PERIYAR UNIVERSITY

NAAC 'A++' Grade – State University – NIRF Rank 56- State Public University Rank 25
SALEM - 636 011, Tamil Nadu, India

CENTRE FOR DISTANCE AND ONLINE EDUCATION

(CDOE)

B.SC COMPUTER SCIENCE

SEMESTER - II

CORE COURSE: DATA STRUCTURES AND ALGORITHMS

(Candidates admitted from 2024 onwards)

PERIYAR UNIVERSITY

CENTRE FOR DISTANCE AND ONLINE EDUCATION (CDOE)

B.Sc COMPUTER SCIENCE 2024 admission onwards

CORE COURSE – III
Data Structure and Algorithms

Prepared by:

Centre for Distance and Online Education (CDOE)
Periyar University, Salem – 11.

LIST OF CONTENTS

UNIT	CONTENTS	PAGE
I	Abstract Data Types (ADTs)- List ADT-array-based implementation-linked list implementation singly linked lists-circular linked lists-doubly-linked lists-applications of lists-Polynomial Manipulation- All operations-Insertion-Deletion-Merge-Traversal	1
II	Stack ADT-Operations- Applications- Evaluating arithmetic expressions – Conversion of infix to postfix expression-Queue ADT-Operations-Circular Queue- Priority Queue-Dequeue applications of queues.	29
III	Tree ADT-tree traversals-Binary Tree ADT-expression trees-applications of trees-binary search tree ADT- Threaded Binary Trees-AVL Trees- B-Tree- B+ Tree – Heap-Applications of heap.	70
IV	Definition- Representation of Graph- Types of graph-Breadth first traversal – Depth first traversal-Topological sort- Bi-connectivity – Cut vertex- Euler circuits-Applications of graphs.	131
V	Searching- Linear search-Binary search-Sorting-Bubble sort-Selection sort-Insertion sort-Shell sort-Radix sort-Hashing-Hash functions-Separate chaining- Open Addressing-Rehashing Extendible Hashing	174

UNIT I: LINEAR DATA STRUCTURES

LINEAR DATA STRUCTURES – LIST Abstract Data Types (ADTs) – List ADT – array-based implementation – linked list implementation —singly linked lists- circularly linked lists- doubly-linked lists– applications of lists –Polynomial Manipulation – All operation (Insertion, Deletion, Merge, Traversal)

Section No	Topic	Page No
	Unit Objectives	
1.1	**Introduction to Abstract Data Types (ADTs)**	
1.1.1	Definition and Importance	
1.2	**List ADT**	
1.2.1	Description	
1.2.2	Key Characteristics	
1.2.3	Operations	
1.3	**Array-Based List Implementation**	
1.3.1	Example Operations (Insertion, Deletion, Traversal)	
1.3.2	Use Cases	
1.4	**Linked List Implementation**	
1.4.1	Types of Linked Lists	
1.4.2	Node Structure	
1.4.3	**Singly Linked List** Insertion, Deletion, Traversal	
1.4.4	**Doubly Linked List Implementation** Node Structure Operations (Insertion, Deletion, Traversal)	
1.4.5	**Circular Linked List Implementation** Node Structure Operations (Insertion, Deletion, Circular Traversal)	
1.5	**Applications of Lists**	
1.5.1	Polynomial Manipulation	
1.6	**Summary**	
	Activities	
	Check Your Progress	
	Self-Assessment Questions	
	Further Reading and References	

Unit Objectives:

This unit aims to provide a comprehensive understanding of linear data structures with a focus on the List Abstract Data Type (ADT). By the end of this unit, students will be able to:

- Define and Understand Abstract Data Types (ADTs).
- Describe and Implement List ADT.
- Implement Array-Based Lists.
- Implement Linked List-Based Lists.
- Apply Lists to Real-World Problems
- Describe various types of operations can be performed in List ADT.

1.1 Introduction to Abstract Data Types (ADTs)

1.1.1 Definition and Importance:

An Abstract Data Type (ADT) is a theoretical model for data structures that encapsulates data and operations, abstracting away implementation details. ADTs are crucial in computer science because they provide a clear interface and enable modular design, allowing developers to focus on higher-level program logic without worrying about low-level implementation details.

1.2 List ADT

1.2.1 Description:

A List Abstract Data Type (ADT) represents a sequence of elements, with a specific order. Elements can be accessed by their positions in the list. The ADT is a fundamental data structure that represents a sequence of elements. This ADT provides a way to organize, store, and manipulate collections of items. Lists are widely used in programming and can be implemented in various ways, including arrays, linked lists, or more complex structures like doubly linked lists or skip lists. Here's an overview of the List ADT:

1.2.2 Key Characteristics

1. **Ordered Collection**: Elements in a list have a specific order, and each element has a unique position or index.
2. **Homogeneous Elements**: Typically, all elements in the list are of the same type.
3. **Dynamic Size**: The size of the list can change dynamically, allowing for insertion and deletion of elements.

1.2.3 Operations:

A List ADT supports several operations that are essential for manipulating the elements. These include:

- **Insertion**: Add an element at a specific position.
- **Deletion**: Remove an element from a specific position.
- **Traversal**: Visit each element in the list in sequence.
- **Search**: Find the position of an element in the list.

Let us sum up:

- **Abstract Data Types (ADTs)**:
  - **Definition**: ADTs are theoretical models that define data structures and their operations without specifying implementation details.
  - **Importance**: They provide a clear interface, promoting modular design and allowing focus on high-level logic.
- **List ADT:**
  - **Description**: Represents a sequence of elements in a specific order, with access based on element positions.
- **Key Characteristics of List ADT:**
  - **Ordered Collection**: Elements have a specific order and unique positions or indices.
  - **Homogeneous Elements**: Typically contains elements of the same type.

- **Dynamic Size**: Allows for flexible resizing with insertion and deletion operations.

➢ **asic Operations:**

- **Insertion**: Adding an element at a specific position in the list.
- **Deletion**: Removing an element from a specific position.
- **Traversal**: Visiting and processing each element in sequence.
- **Search**: Finding the position of a specific element within the list.

➢ **Implementation Variations:**

- **Arrays**: Fixed-size, indexed by positions.
- **Linked Lists**: Dynamic size, elements are linked using pointers.
- **Doubly Linked Lists**: Elements linked in both directions for bidirectional traversal.
- **Skip Lists**: Enhanced linked lists with multiple levels for faster search.

➢ **Applications:**

- **Data Storage**: Managing collections of items in various applications.
- **Dynamic Data Handling**: Useful for applications requiring frequent changes in size and content.

➢ **Advantages:**

- **Flexibility**: Supports dynamic resizing and various manipulation operations.
- **Versatility**: Can be implemented in multiple ways to suit different needs and performance requirements.

Check your Progress:

1. **What does the List ADT represent?**

- A) A single element B) A collection of elements in a specific order
- C) A set of unordered elements D) A stack of elements

Answer: B) A collection of elements in a specific order

2. **Which of the following is a key characteristic of a List ADT?**

A) Fixed size B) Elements are unordered

C) Homogeneous elements D) Elements can only be accessed by index

Answer: C) Homogeneous elements

3. **Which operation allows you to add an element at a specific position in a List ADT?**

A) Deletion B) Traversal C) Insertion D) Search

Answer: C) Insertion

4. **How can the size of a List ADT be described?**

A) Fixed B) Unchangeable C) Dynamic D) Immutable

Answer: C) Dynamic

5. **Which operation is used to find the position of an element in a List ADT?**

A) Deletion B) Traversal C) Insertion D) Search

Answer: D) Search

1.3 Array-Based List Implementation

Array-based lists store elements in contiguous memory locations, enabling direct access to elements via indices.

- Supports fast access (O(1) time complexity) for reading and updating elements by index.
- Insertion and deletion can be slow (O(n) time complexity) due to the need to shift elements.

1.3.1 Example Operations:

(i) **Insertion:**

```
def insert(arr, element, position):
    for i in range(len(arr) - 1, position - 1, -1):
```

```
        arr[i + 1] = arr[i]
    arr[position] = element
```

(ii) **Deletion:**

```
def delete(arr, position):
    for i in range(position, len(arr) - 1):
        arr[i] = arr[i + 1]
    arr[len(arr) - 1] = None
```

(iii) **Traversal:**

```
def traverse(arr):
    for element in arr:
        print(element)
```

1.3.2 Use Cases

- **Ordered Data:** When maintaining an ordered collection of elements are necessary.
- **Dynamic Data:** When the number of elements can frequently change, requiring dynamic resizing.
- **Sequence Operations:** When operations like insertion, deletion, and traversal of a sequence of elements are common.

Let us sum up:

- **Definition and Access**:
  - Array-based lists store elements in contiguous memory locations, providing O(1) time complexity for accessing and updating elements via indices.
- **Insertion and Deletion**:
  - Insertion and deletion operations are O(n) due to the need to shift elements to maintain order, which can impact performance.

- **Example Operations**:
  - **Insertion**: Shifts elements to make space for a new element at a specified position.
  - **Deletion**: Removes an element and shifts subsequent elements to fill the gap.
  - **Traversal**: Iterates through and prints each element in the array.
- **Use Cases:**
  - Suitable for maintaining ordered data, handling dynamic changes in the number of elements, and performing common sequence operations like insertion, deletion, and traversal.
- **Dynamic Resizing**:
  - Array-based lists require resizing when the number of elements changes frequently, impacting performance and memory management.

Check your Progress:

1. **What is the time complexity of accessing an element in an array-based list?**

   A) O(n) B) O(log n) C) O(1) D) O(n^2)

Answer: C) O(1)

2. **What is the time complexity of insertion in an array-based list?**

   A) O(1) B) O(n) C) O(log n) D) O(n^2)

Answer: B) O(n)

3. **In the given insertion operation, what is the purpose of shifting elements to the right?**

   A) To make space for a new element B) To sort the elements

   C) To delete an element D) To traverse the array

Answer: A) To make space for a new element

4. **What happens to the last element of the array after deletion in the provided code?**

A) It is duplicated B) It is shifted to the left

C) It is set to None D) It remains unchanged

Answer: C) It is set to None

5. **When is an array-based list particularly useful?**

A) When maintaining an ordered collection is necessary

B) When you need a highly dynamic data structure

C) When you need to perform frequent insertions and deletions

D) When space complexity is a concern

Answer: A) When maintaining an ordered collection is necessary

1.4 Linked List Implementation

Linked lists use nodes, where each node contains data and a reference (link) to the next node.

Implementing a List ADT using a linked list involves defining nodes that contain the elements and pointers to the next node in the sequence.

1.4.1 Types of Linked Lists:

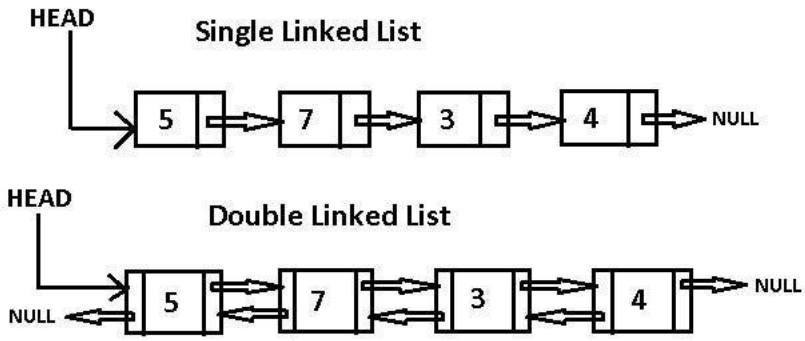
- **Singly Linked List**: Each node points to the next node.
- **Circularly Linked List**: The last node points back to the first node.
- **Doubly Linked List**: Each node has two links, one to the next node and one to the previous node.

1.4.2 Node Structure:

- Each **Node** has two attributes: **data** to store the element and **next** to point to the next node in the list.

Example:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```



1.4.3 Singly Linked List:

A singly linked list is a linear collection of elements, called nodes, where each node stores a data element and a reference (or link) to the next node in the sequence.

(i) Insertion:

Add a new element at the specified position. It handles the special case of inserting at the beginning separately. For other positions, it traverses the list to find the correct insertion point.

```
def insert(head, data):
    new_node = Node(data)
    if not head:
        return new_node
    current = head
    while current.next:
        current = current.next
```

```
current.next = new_node
return head
    def insert(head, data):
    new_node = Node(data)
    if not head:
            return new_node
    current = head
    while current.next:
            current = current.next
    current.next = new_node
    return head
```

(ii) Deletion:

Removes the element at the specified position and returns the removed element's data. It handles the special case of removing the head separately. For other positions, it traverses the list to find the correct node to remove.

```
def delete(head, key):
  current = head
  if current and current.data == key:
    return current.next
  prev = None
  while current and current.data != key:
    prev = current
    current = current.next
  if not current:
    return head
```

```
            prev.next = current.next
            return head
```

(iii) Traversal:

The **traverse** function is a simple utility for iterating through all the elements in a linked list and printing their values. This function demonstrates how to navigate a linked list from the head (starting node) to the end.

```
def traverse(head):
    current = head
    while current:
        print(current.data)
        current = current.next
```

1.4.4 Doubly Linked List Implementation

A Doubly Linked List (DLL) is a type of linked list where each node contains a data part and two pointers, next and prev. The next pointer points to the next node in the sequence, and the prev pointer points to the previous node. This allows traversal in both directions.

Node Structure for DLL:

```
class DNode:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None
```

Operations:

(i) Insertion:

- **At the Beginning:**

```
def insert_at_beginning(head, data):
    new_node = DNode(data)
    if head:
        head.prev = new_node
    new_node.next = head
    return new_node
```

- **At the End:**

```
def insert_at_end(head, data):
    new_node = DNode(data)
    if not head:
        return new_node
    current = head
    while current.next:
        current = current.next
    current.next = new_node
    new_node.prev = current
    return head
```

(ii) Deletion:

```
def delete_node(head, key):
    current = head
    while current and current.data != key:
        current = current.next
    if not current:
        return head
    if current.prev:
        current.prev.next = current.next
    if current.next:
        current.next.prev = current.prev
    if current == head:
        head = current.next
    return head
```

(iii)Traversal:

- **Forward Traversal:**

```
def traverse_forward(head):
    current = head
    while current:
        print(current.data)
        current = current.next
```

- **Backward Traversal:**

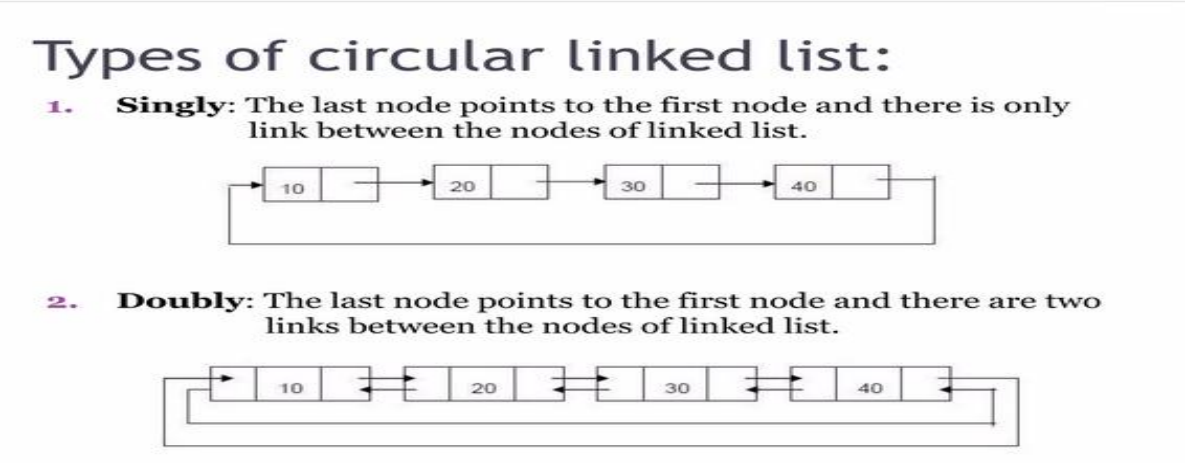
```
def traverse_backward(head):
    current = head
    while current.next:
        current = current.next
    while current:
        print(current.data)
        current = current.prev
```

1.4.5 Circular Linked List Implementation

A Circular Linked List (CLL) is a linked list where the last node points back to the first node, forming a circular structure. It can be singly or doubly linked.

Node Structure for CLL:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```



Operations:

(i) Insertion:

- **At the Beginning:**

```
def insert_at_beginning(head, data):
    new_node = Node(data)
    if not head:
        new_node.next = new_node
        return new_node
    current = head
    while current.next != head:
        current = current.next
    new_node.next = head
    current.next = new_node
    return new_node
```

- **At the End:**

```
def insert_at_end(head, data):
    new_node = Node(data)
    if not head:
        new_node.next = new_node
        return new_node
    current = head
    while current.next != head:
        current = current.next
    current.next = new_node
```

```
    new_node.next = head
    return head
```

(ii) Deletion:

- **Delete a Node:**

```
def delete_node(head, key):
    if not head:
        return None
    if head.data == key and head.next == head:
        return None
    last = head
    d = None
    if head.data == key:
        while last.next != head:
            last = last.next
        last.next = head.next
        head = last.next
        return head
    while last.next != head and last.next.data != key:
        last = last.next
    if last.next.data == key:
        d = last.next
        last.next = d.next
    return head
```

(iii)Traversal:

- **Circular Traversal:**

```
def traverse(head):
    if not head:
        return
    current = head
    while True:
        print(current.data)
        current = current.next
        if current == head:
            break
```

Use Case:

Using a linked list to implement a List Abstract Data Type (ADT) can be particularly beneficial in scenarios where dynamic and efficient manipulation of elements is required. Here are some specific use cases where a linked list implementation of a List ADT is advantageous:

(i) **Dynamic Data Management:** Applications where the number of elements changes frequently.

- Example: A task manager application where tasks are added and removed dynamically.

(ii) **Real-Time Data Processing:** Real-time systems requiring predictable insertion and deletion times

- **Example**: Real-time traffic management systems that handle events (e.g., adding and removing vehicles).

(iii) **Memory Efficiency in Sparse Data:** Managing sparse data where most elements are default or zero.

- **Example**: A sparse matrix representation in scientific computing.

(iv) **Implementation of Other Data Structures:** As a building block for more complex data structures.

- **Example:** Implementing stacks, queues, or adjacency lists for graphs.

Let us sum up:

- **Linked Lists**: Consist of nodes with data and a reference to the next node.
- **Types**:
  - **Singly Linked List**: Nodes have a reference to the next node.
  - **Doubly Linked List**: Nodes have references to both the next and previous nodes.
  - **Circular Linked List**: The last node points back to the first node.
- **Node Structure**:
  - **Singly Linked List**: class Node: def __init__(self, data): self.data = data; self.next = None
  - **Doubly Linked List**: class DNode: def __init__(self, data): self.data = data; self.next = None; self.prev = None
- **Operations**:
  - **Insertion**: At the beginning, end, or a specific position.
  - **Deletion**: Remove nodes by key and update links.
  - **Traversal**: Print elements from start to end, or in both directions for doubly linked lists.
- **Use Cases**: Dynamic data management, real-time processing, memory-efficient storage, and implementing other data structures.

Check your Progress

1. **Which type of linked list allows traversal in both directions?**

   A) Singly Linked List B) Doubly Linked List

   C) Circular Linked List D) All of the Above

   Answer*: B) Doubly Linked List*

2. **In a circular linked list, where does the last node point?**

   A) To the previous node B) To the first node

   C) To a null reference D) To itself

   Answer: *B) To the first node*

3. **What is the purpose of the prev pointer in a doubly linked list?**

   A) To point to the next node B) To point to the previous node

   C) To point to the end of the list D) To store additional data

   Answer: B) To point to the previous *node*

4. **Which operation involves updating links to remove a node in a singly linked list?**

   A) Insertion at the end B) Traversal

   C) Deletion D) Insertion at the beginning

   Answer: C) Deletion

5. **Which use case is best suited for a circular linked list?**

   A) Task manager application B) Sparse matrix representation

   C) Real-time traffic management D) Implementing a queue

   Answer: D) Implementing a queue

1.5 Applications of Lists

1.5.1 Polynomial Manipulation:

Polynomials can be efficiently represented using linked lists, where each node represents a term with a coefficient and exponent.

Example Operations:

(i) Insertion:

```
def insert_term(poly, coefficient, exponent):
    new_term = Node((coefficient, exponent))
    if not poly or poly.data[1] < exponent:
        new_term.next = poly
        return new_term
    current = poly
    while current.next and current.next.data[1] > exponent:
        current = current.next
    new_term.next = current.next
    current.next = new_term
    return poly
```

(ii) Deletion:

```
def delete_term(poly, exponent):
    if not poly:
        return None
    if poly.data[1] == exponent:
        return poly.next
    current = poly
```

```
        while current.next and current.next.data[1] != exponent:
            current = current.next
        if current.next:
            current.next = current.next.next
        return poly
```

(iii) Merge:

```
def merge_poly(poly1, poly2):
    dummy = Node((0, 0))
    tail = dummy
    while poly1 and poly2:
        if poly1.data[1] > poly2.data[1]:
            tail.next = poly1
            poly1 = poly1.next
        elif poly1.data[1] < poly2.data[1]:
            tail.next = poly2
            poly2 = poly2.next
        else:
            coeff = poly1.data[0] + poly2.data[0]
            if coeff != 0:
                tail.next = Node((coeff, poly1.data[1]))
            poly1 = poly1.next
            poly2 = poly2.next
        tail = tail.next
```

```
    tail.next = poly1 if poly1 else poly2
    return dummy.next
```

(iv) Traversal:

```
def traverse_poly(poly):
    current = poly
    while current:
        print(f"{current.data[0]}x^{current.data[1]}", end=" ")
        current = current.next
    print()
```

Let us sum up:

- **Polynomial Representation**: Polynomials are represented using linked lists, where each node contains a term with a coefficient and exponent.
- **Insertion**: To insert a term, traverse the list to find the correct position based on the exponent and insert the new term in the sorted order.
- **Deletion**: To delete a term, find the node with the matching exponent and adjust the pointers to remove it from the list.
- **Merge**: Merging two polynomials involves combining their terms by comparing exponents, summing coefficients of like terms, and maintaining the sorted order.
- **Traversal**: To display a polynomial, traverse through the linked list and print each term in the format of coefficient and exponent.

Check Your Progress

1. **How are polynomials represented in the given content?**
   A) Arrays B) Linked lists C) Stacks D) Queues
   Answer: B) Linked lists

2. **What is the purpose of the insert_term function?**
   A) To remove a term from the polynomial
   B) To add a new term in the correct position based on exponent
   C) To merge two polynomials
   D) To traverse the polynomial
   Answer: B) To add a new term in the correct position based on exponent

3. **What does the delete_term function do?**
   A) Adds a new term to the polynomial
   B) Removes a term with the specified exponent from the polynomial
   C) Merges two polynomials
   D) Traverses through the polynomial list
   Answer: B) Removes a term with the specified exponent from the polynomial

4. **In the merge_poly function, what happens when two terms have the same exponent?**
   A) Both terms are added to the new polynomial
   B) The term with the smaller coefficient is removed
   C) The coefficients of both terms are summed
   D) The terms are merged into a single term with the sum of their exponents
   Answer: C) The coefficients of both terms are summed

5. **What is the purpose of the traverse_poly function?**
   A) To insert a new term into the polynomial
   B) To delete a term from the polynomial
   C) To merge two polynomials
   D) To print all terms of the polynomial in a readable format
   Answer: D) To print all terms of the polynomial in a readable format

Summary:

The introduction to Abstract Data Types (ADTs) defines them as theoretical models for data structures, emphasizing their encapsulation of data and operations while abstracting away implementation details. ADTs are highlighted for their

importance in providing clear interfaces and enabling modular design, freeing developers to focus on higher-level program logic.

Key ADTs are explored, with a detailed focus on the List ADT. Lists are described as ordered collections with homogeneous elements and dynamic sizes, supporting essential operations like insertion, deletion, traversal, and search. Implementation methods such as array-based lists and linked lists are discussed, each with its characteristics, operations, and use cases.

Linked list implementations, including singly linked lists, doubly linked lists, and circular linked lists, are explained in detail, covering their node structures, operations, and traversal methods. Additionally, specific use cases for linked list implementations of List ADTs are provided, highlighting scenarios where dynamic and efficient manipulation of elements is crucial.

Applications of lists are discussed, such as polynomial manipulation, where linked lists offer efficiency, and conclusions emphasize the fundamental role of understanding ADTs in designing efficient algorithms and data structures, enabling effective implementation and usage regardless of internal representation.

Activities

Activity 1: Discussion and Definition Exercise

- **Objective:** Understand and articulate the concept and importance of ADTs.
- **Instructions:**
    1. Divide into small groups.
    2. Each group will discuss and write a concise definition of an ADT.
    3. Groups will then list three reasons why ADTs are important in computer science.
    4. Present findings to the class.

Activity 2: Exploring List ADT Operations

- **Objective:** Implement and demonstrate key operations of a List ADT.
- **Instructions:**

1. Form small groups and assign each group one of the following operations: insertion, deletion, traversal, search.
2. Each group will write code to implement their assigned operation for a List ADT.
3. Groups will demonstrate their implementations using sample data.

Check Your Progress

1. What is an Abstract Data Type (ADT)?

   ---

2. Describe the key operations of the List ADT.

   ---

3. How does an array-based list differ from a linked list?

   ---

4. Explain the structure and advantages of a doubly linked list.

Self-Assessment Questions

1. What is an Abstract Data Type (ADT), and why is it important in computer science?
2. Explain the concept of an array-based implementation of a List ADT.

3. Compare the advantages and disadvantages of using arrays for implementing lists.
4. Describe the structure of a singly linked list and how nodes are linked.
5. Provide an example of inserting an element at the end of a singly linked list.
6. What is a circularly linked list? How does it differ from a singly linked list?
7. Explain the structure of a doubly linked list and its advantages over singly linked lists.
8. Implement a function to delete a node from a doubly linked list.
9. Provide examples of how lists (Array-based or Linked lists) can be used as fundamental data structures in various applications.
10. Discuss specific scenarios where the choice between array-based and linked list implementations is critical.
11. How can linked lists be used to efficiently represent and manipulate polynomials?
12. Implement a function to add two polynomials using linked lists.
13. Explain the steps involved in deleting a node from a linked list.
14. Identify real-world applications where each type of list (array-based, singly linked, doubly linked) is best suited.

Further Reading and References

Textbooks

1. **"Data Structures and Algorithm Analysis in C++" by Mark Allen Weiss**
    - Covers list ADTs, array-based and linked list implementations, and operations like insertion, deletion, and traversal.
    - ISBN: 978-0132847377
2. **"Algorithms, Part I" by Robert Sedgewick and Kevin Wayne**
    - Includes a comprehensive overview of fundamental data structures including lists and their applications.

- ISBN: 978-0321573513

3. **"Data Structures and Algorithmic Thinking with Python" by Narasimha Karumanchi**
   - Provides insights into linked lists, their types, and operations in Python.
   - ISBN: 978-9352604969
4. **"Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein**
   - A foundational textbook for data structures, including lists and their operations.
   - ISBN: 978-0262033848
5. **"Data Structures and Algorithms in Java" by Robert Lafore**
   - Covers various types of linked lists and array-based implementations with Java examples.
   - ISBN: 978-0672331024

Online Resources

1. **GeeksforGeeks - Data Structures**
   - Provides detailed articles and tutorials on list ADTs, array-based and linked list implementations, and various operations.
   - GeeksforGeeks Data Structures
2. **Khan Academy - Algorithms**
   - Offers interactive tutorials on basic data structures, including linked lists and arrays.
   - Khan Academy Algorithms
3. **Coursera - Data Structures and Algorithm Specialization**
   - Covers various data structures, including lists, with video lectures and assignments.
   - Coursera Specialization

4. **W3Schools - JavaScript Data Structures**
    - Provides tutorials on implementing linked lists and arrays in JavaScript.
    - W3Schools Data Structures
5. **TutorialsPoint - Data Structures**
    - Includes tutorials and examples on array-based and linked list implementations.
    - TutorialsPoint Data Structures

Video Material

1. **YouTube - Data Structures Playlist by MyCodeSchool**
    - Offers clear explanations of various data structures, including linked lists and their operations.
    - MyCodeSchool Data Structures
2. **YouTube - Data Structures and Algorithms by Abdul Bari**
    - Provides comprehensive lectures on different data structures and algorithms, including linked lists.
    - Abdul Bari Data Structures
3. **MIT OpenCourseWare - Introduction to Algorithms**
    - Video lectures covering data structures and algorithms in-depth.
    - MIT OpenCourseWare
4. **Udacity - Data Structures and Algorithms**
    - Course videos and practical exercises related to linear data structures.
    - Udacity Data Structures and Algorithms
5. **freeCodeCamp - Data Structures in JavaScript**
    - Provides video tutorials and explanations on implementing data structures like linked lists in JavaScript.
    - freeCodeCamp Data Structures

UNIT 2: STACK & QUEUE ADT

Stack ADT-Operations- Applications- Evaluating arithmetic expressions – Conversion of infix to postfix expression-Queue ADT-Operations-Circular Queue- Priority Queue-deQueue applications of queues.

