used in object-oriented programming to define the skeleton of an algorithm in a superclass, while allowing subclasses to provide specific implementations for certain steps of the algorithm.

The Template Method Pattern can be used to encapsulate the traversal algorithm's structure in a base class, with abstract methods representing the steps that can vary between different traversals. Subclasses can then override these abstract methods to customize the traversal behavior for specific types of trees or traversal orders. This pattern promotes code reusability and flexibility, as it separates the overall algorithm structure from the specific implementations of its steps.

- The template method pattern describes a generic computation mechanism that can be specialized for a particular application by redefining certain steps.
- It involves defining a primary algorithm that calls auxiliary functions, known as hooks, at designated steps of the process to allow customization.

2. **Hooks in Euler Tour Traversal:**
    - In the context of an Euler tour traversal, two separate hooks are defined: a pre-visit hook called before traversing subtrees and a post-visit hook called after completing subtree traversals.
    - These hooks provide opportunities for customization and specialization of the traversal process.
3. **Python Implementation of EulerTour class:**
    - An EulerTour class is provided as a base class for performing Euler tour traversals of a tree.
    - The primary recursive process is defined in the non-public **tour** method, which performs the tour of a subtree rooted at a given position.
    - The EulerTour class defines trivial definitions for pre-visit and post-visit hooks, which can be overridden by subclasses to provide specialized behavior.
4. **Hooks in EulerTour class:**
    - The **hook_previsit** and **hook_postvisit** methods are defined in the EulerTour class and can be overridden by subclasses.
    - The **hook_previsit** method is called once for each position before its subtrees are traversed, while the **hook_postvisit** method is called once for each position after its subtrees are traversed.

- These methods receive parameters such as the position, depth, path, and results from subtree traversals, allowing for flexible customization.

```
class PreorderPrintIndentedTour(EulerTour):
  def _hook_previsit(self, p, d, path):
    print(2*d*' ' + str(p.element()))
```

Code Fragment 8.29: A subclass of EulerTour that produces an indented preorder list of a tree's elements.

```
class PreorderPrintIndentedLabeledTour(EulerTour):
  def _hook_previsit(self, p, d, path):
    label = '.'.join(str(j+1) for j in path)    # labels are one-indexed
    print(2*d*' ' + label, p.element())
```

Code Fragment 8.30: A subclass of EulerTour that produces a labeled and indented, preorder list of a tree's elements.

```
class ParenthesizeTour(EulerTour):
  def _hook_previsit(self, p, d, path):
    if path and path[-1] > 0:                # p follows a sibling
      print(', ', end='')                    # so preface with comma
    print(p.element(), end='')               # then print element
    if not self.tree().is_leaf(p):           # if p has children
      print(' (', end='')                    # print opening parenthesis

  def _hook_postvisit(self, p, d, path, results):
    if not self.tree().is_leaf(p):           # if p has children
      print(')', end='')                     # print closing parenthesis
```

Code Fragment 8.31: A subclass of EulerTour that prints a parenthetic string representation of a tree.

```
class DiskSpaceTour(EulerTour):
  def _hook_postvisit(self, p, d, path, results):
    # we simply add space associated with p to that of its subtrees
    return p.element().space() + sum(results)
```

Code Fragment 8.32: A subclass of EulerTour that computes disk space for a tree.

5. **Examples of Custom Subclasses:**
   - Examples of custom subclasses of EulerTour are provided to demonstrate various customizations of the traversal process.
   - These subclasses override the pre-visit and post-visit hooks to produce different outputs or perform specific computations during traversal.

- Examples include generating indented preorder lists, labeled and indented preorder lists, parenthetic string representations, and computing disk space for a tree.

```
class EulerTour:
  """Abstract base class for performing Euler tour of a tree.

  hook_previsit and hook_postvisit may be overridden by subclasses.
  """
  def __init__(self, tree):
    """Prepare an Euler tour template for given tree."""
    self.tree = tree
  def tree(self):
    """Return reference to the tree being traversed."""
    return self.tree
  def execute(self):
    """Perform the tour and return any result from post visit of root."""
    if len(self.tree) > 0:
      return self.tour(self.tree.root(), 0, [])  # start the recursion
  def tour(self, p, d, path):
    """Perform tour of subtree rooted at Position p.

    p       Position of current node being visited
    d       Depth of p in the tree
    path    List of indices of children on path from root to p
    """
    self.hook_previsit(p, d, path)  # "pre visit" p
    results = []
    path.append(0)  # add new index to end of path before recursion
    for c in self.tree.children(p):
      results.append(self.tour(c, d + 1, path))  # recur on child's subtree
      path[-1] += 1  # increment index
    path.pop()  # remove extraneous index from end of path
    answer = self.hook_postvisit(p, d, path, results)  # "post visit" p
    return answer
  def hook_previsit(self, p, d, path):  # can be overridden
    pass

  def hook_postvisit(self, p, d, path, results):  # can be overridden
    pass
```

Let Us Sum All

Trees provide hierarchical structures crucial in computer science. General trees allow any number of children per node, while binary trees restrict to two children per node. Implementations include linked structures or array-based representations. Traversing trees involves various algorithms such as preorder, inorder, postorder, and breadth-first traversal, crucial for data retrieval and manipulation. Implementing these traversal algorithms efficiently enhances tree processing, aiding applications like expression evaluation, parsing, and document structure analysis.

Check Your Progress

1. What type of trees allows any number of children per node?
   A) General Trees
   B) Binary Trees
   C) Red-Black Trees
   D) AVL Trees
2. Which tree restricts to two children per node?
   A) General Trees
   B) Binary Trees
   C) B-Trees
   D) Trie Trees
3. Which is not a common implementation of trees?
   A) Linked Structures
   B) Array-Based Representations
   C) Hash Tables
   D) Adjacency Lists
4. Which traversal algorithm starts from the root, then visits the left subtree, and finally the right subtree?
   A) Preorder Traversal
   B) Inorder Traversal
   C) Postorder Traversal
   D) Breadth-First Traversal
5. In which traversal algorithm are the nodes visited in the order: left subtree, root, right subtree?
   A) Preorder Traversal
   B) Inorder Traversal
   C) Postorder Traversal
   D) Breadth-First Traversal
6. Which traversal algorithm visits the left subtree, then the right subtree, and finally the root?
   A) Preorder Traversal
   B) Inorder Traversal
   C) Postorder Traversal
   D) Breadth-First Traversal
7. Which traversal algorithm visits nodes level by level?

   A) Preorder Traversal
   B) Inorder Traversal
   C) Postorder Traversal
   D) Breadth-First Traversal
8. Which data structure is commonly used to implement breadth-first traversal?
   A) Stack
   B) Queue
   C) Heap
   D) Deque
9. Which traversal algorithm is commonly used to evaluate mathematical expressions?
   A) Preorder Traversal
   B) Inorder Traversal
   C) Postorder Traversal
   D) Breadth-First Traversal
10. Which traversal algorithm is commonly used to create a sorted list of elements in a binary search tree?
   A) Preorder Traversal
   B) Inorder Traversal
   C) Postorder Traversal
   D) Breadth-First Traversal

Let Us Sum Up

In Unit III, we delved into essential data structures crucial for efficient algorithm design. Stacks, queues, and deques offer distinct methods of organizing data with specific insertion and deletion patterns. Linked lists, including singly, circularly, and doubly linked lists, provide dynamic storage with flexible node connections. Trees, encompassing general and binary trees, facilitate hierarchical data representation. Implementing trees involves diverse underlying structures, and tree traversal algorithms enable systematic exploration of tree nodes. Mastery of these structures equips programmers with powerful tools for solving diverse computational challenges.

Unit Summary

Unit III covers fundamental data structures such as Stacks, Queues, Deques, Linked Lists, and Trees. Stacks follow the Last In, First Out (LIFO) principle, while Queues adhere to the First In, First Out (FIFO) principle. Deques support insertion and deletion at both ends. Linked Lists come in various forms, including Singly Linked Lists, Circularly Linked Lists, and Doubly Linked Lists, offering dynamic storage with different node connections. Trees represent hierarchical structures, with General Trees allowing any number of children per node and Binary Trees restricting nodes to two children. Implementing Trees involves various data structures, and Tree Traversal

Algorithms enable systematic exploration of tree nodes in different orders, such as in-order, pre-order, post-order, and level-order traversals. Understanding these structures is crucial for developing efficient algorithms and solving a wide range of computational problems.

Glossary

- **Stacks:** A data structure that follows the Last In, First Out (LIFO) principle, where elements are inserted and removed from the same end.
- **Queues:** A data structure that follows the First In, First Out (FIFO) principle, where elements are inserted at one end (rear) and removed from the other end (front).
- **Deques (Double-Ended Queues):** A data structure that supports insertion and deletion at both ends.
- **Linked Lists:** A linear data structure consisting of a sequence of elements where each element points to the next one, forming a chain-like structure.
- **Singly Linked Lists:** Each element points to the next element in the sequence.
- **Circularly Linked Lists:** The last element points back to the first element, forming a circle.
- **Doubly Linked Lists:** Each element has two pointers, one pointing to the next element and one to the previous element.
- **Trees:** A hierarchical data structure composed of nodes, where each node has a value and a list of references to its children.
- **General Trees:** Trees where each node can have any number of children.
- **Binary Trees:** Trees where each node has at most two children.
- **Tree Traversal:** The process of visiting each node in a tree in a specific order.
- **In-order Traversal:** Visit the left subtree, then the root, then the right subtree.
- **Pre-order Traversal:** Visit the root, then the left subtree, then the right subtree.
- **Post-order Traversal:** Visit the left subtree, then the right subtree, then the root.
- **Level-order Traversal:** Visit nodes level by level, from left to right.
- **Implementing Trees**: Creating and managing tree data structures using various methods, such as arrays, linked lists, or structures/classes.

Self – Assessment Questions

1. Evaluate the time complexity of the breadth-first search (BFS) algorithm on an adjacency list representation of a graph. How does it differ from the time complexity of BFS on an adjacency matrix representation?
2. Illustrate the process of inserting a new node into a doubly linked list. How does the structure of the list change before and after the insertion?
3. Evaluate the efficiency of using a deque data structure for implementing a double-ended queue compared to using two stacks. What are the performance implications of each approach?
4. Analyze the time complexity of the depth-first search (DFS) algorithm on a graph. How does the presence of cycles in the graph affect the algorithm's performance?

Activities / Exercises / Case Studies

Activities

1. Implementation Practice: Implement basic operations (push, pop, enqueue, dequeue) for stacks, queues, and deques in your preferred programming language.
2. Linked List Construction: Create classes or structures to implement singly linked lists, circularly linked lists, and doubly linked lists. Test their functionalities with various operations.
3. Tree Construction: Build classes or structures for general trees and binary trees. Implement insertion, deletion, and traversal algorithms (in-order, pre-order, post-order, level-order).
4. Algorithm Analysis: Analyze the time and space complexity of various tree traversal algorithms and compare them with different input sizes.
5. Recursion Practice: Solve problems using recursion, such as factorial calculation, Fibonacci sequence generation, or binary tree traversal.

Exercises:

1. Write a function to check if a given binary tree is a binary search tree (BST).
2. Implement an algorithm to reverse a linked list in both iterative and recursive approaches.
3. Write code to perform a level-order traversal of a binary tree using a queue.
4. Implement a depth-first search (DFS) algorithm to traverse a graph represented as an adjacency list.

Case Studies:

1. Memory Management: Explore how memory allocation and deallocation are managed in data structures like linked lists and trees. Compare the memory usage of arrays and linked lists in different scenarios.
2. Performance Optimization: Investigate how different implementations of stacks, queues, and deques impact performance in real-world applications. Analyze the trade-offs between space and time complexity.
3. Application in Graph Algorithms: Study how stacks and queues are utilized in graph algorithms like DFS, BFS, and topological sorting. Analyze their efficiency in solving graph-related problems.
4. Concurrency and Synchronization: Examine how stacks and queues are used in concurrent programming environments to manage shared resources. Discuss synchronization techniques and their implications.
5. Data Structures in Operating Systems: Investigate how operating systems utilize data structures like queues and trees for process scheduling, memory management, and file systems. Analyze their role in enhancing system performance and reliability.

Answers For Check Your Progress

Modules	S. No.	Answers
Module 1	1.	A) Stack
	2.	B) Rear
	3.	C) Deque
	4.	B) Pop
	5.	C) Singly linked list
	6.	C) Peek
	7.	B) Browser history
	8.	A) O(1)
	9.	C) Requires extra space for pointers
	10.	C) Deque
Module 2	1.	A) Singly linked list
	2.	B) The first node
	3.	C) Traversal from end to beginning

	4.	A) O(1)
	5.	B) It does not have a beginning or an end.
	6.	B) Two
	7.	A) O(1)
	8.	A) Insertion at the end
	9.	B) Better memory utilization
	10.	B) O(n)
	11.	D) Both insertion and deletion require O(n) time.
	12.	D) By the last node pointing to the first node
	13.	B) Doubly linked lists have higher memory overhead due to additional pointers.
	14.	B) O(n)
	15.	D) Traversal from end to beginning
Module 3	**1.**	A) General Trees
	2.	B) Binary Trees
	3.	C) Hash Tables
	4.	A) Preorder Traversal
	5.	B) Inorder Traversal
	6.	C) Postorder Traversal
	7.	D) Breadth-First Traversal
	8.	B) Queue
	9.	C) Postorder Traversal
	10.	B) Inorder Traversal

Suggested Readings

1. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). *Data structures and algorithms in Python*. John Wiley & Sons Ltd.
2. Sedgewick, R., & Wayne, K. (2014). *Algorithms: Part I*. Addison-Wesley Professional.
3. Miller, B., & Ranum, D. (2013). Problem solving with algorithms and data

structures. *URL: https://www. cs. auckland. ac. nz/.../ProblemSolvingwith AlgorithmsandDataStructures. pdf (Last accessed: 30.03. 2018)*.

Open-Source E-Content Links

1. GeeksforGeeks – Stacks
2. GeeksforGeeks – Queues
3. GeeksforGeeks – Deques
4. Khan Academy - Stacks and Queues
5. GeeksforGeeks - Linked Lists
6. Khan Academy - Linked Lists
7. GeeksforGeeks – Trees
8. Khan Academy – Trees
9. Coursera - Data Structures

References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.

UNIT IV – INTRODUCTION

Priority Queues: Priority Queue Abstract Data Type- Implementing a Priority Queue – Heaps- Sorting with a Priority Queue. **Maps, Hash Tables, and Skip Lists:** Maps and Dictionaries-Hash Tables- Sorted Maps-Skip Lists-Sets, Multi Sets, and Multi Maps.

Priority Queues

UNIT OBJECTIVE

The course aims to provide a comprehensive understanding of fundamental data structures and their applications. Students will explore priority queues, learning how to implement them efficiently using heaps and integrate them into sorting algorithms. Additionally, they will delve into the realm of maps, hash tables, skip lists, sets, and multi-maps, gaining insight into their design, implementation, and practical usage. By the end of the course, learners will be equipped with the knowledge and skills to effectively utilize these data structures to solve complex computational problems and optimize algorithmic performance.

SECTION 4.1: PRIORITY QUEUES

4.1. 1 – Introducing the Priority Queue ADT

Introducing the Priority Queue ADT

A priority queue is an abstract data type that allows elements to be added arbitrarily but always removes the element with the highest priority (typically the smallest key value). Here are the main operations supported by a priority queue ADT:

- P.add(k, v): Insert an item with key k and value v into the priority queue P.
- P.min(): Return a tuple (k, v) representing the key and value of an item in P with the minimum key without removing it. If the queue is empty, an error occurs.
- P.remove_min(): Remove and return the item with the minimum key from P as a tuple (k, v). If the queue is empty, an error occurs.
- P.is_empty(): Return True if the priority queue P is empty.
- len(P): Return the number of items in the priority queue P.

A priority queue can contain multiple entries with equivalent keys. In such cases, the min and remove_min methods may return any of the items with the minimum key. Additionally, while keys are usually numeric, any Python object that supports comparison can be used as a key.

Example Operations

Consider an initially empty priority queue P. Below is a series of operations and their effects:

Operation	Return Value	Priority Queue
P.add(5, A)		{(5, A)}
P.add(9, C)		{(5, A), (9, C)}
P.add(3, B)		{(3, B), (5, A), (9, C)}
P.add(7, D)		{(3, B), (5, A), (7, D), (9, C)}
P.min()	(3, B)	{(3, B), (5, A), (7, D), (9, C)}
P.remove_min()	(3, B)	{(5, A), (7, D), (9, C)}
P.remove_min()	(5, A)	{(7, D), (9, C)}
len(P)	2	{(7, D), (9, C)}
P.remove_min()	(7, D)	{(9, C)}
P.remove_min()	(9, C)	{}
P.is_empty()	True	{}
P.remove_min()	error	{}

4.1.2 –Implementing a Priority Queue

Implementing a Priority Queue

` To implement a priority queue, we can store its entries in a positional list, keeping entries sorted by key. A key challenge is to manage both the element and its key, even as items move within the data structure. We use the composition design pattern to store items internally as pairs of key k and value v.

The PriorityQueueBase Class

Below is an implementation of the PriorityQueueBase class, which includes a nested Item class to manage key-value pairs. The class also provides a concrete implementation of the is_empty method.

```
class PriorityQueueBase:
    """Abstract base class for a priority queue."""
    class Item:
        """Lightweight composite to store priority queue items."""
        __slots__ = '_key', '_value'
        def __init__(self, k, v):
            self._key = k
            self._value = v
        def __lt__(self, other):
            return self._key < other._key  # Compare items based on their
keys
    def is_empty(self):  # Concrete method assuming abstract __len__
        """Return True if the priority queue is empty."""
        return len(self) == 0
```

This base class provides the foundation for implementing specific priority queue data structures, ensuring that each element remains paired with its key and value, allowing for consistent priority management.

Unsorted List Priority Queue Implementation

In this implementation, the priority queue stores entries in an unsorted list. Each entry is a key-value pair, represented as instances of the Item class. These items are stored in a PositionalList, which is implemented using a doubly-linked list, allowing for constant-time operations on the list itself.

Class Definition and Initialization

Here's the UnsortedPriorityQueue class, inheriting from the PriorityQueueBase class:

```
class UnsortedPriorityQueue(PriorityQueueBase):  # base class defines Item
    """A min-oriented priority queue implemented with an unsorted list."""
    def __init__(self):
        """Create a new empty Priority Queue."""
        self._data = PositionalList()
    def __len__(self):
        """Return the number of items in the priority queue."""
        return len(self._data)
    def add(self, key, value):
        """Add a key-value pair."""
        self._data.add_last(self.Item(key, value))
    def min(self):
        """Return but do not remove (k,v) tuple with minimum key."""
        p = self._find_min()
        item = p.element()
        return (item._key, item._value)
    def remove_min(self):
        """Remove and return (k,v) tuple with minimum key."""
```

```
            p = self._find_min()
            item = self._data.delete(p)
            return (item._key, item._value)
    def _find_min(self):
        """Return Position of item with minimum key."""
        if self.is_empty():  # is_empty inherited from base class
            raise Empty("Priority queue is empty")
        small = self._data.first()
        walk = self._data.after(small)
        while walk is not None:
            if walk.element() < small.element():
                small = walk
            walk = self._data.after(walk)
        return small
```

Internal PositionalList Class

The PositionalList class, which supports O(1) time operations for adding and deleting elements, should be implemented as a doubly-linked list. Below is an outline of the PositionalList class:

```
class PositionalList:
    """A sequential container of elements allowing positional access."""
    class _Node:
        __slots__ = '_element', '_prev', '_next'

        def __init__(self, element, prev, next):
            self._element = element
            self._prev = prev
            self._next = next
    class Position:
        """An abstraction representing the location of a single element.""
        def __init__(self, container, node):
            self._container = container
            self._node = node
        def element(self):
            return self._node._element
    def __init__(self):
        self._header = self._Node(None, None, None)
        self._trailer = self._Node(None, self._header, None)
        self._header._next = self._trailer
        self._size = 0
    def __len__(self):
        return self._size
    def is_empty(self):
        return self._size == 0
    def _validate(self, p):
        if not isinstance(p, self.Position):
            raise ValueError("p must be proper Position type")
        if p._container is not self:
            raise ValueError("p does not belong to this container")
        if p._node._next is None:
            raise ValueError("p is no longer valid")
        return p._node
    def _make_position(self, node):
        if node is self._header or node is self._trailer:
            return None
        else:
            return self.Position(self, node)
    def first(self):
        return self._make_position(self._header._next)
    def after(self, p):
        node = self._validate(p)
        return self._make_position(node._next)
       def add_last(self, e):
        return self._insert_between(e, self._trailer._prev, self._trailer)
       def delete(self, p):
        original = self._validate(p)
        return self._delete_node(original)
        def _insert_between(self, e, predecessor, successor):
        node = self._Node(e, predecessor, successor)
        predecessor._next = node
        successor._prev = node
        self._size += 1
        return self._make_position(node)
        def _delete_node(self, node):
        element = node._element
        node._prev._next = node._next
        node._next._prev = node._prev
        self._size -= 1
        node._prev = node._next = node._element = None
        return element
```

Summary of Running Times

__len__: O(1)

is_empty: O(1)

add: O(1)

min: O(n)

remove_min: O(n)

Explanation

- Initialization (__init__): Creates an empty priority queue using an empty PositionalList.
- Length (__len__): Returns the number of items in the queue by returning the length of the PositionalList.
- Add (add): Adds a new item to the end of the PositionalList, which takes constant time.
- Find Minimum (_find_min): Iterates through the list to find the item with the minimum key, which takes linear time.
- Minimum (min): Calls _find_min to get the position of the item with the minimum key and returns it without removing it.
- Remove Minimum (remove_min): Calls _find_min to get the position of the item with the minimum key and removes it from the list, which also takes linear time.
- By using an unsorted list, the UnsortedPriorityQueue implementation provides constant-time addition but requires linear time to find and remove the minimum element. This trade-off is suitable for scenarios where additions are frequent and accessing the minimum element is less common.