Section No	Topic	Page No
2.1	**Unit Objectives**	
2.2	**STACK ADT**	
2.2.1	Key Characteristics of Stack ADT	
2.2.2	Common Use Cases	
2.3	**Operations of a Stack**	
2.3.1	Working Methodology	
2.3.2	Example Implementations	
2.3.3	Array-Based Implementation	
2.3.4	Linked List-Based Implementation	
2.4	**Stack ADT Applications**	
2.5	Evaluating arithmetic expressions	
2.5.1	Types of Expressions	
2.5.2	Infix to Postfix Conversion	
2.5.3	Evaluating Postfix Expression	
2.6	**Queue ADT (Abstract Data Type)**	
2.7	**Basic Operations on a Queue**	
2.7.1	Queue Implementations Array-Based Queue Linked List-Based Queue	
2.8	**Circular Queue**	
2.8.1	Key Characteristics of a Circular Queue	
2.8.2	Working Methodology	
2.9	**Priority Queue**	
2.9.1	Key Characteristics of Priority Queue	
2.9.2	Operations in a Priority Queue	
2.10	**Deque (Double-Ended Queue)**	
2.10.1	Operations in a Deque	
2.10.2	Working Methodology	
2.11	**Applications of Queues**	
2.12	**Summary**	
	Activities	
	Points to Remember	
	Questions	
	Glossary	
	Further Reading and References	

Unit Objectives:

This unit aims to provide a comprehensive understanding of linear data structures Stack ADT and Queue ADT. The objective of this unit are mentioned below.

- Understand Stack ADT: Learn key operations like push, pop, peek, isEmpty, and isFull, and their practical significance.
- Explore Stack Applications: Analyze how stacks are used in real-world tasks such as function calls, expression evaluation, backtracking, and undo mechanisms.
- Master Arithmetic Expression Evaluation: Grasp the process of converting infix to postfix notation and its importance in compiler design.
- Learn Queue ADT: Comprehend enqueue, dequeue, peek, isEmpty, and isFull operations and their FIFO management utility.
- Study Circular Queue: Understand its definition, advantages over standard queues, and efficient storage utilization.
- Explore Priority Queue: Learn about its operations and applications in managing elements based on priority.

2.1 STACK ADT

A Stack is a type of Abstract Data Type (ADT) that operates in a particular order, specifically Last In, First Out (LIFO). This means that the last element added to the stack will be the first one to be removed. Think of a stack like a stack of plates: you can only take the top plate off the stack, and you can only add a new plate to the top.

2.1.1 Key Characteristics of Stack ADT

- **LIFO Order**: The most recently added element is the first to be removed.
- **Access**: Only the top element of the stack is accessible directly.

2.1.2 Common Use Cases

- **Function Call Management**: The call stack in many programming languages.
- **Expression Evaluation**: For parsing and evaluating expressions.
- **Backtracking Algorithms**: Depth-First Search (DFS) in graphs, navigating mazes, etc.
- **Undo Mechanism**: Implementing undo features in applications.

2.2 Operations of a Stack

(i) **Push**: Adds an element to the top of the stack.

(ii) **Pop**: Removes and returns the element from the top of the stack.

(iii) **Peek (Top)**: Returns the element from the top of the stack without removing it.

(iv) **isEmpty**: Checks if the stack is empty.

(v) **Size**: Returns the number of elements in the stack.

2.2.1 Working Methodology

(i) Push Operation

Step 1: Increment the stack pointer (or index) to point to the next empty location in the stack.

Step 2: Add the new element at the position pointed to by the stack pointer.

(ii) Pop Operation

Step 1: Check if the stack is empty.

Step 2: If not empty, remove the element from the top of the stack by decrementing the stack pointer.

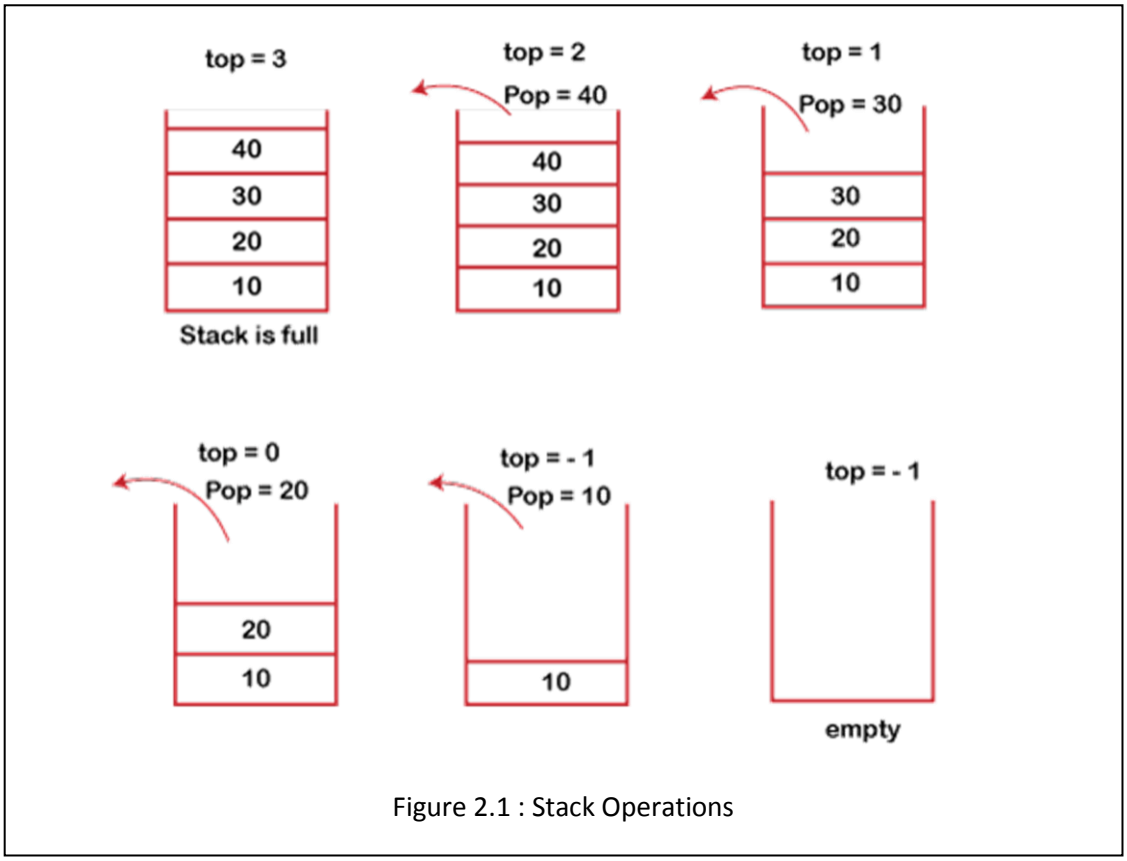


Figure 2.1 : Stack Operations

(iii) Peek Operation

Step 1: Check if the stack is empty.

Step 2: If not empty, return the element at the top of the stack without removing it.

(iv) isEmpty Operation

Step 1: Check if the stack pointer (or index) is at the initial position (indicating an empty stack).

(v) Size Operation

Step 1: Return the current value of the stack pointer (or index), which represents the number of elements in the stack.

2.2.2 Array-Based Implementation

```
class ArrayStack:
    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)



    def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        else:
            raise IndexError("pop from empty stack")

    def peek(self):
        if not self.is_empty():
            return self.stack[-1]
        else:
            raise IndexError("peek from empty stack")

    def is_empty(self):
        return len(self.stack) == 0
```

```
    def is_full(self):
        # Example for a fixed-size stack
        MAX_SIZE = 100
        return len(self.stack) == MAX_SIZE


# Example usage:
stack = ArrayStack()
stack.push(10)
stack.push(20)
print(stack.pop())  # Output: 20
print(stack.peek())  # Output: 10
```

2.2.3 Linked List-Based Implementation

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedListStack:
    def __init__(self):
        self.top = None

    def push(self, item):
        new_node = Node(item)
        new_node.next = self.top
        self.top = new_node

    def pop(self):
        if not self.is_empty():
            data = self.top.data
```

```
            self.top = self.top.next
            return data
        else:
            raise IndexError("pop from empty stack")
        def peek(self):
        if not self.is_empty():
            return self.top.data
        else:
            raise IndexError("peek from empty stack")

    def is_empty(self):
        return self.top is None

# Example usage:
stack = LinkedListStack()
stack.push(10)
stack.push(20)
print(stack.pop())  # Output: 20
print(stack.peek())  # Output: 10
```

In summary, a stack ADT is a fundamental data structure that operates on a LIFO basis, with operations primarily focused on adding, removing, and accessing the top element, as well as checking if the stack is empty or full. The implementation can vary, with common choices being array-based or linked list-based, each having its own trade-offs.

Let us sum up:

- **Stack ADT (LIFO)**: A stack follows the Last In, First Out (LIFO) order, where the last element added is the first one removed.
- **Key Operations**:
    - Push: Adds an element to the top.
    - Pop: Removes and returns the top element.
    - Peek: Returns the top element without removing it.
    - isEmpty: Checks if the stack is empty.

  - Size: Returns the number of elements in the stack.
- **Use Cases**: Function call management, expression evaluation, backtracking algorithms (DFS), and undo mechanisms.
- **Implementation**:
  - Array-based: Simple implementation using a list.
  - Linked list-based: Uses nodes, with each element pointing to the next.
- **Trade-offs**: Array-based stacks are simple but may require resizing, while linked list-based stacks can dynamically grow but have higher memory overhead.

Check Your Progress:

1. **What is the order of operations in a Stack ADT?**

   a) First In, First Out (FIFO) b) Last In, First Out (LIFO)

   c) First In, Last Out (FILO) d) Random Order

   Answer: b) Last In, First Out (LIFO)

2. **Which operation adds an element to the top of the stack?**

   a) Pop b) Push c) Peek d) Size

   Answer: b) Push

3. **What is the purpose of the peek operation in a stack?**

   a) To remove and return the top element

   b) To add an element to the stack

   c) To return the top element without removing it

   d) To check if the stack is full

   Answer: c) To return the top element without removing it

4. **Which of the following is a common use case of a stack?**

   a) Breadth-First Search (BFS) b) Managing function calls

   c) Queue management d) Sorting algorithms

   Answer: b) Managing function calls

5. **In a linked list-based stack implementation, which of the following is true?**

   a) The stack has a fixed size. b) Each node points to the previous node.

   c) It dynamically grows as elements are added.

   d) The is_full function is required.

   Answer: c) It dynamically grows as elements are added.

2.3 Stack ADT Applications

The Stack Abstract Data Type (ADT) is a versatile and fundamental data structure with many applications in computer science and programming. Their LIFO property makes them suitable for tasks that involve managing a dynamic set of elements where the most recently added elements need to be accessed or removed first. Here are some key applications of the stack ADT:

Function Call Management:

- **Call Stack**: In many programming languages, the call stack is used to manage function calls. Each time a function is called, its context (local variables, return address, etc.) is pushed onto the stack. When the function returns, its context is popped from the stack.

2.4 Expression Evaluation and Syntax Parsing:

- **Arithmetic Expression Evaluation**: Stacks are used to evaluate arithmetic expressions written in postfix notation (Reverse Polish Notation). They can also be used to convert infix expressions (common arithmetic notation) to postfix or prefix notation.
- **Syntax Parsing**: Compilers use stacks for parsing the syntax of programming languages. For example, they use stacks to check for balanced parentheses or to evaluate expressions in the correct order.

Backtracking Algorithms:

- **Depth-First Search (DFS)**: DFS in graph traversal algorithms uses a stack to keep track of the vertices to be explored. This is useful in solving puzzles, navigating mazes, and searching through game trees.
- **Backtracking**: Stacks are used in algorithms that involve backtracking, such as finding solutions to the N-Queens problem or solving Sudoku puzzles. The stack keeps track of the decisions made, allowing the algorithm to backtrack to a previous state when a dead end is reached.

Undo Mechanisms in Software:

- **Undo/Redo Functionality**: Many applications (like text editors) use stacks to implement undo and redo functionality. Each action is pushed onto an undo

stack, and when the user undoes an action, it is popped from the undo stack and pushed onto the redo stack.

Memory Management:

- **Expression Memory Management**: Stacks are used in managing memory allocation, particularly in the context of recursive function calls and local variable storage.

Balanced Parentheses and Bracket Matching:

- **Syntax Validation**: Stacks are used to check for balanced parentheses, brackets, and braces in code editors and compilers, ensuring that each opening symbol has a corresponding closing symbol.

String Reversal:

- **Reversing Strings**: Stacks can be used to reverse a string by pushing each character onto the stack and then popping them off, which outputs the characters in reverse order.

Navigation in Web Browsers:

- **Back and Forward Navigation**: Web browsers use stacks to manage the history of visited pages. The back button pops the current page from the stack, and the forward button pushes pages back onto the stack.

Tower of Hanoi:

- **Recursive Solution**: The Tower of Hanoi problem can be solved using recursion, which internally uses a stack to manage recursive function calls.

Palindrome Checking:

- **Checking Palindromes**: Stacks can be used to check if a string is a palindrome by pushing characters onto the stack and then comparing the popped characters with the original string.

Activities

Activity 1. : Compare the performance of array-based and linked list-based stack implementations.

Activity 2: Use animations or interactive elements to illustrate push, pop, and peek operations.

2.4 Evaluating arithmetic expressions

2.4.1 Types of Expressions

1. **Infix Expression**: Operators are between operands

   Example: A + B
2. **Prefix Expression** (Polish Notation): Operators precede operands

   Example: +A B
3. **Postfix Expression** (Reverse Polish Notation): Operators follow operands

   Example: AB+

Evaluating arithmetic expressions is a common application of the stack Abstract Data Type (ADT). This process involves converting an infix expression (where operators are placed between operands, e.g., 3 + 4) into a postfix expression (where operators follow their operands, e.g., 3 4 +) and then evaluating the postfix expression using a stack. Below is a detailed explanation of how this is accomplished:

2.4.2 Infix to Postfix Conversion

The conversion from infix to postfix can be done using the Shunting Yard algorithm developed by Edsger Dijkstra. This algorithm uses a stack to hold operators and ensures that the output (postfix notation) is correctly formatted.

Algorithm:

- Initialize an empty stack for operators and an empty list for the output.
- Read the expression from left to right.
- For each token:
  - If the token is an operand, add it to the output list.
  - If the token is an operator:
    - While there is an operator at the top of the stack with greater precedence, pop operators from the stack to the output list.
    - Push the current operator onto the stack.
  - If the token is a left parenthesis, push it onto the stack.
  - If the token is a right parenthesis, pop from the stack to the output list until a left parenthesis is at the top of the stack. Discard the pair of parentheses.

- After reading the expression, pop all operators from the stack to the output list.

Example:

- Infix: 3 + 4 * 2 / (1 - 5)
- Postfix: 3 4 2 * 1 5 - / +

2.4.3 Evaluating Postfix Expression

Once the expression is converted to postfix notation, it can be evaluated using a stack:

Algorithm:

- Initialize an empty stack.
- Read the postfix expression from left to right.
- For each token:
  - If the token is an operand, push it onto the stack.
  - If the token is an operator, pop the required number of operands from the stack, apply the operator, and push the result back onto the stack.
- The value left in the stack is the result of the expression.

Example:

- Postfix: 3 4 2 * 1 5 - / +
- Evaluation:
  - Push 3
  - Push 4
  - Push 2
  - Pop 2 and 4, compute 4 * 2 = 8, push 8
  - Push 1
  - Push 5
  - Pop 5 and 1, compute 1 - 5 = -4, push -4
  - Pop -4 and 8, compute 8 / -4 = -2, push -2
  - Pop -2 and 3, compute 3 + (-2) = 1, push 1
  - Result is 1

Python Implementation

Here is a Python implementation of the conversion and evaluation process:

def infix_to_postfix(expression):

```
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2}
    stack = []
    output = []
    for token in expression:
        if token.isnumeric():
            output.append(token)
        elif token in precedence:
            while (stack and stack[-1] in precedence and
                   precedence[token] <= precedence[stack[-1]]):
                output.append(stack.pop())
            stack.append(token)
        elif token == '(':
            stack.append(token)
        elif token == ')':
            while stack and stack[-1] != '(':
                output.append(stack.pop())
            stack.pop()  # Remove '(' from the stack
    while stack:
        output.append(stack.pop())
    return output
```

def evaluate_postfix(expression):

```
    stack = []

    for token in expression:
        if token.isnumeric():
            stack.append(int(token))
        else:
            right_operand = stack.pop()
            left_operand = stack.pop()
            if token == '+':
                stack.append(left_operand + right_operand)
```

```
        elif token == '-':
            stack.append(left_operand - right_operand)
        elif token == '*':
            stack.append(left_operand * right_operand)
        elif token == '/':
            stack.append(left_operand / right_operand)
    return stack.pop()

# Example usage
infix_expr = "3 + 4 * 2 / ( 1 - 5 )".split()
postfix_expr = infix_to_postfix(infix_expr)
print("Postfix Expression:", ' '.join(postfix_expr))
result = evaluate_postfix(postfix_expr)
print("Evaluated Result:", result)
```

OUTPUT:

Postfix Expression: 3 4 2 * 1 5 - / +

Evaluated Result: 1.0

Let us sum up:

- **Types of Expressions**:
  - Infix: Operators between operands (e.g., A + B).
  - Prefix (Polish Notation): Operators precede operands (e.g., + A B).
  - Postfix (Reverse Polish Notation): Operators follow operands (e.g., A B +).
- **Infix to Postfix Conversion**:
  - Use the Shunting Yard algorithm, where operators are pushed onto a stack and operands are added to the output list. Parentheses are handled to maintain correct operator precedence.

- **Postfix Expression Evaluation**:
  - Using a stack, operands are pushed, and when an operator is encountered, operands are popped, the operation is performed, and the result is pushed back.
- **Algorithm Steps**:

- Conversion: Traverse the infix expression, handle operators based on precedence, and generate a postfix expression.
- Evaluation: Traverse the postfix expression, compute operations using the stack, and return the final result.

Check Your Progress

1. Which of the following is an example of an *Infix* expression?

A. A B + B. A + B C. + A B D. (A B +)

Answer: B. A + B

2. In a *Prefix* expression, the operator is:

A. Between the operands B. After the operands
C. Before the operands D. Not needed

Answer: C. Before the operands

3. Which of the following is in *Postfix* (Reverse Polish Notation) form?

A. A + B B. + A B C. AB + D. (A + B)

Answer: C. AB +

4. The expression +AB is an example of which type of notation?

A. Infix B. Prefix C. Postfix D. Hybrid

Answer: B. Prefix

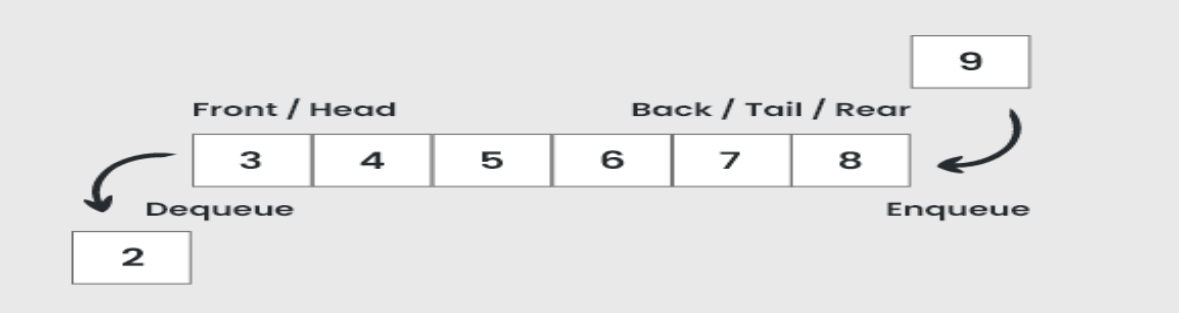
5. In which type of expression does the operator come between two operands?

A. Prefix B. Postfix C. Infix D. Suffix

Answer: C. Infix

2.5 Queue ADT (Abstract Data Type)

A queue is a fundamental data structure in computer science that follows the First-In-First-Out (FIFO) principle, meaning the first element added to the queue will be the first one to be removed. It is similar to a line of people waiting for a service, where the first person in line is the first one served.



2.6 Basic Operations on a Queue

(i) **Enqueue**: Adds an element to the end of the queue.

(ii) **Dequeue**: Removes an element from the front of the queue.

(iii) **Front (or Peek)**: Returns the front element without removing it from the queue.

(iv) **isEmpty**: Checks if the queue is empty.

(v) **isFull**: (For bounded queues) Checks if the queue is full.

(vi) **Size**: Returns the number of elements in the queue.

Detailed Explanation of Operations

(i) Enqueue Operation

The enqueue operation adds an element to the end of the queue.

Algorithm:

Step 1: Check if the queue is full (for a bounded queue).

Step 2: If the queue is not full, add the new element to the end of the queue.

Example:

If you have an empty queue [] and you enqueue 10, the queue becomes [10].

(ii) Dequeue Operation

The dequeue operation removes and returns the front element of the queue.

Algorithm:

Step 1: Check if the queue is empty.

Step 2: If the queue is not empty, remove and return the front element.

Example:

If the queue is [10, 20, 30] and you dequeue, the queue becomes [20, 30], and the dequeued element is 10.

(iii) Front (Peek) Operation

The front operation returns the front element without removing it from the queue.

Algorithm:

Step 1: Check if the queue is empty.

Step 2: If the queue is not empty, return the front element.

Example:

If the queue is [20, 30], the front operation will return 20.

(iv) isEmpty Operation

The isEmpty operation checks if the queue contains any elements.

Algorithm:

Step 1: Return true if the queue is empty, otherwise return false.

Example:

If the queue is [], isEmpty will return true. If the queue is [20, 30], isEmpty will return false.

(v) Size Operation

The size operation returns the number of elements currently in the queue.

Algorithm:

Step 1: Return the number of elements in the queue.

Example:

If the queue is [20, 30], size will return 2.

2.6.1.Queue Implementations

Queues can be implemented using different underlying data structures, such as arrays, linked lists, circular arrays, and priority queues.

Array-Based Queue

In an array-based queue, we use a fixed-size array along with two pointers (or indices), front and rear, to keep track of the front and rear positions in the queue.

Challenges: Managing the circular nature of the queue within a fixed-size array, where the rear pointer wraps around to the beginning of the array when it reaches the end.

Python Implementation:

```
class ArrayQueue:
    def __init__(self, capacity):
        self.capacity = capacity
        self.queue = [None] * capacity
        self.front = 0
        self.rear = 0
        self.size = 0

    def is_empty(self):
        return self.size == 0

    def is_full(self):
        return self.size == self.capacity

    def enqueue(self, item):
        if self.is_full():
            raise Exception("Queue is full")
        self.queue[self.rear] = item
        self.rear = (self.rear + 1) % self.capacity
        self.size += 1

    def dequeue(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        item = self.queue[self.front]
        self.front = (self.front + 1) % self.capacity
        self.size -= 1
        return item

    def peek(self):
        if self.is_empty():
```

```
            raise Exception("Queue is empty")
        return self.queue[self.front]

    def display(self):
        if self.is_empty():
            print("Queue is empty")
        else:
            idx = self.front
            for _ in range(self.size):
                print(self.queue[idx], end=" ")
                idx = (idx + 1) % self.capacity
            print()

# Example usage
queue = ArrayQueue(5)
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
queue.enqueue(4)
queue.display()
print(queue.dequeue())
queue.display()
```

OUTPUT:

```
1 2 3 4
1
2 3 4
```

Linked List-Based Queue

In a linked list-based queue, we use a linked list where each node points to the next node in the sequence. This implementation is dynamic and can grow or shrink as needed without the need for wrapping pointers.

Advantages: It overcomes the size limitation and wrapping issues present in array-based implementations.

Python Implementation:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedListQueue:
    def __init__(self):
        self.front = None
        self.rear = None

    def is_empty(self):
        return self.front is None

    def enqueue(self, item):
        new_node = Node(item)
        if self.rear:
            self.rear.next = new_node
        self.rear = new_node
        if not self.front:
            self.front = new_node

    def dequeue(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        item = self.front.data
        self.front = self.front.next
        if not self.front:
            self.rear = None
        return item
```

```
    def peek(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        return self.front.data

    def display(self):
        if self.is_empty():
            print("Queue is empty")
        else:
            current = self.front
            while current:
                print(current.data, end=" ")
                current = current.next
            print()
# Example usage
queue = LinkedListQueue()
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
queue.enqueue(4)
queue.display()
print(queue.dequeue())
queue.display()
```

output:

1 2 3 4

1

2 3 4

Let us sum up:

- **Queue Basics**: A queue is a First-In-First-Out (FIFO) data structure where the first element added is the first one removed. Basic operations include Enqueue (add), Dequeue (remove), Front/Peek (view first element), isEmpty (check if empty), isFull (check if full), and Size (number of elements).

- **Enqueue and Dequeue**:
  - **Enqueue** adds an element to the end if the queue is not full.
  - **Dequeue** removes and returns the front element if the queue is not empty.
- **Queue Implementations**:
  - **Array-based queue**: Uses a fixed-size array with front and rear pointers. It requires handling the circular nature of the queue.
  - **Linked list-based queue**: Uses a dynamic linked list, avoiding fixed size and wrapping issues.
- **Python Array-Based Queue**:
  - A circular queue is implemented using an array. Operations like enqueue, dequeue, and peek are performed using front and rear pointers, with size tracking.
- **Python Linked List-Based Queue**:
  - A queue is implemented using a linked list, where nodes point to each other. It dynamically grows or shrinks, and operations like enqueue, dequeue, and peek are handled via the front and rear nodes.

Check your progress:

1. Which operation adds an element to the queue?

   A. Dequeue

   B. Enqueue

   C. Peek

   D. isEmpty

   Answer: B. Enqueue

2. What does the Dequeue operation do?

   A. Adds an element to the front of the queue

   B. Removes an element from the rear

   C. Removes an element from the front of the queue

   D. Returns the rear element

   Answer: C. Removes an element from the front of the queue

3. In an array-based queue, which challenge needs to be handled?

   A. Managing pointers for wrapping around B. Unlimited queue size

   C. Linked nodes D. Storing the front element

   Answer: A. Managing pointers for wrapping around

4. Which implementation overcomes the size limitation of an array-based queue?

   A. Array-based Queue B. Linked List Queue

   C. Circular Queue D. Stack

   Answer: B. Linked List Queue

5. In the context of queue operations, what does *isEmpty* check?

   A. If the queue is full B. If the queue contains elements

   C. If the queue has reached capacity D. The size of the queue

   Answer: B. If the queue contains elements

2.7 Circular Queue

A circular queue is a type of queue in which the last position is connected back to the first position to make a circle. It overcomes the limitations of the standard array-based implementation of queues by reusing empty spaces. This helps in utilizing the space efficiently.

A circular queue, also known as a circular buffer or ring buffer, is a type of queue in which the last position is connected back to the first position to make a circle. This circular arrangement allows for efficient use of space and eliminates the need to shift elements, as is necessary in a linear queue implemented with arrays.

2.7.1 Key Characteristics of a Circular Queue

- **Fixed Size**: It has a fixed size, and the positions are reused.
- **Two Pointers**: It uses two pointers, front and rear, to keep track of the start and end of the queue.
- **Wrap-around**: When the rear pointer reaches the end of the queue, it wraps around to the beginning if there is space.

2.7.2 Basic Operations of a Circular Queue

(i) **Enqueue**: Adds an element to the end of the queue.

(ii) **Dequeue**: Removes an element from the front of the queue.

(iii) **Front**: Returns the front element without removing it.

(iv) **isEmpty**: Checks if the queue is empty.

(v) **isFull**: Checks if the queue is full.

Working Methodology

(i) Enqueue Operation

Step 1: Check if the queue is full. If ((rear + 1) % capacity == front), the queue is full.

Step 2: If the queue is not full, insert the element at the rear position and move the rear pointer to the next position (rear = (rear + 1) % capacity).

(ii) Dequeue Operation

Step 1: Check if the queue is empty. If (front == rear), the queue is empty.

Step 2: If the queue is not empty, remove the element at the front position and move the front pointer to the next position (front = (front + 1) % capacity).

(iii) Front (Peek) Operation

Step 1: Check if the queue is empty. If (front == rear), the queue is empty.

Step 2: If the queue is not empty, return the element at the front position.

(iv) isEmpty Operation

Step 1: If (front == rear), return true; otherwise, return false.