Sorted List Priority Queue Implementation

In this implementation, the priority queue maintains a sorted list of entries. Each entry is a key-value pair, represented as instances of the Item class. These items are stored in a PositionalList, implemented as a doubly-linked list, ensuring that the first element is always the item with the smallest key.

Class Definition and Initialization

Here's the SortedPriorityQueue class, inheriting from the PriorityQueueBase class:

```
class SortedPriorityQueue(PriorityQueueBase):  # base class defines
Item
   """A min-oriented priority queue implemented with a sorted list."""
   def __init__(self):
      """Create a new empty Priority Queue."""
```

```
        self._data = PositionalList()
    def __len__(self):
        """Return the number of items in the priority queue."""
        return len(self._data)
    def add(self, key, value):
        """Add a key-value pair."""
        newest = self.Item(key, value)  # make new item instance
        walk = self._data.last()  # walk backward looking for smaller key
        while walk is not None and newest < walk.element():
            walk = self._data.before(walk)
        if walk is None:
            self._data.add_first(newest)  # new key is smallest
        else:
            self._data.add_after(walk, newest)  # newest goes after walk
    def min(self):
        """Return but do not remove (k,v) tuple with minimum key."""
        if self.is_empty():
            raise Empty("Priority queue is empty.")
        p = self._data.first()
        item = p.element()
        return (item._key, item._value)
        def remove_min(self):
        """Remove and return (k,v) tuple with minimum key."""
        if self.is_empty():
            raise Empty("Priority queue is empty.")
        item = self._data.delete(self._data.first())
        return (item._key, item._value)
```

Explanation of Methods

- Initialization (__init__): Creates an empty priority queue using an empty PositionalList.
- Length (__len__): Returns the number of items in the queue by returning the length of the PositionalList.
- Add (add): Adds a new item to the list, maintaining the sorted order. It starts from the end of the list and walks backward to find the appropriate position for the new item. This operation takes O(n) time.
- Minimum (min): Returns the item with the smallest key without removing it. Since the list is sorted, this is always the first element. This operation takes O(1) time.

- Remove Minimum (remove_min): Removes and returns the item with the smallest key. This is always the first element of the list, making the operation O(1) time.
- Internal PositionalList Class

The PositionalList class, which supports O(1) time operations for adding and deleting elements, is implemented using a doubly-linked list. Here is the same PositionalList class as provided earlier:

```
class PositionalList:
  """A sequential container of elements allowing positional access."""
  class _Node:
    __slots__ = '_element', '_prev', '_next'

    def __init__(self, element, prev, next):
      self._element = element
      self._prev = prev
      self._next = next
   class Position:
    """An abstraction representing the location of a single element."""
    def __init__(self, container, node):
      self._container = container
      self._node = node
    def element(self):
      return self._node._element
   def __init__(self):
    self._header = self._Node(None, None, None)
    self._trailer = self._Node(None, self._header, None)
    self._header._next = self._trailer
    self._size = 0
   def __len__(self):
    return self._size
   def is_empty(self):
    return self._size == 0
  def _validate(self, p):
    if not isinstance(p, self.Position):
      raise ValueError("p must be proper Position type")
    if p._container is not self:
      raise ValueError("p does not belong to this container")
    if p._node._next is None:
      raise ValueError("p is no longer valid")
    return p._node
    def _make_position(self, node):
    if node is self._header or node is self._trailer:
        return None
    else:
        return self.Position(self, node)

def first(self):
    return self._make_position(self._header._next)
  def last(self):
    return self._make_position(self._trailer._prev)
  def before(self, p):
    node = self._validate(p)
    return self._make_position(node._prev)
  def after(self, p):
    node = self._validate(p)
    return self._make_position(node._next)
  def add_first(self, e):
    return self._insert_between(e, self._header, self._header._next)
  def add_last(self, e):
    return self._insert_between(e, self._trailer._prev, self._trailer)
  def add_before(self, p, e):
    original = self._validate(p)
    return self._insert_between(e, original._prev, original)
  def add_after(self, p, e):
    original = self._validate(p)
    return self._insert_between(e, original, original._next)
  def delete(self, p):
    original = self._validate(p)
    return self._delete_node(original)
  def _insert_between(self, e, predecessor, successor):
    node = self._Node(e, predecessor, successor)
    predecessor._next = node
    successor._prev = node
    self._size += 1
    return self._make_position(node)
  def _delete_node(self, node):
    element = node._element
    node._prev._next = node._next
    node._next._prev = node._prev
    self._size -= 1
    node._prev = node._next = node._element = None
```

```
            return element
```

Comparison of List-Based Implementations Unsorted List:

- len: O(1)
- is_empty: O(1)
- add: O(1)
- min: O(n)
- remove_min: O(n)
- Sorted List:
- len: O(1)
- is_empty: O(1)
- add: O(n)
- min: O(1)
- remove_min: O(1)

The sorted list implementation provides constant-time operations for accessing and removing the minimum element but incurs a linear-time cost for adding new elements. This trade-off is appropriate for applications where the priority queue is read-heavy, meaning that finding and removing the minimum element occurs more frequently than inserting new elements.

4.1.2 – Heap Sorts with Priority Queue

Binary Heap Data Structure

The binary heap is a data structure that efficiently supports the operations required for a priority queue, providing logarithmic time complexity for both insertions and deletions. This improvement is achieved by combining the benefits of a binary tree structure with specific ordering and completeness properties.

Key Properties of a Binary Heap

Heap-Order Property:

For every position p other than the root in a heap T, the key at p is greater than or equal to the key at p's parent. This ensures that the smallest key is always at the root. Consequently, keys along any path from the root to a leaf are in nondecreasing order.

Complete Binary Tree Property:

The heap T is a complete binary tree, meaning all levels except possibly the last are fully filled, and all nodes in the last level are as far left as possible.

Structure and Height of a Heap

A complete binary tree of height h has:

The maximum number of nodes in levels 0 through h-1.

Nodes in level h fill the leftmost positions.

Example Heap

Consider the heap depicted below:

```
      (4,C)
     /    \
   (5,A)   (6,Z)
  /  \   /  \
(15,K)(9,F)(7,Q)(20,B)
 / \   / \   / \
(16,X)(25,J)(13,W)(12,H)(11,S)
```

This heap maintains the heap-order property and is a complete binary tree. It allows efficient access to the minimum element at the root.

Heap Operations

Insert Operation

To add an item to the heap:

1. Insert the new item at the next available position, maintaining the complete binary tree property.
2. Restore the heap-order property by "bubbling up" the new item:
   - Compare the new item with its parent.
   - If the new item is smaller, swap it with the parent.
   - Repeat this process until the heap-order property is restored.

Remove Minimum Operation

To remove the minimum item (root) from the heap:

1. Replace the root with the last item in the heap.
2. Restore the heap-order property by "bubbling down" the new root:

- Compare the new root with its children.
- If the new root is larger than any child, swap it with the smallest child.
- Repeat this process until the heap-order property is restored.

Height of a Heap

The height of a complete binary tree with n nodes is O(log n). This results in efficient logarithmic time operations for both insertions and deletions.

Here is a simple Python implementation of a min-oriented binary heap:

```
class Empty(Exception):
    """Error attempting to access an element from an empty container."""
    pass
class HeapPriorityQueue:
    """A min-oriented priority queue implemented with a binary heap."""
    def __init__(self):
        """Create a new empty Priority Queue."""
        self._data = []
    def __len__(self):
        """Return the number of items in the priority queue."""
        return len(self._data)
    def is_empty(self):
        """Return True if the priority queue is empty."""
        return len(self._data) == 0
    def _parent(self, j):
        return (j - 1) // 2
    def _left(self, j):
        return 2 * j + 1
    def _right(self, j):
        return 2 * j + 2
    def _has_left(self, j):
        return self._left(j) < len(self._data)
    def _has_right(self, j):
        return self._right(j) < len(self._data)
    def _swap(self, i, j):
        """Swap the elements at indices i and j of array."""
        self._data[i], self._data[j] = self._data[j], self._data[i]
    def _upheap(self, j):
        parent = self._parent(j)
        if j > 0 and self._data[j] < self._data[parent]:
            self._swap(j, parent)
            self._upheap(parent)
    def _downheap(self, j):
        if self._has_left(j):
            left = self._left(j)
            small_child = left
            if self._has_right(j):
                right = self._right(j)
                if self._data[right] < self._data[left]:
                    small_child = right
            if self._data[small_child] < self._data[j]:
                self._swap(j, small_child)
                self._downheap(small_child)
    def add(self, key, value):
        """Add a key-value pair to the priority queue."""
        self._data.append((key, value))
        self._upheap(len(self._data) - 1)
    def min(self):
        """Return but do not remove (k,v) tuple with minimum key."""
        if self.is_empty():
            raise Empty("Priority queue is empty.")
        return self._data[0]
    def remove_min(self):
        """Remove and return (k,v) tuple with minimum key."""
        if self.is_empty():
            raise Empty("Priority queue is empty.")
        self._swap(0, len(self._data) - 1)
        item = self._data.pop()
        self._downheap(0)
        return item
```

By using a binary heap, we can efficiently implement a priority queue with logarithmic time complexity for both insertions and removals. The binary heap's properties ensure that the minimum element is always accessible at the root, and its complete binary tree structure guarantees that the tree height is logarithmic relative to the number of elements, ensuring efficient operations.

Implementing a Priority Queue with a Heap

The binary heap is a powerful data structure for efficiently implementing a priority queue. In this implementation, both insertion and removal operations can be performed in logarithmic time, leveraging the properties of a binary heap. Below is the detailed Python implementation of a priority queue using a binary heap.

Priority Queue with Heap: Python Implementation

This implementation uses an array-based representation of a binary heap to manage the priority queue.

```
class PriorityQueueBase:
    """Abstract base class for a priority queue."""
    class Item:
        """Lightweight composite to store priority queue items."""
        __slots__ = '_key', '_value'
        def __init__(self, k, v):
            self._key = k
            self._value = v
        def __lt__(self, other):
            return self._key < other._key  # compare items based on their
keys
        def __repr__(self):
            return f'({self._key}, {self._value})'
class HeapPriorityQueue(PriorityQueueBase):
    """A min-oriented priority queue implemented with a binary heap."""

    def __init__(self):
        self._data = []
    #------------------------------ nonpublic behaviors ------------------------------
    def _parent(self, j):
        return (j-1) // 2
    def _left(self, j):
        return 2*j + 1
    def _right(self, j):
        return 2*j + 2
    def _has_left(self, j):
        return self._left(j) < len(self._data)
    def _has_right(self, j):
        return self._right(j) < len(self._data)
    def _swap(self, i, j):
        """Swap the elements at indices i and j of array."""
        self._data[i], self._data[j] = self._data[j], self._data[i]
    def _upheap(self, j):
        parent = self._parent(j)
        if j > 0 and self._data[j] < self._data[parent]:
            self._swap(j, parent)
            self._upheap(parent)  # recur at position of parent
    def _downheap(self, j):
        if self._has_left(j):
            left = self._left(j)
            small_child = left  # although right may be smaller
            if self._has_right(j):
                right = self._right(j)
                if self._data[right] < self._data[left]:
                    small_child = right
            if self._data[small_child] < self._data[j]:
                self._swap(j, small_child)
                self._downheap(small_child)  # recur at position of small
child
    #----------------------------- public behaviors -----------------------------
    def __len__(self):
        return len(self._data)
    def is_empty(self):
        return len(self._data) == 0
    def min(self):
        """Return but do not remove (k,v) tuple with minimum key."""
        if self.is_empty():
            raise ValueError('Priority queue is empty.')
        item = self._data[0]
        return (item._key, item._value)
    def add(self, key, value):
        """Add a key-value pair to the priority queue."""
        new_item = self.Item(key, value)
        self._data.append(new_item)
        self._upheap(len(self._data) - 1)  # upheap newly added position
    def remove_min(self):
        """Remove and return (k,v) tuple with minimum key."""
        if self.is_empty():
            raise ValueError('Priority queue is empty.')
        self._swap(0, len(self._data) - 1)  # put minimum item at the end
        item = self._data.pop()  # and remove it from the list
        if not self.is_empty():
            self._downheap(0)  # then fix new root
        return (item._key, item._value)
```

Explanation of Key Methods

Add Method:

- **Insertion:** The add method appends the new item to the end of the list to maintain the complete binary tree property.

- **Up-Heap Bubbling**: After insertion, it restores the heap-order property by repeatedly swapping the new item with its parent if it is smaller. This process continues until the heap-order property is satisfied or the item reaches the root.

Remove Min Method:

- Removal: The remove_min method removes the minimum item (at the root) by swapping it with the last item in the heap.
- Down-Heap Bubbling: After removal, it restores the heap-order property by repeatedly swapping the new root with its smaller child if the root is larger. This process continues until the heap-order property is satisfied or the item reaches a leaf.

Utility Methods:

- Methods like _parent, _left, _right, _has_left, _has_right, _swap, _upheap, and _downheap are utility methods to manage heap operations efficiently.

Benefits of Using a Heap

- Efficiency: Both add and remove_min operations are performed in O(log n) time, making the heap-based implementation of a priority queue efficient for large datasets.

Array-Based Representation: The array-based representation simplifies the implementation and avoids the complexity of a linked structure. This implementation ensures that the priority queue operations are both time-efficient and space-efficient, leveraging the properties of binary heaps and arrays effectively.

Bottom-Up Heap Construction Overview

Constructing a heap from a list of n elements can be optimized to run in O(n) time using a bottom-up construction approach. This is significantly more efficient than inserting elements one-by-one, which would take O(nlogn) time. The bottom-up method leverages the structure of a complete binary tree and performs down-heap operations starting from the deepest non-leaf nodes up to the root.

Steps of Bottom-Up Heap Construction:

- Initial Step: Treat each element as a single-entry heap.
- Subsequent Steps: Combine pairs of heaps into larger heaps by adding a new entry and performing a down-heap operation to maintain heap order.
- Final Step: Continue this process until the entire heap is formed.

- Implementation in Python

To implement bottom-up heap construction, we can utilize an array-based representation of the heap. We start by placing all elements in an array, then call the downheap method starting from the deepest non-leaf nodes up to the root.

Here's a Python implementation based on this approach:

```
class HeapPriorityQueue:
    """A min-oriented priority queue implemented with a binary heap."""
    class Item:
        """Lightweight composite to store priority queue items."""
        __slots__ = 'key', 'value'
        def __init__(self, k, v):
            self.key = k
            self.value = v
        def __lt__(self, other):
            return self.key < other.key  # compare items based on their
keys
    def __init__(self, contents=()):
        """Create a new priority queue.
        By default, queue will be empty. If contents is given, it should be
as an
        iterable sequence of (k,v) tuples specifying the initial contents.
        """
        self.data = [self.Item(k, v) for k, v in contents]  # empty by default
        if len(self.data) > 1:
            self.heapify()
    def __len__(self):
        """Return the number of items in the priority queue."""
        return len(self.data)
    def parent(self, j):
        return (j - 1) // 2
    def left(self, j):
        return 2 * j + 1
    def right(self, j):
        return 2 * j + 2
    def has_left(self, j):
        return self.left(j) < len(self.data)
   def has_right(self, j):
        return self.right(j) < len(self.data)
    def swap(self, i, j):
        """Swap the elements at indices i and j of array."""
        self.data[i], self.data[j] = self.data[j], self.data[i]
    def downheap(self, j):
        """Perform down-heap bubbling from index j."""
        while self.has_left(j):
            left = self.left(j)
            small_child = left
            if self.has_right(j):
                right = self.right(j)
                if self.data[right] < self.data[left]:
                    small_child = right
            if self.data[small_child] < self.data[j]:
                self.swap(j, small_child)
                j = small_child
            else:
                break
    def heapify(self):
        """Perform bottom-up construction of the heap."""
        start = self.parent(len(self.data) - 1)  # start at PARENT of last leaf
        for j in range(start, -1, -1):  # going to and including the root
            self.downheap(j)
```

# Additional methods such as add, min, remove_min can be added for full functionality.

Asymptotic Analysis

The bottom-up heap construction is more efficient than incremental insertion due to the following reasons:

- Down-heap Operations: Each non-leaf node needs at most
- O(logn) operations, but the total time across all nodes sums up to O(n).
- Edge-Disjoint Paths: The paths for down-heap operations are edge-disjoint, ensuring that each edge in the heap is involved in at most one down-heap operation.

Justification

The primary cost is the down-heap operation. The sum of the lengths of the paths from each non-leaf node to its "inorder successor" leaf is bounded by the number of edges in the tree, which is O(n). Thus, bottom-up construction takes O(n) time, making it asymptotically faster than the incremental approach.

By using bottom-up heap construction, we can efficiently build a heap in O(n) time. This method is particularly useful when we need to build a heap from an existing collection of elements, such as during the first phase of heap-sort.

To implement a modified pq_sort function in Python that supports an optional key parameter for customized sorting, we can follow these steps:

Define the PriorityQueue Class: Implement a simple priority queue class using a heap. Python's heapq module can be used to simplify this process.

Modify pq_sort Function: Update the pq_sort function to accept an optional key parameter and use this key for the priority queue operations.

Here's how the implementation might look:

PriorityQueue Class

We'll implement a priority queue using Python's heapq module for efficiency. This class will support adding elements and removing the minimum element while optionally using a key function to determine the ordering.

```
import heapq
class PriorityQueue:
    def __init__(self):
        self._data = []
    def add(self, key, value):
        heapq.heappush(self._data, (key, value))
    def remove_min(self):
        return heapq.heappop(self._data)
```

pq_sort Function

The pq_sort function will be updated to accept an optional key parameter. If provided, the key will be used to compute the keys for the priority queue. If not, the elements themselves will be used as keys.

```
def pq_sort(C, key=None):
    """Sort a collection of elements stored in a positional list."""
    n = len(C)
    P = PriorityQueue()
    if key is None:
        # Use elements as their own keys
        for j in range(n):
            element = C.delete(C.first())
            P.add(element, element)
    else:
        # Use the provided key function to determine the keys
        for j in range(n):
            element = C.delete(C.first())
            P.add(key(element), element)
    for j in range(n):
        (k, v) = P.remove_min()
        C.add_last(v)  # Store smallest remaining element in C
```

Usage Example

Here's how you can use the modified pq_sort function with and without a key parameter:

```
class PositionalList:
    # Assuming PositionalList is implemented with necessary methods
    # like first(), delete(), add_last(), and len()
    pass
# Example usage:
C = PositionalList()
# Populate C with elements...
# Sorting without a key (natural order)
pq_sort(C)
# Sorting with a key (e.g., sorting strings as integers)
pq_sort(C, key=int)
```

PriorityQueue Class: This class encapsulates the priority queue operations using a heap. The add method pushes a tuple (key, value) onto the heap, and the remove_min method pops the smallest element.

pq_sort Function: This function sorts the collection C by first transferring all elements into a priority queue, and then repeatedly removing the minimum element and adding it back to C.

If a key function is provided, it is used to compute the keys for the priority queue.

If no key function is provided, the elements themselves are used as keys.

Positional List Assumption

The above code assumes a PositionalList class with methods like first(), delete(), add_last(), and __len__() is already implemented as part of your collection's structure. This class should handle the operations on the positional list that are used in the pq_sort function.

This approach ensures that the sorting is done according to the specified or default order while leveraging the efficiency of a heap-based priority queue for the sorting operations.

PRIORITY QUEUES

A priority queue is a type of abstract data type (ADT) where each element is associated with a priority, and elements are served according to their priority. Higher priority elements are dequeued before lower priority ones. This structure is commonly used in scenarios where it's essential to process elements based on their importance or urgency.

Here are some key concepts and details about priority queues:

Characteristics of Priority Queues:

1.Elements with Priorities:

Each element in a priority queue has an associated priority. This priority determines the order in which elements are removed.

2.Operations:

- Insert (enqueue): Add an element to the queue with an assigned priority.
- Remove (dequeue): Remove and return the element with the highest priority.
- Peek/Top: Return the element with the highest priority without removing it.

3.Implementation:

Priority queues can be implemented using various data structures, including arrays, linked lists, binary heaps, Fibonacci heaps, and balanced binary search trees.

The most common implementation is the binary heap, which provides efficient insertion and removal operations.

4.Types of Priority Queues:

- Max-priority queue: The element with the highest priority is dequeued first.
- Min-priority queue: The element with the lowest priority is dequeued first.

Applications of Priority Queues:

Scheduling Algorithms: Used in operating systems to manage tasks by priority.

Graph Algorithms: Used in Dijkstra's and A* algorithms to select the next node to process based on shortest path or minimum cost.

Event Simulation: Manage events in discrete event simulation systems.

Data Compression: Used in algorithms like Huffman coding.