(v) isFull Operation

Step 1: If ((rear + 1) % capacity == front), return true; otherwise, return false**.**

Implementation in Python

Here's how you can implement a circular queue in Python:

```
class CircularQueue:
    def __init__(self, capacity):
        self.capacity = capacity
        self.queue = [None] * capacity
        self.front = 0
        self.rear = 0

    def enqueue(self, item):
        if self.isFull():
```

```
            raise OverflowError("Queue is full")
        self.queue[self.rear] = item
        self.rear = (self.rear + 1) % self.capacity
        print(f"Enqueued: {item}")

    def dequeue(self):
        if self.isEmpty():
            raise IndexError("Queue is empty")
        item = self.queue[self.front]
        self.queue[self.front] = None
        self.front = (self.front + 1) % self.capacity
        print(f"Dequeued: {item}")
        return item

    def frontElement(self):
        if self.isEmpty():
            raise IndexError("Queue is empty")
        return self.queue[self.front]

    def isEmpty(self):
        return self.front == self.rear

    def isFull(self):
        return (self.rear + 1) % self.capacity == self.front

    def size(self):
        if self.rear >= self.front:
            return self.rear - self.front
        return self.capacity - (self.front - self.rear)

# Example usage
cq = CircularQueue(5)
```

```
cq.enqueue(10)
cq.enqueue(20)
cq.enqueue(30)
print("Front element:", cq.frontElement())  # Output: 10
print("Queue size:", cq.size())             # Output: 3
cq.dequeue()
print("Front element:", cq.frontElement())  # Output: 20
print("Queue size:", cq.size())             # Output: 2
cq.enqueue(40)
cq.enqueue(50)
cq.enqueue(60)
print("Is queue full?", cq.isFull())        # Output: True
```

Visualization of Circular Queue Operations

1. **Initial State**

   Queue: [None, None, None, None, None]

   Front: 0

   Rear: 0

2. **Enqueue Operations**

   - **Enqueue 10**

     Queue: [10, None, None, None, None]

     Front: 0

     Rear: 1

   - **Enqueue 20**

     Queue: [10, 20, None, None, None]

     Front: 0

     Rear: 2

   - **Enqueue 30**

     Queue: [10, 20, 30, None, None]

     Front: 0

     Rear: 3

3. **Dequeue Operation**
   - **Dequeue**

     Queue: [None, 20, 30, None, None]

     Front: 1

     Rear: 3
4. **Front Operation**

     Queue: [None, 20, 30, None, None]

     Front: 1

     Rear: 3

     Front Element: 20
5. **isFull Operation**
   - **Enqueue 40**
   - **Enqueue 50**
   - **Enqueue 60**

     Queue: [60, 20, 30, 40, 50]

     Front: 1

     Rear: 0

     Is Full: True

This detailed explanation and the Python implementation should provide a comprehensive understanding of how a circular queue works and how to perform its basic operations.

Let us sum up

- **Definition:** A circular queue is a variation of the queue where the last position is connected to the first position, forming a circle, which allows efficient use of space.
- **Overflow Handling:** Unlike a linear queue, a circular queue overcomes the limitation of unused spaces by wrapping around when it reaches the end of the queue.
- **Pointers:** It uses two pointers—*front* (tracks the first element) and *rear* (tracks the last element). These pointers wrap around the queue.

- **Full and Empty Conditions:** A circular queue is considered full when (rear + 1) % capacity == front and empty when front == rear.
- **Applications:** Circular queues are used in scenarios like memory management, buffering in data streams, and scheduling processes.

Check your progress

1. **Which condition indicates that a circular queue is full?**

   A. front == rear

   B. rear + 1 == front

   C. (rear + 1) % capacity == front

   D. rear == front - 1

   Answer: C. (rear + 1) % capacity == front

2. **What is the primary advantage of a circular queue over a linear queue?**

   A. Easier to implement

   B. Avoids unused space when elements are dequeued

   C. Faster enqueue operation

   D. Can store unlimited elements

   Answer: B. Avoids unused space when elements are dequeued

3. **In a circular queue, what happens when the rear pointer reaches the last position?**

   A. The rear pointer moves back to the front

   B. The queue becomes full

   C. The rear pointer stays at the last position

   D. The queue is reset

   Answer:A. The rear pointer moves back to the front

4. **Which of the following is not an application of circular queues?**

   A. Memory management

   B. Process scheduling

   C. File management

   D. Buffering in data streams

   Answer: C. File management

5. **What is the condition for an empty circular queue?**

A. rear == front - 1

B. front == rear

C. rear == capacity - 1

D. front + 1 == rear

Answer: B. front == rear

2.8 Priority Queue

A priority queue is an abstract data type that operates similar to a regular queue or stack but with a key difference: each element in the queue has an associated priority. Elements with higher priority are dequeued before elements with lower priority, regardless of their arrival order. Priority queues are commonly used in scenarios where tasks or jobs need to be processed based on their urgency or importance.

2.8.1 Key Characteristics of Priority Queue

- **Priority Order**: Elements are removed from the queue based on their priority level.
- **Insertion**: Elements are added to the queue according to their priority.
- **No FIFO Order**: Unlike regular queues, elements with higher priority are dequeued first, irrespective of when they were added.
- **Implementation**: Can be implemented using various data structures like heaps, balanced binary search trees, or unordered arrays.

2.8.2 Operations in a Priority Queue

1. **Insertion**: Adds an element with its priority to the queue.
2. **Deletion**: Removes and returns the element with the highest priority.
3. **Peek**: Returns the highest priority element without removing it.
4. **isEmpty**: Checks if the priority queue is empty.
5. **Size**: Returns the number of elements in the priority queue.

Working Methodology

Priority queues can be implemented using various underlying data structures, but one of the most common and efficient implementations are using a binary heap. Here's a basic overview of how a priority queue typically works:

- **Insertion Operation**

  Step 1: Insert the new element into the priority queue along with its priority.

  Step 2: Adjust the position of the element to maintain the order (e.g., using heapify operation in a heap-based implementation).

- **Deletion (or Extraction) Operation**

  Step 1: Identify and remove the element with the highest priority from the priority queue.

  Step 2: Reorganize the remaining elements to ensure that the next highest priority element is ready to be dequeued efficiently.

- **Peek Operation**

  Step 1: Return the element with the highest priority without removing it from the queue.

- **isEmpty Operation**

  Step 1: Check if there are any elements in the priority queue.

- **Size Operation**

  Step 1: Return the number of elements currently in the priority queue.

Python Implementation:

```
import heapq
class PriorityQueue:
    def __init__(self):
        self.queue = []

    def is_empty(self):
        return len(self.queue) == 0

    def enqueue(self, item, priority):
        heapq.heappush(self.queue, (priority, item))

    def dequeue(self):
```

```
        if self.is_empty():
            raise Exception("Queue is empty")
        return heapq.heappop(self.queue)[1]

    def display(self):
        print(sorted(self.queue))

# Example usage
priority_queue = PriorityQueue()
priority_queue.enqueue("task1", 2)
priority_queue.enqueue("task2", 1)
priority_queue.enqueue("task3", 3)
priority_queue.display()
print(priority_queue.dequeue())
priority_queue.display()
```

OUTPUT:

```
[(1, 'task2'), (2, 'task1'), (3, 'task3')]
task2
[(2, 'task1'), (3, 'task3')]
```

Let us sum up:

- **Definition:** A priority queue is a type of queue where each element has a priority level, and elements are dequeued based on their priority rather than their order of insertion.
- **Enqueue Operation:** Elements are added to the queue with a priority value. Higher-priority elements are processed before lower-priority ones.
- **Dequeue Operation:** The element with the highest priority is dequeued first, regardless of its insertion time.
- **Types of Implementations:** Priority queues can be implemented using arrays, linked lists, binary heaps, or binary search trees.
- **Applications:** Priority queues are widely used in algorithms like Dijkstra's shortest path, Huffman encoding, and task scheduling systems.

Check your progress

1. In a priority queue, which element is dequeued first?
   A. The element that was enqueued first
   B. The element with the highest priority
   C. The element that was enqueued last
   D. Any random element
   Answer: B. The element with the highest priority

2. Which of the following data structures is commonly used to implement a priority queue?
   A. Stack B. Binary Heap
   C. Linked List D. Circular Queue
   Answer: B. Binary Heap

3. In a min-priority queue, which element is dequeued first?
   A. The element with the lowest priority
   B. The element with the highest priority
   C. The last inserted element
   D. The middle element
   Answer: A. The element with the lowest priority

4. What is the main difference between a priority queue and a regular queue?
   A. Elements in a priority queue are dequeued based on their insertion order
   B. Elements in a priority queue are dequeued based on their priority
   C. A priority queue has a limited size
   D. A priority queue does not allow dequeue operations

Answer: B. Elements in a priority queue are dequeued based on their priority

5. Which of the following is a common use case of a priority queue?
   A. Storing web pages
   B. Managing task scheduling
   C. Implementing LIFO order
   D. Storing real-time logs

Answer: B. Managing task scheduling

2.9 Deque (Double-Ended Queue)

A double-ended queue (deque), pronounced as "deck", is a versatile data structure that allows insertion and deletion of elements from both ends. It combines the features of stacks (Last In, First Out - LIFO) and queues (First In, First Out - FIFO) into a single data structure. Double-ended queues are commonly used in scenarios where efficient insertion and deletion operations are required at both ends of the queue.

2.9.1 Key Characteristics of Deque

1. **Double-Ended Operations**: Elements can be added or removed from both the front and the rear of the deque.
2. **Dynamic Size**: Unlike arrays with fixed sizes, deques can dynamically resize themselves to accommodate new elements.
3. **Versatility**: Can function as a queue (FIFO), stack (LIFO), or a hybrid depending on the application needs.

2.9.2 Operations in a Deque

(i) **Insertion (Addition)**

- **addFront(item)**: Adds an element to the front of the deque.
- **addRear(item)**: Adds an element to the rear of the deque.

(ii) **Deletion (Removal)**

- **removeFront()**: Removes and returns the element from the front of the deque.
- **removeRear()**: Removes and returns the element from the rear of the deque.

(iii) **Access (Peek)**

- **peekFront()**: Returns the element at the front of the deque without removing it.
- **peekRear()**: Returns the element at the rear of the deque without removing it.

(iv) **Size and Empty Check**

- **isEmpty()**: Checks if the deque is empty.

- **size()**: Returns the number of elements in the deque.

Working Methodology

Deques can be implemented using various underlying data structures, such as doubly linked lists or arrays. Here's a basic overview of how a deque typically works:

(i) Insertion Operations

- **addFront(item)**:
  - Create a new node (in the case of a doubly linked list) or adjust array indices (in the case of arrays) to add the item at the front.
- **addRear(item)**:
  - Add the item at the end of the deque. Adjust the tail pointer or array indices accordingly.

(ii) Deletion Operations

- **removeFront():**
  - Remove and return the item at the front of the deque. Adjust head pointer or array indices accordingly.
- **removeRear():**
  - Remove and return the item at the rear of the deque. Adjust tail pointer or array indices accordingly.

(iii) Access Operations

- **peekFront():**
  - Return the item at the front of the deque without removing it.
- **peekRear()**:
  - Return the item at the rear of the deque without removing it.

(iv)Size and Empty Check Operations

- **isEmpty()**:
  - Check if the deque is empty by verifying the size.
- **size()**:
  - Return the current number of elements in the deque.

Python Implementation:

```
class Deque:
  def __init__(self):
```

```
        self.deque = []

    def is_empty(self):
        return len(self.deque) == 0

    def enqueue_front(self, item):
        self.deque.insert(0, item)

    def enqueue_rear(self, item):
        self.deque.append(item)

    def dequeue_front(self):
        if self.is_empty():
            raise Exception("Deque is empty")
        return self.deque.pop(0)

    def dequeue_rear(self):
        if self.is_empty():
            raise Exception("Deque is empty")
        return self.deque.pop()

    def display(self):
        print(self.deque)

# Example usage
deque = Deque()
deque.enqueue_rear(1)
deque.enqueue_rear(2)
deque.enqueue_front(0)
deque.display()
print(deque.dequeue_front())
deque.display()
```

OUTPUT:

[0, 1, 2]

0

[1, 2]

Let us sum up

- **Dequeue Definition:** The dequeue operation removes and returns the front element of a queue, following the FIFO principle.
- **Queue Check:** Before dequeuing, it's essential to check if the queue is empty. Attempting to dequeue from an empty queue results in an error.
- **Effect on Queue:** After a dequeue operation, the queue's front pointer moves to the next element, reducing the size of the queue by one.
- **Example of Dequeue:** If the queue is [10, 20, 30], dequeuing will remove 10, and the new queue becomes [20, 30].
- **Python Implementation:** The dequeue() method typically decreases the size of the queue, adjusts the front pointer, and returns the dequeued element.

Check your progress

1. What does the dequeue operation do?
   A. Adds an element to the queue
   B. Removes an element from the rear of the queue
   C. Removes and returns the front element of the queue
   D. Returns the size of the queue
   Answer: C. Removes and returns the front element of the queue

2. What is the first step in the dequeue operation?
   A. Add an element to the queue B. Check if the queue is full
   C. Check if the queue is empty D. Return the front element
   Answer:C. Check if the queue is empty

3. If the queue is [10, 20, 30], what will be the result after a dequeue operation?
   A. The queue becomes [10, 30]

B. The queue becomes [20, 30]

C. The queue becomes [10]

D. The queue remains unchanged

Answer: B. The queue becomes [20, 30]

4. In an array-based queue, what happens to the front pointer after a dequeue operation?

   A. It moves to the previous element

   B. It moves to the next element

   C. It remains unchanged

   D. It wraps around to the rear

   Answer: B. It moves to the next element

5. What happens if you try to dequeue from an empty queue?

   A. The last element is returned

   B. The queue size increases

   C. An error or exception is raised

   D. The queue resets

   Answer: C. An error or exception is raised

2.10 Applications of Queues

(i) **CPU Scheduling:** Queues manage processes in scheduling algorithms like Round Robin, ensuring fair CPU time distribution.

(ii) **Disk Scheduling:** Queues handle disk I/O requests, optimizing read/write operations.

(iii) **Breadth-First Search (BFS):** In graph traversal, BFS uses a queue to explore nodes level by level.

(iv) **Print Spooling:** Print jobs are managed in a queue, ensuring first-come, first-served order.

(v) **Network Buffer:** Data packets are managed in queues for proper transmission order.

(vi) **Call Center Systems:** Customer service systems use queues to manage incoming calls, ensuring fair handling.

(vii) **Simulation of Real-World Systems:** Queues are used in simulations of real-world systems like checkout lines, traffic management, and service centers.

Summary

Queues are essential data structures used in various applications for their FIFO properties. Implementations can vary, including array-based, linked list-based, circular queues, priority queues, and deques, each suitable for different use cases and optimizations. Understanding these concepts and their applications is crucial for effective problem-solving in computer science and real-world systems.

Activities

Activity 1: Compare the performance of array-based and linked list-based stack implementations.

- **Objective**: Understand the differences in performance between array-based and linked list-based stack implementations.
- **Steps**:
    1. Implement both stack types in Python.
    2. Measure the time complexity for push, pop, and peek operations.
    3. Compare the memory usage of both implementations.
    4. Write a report summarizing the findings.
- **Expected Outcome**: Students will learn about the trade-offs between different implementations in terms of performance and memory usage.

Activity 2: Use animations or interactive elements to illustrate push, pop, and peek operations.

- **Objective**: Visualize the operations of a stack to enhance understanding.
- **Steps**:
    1. Use an animation tool (like Pygame for Python) to create a visual representation of a stack.
    2. Implement animations for push, pop, and peek operations.
    3. Allow users to input values and see the operations in action.
- **Expected Outcome**: Students will gain a clearer understanding of stack operations through visual learning.

Activity 3: Implement a Stack:

- Write code to implement stack operations (push, pop, peek, isEmpty, isFull).

- Test the stack with different data types (integers, strings, objects).

Activity 4: Arithmetic Expression Evaluation:

- Implement an algorithm to evaluate postfix expressions using a stack.
- Extend the implementation to support infix to postfix conversion.

Activity 5: Implement a Queue:

- Write code to implement queue operations (enqueue, dequeue, front, isEmpty, isFull).
- Test the queue with different data types.

Activity 6: Simulate a Waiting Line:

- Use a queue to simulate a real-world waiting line (e.g., at a bank or restaurant).

Activity 7: Circular Queue Implementation:

- Implement a circular queue and demonstrate its operations.
- Discuss the advantages of using a circular queue over a linear queue.

Points to Remember

- Stacks follow Last In First Out (LIFO) principle.
- Key operations: push (add item), pop (remove item), peek (view top item), isEmpty (check if stack is empty), isFull (check if stack is full).
- Common applications: function call management in recursion, expression evaluation, undo mechanisms in text editors.
- Queues follow First In First Out (FIFO) principle.
- Key operations: enqueue (add item), dequeue (remove item), front (view front item), isEmpty (check if queue is empty), isFull (check if queue is full).
- Types of queues: simple queue, circular queue, priority queue, double-ended
- queue (deque).

Questions

1. What is the difference between a stack and a queue?
2. Explain the LIFO principle with an example.
3. How can a stack be used to evaluate arithmetic expressions?
4. What are the benefits and limitations of using a stack?

5. Write the algorithm for converting an infix expression to a postfix expression.
6. What is the difference between a linear queue and a circular queue?
7. Explain the FIFO principle with an example.
8. How does a priority queue differ from a regular queue?
9. What are the benefits and limitations of using a queue?
10. Describe a real-world scenario where a queue would be useful.

Glossary

- **Stack:** A data structure that follows the LIFO principle.
- **Push:** Operation to add an element to the stack.
- **Pop:** Operation to remove the top element from the stack.
- **Peek:** Operation to view the top element without removing it.
- **LIFO:** Last In First Out, describes the order in which elements are accessed in a stack.
- **Queue:** A data structure that follows the FIFO principle.
- **Enqueue:** Operation to add an element to the rear of the queue.
- **Dequeue:** Operation to remove the front element from the queue.
- **FIFO:** First In First Out, describes the order in which elements are accessed in a queue.
- **Circular Queue:** A type of queue where the end of the queue wraps around to the beginning, forming a circle.
- **Priority Queue:** A type of queue where elements are removed based on priority rather than order of arrival.
- **Deque:** Double-ended queue, allows elements to be added or removed from both ends.

Further Reading and References

1. Textbooks:

- **"Data Structures and Algorithm Analysis in C" by Mark Allen Weiss**:
  - This book offers clear explanations of stack and queue operations, their implementations, and various applications.
- **"Data Structures Using C" by Reema Thareja**:

  - A well-structured book that covers stacks, queues, circular queues, priority queues, and their applications.
- **"Fundamentals of Data Structures in C" by Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed**:
  - This classic book covers both stacks and queues in depth, with an emphasis on ADTs and practical applications.
- **"Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein**:
  - Focuses on data structures and algorithms, with good coverage of stacks, queues, and their role in algorithm design.

2. Online Materials:

- **GeeksforGeeks (Stacks and Queues)**:
  - Stacks: Operations and Applications
  - Queues: Operations, Circular Queue, and Applications
  - GeeksforGeeks provides comprehensive tutorials, code examples, and explanations for both stacks and queues.
- **TutorialsPoint (Stack and Queue ADTs)**:
  - Stack ADT
  - Queue ADT
  - Tutorialspoint is known for beginner-friendly content with clear explanations, diagrams, and implementations in multiple languages.
- **Programiz (Data Structures)**:
  - Stack in Data Structure
  - Queue in Data Structure
  - Programiz offers easy-to-understand tutorials with code examples and visualizations for both stack and queue implementations.

3. Video Materials:

- **Neso Academy (YouTube)**:
  - Stack Data Structure
  - Queue Data Structure

- Neso Academy provides detailed video lectures on both stacks and queues with thorough explanations of operations, applications, and examples.

- **MyCodeSchool (YouTube)**:
  - Stack Operations and Applications
  - Queue Operations and Circular Queue
  - MyCodeSchool offers excellent visual tutorials with practical coding examples on stack and queue operations.
- **CS50 Harvard Course (EdX and YouTube)**:
  - Stacks and Queues Lecture
  - CS50 is a free, high-quality course on data structures and algorithms, and their stack/queue lecture covers everything from basic concepts to complex applications.

UNIT 3: TREE ADT

Tree ADT-tree traversals-Binary Tree ADT-expression trees-applications of trees-binary search tree ADT- Threaded Binary Trees-AVL Trees- B-Tree- B+ Tree – Heap-Applications of heap.

Section No	Topic	Page No
	Objectives	
3.1	**Tree ADT**	
3.1.1	Terminologies in Tree Data Structure	
3.1.2	Properties of Trees	
3.1.3	Operations in Tree ADT	
3.1.4	Applications of Trees	
3.2	**Tree Traversal**	
3.2.1	Preorder Traversal	
3.2.2	Inorder Traversal	
3.2.3	Postorder Traversal	
3.2.4	Applications of Tree Traversals	
3.3	**Binary Tree ADT**	
3.3.1	Properties of Binary Trees	
3.3.2	Operations in Binary Tree ADT	
3.3.3	Implementations of Binary Tree ADT	
3.3.4	Types of Binary Trees	
3.3.5	Applications of Binary Trees	
3.4	**Expression Tree**	
3.4.1	Construction of Expression Tree	
3.5	**Applications of Tree**	
3.6	**Binary Search Tree**	
3.6.1	Operations in Binary Search Tree (BST)	
3.6.2	Example Usage	
3.7	**Threaded Binary Tree**	
3.7.1	Types of Threaded Binary Tree	

3.7.2	Advantages of Threaded Binary Tree	
3.7.3	Disadvantages of Threaded Binary Tree	
3.8	**AVL Trees: Operations and Methodologies**	
3.8.1	Key Properties of AVL Trees	
3.8.2	Operations in AVL Trees	
3.9	**B-Tree**	
3.9.1	Key Characteristics	
3.9.2	Operations	
3.9.3	Use Cases	
3.9.4	Advantages	
3.10	**B+ Tree**	
3.10.1	Key Characteristics	
3.10.2	Operations	
3.10.3	Use Cases	
3.10.4	Advantages	
3.11	**Heap Tree**	
3.11.1	Key Characteristics	
3.11.2	Operations	
3.11.3	Use Cases	
3.11.4	Advantages	
3.11.5	Applications of Heap	
	Activities	
	Points to Remember	
	Questions	
	Glossary	
	Further Reading and References	

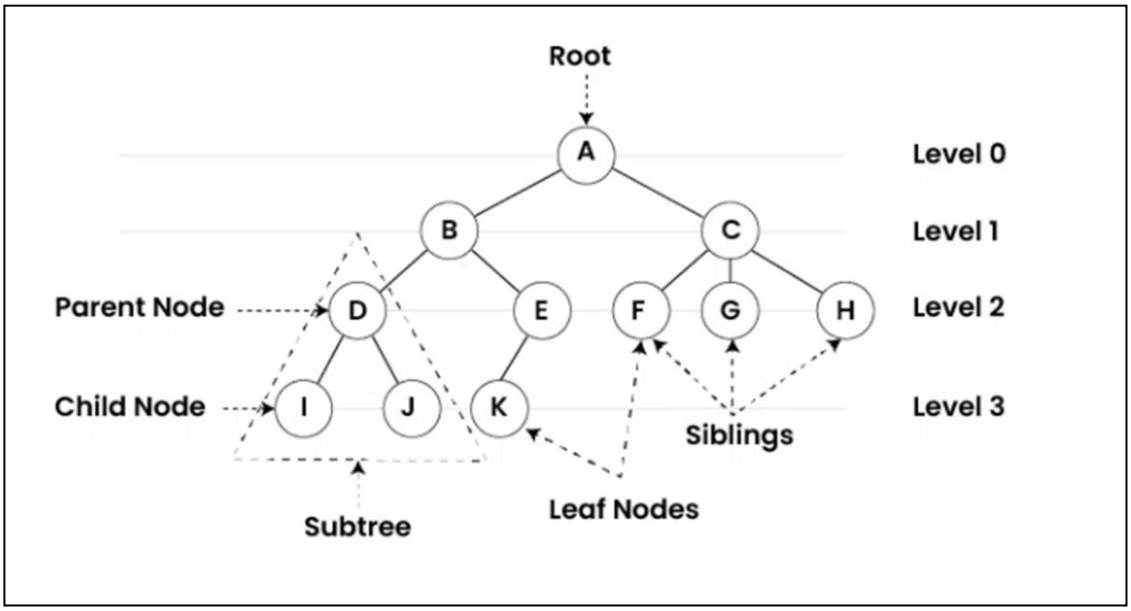
Objectives:

- This unit aims to understand the fundamental concepts, operations, and applications of various types of trees in computer science, including Binary Trees, Binary Search Trees (BST), Expression Trees, Threaded Binary Trees, AVL Trees, B-Trees, B+ Trees, and Heaps.
- The objective includes gaining proficiency in implementing and manipulating these tree structures, analyzing their time complexities for operations, and exploring their practical applications in data storage, searching, sorting, and other computational tasks.

3.1 Tree ADT (Abstract Data Type)

Tree Data Structure is a non-linear data structure in which a collection of elements known as nodes are connected to each other via edges such that there exists exactly one path between any two nodes.

A Tree Abstract Data Type (ADT) is a hierarchical data structure that consists of nodes connected by edges. It starts with a root node and each node can have zero or more child nodes, forming a structure resembling a tree in nature. Trees are widely used in computer science for organizing data efficiently, enabling fast searches, insertions, and deletions



3.1.1 Terminologies In Tree Data Structure

- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. {B} is the parent node of {D, E}.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: {D, E} are the child nodes of {B}.
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. {K, L, M, N, O, P, G} are the leaf nodes of the tree.
- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {A,B} are the ancestor nodes of the node {E}
- **Descendant:** A node x is a descendant of another node y if and only if y is an ancestor of y.
- **Sibling:** Children of the same parent node are called siblings. {D,E} are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level 0.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbor of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree:** Any node of the tree along with its descendant.

3.1.2 Properties of Trees:

- **Hierarchical Structure:** Nodes are arranged in levels or layers, with each level containing nodes that are connected by edges originating from nodes at the level directly above.
- **Single Path:** There is a unique path between any pair of nodes in the tree, starting from the root node.
- **Acyclic:** Trees do not contain cycles or loops. Moving from node to node via edges will never bring you back to a node you have visited before.

3.1.3 Operations in Tree ADT:

(i) **Traversal:** Visit nodes in a specific order, such as Preorder, Inorder, Postorder, or Level-order traversal.

(ii) **Insertion:** Adding a new node to the tree structure.

(iii) **Deletion:** Removing a node and potentially rearranging the structure to maintain tree properties.

(iv) **Search:** Finding a particular node based on its value.

(v) **Modification:** Updating the value of a node.

(vi) **Height Calculation:** Determining the maximum number of edges from the root to any leaf node (also known as the height of the tree).

3.1.4 Applications of Trees:

- **File Systems:** Representing directory structures where each directory can contain files or other directories.
- **Binary Search Trees (BST):** Efficient searching, insertion, and deletion operations with average time complexity of O(log n) for balanced trees.
- **Expression Trees:** Representing mathematical expressions in a way that facilitates evaluation.
- **Heap Data Structure:** Used in priority queues and heap sort algorithms.
- **Decision Trees:** Used in machine learning for decision-making processes.

Implementations:

Trees can be implemented in various ways depending on the specific application and requirements:

- **Linked Representation:** Each node is an object containing a data element and references to its child nodes.
- **Array Representation:** Especially for complete binary trees, nodes are stored in an array based on their level and position.

Let us Sum Up:

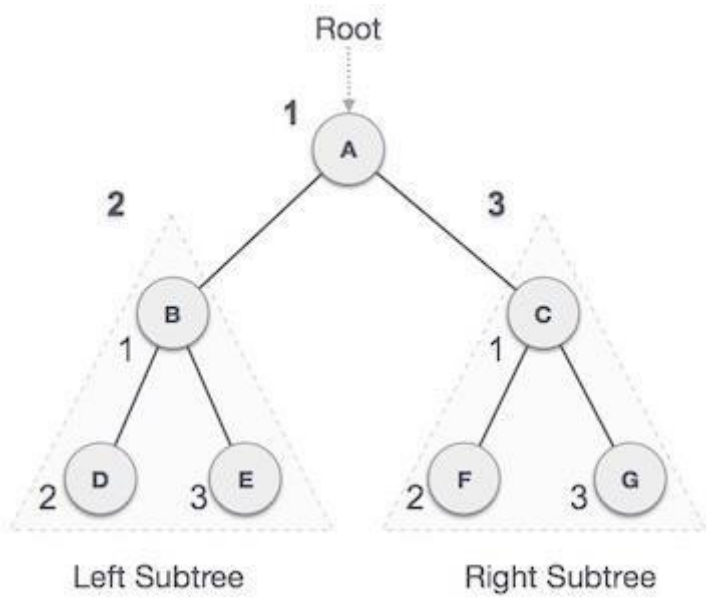
- Tree ADT Structure: A tree is a non-linear, hierarchical data structure with nodes connected by edges. It starts with a root node, and each node can have multiple child nodes, forming a tree-like structure with exactly one path between any two nodes.
- Key Terminologies: Important concepts include parent and child nodes, root node (the topmost node), leaf nodes (nodes with no children), internal nodes (with children), ancestors, descendants, and siblings.
- Tree Properties: Trees have a hierarchical structure, a single unique path between nodes, and are acyclic, meaning they do not contain cycles or loops.
- Operations on Trees: Common operations include traversal (Preorder, Inorder, Postorder, and Level-order), insertion, deletion, searching, modification, and height calculation.
- Applications: Trees are used in file systems, binary search trees (BSTs), expression trees, heap data structures, and decision trees, making them essential in fields like machine learning, data structures, and algorithms.

3.2 Tree Traversal

Tree traversal refers to the process of visiting (or accessing) all nodes in a tree data structure in a specific order. There are several ways to traverse a tree, each suitable for different purposes or tasks. The most common tree traversal methods include:

3.2.1 Preorder Traversal:

- Visit the root node first, then recursively traverse the left subtree and finally the right subtree.
- Order: Root, Left, Right.
- Example: For a tree with root A, left child B, and right child C, the preorder traversal would be A -> B -> C.

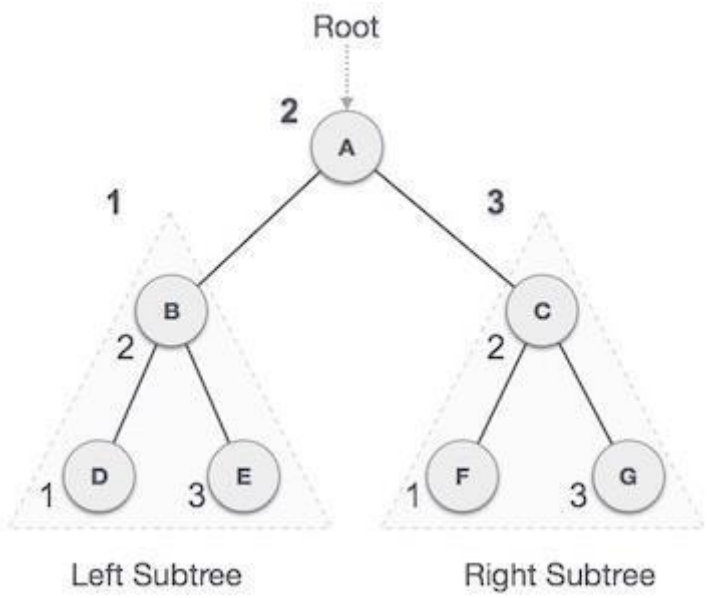


We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

Preorder Traversal: A → B → D → E → C → F → G

3.2.2 Inorder Traversal:

- Traverse the left subtree recursively, visit the root node, and then traverse the right subtree recursively.
- Order: Left, Root, Right.
- Example: For a tree with root A, left child B, and right child C, the inorder traversal would be B -> A -> C.
- Note: Inorder traversal of a binary search tree (BST) results in nodes being visited in ascending order (if the tree elements are ordered).

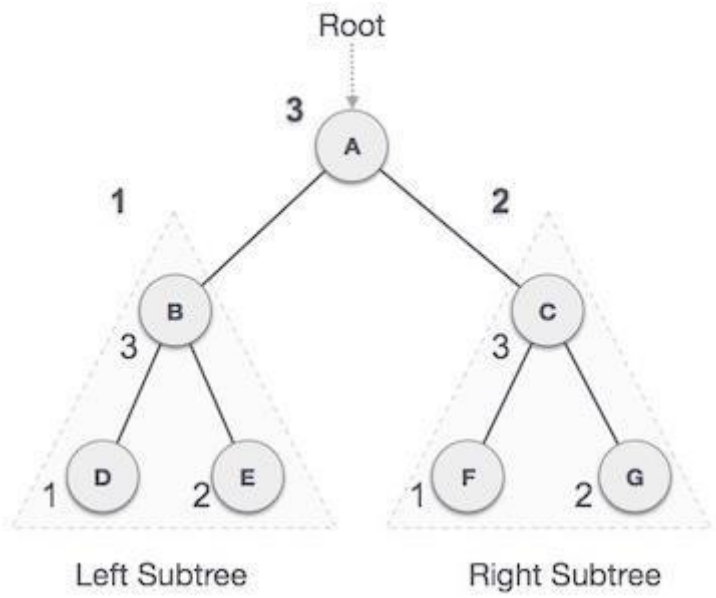


We start from **A**, and following in-order traversal, we move to its left subtree **B**.**B** is also traversed in-order. The process goes on until all the nodes are visited. The output of in-order traversal of this tree will be −

Inorder Traversal: D → B → E → A → F → C → G

3.2.3 Postorder Traversal:

- Traverse the left subtree recursively, then the right subtree recursively, and finally visit the root node.
- Order: Left, Right, Root.
- Example: For a tree with root A, left child B, and right child C, the postorder traversal would be B -> C -> A.



We start from **A**, and following pre-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

Post order Traversal: D → E → B → F → G → C → A

Uses and Applications of Tree Traversals:

- **Searching and Retrieval:** Traversals are essential for searching specific nodes or values within a tree.
- **Expression Evaluation:** In expression trees, traversals help evaluate mathematical expressions by visiting nodes in a specific order (e.g., inorder for infix expressions).
- **Copying/Moving Tree Structures:** Traversals facilitate copying or moving subtrees or entire trees.
- **Sorting:** In BSTs, inorder traversal yields elements in sorted order, making it useful for implementing efficient sorting algorithms like in-order tree sort.

Let us Sum up:

- Tree Traversal Overview: Tree traversal involves visiting all nodes in a tree in a specific order. The main methods are Preorder, Inorder, and Postorder traversal, each serving different purposes.
- Preorder Traversal: Visit the root node first, followed by the left subtree and then the right subtree (Root → Left → Right). Example: A → B → D → E → C → F → G.
- Inorder Traversal: Traverse the left subtree first, visit the root, then traverse the right subtree (Left → Root → Right). This results in ascending order traversal for binary search trees. Example: D → B → E → A → F → C → G.
- Postorder Traversal: First traverse the left subtree, then the right subtree, and finally visit the root node (Left → Right → Root). Example: D → E → B → F → G → C → A.
- Applications of Traversal: Tree traversals are essential for searching nodes, evaluating expressions, copying/moving tree structures, and sorting, especially in binary search trees (BSTs).

Check your progress:

1. In a Tree ADT, what is the name of the topmost node?

   A. Leaf B. Child C. Root D. Parent

   Answer: C. Root

2. Which of the following is true about a binary tree?

   A. Every node can have only one child

   B. Every node can have at most two children

   C. A binary tree has no limit on the number of children per node

   D. Every node has exactly two children

   Answer: B. Every node can have at most two children

3. In a Tree ADT, the nodes that do not have any children are called:

   A. Root nodes B. Internal nodes

   C. Leaf nodes D. Sibling nodes

Answer: C. Leaf nodes

4. What is the relationship between nodes in a tree if they share the same parent?

   A. Ancestors B. Descendants

   C. Siblings D. Neighbors

   Answer: C. Siblings

5. Which traversal technique visits the root node first, followed by the left subtree, and then the right subtree?

   A. Inorder traversal B. Preorder traversal

   C. Postorder traversal D. Level-order traversal

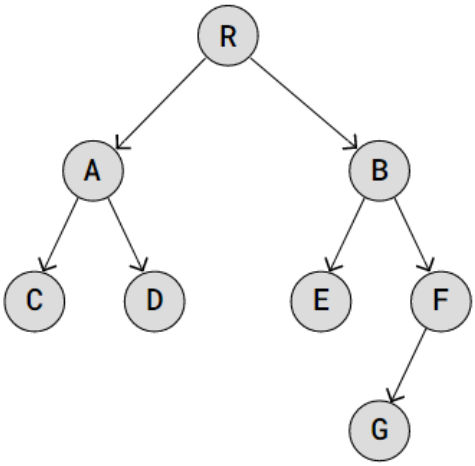
   Answer:B. Preorder traversal

3.3 Binary Tree ADT (Abstract Data Type)

A Binary Tree is a hierarchical data structure consisting of nodes, where each node has at most two children, referred to as the left child and the right child. Binary Trees are fundamental in computer science and are used as the foundation for various other data structures and algorithms.

3.3.1 Properties of Binary Trees:

- **Binary Nature:** Each node in a binary tree can have at most two children.
- **Shape and Structure:** The structure of a binary tree can vary widely depending on how nodes are arranged. This includes balanced trees, skewed trees, complete trees, etc.
- **Depth:** The depth of a node is the number of edges from the root to that node.
- **Height (Depth):** The height of a tree is the maximum depth of any node in the tree.



3.3.2. Operations in Binary Tree ADT:

1. **Traversal:**
    - **Inorder Traversal:** Visit left subtree, then the root, then right subtree.
    - **Preorder Traversal:** Visit the root, then the left subtree, then the right subtree.
    - **Postorder Traversal:** Visit left subtree, then right subtree, then the root.
    - **Level-order Traversal (Breadth-First Traversal):** Visit nodes level by level, from left to right.
2. **Search:** Find a specific element (node) in the tree.
3. **Insertion:** Add a new element (node) to the tree.
4. **Deletion:** Remove a node from the tree while maintaining binary tree properties.
5. **Height Calculation:** Determine the height of the tree (the number of edges on the longest path from the root to a leaf).
6. **Traversal Algorithms:** Implementing various traversal methods using recursion or iteration.

3.3.3 Implementations of Binary Tree ADT:

- **Linked Representation:** Nodes are represented using objects (or structs) with pointers to left and right child nodes.
- **Array Representation:** A binary tree can also be represented using an array where the left child of a node at index i is stored at index 2*i + 1 and the right child is stored at index 2*i + 2.

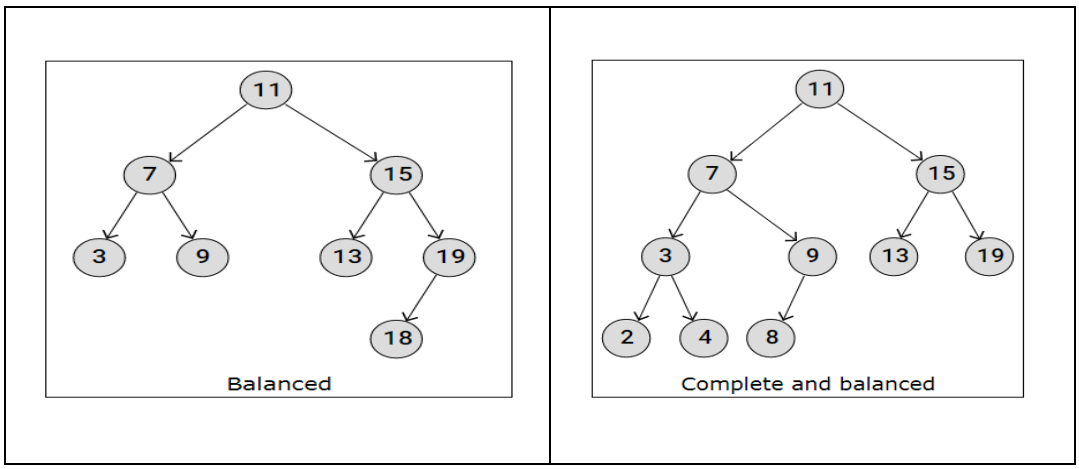
3.3.4 Types of Binary Trees

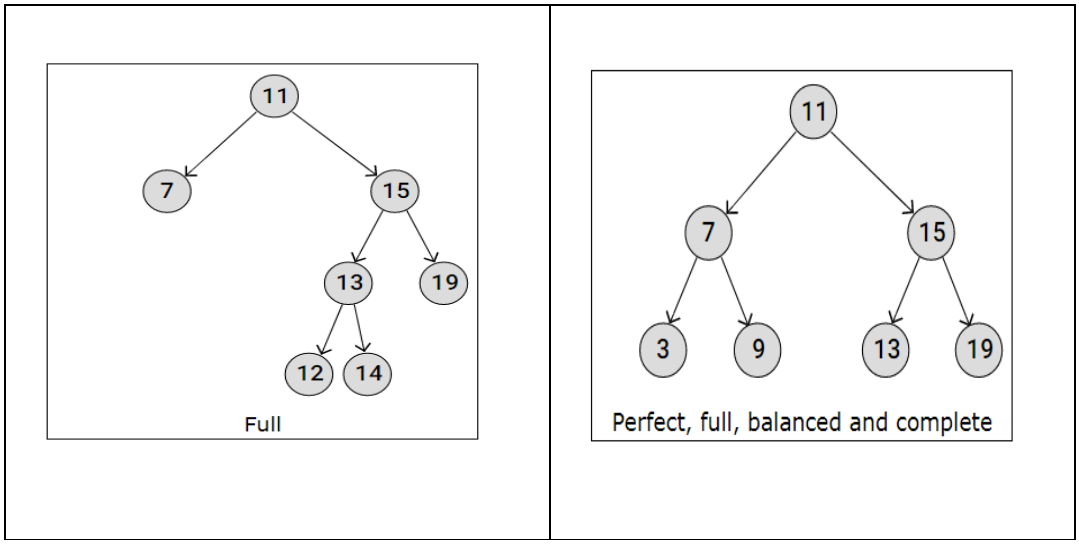
There are different variants, or types, of Binary Trees worth discussing to get a better understanding of how Binary Trees can be structured.

- A **balanced** Binary Tree has at most 1 in difference between its left and right subtree heights, for each node in the tree.

- A **complete** Binary Tree has all levels full of nodes, except the last level, which is can also be full, or filled from left to right. The properties of a complete Binary Tree means it is also balanced.

- A **full** Binary Tree is a kind of tree where each node has either 0 or 2 child nodes.





- A **perfect** Binary Tree has all leaf nodes on the same level, which means that all levels are full of nodes, and all internal nodes have two child nodes. The properties of a perfect Binary Tree means it is also full, balanced, and complete.

3.3.5 Applications of Binary Trees:

- **Binary Search Trees (BSTs):** Used for efficient searching, insertion, and deletion operations.
- **Expression Trees:** Represent mathematical expressions for efficient evaluation.
- **Heap Data Structure:** Used in priority queues and heapsort algorithms.
- **Binary Tree Serialization/Deserialization:** Storing and reconstructing binary trees in memory or on disk.

Let us sum up:

- **Binary Tree Overview**: A binary tree is a hierarchical data structure where each node has at most two children (left and right). It serves as the foundation for various other data structures and algorithms in computer science.
- **Key Properties**: Binary trees can vary in structure (balanced, skewed, complete, etc.), and important concepts include node depth (distance from root) and tree height (maximum depth).

- **Binary Tree Operations**: Common operations include traversal (Inorder, Preorder, Postorder, and Level-order), searching, insertion, deletion, and height calculation.
- **Types of Binary Trees**: Variants include balanced (height difference ≤ 1), complete (fully filled except possibly the last level), full (nodes have 0 or 2 children), and perfect (all levels filled, balanced, and full).
- **Applications**: Binary trees are used in Binary Search Trees (BSTs), expression evaluation, heap data structures, and serialization/deserialization for memory storage. They are crucial in areas like database systems, AI, and algorithm design.

Check your progress

1. In a binary tree, which node can have at most two children?
   A. Root node
   B. Internal node
   C. Leaf node
   D. All nodes

   Answer: D. All nodes

   Answer: B. Every node can have at most two children
2. Which of the following is a characteristic of a full binary tree?
   A. Every node has either 0 or 1 child
   B. Every node has exactly two children, except for the leaves
   C. All nodes have exactly two children
   D. All levels are fully filled except possibly the last level
   Answer: B. Every node has exactly two children, except for the leaves
3. In a binary tree, the nodes of which level are at the maximum distance from the root?
   A. Root level B. Intermediate level C. Leaf level D. Middle level
   Answer: C. Leaf level

4. What is the key property of a binary search tree (BST)?

   A. Every node has exactly two children

   B. The left subtree of a node contains only nodes with values less than the node's value, and the right subtree only nodes with values greater

   C. All nodes are at the same level

   D. The binary tree is perfectly balanced

Answer B. The left subtree of a node contains only nodes with values less than the node's value, and the right subtree only nodes with values greater

5. Which traversal method of a binary tree visits nodes in the following order: left subtree, root node, right subtree?

   A. Preorder traversal

   B. Postorder traversal

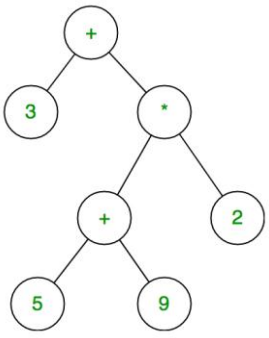
   C. Inorder traversal

   D. Level-order traversal

   Answer C. Inorder traversal

3.4 Expression Tree

The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand so for example expression tree for 3 + ((5+9)*2) would be:



Inorder traversal of expression tree produces infix version of given postfix expression (same with postorder traversal it gives postfix expression)

Construction of Expression Tree:

Now For constructing an expression tree we use a stack. We loop through input expression and do the following for every character.

1. If a character is an operand push that into the stack
2. If a character is an operator pop two values from the stack make them its child and push the current node again.

In the end, the only element of the stack will be the root of an expression tree.

Let us sum up:

- **Expression Tree Definition**: An expression tree is a binary tree where internal nodes represent operators, and leaf nodes represent operands. For example, the expression "3 + ((5 + 9) * 2)" can be represented as a tree.
- **Traversal and Expression Forms**: Inorder traversal of an expression tree yields the infix version of the expression, while postorder traversal yields the postfix version.
- **Construction Process**: Expression trees are built using a stack. Operands are pushed onto the stack, and when an operator is encountered, two operands are popped, and the operator becomes their parent node.
- **Operand Handling**: Each operand is treated as a leaf node, while operators form the internal structure of the tree, organizing the expression hierarchy.
- **Final Tree Structure**: Once the expression is fully processed, the remaining item in the stack is the root of the completed expression tree.

Check your progress

1. What does an expression tree represent?
   A. A sequence of operators and operands in a linear format
   B. A hierarchical structure for evaluating arithmetic expressions
   C. A way to store data in sorted order
   D. A graph structure for representing algorithms

Answer: B. A hierarchical structure for evaluating arithmetic expressions

2. In an expression tree, where are the operators typically located?

   A. At the leaves

   B. At the nodes

   C. At the root

   D. At the end of the tree

   Answer: B. At the nodes

3. What is the result of the expression tree for the infix expression A + B * C?

   A. The tree has + as the root, with A and * as children, and B and C as children of *

   B. The tree has * as the root, with A and + as children, and B and C as children of +

   C. The tree has + as the root, with B and C as children, and A as a child of *

   D. The tree has B and C as the root with * as a child, and A as a child of +

   Answer: A. The tree has + as the root, with A and * as children, and B and C as children of *

4. In a postfix expression tree, how are operators and operands arranged?

   A. Operators are at the top, and operands are at the leaves

   B. Operands are at the top, and operators are at the leaves

   C. Operators are at the leaves, and operands are at the top

   D. Operands and operators are interspersed without a fixed arrangement

   Answer: A. Operators are at the top, and operands are at the leaves

5. Which traversal method would you use to obtain the postfix expression from an expression tree?

   A. Preorder traversal

   B. Inorder traversal

   C. Postorder traversal

   D. Level-order traversal

   Answer: C. Postorder traversal

3.5 Applications of Tree

Trees have numerous applications across various domains in computer science and beyond. Their hierarchical structure and efficient operations make them suitable for a wide range of tasks. Here are some common applications of trees:

1. File Systems:

Trees are widely used to represent file systems where each directory or folder can contain files or other directories. This hierarchical structure allows for efficient organization, navigation, and manipulation of files.

2. Database Systems:

In database systems, trees are used in the form of B-Trees and B+ Trees for indexing. These tree structures enable fast search, insertion, and deletion operations, making them essential for maintaining data integrity and optimizing query performance.

3. Binary Search Trees (BSTs):

BSTs are specifically designed for fast searching of data. They maintain an ordered structure where elements in the left subtree are smaller and elements in the right subtree are larger than the root node. BSTs are used in databases, libraries, and compilers for efficient data retrieval and storage.

4. Expression Trees:

Expression trees are used to represent mathematical expressions in a way that facilitates evaluation. They are crucial in compilers and interpreters for parsing expressions, optimizing computations, and generating machine code.

5. Decision Trees (Machine Learning):

In machine learning, decision trees are used for classification and regression tasks. They help in partitioning data based on features to make decisions at each node, leading to interpretable models and insights into data patterns.

6. AVL Trees and Red-Black Trees:

AVL trees and Red-Black trees are balanced binary search trees that ensure logarithmic time complexity for search, insert, and delete operations. They are used in databases and other applications where guaranteed performance is critical.

7. XML/HTML Parsing:

Trees are used extensively in parsing XML and HTML documents. XML and HTML documents are inherently hierarchical, and parsing them into a tree structure allows for efficient traversal, querying, and manipulation of document contents.

8. Trie (Prefix Tree):

Tries are specialized trees used for storing and searching strings efficiently. They are particularly useful in applications such as autocomplete features in text editors, spell checkers, and IP routing tables.

9. Huffman Coding (Min-Heap):

Huffman coding uses a binary tree (often implemented using a min-heap) to achieve lossless data compression. It assigns variable-length codes to different characters, with shorter codes assigned to more frequent characters, thereby minimizing the overall size of encoded data.

10. Parse Trees (Syntax Trees):

Parse trees or syntax trees are used in compilers and interpreters to represent the syntactic structure of source code or programming languages. They facilitate syntax analysis, semantic analysis, optimization, and code generation processes.

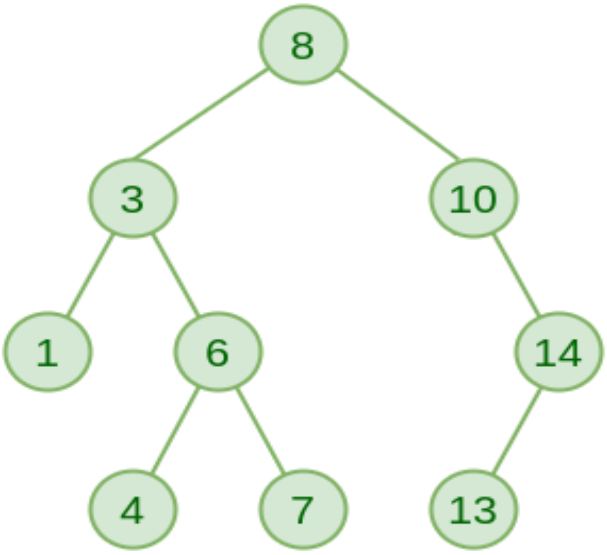
Let us sum up:

- **File Systems and Databases**: Trees are used to represent hierarchical file systems and database indexing (e.g., B-Trees, B+ Trees), providing efficient organization, navigation, and search operations.
- **Binary Search Trees (BSTs)**: BSTs allow fast searching, insertion, and deletion by maintaining an ordered structure, making them useful in databases, compilers, and libraries for data retrieval.

- **Expression and Decision Trees**: Expression trees represent and evaluate mathematical expressions, while decision trees in machine learning aid in classification and regression by making feature-based decisions.
- **Balanced Trees (AVL and Red-Black Trees)**: These trees maintain balance to ensure operations like search, insert, and delete run in logarithmic time, essential in performance-critical applications.
- **Specialized Trees (Trie, Huffman, Syntax Trees)**: Tries are used for efficient string operations like autocomplete, Huffman trees optimize data compression, and syntax trees aid in parsing and interpreting programming languages.

3.6 Binary Search Tree

A **Binary Search Tree** is a data structure used in computer science for organizing and storing data in a sorted manner. Each node in a **Binary Search Tree** has at most two children, a **left** child and a **right** child, with the **left** child containing values less than the parent node and the **right** child containing values greater than the parent node. This hierarchical structure allows for efficient **searching**, **insertion**, and **deletion** operations on the data stored in the tree.



Binary Search Tree

3.6.1 Operations in Binary Search Tree (BST)

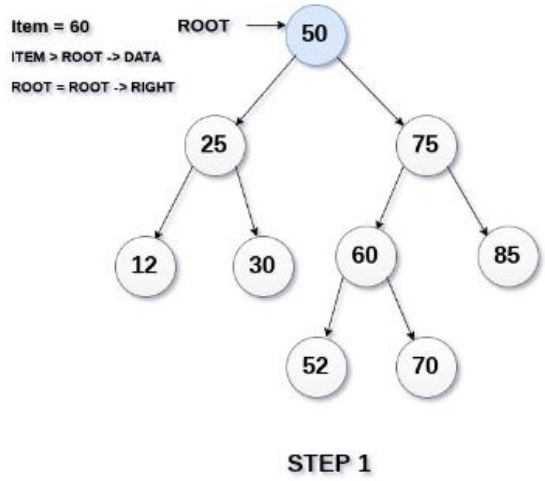
Binary Search Trees (BSTs) allow efficient data storage, retrieval, and management through a set of standard operations. Here, we will discuss the working methodology of the following BST operations in detail: search, insertion, deletion, and traversal.

3.6.1.1 Search Operation

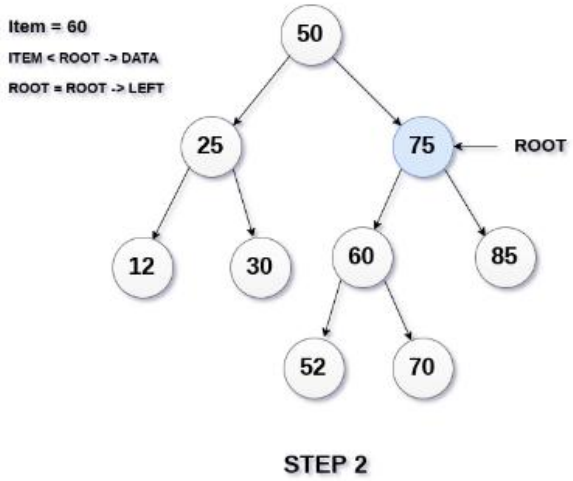
The search operation checks whether a specific value exists in the BST.

Working Methodology:

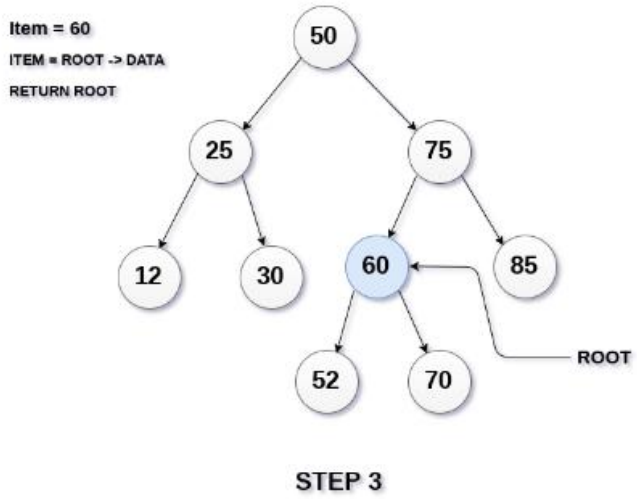
- Start at the root node.
- Compare the target value with the value of the current node:
  - If they are equal, the search is successful, and the node is found.
  - If the target value is less than the current node's value, move to the left child.
  - If the target value is greater than the current node's value, move to the right child.
- Repeat the process until the target value is found or the subtree becomes null (indicating the value is not present).



STEP 1



STEP 2



STEP 3

Python Implementation:

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


def search(root, key):
    if root is None or root.value == key:
        return root
    if key < root.value:
        return search(root.left, key)
    return search(root.right, key)
```

3.6.1.2. Insertion Operation

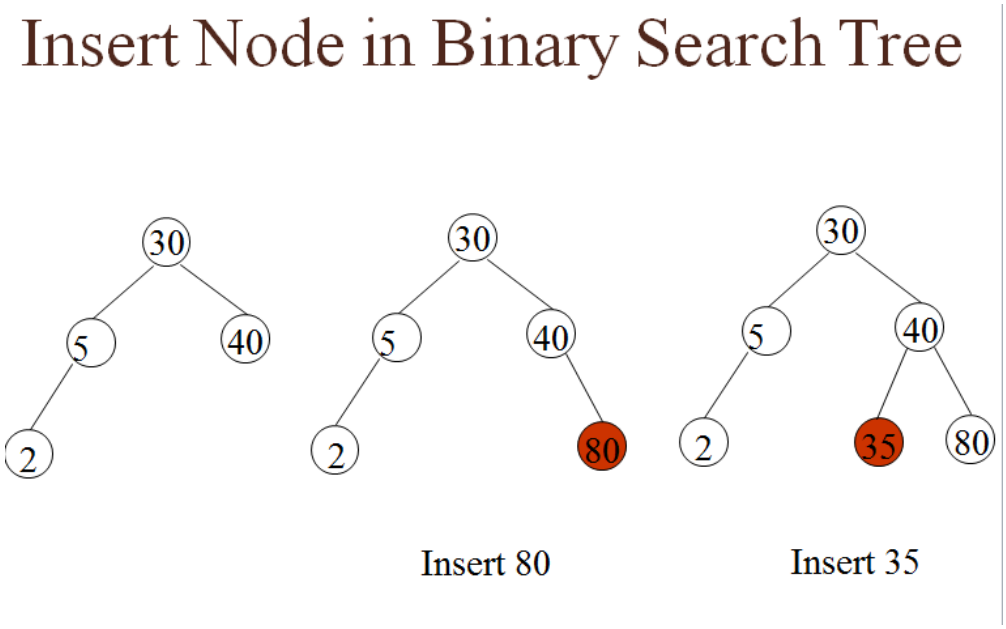
The insertion operation adds a new node with a specific value to the BST while maintaining the BST properties.

Working Methodology:

- Start at the root node.
- Compare the value to be inserted with the value of the current node:

- If the value to be inserted is less than the current node's value, move to the left child.
- If the value to be inserted is greater than the current node's value, move to the right child.

- When a null subtree is reached, insert the new node there.



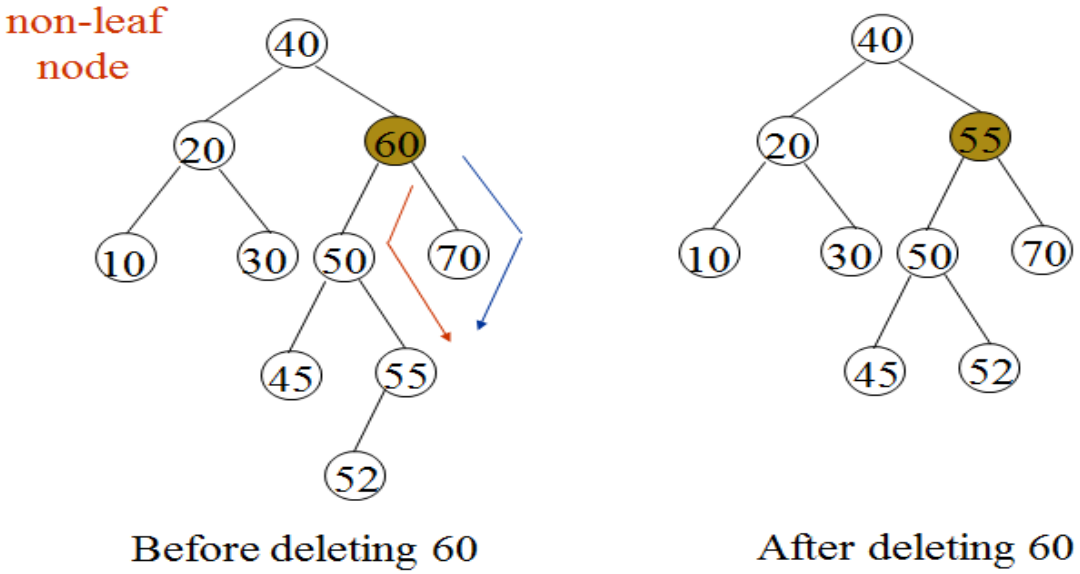
Python Implementation:

```
def insert(root, key):
    if root is None:
        return TreeNode(key)
    if key < root.value:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
    return root
```

3.6.1.3. Deletion Operation

The deletion operation removes a node with a specific value from the BST while maintaining the BST properties. There are three cases to consider:

1. **Node to be deleted has no children (leaf node):**
    - Simply remove the node.
2. **Node to be deleted has one child:**
    - Replace the node with its child.
3. **Node to be deleted has two children:**
    - Find the node's in-order successor (smallest value in the right subtree).
    - Replace the node's value with the in-order successor's value.
    - Delete the in-order successor.



Before deleting 60 After deleting 60

Working Methodology:

```
def minValueNode(node):
    current = node
    while current.left:
        current = current.left
    return current

def deleteNode(root, key):
```

```
if root is None:
    return root
if key < root.value:
    root.left = deleteNode(root.left, key)
elif key > root.value:
    root.right = deleteNode(root.right, key)
else:
    if root.left is None:
        return root.right
    elif root.right is None:
        return root.left
    temp = minValueNode(root.right)
    root.value = temp.value
    root.right = deleteNode(root.right, temp.value)
return root
```

3.6.1.4. Traversal Operations

Traversal operations visit all the nodes in the BST in a specific order. The main types of traversal are:

- **Inorder Traversal (Left, Root, Right)**

  Methodology: Visit the left subtree, then the root, and finally the right subtree.