Network Routing: Prioritize data packets based on importance or urgency.

Implementing a Priority Queue

Using a Binary Heap

A binary heap is a complete binary tree that satisfies the heap property:

- **Max-Heap:** The key at a node is greater than or equal to the keys of its children.
- **Min-Heap:** The key at a node is less than or equal to the keys of its children.

Operations in a Binary Heap:

1.Insert:

- Add the new element to the end of the heap (maintains the complete tree property).
- "Bubble up" this element by swapping it with its parent until the heap property is restored.
- Time complexity O(logn)

2.Remove (Extract-Max/Min):

- Replace the root of the heap (maximum or minimum element) with the last element.
- "Bubble down" this element by swapping it with its larger (for max-heap) or smaller (for min-heap) child until the heap property is restored.
- Time complexity O(logn)

3.Peek/Top:

- Return the root element without removing it.
- Time complexity: O(1)

Example in Python:

```
import heapq
class PriorityQueue:
    def __init__(self):
        self.heap = []
    def push(self, priority, item):
        heapq.heappush(self.heap, (priority, item))
    def pop(self):
        return heapq.heappop(self.heap)[1]
    def peek(self):
        return self. heap [0][1] if self.heap else None
# Example usage
pq = PriorityQueue()
pq.push(2, "task2")
pq.push(1, "task1")
pq.push(3, "task3")
print(pq.pop())  # Outputs: task1
print(pq.peek()) # Outputs: task2
```

In this example, a min-priority queue is implemented using Python's heapq module, which provides a binary heap. The priority queue stores elements as tuples, where the first element is the priority.

Priority queues are versatile and efficient data structures essential for various algorithms and applications where elements need to be processed based on priority.

Priority Queue Abstract Data Type:

A Priority Queue is an abstract data type (ADT) that operates similarly to a regular queue, but with an added feature: each element has a priority associated with it. Elements are dequeued not in the order they were enqueued, but rather according to their priority. The element with the highest priority is served before those with lower priorities. Here's a detailed breakdown:

Key Characteristics

Priority: Each element has an associated priority. Elements with higher priorities are dequeued before elements with lower priorities.

Ordering: If two elements have the same priority, they are dequeued based on their order in the queue, usually the order in which they were enqueued.

Operations:

A Priority Queue supports the following primary operations:

1.Insert (or Enqueue):

- Adds an element with a given priority to the queue.
- Typical signature: insert(element, priority).

2.Delete (or Dequeue) the Minimum/Maximum:

- Removes and returns the element with the highest priority (in a max-priority queue) or the lowest priority (in a min-priority queue).
- Typical signature: deleteMin() or deleteMax().

3.Peek (or Find-Min/Find-Max):

- Returns the element with the highest priority without removing it.
- Typical signature: peekMin() or peekMax().

4.IsEmpty:

- Checks if the priority queue is empty.
- Typical signature: isEmpty().

5.Size:

- Returns the number of elements in the priority queue.
- Typical signature: size().

Implementations:

Priority Queues can be implemented using various data structures, each with different performance characteristics for the operations listed above:

1.Binary Heap:

- A common implementation that offers good average-case performance.
- Binary Min-Heap: Supports insert and deleteMin operations in O(logn) time.
- Binary Max-Heap: Supports insert and deleteMax operations in O(logn) time.

2.Binomial Heap:

- Allows for faster merging of two heaps.
- Supports insert in O(logn), deleteMin in O(logn), and merge in O(logn).

3.Fibonacci Heap:

- Offers better amortized time complexity for decrease key and delete operations.
- Supports insert in O(1) amortized time, deleteMin in O(logn) amortized time, and decreaseKey in O(1) amortized time.

4.Pairing Heap:

- Similar to Fibonacci heap with simpler structure and practical performance.
- Supports insert and deleteMin in O(logn) amortized time.

5.Unsorted List:

- Simplest to implement but inefficient for deletion.
- insert in O(1), deleteMin in O(n).

6.Sorted List:

- Keeps elements sorted at all times.
- insert in O(n), deleteMin in O(1).

Use Cases

Priority Queues are used in various applications, including:

- Dijkstra's Algorithm for shortest path.
- Huffman Coding for data compression.
- Event-driven simulation where events are processed in a specific order.
- Task Scheduling in operating systems where tasks are prioritized.

Example (Using a Binary Min-Heap)

```
class MinHeap:
    def __init__(self):
        self.heap = []
    def insert(self, element):
        self.heap.append(element)
        self._heapify_up(len(self.heap) - 1)
    def deleteMin(self):
        if len(self.heap) == 1:
            return self.heap.pop()
        root = self.heap[0]
        self.heap[0] = self.heap.pop()
        self._heapify_down(0)
        return root
    def peekMin(self):
        return self.heap[0] if self.heap else None
    def isEmpty(self):
        return len(self.heap) == 0
    def size(self):
        return len(self.heap)
    def _heapify_up(self, index):
        parent = (index - 1) // 2
        if index > 0 and self.heap[index] < self.heap[parent]:
            self. heap[index], self.heap[parent] = self.heap[parent],
self.heap[index]
            self._heapify_up(parent)
    def _heapify_down(self, index):
        child = 2 * index + 1
        if child >= len(self.heap):
            return
        if child + 1 < len(self.heap) and self.heap[child + 1] <
self.heap[child]:
            child += 1
        if self.heap[index] > self.heap[child]:
            self.heap[index],    self.heap[child]    =    self.heap[child],
self.heap[index]
            self._heapify_down(child)
# Example usage
pq = MinHeap()
pq.insert(10)
pq.insert(5)
pq.insert(20)
print(pq.deleteMin())  # Outputs: 5
print(pq.peekMin())    # Outputs: 10
```

This is a simplified version to illustrate the concept. In practice, error handling and additional features might be necessary depending on the application.

4.1.3 – Implementing a Priority Queue

Implementing a Priority Queue can be achieved using various underlying data structures. One of the most common and efficient implementations is through a binary heap, specifically a min-heap if we want the priority queue to always serve the smallest element first. Here's a detailed implementation in Python using a binary min-heap.

Priority Queue Implementation Using a Binary Min-Heap

```
class PriorityQueue:
    def __init__(self):
        self.heap = []
    def insert(self, element):
        """Insert an element into the priority queue."""
        self.heap.append(element)
        self._heapify_up(len(self.heap) - 1)
   def deleteMin(self):
        """Remove  and  return  the  smallest  element  from  the  priority
queue."""
        if self.isEmpty():
            raise IndexError("deleteMin() called on empty priority queue")
        if len(self.heap) == 1:
            return self.heap.pop()
        min_element = self.heap[0]
        self.heap[0] = self.heap.pop()  # Move the last element to the root
        self._heapify_down(0)
        return min_element
    def peekMin(self):
        """Return the smallest element without removing it."""
        if self.isEmpty():
            raise IndexError("peekMin() called on empty priority queue")
        return self.heap[0]
    def isEmpty(self):
        """Check if the priority queue is empty."""
        return len(self.heap) == 0
    def size(self):
        """Return the number of elements in the priority queue."""
        return len(self.heap)
    def _heapify_up(self, index):
        """Ensure the heap property is maintained after insertion."""
        parent = (index - 1) // 2
        if index > 0 and self.heap[index] < self.heap[parent]:
            self.heap[index], self.heap[parent] = self.heap[parent], self.heap[index]
            self._heapify_up(parent)
    def _heapify_down(self, index):
        """Ensure the heap property is maintained after deletion."""
        child = 2 * index + 1
        if child >= len(self.heap):
            return
        if child + 1 < len(self.heap) and self.heap[child + 1] < self.heap[child]:
            child += 1
        if self.heap[index] > self.heap[child]:
            self.heap[index], self.heap[child] = self.heap[child], self.heap[index]
            self._heapify_down(child)
# Example usage
pq = PriorityQueue()
pq.insert(10)
pq.insert(5)
pq.insert(20)
print(pq.deleteMin())  # Outputs: 5
print(pq.peekMin())    # Outputs: 10
print(pq.size())       # Outputs: 2
```

Explanation

1.Initialization:

The PriorityQueue class is initialized with an empty list self.heap that will store the heap elements.

2.Insert Operation:

The insert method appends the new element to the heap and then calls _heapify_up to maintain the heap property by comparing the inserted element with its parent and swapping them if necessary.

3.DeleteMin Operation:

The deleteMin method removes the smallest element (root of the heap). If the heap is empty, it raises an IndexError. Otherwise, it replaces the root with the last element in the heap, removes the last element, and then calls _heapify_down to maintain the heap property by comparing the new root with its children and swapping them if necessary.

4.PeekMin Operation:

The peekMin method returns the smallest element without removing it. If the heap is empty, it raises an IndexError.

5.IsEmpty Operation:

The isEmpty method checks if the heap is empty by returning the boolean value of the heap's length.

6.Size Operation:

The size method returns the number of elements in the heap.

7._heapify_up and _heapify_down Helper Methods:

- _heapify_up ensures the heap property is maintained after insertion by moving the element up the tree until it is greater than its parent.
- _heapify_down ensures the heap property is maintained after deletion by moving the element down the tree until it is less than its children.

This implementation ensures efficient insert and deleteMin operations with a time complexity of O(logn), where n is the number of elements in the priority queue.

Heaps:

Heaps are specialized tree-based data structures that satisfy the heap property. Heaps are commonly used to implement priority queues because they provide efficient access to the minimum or maximum element.

Types of Heaps:

Binary Heap:

- A binary heap is a complete binary tree where each node is smaller (in a min-heap) or larger (in a max-heap) than its children.
- Min-Heap: The key at the root must be the minimum among all keys present in the binary heap.
- Max-Heap: The key at the root must be the maximum among all keys present in the binary heap.

Binomial Heap:

- A collection of binomial trees (a specific type of tree) that are linked together.
- Useful for merging heaps quickly.

Fibonacci Heap:

- Consists of a collection of trees with a more relaxed structure than binomial heaps.
- Provides better amortized time complexity for many operations, particularly decreaseKey.

Pairing Heap:

- Simplified structure similar to Fibonacci heaps.
- Often used in practice due to good performance on typical operations.

Binary Heap Details

<u>Binary Min-Heap Properties</u>

1.Heap Property:

Every parent node is less than or equal to its child nodes.

2.Shape Property:

A binary heap is a complete binary tree, meaning all levels are fully filled except possibly for the last level, which is filled from left to right.

Operations on Binary Min-Heap:

<u>Insert Operation</u>

- Add the element to the end of the heap (maintain complete tree property).
- Heapify up: Compare the added element with its parent; if smaller, swap them.
- Repeat the process until the heap property is restored.

DeleteMin Operation

- Replace the root (minimum element) with the last element in the heap.
- Remove the last element.
- Heapify down: Compare the new root with its children; if larger than a child, swap it with the smaller child.
- Repeat the process until the heap property is restored.

PeekMin Operation:

Return the root element of the heap.

Helper Methods: _heapify_up and _heapify_down

```
class MinHeap:
   def __init__(self):
      self.heap = []
   def insert(self, element):
      """Insert an element into the heap."""
      self.heap.append(element)
      self._heapify_up(len(self.heap) - 1)
   def deleteMin(self):
      """Remove and return the smallest element from the heap."""
      if self.isEmpty():
         raise IndexError("deleteMin() called on empty heap")
      if len(self.heap) == 1:
         return self.heap.pop()
      min_element = self.heap[0]
      self.heap[0] = self.heap.pop()  # Move the last element to the root
      self._heapify_down(0)
      return min_element
def peekMin(self):
      """Return the smallest element without removing it."""
      if self.isEmpty():
         raise IndexError("peekMin() called on empty heap")
      return self.heap[0]
   def isEmpty(self):
      """Check if the heap is empty."""
      return len(self.heap) == 0
   def size(self):
      """Return the number of elements in the heap."""
      return len(self.heap)
```

```
        def _heapify_up(self, index):
            """Ensure the heap property is maintained after insertion."""
            parent = (index - 1) // 2
            if index > 0 and self.heap[index] < self.heap[parent]:
                self.heap[index], self.heap[parent] = self.heap[parent], self.heap[index]
                self._heapify_up(parent)
        def _heapify_down(self, index):
            """Ensure the heap property is maintained after deletion."""
            child = 2 * index + 1
            if child >= len(self.heap):
                return
            if child + 1 < len(self.heap) and self.heap[child + 1] < self.heap[child]:
                child += 1
            if self.heap[index] > self.heap[child]:
                self.heap[index], self.heap[child] = self.heap[child], self.heap[index]
                self._heapify_down(child)
    # Example usage
    pq = MinHeap()
    pq.insert(10)
    pq.insert(5)
    pq.insert(20)
    print(pq.deleteMin())  # Outputs: 5
    print(pq.peekMin())    # Outputs: 10
    print(pq.size())       # Outputs: 2
```

Explanation:

1.Initialization:

The MinHeap class initializes an empty list self.heap to store heap elements.

2.Insert Operation:

nsert(element): Adds the element to the heap, then calls _heapify_up to restore the heap property by comparing and swapping the element with its parent until the property is satisfied.

3.DeleteMin Operation:

deleteMin(): Replaces the root with the last element, removes the last element, and calls _heapify_down to restore the heap property by comparing and swapping the element with its children until the property is satisfied.

4.PeekMin Operation:

peekMin(): Returns the root element without removing it. Raises an error if the heap is empty.

5.Helper Methods:

- _heapify_up(index): Ensures the heap property is maintained after insertion by moving the element up the tree.
- _heapify_down(index): Ensures the heap property is maintained after deletion by moving the element down the tree.

Complexity:

- Insert: O(logn)
- DeleteMin: O(logn)
- PeekMin: O(1)
- IsEmpty: O(1)
- Size: O(1)

This binary heap implementation efficiently supports the essential operations required for a priority queue**.**

Sorting with a Priority Queue:

Using a priority queue to sort a collection of elements is an example of Heap Sort. The basic idea is to insert all elements into a priority queue (a min-heap for ascending order sorting or a max-heap for descending order sorting) and then extract the elements one by one in sorted order. Here is a step-by-step breakdown of Heap Sort using a min-heap for ascending order:

Insert all elements into the priority queue (min-heap).

Extract the minimum element repeatedly to get the elements in sorted order.

Heap Sort Implementation in Python

```
class MinHeap:
    def __init__(self):
        self.heap = []
    def insert(self, element):
        self.heap.append(element)
        self._heapify_up(len(self.heap) - 1)
    def deleteMin(self):
        if self.isEmpty():
            raise IndexError("deleteMin() called on empty heap")
        if len(self.heap) == 1:
            return self.heap.pop()
        min_element = self.heap[0]
```

```
            self.heap[0] = self.heap.pop()
            self._heapify_down(0)
            return min_element
        def peekMin(self):
            if self.isEmpty():
                raise IndexError("peekMin() called on empty heap")
            return self.heap[0]
        def isEmpty(self):
            return len(self.heap) == 0
        def size(self):
            return len(self.heap)
        def _heapify_up(self, index):
            parent = (index - 1) // 2
            if index > 0 and self.heap[index] < self.heap[parent]:
                self.heap[index], self.heap[parent] = self.heap[parent], self.heap[index]
                self._heapify_up(parent)
        def _heapify_down(self, index):
            child = 2 * index + 1
            if child >= len(self.heap):
                return
            if child + 1 < len(self.heap) and self.heap[child + 1] < self.heap[child]:
                child += 1
            if self.heap[index] > self.heap[child]:
                self.heap[index], self.heap[child] = self.heap[child], self.heap[index]
                self._heapify_down(child)
    def heap_sort(elements):
        min_heap = MinHeap()
        sorted_elements = []
        for element in elements:
            min_heap.insert(element)
        while not min_heap.isEmpty():
            sorted_elements.append(min_heap.deleteMin())
        return sorted_elements
    # Example usage
    unsorted_list = [10, 3, 76, 34, 23, 32]
    sorted_list = heap_sort(unsorted_list)
    print(sorted_list)  # Outputs: [3, 10, 23, 32, 34, 76]
```

Sorting with a Priority Queue:

Using a priority queue to sort a collection of elements is an example of Heap Sort. The basic idea is to insert all elements into a priority queue (a min-heap for ascending order sorting or a max-heap for descending order sorting) and then extract the elements one by one in sorted order.

Here is a step-by-step breakdown of Heap Sort using a min-heap for ascending order:

- Insert all elements into the priority queue (min-heap).
- Extract the minimum element repeatedly to get the elements in sorted order.

Heap Sort Implementation in Python

```
class MinHeap:
    def __init__(self):
        self.heap = []
    def insert(self, element):
        self.heap.append(element)
        self._heapify_up(len(self.heap) - 1)
    def deleteMin(self):
        if self.isEmpty():
            raise IndexError("deleteMin() called on empty heap")
        if len(self.heap) == 1:
            return self.heap.pop()
        nonelements = self.heap[0]
        self.heap[0] = self.heap.pop()
        self._heapify_down(0)
        return min_element
    def peekMin(self):
        if self.isEmpty():
            raise IndexError("peekMin() called on empty heap")
        return self.heap[0]
    def isEmpty(self):
        return len(self.heap) == 0
    def size(self):
        return len(self.heap)
    def _heapify_up(self, index):
        parent = (index - 1) // 2
        if index > 0 and self.heap[index] < self.heap[parent]:
            self.heap[index], self.heap[parent] = self.heap[parent], self.heap[index]
            self. Heapify_up(parent)
    def _heapify_down(self, index):
        child = 2 * index + 1
        if child >= len(self.heap):
            return
        if child + 1 < len(self.heap) and self.heap[child + 1] < self.heap[child]:
            child += 1
            if self.heap[index] > self.heap[child]:
        self.heap[index], self.heap[child] = self.heap[child], self.heap[index]
        self._heapify_down(child)
def heap_sort(elements):
    min_heap = MinHeap()
    sorted_elements = []
    for element in elements:
        min_heap.insert(element)
    while not min_heap.isEmpty():
        sorted_elements.append(min_heap.deleteMin())
    return sorted_elements
# Example usage
unsorted_list = [10, 3, 76, 34, 23, 32]
sorted_list = heap_sort(unsorted_list)
print(sorted_list)  # Outputs: [3, 10, 23, 32, 34, 76]
```

Let us Sum All

A Priority Queue is an abstract data type where each element has a priority, and elements are dequeued in priority order. It can be efficiently implemented using heaps, which are complete binary trees satisfying the heap property (max-heap or min-heap). Key operations include insert, extract-min/max, and peek. Heapsort leverages the heap structure to sort elements by inserting all elements into a heap and then extracting them in sorted order, achieving a time complexity of O(nlogn). This combination provides efficient priority management and sorting capabilities.

Check Your Progress

1. Which of the following operations is the most efficient in a heap?
   - A) Insert
   - B) Delete
   - C) Search
   - D) Update
2. In a max-heap, the root node contains the:
   - A) Minimum element
   - B) Middle element
   - C) Maximum element
   - D) Random element
3. What is the time complexity of inserting an element into a heap?

A) $O(1)$

B) $O(n)$

C) $O(\log n)$

D) $O(n\log n)$

4. Which data structure is commonly used to implement a priority queue?

   A) Linked List

   B) Binary Search Tree

   C) Hash Table

   D) Heap

5. What is the first step in the Heapsort algorithm?

   A) Extract all elements from the heap

   B) Build a heap from the input data

   C) Merge sorted subarrays

   D) Partition the array

6. In a binary heap implemented using an array, the left child of the node at index ii is at index:

   A) i+1

   B) 2i+1

   C) 2i

   D) 2i+2

7. Which of the following is not a valid operation on a priority queue?

   A) Insert

   B) Extract-Min/Max

   C) Peek

   D) Rotate

8. What is the main difference between a min-heap and a max-heap?

   A) Structure of the tree

   B) Number of elements

   C) Order of elements

   D) Size of elements

9. During the "down-heap" operation, which node is compared with its children?

   A) The root node

B) The last node

C) Any leaf node

D) The current node being adjusted

10. What is the time complexity of extracting the minimum element from a min-heap?

A) $O(1)$

B) $O(n)$

C) $O(\log n)$

D) $O(n \log n)$

SECTION 4.2: . MAPS, HASH TABLES, AND SKIP LISTS

4.2.1 – Maps and Dictionaries

The Map ADT

The Map Abstract Data Type (ADT) and define its behaviors to be consistent with those of Python's built-in dict class. Here are the most significant behaviors of a map M[k]: Return the value v associated with key k in map M, if one exists; otherwise, raise a KeyError. In Python, this is implemented with the special method __getitem__. M[k] = v: Associate value v with key k in map M, replacing the existing value if the map already contains an item with key equal to k

In Python, this is implemented with the special method __setitem__.

- del M[k]: Remove from map M
- M the item with key equal to k if M has no such item, then raise a KeyError. In Python, this is implemented with the special method __delitem__.
- len(M): Return the number of items in map M .In Python, this is implemented with the special method __len__.
- iter(M): The default iteration for a map generates a sequence of keys in the map. In Python, this is implemented with the special method __iter__, and it allows loops of the form for k in M.