```
def inorderTraversal(root):
    if root:
        inorderTraversal(root.left)
        print(root.value, end=' ')
        inorderTraversal(root.right)
```

- **Preorder Traversal (Root, Left, Right)**

  Methodology: Visit the root, then the left subtree, and finally the right subtree.

```
def preorderTraversal(root):
    if root:
        print(root.value, end=' ')
```

```
        preorderTraversal(root.left)
        preorderTraversal(root.right)
```

- **Postorder Traversal (Left, Right, Root)**

  Methodology: Visit the left subtree, then the right subtree, and finally the root.

```
def postorderTraversal(root):
    if root:
        postorderTraversal(root.left)
        postorderTraversal(root.right)
        print(root.value, end=' ')
```

- **Level-order Traversal (Breadth-First Search)**

  Methodology: Visit nodes level by level from left to right.

```
from collections import deque

def levelOrderTraversal(root):
    if not root:
        return
    queue = deque([root])
    while queue:
        node = queue.popleft()
        print(node.value, end=' ')
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
```

3.6.2 Example Usage

Combining these operations, here is an example usage demonstrating the BST operations:

```
# Constructing a binary search tree
root = None
```

```
keys = [10, 5, 20, 3, 7, 15, 25]

for key in keys:
    root = insert(root, key)

# Searching for a value
print("Searching for 15:", search(root, 15) is not None)  # Output: True

# Traversals
print("Inorder traversal:")
inorderTraversal(root)  # Output: 3 5 7 10 15 20 25
print("\nPreorder traversal:")
preorderTraversal(root)  # Output: 10 5 3 7 20 15 25
print("\nPostorder traversal:")
postorderTraversal(root)  # Output: 3 7 5 15 25 20 10
print("\nLevel-order traversal:")
levelOrderTraversal(root)  # Output: 10 5 20 3 7 15 25

# Deleting a value
root = deleteNode(root, 5)
print("\nInorder traversal after deleting 5:")
inorderTraversal(root)  # Output: 3 7 10 15 20 25
```

Let us sum up:

 Binary Search Tree (BST) Overview: A BST organizes data such that each node has at most two children. The left child contains values smaller than the parent, and the right child contains values larger, enabling efficient searching, insertion, and deletion.

 Key Operations in BST:

- **Search**: Navigate from root, comparing the target with node values, moving left or right as appropriate.
- **Insertion**: Compare the value to be inserted with the current node and place it in the correct position (left for smaller, right for larger).
- **Deletion**: Handle three cases—leaf node, node with one child, and node with two children (replacing with in-order successor).

 Traversal Methods:

- **Inorder (Left, Root, Right)**: Yields nodes in sorted order.
- **Preorder (Root, Left, Right)**: Processes root before its subtrees.
- **Postorder (Left, Right, Root)**: Processes subtrees before the root.
- **Level-order (Breadth-First)**: Visits nodes level by level, using a queue.

Check your progress:

1. In a Binary Search Tree (BST), which property must be true for every node?
   A. The left child must be greater than the node
   B. The right child must be smaller than the node
   C. The left child must be smaller than the node and the right child must be greater than the node
   D. All children must be equal to the node

   Answer: C. The left child must be smaller than the node and the right child must be greater than the node

2. What is the time complexity for searching an element in a balanced Binary Search Tree?
   A. O(1)
   B. O(log n)
   C. O(n)
   D. O(n^2)

   Answer: B. O(log n)

3. Which of the following operations is generally not efficiently supported by a Binary Search Tree?
   A. Insertion
   B. Deletion

C. Search
D. Finding the maximum element in constant time

Answer: D. Finding the maximum element in constant time

4. What traversal method of a Binary Search Tree would produce the nodes in ascending order?
   A. Preorder traversal
   B. Inorder traversal
   C. Postorder traversal
   D. Level-order traversal

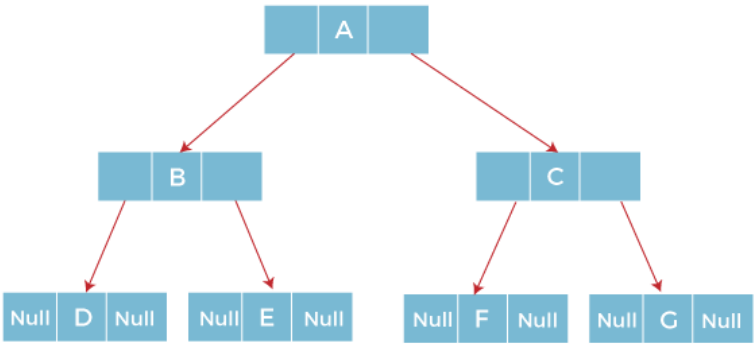
   Answer: B. Inorder traversal

5. What is the primary drawback of an unbalanced Binary Search Tree?
   A. It uses more memory than a balanced tree
   B. It cannot perform deletions
   C. It may degrade to O(n) time complexity for search operations
   D. It requires complex algorithms for insertion

   Answer: C. It may degrade to O(n) time complexity for search operations

3.7 Threaded Binary Tree

In the linked representation of binary trees, more than one half of the link fields contain NULL values which results in wastage of storage space. If a binary tree consists of **n** nodes then **n+1** link fields contain NULL values. So in order to effectively manage the space, a method was devised by Perlis and Thornton in which the NULL links are replaced with special links known as threads. Such binary trees with threads are known as **threaded binary trees**. Each node in a threaded binary tree either contains a link to its child node or thread to other nodes in the tree.



Threaded binary tree

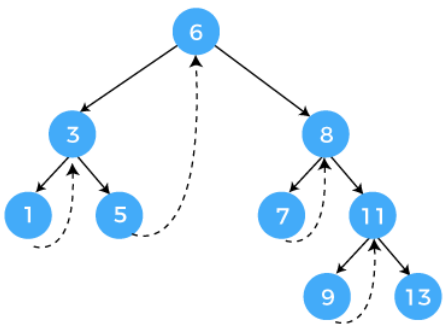
3.7.1 Types of Threaded Binary Tree

There are two types of threaded Binary Tree.

- One-way threaded Binary Tree
- Two-way threaded Binary Tree

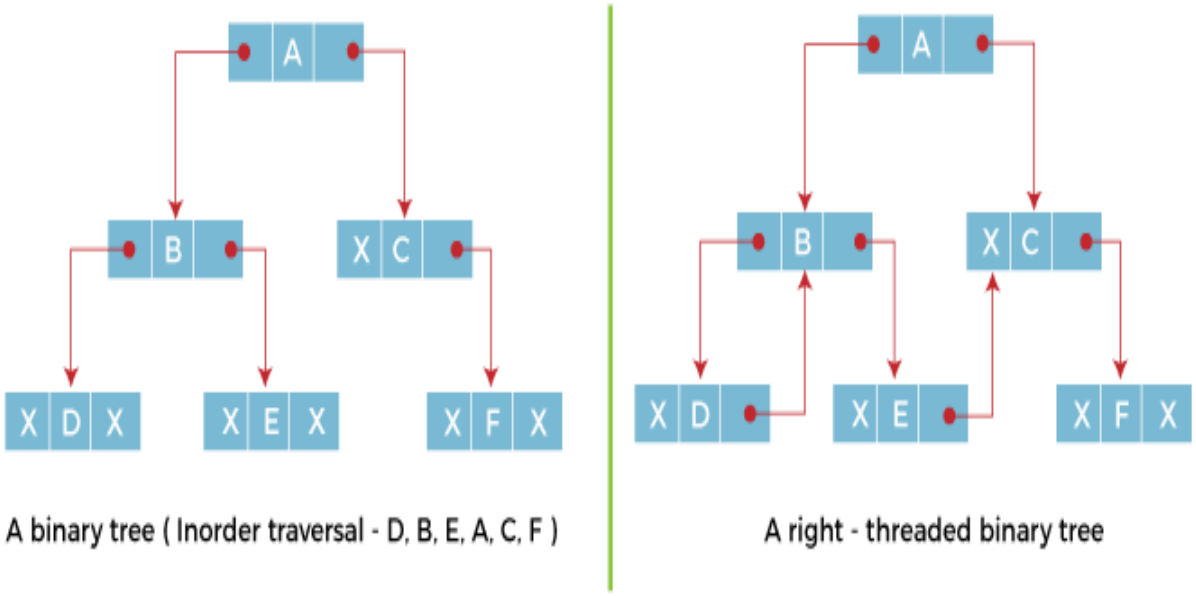
3.7.1.1.One-way threaded Binary trees:

In one-way threaded binary trees, a thread will appear either in the right or left link field of a node. If it appears in the right link field of a node then it will point to the next node that will appear on performing in order traversal. Such trees are called **Right threaded binary trees**. If thread appears in the left field of a node then it will point to the nodes inorder predecessor. Such trees are called **Left threaded binary trees.** Left threaded binary trees are used less often as they don't yield the last advantages of right threaded binary trees. In one-way threaded binary trees, the right link field of last node and left link field of first node contains a NULL. In order to distinguish threads from normal links they are represented by dotted lines.

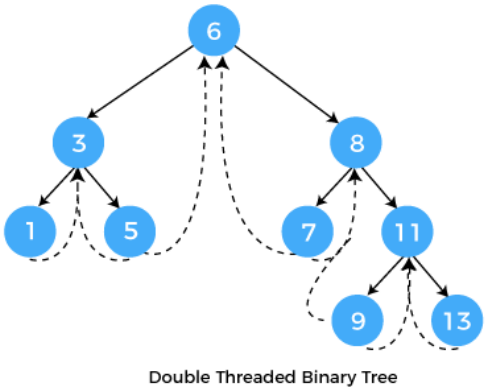


Single Threaded Binary Tree

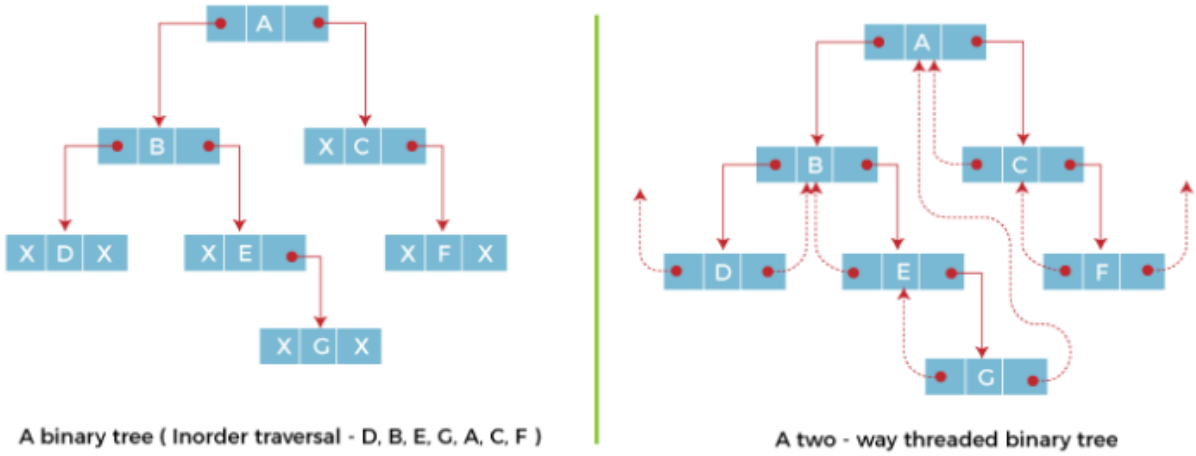
The below figure shows the inorder traversal of this binary tree yields D, B, E, A, C, F. When this tree is represented as a right threaded binary tree, the right link field of leaf node D which contains a NULL value is replaced with a thread that points to node B which is the inorder successor of a node D. In the same way other nodes containing values in the right link field will contain NULL value.



3.7.1.2.Two-way threaded Binary Trees:

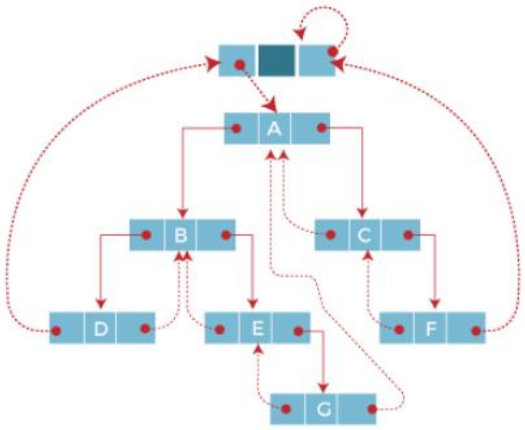


In two-way threaded Binary trees, the right link field of a node containing NULL values is replaced by a thread that points to nodes inorder successor and left field of a node containing NULL values is replaced by a thread that points to nodes inorder predecessor.



A binary tree (Inorder traversal - D, B, E, G, A, C, F) A two - way threaded binary tree

The above figure shows the inorder traversal of this binary tree yields D, B, E, G, A, C, F. If we consider the two-way threaded Binary tree, the node E whose left field contains NULL is replaced by a thread pointing to its inorder predecessor i.e. node B. Similarly, for node G whose right and left linked fields contain NULL values are replaced by threads such that right link field points to its inorder successor and left link field points to its inorder predecessor. In the same way, other nodes containing NULL values in their link fields are filled with threads.



Two-way threaded - tree with header node

In the above figure of two-way threaded Binary tree, we noticed that no left thread is possible for the first node and no right thread is possible for the last node. This is because they don't have any inorder predecessor and successor respectively. This is indicated by threads pointing nowhere. So in order to maintain the uniformity of threads, we maintain a special node called the **header node**. The header node does not contain any data part and its left link field points to the root node and its right link field points to itself. If this header node is included in the two-way threaded Binary tree then this node becomes the inorder predecessor of the first node and inorder successor of the last node. Now threads of left link fields of the first node and right link fields of the last node will point to the header node.

3.7.2 Advantages of Threaded Binary Tree:

- In threaded binary tree, linear and fast traversal of nodes in the tree so there is no requirement of stack. If the stack is used then it consumes a lot of memory and time.
- It is more general as one can efficiently determine the successor and predecessor of any node by simply following the thread and links. It almost behaves like a circular linked list.

3.7.3 Disadvantages of Threaded Binary Tree:

- When implemented, the threaded binary tree needs to maintain the extra information for each node to indicate whether the link field of each node points to an ordinary node or the node's successor and predecessor.
- Insertion into and deletion from a threaded binary tree are more time consuming since both threads and ordinary links need to be maintained.

Let us Sum up:

- Threaded Binary Trees: In threaded binary trees, NULL links in a binary tree are replaced by special links, called threads, which point to the inorder successor or predecessor, optimizing space and enabling faster traversals.
- Types of Threaded Binary Trees:

  - One-way Threaded: Threads exist only in the left or right link, with right threads pointing to the inorder successor or left threads pointing to the predecessor.
  - Two-way Threaded: Both left and right NULL links are replaced by threads pointing to inorder predecessor and successor, respectively.
  - Header Node: In two-way threaded trees, a special header node is introduced to maintain uniformity, pointing to the root and linking to both the inorder predecessor of the first node and the inorder successor of the last node.
- Advantages: Threaded binary trees enable fast, linear traversals without needing a stack, and provide easy access to successor and predecessor nodes, making traversal efficient.
- Disadvantages: Extra space is required for tracking whether a link is a thread or a regular link, and insertion/deletion operations are more complex as threads must be maintained alongside regular links.t us sum up:

Check your progress

1. What is the primary purpose of a threaded binary tree?
   A. To store data in a sorted manner
   B. To make in-order traversal more efficient
   C. To optimize memory usage in binary search trees
   D. To improve the height balance of the tree
   Answer: B. To make in-order traversal more efficient

2. In a threaded binary tree, what does a "thread" typically refer to?
   A. A link from a node to its left or right child
   B. A pointer to the previous or next node in the in-order traversal
   C. A reference to the parent node
   D. A link to the deepest node in the tree
   Answer: B. A pointer to the previous or next node in the in-order traversal

3. Which type of binary tree traversal is optimized by using threads in a threaded binary tree?

A. Preorder traversal

B. Postorder traversal

C. In-order traversal

D. Level-order traversal

Answer: C. In-order traversal

4. In a threaded binary tree, if a node's right child pointer is a thread, what does it point to?

   A. The left child of the node

   B. The root of the tree

   C. The successor node in in-order traversal

   D. The parent of the node

   Answer: C. The successor node in in-order traversal

5. How does a threaded binary tree differ from a regular binary tree in terms of traversal efficiency?

   A. It makes traversal more complex

   B. It reduces traversal efficiency

   C. It allows in-order traversal to be done without using a stack or recursion

   D. It eliminates the need for parent pointers

   Answer: C. It allows in-order traversal to be done without using a stack or recursion

3.8 AVL Trees: Operations and Methodologies

An AVL Tree is a self-balancing binary search tree where the difference in heights between the left and right subtrees of any node is at most one. Named after its inventors Adelson-Velsky and Landis, AVL trees maintain this balance through rotations, ensuring efficient operations even in the worst-case scenarios.

3.8.1 Key Properties of AVL Trees

(i) **Balance Factor**: For any node in the AVL tree, the balance factor (height of left subtree - height of right subtree) should be -1, 0, or 1.

(ii) **Height-Balanced**: The AVL tree maintains its balance by performing rotations during insertion and deletion to ensure the balance factor property.

3.8.2 Operations in AVL Trees

3.8.2.1. Insertion

Insertion in an AVL tree follows the same steps as in a binary search tree, followed by rotations to maintain balance.

Methodology:

1. Insert the node using the standard BST insertion method.
2. Update the height of the ancestor nodes.
3. Check the balance factor of each ancestor node.
4. Perform rotations to maintain the AVL property if the balance factor becomes unbalanced (-2 or +2).

Types of Rotations:

- **Right Rotation (LL Rotation)**: Applied when the left subtree is higher by 2.
- **Left Rotation (RR Rotation)**: Applied when the right subtree is higher by 2.
- **Left-Right Rotation (LR Rotation)**: Applied when the left subtree of the right child is higher.
- **Right-Left Rotation (RL Rotation)**: Applied when the right subtree of the left child is higher.

Diagram and Example:

Let's insert nodes 10, 20, 30 into an AVL tree to demonstrate rotations.

Initial tree (after inserting 10 and 20):

```
10
  \
  20
```

After inserting 30, the tree becomes:

```
10
  \
  20
    \
```

30

This tree is unbalanced (balance factor of root 10 is -2). Apply an RR Rotation:

```
  20
 /  \
10  30
```

Now the tree is balanced.

3.8.2.2. Deletion

Deletion in an AVL tree also follows the same steps as in a binary search tree, followed by rotations to maintain balance.

Methodology:

1. Perform standard BST deletion.
2. Update the height of the ancestor nodes.
3. Check the balance factor of each ancestor node.
4. Perform rotations to maintain the AVL property if the balance factor becomes unbalanced (-2 or +2).

Example and Diagram:

Let's delete node 10 from the following AVL tree:

```
  20
 /  \
10  30
```

After deleting 10:

```
 20
  \
  30
```

No rotations are needed as the tree is still balanced.

3.8.2.3. Traversal

Traversal operations in an AVL tree are the same as in a binary search tree, with the addition that each node maintains a height attribute. As per the previous section the

three types of tree traversal techniques inorder, preorder and post order traversal are possible in this tree.

Let us sum up:

- AVL Trees Overview: AVL trees are self-balancing binary search trees where the height difference (balance factor) between the left and right subtrees of any node is at most 1, ensuring efficient operations through balancing rotations.
- Balance Factor: The balance factor of each node is calculated as the height difference between its left and right subtrees. If this factor becomes -2 or +2, rotations are used to restore balance.
- Insertion: After inserting a node using the standard binary search tree method, rotations are performed to maintain balance. Four types of rotations are used: Right Rotation (LL), Left Rotation (RR), Left-Right Rotation (LR), and Right-Left Rotation (RL).
- Deletion: Like insertion, deletion follows standard BST operations, and rotations are applied afterward if the balance factor is disturbed.
- Efficiency: AVL trees guarantee that the tree remains balanced after every insertion and deletion, ensuring optimal time complexity for search, insertion, and deletion operations, making them ideal for applications with frequent modifications.

Check your progress

1. What is the primary characteristic of an AVL tree?

   A. All nodes have exactly two children

   B. It is a type of binary search tree that maintains balance by ensuring the heights of subtrees differ by at most one

   C. It allows duplicate keys

   D. It is a type of threaded binary tree

   Answer: B. It is a type of binary search tree that maintains balance by ensuring the heights of subtrees differ by at most one

2. What does an AVL tree balance factor of a node represent?
   A. The total number of nodes in the subtree
   B. The height of the left subtree minus the height of the right subtree
   C. The difference in value between the left and right child nodes
   D. The height of the node itself
   Answer: B. The height of the left subtree minus the height of the right subtree

3. Which rotation is used to fix an AVL tree when a left-left case imbalance occurs?
   A. Right Rotation
   B. Left Rotation
   C. Right-Left Rotation
   D. Left-Right Rotation
   Answer: A. Right Rotation

4. What is the time complexity for search, insert, and delete operations in an AVL tree?
   A. O(n)
   B. O(log n)
   C. O(n log n)
   D. O(1)
   Answer: B. O(log n)

5. When inserting a new node into an AVL tree, which of the following might be required to maintain balance?
   A. Replacing the root node
   B. Rotation of nodes (single or double rotations)
   C. Rebalancing only the subtree with the newly added node
   D. Removing the deepest node
   Answer: B. Rotation of nodes (single or double rotations)

3.9 B-Tree:

A **B-Tree** is a self-balancing tree data structure that maintains sorted data and allows for efficient insertion, deletion, and search operations. B-Trees are commonly used in databases and file systems.

3.9.1 Key Characteristics:

(i) **Order**: A B-Tree of order m can have a maximum of m−1keys and m children per node.

(ii) **Node Properties**:

a. Every node contains at most m−1 keys.

b. Every node except the root must have at least ⌈m/2⌉−1 keys.

c. The root node must have at least one key.

(iii) **Height-Balanced**: All leaves are at the same level, ensuring balanced height.

(iv) **Keys and Children**: Keys in each node are maintained in a sorted order. Internal nodes act as guides for searching and have children pointers.

3.9.2 Operations:

3.9.2.1 Search:

- Begin at the root.
- Traverse through nodes, comparing the target key with the keys in the node.
- Follow the appropriate child pointer until the target key is found or a leaf node is reached.

3.9.2.2 Insertion:

- Insertions are performed at the leaf nodes.
- If a leaf node overflows (i.e., exceeds m−1keys), it splits into two nodes:
  - The median key is moved up to the parent.
  - This split operation may propagate up to the root, potentially increasing the height of the tree.

3.9.2.3 Deletion:

- Deletion may require rebalancing the tree to ensure all nodes meet the minimum key requirement.
- If a node underflows (i.e., has fewer than $\lceil m/2 \rceil - 1$ keys):
  - Borrow a key from a sibling, or
  - Merge with a sibling and adjust the parent node accordingly,
- Similar to insertion, rebalancing may propagate up to the root.

Example:

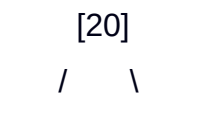
Consider a B-Tree of order 4, meaning each node can have up to 3 keys and 4 children. The tree structure evolves through insertions and deletions while maintaining balance and sorted order of keys.

Initial (Empty Tree):

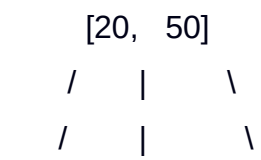
Insert keys: 10, 20, 30

[10, 20, 30]

Insert key: 40 (causes split)



Insert keys: 50, 60, 70 (causes another split)



3.9.3 Use Cases:

- **Databases**: B-Trees are used in databases to manage indices, allowing quick data retrieval and efficient use of storage.
- **File Systems**: File systems use B-Trees to manage file directories and metadata, ensuring fast access and modifications.

3.9.4 Advantages:

- **Balanced Tree Structure**: Ensures logarithmic height, resulting in efficient operations.
- **Efficient Disk Use**: Minimizes disk I/O operations by keeping nodes partially filled and balanced.
- **Scalable**: Handles a large number of insertions and deletions gracefully, maintaining performance.

Let us sum up:

 Definition and Purpose: A B-Tree is a self-balancing tree data structure used for maintaining sorted data and allowing efficient insertion, deletion, and search operations, commonly employed in databases and file systems.

 Key Characteristics:

- **Order**: A B-Tree of order mmm can have up to m−1m-1m−1 keys and mmm children per node.
- **Node Properties**: Each node holds at most m−1m-1m−1 keys; non-root nodes must have at least ⌈m/2⌉−1\lceil m/2 \rceil - 1⌈m/2⌉−1 keys; the root must have at least one key.
- **Height-Balanced**: All leaves are at the same level, ensuring a balanced tree height.

 Operations:

- **Search**: Traverse from the root, comparing keys and following child pointers until the target key is found or a leaf node is reached.
- **Insertion**: Insertions are made at leaf nodes. If a node overflows, it splits and propagates the median key up to the parent, which may increase the tree height.
- **Deletion**: Deletions may cause underflows in nodes, requiring rebalancing through borrowing or merging nodes, with changes potentially propagating up to the root.

 Advantages:

- **Balanced Structure**: Maintains logarithmic height for efficient operations.
- **Efficient Disk Use**: Reduces disk I/O by keeping nodes partially filled and balanced.
- **Scalability**: Handles large datasets and frequent updates efficiently, making it ideal for databases and file systems.

Check your progress

1. What is the maximum number of children a node in a B-Tree of order mmm can have?
   A. m−1m-1m−1 B. mmm C. m+1m+1m+1 D. 2m2m2m
   Answer: B. mmm

2. In a B-Tree, what happens when a node overflows during insertion?
   A. The node is removed from the tree
   B. The node splits into two, and the median key is moved up to the parent
   C. The tree is restructured
   D. The insertion is aborted
   Answer: B. The node splits into two, and the median key is moved up to the parent

3. How does a B-Tree ensure that all leaves are at the same level?
   A. By reorganizing nodes during deletions
   B. By rebalancing the tree during insertions and deletions
   C. By maintaining a fixed number of children per node
   D. By sorting keys in ascending order
   Answer:B. By rebalancing the tree during insertions and deletions

4. What is the purpose of maintaining a balanced height in a B-Tree?
   A. To ensure that all nodes have the same number of keys
   B. To optimize search, insertion, and deletion operations by maintaining logarithmic height
   C. To keep nodes completely filled
   D. To eliminate the need for balancing operations
   Answer: B. To optimize search, insertion, and deletion operations by maintaining logarithmic height

5. What action is required when a node underflows (i.e., has fewer than [m/2]−1\lceil m/2 \rceil - 1[m/2]−1 keys) during deletion in a B-Tree?
   A. Insert additional keys into the node
   B. Remove the node from the tree
   C. Borrow a key from a sibling or merge with a sibling
   D. Increase the height of the tree
   Answer: C. Borrow a key from a sibling or merge with a sibling

3.10 B+ Tree:

A **B+ Tree** is an extension of the B-Tree data structure, commonly used in databases and file systems to store large amounts of sorted data. It enhances B-Trees by providing efficient data retrieval through a linked list of leaf nodes.

3.10.1 Key Characteristics:

1. **Order**: A B+ Tree of order mmm (often denoted as Bm+B^+_mBm+) can have a maximum of m−1m-1m−1 keys per internal node and mmm children.

2. **Node Properties**:
    - **Internal Nodes**: Store keys to guide the search process and have mmm children pointers.
    - **Leaf Nodes**: Contain all the actual data records and have m−1m-1m−1 keys.
    - Every leaf node contains a pointer to the next leaf node, forming a linked list.
3. **Height-Balanced**: All leaf nodes are at the same level, ensuring balanced height.
4. **Separation of Index and Data**: Internal nodes only contain keys and pointers, while leaf nodes store the actual data.

3.10.2 Operations:

1. **Search**:
    - Begin at the root.
    - Traverse through internal nodes, comparing the target key with the keys in the node.
    - Follow the appropriate child pointer until a leaf node is reached.
    - Perform a linear search within the leaf node.
2. **Insertion**:
    - Insertions are performed at the leaf nodes.
    - If a leaf node overflows (i.e., exceeds m−1keys), it splits into two nodes:
        - The median key is propagated up to the parent.
        - This split operation may propagate up to the root, potentially increasing the height of the tree.
    - Update pointers to maintain the linked list of leaf nodes.
3. **Deletion**:
    - Deletions are performed at the leaf nodes.
    - If a leaf node underflows (i.e., has fewer than ⌈m/2⌉−1 keys):
        - Borrow a key from a sibling, or

- Merge with a sibling and adjust the parent node accordingly.
- Similar to insertion, rebalancing may propagate up to the root.
- Ensure the linked list of leaf nodes remains intact.

Example:

Consider a B+ Tree of order 4, meaning each node can have up to 3 keys and 4 children.

Initial (Empty Tree):

Insert keys: 10, 20, 30, 40, 50, 60

Internal: [30]

/ \

Leaf: [10, 20] [30, 40, 50, 60]

Insert key: 70 (causes split in leaf and internal node)

Internal: [30, 50]

/ | \

Leaf: [10, 20] [30, 40] [50, 60, 70]

3.10.3 Use Cases:

- **Databases**: B+ Trees are widely used in databases to manage indices, enabling fast range queries and efficient disk usage.
- **File Systems**: File systems utilize B+ Trees to manage directories and file metadata, allowing quick access and updates.

3.10.4 Advantages:

- **Efficient Range Queries**: The linked list of leaf nodes allows quick sequential access, making range queries efficient.
- **Balanced Tree Structure**: Ensures logarithmic height, resulting in efficient operations.
- **Separation of Index and Data**: Improves space utilization and speeds up search operations since internal nodes are smaller.

Let us sum up:

 Definition and Purpose: A B+ Tree is an extension of the B-Tree data structure that provides efficient data retrieval by linking leaf nodes in a sequential manner. It is commonly used in databases and file systems to manage large volumes of sorted data.

 Key Characteristics:

- **Order**: A B+ Tree of order mmm can have up to m−1m-1m−1 keys per internal node and mmm children.
- **Node Properties**:
  - **Internal Nodes**: Store keys and pointers to guide searches, with mmm children pointers.
  - **Leaf Nodes**: Store actual data records, with m−1m-1m−1 keys and pointers to the next leaf node, forming a linked list.
- **Height-Balanced**: All leaf nodes are at the same level, ensuring balanced height.
- **Separation of Index and Data**: Internal nodes contain only keys and pointers, while leaf nodes contain the actual data and links to other leaf nodes.

 Operations:

- **Search**: Traverse from the root through internal nodes, following child pointers, and perform a linear search within the appropriate leaf node.
- **Insertion**: Insertions are made at leaf nodes. If a leaf node overflows, it splits, propagating the median key up to the parent, and updates the leaf node linked list.
- **Deletion**: Perform deletions at leaf nodes. If a node underflows, it may borrow a key from or merge with a sibling, and rebalancing may affect the root. The linked list of leaf nodes must remain intact.

Check your progress

1. What distinguishes a B+ Tree from a B-Tree?
   A. Internal nodes do not contain keys; only leaf nodes do
   B. All keys are stored in internal nodes, while leaf nodes only store pointers
   C. Internal nodes store keys and pointers to child nodes, while leaf nodes store keys and pointers to data or to other leaf nodes
   D. The tree is unbalanced

Answer: C. Internal nodes store keys and pointers to child nodes, while leaf nodes store keys and pointers to data or to other leaf nodes

2. In a B+ Tree, what is the primary purpose of having all data stored in the leaf nodes?
   A. To ensure that data retrieval operations are more efficient
   B. To reduce the height of the tree
   C. To facilitate quicker key insertions and deletions
   D. To make traversal of the tree faster
   Answer: A. To ensure that data retrieval operations are more efficient

3. How are leaf nodes typically organized in a B+ Tree?
   A. They are linked together in a linked list to allow sequential access
   B. They are randomly distributed across the tree
   C. They store a fixed number of keys and pointers to data
   D. They are arranged in a binary search pattern
   Answer: A. They are linked together in a linked list to allow sequential access

4. What is a key advantage of using a B+ Tree over a B-Tree for database indexing?
   A. B+ Trees have a smaller number of nodes
   B. B+ Trees allow for faster searching, insertion, and deletion operations
   C. B+ Trees provide better support for range queries due to their sequentially linked leaf nodes
   D. B+ Trees require less memory

Answer: C. B+ Trees provide better support for range queries due to their sequentially linked leaf nodes

5. What is the maximum number of children a node in a B+ Tree of order mmm can have?

   A. m B. m−1 C. m+1 D. 2m

   Answer: A. m

3.11 Heap Tree:

A **Heap Tree** is a special binary tree-based data structure that satisfies the heap property. Heaps are commonly used to implement priority queues and for efficient sorting algorithms like heapsort.

3.11.1 Key Characteristics:

1. **Binary Heap**: A complete binary tree where all levels are fully filled except possibly the last, which is filled from left to right.
2. **Heap Property**: Two types of heap properties define the structure:
   - **Max-Heap**: For any given node iii, the value of iii is greater than or equal to the values of its children.
   - **Min-Heap**: For any given node iii, the value of iii is less than or equal to the values of its children.

3.11.2 Operations:

1. **Insertion**:
   - Insert the new element at the end of the tree (the next available position).
   - **Heapify Up**: Compare the inserted node with its parent; if the heap property is violated, swap them. Continue this process until the heap property is restored.
2. **Deletion (Removing the Root)**:
   - Swap the root with the last element.

- Remove the last element (the original root).
- **Heapify Down**: Compare the new root with its children; if the heap property is violated, swap it with the larger child (in a max-heap) or the smaller child (in a min-heap). Continue this process until the heap property is restored.

3. **Peek (Find Min/Max)**:
   - Return the root element of the heap without removing it. In a max-heap, this is the maximum element, and in a min-heap, it is the minimum element.
4. **Heapify**:
   - Convert an arbitrary array into a heap.
   - Start from the last non-leaf node and apply the heapify process iteratively to all nodes up to the root.

Example:

Consider a max-heap example:

Initial Array: [10, 20, 15, 30, 40]

Building the Max-Heap:

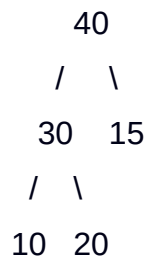
1. Start from the last non-leaf node (index 1) and heapify.

Array: [10, 40, 15, 30, 20]

2. Move to the root (index 0) and heapify.

Array: [40, 30, 15, 10, 20]

Max-Heap:



3.11.3 Use Cases:

- **Priority Queues**: Heaps are used to implement priority queues where the highest (or lowest) priority element is accessed first.
- **Heapsort**: An efficient comparison-based sorting algorithm that uses a heap data structure.
- **Graph Algorithms**: Used in algorithms like Dijkstra's shortest path and Prim's minimum spanning tree.

3.11.4 Advantages:

- **Efficient Operations**: Insertion, deletion, and access to the minimum or maximum element are efficient, with logarithmic time complexity O(log n).
- **Space-Efficient**: Heaps can be implemented as arrays, making them space-efficient.

3.11.4 Applications of Heap

Heaps are versatile data structures used in various computational tasks due to their efficient insertion, deletion, and retrieval operations. Here are some key applications of heaps:

1. Priority Queues

- **Implementation**: Heaps are used to implement priority queues, where each element has a priority. The highest (in a max-heap) or lowest (in a min-heap) priority element is accessed first.
- **Use Cases**: Task scheduling in operating systems, managing job queues in printers, and handling real-time event scheduling.

2. Heapsort

- **Algorithm**: Heapsort is an efficient, comparison-based sorting algorithm that uses a heap to sort elements.
- **Process**: The algorithm involves building a max-heap from the input data and then repeatedly extracting the maximum element to build the sorted array.

- **Advantages**: Heapsort has a time complexity of O(nlog⁡n)O(n \log n)O(nlogn) and is in-place, meaning it requires only a small, constant amount of additional memory space.

3. Graph Algorithms

- **Dijkstra's Algorithm**: Used for finding the shortest path in a graph. A min-heap is used to efficiently retrieve the vertex with the minimum distance.
- **Prim's Algorithm**: Used for finding the minimum spanning tree of a graph. A min-heap helps in selecting the edge with the minimum weight.

4. Median Maintenance

- **Dynamic Median**: Heaps can be used to dynamically maintain the median of a stream of numbers by using two heaps:
  - A max-heap to store the smaller half of the numbers.
  - A min-heap to store the larger half.
- **Efficiency**: This allows the median to be retrieved efficiently in O(1)O(1)O(1) time, and insertion operations can be handled in O(log⁡n)O(\log n)O(logn) time.

5. Order Statistics

- **Kth Largest/Smallest Element**: Heaps can be used to find the kkkth largest or smallest element in an unsorted array efficiently.
  - For the kkkth smallest element, a min-heap of size kkk can be maintained.
  - For the kkkth largest element, a max-heap of size kkk is used.

6. Event Simulation

- **Event Scheduling**: Heaps are used in discrete event simulation systems to manage and schedule events. The event with the nearest upcoming time is processed first, efficiently managed by a min-heap.

7. Merging Sorted Lists

- **K-Way Merge**: Heaps are used to merge multiple sorted lists into a single sorted list. A min-heap can efficiently track the smallest element among the heads of the lists, facilitating the merge process.

8. Interval Management

- **Interval Heaps**: Used in applications where intervals need to be managed, such as scheduling and resource allocation. Interval heaps help in efficiently managing overlapping intervals.

Let us sum up:

 Definition: A Heap Tree is a complete binary tree that satisfies the heap property, used for priority queues and sorting algorithms like heapsort.

 Key Characteristics:

- **Binary Heap**: A complete binary tree with all levels fully filled except possibly the last, which is filled from left to right.
- **Heap Property**:
  - **Max-Heap**: Each node's value is greater than or equal to its children's values.
  - **Min-Heap**: Each node's value is less than or equal to its children's values.

 Operations:

- **Insertion**: Add an element at the end of the tree and "Heapify Up" to restore the heap property.
- **Deletion**: Remove the root by swapping it with the last element, then "Heapify Down" to restore the heap property.
- **Peek**: Return the root element (max in a max-heap, min in a min-heap) without removing it.
- **Heapify**: Convert an array into a heap by applying heapify from the last non-leaf node up to the root.

 Use Cases:

- **Priority Queues**: Manage elements with varying priorities efficiently.
- **Heapsort**: An efficient sorting algorithm based on heap data structure.
- **Graph Algorithms**: Utilized in algorithms like Dijkstra's shortest path and Prim's minimum spanning tree.

 Advantages:

- **Efficient Operations**: Insertions, deletions, and access to min/max elements are performed in O(log⁡n)O(\log n)O(logn) time.
- **Space-Efficient**: Can be implemented using arrays, optimizing space utilization.

Check your progress

1. What is the primary characteristic of a **min-heap**?
   A. The value of each node is greater than or equal to the values of its children
   B. The value of each node is less than or equal to the values of its children
   C. The tree is balanced but not necessarily complete
   D. The root node contains the maximum value
   Answer: B. The value of each node is less than or equal to the values of its children

2. What is the time complexity of **inserting** a new element into a binary heap?
   A. O(1) B. O(log n) C. O(n) D. O(n log n)
   Answer:B. O(log n)

3. Which of the following operations is used to maintain the heap property after removing the root from a heap?
   A. Heapify B. Sort C. Balance D. Merge
   Answer:A. Heapify

4. In a binary heap, which property ensures that the tree remains complete?
   A. The heap property B. The completeness property
   C. The binary tree property D. The structure property

Answer:D. The structure property

5. What is the time complexity of **finding** the maximum (in a max-heap) or minimum (in a min-heap) element?

   A. O(1) B. O(log n) C. O(n) D. O(n log n)

   Answer: A. O(1)

Activities:

1) Write a program to traverse a tree using pre-order, in-order, and post-order traversal methods.
2) Write a program to evaluate an expression tree.
3) Implement a binary search tree and perform various operations such as insert, delete, and search and visualize the tree after each operation to understand the changes.

Points to Remember

- A tree is a hierarchical data structure with nodes connected by edges.
- The top node is called the root, and nodes without children are leaves.
- Each node contains a value and references to its children.
- **Pre-order:** Visit the root, traverse the left subtree, then the right subtree.
- **In-order:** Traverse the left subtree, visit the root, then the right subtree.
- **Post-order:** Traverse the left subtree, the right subtree, then visit the root.
- A binary tree is a tree where each node has at most two children.
- The two children are referred to as the left child and the right child.
- An expression tree is a binary tree used to represent arithmetic expressions.
- Each leaf node represents an operand, and each internal node represents an operator.
- A binary search tree (BST) is a binary tree where each node follows the property: left child < parent < right child.
- BSTs provide efficient search, insertion, and deletion operations.
- A threaded binary tree uses null pointers to store pointers to in-order predecessor or successor which allows for efficient in-order traversal without using a stack or recursion.

- B-trees are balanced search trees designed for systems that read and write large blocks of data, They maintain balance by splitting and merging nodes.
- A heap is a complete binary tree where each node is greater (max heap) or smaller (min heap) than its children.
- Heaps are used to implement priority queues.
- B+ trees are an extension of B-trees with all values at the leaf level.
- Internal nodes only store keys, which makes range queries efficient.
- AVL trees are self-balancing binary search trees, After every insertion or deletion, the tree is balanced using rotations.

Questions

1. What is a tree data structure?
2. Explain the difference between a tree and a binary tree.
3. How do you calculate the height of a tree?
4. Describe the three main types of tree traversals.
5. Which traversal would you use to get a sorted list from a binary search tree?
6. How does in-order traversal of a binary search tree work?
7. What is a binary tree?
8. How is a binary tree different from a general tree?
9. What are the maximum and minimum heights of a binary tree with n nodes?
10. What is an expression tree?
11. How do you construct an expression tree from an infix expression?
12. How is an expression tree evaluated?
13. What are some real-world applications of trees?
14. How are trees used in file systems?
15. Why are trees suitable for representing hierarchical data?
16. What is a binary search tree?
17. How do you insert an element in a BST?
18. What are the differences between single and double threading?
19. What is an AVL tree?
20. How does an AVL tree maintain its balance?

21. What are the different types of rotations used in AVL trees?
22. What is a B-tree?
23. How does a B-tree differ from a binary search tree?
24. What are the advantages of using a B-tree?
25. What is a B+ tree?
26. How do B+ trees improve upon B-trees?
27. What are the benefits of having all values at the leaf level?
28. What is a heap?
29. How do you insert an element into a heap?

Glossary

- **Node:** A fundamental part of a tree that contains data.
- **Edge:** The connection between two nodes.
- **Root:** The topmost node of a tree.
- **Leaf:** A node with no children.
- **Traversal:** The process of visiting all nodes in a tree in a specific order.
- **Pre-order:** Root-Left-Right traversal.
- **In-order:** Left-Root-Right traversal.
- **Post-order:** Left-Right-Root traversal.
- **Binary Tree:** A tree in which each node has at most two children.
- **Left Child:** The left node in a binary tree.
- **Right Child:** The right node in a binary tree.
- **Expression Tree:** A binary tree representing an arithmetic expression.
- **Operand:** A value on which an operation is performed.
- **Operator:** A symbol denoting a mathematical operation.
- **File System:** A method of storing and organizing files and their data.
- **Database Indexing:** Using trees to speed up data retrieval operations.
- **Binary Search Tree (BST):** A binary tree with an ordered property.
- **Threaded Binary Tree:** A binary tree where null pointers are replaced by pointers to the in-order predecessor or successor.
- **Single Threading:** Only the right (or left) null pointers are used for threading.

- **Double Threading:** Both left and right null pointers are used for threading.
- **AVL Tree:** A self-balancing binary search tree.
- **Rotation:** An operation to maintain tree balance after insertion or deletion.
- **B-Tree:** A balanced tree data structure optimized for systems that read and write large blocks of data.
- **Node Splitting:** The process of dividing a full node into two nodes.
- **B+ Tree:** A balanced tree with all values stored at the leaf level.
- **Leaf Node:** The nodes at the lowest level of a tree.
- **Heap:** A specialized tree-based data structure.
- **Max Heap:** A heap where the parent node is always greater than the children.
- **Min Heap:** A heap where the parent node is always smaller than the children.
- **Priority Queue:** A data structure where each element has a priority.
- **Heap Sort:** A comparison-based sorting algorithm using a heap.

Further Reading and References

Books

1. **"Introduction to Algorithms"** by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
    - Comprehensive coverage of various tree structures including binary trees, AVL trees, and B-trees.
2. **"Data Structures and Algorithm Analysis in C++"** by Mark Allen Weiss
    - Detailed explanations of different tree types and their implementations in C++.
3. **"Algorithms"** by Robert Sedgewick and Kevin Wayne
    - Covers fundamental tree structures and algorithms, including balanced trees and binary search trees.
4. **"Data Structures and Algorithmic Thinking"** by Narasimha Karumanchi
    - Provides a practical approach to tree data structures with a focus on problem-solving.
5. **"The Algorithm Design Manual"** by Steven S. Skiena

- Includes practical guidance and examples on various tree algorithms and data structures.

Online Resources

1. **GeeksforGeeks - Trees**
   - GeeksforGeeks Trees
   - Extensive articles and tutorials on different types of trees and their operations.
2. **Visualgo - Tree Visualizations**
   - Visualgo Trees
   - Interactive visualizations for understanding tree structures and algorithms.
3. **TutorialsPoint - Data Structures and Algorithms**
   - TutorialsPoint Trees
   - Guides and explanations on different tree data structures and their operations.
4. **Khan Academy - Data Structures**
   - Khan Academy Data Structures
   - Introductory videos and explanations on basic tree structures and algorithms.
5. **Stack Overflow - Tree Data Structures**
   - Stack Overflow Tree Discussions
   - Community-driven discussions and answers on various tree data structure problems and solutions.

Video Resources

1. **MIT OpenCourseWare - Introduction to Algorithms**
   - MIT OpenCourseWare Algorithms
   - Lecture videos that cover various tree structures and their algorithms.
2. **Coursera - Data Structures and Algorithm Specialization**
   - Coursera Data Structures

- Online courses that include videos and assignments on tree data structures.

3. **YouTube - mycodeschool**
   - mycodeschool Trees Playlist
   - Tutorials and visual explanations on various tree types and operations.
4. **YouTube - HackerRank**
   - HackerRank Trees Playlist
   - A collection of videos explaining tree data structures and algorithms.
5. **YouTube - CS50**
   - CS50 Introduction to Trees
   - Harvard's CS50 course includes tree data structures as part of their computer science curriculum.

UNIT 4: GRAPHS

Definition- Representation of Graph- Types of graph-Breadth first traversal – Depth first traversal-Topological sort- Bi-connectivity – Cut vertex- Euler circuits-Applications of graphs.

S.No	Topic	Page No
	Objectives	
4.1	**GRAPHS**	
4.1.1	Terminologies	
4.1.2	Different Types of Graphs in Data Structures	
4.2	**REPRESENTATION OF GRAPHS**	
4.2.1	Set Representation	
4.2.2	Linked Representation	
4.2.3	Matrix Representation	
4.3	**GRAPH TRAVERSAL**	
4.3.1	Depth First Search (DFS)	
4.3.2	Breadth First Search (BFS)	
4.4	**Biconnectivity of Graphs**	
4.4.1	Definition	
4.4.2	Articulation Points	
4.4.3	Biconnected Components	
4.2	**Euler Circuits in Data Structures**	
4.5.1	Definition	
4.5.2	Eulerian Graph	
4.5.3	Euler Path vs. Euler Circuitx.	
4.5.4	Algorithm	
4.5.5	Applications	
4.6	**Application of Graph Structures**	
4.6.1	Shortest Path Problem	
	Floyd-Warshall Algorithm	
	Dijkstra's Algorithm	

4.6.2	Topological Sorting	
4.6.3	Minimum Spanning Tree	
	Kruskal's Algorithm	
	Prim's Algorithm	
	Summary	
	Activities	
	Points to Remember	
	Questions	
	Glossary	
	Further Reading and References	

Objectives:

- To understand the basic concept of a graph, types and different methods of representing graphs.
- To understand the searching, sorting algorithm, used in the graphs
- To understand the concept of bi-connectivity in graphs,
- To learn about cut vertices (or articulation points),
- To understand Euler circuits and to examine the wide range of real-world applications of graphs.

4.1 GRAPHS

- Graph is another non-linear data structure.
- It is hierarchical relationship between parent and children's.
- A graph G consist of two sets a set of all vertices (V)(or nodes) and set of all edges(E) (or arcs) Ex: G= { V,E }

  For example, in G1

  V = {v1, v2, v3, v4}

  E = {(v1, v2), (v1, v3), (v1, v4), (v2, v3), (v3, v4)}

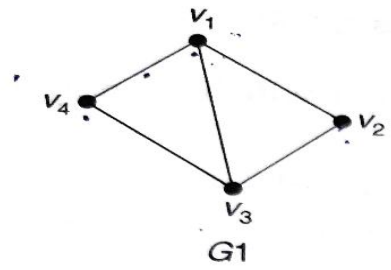
4.1.1 Terminologies

Digraph:

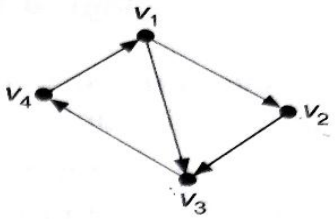
- A digraph is also called a directed graph. If a graph G, such that G=<V,E>, where V is the set of all vertices and E is the set of ordered pair of elements from V.
- Here G2 is a Digraph where

  V = {v1, v2, v3, v4}

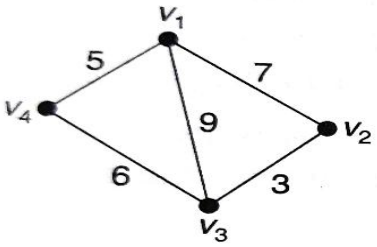
  E = {(v1, v2), (v1, v3), (v2, v3), (v3, v4), (v4, v1)}



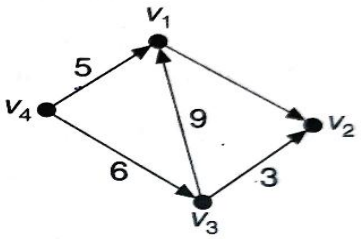
G1



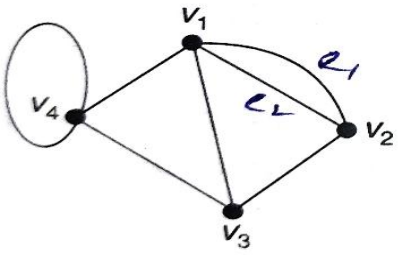
G2



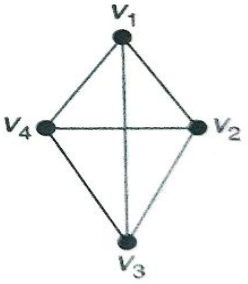
G3



G4

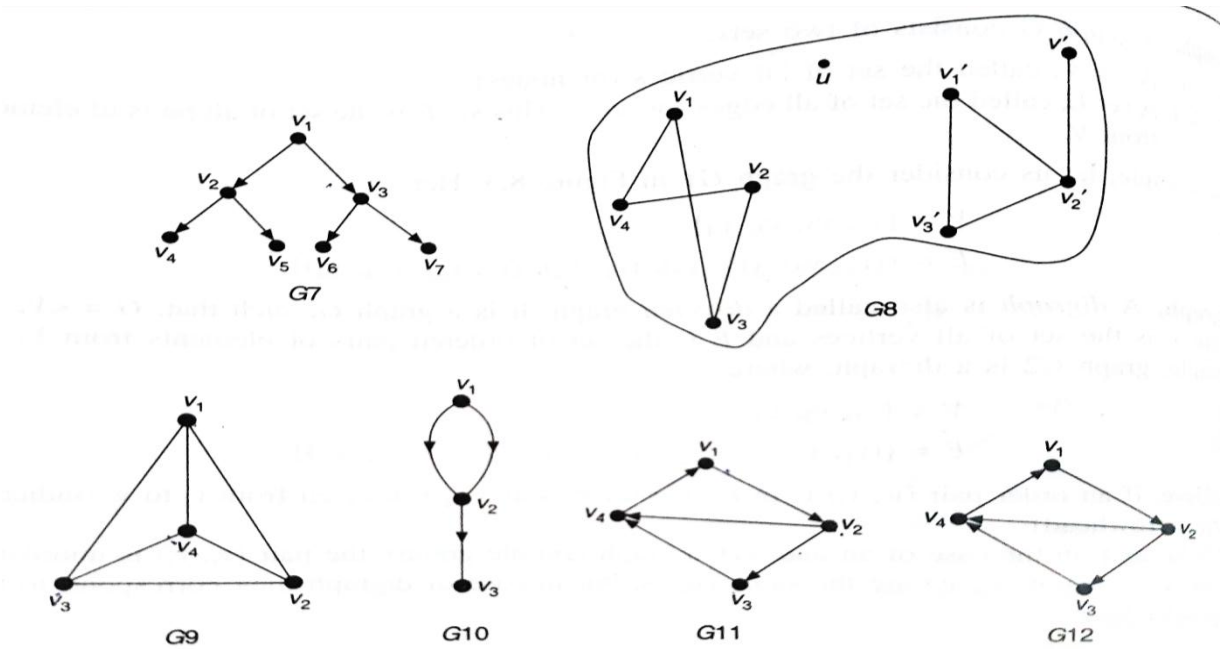


G5



G6

-



Weighted graph:

- A graph is termed as weighted graph if all the edges in it are labeled with some weight.
- Ex: G3 and G4 are two weighted graphs.

Adjacent vertices:

- A vertex v_i is adjacent to another vertex say v_j if there is an edge from v_i to v_j.
- Ex: Graph G11, v2 is adjacent to v3 and v4.

Self loop:

- If there is an edge whose starting and end vertices are same, that is (vi,vj) is an edge then it is calleda self loop.
- Ex: GraphG5

Parallel edges:

- If there are more than one edges between the same pair of vertices,

then they are known as theparallel edge.

- Ex: Graph G5.

Isolated vertex:

- A vertex is isolated if there is no edge connected from any other vertex to the vertex.
- Ex: Graph G8.

Degree of vertex:

- The number of edges connected with vertex v_i called the degree of vertex v_i and is denoted by degree(v_i).
- In digraph there are two degrees: indegree and ourdegree.
- **Indegree** of v_i denoted as indegree(v_i) = number of edges incident into v_i.
- **Outdegree**(vi) = number of edges emanating from vi.
- Ex: In graph G4 indegree(v_1)=2, outdegree(v_1)=1indegree(v_2)=2

Pendent vertex:

- A vertex v_i is pendent if its indegree(v_i)=1 and outdegree (v_i)=0
- Ex: G8 is a pendent vertex.

Connected graph:

- In a graph G two vertices v_I and v_j are said to be connected if there is a path in G from v_i to v_j.
- A graph is said to be connected if for every pair of distinct vertices v_i, v_j in G there is a path.
- Ex: Graph G1, G3 and G6.

Check your progress:

1. What is a graph in the context of data structures?

   A. A collection of nodes and edges where each edge connects two nodes

   B. A collection of nodes only

C. A linear data structure where elements are accessed in a sequence

D. A hierarchical data structure with only parent-child relationships

Answer: A. A collection of nodes and edges where each edge connects two nodes

2. What does an edge in a graph represent?

   A. A node's value B. A connection between two nodes

   C. A node's parent D. The weight of a node

   Answer: B. A connection between two nodes

3. In a **directed graph**, what is the term used to describe an edge with a direction?

   A. Undirected edge B. Directed edge

   C. Bidirectional edge D. Weighted edge

   Answer: B. Directed edge

4. What is the difference between a **directed graph** and an **undirected graph**?

   A. In a directed graph, edges have no direction; in an undirected graph, edges do have direction.

   B. In an undirected graph, edges have no direction; in a directed graph, edges have direction.

   C. Directed graphs do not have weights; undirected graphs do.

   D. Directed graphs can only have one edge between nodes; undirected graphs can have multiple.

   Answer: B. In an undirected graph, edges have no direction; in a directed graph, edges have direction.

5. What is a **weighted graph**?

   A. A graph where edges have weights or costs associated with them

   B. A graph where all nodes have equal value

   C. A graph where edges do not have weights

   D. A graph where weights are assigned to nodes only

   Answer: A. A graph where edges have weights or costs associated with them

6. Which of the following terms describes a graph with no cycles?
   A. Cyclic graph B. Acyclic graph
   C. Connected graph D. Complete graph
   Answer: B. Acyclic graph

7. What is a **tree** in graph theory?
   A. A cyclic graph with a single root
   B. A connected acyclic graph
   C. A disconnected graph
   D. A graph with all nodes connected by multiple edges
   Answer: B. A connected acyclic graph

8. Which type of graph allows multiple edges between the same pair of nodes?
   A. Simple graph B. Multigraph C. Complete graph D. Directed acyclic graph
   Answer: B. Multigraph
9. What is a **subgraph**?
   A. A graph that contains all the edges and nodes of the original graph
   B. A graph that is part of a larger graph and contains a subset of the original graph's nodes and edges
   C. A graph that is disconnected from the original graph
   D. A graph that has no edges
   Answer: B. A graph that is part of a larger graph and contains a subset of the original graph's nodes and edges

4.1.2 Different Types of Graphs in Data Structures

1. Undirected Graph

- **Definition**: A graph where edges have no direction. If there is an edge between vertices u and v, it can be traversed both ways.
- **Example**: G1, G3, G5,G6,G9 Social networks where friendship is mutual.

2. Directed Graph (Digraph)

- **Definition**: A graph where edges have a direction. Each edge is an ordered pair (u,v)(u, v)(u,v) meaning it goes from u to v.
- **Example**: G2, G10,G11, Twitter following relationships where followings are not necessarily mutual.

3. Weighted Graph

- **Definition**: A graph where each edge has a weight (or cost) associated with it.
- **Example**: G3,G4, Road networks where edges represent roads and weights represent distances or travel times.

4. Unweighted Graph

- **Definition**: A graph where edges do not have weights.
- **Example**: G1,G2, Simple social networks where edges represent friendships.

5. Simple Graph

- **Definition**: A graph with no loops (edges connecting a vertex to itself) and no more than one edge between any pair of vertices.
- **Example**: Except G5 and G10, Basic networks like friend relationships without any self-connections.

6. Multigraph

- **Definition**: A graph that may have multiple edges (parallel edges) between the same pair of vertices.
- **Example**: G8, Different modes of transportation between two cities (flights, trains, etc.).

7. Complete Graph

- **Definition**: A graph where there is a unique edge between every pair of vertices.

- **Example**: G6 and G9, A network where every user is directly connected to every other user.

8. Connected Graph

- **Definition**: A graph where there is a path between any pair of vertices.
- **Example**: G1,G3 and G6 are connected graph but not G8. A computer network where each computer can communicate with any other computer.

9. Disconnected Graph

- **Definition**: A graph where some pairs of vertices do not have a path between them.
- **Example**: G8, Islands in an archipelago without bridges or boats connecting them.

10. Cyclic Graph

- **Definition**: A graph that contains at least one cycle (a path of edges and vertices wherein a vertex is reachable from itself).
- **Example**: G5, Network of one-way streets forming a loop.

11. Acyclic Graph

- **Definition**: A graph with no cycles.
- **Example**: Tree data structures and dependency graphs.

12. Tree

- **Definition**: A connected acyclic graph.
- **Example**: Hierarchical organizational charts.

13. Forest

- **Definition**: A collection of disjoint trees.
- **Example**: Multiple disconnected hierarchies or organizational structures.

4.2 REPRESENTATION OF GRAPHS

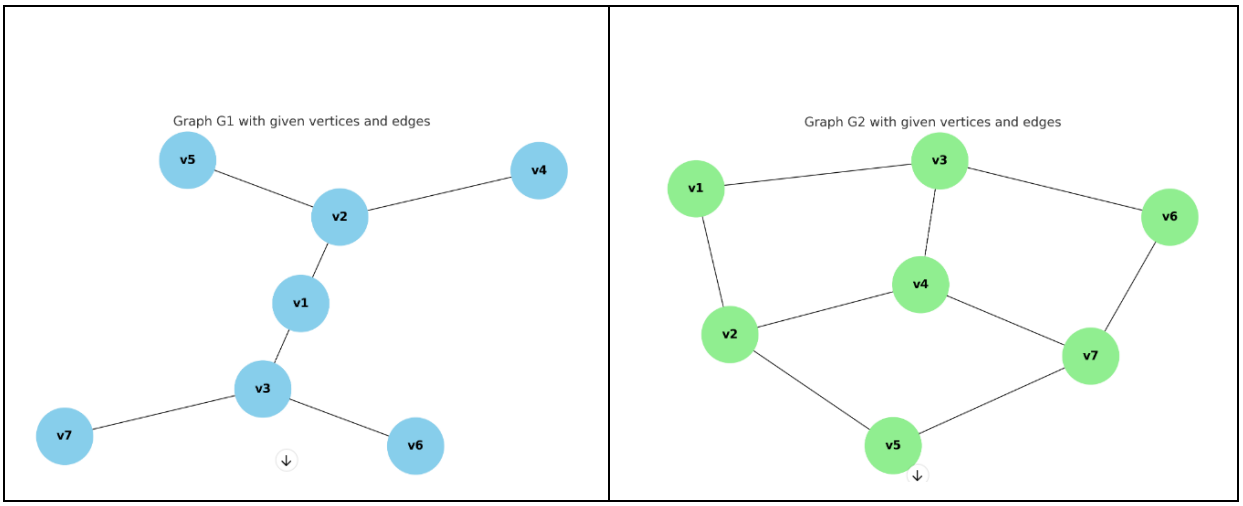
- A graph can be represented in many ways
  1. Set representation
  2. Linked representation
  3. Sequential (matrix) representation

4.2.1 Set representation:

This is one of the straightforward methods of representing a graph. In this method two sets aremaintained

(i) V is the set of vertices

(ii) E is the set of edges.



Graph G1

V(G1)= { v1,v2,v3,v4,v5,v6,v7}

E(G1)= { (v1,v2), (v1,v3), (v2,v4), (v2,v5),(v3,v6),(v3,v7)}

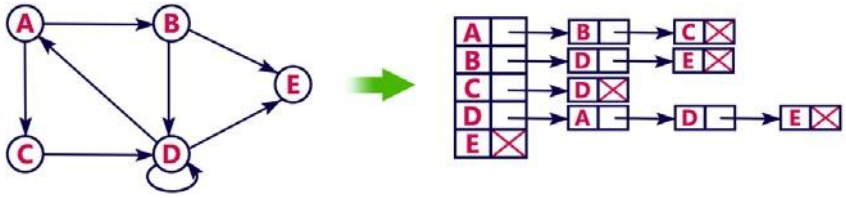
Graph G2

V(G2)= { v1,v2,v3,v4,v5,v6,v7}

E(G2)= { (v1,v2), (v1,v3), (v2,v4), (v2,v5),(v3,v4),(v3,v6), (v4,v7),(v5,v7),(v6,v7)}

4.2.2 Linked representation:

- Linked representation is another space-saving way of graph representation.
- Node structure is,

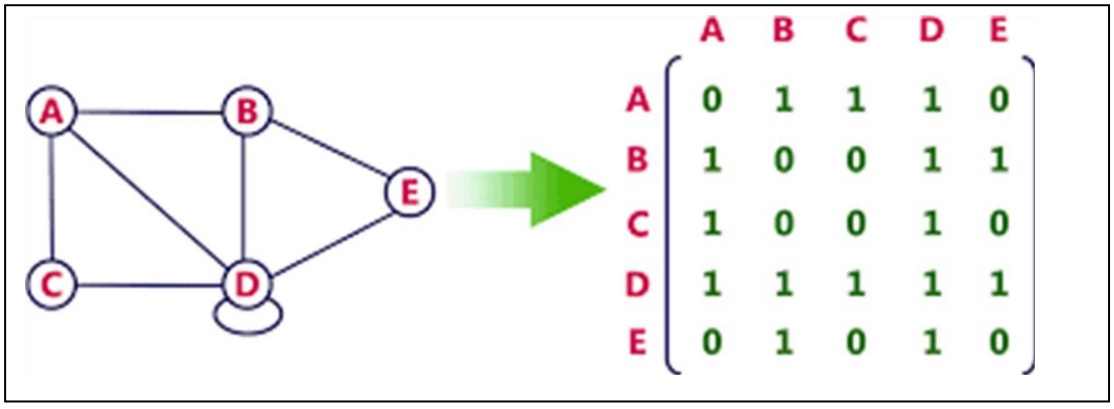


- Linked representation of graphs, the number of lists depends on the number of vertices in the graph.
- The header node in each list maintains a list of all adjacent vertices of anode for which header nodeis meant.

4.2.3 Matrix Representation

- The adjacency matrix is represents in 2D array of size n*n matrix in which n is the number ofvertices.

Ex: A[i][j]=1 means that the adjacency matrix has one edge. A[i][j]=0 means that the adjacency matrix has no edges.



Adjacency matrix for undirected graph:

- The adjacency matrix for an undirected graph is symmetric.

- The adjacency matrix for a directed graph need not be symmetric.
- The space needed to represent a graph using its adjacency matrix is n^2 locations.

Let us sum up:

Graphs can be represented in various ways:

1. **Set Representation**: Uses two sets, one for vertices (V) and one for edges (E), to define the graph structure.
2. **Linked Representation**: Utilizes linked lists where each list corresponds to a vertex and contains its adjacent vertices, providing a space-efficient way to represent the graph.
3. **Matrix Representation**: Employs a 2D array (adjacency matrix) where the entry A[i][j] indicates the presence (1) or absence (0) of an edge between vertices i and j. For undirected graphs, this matrix is symmetric; for directed graphs, it is not necessarily so.

Check your progress:

1. Which graph representation method uses a set to store adjacency information for each node?
   A. Adjacency Matrix
   B. Adjacency List
   C. Incidence Matrix
   D. Edge List
   Answer: B. Adjacency List

2. In the **adjacency matrix** representation of a graph, how is the presence of an edge between nodes iii and jjj typically denoted?
   A. By setting the entry at row iii and column jjj to 0
   B. By setting the entry at row iii and column jjj to 1 (or the weight of the edge)
   C. By setting the entry at row iii and column jjj to a negative value
   D. By storing the edge in a separate list

Answer: B. By setting the entry at row iii and column jjj to 1 (or the weight of the edge)

3. Which graph representation is most efficient for sparse graphs with relatively few edges compared to the number of vertices?
   A. Adjacency Matrix
   B. Adjacency List
   C. Incidence Matrix
   D. Edge List
   Answer: B. Adjacency List

4. In the **edge list** representation of a graph, how is the graph stored?
   A. As a list of nodes and their connections
   B. As a matrix with rows and columns representing nodes and their edges
   C. As a list of pairs, where each pair represents an edge between two nodes
   D. As a linked list where each node points to other nodes
   Answer: C. As a list of pairs, where each pair represents an edge between two nodes

5. Which of the following is true about the **incidence matrix** representation of a graph?
   A. It represents the graph using a matrix where rows represent vertices and columns represent edges
   B. It represents the graph using a matrix where both rows and columns represent vertices
   C. It is only applicable to directed graphs
   D. It uses an adjacency list to store graph connections

Answer: A. It represents the graph using a matrix where rows represent vertices and columns represent edges

4.3 GRAPH TRAVESAL

In the traversal of a binary tree there are two ways as follows.

- **Depth First search**
- **Breadth first search**

4.3.1 Depth First Search:

- Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack toremember to get the next vertex to start a search, when a dead end occurs in any iteration.
- As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, thento F and lastly to C.
- It employs the following rules.

Step 1: Rule 1 − Visit the adjacent unvisited vertex.

Step 2: Mark it as visited.

Step 3: Display it.

Step 4: Push it in a stack.

Step 5: Rule 2 − If no adjacent vertex is found, pop up a vertex from the stack.

Step 6: It will pop up all the vertices from the stack, which do not have adjacent vertices.

Step 7: Rule 3 − Repeat Rule 1 and Rule 2 until the stack is empty.

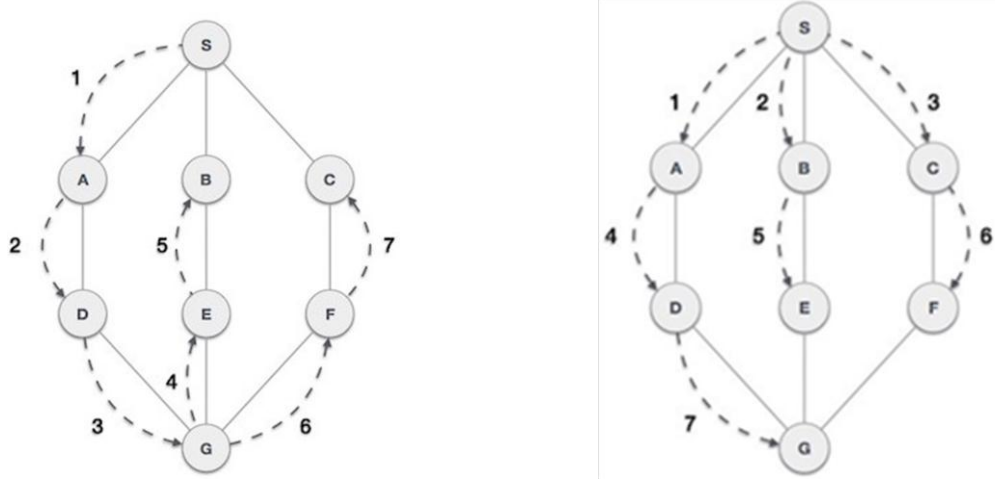


Figure: Depth First Search and Breadth First Search

4.3.2 Breadth First Search:

- Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue toremember to get the next vertex to start a search, when a dead end occurs in any iteration.
- As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and Glastly to D. It employs the following rules.

STEP 1: Rule 1 − Visit the adjacent unvisited vertex.

STEP 2: Mark it as visited.

STEP 3: Display it.

STEP 4: Insert it in a queue.

STEP 5: Rule 2 − If no adjacent vertex is found, remove the first vertex from the queue.

STEP 6: Rule 3 − Repeat Rule 1 and Rule 2 until the queue is empty.

Let us sum up:

- **Depth First Search (DFS)**: Traverses a graph deeply using a stack to backtrack when dead ends are encountered, visiting adjacent vertices and marking them as visited.

- **Breadth First Search (BFS)**: Traverses a graph level by level using a queue to track the next vertex to explore, visiting all adjacent vertices before moving to the next level.
- **DFS Process**: Visit unvisited adjacent vertices, mark and display them, push onto a stack, and pop vertices when no adjacent vertices are found, continuing until the stack is empty.
- **BFS Process**: Visit unvisited adjacent vertices, mark and display them, enqueue them, and dequeue vertices when no adjacent vertices are found, continuing until the queue is empty.
- **Traversal Order**: DFS explores as far as possible along each branch before backtracking, while BFS explores all nodes at the present depth level before moving on to nodes at the next depth level.

Check your progress:

1. Which of the following graph traversal algorithms explores a graph level by level starting from the source node?
   A. Depth-First Search (DFS) B. Breadth-First Search (BFS)
   C. Dijkstra's Algorithm D. Prim's Algorithm
   Answer:B. Breadth-First Search (BFS)

2. In which graph traversal algorithm are nodes explored by going as deep as possible along each branch before backtracking?
   A. Depth-First Search (DFS) B. Breadth-First Search (BFS)
   C. Bellman-Ford Algorithm D. Kruskal's Algorithm
   Answer:A. Depth-First Search (DFS)

3. Which data structure is typically used to implement **Breadth-First Search (BFS)**?
   A. Stack B. Queue C. Priority Queue D. Hash Table
   Answer:B. Queue

4. What is the primary use of the **Depth-First Search (DFS)** traversal algorithm?
   A. Finding the shortest path in a weighted graph

B. Finding the minimum spanning tree

C. Checking for cycles in a graph

D. Calculating shortest paths from a source node to all other nodes

Answer:C. Checking for cycles in a graph

5. Which traversal algorithm is most appropriate for finding the shortest path in an unweighted graph?

A. Depth-First Search (DFS) B. Breadth-First Search (BFS)

C. Dijkstra's Algorithm D. Bellman-Ford Algorithm

Answer:B. Breadth-First Search (BFS)

4.4 Biconnectivity of Graphs

4.4.1 Definition

- **Biconnectivity** refers to the property of a connected graph where the graph remains connected even after the removal of any single vertex. Such graphs are called **biconnected graphs**.

4.4.2 Articulation Points

- An **articulation point** (or cut vertex) is a vertex which, when removed along with its incident edges, makes the graph disconnected or increases the number of connected components.
- A graph with no articulation points is biconnected.

4.4.3 Biconnected Components

- A **biconnected component** is a maximal biconnected subgraph.
- Every edge of a graph belongs to exactly one biconnected component, but vertices can be shared among multiple components.
- Biconnected components and articulation points can be identified using Depth First Search (DFS).
- **DFS-Based Algorithm**:
    - Use a DFS tree.

- Maintain the discovery time and the lowest point reachable (low value) for each vertex.
- A vertex v is an articulation point if:
  - It is the root of the DFS tree and has more than one child.
  - It is not the root, and there exists a child u such that no vertex in the subtree rooted at u has a back edge to any of the ancestors of v.

Steps for Finding Articulation Points and Biconnected Components:

1. Perform a DFS traversal of the graph to determine the discovery time of each vertex.
2. For each vertex, determine the earliest visited vertex that is reachable from its subtree (low value).
3. Identify articulation points based on the conditions stated above.
4. Construct biconnected components using the articulation points and edges.

Applications

- **Network Reliability**: Ensuring robustness in network designs by identifying critical nodes.
- **Biological Networks**: Analyzing the resilience of biological systems.
- **Collaborative Networks**: Understanding critical connections in social and collaborative networks.

Example

Consider the graph:

A
/ \
B C
| |
D---E

- Performing DFS, let's say starting from A, we get:
    - A (root) with children B and C.
    - B connects to D and E through D.
    - C connects to E.
- E can reach back to D and B, showing multiple paths connecting parts of the graph.
- If A or E is removed, the graph remains connected, showing it's biconnected.

Let us Sum up:

- **Definition**: Biconnectivity means a connected graph remains connected if any single vertex is removed. Such graphs are called biconnected.
- **Articulation Points**: These are vertices that, if removed, would disconnect the graph or increase its number of components. A biconnected graph has no articulation points.
- **Biconnected Components**: These are maximal subgraphs where every edge belongs to exactly one component. Vertices may be shared among multiple components.
- **DFS-Based Algorithm**: Use Depth First Search (DFS) to identify articulation points and biconnected components by maintaining discovery times and low values for vertices.
- **Applications**: Useful for network reliability, biological networks, and understanding critical connections in collaborative networks.

Check your progress

1. What is a **biconnected graph**?

A. A graph where removing any single vertex does not disconnect the graph

B. A graph where removing any single edge does not disconnect the graph

C. A graph where all vertices are connected by exactly two edges

D. A graph that has exactly two connected components

Answer: A. A graph where removing any single vertex does not disconnect the graph

2. What is a **biconnected component** in a graph?

   A. A connected subgraph where any two vertices are connected by at least two disjoint paths

   B. A connected subgraph where removing any single vertex does not disconnect the subgraph

   C. A connected subgraph where any two vertices are connected by exactly two edges

   D. A connected component that includes all edges in the original graph

   Answer: B. A connected subgraph where removing any single vertex does not disconnect the subgraph

3. Which of the following algorithms can be used to find all the **biconnected components** of a graph?

   A. Depth-First Search (DFS) B. Breadth-First Search (BFS)

   C. Dijkstra's Algorithm D. Prim's Algorithm

   Answer: A. Depth-First Search (DFS)

4. In a **biconnected graph**, what is a **cut vertex**?

   A. A vertex whose removal does not affect the graph's connectivity

   B. A vertex whose removal increases the number of connected components of the graph

   C. A vertex that connects all components of the graph

   D. A vertex that has exactly two neighbors

   Answer: B. A vertex whose removal increases the number of connected components of the graph

5. Which property is **not** true for a biconnected graph?

   A. It has no articulation points

   B. It has at least two vertex-disjoint paths between any pair of vertices

   C. It remains connected after removing any single vertex

   D. It can be disconnected by removing a single edge

   Answer: D. It can be disconnected by removing a single edge

4.5 Euler Circuits in Data Structures

4.5.1 Definition

- An **Euler circuit** (or Eulerian circuit) is a path in a graph that starts and ends at the same vertex, visiting every edge exactly once.

4.5.2 Eulerian Graph

- A graph that contains an Euler circuit is called an **Eulerian graph**.

Necessary and Sufficient Conditions

1. **Connectedness**: The graph must be connected, meaning there is a path between any pair of vertices.
2. **Even Degree**: Every vertex in the graph must have an even degree (i.e., an even number of edges).

4.5.3 Euler Path vs. Euler Circuit

- An **Euler path** (or Eulerian path) is a path that visits every edge exactly once but does not necessarily start and end at the same vertex.
- A graph has an Euler path if and only if:
    1. It is connected.
    2. Exactly zero or two vertices have an odd degree.

4.5.4 Algorithm to Find Euler Circuit

- **Hierholzer's Algorithm** is commonly used:
    1. Start from any vertex with non-zero degree.
    2. Follow edges one at a time, removing them from the graph.
    3. If you return to the starting vertex and all edges are visited, you've found an Euler circuit.
    4. If you get stuck at a vertex with remaining edges, recursively find a circuit from that vertex and merge it with the main circuit.

Steps for Hierholzer's Algorithm:

1. **Initialization**: Start at any vertex with edges.
2. **Traverse**: Follow edges until you return to the starting vertex, forming a cycle.
3. **Merge**: If there are any remaining edges, find a vertex on the current cycle with unused edges, and repeat the traversal and merging process.

4.5.5 Applications

- **Network Design**: Ensuring efficient traversal of a network.
- **DNA Sequencing**: Assembling DNA fragments.
- **Graph Theory Problems**: Solving various theoretical problems and puzzles.

Example

Consider the graph:

A
/ \
B---C
\ /
D

- Vertices A, B, C, and D all have an even degree (2).
- Starting at A, an Euler circuit could be A -> B -> C -> D -> B -> D -> A.

By understanding Euler circuits and the conditions under which they exist, we can solve practical problems in network design, bioinformatics, and many other fields where traversing paths efficiently is critical.

Let us Sum up:

- **Euler Circuit**: A path that starts and ends at the same vertex, visiting every edge exactly once, is called an Euler circuit.
- **Eulerian Graph**: A graph with an Euler circuit must be connected and have all vertices with even degrees.

- **Euler Path vs. Circuit**: An Euler path visits every edge exactly once but does not necessarily start and end at the same vertex; it exists if the graph is connected and has zero or two vertices with odd degrees.
- **Hierholzer's Algorithm**: Used to find Euler circuits by starting from any vertex, following edges, and merging circuits as needed.
- **Applications**: Euler circuits are useful in network design, DNA sequencing, and solving various theoretical problems in graph theory.

Check your progress

1. What is an **Euler Circuit** in a graph?
   A. A path that visits every edge exactly once and ends at a different vertex
   B. A path that visits every vertex exactly once
   C. A path that visits every edge exactly once and returns to the starting vertex
   D. A path that visits every vertex and edge exactly once

Answer: C. A path that visits every edge exactly once and returns to the starting vertex

2. Which of the following conditions must be true for an Euler Circuit to exist in an undirected graph?
   A. All vertices must have an even degree
   B. All vertices must have an odd degree
   C. The graph must be acyclic
   D. The graph must contain no more than two vertices with an odd degree
   Answer: A. All vertices must have an even degree

3. Which algorithm is commonly used to find an **Euler Circuit** in a graph?
   A. Dijkstra’s Algorithm
   B. Depth-First Search (DFS)
   C. Fleury’s Algorithm
   D. Kruskal’s Algorithm

   Answer: C. Fleury’s Algorithm

4. In a **directed graph**, what is the necessary condition for the existence of an **Euler Circuit**?

   A. All vertices must have equal in-degree and out-degree

   B. All vertices must have at least one incoming and one outgoing edge

   C. The graph must be strongly connected

   D. The sum of the in-degrees must equal the sum of the out-degrees

   Answer: A. All vertices must have equal in-degree and out-degree

5. Which of the following is **true** for an Euler Circuit in any graph?

   A. It must pass through all vertices

   B. It must pass through every edge exactly once

   C. It must pass through every vertex exactly once

   D. It must be a shortest path between two vertices

   Answer: B. It must pass through every edge exactly once

4.7 APPLICATION OF GRAPH STRUCTURES

- Graph is an important data structure whose extensive applications are known in almost allapplication areas.
- Conversion of a particular problem into this general graph theoretic problem will rest on the reader.

1. Shortest path problem
2. Topological sorting of a graph
3. Spanning trees

4.7.1 SHORTEST PATH PROBLEM

- The shortest path problem is about finding a path between 2 vertices in a graph such that the totalsum of the edges weights is minimum.

- To find shortest problem, we may use the following two algorithms,
    1. Floyd - Warshall's Algorithms
    2. Dijkstra's Algorithm

4.7.1.2 Floyd-Warshall's Algorithms:

Definition

- **Floyd - Warshall's Algorithm** is used to find the transitive closure of a directed graph. The transitive closure of a graph is a new graph that indicates which vertices are reachable from each other.

Key Concepts

- **Transitive Closure**: If there is a path from vertex u to vertex v and from vertex v to vertex w, then there is a direct path from u to w.
- The algorithm updates the reachability matrix to include indirect connections.

Algorithm

1. **Initialization**: Start with the adjacency matrix of the graph, where A[i][j] is 1 if there is a direct edge from i to j, and 0 otherwise.
2. **Process**:
    - For each vertex k (intermediate vertex):
        - For each pair of vertices i and j:
            - Update A[i][j]: A[i][j]=A[i][j] OR (A[i][k] AND A[k][j])
3. **Result**: The matrix A now represents the transitive closure of the graph, where A[i][j] is 1 if there is a path from i to j, and 0 otherwise.

Steps

1. **Initialize** the reachability matrix R to be the same as the adjacency matrix of the graph.
2. **Iterate** through each vertex k (consider it as an intermediate point).
3. **Update** the matrix R for each pair of vertices (i,j) based on the relation:

- R[i][j] = R[i][j] OR (R[i][k] AND R[k][j])

Time Complexity

- The algorithm runs in O(V3) time, where V is the number of vertices.

Example

Consider the graph with adjacency matrix:

0 1 2

0 [0, 1, 0]

1 [0, 0, 1]

2 [1, 0, 0]

Steps of Warshall's Algorithm:

1. **Initialization**:

R = [[0, 1, 0],
[0, 0, 1],
[1, 0, 0]]

2. **Iteration with k = 0**:

R = [[0, 1, 0],
[0, 0, 1],
[1, 1, 0]]

(R[2][1] updated using R[2][0] and R[0][1])

3. **Iteration with k = 1**:

R = [[0, 1, 1],
[0, 0, 1],
[1, 1, 1]]

(R[0][2] and R[2][2] updated)

4. **Iteration with k = 2**:

R = [[1, 1, 1],
[1, 1, 1],

[1, 1, 1]]

(All pairs updated since R[0][0], R[1][0], R[1][1], R[2][0] are now reachable)

Final Transitive Closure:

R = [[1, 1, 1],
[1, 1, 1],
[1, 1, 1]]

Applications

- **Reachability Analysis**: Determine if there is a path between any pair of vertices.
- **Database Query Optimization**: Resolve indirect relationships.
- **Network Connectivity**: Analyze the connectivity of networks, such as communication or transportation networks.

Warshall's Algorithm is a fundamental technique in graph theory, providing a clear method to compute reachability information efficiently.

4.7.1.3 Dijkstra's Algorithm:

- Dijkstra's algorithm solves the single-source shortest-paths problem on a directed weighted graph G = (V, E), where all the edges are non-negative.

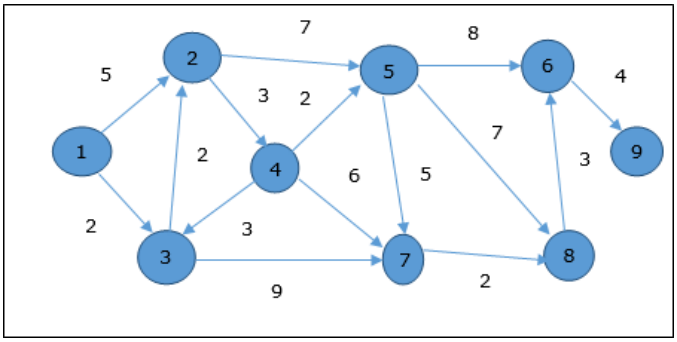
Example

- Let us consider vertex 1 and 9 as the start and destination vertex respectively.
- Initially, all the vertices except the start vertex are marked by ∞ and the start vertex is marked by 0.

Vertex	Initial	Step 1V1	Step 2V3	Step 3V2	Step 4V4	Step 5V5	Step 6V7	Step 7V8	Step 8V6
1	0	0	0	0	0	0	0	0	0
2	∞	5	4	4	4	4	4	4	4
3	∞	2	2	2	2	2	2	2	2
4	∞	∞	∞	7	7	7	7	7	7
5	∞	∞	∞	11	9	9	9	9	9
6	∞	∞	∞	∞	∞	17	17	16	16
7	∞	∞	11	11	11	11	11	11	11
8	∞	∞	∞	∞	∞	16	13	13	13
9	∞	∞	∞	∞	∞	∞	∞	∞	20

- Hence, the minimum distance of vertex 9 from vertex 1 is 20. And the path is

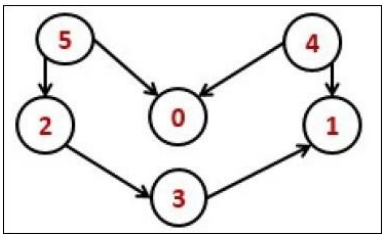
  1→ 3→ 7→ 8→ 6→ 9



Topological Sorting:

- Topological sorting is an ordering of vertices of a graph, such that if there is a path from u to v in the graph then u appears before v in the ordering.
- A topological ordering is not possible if the graph has a cycle, since for two vertices u and v on thecycle, u precedes v and v precedes u.

- A simple algorithm to find a topological ordering is to find out any vertex with in degree zero, that isvertex without any predecessor.



Algorithm:
Begin
Initially mark all nodes as unvisited
For all nodes v of the graph,
do
If v is not visited, then TopoSort (i, visited, stack)
Done
Pop and print all elements from the stack
End.

Output: 5 4 2 3 1 0

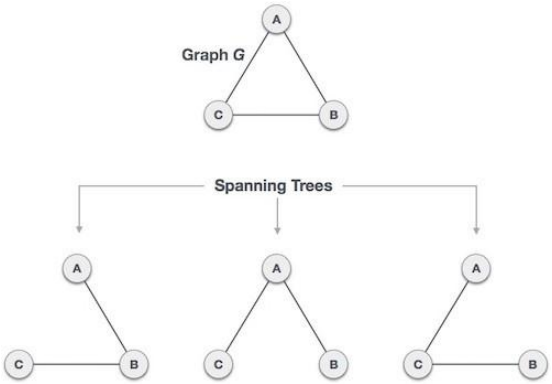
Explanation:

- The first vertex in topological sorting is always a vertex with an in-degree of 0 (a vertex with no incoming edges).
- A topological sorting of the following graph is “5 4 2 3 1 0”.
- There can be more than one topological sorting for a graph.
- Another topological sorting of the following graph is “4 5 2 3 1 0”.

4.7.1.3 MINIMUM SPANNING TREE

- A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possiblenumber of edges.
- Hence, a spanning tree does not have cycles and it cannot be disconnected.

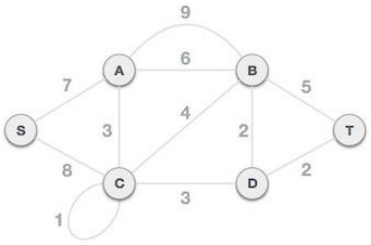
- A disconnected graph does not have any spanning tree.



- There are two methods are efficient:
  - (i) Kruskal's algorithm
  - (ii) Prim's Algorithm

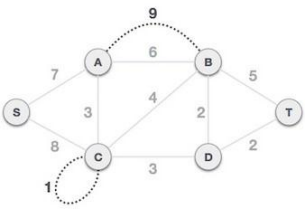
(i) Kruskal's Algorithm:

- Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach.
- This algorithm treats the graph as a forest and every node it has as an individual tree.
- A tree connects to another only and only if, it has the least cost among all available options and does notviolate MST properties.
- To understand Kruskal's algorithm let us consider the following example −



Step 1 - Remove all loops and Parallel Edges

- Remove all loops and parallel edges from the given graph.



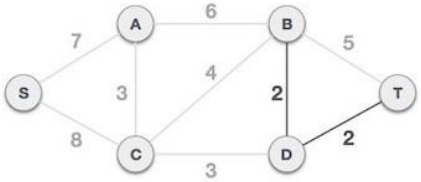
Step 2 - Arrange all edges in their increasing order of weight

- The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

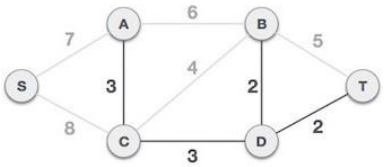
Step 3 - Add the edge which has the least weightage

- Now we start adding edges to the graph beginning from the one which has the least weight.
- In case, by adding one edge, the spanning tree property does not

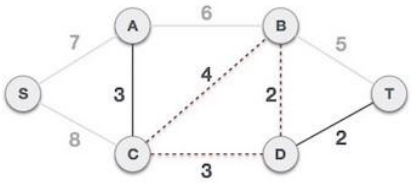


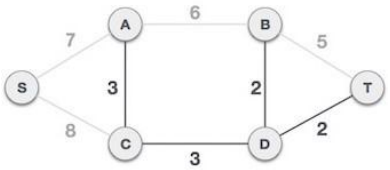
hold then we shall consider notincluding the edge in the graph.

- The least cost is 2 and edges involved are B, D and D,T.
- We add them. Adding them does not violate spanning tree properties, so we continue to our next edgeselection.

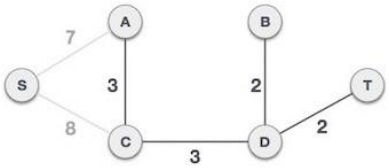


- Next cost is 3, and associated edges are A,C and C,D. We add them again –
- Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. –

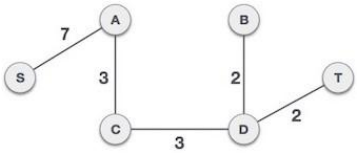




- We ignore it. In the process we shall ignore/avoid all edges that create a circuit.
- We observe that edges with cost 5 and 6 also create circuits.
- We ignore them and move on.



- Now we are left with only one node to be added.
- Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.

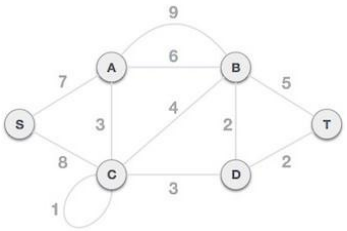


- By adding edge S, A we have included all the nodes of the graph and we now have minimum costspanning tree.

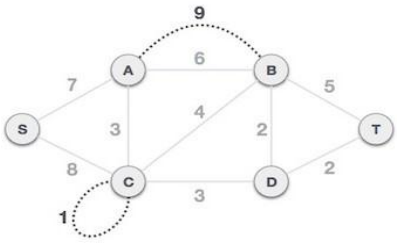
(ii) Prim's Algorithm:

- Prim's algorithm to find minimum cost spanning tree.
- Prim's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning treefrom the given graph.

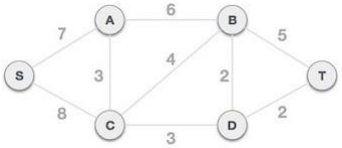
Example –



Step 1 - Remove all loops and parallel edges



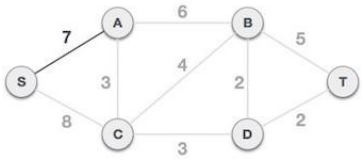
- Remove all loops and parallel edges from the given graph.



- In case of parallel edges, keep the one which has the least cost associated and remove all others.

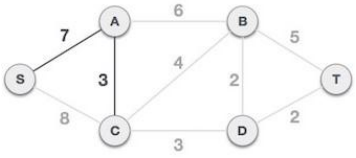
Step 3 - Check outgoing edges and select the one with less cost

- After choosing the root node S, we see that S, A and S,C are two



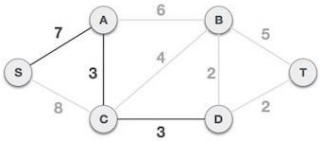
edges with weight 7 and 8, respectively. We choose the edge S, A as it is lesser than the other.

- Now, the tree S-7-A is treated as one node and we check for all edges going out from it

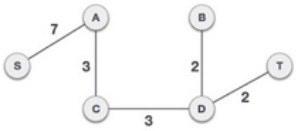


- We select the one which has the lowest cost and include it in the tree.

- After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edgesagain.
- However, we will choose only the least cost edge.
- In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



- After adding node D to the spanning tree, we now have two edges going out of it having the same cost,
  i.e. D-2-T and D-2-B.
- Thus, we can add either one.
- But the next step will again yield edge 2 as the least cost.



Hence, we are showing a spanning tree with both edges included.

Let us sum up:

- **Graph Applications**: Graphs are widely used across various fields to model relationships and solve problems, such as finding the shortest path, topological sorting, and constructing spanning trees.
- **Shortest Path Problem**: This involves finding the minimum path between two vertices. Algorithms like Floyd-Warshall and Dijkstra's are commonly used, with Floyd-Warshall focusing on all-pairs shortest paths and Dijkstra's on single-source shortest paths.
- **Topological Sorting**: This orders vertices of a Directed Acyclic Graph (DAG) so that for any edge u → v, u appears before v. It is useful for scheduling tasks and resolving dependencies.
- **Spanning Trees**: These are subsets of a graph that include all vertices with the

minimum number of edges, avoiding cycles. Kruskal's and Prim's algorithms are used to find minimum spanning trees.

- **Applications**: Graphs are instrumental in network design, database query optimization, and many other areas where efficient traversal, reachability analysis, and structure optimization are critical.

Check your progress

1. Which of the following algorithms is used to find the shortest path in a **weighted graph** with **non-negative** weights?

   A. Kruskal's Algorithm B. Dijkstra's Algorithm

   C. Bellman-Ford Algorithm D. Floyd-Warshall Algorithm

   Answer: B. Dijkstra's Algorithm

2. Which algorithm can be used to find the shortest path in a graph with **negative edge weights**?

   A. Dijkstra's Algorithm B. Prim's Algorithm

   C. Bellman-Ford Algorithm D. Breadth-First Search (BFS)

   Answer: C. Bellman-Ford Algorithm

3. What is the time complexity of **Dijkstra's Algorithm** when using a priority queue?

   A. O(V + E) B. $O(V^2)$ C. O((V + E) log V) D. O(V log E)

   Answer: C. O((V + E) log V)

4. Which of the following is not a characteristic of the Floyd-Warshall Algorithm?

   A. It finds the shortest path between every pair of vertices

   B. It is used for directed graphs

   C. It works with negative edge weights

   D. It is only used for unweighted graphs

Answer: D. It is only used for unweighted graphs

5. What is a **topological sort** of a directed acyclic graph (DAG)?

   A. A way of sorting edges in decreasing order

B. A linear ordering of vertices such that for every directed edge (u, v), vertex u comes before vertex v

C. A method to sort the vertices of a graph in ascending order of their degrees

D. A traversal method used in undirected graphs

Answer: B. A linear ordering of vertices such that for every directed edge (u, v), vertex u comes before vertex v

6. Which data structure is typically used to implement **topological sorting** using Depth-First Search (DFS)?

   A. Queue B. Stack C. Linked List D. Priority Queue

   Answer: B. Stack

7. What is the primary condition for performing **topological sorting** on a graph?

   A. The graph must be cyclic

   B. The graph must have no self-loops

   C. The graph must be a directed acyclic graph (DAG)

   D. The graph must be strongly connected

   Answer: C. The graph must be a directed acyclic graph (DAG)

8. Which of the following algorithms can be used to perform **topological sorting** of a graph?

   A. Bellman-Ford Algorithm B. Dijkstra's Algorithm

   C. Kahn's Algorithm D. Kruskal's Algorithm

   Answer:C. Kahn's Algorithm

9. Which of the following algorithms is used to find a **Minimum Spanning Tree** (MST) of a graph?

   A. Depth-First Search (DFS) B. Kruskal's Algorithm

   C. Bellman-Ford Algorithm D. Floyd-Warshall Algorithm

   Answer: B. Kruskal's Algorithm

10. In a **Minimum Spanning Tree (MST)**, the total weight of all edges is:

    A. The sum of the longest edges

B. The maximum among all possible spanning trees

C. The minimum among all possible spanning trees

D. Equal to the sum of vertex degrees

Answer: C. The minimum among all possible spanning trees

Summary:

Graphs are a versatile and powerful data structure used in a wide range of applications. They are hierarchical and consist of vertices (nodes) and edges (arcs). Various types of graphs include directed graphs, undirected graphs, weighted graphs, and more, each serving different purposes. Representation methods such as set representation, linked representation, and matrix representation provide flexibility in implementing graphs for different use cases.

Graph traversal techniques like Depth First Search (DFS) and Breadth First Search (BFS) are essential for exploring graph structures. Concepts like biconnectivity, articulation points, and biconnected components help in understanding the robustness and connectivity of graphs. Euler circuits and paths address specific traversal problems, particularly where each edge needs to be visited exactly once.Applications of graph structures are vast, including solving the shortest path problem using algorithms like Dijkstra's and Warshall’s. Topological sorting is crucial in ordering tasks while ensuring no cyclic dependencies. Minimum spanning tree algorithms, such as Kruskal's and Prim's, are fundamental in network design, ensuring efficient connectivity with minimal cost.

In summary, mastering graph structures and their associated algorithms is crucial for tackling complex problems in computer science, network design, bioinformatics, and various other fields. The ability to convert specific problems into graph theoretic problems allows for effective and efficient solutions.

Activitiy 1:

Discussion Session: Discuss the basic concepts of graphs, including vertices and edges, in a classroom setting.

Activitiy 2:

Concept Mapping: Create a concept map to show the components and properties of a graph.

Activitiy 3:

Implementation Exercise: Write a program to represent a graph using an adjacency matrix and an adjacency list.

Activitiy 4:

Comparison Activity: Compare and contrast different graph representations (adjacency matrix, adjacency list, incidence matrix) and discuss their advantages and disadvantages.

Activitiy 5:

Classification Task: Given a list of graph examples, classify each as directed, undirected, weighted, unweighted, cyclic, acyclic, etc.

Glossary:

- **Graph:** A collection of vertices (nodes) and edges (arcs) connecting pairs of vertices.
- **Vertex (Node):** A fundamental unit of a graph, representing an entity or a point.
- **Edge (Arc):** A connection between two vertices in a graph.
- **Adjacency Matrix:** A 2D array used to represent a graph, where the element at row iii and column jjj is 1 if there is an edge from vertex iii to vertex jjj, and 0 otherwise.
- **Adjacency List:** A collection of lists or arrays used to represent a graph, where each list corresponds to a vertex and contains a list of adjacent vertices.
- **Incidence Matrix:** A matrix used to represent a graph, where rows represent vertices and columns represent edges, indicating which vertices are incident to which edges.
- **Directed Graph (Digraph):** A graph where edges have a direction, going from one vertex to another.
- **Undirected Graph:** A graph where edges have no direction, meaning they connect two vertices bidirectionally.
- **Weighted Graph:** A graph where edges have associated weights or costs.

- **Unweighted Graph:** A graph where edges do not have associated weights.
- **Cyclic Graph:** A graph that contains at least one cycle.
- **Acyclic Graph:** A graph that does not contain any cycles.
- **Complete Graph:** A graph where there is an edge between every pair of vertices.
- **Bipartite Graph:** A graph whose vertices can be divided into two disjoint sets such that no two graph vertices within the same set are adjacent.
- **Breadth-First Traversal (BFS):** An algorithm for traversing or searching graph data structures, starting from a selected node and exploring neighbors level by level.
- **Queue:** A data structure used in BFS to keep track of the vertices to be explored next.
- **Depth-First Traversal (DFS):** An algorithm for traversing or searching graph data structures, starting from a selected node and exploring as far as possible along each branch before backtracking.
- **Stack:** A data structure used in DFS to keep track of the vertices to be explored next.
- **Topological Sort:** A linear ordering of vertices in a directed acyclic graph (DAG) such that for every directed edge uvuvuv, vertex uuu comes before vvv in the ordering.
- **Directed Acyclic Graph (DAG):** A directed graph with no cycles, where topological sorting is applicable.
- **Bi-connectivity:** A property of a graph where it remains connected even after the removal of any single vertex.
- **Bi-connected Component:** A maximal subgraph that remains connected upon the removal of any single vertex.
- **Articulation Point (Cut Vertex):** A vertex whose removal increases the number of connected components of the graph.
- **Cut Vertex (Articulation Point):** A vertex in a graph whose removal disconnects the graph or increases the number of connected components.
- **Euler Circuit (Eulerian Circuit):** A circuit that visits every edge of a graph exactly once and returns to the starting vertex.

- **Euler Path (Eulerian Path):** A path that visits every edge of a graph exactly once but does not necessarily return to the starting vertex.

Questions

1) What is a graph in the context of data structures?
2) Define a vertex and an edge in a graph.
3) How does a graph differ from a tree?
4) Explain the difference between an adjacency matrix and an adjacency list.
5) How do you represent a weighted graph using an adjacency matrix?
6) Describe how an incidence matrix represents a graph.
7) What are the advantages and disadvantages of using an adjacency list over an adjacency matrix?
8) What is a directed graph and how does it differ from an undirected graph?
9) Explain what a bipartite graph is.
10) What distinguishes a cyclic graph from an acyclic graph?
11) How can you determine if a graph is connected or disconnected?
12) Describe the BFS algorithm and its primary use.
13) What data structure is commonly used to implement BFS and why?
14) How does BFS ensure that all vertices at the current level are visited before moving on to the next level?
15) Provide a step-by-step BFS traversal for a given graph starting from a specified vertex.
16) Explain the DFS algorithm and its primary use.
17) What data structure is commonly used to implement DFS and why?
18) Compare and contrast DFS with BFS in terms of their traversal approach.
19) Provide a step-by-step DFS traversal for a given graph starting from a specified vertex.
20) What is a topological sort and in which type of graph is it applicable?
21) Explain why a topological sort is not possible for cyclic graphs.
22) Describe an algorithm to perform a topological sort on a directed acyclic graph (DAG).
23) Define bi-connectivity in a graph.

24) What is a bi-connected component and how is it identified?
25) Explain the concept of an articulation point (cut vertex).
26) What is a cut vertex and how does it affect the connectivity of a graph?
27) Describe a method to identify cut vertices in a graph.
28) How can the removal of a cut vertex impact the structure of a graph?
29) Define an Euler circuit and explain the conditions necessary for its existence.
30) What is the difference between an Euler path and an Euler circuit?

Further Reading and References

1. Books

- **"Introduction to Graph Theory" by Douglas B. West**
  A comprehensive book covering the basics of graph theory, representations, and advanced topics like Euler circuits, bi-connectivity, and graph traversals.
- **"Graph Theory with Applications to Engineering and Computer Science" by Narsingh Deo**
  This book provides a thorough introduction to graph theory with practical examples and applications, covering BFS, DFS, Euler circuits, and graph types.
- **"Data Structures and Algorithm Analysis in C++" by Mark A. Weiss**
  This book has a dedicated section on graph theory and covers graph traversal algorithms, topological sorting, and applications, using C++ for implementation.
- **"Algorithm Design Manual" by Steven S. Skiena**
  Focuses on algorithms for graph traversal, topological sorting, and bi-connectivity, providing practical advice and implementation tips.
- **"Algorithms" by Robert Sedgewick and Kevin Wayne**
  A well-structured book for graph algorithms, including BFS, DFS, topological sorting, and bi-connectivity, with implementation details.

2. Online Resources

- **GeeksforGeeks**
  - **Graph Data Structure**
    Covers the fundamentals of graph representation, types, and traversal techniques (BFS, DFS).
  - **Topological Sort**
    Detailed explanation with algorithms for topological sorting.
  - **Euler Circuits**
    A step-by-step guide on Euler circuits and their applications.
- **TutorialsPoint**
  - **Graph Theory**
    An easy-to-follow tutorial covering definitions, types of graphs, and traversal algorithms.
- **Brilliant.org**
  - **Graph Theory Course**
    An interactive course that covers everything from graph representation to advanced topics like bi-connectivity, cut vertices, and Euler circuits.
- **MIT OpenCourseWare: Algorithms Course**
  - **Graph Algorithms**
    Includes lectures, notes, and problem sets on graph theory, traversal algorithms, and applications.

3. Video Resources

- **YouTube: MIT OpenCourseWare**
  - **Graph Algorithms**
    A playlist of MIT's Introduction to Algorithms lectures, covering graph traversal (BFS, DFS), topological sorting, and more.
- **YouTube: Neso Academy**
  - **Graph Theory**
    A series of tutorials on graph theory, including graph representation, BFS, DFS, and applications.

- **Coursera: Algorithms Specialization by Stanford University**
  - **Graph Search, Shortest Paths, and Data Structures**
    A course that dives deep into graph algorithms like BFS, DFS, and graph traversal applications.
- **Udemy: Mastering Data Structures & Algorithms using C and C++**
  - **Graph Algorithms**
    This course includes detailed coverage of graph representation, traversals (BFS, DFS), and sorting techniques with code examples.

UNIT 5: SEARCHING, SORTING & HASHING TECHNIQUES

Searching- Linear search-Binary search-Sorting-Bubble sort-Selection sort-Insertion sort-Shell sort-Radix sort- Hashing-Hash functions-Separate chaining- Open Addressing-Rehashing Extendible Hashing

Section No	Topic	Page No
5.1	**Searching**	
5.1.1	Definition	
5.1.2	Objectives	
5.1.3	Common Searching Techniques	
	Linear Search	
	Binary Search	
5.2	**Sorting**	
5.2.1	Definition	
5.2.2	Objectives	
5.2.3	Bubble Sort	
5.2.4	Selection Sort	
5.2.5	Insertion Sort	
5.2.6	Shell Sort	
5.2.7	Radix Sort	
5.3	**Hashing**	
5.3.1	Hash Functions	
5.3.2	Separate Chaining in Hash Tables	
5.3.3	Open Addressing	
5.3.4	Rehashing	
5.3.5	Extendible Hashing	
5.3.6	Comparison	
	Summary	
	Activities	
	Points to Remember	
	Questions	
	Glossary	
	Further Reading and References	

Objectives:

- This unit aims to learn various searching techniques to efficiently locate elements within data structures.
- This unit aims to Explore different sorting algorithms to arrange elements in specific orders.
- The objective includes to Explore hashing techniques for efficient data retrieval and storage.

5.1 Searching

5.1.1 Definition

- **Searching** in data structures refers to the process of locating a specific item or element within a collection of data. The efficiency and effectiveness of searching algorithms depend on the structure and organization of the data.

5.1.2 Objectives

- **Efficiency**: Find the desired item in the shortest possible time.
- **Applicability**: Understand which searching algorithm suits specific data structures and scenarios.
- **Comparison**: Analyze advantages, disadvantages, and time complexities of different search algorithms.
- **Implementation**: Learn how to implement and integrate search algorithms into various applications and systems.

5.1.3 Common Searching Techniques

1. **Linear Search**
    - **Description**: Sequentially checks each element in a list until the target element is found or the list is exhausted.
    - **Time Complexity**: O(n) in the worst case, where n is the number of elements.
2. **Binary Search**
    - **Description**: Efficiently searches a sorted array by repeatedly dividing the search interval in half.
    - **Time Complexity**: O(log n) in the average and worst cases, suitable for sorted data.

3. **Hashing**
    - **Description**: Maps keys to values using hash functions, allowing for direct access to elements based on keys.
    - **Time Complexity**: O(1) on average for lookups, depending on the quality of the hash function and collision handling.

Applications

- **Database Systems**: Retrieving records based on query criteria.
- **Sorting Algorithms**: Locating elements during sorting processes.
- **Network Routing**: Finding optimal paths in networks.
- **Artificial Intelligence**: Searching through decision trees or state spaces.

5.1.3.1 Linear Search

Definition

- **Linear Search** is a simple searching algorithm that checks each element in a list sequentially until the target element is found or the list is exhausted.

Key Characteristics

- **Unsorted Data**: Works on both sorted and unsorted data.
- **Sequential Access**: Examines each element one by one.
- **Simplicity**: Easy to implement and understand.

Steps of the Algorithm

STEP 1: **Initialization**: Start from the first element of the list.

STEP 2: **Comparison**: Compare the current element with the target element.

STEP 3: **Match**: If the current element matches the target, return the index of the element.

STEP 4: **Continue**: If not, move to the next element.

STEP 5: **End**: Repeat steps 2-4 until the target is found or the end of the list is reached.

STEP 6: **Result**: If the target element is not found, return an indication (e.g., -1 or "not found").

Pseudocode