Additional behaviors include:

- k in M: Return True if the map contains an item with key k. In Python, this is implemented with the special method __contains__.
- M.get(k, d=None): Return M[k] if key k exists in the map; otherwise, return default value d. M.setdefault(k, d): If key k exists in the map, simply return M[k]; if key k does not exist, set M[k] = d and return that value.
- M.pop(k, d=None): Remove the item associated with key k from the map and return its associated value v. If key k is not in the map, return default value d or raise KeyError if parameter d is None).
- M.popitem(): Remove an arbitrary key-value pair from the map and return a (k,v) tuple representing the removed pair. If the map is empty, raise a KeyError.
- M.clear(): Remove all key-value pairs from the map.
- M.keys(): Return a set-like view of all keys of M
- M.values(): Return a set-like view of all values of M
- M.items(): Return a set-like view of (k,v) tuples for all entries of M
- M.update(M2): Assign M[k] = v for every (k,v) pair in map M2.
- M == M2: Return True if maps
- M and M2 have identical key-value associations.
- M != M2: Return True if maps M
- M and M2 do not have identical key-value associations.

Map Operations

The following sequence of operations demonstrates the core behaviors of a map using Python's dict class:

```
M = {}
len(M)          # 0
M['K'] = 2      # {'K': 2}
M['B'] = 4      # {'K': 2, 'B': 4}
M['U'] = 2      # {'K': 2, 'B': 4, 'U': 2}
M['V'] = 8      # {'K': 2, 'B': 4, 'U': 2, 'V': 8}
M['K'] = 9      # {'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['B']          # 4
```

```
M['X']            # KeyError
M.get('F')         # None
M.get('F', 5)      # 5
M.get('K', 5)      # 9
len(M)             # 4
del M['V']         # {'K': 9, 'B': 4, 'U': 2}
M.pop('K')         # 9, {'B': 4, 'U': 2}
M.keys()           # dict_keys(['B', 'U'])
M.values()         # dict_values([4, 2])
M.items()          # dict_items([('B', 4), ('U', 2)])
M.setdefault('B', 1) # 4, {'B': 4, 'U': 2}
M.setdefault('A', 1) # 1, {'A': 1, 'B': 4, 'U': 2}
M.popitem()        # ('B', 4), {'A': 1, 'U': 2}
```

Application: Counting Word Frequencies

Consider the problem of counting the number of occurrences of words in a document. This can be achieved using a map where words are keys and word counts are values. The following code snippet illustrates this application:

```
freq = {}
for piece in open(filename).read().lower().split():
   word = ''.join(c for c in piece if c.isalpha())
   if word:
      freq[word] = 1 + freq.get(word, 0)
max_word = ''
max_count = 0
for (w, c) in freq.items():
   if c > max_count:
      max_word = w
      max_count = c
print("The most frequent word is", max_word)
print("Its number of occurrences is", max_count)
```

Python's MutableMapping Abstract Base Class

Python's collections module provides abstract base classes (ABCs) such as Mapping and MutableMapping for defining map-like structures. The MutableMapping class extends Mapping to include mutating methods. To create a user-defined map class, inheriting from MutableMapping is useful as it provides concrete implementations for all behaviors except for the core ones (__getitem__, __setitem__, __delitem__, __len__, and __iter__).

Example implementations of derived behaviors:

```
def __contains__(self, k):
    try:
        self[k]
        return True
    except KeyError:
        return False
def setdefault(self, k, d):
    try:
        return self[k]
    except KeyError:
        self[k] = d
        return d
```

Our MapBase Class

To facilitate various map implementations, we define our own MapBase class that extends MutableMapping and includes a nonpublic Item class for storing key-value pairs.

```
class MapBase(MutableMapping):
    """Our own abstract base class that includes a nonpublic Item
class."""
    class Item:
        """Lightweight composite to store key-value pairs as map items."""
        __slots__ = '_key', '_value'
        def __init__(self, k, v):
            self._key = k
            self._value = v
        def __eq__(self, other):
            return self._key == other._key
        def __ne__(self, other):
            return not (self == other)
        def __lt__(self, other):
            return self._key < other._key
```

Simple Unsorted Map Implementation

We implement a simple map using an unsorted list to store key-value pairs. This class demonstrates basic map operations but is not efficient due to the linear search required for key lookups.

```
class UnsortedTableMap(MapBase):
    """Map implementation using an unordered list."""
    def __init__(self):
        """Create an empty map."""
        self._table = []
    def __getitem__(self, k):
        """Return value associated with key k (raise KeyError if not
found)."""
        for item in self._table:
            if k == item._key:
                return item._value
        raise KeyError(f'Key Error: {repr(k)}')
    def __setitem__(self, k, v):
        """Assign value v to key k, overwriting existing value if present."""
        for item in self._table:
            if k == item._key:
                item._value = v
                return
        self._table.append(self.Item(k, v))
    def __delitem__(self, k):
        """Remove item associated with key k (raise KeyError if not
found)."""
        for j in range(len(self._table)):
            if k == self._table[j]._key:
                self._table.pop(j)
                return
        raise KeyError(f'Key Error: {repr(k)}')
    def __len__(self):
        """Return number of items in the map."""
        return len(self._table)
    def __iter__(self):
        """Generate iteration of the map's keys."""
        for item in self._table:
            yield item._key
```

This implementation provides a straightforward introduction to the Map ADT, paving the way for more advanced map structures discussed in subsequent sections.Hash tables, one of the most practical data structures for implementing a map, and the structure used by Python’s dict class. Hash tables use a hash function to map keys to indices in an array, known as a bucket array. This allows the basic map operations of getitem, setitem, and delitem to be performed in O(1) average time.

Maps, also known as dictionaries, are abstract data types that store key-value pairs. They allow efficient insertion, deletion, and retrieval of values based on keys. Maps are fundamental in many areas of computer science and programming due to their versatility and efficiency. Let's explore maps in more detail, including their implementation using hash tables and skip lists.

Maps and Their Uses:

Basic Operations

- Insertion: Add a key-value pair to the map.
- Deletion: Remove a key-value pair from the map.
- Lookup: Retrieve the value associated with a given key.

Maps are used in a variety of applications, such as:

- Database indexing: Quickly retrieving records.
- Caching: Storing computed results for fast retrieval.
- Counting: Tracking occurrences of items (e.g., word frequency in text).
- Associative arrays: Commonly used in programming languages for data storage.

Intuitive Example of a Map Using Integers as Keys

Consider a restricted setting where a map uses keys that are known to be integers in a range from 0 to N−1. We can represent this map using a lookup table of length N

0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

In this representation, the value associated with key k is stored at index k of the table.

Challenges with Larger and Non-Integer Keys

There are two main challenges in extending this framework:

- We may not want to devote an array of length N if N≫n, where n is the number of items.
- Keys in a map are not generally integers.

4.2.3 –Hash Tables

The Concept of Hash Functions

A hash function h maps each key k to an integer within the range [0,N−1], where N is the capacity of the bucket array. We use the hash function value h(k) as an index into our bucket array, A. However, distinct keys can have the same hash value, resulting in collisions. Therefore, we use a bucket array where each bucket can manage a collection of items sent to that index by the hash function.

Structure of a Bucket Array

0 1 2 3 4 5 6 7 8 9 10

(1, D) (25, C) (3, F) (14, Z) (39, C) (6, A) (7, Q)

Hash Functions

The goal of a hash function is to minimize collisions and distribute keys uniformly across the bucket array.

Hash Codes

A hash code maps a key k to an integer. This integer can be negative or exceed the range [0,N−1]. The computation of a hash code is often independent of the specific hash table size.

Compression Functions

A compression function maps the hash code to an index within the range [0,N−1].

Examples of Hash Codes

Bit Representation as an Integer: For data types represented using at most 32 bits, we can use the bit representation directly as the hash code.

Polynomial Hash Codes: For variable-length objects like strings, a polynomial hash code uses a formula like $x_0a^{n-1} + x_1a^{n-2} + \ldots + x_{n-1}$, where a is a chosen constant.

Cyclic-Shift Hash Codes: A variant that uses cyclic shifts instead of multiplications to spread the influence of each component across the resulting hash code

Hash Codes in Python Python's built-in function hash(x) returns an integer hash code for an object x. Only immutable data types are hashable in Python to ensure that a particular object's hash code remains constant during its lifespan.

Compression Functions

Division Method: Maps an integer imodN. Choosing N as a prime number helps spread out the distribution of hash values.

MAD Method: Maps an integer [(ai+b)modp]modN, where p is a prime number larger than N, and a and b are randomly chosen integers.

Collision-Handling Schemes

Separate Chaining

Each bucket A[j] stores a secondary container, typically a small list or another map, to handle collisions. This method is simple and efficient but requires extra space for the secondary containers.

Open Addressing

An alternative to separate chaining, open addressing stores items directly in the bucket array and handles collisions by finding another open slot.

Linear Probing: If a bucket is occupied, the next sequential bucket is checked.

Quadratic Probing: Uses a quadratic function to determine the next bucket to check.

Double Hashing: Uses a second hash function to determine the interval between probes. Hash tables provide an efficient way to implement a map with average O(1) time complexity for insertions, deletions, and searches. Properly designed hash functions and effective collision-handling schemes are crucial for maintaining performance and minimizing collisions.

Implementations of Maps

Hash Tables:

Hash tables are one of the most common implementations of maps. They use a hash function to compute an index into an array of buckets, from which the desired value can be found.

Hash Function: A function that takes a key and returns an integer, which is then used as an index in the array.

Buckets: Each bucket can store multiple key-value pairs to handle collisions.

Collision Resolution Techniques:

Chaining: Each bucket contains a linked list of entries that hash to the same index. When a collision occurs, the new entry is added to the list.

Open Addressing: When a collision occurs, the algorithm searches for the next available bucket according to a probing sequence (e.g., linear probing, quadratic probing, or double hashing).

Performance:

- Average-case Time Complexity: O(1) for insertion, deletion, and lookup, assuming a good hash function and a low load factor.
- Worst-case Time Complexity: O(n) if many collisions occur, which can happen if the hash function is poor or the load factor is too high.

Advantages and Disadvantages:

- Advantages: Fast average-case performance, simple implementation.
- Disadvantages: Poor worst-case performance, no inherent order among elements, sensitive to hash function quality and load factor.

Load Factors, Rehashing, and Efficiency

In hash table implementations, the load factor (λ), defined as the ratio of the number of entries n to the number of buckets N (λ = n/N), plays a crucial role in maintaining efficiency.

Separate Chaining:

Load Factor Considerations: With separate chaining, the load factor λ should ideally be kept below 0.9. As λ approaches 1, the probability of collisions increases significantly, leading to longer chains and thus more overhead during operations.

Performance: If λ is maintained below 0.9, hash table operations generally remain efficient because the chains remain reasonably short. However, if λ exceeds this threshold, performance degrades due to the increased likelihood of collisions.

Open Addressing:

Load Factor Considerations: In open addressing schemes, such as linear probing, the load factor should be maintained below 0.5. As λ approaches 1, clusters of entries form, causing the probing process to "bounce around" the array, leading to inefficiencies.

Performance: For linear probing, keeping λ below 0.5 is crucial to avoid performance degradation. In other probing methods, such as quadratic probing, λ can be slightly higher (e.g., 2/3) before performance significantly degrades.

Rehashing

When an insertion causes the load factor to exceed the specified threshold, the table is typically resized to reduce the load factor. The process involves:

Doubling the Size: The table size is doubled, and all existing entries are rehashed into the new table. This helps distribute entries more uniformly across the new, larger array.

Reapplying the Compression Function: Each entry’s hash code is recalculated using the new table size. This helps scatter the items throughout the new bucket array. Rehashing is generally an expensive operation, but if the table size is doubled each time, the amortized cost per insertion remains O(1).

Efficiency of Hash Tables

The average-case efficiency of hash tables relies on the assumption that a good hash function uniformly distributes entries across the bucket array. The expected number of keys in a bucket is n/N, which is O(1) if n is O(N).

Amortized Cost: The periodic rehashing contributes an additional O(1) amortized cost for setitem and getitem operations.

Worst-Case Scenario: A poor hash function that maps all items to the same bucket can degrade performance to linear time, making hash table operations as inefficient as those of unsorted lists.

Comparison of Running Times

The following table summarizes the expected and worst-case running times for various operations in an unsorted list and a hash table:

Operation	**List (expected)**	**Hash Table (expected)**	**List (worst-case)**	**Hash Table**
getitem	O(n)	O(1)	O(n)	O(n)
setitem	O(n)	O(1)	O(n)	O(n)
delitem	O(n)	O(1)	O(n)	O(n)
len	O(1)	O(1)	O(1)	O(1)
iter	O(n)	O(n)	O(n)	O(n)

Practical Considerations

Python’s dict Class: The dict class in Python is implemented using hashing and is highly efficient, typically providing constant-time performance for basic operations.

Python's dict implementation uses an open addressing scheme with a load factor threshold enforced to be less than 2/3.

Security Considerations: Hash tables can be vulnerable to Denial-of-Service (DoS) attacks if an attacker can cause many keys to collide. Modern implementations, including Python's, introduce randomization in hash code computation to mitigate such risks.

Implementations of Hash Tables

HashMapBase Class

The HashMapBase class provides the core functionality for hash table operations, such as handling the bucket array, maintaining the number of entries, and performing hash calculations. Here are the key elements:

```
class HashMapBase(MapBase):
    """Abstract base class for map using hash-table with MAD 
compression."""
    def __init__(self, cap=11, p=109345121):
        """Create an empty hash-table map."""
        self.table = cap * [None]
        self.n = 0  # number of entries in the map
        self.prime = p  # prime for MAD compression
        self.scale = 1 + randrange(p-1)  # scale from 1 to p-1 for MAD
        self.shift = randrange(p)  # shift from 0 to p-1 for MAD
    def hash_function(self, k):
        return (hash(k) * self.scale + self.shift) % self.prime % 
len(self.table)
    def __len__(self):
        return self.n
    def __getitem__(self, k):
        j = self.hash_function(k)
        return self.bucket_getitem(j, k)  # may raise KeyError
    def __setitem__(self, k, v):
        j = self.hash_function(k)
        self.bucket_setitem(j, k, v)  # subroutine maintains self.n
        if self.n > len(self.table) // 2:  # keep load factor <= 0.5
            self.resize(2 * len(self.table) - 1)  # number 2^x - 1 is often 
prime
    def __delitem__(self, k):
        j = self.hash_function(k)
        self.bucket_delitem(j, k)  # may raise KeyError
        self.n -= 1
    def resize(self, c):  # resize bucket array to capacity c
        old = list(self.items())  # use iteration to record existing items
        self.table = c * [None]  # then reset table to desired capacity
        self.n = 0  # n recomputed during subsequent adds
        for (k, v) in old:
            self[k] = v  # reinsert old key-value pair
```

Separate Chaining

The ChainHashMap class implements a hash table using separate chaining:

```
class ChainHashMap(HashMapBase):
    """Hash map implemented with separate chaining for collision
resolution."""
    def bucket_getitem(self, j, k):
        bucket = self.table[j]
        if bucket is None:
            raise KeyError('Key Error: ' + repr(k))  # no match found
        return bucket[k]  # may raise KeyError
    def bucket_setitem(self, j, k, v):
        if self.table[j] is None:
            self.table[j] = UnsortedTableMap()  # bucket is new to the table
        oldsize = len(self.table[j])
        self.table[j][k] = v
        if len(self.table[j]) > oldsize:  # key was new to the table
            self.n += 1  # increase overall map size
    def bucket_delitem(self, j, k):
        bucket = self.table[j]
        if bucket is None:
            raise KeyError('Key Error: ' + repr(k))  # no match found
        del bucket[k]  # may raise KeyError
    def __iter__(self):
        for bucket in self.table:
            if bucket is not None:  # a nonempty slot
                for key in bucket:
                    yield key
```

Linear Probing

The ProbeHashMap class implements a hash table using open addressing with linear probing:

```
class ProbeHashMap(HashMapBase):
    """Hash map implemented with linear probing for collision
resolution."""
    AVAIL = object()  # sentinel marks locations of previous deletions
      def is_available(self, j):
```

```
    """Return True if index j is available in table."""
    return self.table[j] is None or self.table[j] is ProbeHashMap.AVAIL
    def find_slot(self, j, k):
    """Search for key k in bucket at index j.
       Return (success, index) tuple, described as follows:
    If match was found, success is True and index denotes its
location.
    If no match found, success is False and index denotes first
available slot.
    """
    firstAvail = None
    while True:
      if self.is_available(j):
        if firstAvail is None:
          firstAvail = j  # mark this as first avail
        if self.table[j] is None:
          return (False, firstAvail)  # search has failed
      elif k == self.table[j].key:
        return (True, j)  # found a match
      j = (j + 1) % len(self.table)  # keep looking (cyclically)
    def bucket_getitem(self, j, k):
    found, s = self.find_slot(j, k)
    if not found:
      raise KeyError('Key Error: ' + repr(k))  # no match found
    return self.table[s].value
    def bucket_setitem(self, j, k, v):
    found, s = self.find_slot(j, k)
    if not found:
      self.table[s] = self._Item(k, v)  # insert new item
      self.n += 1  # size has increased
    else:
      self.table[s].value = v  # overwrite existing
    def bucket_delitem(self, j, k):
    found, s = self.find_slot(j, k)
    if not found:
      raise KeyError('Key Error: ' + repr(k))  # no match found
    self.table[s] = ProbeHashMap.AVAIL  # mark as vacated
    self.n -= 1
    def __iter__(self):
    for j in range(len(self.table)):
      if not self.is_available(j):
        yield self.table[j].key
```

Understanding the principles of load factors, rehashing, and the implementation details of hash tables is essential for leveraging their efficiency. By keeping load factors within recommended thresholds and implementing effective collision resolution strategies, hash tables can maintain average-case constant time complexity for most operations. However, care must be taken to ensure that hash functions are well-designed and that security considerations, such as protection against hash collisions, are addressed.

In this implementation, the SortedTableMap class supports all the operations of the sorted map ADT. It includes traditional map operations (__getitem__, __setitem__, __delitem__) and additional operations for ordered searches (find_min, find_max, find_lt, find_le, find_gt, find_ge, find_range). This class stores the map's items in an array-based sequence to maintain the keys in increasing order, allowing efficient binary search.

Implementation of the SortedTableMap Class

```
class SortedTableMap(MapBase):
    """Map implementation using a sorted table."""
    #---------------------------- nonpublic behaviors ----------------------------
    def _find_index(self, k, low, high):
        """Return index of the leftmost item with key greater than or equal
to k.
        Return high + 1 if no such item qualifies.
        That is, j will be returned such that:
        all items of slice table[low:j] have key < k
        all items of slice table[j:high+1] have key >= k
        """
        if high < low:
            return high + 1  # no element qualifies
        else:
            mid = (low + high) // 2
            if k == self._table[mid]._key:
                return mid  # found exact match
            elif k < self._table[mid]._key:
                return self._find_index(k, low, mid - 1)  # Note: may return
mid
            else:
               return self._find_index(k, mid + 1, high)  # answer is right of
mid
    #---------------------------- public behaviors ----------------------------
    def __init__(self):
        """Create an empty map."""
        self._table = []
    def __len__(self):
        """Return number of items in the map."""
        return len(self._table)
    def __getitem__(self, k):
        """Return value associated with key k (raise KeyError if not
found)."""
        j = self._find_index(k, 0, len(self._table) - 1)
        if j == len(self._table) or self._table[j]._key != k:
            raise KeyError('Key Error: ' + repr(k))
        return self._table[j]._value
    def __setitem__(self, k, v):
        """Assign value v to key k, overwriting existing value if present."""
        j = self._find_index(k, 0, len(self._table) - 1)
        if j < len(self._table) and self._table[j]._key == k:
            self._table[j]._value = v  # reassign value
        else:
            self._table.insert(j, self._Item(k, v))  # adds new item
    def __delitem__(self, k):
        """Remove item associated with key k (raise KeyError if not
found)."""
        j = self._find_index(k, 0, len(self._table) - 1)
        if j == len(self._table) or self._table[j]._key != k:
            raise KeyError('Key Error: ' + repr(k))
        self._table.pop(j)  # delete item
    def __iter__(self):
        """Generate keys of the map ordered from minimum to
maximum."""
        for item in self._table:
            yield item._key
    def __reversed__(self):
        """Generate keys of the map ordered from maximum to
minimum."""
        for item in reversed(self._table):
            yield item._key
    def find_min(self):
        """Return (key,value) pair with minimum key (or None if empty)."""
        if len(self._table) > 0:
            return (self._table[0]._key, self._table[0]._value)
        else:
            return None
    def find_max(self):
        """Return (key,value) pair with maximum key (or None if empty)."""
        if len(self._table) > 0:
            return (self._table[-1]._key, self._table[-1]._value)
        else:
            return None
    def find_ge(self, k):
        """Return (key,value) pair with least key greater than or equal to
k."""
        j = self._find_index(k, 0, len(self._table) - 1)  # j's key >= k
        if j < len(self._table):
            return (self._table[j]._key, self._table[j]._value)
        else:
            return None
    def find_lt(self, k):
        """Return (key,value) pair with greatest key strictly less than k."""
        j = self._find_index(k, 0, len(self._table) - 1)  # j's key >= k
        if j > 0:
            return (self._table[j-1]._key, self._table[j-1]._value)  # Note use
of j-1
        else:
            return None
    def find_gt(self, k):
        """Return (key,value) pair with least key strictly greater than k."""
        j = self._find_index(k, 0, len(self._table) - 1)  # j's key >= k
        if j < len(self._table) and self._table[j]._key == k:
            j += 1  # advanced past match
        if j < len(self._table):
            return (self._table[j]._key, self._table[j]._value)
        else:
            return None
    def find_range(self, start, stop):
        """Iterate all (key,value) pairs such that start <= key < stop.
        If start is None, iteration begins with minimum key of map.
        If stop is None, iteration continues through the maximum key of
map.
        """
        if start is None:
            j = 0
        else:
            j = self._find_index(start, 0, len(self._table) - 1)  # find first result
        while j < len(self._table) and (stop is None or self._table[j]._key <
    stop):
            yield (self._table[j]._key, self._table[j]._value)
            j += 1
```

Explanation of Key Methods

- Binary Search (_find_index): A recursive binary search is used to locate the index of the leftmost item with a key greater than or equal to k. This helps to quickly find elements for exact or inexact searches.
- Item Access (__getitem__): Uses _find_index to locate the item and raise a KeyError if the key is not found.
- Item Assignment (__setitem__): Uses _find_index to determine where to insert a new item or update an existing one.
- Item Deletion (__delitem__): Uses _find_index to find the item to remove and raise a KeyError if the key is not found.
- Finding Extremes (find_min, find_max): These methods return the minimum or maximum (key, value) pairs in constant time by accessing the first or last element of the sorted table.
- Inexact Searches (find_ge, find_lt, find_gt): These methods use _find_index to locate items with keys near the specified target, handling edge cases to return appropriate results.
- Range Search (find_range): Iterates through all items in the specified key range, using _find_index to find the starting point and a loop to iterate until the stopping condition is met.