```
function linearSearch(arr, target):
    for i from 0 to length(arr) - 1:
        if arr[i] == target:
            return i
    return -1
```

Example

Given an array [4, 2, 7, 1, 3] and the target element 7:

- **Step-by-step Execution**:
  (i) Compare 4 with 7 (no match).
  (ii) Compare 2 with 7 (no match).
  (iii) Compare 7 with 7 (match found, return index 2).

Linear search is a fundamental search algorithm that, despite its inefficiency for large datasets, serves as an essential tool for understanding more complex search algorithms and is useful for small or simple search tasks.

Time Complexity

- **Best Case**: O(1) - Target element is the first element.
- **Average Case**: O(n) - Target element is in the middle or not present.
- **Worst Case**: O(n) - Target element is the last element or not present.

Space Complexity

- **Space Complexity**: O(1)) - Requires a constant amount of extra space.

Advantages

- **Simplicity**: Easy to understand and implement.
- **No Pre-processing**: Works on unsorted data without any additional pre-processing.
- **Versatility**: Can be used on any data structure that allows sequential access (arrays, linked lists).

Disadvantages

- **Inefficiency**: Slow for large datasets as it checks each element sequentially.
- **Scalability**: Not suitable for performance-critical applications with large datasets.

Applications

- **Small Data Sets**: Efficient for small lists where the simplicity outweighs the inefficiency.
- **Unordered Lists**: Useful when data is not sorted and the dataset is relatively small.
- **Simple Search Problems**: Quick implementation for simple search needs in various applications.

5.1.3.2 Binary Search

Definition

- **Binary Search** is an efficient searching algorithm that finds the position of a target value within a sorted array by repeatedly dividing the search interval in half.

Key Characteristics

- **Sorted Data**: Requires the data to be sorted.
- **Divide and Conquer**: Reduces the search interval by half with each step.
- **Efficiency**: Significantly faster than linear search for large datasets.

Steps of the Algorithm

STEP 1: **Initialization**: Set the initial search range (start and end indices).

STEP 2: **Middle Element**: Calculate the middle index of the current search range.

STEP 3: **Comparison**: Compare the target value with the middle element.

STEP 4: **Match**: If the middle element matches the target, return the index.

STEP 5: **Narrow Search**: If the target is less than the middle element, search the left half; if greater, search the right half.

STEP 6: **Repeat**: Continue narrowing the search range until the target is found or the range is empty.

Pseudocode