Performance Analysis

- Search Operations: Binary search ensures O(log n) time complexity for search-related operations (__getitem__, find_ge, find_lt, find_gt).
- Update Operations: Insertions and deletions require shifting elements to maintain order, resulting in O(n) worst-case time complexity for __setitem__ and __delitem__.
- Iteration: Iterating keys in order takes O(n) time for __iter__ and __reversed__.

Applications of Sorted Maps

- Flight Databases: Efficiently find flights within a specific time range using lexicographically ordered keys.
- Maxima Sets: Maintain a set of cost-performance pairs to quickly find the best performance within a given cost, updating the set as new pairs are added.

These applications demonstrate the practical benefits of using sorted maps for problems requiring ordered key operations and inexact searches.

4.2.4 –Skip Lists

Skip Lists: Overview and Operations

Skip Lists are an efficient data structure for implementing a sorted map Abstract Data Type (ADT) that combines the fast search capabilities of binary search with the dynamic update capabilities of linked lists. Here, we explore the structure, operations, and performance of skip lists.

Structure of Skip Lists

A skip list consists of multiple levels of linked lists:

- **Level S0**: Contains every item in the map, plus two sentinel keys **-∞** and **+∞**.
- **Level S1, S2, ..., Sh**: Each subsequent level contains a subset of the items in the level directly below it, chosen randomly, plus the sentinel keys. The topmost level, Sh, contains only the sentinel keys.

Intuitively, each higher level in the skip list helps to "skip" over a larger number of elements, thus facilitating faster search operations. The average number of elements in level Si is about **n/2^i**.

Operations in Skip Lists

1. **Search**
   - **Algorithm**: **SkipSearch(k)**
   - **Process**: Start from the top-left corner, move right while the current key is less than or equal to **k**, drop down a level when you can't move right anymore. Continue this until you reach the bottom level.
   - **Time Complexity**: Expected O(log n)

2. **Insertion**
   - **Algorithm**: **SkipInsert(k, v)**
   - **Process**: Use **SkipSearch(k)** to find the appropriate position at the bottom level, insert the new element, then randomly decide how many levels the new element should span (simulated by flipping a coin). Insert the new element at each higher level until a "tails" is flipped.
   - **Time Complexity**: Expected O(log n)
3. **Deletion**
   - **Algorithm**: **SkipDelete(k)**
   - **Process**: Use **SkipSearch(k)** to find the element, remove it from all levels it spans.
   - **Time Complexity**: Expected O(log n)

Detailed Skip List Algorithms

SkipSearch Algorithm:

```
def SkipSearch(k):
    p = start  # begin at the start position (top-left corner)
    while below(p) is not None:
        p = below(p)  # drop down one level
        while k >= key(next(p)):
            p = next(p)  # move right within the current level
    return p
```

SkipInsert Algorithm:

```
def SkipInsert(k, v):
    p = SkipSearch(k)
    q = None  # q will represent the top node in the new item's tower
    i = -1
    while True:
        i += 1
        if i >= h:
            h += 1  # add a new level to the skip list
            s = insertAfterAbove(None, s, (-∞, None))  # grow leftmost
tower
            insertAfterAbove(s, t, (+∞, None))  # grow rightmost tower
        while above(p) is None:
            p = prev(p)  # scan backward
        p = above(p)  # jump up to higher level
```

```
            q = insertAfterAbove(p, q, (k, v))  # increase height of new item's
        tower
            if coinFlip() == tails:
                break
          n += 1
          return q
```

Probabilistic Analysis

- **Height of Skip List**: With high probability, the height **h** of the skip list is O(log n). The probability that any level **i** has at least one element decreases exponentially, ensuring that **h** remains logarithmic with respect to **n**.
- **Search Time**: The expected number of forward moves at any level is O(1), as each element has a 50% chance of appearing in the next level up. With O(log n) levels, the total expected search time is O(log n).
- **Space Usage**: The expected number of elements across all levels combined is O(n).

Summary of Performance

Operation	**Expected Time Complexity**
Length (len(M))	O(1)
Search (k in M)	O(log n)
Insert (M[k] = v)	O(log n)
Delete (del M[k])	O(log n)
Find min, find max	O(1)
Find lt, gt, le, ge	O(log n)
Find range (start, stop)	O(s + log n)
Iteration (iter(M), reversed)	O(n)

Skip lists provide a balanced trade-off, combining the efficient search of binary search with the dynamic update capabilities of linked lists, making them a versatile choice for implementing sorted maps.

Skip Lists:

Skip lists are a probabilistic data structure that maintains elements in sorted order, allowing for fast search, insertion, and deletion.

- Levels: Multiple layers of linked lists, with higher levels skipping more elements. The bottom level contains all elements, and each higher level contains a subset of elements.
- Insertion and Deletion: Performed by adjusting pointers at multiple levels to maintain the structure.

Structure:

- Level Assignment: Elements are assigned levels randomly, with each higher level having fewer elements.
- Search Operation: Starts at the highest level and moves forward or downward until the target element is found.

Performance

- Average-case Time Complexity: O(log n) for insertion, deletion, and lookup.
- Worst-case Time Complexity: O(n), although this is very unlikely due to the probabilistic nature of level assignment.

Advantages and Disadvantages

- Advantages: Maintains sorted order, supports efficient range queries, easier to implement than balanced trees.
- Disadvantages: Uses more memory due to multiple levels, performance can vary based on randomness.

Python does not have a built-in skip list, but you can use the sortedcontainers module, which provides similar functionality.

```
from sortedcontainers import SortedList
# Sorted List (similar to Skip List)
skip_list = SortedList()
# Insertion
skip_list.add(3)
skip_list.add(1)
skip_list.add(2)
# Deletion
skip_list.remove(1)
# Lookup
```

```
index = skip_list.index(2)
print(skip_list)  # Output: SortedList([2, 3])
print(index)  # Output: 0
```

Install the sortedcontainers module using pip if you haven't already:

“pip install sortedcontainers”

4.2.5 – Sets, Mutl sets, Multimaps

Sets, Multisets, and Multimaps

These three data structures extend the concept of the map ADT, offering different ways to manage collections of elements or key-value pairs. Let's dive deeper into each one:

Set ADT

A **set** is an unordered collection of unique elements. Python provides built-in classes **set** and **frozenset** for mutable and immutable sets, respectively. Sets are often implemented using hash tables to allow for efficient membership tests and other operations.

Fundamental operations for a set S include:

- **S.add(e)**: Add element **e** to the set.
- **S.discard(e)**: Remove element **e** from the set if it exists.
- **e in S**: Check if **e** is in the set.
- **len(S)**: Get the number of elements in the set.
- **iter(S)**: Iterate over the elements of the set.

Additional operations include:

- Removing elements: **S.remove(e)**, **S.pop()**, **S.clear()**
- Boolean comparisons: **S == T**, **S != T**, **S <= T**, **S < T**, **S >= T**, **S > T**, **S.isdisjoint(T)**
- Set operations: **S | T**, **S |= T**, **S & T**, **S &= T**, **S ^ T**, **S ^= T**, **S - T**, **S -= T**

Python's **collections** module offers **MutableSet** as an abstract base class for sets. This base class provides default implementations for many methods, relying on the subclass to implement the core methods: **add**, **discard**, **__contains__**, **__len__**, and **__iter__**.

Example implementations for some derived methods include:

- Checking if one set is a proper subset of another (__**lt**__ method):

```
def __lt__(self, other):
    if len(self) >= len(other):
        return False
    for e in self:
        if e not in other:
            return False
    return True
```

Union of two sets (__or__ method):

```
def __or__(self, other):
    result = type(self)()
    for e in self:
        result.add(e)
    for e in other:
        result.add(e)
    return result
```

In-place union (__ior__ method):

```
def __ior__(self, other):
    for e in other:
        self.add(e)
    return self
```

Multiset

A **multiset** (or bag) allows for duplicate elements. One way to implement a multiset is by using a map where the key is an element and the value is the count of occurrences. Python's **collections.Counter** class effectively serves as a multiset.

Example usage:

```
from collections import Counter
multiset = Counter(['a', 'b', 'a', 'c', 'b', 'a'])
print(multiset)  # Output: Counter({'a': 3, 'b': 2, 'c': 1})
```

Multimaps

A **multimap** allows a single key to be associated with multiple values. This can be implemented using a standard map where each key maps to a container (like a list) of values.

Example implementation of a **MultiMap**:

```
class MultiMap:
    """A multimap class built upon use of an underlying map for storage."""
    MapType = dict  # Map type; can be redefined by subclass
    def __init__(self):
        self.map = self.MapType()
        self.n = 0

    def __iter__(self):
        for k, secondary in self.map.items():
            for v in secondary:
                yield (k, v)
    def add(self, k, v):
        container = self.map.setdefault(k, [])
        container.append(v)
        self.n += 1
    def pop(self, k):
        secondary = self.map[k]
        v = secondary.pop()
        if not secondary:
            del self.map[k]
        self.n -= 1
        return (k, v)
    def find(self, k):
        secondary = self.map[k]
        return (k, secondary[0])
    def find_all(self, k):
        secondary = self.map.get(k, [])
        for v in secondary:
            yield (k, v)
```

In this **MultiMap** class:

- **add(k, v)**: Adds the pair **(k, v)** to the multimap.
- **pop(k)**: Removes and returns an arbitrary **(k, v)** pair with key **k**.
- **find(k)**: Returns an arbitrary **(k, v)** pair with key **k**.
- **find_all(k)**: Generates all **(k, v)** pairs with the given key.

This approach leverages a dictionary where each key points to a list of values, enabling the storage of multiple values per key efficiently.

Maps and Dictionaries

Maps and dictionaries are fundamental data structures used to store key-value pairs. They allow for efficient data retrieval, insertion, and deletion based on keys. In Python, dictionaries (dict) are the primary implementation of maps.

Dictionaries in Python

A Python dictionary is a collection of key-value pairs, where each key is unique. Dictionaries are implemented as hash tables, providing average O(1) time complexity for insertion, deletion, and lookup operations.

Basic Operations

1.Creating a Dictionary

```
# Creating an empty dictionary
my_dict = {}
# Creating a dictionary with initial key-value pairs
my_dict = {
    "apple": 1,
    "banana": 2,
    "cherry": 3
}
```

2.Insertion

```
# Adding a new key-value pair
my_dict["date"] = 4
print(my_dict)  # Output: {'apple': 1, 'banana': 2, 'cherry': 3, 'date': 4}
```

3.Deletion

```
# Removing a key-value pair
del my_dict["banana"]
print(my_dict)  # Output: {'apple': 1, 'cherry': 3, 'date': 4}
# Using pop() to remove and return a value
value = my_dict.pop("cherry")
print(value)  # Output: 3
print(my_dict)  # Output: {'apple': 1, 'date': 4}
```

4.Lookup

```
# Accessing a value by key
value = my_dict["apple"]
print(value)  # Output: 1
# Using get() to avoid KeyError
```

```
value = my_dict.get("date", "Not Found")
print(value)  # Output: 4
```

5.Iteration

```
# Iterating over keys
for key in my_dict:
    print(key, my_dict[key])
# Output:
# apple 1
# date 4
# Iterating over key-value pairs
for key, value in my_dict.items():
    print(key, value)
# Output:
# apple 1
# date 4
```

6.Checking for Keys

```
# Checking if a key exists in the dictionary
if "apple" in my_dict:
    print("Apple is in the dictionary")
# Output: Apple is in the dictionary
```

Advanced Usage

1.Default Values with ‘defaultdict’

Using defaultdict from the collections module to handle missing keys.

```
from collections import defaultdict
    # Creating a defaultdict with a default value type of int
    ddict = defaultdict(int)
    ddict["apple"] += 1  # Increments the count for "apple"
    print(ddict)  # Output: defaultdict(<class 'int'>, {'apple': 1})
```

2.Counting with ‘Counter’

Using Counter from the collections module for counting occurrences.

```
from collections import Counter
    # Creating a Counter from a list of items
    fruits = ["apple", "banana", "apple", "cherry", "banana", "banana"]
    fruit_count = Counter(fruits)
    print(fruit_count)  # Output: Counter({'banana': 3, 'apple': 2, 'cherry': 1})
```

3.Ordered Dictionaries

Using OrderedDict to maintain the order of keys as they are added.

```
from collections import OrderedDict
          # Creating an OrderedDict
          ordered_dict = OrderedDict()
          ordered_dict["banana"] = 2
          ordered_dict["apple"] = 1
          ordered_dict["cherry"] = 3
          # OrderedDict maintains the insertion order
          for key, value in ordered_dict.items():
              print(key, value)
          # Output:
          # banana 2
          # apple 1
          # cherry 3
```

Sorted Maps:

Sorted maps maintain their elements in a specific order. They can be implemented using data structures like balanced trees (e.g., AVL trees, Red-Black trees) or skip lists.

Balanced Trees

- AVL Trees: A self-balancing binary search tree where the difference in heights of left and right subtrees cannot be more than one.
- Red-Black Trees: A self-balancing binary search tree where each node has an extra bit for denoting the color of the node, ensuring the tree remains balanced.

Time Complexity: O(logn) for insertion, deletion, and lookup.

Using sortedcontainers for a sorted dictionary:

```
from sortedcontainers import SortedDict
# Sorted Dictionary (Sorted Map)
sorted_map = SortedDict()
# Insertion
sorted_map['banana'] = 2
sorted_map['apple'] = 1
sorted_map['cherry'] = 3
# Deletion
del sorted_map['banana']
# Lookup
value = sorted_map['cherry']
print(sorted_map)  # Output: SortedDict({'apple': 1, 'cherry': 3})
print(value)  # Output: 3
```

Skip Lists:

Skip lists are an alternative to balanced trees and provide efficient ordered maps.

Structure

- Multiple Levels: Each level skips over some elements, allowing faster search.
- Random Level Assignment: Elements are randomly assigned levels, creating a probabilistic balance.

Operations:

Search, Insert, Delete: All have an average time complexity of O(logn).

Sets and MultiSets:

Sets:

A set is a collection of unique elements.

- HashSet: Implements a set using a hash table.
- TreeSet: Implements a set using a balanced tree (maintains order).

Operations:

Add, Remove, Contains: Typically O(1) for hash sets, O(logn) for tree sets.

MultiSets:

A multiset (or bag) allows multiple occurrences of the same element.

- HashMultiSet: Uses a hash table to store element counts.
- TreeMultiSet: Uses a balanced tree to store elements and their counts (maintains order).

MultiMaps:

A multimap is like a map but allows multiple values for a single key.

- HashMultiMap: Implements a multimap using a hash table, where each key maps to a collection of values.
- TreeMultiMap: Implements a multimap using a balanced tree, maintaining order and allowing multiple values per key.

Python's built-in set and collections. Counter can be used to implement sets and multisets.

```
# Set
set_example = {1, 2, 3}
```

```
# Insertion
set_example.add(4)
# Deletion
set_example.remove(2)
# Lookup
exists = 3 in set_example
print(set_example)  # Output: {1, 3, 4}
print(exists)  # Output: True
```

For multisets, use collections.Counter:

```
from collections import Counter
# Multiset
multiset = Counter()
# Insertion
multiset.update([1, 2, 2, 3, 3, 3])
# Deletion
multiset.subtract([2, 3])
# Lookup
count = multiset[3]
print(multiset)  # Output: Counter({3: 2, 1: 1, 2: 1})
print(count)  # Output: 2
```

Summary

- **Maps and Dictionaries**: Abstract data types for key-value pairs, implemented using hash tables or skip lists.
- **Priority Queue and Heap Sort**: Efficient sorting using binary heaps.
- **Hash Tables**: Efficient for average-case operations, using chaining or open addressing for collision resolution.
- **Skip Lists**: Provide efficient ordered maps with probabilistic balancing.
- **Sorted Maps**: Maintain elements in order, implemented using balanced trees or skip lists.
- **Sets and MultiSets**: Collections of unique elements (sets) or elements with counts (multisets), implemented using hash tables or balanced trees.
- **MultiMaps**: Allow multiple values for a single key, implemented using hash tables or balanced trees.

Each data structure has its own strengths and is suited for different types of applications. Understanding these structures helps in choosing the right one for a given problem.

Let Us Sums Up

Maps and Dictionaries are data structures that store key-value pairs, enabling efficient data retrieval, insertion, and deletion based on keys. Hash Tables implement maps using a hash function to index into an array of buckets, achieving average-case $O(1)O(1)$ time complexity for most operations. Sorted Maps maintain elements in a sorted order using self-balancing binary search trees, ensuring $O(\log n)$ time complexity for operations. Skip Lists are probabilistic structures that facilitate fast search, insertion, and deletion within an ordered sequence, with average-case $O(\log n)O(\log n)$ time complexity. Sets store unique elements, while Multi Sets allow duplicates, and Multi Maps handle multiple values per key, offering versatile ways to manage collections and associations of items.

Check Your Progress

1. Which data structure stores key-value pairs and allows efficient retrieval, insertion, and deletion based on keys?
   A) Array
   B) Stack
   C) Map
   D) Queue
2. What is the average-case time complexity for lookups in a hash table?
   A) $O(n)$
   B) $O(\log n)$
   C) $O(1)$
   D) $O(n\log n)$
3. Which data structure maintains its elements in a sorted order based on keys?
   A) Hash Table
   B) Sorted Map
   C) Skip List
   D) Set
4. What is the average-case time complexity for search, insertion, and deletion operations in a skip list?
   A) $O(1)$
   B) $O(\log n)$

C) $O(n)$

D) $O(n\log n)$

5. Which data structure stores unique elements without duplicates?

   A) Multi Set

   B) Multi Map

   C) Set

   D) Skip List

6. In which data structure can multiple values be associated with a single key?

   A) Map

   B) Multi Map

   C) Hash Table

   D) Sorted Map

7. What is the main advantage of using a hash table over other types of maps?

   A) Sorted order of elements

   B) Fast average-case lookups

   C) Allows duplicate keys

   D) Simple implementation

8. Which of the following data structures is probabilistic and allows for efficient ordered operations?

   A) Hash Table

   B) Sorted Map

   C) Skip List

   D) Set

9. Which data structure allows for duplicate elements and is also known as a bag?

   A) Set

   B) Multi Map

   C) Skip List

   D) Multi Set

10. Which data structure is typically implemented using self-balancing binary search trees?

    A) Hash Table

B) Sorted Map

C) Skip List

D) Multi Set

11. Which operation is typically not supported by a hash table?

A) Insertion

B) Deletion

C) Sorted traversal

D) Lookup

12. In a skip list, what is the purpose of having multiple levels?

A) To increase the size of the structure

B) To reduce the time complexity of operations

C) To ensure uniqueness of elements

D) To store additional data

13. Which data structure uses a hash function to map keys to specific locations?

A) Sorted Map

B) Skip List

C) Hash Table

D) Set

14. What is the worst-case time complexity for insertion in a hash table with a good hash function and proper collision resolution?

A) $O(1)$

B) $O(\log n)$

C) $O(n)$

D) $O(n2)$

15. Which type of map would you use if you need to maintain order based on keys?

A) Hash Table

B) Unordered Map

C) Sorted Map

D) Multi Map

16. Which of the following is true about a set?

A) It allows duplicate elements.

B) It stores elements in a specific order.

C) It is designed to store unique elements.

D) It always uses a hash table for implementation.

17. In a hash table, what technique is used to handle collisions?

A) Binary search

B) Chaining

C) Sorting

D) Recursion

18. What is a characteristic feature of a multi map?

A) Keys are unique but values can be duplicated.

B) Both keys and values are unique.

C) Multiple keys can map to a single value.

D) A single key can map to multiple values.

19. Which data structure allows fast membership testing and is implemented using hash functions?

A) Array

B) Stack

C) Set

D) Queue

20. Which data structure is designed to store elements in such a way that they can be retrieved in sorted order without additional sorting?

A) Hash Table

B) Sorted Map

C) Set

D) Multi Set

Unit Summary

Unit 4 delves into advanced data structures crucial for efficient data manipulation and retrieval. It covers Priority Queues, Maps, Hash Tables, and Skip Lists. Priority Queues manage elements based on priority levels, with implementations using Heaps and facilitating sorting operations. Maps and Dictionaries, implemented using Hash Tables, allow for associative data storage and retrieval. Sorted Maps enhance search capabilities, while Skip Lists offer efficient probabilistic data structures for search and insertion operations. Mastery of these structures is essential for designing efficient algorithms and systems, making Unit 4 a cornerstone in computer science education.

Glossary

- **Priority Queue**: An abstract data type that manages elements based on priority levels.
- **Heap:** A specialized tree-based data structure used to implement priority queues.
- **Sorting:** The process of arranging elements in a specific order, often facilitated by priority queues.
- **Map:** A data structure that stores key-value pairs, allowing efficient retrieval of values based on keys.
- **Hash Table**: A data structure that implements associative arrays, providing constant-time average case lookup.
- **Sorted Map**: A map that maintains elements in sorted order, enhancing search capabilities.
- **Skip List:** A probabilistic data structure used for associative data storage and retrieval, offering logarithmic-time average case insertion, deletion, and search operations.
- **Set:** A collection of unique elements, often implemented using hash tables or skip lists.
- **Multi-Map:** A map that allows multiple values to be associated with the same key.
- **Priority Queue Sorting**: Sorting algorithms that utilize priority queues for efficient sorting, such as Heap Sort.

Self – Assessment Questions

1. What is the primary advantage of using a heap-based priority queue over other data structures?
2. Explain how a hash table resolves collisions and maintains constant-time average case lookup.
3. Compare and contrast sorted maps and hash tables in terms of search efficiency and memory usage.

4. How does a skip list achieve logarithmic-time average case insertion, deletion, and search operations?
5. Discuss the advantages and disadvantages of using a skip list compared to a balanced tree for storing sorted data.
6. Describe the process of sorting elements using a priority queue.
7. Explain the significance of the "priority" concept in priority queues and how it influences element retrieval.
8. Discuss scenarios where using a multi-map would be more advantageous than a traditional map.
9. How does the choice of hash function impact the performance of a hash table?
10. Compare the space complexity of various data structures covered in Unit 4 and discuss their implications for memory usage.

Activities / Exercises / Case Studies

1. Activity: Implement a priority queue using a heap data structure.
   Exercise: Design and implement various operations for the priority queue, such as insertion, extraction of the highest priority element, and updating priorities.
   Case Study: Analyze real-world applications that benefit from priority queues, such as job scheduling in operating systems or event handling in event-driven programming.
2. Activity: Implement a hash table data structure from scratch.
   Exercise: Develop functions for key insertion, lookup, and deletion, and handle collisions using chaining or probing techniques.
   Case Study: Investigate how popular programming languages implement hash tables in their standard libraries and compare their approaches.
3. Activity: Implement a skip list data structure and visualize its internal structure.
4. Exercise: Conduct performance tests to compare the average-case time complexity of skip lists with other data structures for various operations.
5. Case Study: Explore real-world applications of skip lists, such as database indexing or network routing algorithms.

Answers For Check Your Progress

Modules	S. No.	Answers
Module 1	**1.**	C) Search
	2.	C) Maximum element
	3.	C) *O*(log*n*)
	4.	D) Heap
	5.	B) Build a heap from the input data
	6.	B) 2i+1
	7.	D) Rotate
	8.	C) Order of elements
	9.	D) The current node being adjusted
	10.	C) *O*(log*n*)
Module 2	**1.**	C) Map
	2.	C) $O(1)$*O*(1)
	3.	B) Sorted Map
	4.	B) $O(\log n)$*O*(log*n*)
	5.	C) Set
	6.	B) Multi Map
	7.	B) Fast average-case lookups
	8.	C) Skip List
	9.	D) Multi Set
	10.	B) Sorted Map
	11.	C) Sorted traversal
	12.	B) To reduce the time complexity of operations
	13.	C) Hash Table
	14.	C) $O(n)$*O*(*n*)
	15.	C) Sorted Map
	16.	C) It is designed to store unique elements
	17.	B) Chaining

	18.	D) A single key can map to multiple values
	19.	C) Set
	20.	B) Sorted Map

Suggested Readings

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.
2. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). *Data structures and algorithms in Python*. John Wiley & Sons Ltd.

Open-Source E-Content Links

1. GeeksforGeeks - Priority Queue
2. Khan Academy - Heaps
3. Coursera - Data Structures and Algorithms Specialization
4. GeeksforGeeks - Hash Table
5. GeeksforGeeks - Skip List
6. Khan Academy - Hash Tables
7. Coursera - Data Structures

References

1. Sedgewick, R., & Wayne, K. (2011). *Algorithms*. Addison-wesley professional.
2. Cormen, T. H. (2013). *Algorithms unlocked*. Mit Press.

UNIT V – SEARCH TREES

Search Trees: Binary Search Trees-Balanced Search Trees-AVL Trees-Splay Trees. **Sorting and Selection:** Merge Sort - Quick sort - Sorting through an Algorithmic Lens- Comparing Sorting Algorithms-Selection. **Graph Algorithms:** Graphs-Data Structures for Graphs-Graph Traversals- Shortest Paths-Minimum Spanning Trees.

Search Trees

UNIT OBJECTIVE

The objective of this course is to provide students with a comprehensive understanding of fundamental and advanced data structures and algorithms. Students will learn about binary search trees, balanced search trees, AVL trees, and splay trees, gaining insights into their implementation and use cases. The course will cover sorting algorithms like merge sort and quick sort, examining their performance through an algorithmic lens and comparing them. Additionally, it will explore graph algorithms, including graph representations, traversals, shortest paths, and minimum spanning trees, equipping students with the tools to efficiently solve complex computational problems.

SECTION 5.1: SEARCH TREES

5.1.1 Search Trees

Search Trees

Search trees are a fundamental data structure in computer science used to store and manage information in a way that enables efficient search, insertion, deletion, and traversal operations.

- Binary Search Tree (BST)
- Balanced Binary Search Trees
- AVL Trees
- Splay Trees

5.1.2 BINARY SEARCH TREES

A Binary Search Tree (BST) is a tree data structure where each node contains a key, and has at most two children, referred to as the left child and the right child. BSTs are used to store and manage data in a sorted order, allowing for efficient insertion, deletion, and search operations.

Properties:

Node Structure:

- Each node has a key and pointers to its left and right children.
- Optionally, nodes can also contain a value or additional data.

Binary Search Tree Property:

- For any given node, all keys in its left subtree are less than the node's key.
- All keys in its right subtree are greater than the node's key.

Traversal Orders:

- In-order Traversal: Visits nodes in ascending order of their keys (left, root, right).
- Pre-order Traversal: Visits the root node first, followed by the left subtree, then the right subtree.
- Post-order Traversal: Visits the left subtree, the right subtree, and then the root node.

Operations:

Search

1.Objective: Find a node with a specific key.

2.Algorithm:

- Start at the root.
- Compare the key with the current node's key.
- If the key is equal, return the node.
- If the key is less, move to the left child.
- If the key is greater, move to the right child.

3.Time Complexity: O(h), where h is the height of the tree.

Insertion

1.Objective: Insert a new node with a given key.

2.Algorithm:

- Start at the root.
- Compare the new key with the current node's key.
- Move to the left child if the new key is less, or to the right child if the new key is greater.
- Repeat until finding an appropriate null child position to insert the new node.

3.Time Complexity: O(h), where h is the height of the tree.

Deletion

1.Objective: Remove a node with a specific key.

2.Algorithm:

- Search for the node to be deleted.
- If the node has no children, simply remove it.
- If the node has one child, replace it with its child.
- If the node has two children, find its in-order successor (smallest node in the right subtree), replace the node's key with the in-order successor's key, and delete the in-order successor.

3.Time Complexity: O(h), where h is the height of the tree.

Balancing Issues

- **Unbalanced Tree:** If insertions and deletions are not balanced, the tree can become skewed, resembling a linked list, leading to O(n) time complexity for operations.
- **Balancing:** To avoid performance degradation, self-balancing trees such as AVL trees or Red-Black trees are used to maintain a balanced structure, ensuring O(log n) time complexity for operations.

Example

Consider a BST with the following sequence of insertions: 15, 10, 20, 8, 12, 17, 25.

- **Insertion:** Starting from the root, place each number in the appropriate position according to the BST property.
- **Tree Structure:**

```
markdown
    15
   /  \
  10   20
 / \   / \
8  12 17 25
```

- **Search for 12:** Start at the root (15), move left to 10, then move right to 12. Node found.
- **Delete 20:** Node 20 has two children. Find in-order successor (25), replace 20 with 25, and delete the original 25 node.

Applications

- **Databases:** Efficiently store and retrieve records.
- **File Systems:** Organize and manage files hierarchically.
- **Networking:** Routing tables to manage routing paths.
- **Game Development:** Manage game states and decision trees.

Understanding and implementing BSTs are fundamental for efficient data management and form the basis for more advanced data structures and algorithms in computer science.

5.1.3 Balanced Binary Search Trees

Balanced Binary Search Trees:

Balanced Binary Search Trees are a class of Binary Search Trees (BSTs) that automatically maintain their height to ensure optimal performance for operations such as insertion, deletion, and search. This is achieved through various rebalancing techniques, which prevent the tree from becoming skewed and thus guarantee logarithmic time complexity for these operations.

5.1.4 AVL Trees and Red Black Trees

Types of Balanced Binary Search Trees:

1. AVL Tree:

An AVL Tree is a self-balancing BST where the heights of the two child subtrees of any node differ by at most one. If at any time they differ by more than one, rebalancing is performed through rotations.

Properties:

- Each node stores its height.
- The balance factor (difference in heights of left and right subtrees) of any node is -1, 0, or 1.

Rotations:

Single Rotation:

- Left Rotation: Performed when the right subtree is higher.

- Right Rotation: Performed when the left subtree is higher.

Double Rotation:

- Left-Right Rotation: Performed when the left subtree of the right child is higher.
- Right-Left Rotation: Performed when the right subtree of the left child is higher.

Time Complexity: O(log n) for search, insertion, and deletion.

Example:

Consider inserting 10, 20, 30 into an empty AVL tree:

- Insert 10: Tree is balanced.
- Insert 20: Tree is still balanced.
- Insert 30: Tree becomes unbalanced. Perform a left rotation on 10.

2. Red-Black Tree:

A Red-Black Tree is a BST where each node has an extra bit for color (red or black) that ensures the tree remains approximately balanced.

Properties:

- Each node is either red or black.
- The root is always black.
- Red nodes cannot have red children (no two red nodes can be adjacent).
- Every path from a node to its descendant NULL nodes must have the same number of black nodes.

Balancing:

- Insertions and deletions are followed by recoloring and rotations to maintain balance.

Time Complexity: O(log n) for search, insertion, and deletion.

Example:

Consider inserting 10, 20, 30 into an empty Red-Black tree:

- Insert 10: Color it black (root must be black).
- Insert 20: Color it red.
- Insert 30: Color it red, leading to two consecutive red nodes. Recolor and perform a left rotation on 10.

3. B-Tree:

A B-Tree is a generalization of a BST that can have more than two children. It is designed to work well on storage systems that read and write large blocks of data.

Properties:

- Nodes can have multiple keys.
- All leaves are at the same level.
- A B-Tree of order m can have at most m children and at least ⌈m/2⌉ children.

Balancing:

Ensured through splitting and merging nodes during insertion and deletion.

Time Complexity: O(log n) for search, insertion, and deletion.

Example:

Consider a B-Tree of order 3 (each node can have at most 2 keys):

- Insert 10, 20, 30, 40, 50.
- When inserting 30, the node (10, 20) is full, so it splits into two nodes and 20 is promoted to the parent node.

Balanced Binary Search Trees provide efficient means for maintaining sorted data and ensuring that operations such as insertion, deletion, and search remain performant.

Splay Trees:

A Splay Tree is a self-adjusting binary search tree that performs splay operations to move recently accessed elements to the root. This structure helps to keep frequently accessed elements near the top of the tree, potentially improving access times for certain usage patterns.

Key Concepts:

Splaying:

- When a node is accessed, it is moved to the root of the tree through a series of tree rotations.
- The goal is to bring the accessed node closer to the root, improving future access times for that node.

Tree Rotations:

- Zig: Single rotation performed when the node to splay is a child of the root.

- Zig-Zig: Double rotation performed when the node to splay and its parent are both left or both right children.
- Zig-Zag: Double rotation performed when the node to splay is a left child and its parent is a right child, or vice versa.

Operations

Search:

Objective: Find a node with a specific key and move it to the root.

Algorithm:

- Perform standard BST search.
- Splay the node to the root if found.

Time Complexity: Amortized O(log n).

Insertion:

Objective: Insert a new node and splay it to the root.

Algorithm:

- Perform standard BST insertion.
- Splay the newly inserted node to the root.

Time Complexity: Amortized O(log n).

Deletion

Objective: Remove a node with a specific key.

Algorithm:

- Splay the node to be deleted to the root.
- Remove the root node.
- If the tree had a left subtree, splay the maximum node of the left subtree to the root and attach the right subtree.

Time Complexity: Amortized O(log n).

Properties:

- Self-Adjusting: Automatically moves frequently accessed elements closer to the root.
- Amortized Efficiency: While a single operation can take O(n) in the worst case, a sequence of m operations takes O(m log n) time.
- No Balance Information: Unlike AVL or Red-Black trees, splay trees do not store additional information like balance factors or colors.

Advantages:

- Simplicity: No need to maintain balance factors or colors.
- Amortized Performance: Guarantees good performance over a sequence of operations.
- Locality of Reference: Frequently accessed elements are faster to access after their first access.

Disadvantages:

- Worst-Case Performance: Individual operations can be slow (O(n)) in the worst case.
- Reorganization Overhead: The process of splaying can be costly, especially for uniformly distributed access patterns.

Example

Consider a splay tree with the following sequence of operations:

1.Insert 10, 20, 30:

- Insert 10: Root is 10.
- Insert 20: Tree becomes:

markdown

```
  10
    \
     20
```

Splay 20 to root:

```
  20
 /
10
```

Insert 30: Tree becomes:

```
  20
 /  \
10   30
```

Splay 30 to root:

```
  30
 /
 20
```

```
/
10
```

2.Search for 10:

Find 10 and splay it to the root:

```
markdown
  10
    \
    30
  /
20
```

3.Delete 20:

Splay 20 to the root:

```
  20
/  \
10 30

diff
- Remove 20:
10
30
```

Applications:

Splay trees are suitable for applications where:

- ➢ The access pattern is highly non-uniform, with some elements being accessed much more frequently than others.
- ➢ Simplicity and amortized performance are more critical than guaranteed worst-case performance.

Common use cases include caches, where recently accessed items are likely to be accessed again soon, and certain memory management systems.

Let Us Sum Up

Search Trees, including Binary Search Trees (BSTs), Balanced Search Trees, AVL Trees, and Splay Trees, are data structures designed for efficient searching, insertion, and deletion operations. Binary Search Trees ensure logarithmic time complexity for average-case operations but may become unbalanced, leading to worst-case linear time complexity. Balanced Search Trees like AVL Trees maintain balance through rotations, ensuring logarithmic time complexity for all operations. Splay Trees adaptively adjust the tree structure based on recent accesses, optimizing for frequently accessed elements at the root, offering amortized logarithmic time complexity for operations and flexibility in various access patterns. These trees play crucial roles in maintaining efficient search capabilities across different applications and data structures.

Check Your Progress

1. What is the worst-case time complexity for searching in a Binary Search Tree (BST)?
   - A) $O(1)$
   - B) $O(\log n)$
   - C) $O(n)$
   - D) $O(n\log n)$
2. Which type of tree is specifically designed to maintain balance to ensure optimal performance for all operations?
   - A) Binary Search Tree (BST)
   - B) AVL Tree
   - C) Splay Tree
   - D) Heap Tree
3. What is the main advantage of using an AVL Tree over a regular Binary Search Tree (BST)?
   - A) Simpler implementation
   - B) Faster search operations
   - C) Guaranteed logarithmic time complexity for all operations
   - D) Better memory utilization
4. Which operation is not typically supported by a Binary Search Tree (BST)?
   - A) Insertion
   - B) Deletion
   - C) Rotation
   - D) Search
5. Which type of tree adapts its structure based on recent access patterns, optimizing frequently accessed elements at the root?

A) Binary Search Tree (BST)

B) Balanced Search Tree

C) AVL Tree

D) Splay Tree

6. In an AVL Tree, what kind of rotations are performed to maintain balance after an insertion or deletion?

   A) Left rotation

   B) Right rotation

   C) Both left and right rotations

   D) No rotations are performed

7. What is the maximum height of an AVL Tree with *n* nodes?

   A) $O(1)$

   B) $O(\log n)$

   C) $O(n)$

   D) $O(n2)$

8. Which type of tree is known for its self-adjusting behavior, where frequently accessed elements move closer to the root?

   A) Binary Search Tree (BST)

   B) Balanced Search Tree

   C) AVL Tree

   D) Splay Tree

9. Which operation in a Splay Tree adapts the tree structure based on recent accesses?

   A) Split

   B) Join

   C) Splay

   D) Balance

10. What is the amortized time complexity for search, insertion, and deletion in a Splay Tree?

    A) $O(1)$

    B) $O(\log n)$

    C) $O(n)$

    D) $O(n\log n)$

SECTION 5.2: SORTING AND SELECTION

Sorting and Selection:

Sorting is a fundamental operation in computer science, where the goal is to arrange elements in a specific order, typically ascending or descending. Here are some common sorting algorithms, categorized by their approach and performance characteristics.

Comparison-Based Sorting Algorithms:

1.Bubble Sort:

- Description: Repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. This process is repeated until the list is sorted.
- Time Complexity: O(n^2) in the worst and average cases, O(n) in the best case (when the list is already sorted).
- Space Complexity: O(1) auxiliary space.
- Stability: Yes.

2.Selection Sort:

- Description: Divides the list into a sorted and an unsorted region. Repeatedly selects the smallest (or largest) element from the unsorted region and moves it to the end of the sorted region.
- Time Complexity: O(n^2) for all cases.
- Space Complexity: O(1) auxiliary space.
- Stability: No.

3.Insertion Sort:

- Description: Builds the sorted array one element at a time by repeatedly taking the next element and inserting it into its correct position.
- Time Complexity: O(n^2) in the worst case, O(n) in the best case.
- Space Complexity: O(1) auxiliary space.
- Stability: Yes.

5.2.1 – Merge Sort

Merge Sort:

Merge Sort is a comparison-based, divide-and-conquer sorting algorithm. It divides the input array into two halves, recursively sorts each half, and then merges the two sorted halves to produce the sorted array.

How Merge Sort Works

1. Divide: Split the array into two halves.
2. Conquer: Recursively sort each half.
3. Combine: Merge the two sorted halves to produce the sorted array.

Detailed Steps

1.Splitting the Array:

- If the array has one or zero elements, it is already sorted.
- Otherwise, split the array into two halves.

2.Recursive Sorting:

- Recursively sort each half using merge sort.

3.Merging: Merge the two sorted halves into a single sorted array. This involves comparing the smallest remaining elements of each half and appending the smaller element to the result array.

Merge Sort Algorithm

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2  # Find the middle point
        L = arr[:mid]        # Divide the array elements into 2 halves
        R = arr[mid:]

        merge_sort(L)        # Sort the first half
        merge_sort(R)        # Sort the second half
          i = j = k = 0
          # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
                # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
          while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
# Example usage
arr = [12, 11, 13, 5, 6, 7]
merge_sort(arr)
print("Sorted array is:", arr)
```

Time and Space Complexity

1.Time Complexity: O(n log n) for all cases (best, average, worst).

This is because the array is divided in half log n times, and merging the halves takes linear time.

2.Space Complexity: O(n) auxiliary space.

Merge sort requires additional space proportional to the size of the input array to hold the two halves and the merged array.

Example

Consider the array: [12, 11, 13, 5, 6, 7]

1.Divide:

Split into two halves: [12, 11, 13] and [5, 6, 7]

2.Recursive Sorting:

- [12, 11, 13]
- Split into [12] and [11, 13]
- Sort [11, 13]
- Split into [11] and [13]
- Merge [11] and [13] to get [11, 13]
- Merge [12] and [11, 13] to get [11, 12, 13]
- Sort [5, 6, 7]
- Split into [5] and [6, 7]

- Sort [6, 7]
- Split into [6] and [7]
- Merge [6] and [7] to get [6, 7]
- Merge [5] and [6, 7] to get [5, 6, 7]

3.Merge:

Merge [11, 12, 13] and [5, 6, 7] to get [5, 6, 7, 11, 12, 13]

Advantages of Merge Sort

- Consistent Time Complexity: Always O(n log n), regardless of the input data distribution.
- Stable Sort: Maintains the relative order of equal elements.
- Parallelizable: Can be easily parallelized as the merge operations on different halves are independent.

Disadvantages of Merge Sort

- Space Complexity: Requires O(n) additional space, which may be a disadvantage for large datasets or memory-constrained environments.
- Overhead: The recursive calls and additional array allocations can introduce overhead compared to in-place sorting algorithms like quicksort.

Use Cases

Merge Sort is particularly useful when:

- The data is too large to fit into memory, and external sorting is needed.
- Stability is important, and the relative order of equal elements must be preserved.
- Consistent performance is required, regardless of the input data's initial order.
-

5.2.2 – Merge Sort

Quick Sort:

Quick Sort is a highly efficient and commonly used comparison-based sorting algorithm. It follows the divide-and-conquer paradigm and works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

How Quick Sort Works

1.Partition: Choose a pivot element and partition the array into two halves:

Elements less than the pivot.

Elements greater than or equal to the pivot.

2.Recursively Sort: Apply the same procedure to the two halves.

3.Combine: Since the arrays are sorted in place, no merging is required.

Detailed Steps

1.Choosing a Pivot:

Common strategies include selecting the first element, the last element, a random element, or the median.

2.Partitioning the Array:

- Rearrange elements in the array such that elements less than the pivot come before it, and elements greater than the pivot come after it.
- Return the index of the pivot after partitioning.

3.Recursive Sorting:

Recursively apply the above steps to the sub-arrays formed by partitioning.

Quick Sort Algorithm

```
def quick_sort(arr):
    quick_sort_helper(arr, 0, len(arr) - 1)
def quick_sort_helper(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quick_sort_helper(arr, low, pi - 1)
        quick_sort_helper(arr, pi + 1, high)
def partition(arr, low, high):
    pivot = arr[high]  # Choosing the last element as the pivot
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
# Example usage
arr = [10, 7, 8, 9, 1, 5]
```

```
    quick_sort(arr)
    print("Sorted array is:", arr)
```

Time and Space Complexity:

Time Complexity:

- Average Case: O(n log n), which is why quick sort is preferred for many practical applications.
- Best Case: O(n log n), occurs when the pivot always splits the array into two nearly equal halves.
- Worst Case: O(n^2), occurs when the pivot is the smallest or largest element in each partition (e.g., a sorted array).
- Space Complexity: O(log n) auxiliary space due to the recursion stack (in-place sorting).

Example

Consider the array: [10, 7, 8, 9, 1, 5]

1.Choose Pivot: Let's choose the last element, 5, as the pivot.

2.Partitioning:

- Rearrange elements such that all elements less than 5 are on the left and all elements greater than or equal to 5 are on the right.
- After partitioning: [1, 5, 8, 9, 10, 7], pivot index is 1.

3.Recursive Sort:

Apply quick sort on sub-arrays [1] and [8, 9, 10, 7].

Advantages of Quick Sort:

- Efficient Average Performance: Quick sort is generally faster in practice than other O(n log n) algorithms like merge sort and heap sort due to lower constant factors and cache efficiency.
- In-Place Sorting: Requires only a small, constant amount of additional storage space.
- Simple Implementation: The basic idea is straightforward and easy to implement.

Disadvantages of Quick Sort

- Worst-Case Performance: O(n^2) time complexity, though this can be mitigated by using strategies like random pivot selection or median-of-three.
- Unstable Sort: Does not preserve the relative order of equal elements.

Optimizations

1.Randomized Quick Sort: Randomly select the pivot to avoid worst-case scenarios.

2.Median-of-Three: Choose the pivot as the median of the first, middle, and last elements to improve partitioning.

3.Tail Recursion Elimination: Convert tail recursion to iteration to reduce the recursion stack depth.

4.Hybrid Algorithms: Switch to insertion sort for small sub-arrays to improve performance.

Use Cases

Quick Sort is well-suited for large datasets where average-case performance is crucial, and in-place sorting is necessary. It is widely used in systems and applications where quick, efficient sorting is required, such as in database query optimizations and file sorting.

5.2.3 –Sorting through Algorithmic Lens

Sorting through an Algorithmic Lens:

Sorting algorithms can be analyzed through various algorithmic lenses to understand their performance, efficiency, and behavior in different contexts. This approach involves examining their time and space complexities, stability, adaptability, and the underlying principles they use. Here's an in-depth look at sorting algorithms through these different algorithmic perspectives:

1. Time Complexity:

Time complexity measures how the execution time of an algorithm scales with the size of the input.

- Bubble Sort: O(n^2) in the worst and average cases, O(n) in the best case (already sorted).
- Selection Sort: O(n^2) for all cases.
- Insertion Sort: O(n^2) in the worst case, O(n) in the best case (nearly sorted).
- Merge Sort: O(n log n) for all cases.
- Quick Sort: O(n log n) on average, O(n^2) in the worst case.
- Heap Sort: O(n log n) for all cases.
- Counting Sort: O(n + k), where k is the range of the input.

- Radix Sort: O(nk), where k is the number of digits.
- Bucket Sort: O(n + k), where k is the number of buckets.

2. Space Complexity:

Space complexity measures the amount of memory an algorithm needs in addition to the input data.

- Bubble Sort, Selection Sort, Insertion Sort: O(1) auxiliary space (in-place).
- Merge Sort: O(n) auxiliary space.
- Quick Sort: O(log n) auxiliary space (due to recursion).
- Heap Sort: O(1) auxiliary space.
- Counting Sort, Radix Sort, Bucket Sort: O(n + k) auxiliary space.

3. Stability

A sorting algorithm is stable if it maintains the relative order of equal elements.

- Stable: Bubble Sort, Insertion Sort, Merge Sort, Counting Sort, Radix Sort, Bucket Sort.
- Unstable: Selection Sort, Quick Sort, Heap Sort (though they can be modified to be stable).

4. Adaptability

- Adaptive algorithms take advantage of existing order in the input data to improve performance.
- Adaptive: Insertion Sort (efficient for nearly sorted data), Bubble Sort (with optimizations).
- Non-Adaptive: Selection Sort, Merge Sort, Quick Sort, Heap Sort, Counting Sort, Radix Sort, Bucket Sort.

5.Divide-and-Conquer

Divide-and-conquer algorithms break the problem into smaller subproblems, solve them independently, and combine their solutions.

- Merge Sort: Divides the array into halves, recursively sorts each half, and merges them.
- Quick Sort: Divides the array based on a pivot, recursively sorts the partitions.
- Heap Sort: Does not explicitly use divide-and-conquer but uses a heap data structure for sorting.

6. Comparison-Based vs. Non-Comparison-Based

Comparison-based algorithms compare elements to determine order, whereas non-comparison-based algorithms use other means (like counting).

- Comparison-Based: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort.
- Non-Comparison-Based: Counting Sort, Radix Sort, Bucket Sort.

7. Practical Considerations

- Merge Sort is ideal for sorting linked lists and external sorting (due to its stability and consistent O(n log n) time complexity).
- Quick Sort is often preferred for in-memory sorting due to its efficient average-case performance and low overhead.
- Heap Sort is useful when in-place sorting with guaranteed O(n log n) time is needed.
- Counting Sort, Radix Sort, Bucket Sort are effective for sorting integers or fixed-size data where range or distribution is known.

8. Algorithmic Improvements and Variants

- Optimized Bubble Sort: Stops early if no swaps are made.
- Optimized Quick Sort: Randomized pivot selection, median-of-three partitioning, tail recursion elimination.
- Hybrid Algorithms: Combining algorithms for better performance, such as using Insertion Sort for small subarrays within Quick Sort or Merge Sort.
-

5.2.4 – Comparing Sorting Algorithms

Comparing Sorting Algorithms:

To compare sorting algorithms effectively, we can evaluate them based on several criteria including time complexity, space complexity, stability, adaptability, and practical performance. Below is a detailed comparison of some common sorting algorithms:

1.Bubble Sort

- Time Complexity:
    - Worst Case: O(n^2)
    - Average Case: O(n^2)

    - Best Case: O(n) (if the array is already sorted)
- Space Complexity: O(1)
- Stability: Yes
- Adaptability: Yes, with an optimized version that stops early if no swaps are made.
- Practical Use: Generally not used in practice due to inefficiency for large datasets.

2.Selection Sort

- Time Complexity:
    - Worst Case: O(n^2)
    - Average Case: O(n^2)
    - Best Case: O(n^2)
- Space Complexity: O(1)
- Stability: No
- Adaptability: No
- Practical Use: Simple to implement but rarely used due to poor performance.

3.Insertion Sort:

- Time Complexity:
    - Worst Case: O(n^2)
    - Average Case: O(n^2)
- Best Case: O(n) (if the array is nearly sorted)
- Space Complexity: O(1)
- Stability: Yes
- Adaptability: Yes
- Practical Use: Efficient for small datasets or nearly sorted arrays; often used as a subroutine in more complex algorithms (e.g., hybrid quick sort).

4.Merge Sort

- Time Complexity:
    - Worst Case: O(n log n)
    - Average Case: O(n log n)
    - Best Case: O(n log n)
- Space Complexity: O(n) auxiliary space

- Stability: Yes
- Adaptability: No
- Practical Use: Useful for large datasets and external sorting; consistent performance.

5.Quick Sort

- Time Complexity:
  - Worst Case: O(n^2) (rare, but can be mitigated with good pivot selection)
  - Average Case: O(n log n)
  - Best Case: O(n log n)
- Space Complexity: O(log n) auxiliary space (due to recursion)
- Stability: No
- Adaptability: No
- Practical Use: Widely used due to its efficient average-case performance and low overhead; often the default sort in many libraries.

6.Heap Sort

- Time Complexity:
  - Worst Case: O(n log n)
  - Average Case: O(n log n)
  - Best Case: O(n log n)
- Space Complexity: O(1)
- Stability: No
- Adaptability: No
- Practical Use: Used when in-place sorting with guaranteed O(n log n) performance is required; not as cache-friendly as quick sort.

7.Counting Sort

- Time Complexity: O(n + k), where k is the range of the input values
- Space Complexity: O(k)
- Stability: Yes
- Adaptability: No
- Practical Use: Efficient for sorting integers or data with a small range; not suitable for large ranges or non-integer data.

8.Radix Sort

- Time Complexity: O(nk), where k is the number of digits
- Space Complexity: O(n + k)
- Stability: Yes
- Adaptability: No
- Practical Use: Useful for sorting large numbers or strings where the number of digits/characters (k) is relatively small compared to n.

9.Bucket Sort

- Time Complexity: O(n + k), where k is the number of buckets
- Space Complexity: O(n + k)
- Stability: Yes
- Adaptability: No
- Practical Use: Effective when the input is uniformly distributed over a range.

Comparison Summary

Efficiency:

- For general-purpose, in-memory sorting: Quick Sort (average case O(n log n)) is usually preferred.
- For external sorting and linked lists: Merge Sort (O(n log n)) is preferred due to its stable and consistent performance.
- For small arrays or nearly sorted data: Insertion Sort (O(n) best case) is efficient.
- When space is constrained: Heap Sort (O(1) space) is a good choice.
- For specific integer ranges or digit-based sorting: Counting Sort, Radix Sort, and Bucket Sort are very efficient.

Stability:

- If stability is required: Merge Sort, Bubble Sort, Counting Sort, Radix Sort, and Bucket Sort are suitable.
- If stability is not critical: Quick Sort and Heap Sort are often chosen for their performance benefits.

Adaptability:

- Adaptive algorithms like Insertion Sort and optimized Bubble Sort are useful for nearly sorted data.

- Practical Considerations
- Merge Sort is particularly useful for large datasets and external sorting.
- Quick Sort is favored for its efficient average-case performance and low overhead, making it the default in many libraries.
- Insertion Sort is efficient for small or nearly sorted datasets and is often used in hybrid algorithms.
- Heap Sort is used for in-place sorting with guaranteed O(n log n) performance.
- Counting, Radix, and Bucket Sorts are non-comparison-based and very efficient for specific types of data, particularly when sorting integers or fixed-size keys.

Let Us Sum Up

Sorting and selection algorithms play a pivotal role in organizing and managing data efficiently. Merge Sort employs a divide-and-conquer strategy, breaking down the array into smaller segments, sorting them recursively, and then merging them, ensuring a time complexity of $O(n\log n)$. Quick Sort, another popular method, partitions the array around a pivot element, recursively sorting each partition, and then combining them, achieving an average-case time complexity of $O(n\log n)$. Analyzing sorting algorithms involves considerations like time complexity, space complexity, stability, and adaptability to different input types. Through this lens, algorithms are compared based on their performance in worst-case, average-case, and best-case scenarios, as well as their behavior on partially sorted or nearly sorted data sets. Additionally, selection algorithms like Quick Select extend the principles of Quick Sort to efficiently find the k*k*th smallest or largest element in an array, offering solutions with an average time complexity of $O(n)O(n)$. Mastery of these algorithms equips practitioners with powerful tools for effective data manipulation and retrieval across various computational tasks.

Check Your Progress

1. Which sorting algorithm has a time complexity of $O(n\log n)$ in all cases?
   A) Bubble Sort
   B) Merge Sort
   C) Insertion Sort
   D) Selection Sort
2. In Quick Sort, which element is typically chosen as the pivot?

A) First element

B) Last element

C) Random element

D) Middle element

3. Which sorting algorithm exhibits a divide-and-conquer strategy?

   A) Quick Sort

   B) Bubble Sort

   C) Insertion Sort

   D) Selection Sort

4. Which property is desirable in a sorting algorithm to maintain the order of equal elements?

   A) Space complexity

   B) Time complexity

   C) Stability

   D) Adaptability

5. Which sorting algorithm has the best average-case time complexity?

   A) Quick Sort

   B) Merge Sort

   C) Bubble Sort

   D) Selection Sort

6. What is the worst-case time complexity of Bubble Sort?

   A) $O(n)$

   B) $O(n\log n)$

   C)$O(n2)$

   D)$O(1)$

7. Which sorting algorithm is well-suited for partially sorted or nearly sorted arrays?

   A) Merge Sort

   B) Quick Sort

   C) Bubble Sort

   D) Insertion Sort

8. In Quick Select, what is the average-case time complexity for finding the kkth smallest element in an array?

   A) $O(1)$

B) $O(n)$

C) $O(n\log n)$

D) $O(n2)$

9. Which selection algorithm extends the principles of Quick Sort to find the kkth smallest or largest element in an array?

A) Merge Select

B) Heap Select

C) Quick Select

D) Bubble Select

10. Which characteristic is common to both Merge Sort and Quick Sort?

A) In-place sorting

B) Stable sorting

C) Divide-and-conquer strategy

D) Linear time complexity

11. Which sorting algorithm can be implemented efficiently using recursion?

A) Bubble Sort

B) Insertion Sort

C) Merge Sort

D) Selection Sort

12. Which sorting algorithm involves repeatedly swapping adjacent elements if they are in the wrong order?

A) Bubble Sort

B) Quick Sort

C) Merge Sort

D) Insertion Sort

13. Which sorting algorithm exhibits the best performance on large datasets with random elements?

A) Bubble Sort

B) Insertion Sort

C) Quick Sort

D) Selection Sort

14. Which sorting algorithm inherently maintains stability?

A) Merge Sort

B) Quick Sort

C) Bubble Sort

D) Selection Sort

15. Which sorting algorithm is based on the concept of partitioning the array around a pivot element?

A) Merge Sort

B) Quick Sort

C) Bubble Sort

D) Insertion Sort

SECTION 5.3: THE GRAPH ALGORITHMS

Graph Algorithms

Graph algorithms are a set of instructions that traverse (visit nodes) and process graphs. Graphs are abstract data types that represent a set of objects (nodes or vertices) where some pairs of objects are connected by links (edges).

5.3.1 – Graph Data Structures

Graphs-Data Structures for Graphs:

Graphs can be represented using various data structures, each with its own advantages and disadvantages. The choice of data structure often depends on the specific requirements of the application, such as the type of operations performed on the graph, memory efficiency, and the nature of the graph itself (sparse or dense). Here are some commonly used data structures for representing graphs:

1. Adjacency Matrix

An adjacency matrix is a 2D array where each cell matrix[i][j] represents the presence (and possibly weight) of an edge between vertex i and vertex j. For unweighted graphs, the cell value may be 1 to indicate the presence of an edge, and 0 otherwise. For weighted graphs, the cell value may represent the weight of the edge.

- Space Complexity: O(V^2), where V is the number of vertices.
- Pros:
  - Simple and intuitive representation.
  - Efficient for dense graphs.
- Cons:
  - Consumes a lot of space for sparse graphs.

- Inefficient for graphs with many isolated vertices.

2. Adjacency List

An adjacency list is a collection of lists or arrays where each list adj_list[i] represents the neighbors of vertex i. Each element in the list may also contain information about the edge weight.

- Space Complexity: O(V + E), where E is the number of edges.
- Pros:
    - More space-efficient for sparse graphs.
    - Efficient for graph traversal algorithms.
- Cons:
    - Slower for certain operations like checking edge presence between two vertices.

3. Edge List

An edge list is a list of all edges in the graph, where each edge is represented by a pair (or tuple) of vertices (and possibly the edge weight). This data structure is commonly used in algorithms that process edges one by one.

- Space Complexity: O(E), where E is the number of edges.
- Pros:
    - Very space-efficient, especially for graphs with relatively few edges.
    - Suitable for algorithms that process edges individually.
- Cons:
    - Inefficient for operations that require adjacency checks or vertex degree calculations.

4. Incidence Matrix

An incidence matrix is a 2D array where each row represents a vertex and each column represents an edge. The entry matrix[i][j] is 1 if vertex i is incident to edge j, -1 if vertex i is the tail of edge j, and 0 otherwise. This representation is primarily used in directed graphs.

- Space Complexity: O(V * E), where V is the number of vertices and E is the number of edges.
- Pros:
    - Can efficiently represent directed graphs.

- Cons:
    - Consumes a lot of space, especially for dense graphs.
    - Less commonly used compared to adjacency matrix and adjacency list.

5. Combinations and Hybrid Structures

In practice, hybrid data structures or combinations of the above representations may be used to balance memory usage and computational efficiency. For example, a graph may be represented using an adjacency list for most operations, but an adjacency matrix may be used for quick edge lookups in certain scenarios.

5.3.2 – Graph Traversals

Graph Traversals:

Graph traversal algorithms are used to visit all the vertices and edges of a graph systematically. There are two main approaches to graph traversal: breadth-first search (BFS) and depth-first search (DFS). Each approach has its advantages and is suited to different types of problems.

Breadth-First Search (BFS):

Breadth-first search explores the neighbor nodes level by level, starting from a chosen source vertex. It visits all the vertices at the present depth before moving on to the vertices at the next depth.

Algorithm:

- Choose a starting vertex (source).
- Enqueue the starting vertex into a queue and mark it as visited.
- While the queue is not empty:
    - Dequeue a vertex from the queue.
    - Visit the dequeued vertex.
    - Enqueue all the adjacent vertices of the dequeued vertex that have not been visited and mark them as visited.
- Repeat step 3 until the queue is empty.

Application:

- Shortest path in unweighted graphs
- Level order traversal of a tree or graph
- Finding connected components

Python Implementation:

```
from collections import deque
def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)
    while queue:
        vertex = queue.popleft()
        print(vertex, end=" ")
        for neighbor in graph[vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
bfs(graph, 'A')
```

Depth-First Search (DFS)

Depth-first search explores as far as possible along each branch before backtracking. It traverses one branch of the graph as deeply as possible before exploring the next branch.

Algorithm:

1. Choose a starting vertex (source).
2. Mark the starting vertex as visited.
3. Recursively visit all unvisited neighbors of the current vertex.
4. Repeat step 3 until all vertices are visited.

Application:

1. Detecting cycles in a graph
2. Topological sorting
3. Finding connected components

Python Implementation:

```
def dfs(graph, start, visited=None):
   if visited is None:
      visited = set()
   visited.add(start)
   print(start, end=" ")
   for neighbor in graph[start]:
      if neighbor not in visited:
         dfs(graph, neighbor, visited)
# Example usage (same graph as before)
dfs(graph, 'A')
```

- ➢ Breadth-first search (BFS) systematically explores all the vertices level by level, while depth-first search (DFS) explores as far as possible along each branch before backtracking.
- ➢ BFS is suitable for finding the shortest path in unweighted graphs and for level-order traversal, while DFS is useful for detecting cycles and topological sorting.
- ➢ Both BFS and DFS have a time complexity of O(V + E), where V is the number of vertices and E is the number of edges, making them efficient for most practical purposes.

  Shortest path algorithms are used to find the shortest path between two vertices in a graph. The "shortest" path can be defined based on various criteria such as the sum of edge weights or the number of edges.

Shortest Paths:

Shortest path algorithms are used to find the shortest path between two vertices in a graph. The "shortest" path can be defined based on various criteria such as the sum of edge weights or the number of edges.

Dijkstra's Algorithm

Dijkstra's algorithm is used to find the shortest path from a single source vertex to all other vertices in a weighted graph with non-negative weights.

Algorithm:

1.Initialize distances to all vertices as infinity, except for the source vertex, which is set to 0.

2.Create a priority queue (min-heap) and insert the source vertex with distance 0.

3.While the priority queue is not empty:

- Extract the vertex with the minimum distance from the priority queue.
- Update the distances of its neighbors if a shorter path is found.
- Enqueue the updated neighbors into the priority queue.

4.Repeat step 3 until all vertices are visited.

Time Complexity:

- O((V + E) log V), where V is the number of vertices and E is the number of edges.

Python Implementation:

```
import heapq
def dijkstra(graph, start):
    min_heap = [(0, start)]
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0
    while min_heap:
        current_distance, current_vertex = heapq.heappop(min_heap)
        if current_distance > distances[current_vertex]:
            continue
        for neighbor, weight in graph[current_vertex]:
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(min_heap, (distance, neighbor))
    return distances
# Example usage
graph = {
    'A': [('B', 4), ('C', 2)],
    'B': [('A', 4), ('D', 5)],
    'C': [('A', 2), ('D', 7)],
    'D': [('B', 5), ('C', 7)]
}
shortest_distances = dijkstra(graph, 'A')
print(shortest_distances)
```

Bellman-Ford Algorithm:

Bellman-Ford algorithm is used to find the shortest paths from a single source vertex to all other vertices in a graph that may have negative weight edges.

Algorithm:

1. Initialize distances to all vertices as infinity, except for the source vertex, which is set to 0.
2. Relax all edges V-1 times, where V is the number of vertices.
3. Check for negative weight cycles.

Time Complexity:

O(VE), where V is the number of vertices and E is the number of edges.

Python Implementation:

```
def bellman_ford(graph, start):
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0
        for _ in range(len(graph) - 1):
        for vertex in graph:
            for neighbor, weight in graph[vertex]:
                if distances[vertex] + weight < distances[neighbor]:
                    distances[neighbor] = distances[vertex] + weight
      # Check for negative weight cycles
    for vertex in graph:
        for neighbor, weight in graph[vertex]:
            if distances[vertex] + weight < distances[neighbor]:
                raise ValueError("Graph contains a negative weight cycle")
      return distances
# Example usage
graph = {
    'A': [('B', 4), ('C', 2)],
    'B': [('A', -1), ('D', 5)],
    'C': [('A', 2), ('D', 7)],
    'D': [('B', 5), ('C', 7)]
}
shortest_distances = bellman_ford(graph, 'A')
print(shortest_distances)
```

Floyd-Warshall Algorithm

Floyd-Warshall algorithm is used to find shortest paths between all pairs of vertices in a weighted graph.

Algorithm:

1. Initialize the shortest distance matrix with the edge weights if an edge exists, or infinity otherwise.
2. Update the shortest distance matrix by considering all intermediate vertices.

Time Complexity:

O(V^3), where V is the number of vertices.

Python Implementation:

```
def floyd_warshall(graph):
    dist = [[float('infinity')] * len(graph) for _ in range(len(graph))]
    for i in range(len(graph)):
        for j in range(len(graph)):
            if i == j:
                dist[i][j] = 0
            elif graph[i][j] != 0:
                dist[i][j] = graph[i][j]
    for k in range(len(graph)):
        for i in range(len(graph)):
            for j in range(len(graph)):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
    return dist
# Example usage
graph = [
    [0, 4, 2, 0],
    [0, 0, 0, 5],
    [0, 0, 0, 7],
    [0, 0, 0, 0]
]
shortest_distances = floyd_warshall(graph)
for row in shortest_distances:
    print(row)
```

Shortest path algorithms provide efficient methods to find the shortest path between vertices in a graph. Dijkstra's algorithm is suitable for graphs with non-negative edge weights, Bellman-Ford algorithm can handle graphs with negative edge weights (but no negative weight cycles), and Floyd-Warshall algorithm finds shortest paths between all pairs of vertices in a graph.

5.3.3 – Minimum Spanning Trees

A Minimum Spanning Tree (MST) is a subset of the edges of a connected, undirected graph that connects all the vertices together without any cycles and with the minimum possible total edge weight. Minimum spanning trees are widely used in network design, clustering, and optimization problems.

Kruskal's Algorithm:

Kruskal's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected, undirected graph.

Algorithm:

- Sort all the edges in non-decreasing order of their weights.
- Initialize an empty minimum spanning tree (MST).
- Iterate through the sorted edges, adding each edge to the MST if it doesn't create a cycle.
- Stop when the MST contains V-1 edges, where V is the number of vertices in the graph.

Time Complexity:

O(E log E), where E is the number of edges.

Python Implementation:

```
class UnionFind:
    def __init__(self, size):
        self.parent = list(range(size))
        self.rank = [0] * size
    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]
    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)
        if root_u != root_v:
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            else:
                self.parent[root_u] = root_v
                if self.rank[root_u] == self.rank[root_v]:
                    self.rank[root_v] += 1
def kruskal(graph):
    edges = sorted(graph['edges'], key=lambda x: x[2])
    uf = UnionFind(graph['vertices'])
    mst = []
    for u, v, weight in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            mst.append((u, v, weight))
    return mst
# Example usage
graph = {
    'vertices': 4,
    'edges': [
        (0, 1, 10),
        (0, 2, 6),
        (0, 3, 5),
        (1, 3, 15),
        (2, 3, 4)
    ]
}
mst = kruskal(graph)
print(mst)
```

Prim's Algorithm

Prim's algorithm is another greedy algorithm that finds a minimum spanning tree for a connected, undirected graph.

Algorithm:

1. Initialize a priority queue (min-heap) with the source vertex and initialize an empty minimum spanning tree (MST).
2. While the priority queue is not empty:
    - Extract the vertex with the minimum key from the priority queue.
    - Add the vertex and its corresponding edge to the MST.
    - Update the keys of adjacent vertices in the priority queue if necessary.
3. Stop when the MST contains V-1 edges, where V is the number of vertices in the graph.

Time Complexity:

O((V + E) log V), where V is the number of vertices and E is the number of edges.

Python Implementation:

```
import heapq
def prim(graph, start):
    mst = []
    visited = set([start])
    edges = [(weight, start, to) for to, weight in graph[start]]
    heapq.heapify(edges)
    while edges:
        weight, frm, to = heapq.heappop(edges)
        if to not in visited:
            visited.add(to)
            mst.append((frm, to, weight))
            for to_next, weight in graph[to]:
                if to_next not in visited:
                    heapq.heappush(edges, (weight, to, to_next))
    return mst
# Example usage (same graph as before)
graph = {
    'A': [('B', 4), ('C', 2)],
    'B': [('A', 4), ('D', 5)],
    'C': [('A', 2), ('D', 7)],
    'D': [('B', 5), ('C', 7)]
}
mst = prim(graph, 'A')
print(mst)
```

Minimum spanning trees are important structures in graph theory and have various applications in network design, clustering, and optimization problems. Kruskal's algorithm and Prim's algorithm are two widely used methods to find the minimum spanning tree of a graph.

Let Us Sum Up

Graph algorithms are fundamental tools for analyzing and manipulating interconnected data structures. Graphs represent relationships between objects, vertices, connected by edges. Data structures for graphs include adjacency matrix and adjacency list, enabling efficient representation and traversal. Graph traversals, such as Depth-First Search (DFS) and Breadth-First Search (BFS), explore and discover vertices and edges. Shortest path algorithms, like Dijkstra's and Bellman-Ford, find the shortest path between two vertices, crucial for network routing and optimization.

Minimum Spanning Tree algorithms, such as Prim's and Kruskal's, construct a subgraph with the minimum total edge weight, useful in network design and clustering. These algorithms are essential for solving various real-world problems efficiently.

Check Your Progress

1. **Which data structure is commonly used to represent graphs?**
   A) Array
   B) Linked List
   C) Hash Table
   D) Tree
2. **What does a vertex represent in a graph?**
   A) A path between two nodes
   B) A city in a transportation network
   C) A connection between two edges
   D) A node or point in the graph
3. **Which graph traversal algorithm explores vertices as far as possible along each branch before backtracking?**
   A) Depth-First Search (DFS)
   B) Breadth-First Search (BFS)
   C) Dijkstra's Algorithm
   D) Prim's Algorithm
4. **Which algorithm is used to find the shortest path between two vertices in a weighted graph?**
   A) Depth-First Search (DFS)
   B) Breadth-First Search (BFS)
   C) Dijkstra's Algorithm
   D) Prim's Algorithm
5. **What is the time complexity of Dijkstra's Algorithm?**
   A) $O(V)$
   B) $O(E)$
   C) $O(V2)$
   D) $O(ElogV)$
6. **Which algorithm is used to find the minimum spanning tree in a graph?**
   A) Depth-First Search (DFS)
   B) Breadth-First Search (BFS)

C) Dijkstra's Algorithm

D) Prim's Algorithm

7. **What is the primary objective of the Minimum Spanning Tree (MST) algorithm?**

   A) To find the shortest path between two vertices

   B) To find the longest path in a graph

   C) To find a subset of edges that connects all vertices with minimum total edge weight

   D) To find the shortest path between all pairs of vertices

8. **Which algorithm selects the edge with the minimum weight connected to the current spanning tree in Prim's Algorithm?**

   A) Depth-First Search (DFS)

   B) Breadth-First Search (BFS)

   C) Dijkstra's Algorithm

   D) Prim's Algorithm

9. **What does the term "spanning tree" mean in the context of graphs?**

   A) A tree that spans all vertices of the graph

   B) A tree with the maximum number of edges

   C) A tree with the minimum number of edges

   D) A tree with only one edge

10. **What is the time complexity of Prim's Algorithm?**

   A) $O(V)$

   B) $O(E)$

   C) $O(V2)$

   D) $O(E\log V)$

11. **What is the time complexity of Breadth-First Search (BFS) in a graph with VV vertices and EE edges?**

   A) $O(V)$

   B) $O(E)$

   C) $O(V+E)$

   D) $O(V\log E)$

12. **Which algorithm is used to detect cycles in a graph?**

   A) Breadth-First Search (BFS)

   B) Depth-First Search (DFS)

C) Dijkstra's Algorithm

D) Prim's Algorithm

13. **In graph theory, what is the degree of a vertex?**

A) The number of edges connected to the vertex

B) The weight assigned to the vertex

C) The number of vertices in the graph

D) The number of cycles passing through the vertex

14. **Which data structure is commonly used to implement Breadth-First Search (BFS)?**

A) Stack

B) Queue

C) Heap

D) Hash Table

15. **What is the main advantage of using an adjacency matrix to represent a graph?**

A) It requires less memory compared to other representations.

B) It provides fast lookup for edge existence.

C) It efficiently supports dynamic resizing.

D) It allows for easy traversal of the graph.

16. **Which of the following statements about Dijkstra's Algorithm is true?**

A) It works correctly for graphs with negative-weight edges.

B) It guarantees the shortest path from the source to all vertices in the graph.

C) It selects the edge with the maximum weight at each step.

D) It has a time complexity of $O(V2)$O(V2).

17. **Which algorithm can be used to find the longest path in a graph?**

A) Breadth-First Search (BFS)

B) Depth-First Search (DFS)

C) Dijkstra's Algorithm

D) Bellman-Ford Algorithm

18. **What is the main disadvantage of using Depth-First Search (DFS) for finding shortest paths in weighted graphs?**

A) It has a time complexity of $O(V\log V)$O(VlogV).

B) It does not guarantee the shortest path.

C) It requires a lot of memory.

D) It is not suitable for dense graphs.

19. **Which graph traversal algorithm is typically used for topological sorting?**

A) Breadth-First Search (BFS)

B) Depth-First Search (DFS)

C) Dijkstra's Algorithm

D) Prim's Algorithm

20. **What does Prim's Algorithm prioritize when selecting edges to add to the minimum spanning tree?**

A) Edges with the maximum weight

B) Edges with the minimum weight

C) Edges with the maximum degree

D) Edges with the minimum degree

21. **Which algorithm can be used to find the shortest path in a graph with negative-weight edges?**

A) Breadth-First Search (BFS)

B) Depth-First Search (DFS)

C) Dijkstra's Algorithm

D) Bellman-Ford Algorithm

22. **What is the time complexity of Bellman-Ford Algorithm?**

A) $O(V)$

B) $O(E)$

C) $O(V2)$

D) $O(VE)$

23. **In a weighted graph, what does the weight of an edge represent?**

A) The distance between two vertices

B) The number of paths between two vertices

C) The number of edges connected to a vertex

D) The probability of traversing the edge

24. **Which algorithm is suitable for finding the shortest path between all pairs of vertices in a graph?**

A) Breadth-First Search (BFS)

B) Depth-First Search (DFS)

C) Dijkstra's Algorithm

D) Floyd-Warshall Algorithm

25. **What is the purpose of the Floyd-Warshall Algorithm?**

A) To find the shortest path between two vertices in a graph.

B) To find the longest path in a graph.

C) To find the minimum spanning tree of a graph.

D) To find the shortest path between all pairs of vertices in a graph.

26. **What is the time complexity of Floyd-Warshall Algorithm?**

A) $O(V)$

B) $O(E)$

C) $O(V2)$

D) $O(V3)$

27. **Which graph algorithm can be used to identify strongly connected components in a directed graph?**

A) Breadth-First Search (BFS)

B) Depth-First Search (DFS)

C) Dijkstra's Algorithm

D) Tarjan's Algorithm

28. **In a graph, what does a cycle represent?**

A) A set of vertices connected by edges

B) A path that connects two vertices

C) A closed loop of edges that starts and ends at the same vertex

D) A disconnected component of the graph

29. **What is the main purpose of Kruskal's Algorithm?**

A) To find the shortest path between two vertices in a graph.

B) To find the longest path in a graph.

C) To find the minimum spanning tree of a graph.

D) To find the shortest path between all pairs of vertices in a graph.

30. **Which algorithm is used to detect bridges and articulation points in a graph?**

A) Breadth-First Search (BFS)

B) Depth-First Search (DFS)

C) Dijkstra's Algorithm

D) Tarjan's Algorithm

Unit Summary

This unit covers essential data structures and algorithms for efficient data management and problem-solving. It includes balanced and self-adjusting search trees like AVL and Splay Trees, ensuring quick operations through balancing and splaying mechanisms. Key sorting algorithms such as Merge Sort and Quick Sort are explored, highlighting their time complexities and stability properties. Graph algorithms are detailed with a focus on traversal techniques (DFS, BFS), shortest path algorithms (Dijkstra's, Bellman-Ford), and Minimum Spanning Trees (Kruskal's, Prim's), emphasizing their applications and efficiency. The unit provides a comprehensive overview of these foundational topics, crucial for computational efficiency and optimization.

Glossary

1. **Binary Search Trees (BST):** A data structure that maintains sorted data and supports fast search, insertion, and deletion operations.
2. **Balanced Search Trees:** Search trees that are balanced to ensure efficient operations by minimizing height disparities between branches.
3. **AVL Trees:** A type of balanced binary search tree that ensures the height difference between left and right subtrees is no more than one, maintaining balance through rotations.
4. **Splay Trees:** A self-adjusting binary search tree where recently accessed elements are moved to the root, optimizing access to frequently used data.
5. **Merge Sort:** A sorting algorithm that divides the array into smaller subarrays, recursively sorts them, and merges them back together.
6. **Quick Sort:** A sorting algorithm that partitions the array around a pivot element, recursively sorting each partition.
7. **Sorting through an Algorithmic Lens:** Analyzing and comparing sorting algorithms based on various factors like time complexity, space complexity, stability, and adaptability to different input types.
8. **Selection:** Algorithms focused on finding specific elements, like the kth smallest or largest, often utilizing techniques similar to those in sorting algorithms.

9. **Graphs:** Data structures consisting of vertices (nodes) and edges (connections) between them, representing relationships.
10. **Data Structures for Graphs:** Methods for representing graphs, such as adjacency matrices or adjacency lists, which influence the efficiency of graph algorithms.
11. **Graph Traversals:** Techniques for systematically visiting all the vertices and edges of a graph, including Depth-First Search (DFS) and Breadth-First Search (BFS).
12. **Shortest Paths:** Algorithms finding the shortest path between two vertices in a graph, essential for optimization in network routing.
13. **Minimum Spanning Trees:** Algorithms constructing a subgraph with the minimum total edge weight, useful for connecting all vertices in a graph with minimum cost.

Self – Assessment Questions

1. Explain the concept of a Binary Search Tree (BST) and its operations. Describe the characteristics and benefits of balanced search trees.
2. Compare and contrast AVL Trees and Splay Trees in terms of their balancing mechanisms and performance. Discuss scenarios where each type of search tree would be most suitable.
3. Outline the Merge Sort algorithm and analyze its time complexity. Describe the partitioning process in Quick Sort and explain its impact on performance.
4. Compare various sorting algorithms based on their time complexity, stability, and adaptability.
5. Explain the concept of selection algorithms and their role in finding specific elements in an array.
6. Define graphs and discuss different methods for representing them. Explain the difference between graph traversals, such as DFS and BFS, and their applications.
7. Discuss the importance of shortest path algorithms in network routing and optimization. Describe the purpose of Minimum Spanning Tree algorithms and compare Prim's and Kruskal's algorithms.
8. Explain the concept of algorithmic complexity and its significance in evaluating the efficiency of algorithms. Discuss the trade-offs between time complexity and

space complexity in algorithm design. Analyze real-world applications of search trees, sorting algorithms, and graph algorithms.

Activities / Exercises / Case Studies

Activities:

1. **Implementing Search Trees:**
    - Write code to implement Binary Search Trees (BST), AVL Trees, and Splay Trees in your preferred programming language.
    - Create functions for insertion, deletion, and searching in each type of search tree.
    - Test your implementations with various input datasets and analyze their performance.
2. **Sorting Algorithm Visualization:**
    - Use a visualization tool or create your own to illustrate how Merge Sort and Quick Sort work step by step.
    - Observe how the arrays are divided, sorted, and merged in Merge Sort.
    - Track the partitioning process and element swaps in Quick Sort.

Exercises:

1. **Search Trees:**
    - Given a set of numbers, construct a Binary Search Tree (BST) and perform insertion, deletion, and search operations.
    - Experiment with different insertion orders to observe the effect on tree balance and performance.
2. **Sorting and Selection:**
    - Implement Merge Sort and Quick Sort algorithms and compare their performance on large datasets.
    - Write functions to find the kth smallest and largest elements in an array using selection algorithms like Quick Select.

Case Study:

Title: Network Optimization with Graph Algorithms

Description: A telecommunications company wants to optimize its network infrastructure to minimize costs while maximizing performance. The company's network consists of multiple nodes (representing cities) connected by edges (representing communication links). They need to:

1. Identify the shortest paths between critical network nodes to optimize data routing and minimize latency.
2. Determine the minimum spanning tree to connect all network nodes with the minimum total edge weight, ensuring efficient network coverage while minimizing costs.
3. Implement algorithms like Dijkstra's for finding shortest paths and Prim's/Kruskal's for constructing minimum spanning trees.
4. Analyze the impact of different algorithms on network performance and scalability.

Activities:

- Analyze the network topology and identify critical nodes and communication links.
- Implement graph algorithms to find shortest paths and construct minimum spanning trees.
- Simulate network optimization scenarios and measure the performance improvements achieved through algorithmic optimizations.

Exercises:

- Given a network topology represented as a graph, calculate the shortest paths between specified nodes using Dijkstra's algorithm.
- Construct a minimum spanning tree for the network graph using Prim's or Kruskal's algorithm and analyze its efficiency in connecting all network nodes with minimal cost.

Answers For Check Your Progress

Modules	S. No.	Answers
Module 1	1.	B) $O(\log n)$
	2.	B) AVL Tree
	3.	C) Guaranteed logarithmic time complexity for all operations
	4.	C) Rotation
	5.	D) Splay Tree
	6.	C) Both left and right rotations
	7.	B) $O(\log n)$
	8.	D) Splay Tree

	9.	C) Splay
	10.	B) $O(\log n)$
	1.	B) Merge Sort
	2.	C) Random element
	3.	B) Quick Sort
	4.	C) Stability
	5.	A) Quick Sort
	6.	C) $O(n2)$
	7.	D) Insertion Sort
	8.	B) $O(n)$
	9.	C) Quick Select
	10.	C) Divide-and-conquer strategy
	11.	C) Merge Sort
	12.	A) Bubble Sort
	13.	C) Quick Sort
	14.	A) Merge Sort
	15.	B) Quick Sort
Module 3	**1.**	B) Linked List
	2.	B) $O(\log n)$
	3.	A) Merge Sort
	4.	C) Middle element
	5.	A) Array
	6.	A) Depth-First Search (DFS)
	7.	C) $O(V+E)$
	8.	C) Dijkstra's Algorithm
	9.	C) Dijkstra's Algorithm
	10.	C) A tree with the minimum number of edges
	11.	C) $O(V+E)$
	12.	B) Depth-First Search (DFS)

	13.	A) The number of edges connected to the vertex
	14.	B) Queue
	15.	B) It provides fast lookup for edge existence.
	16.	B) It guarantees the shortest path from the source to all vertices in the graph.
	17.	D) Bellman-Ford Algorithm
	18.	B) It does not guarantee the shortest path.
	19.	B) Depth-First Search (DFS)
	20.	B) Edges with the minimum weight
	21.	D) Bellman-Ford Algorithm
	22.	D) $O(VE)$
	23.	A) The distance between two vertices
	24.	D) Floyd-Warshall Algorithm
	25.	D) To find the shortest path between all pairs of vertices in a graph.
	26.	D) $0O(V3)$
	27.	D) Tarjan's Algorithm
	28.	C) A closed loop of edges that starts and ends at the same vertex
	29.	C) To find the minimum spanning tree of a graph.
	30.	D) Tarjan's Algorithm

Suggested Readings

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.

Open-Source E-Content Links

1. GeeksforGeeks - Binary Search Trees
2. GeeksforGeeks - AVL Trees
3. GeeksforGeeks - Splay Trees
4. GeeksforGeeks - Merge Sort
5. GeeksforGeeks - Quick Sort
6. Khan Academy - Sorting Algorithms
7. GeeksforGeeks - Graph Data Structure
8. Khan Academy - Graph Algorithms
9. Coursera - Graph Algorithms

References

1. Zelle, J. M. (2004). *Python programming: an introduction to computer science*. Franklin, Beedle & Associates, Inc..