```
function binarySearch(arr, target):
    low = 0
    high = length(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Example

Given a sorted array [1, 2, 3, 4, 5, 6, 7, 8, 9] and the target element 5:

- **Step-by-step Execution**:
    1. Initial range: low=0, high=8 (middle index 4).
    2. Compare arr[4]=5 with 5 (match found, return index 4).

Time Complexity

- **Best Case**: O(1) - Target is at the middle index.
- **Average Case**: O(log n) - Each comparison halves the search range.
- **Worst Case**: O(log n) - Target not found or located at the ends of the search range.

Space Complexity

- **Iterative Implementation**: O(1) - Requires a constant amount of extra space.
- **Recursive Implementation**: O(log n) - Due to recursive call stack.

Advantages

- **Efficiency**: Much faster than linear search, especially for large datasets.

- **Predictable Performance**: Consistently performs well with a logarithmic time complexity.

Disadvantages

- **Sorted Data Requirement**: Only works on sorted arrays.
- **Data Management**: May require additional steps to sort the data before searching.

Applications

- **Large Datasets**: Efficiently search through large sorted arrays or lists.
- **Database Indexing**: Commonly used in database systems for quick data retrieval.
- **Algorithm Optimization**: Fundamental for algorithms that require fast lookup, such as those in search engines and real-time systems.

Binary search is a fundamental and efficient algorithm for searching sorted data. Its logarithmic time complexity makes it particularly suitable for applications requiring fast and frequent data retrieval.

Let us Sum up:

- **Definition and Objectives**:
  - Searching involves locating a specific item within a data collection, focusing on efficiency, applicability to data structures, comparison of algorithms, and practical implementation.
- **Common Searching Techniques**:
  - **Linear Search**: Sequentially checks each element in a list, with a time complexity of O(n). Suitable for unsorted data.
  - **Binary Search**: Efficiently searches sorted arrays by halving the search interval, with a time complexity of O(log n). Requires sorted data.
  - **Hashing**: Uses hash functions to map keys to values for direct access, with average time complexity O(1). Efficient for quick lookups.
- **Linear Search Characteristics**:
  - Simple and versatile, works on unsorted data with a time complexity of O(n) in the worst case. Ideal for small datasets but inefficient for large ones.

- **Binary Search Characteristics**:
  - Efficient for large datasets with a time complexity of O(log n), but requires sorted data. Faster than linear search and used in database indexing and large data structures.
- **Applications**:
  - Searching is crucial in various fields, including database systems, network routing, sorting algorithms, and artificial intelligence, for tasks such as record retrieval, pathfinding, and decision-making.

Check your progress

1. What is the time complexity of a linear search in the worst case?

a) O(log n) b) O(n) c) O(1)

d) O(n log n)

Answer: b) O(n)

2. Binary search can be applied to:

a) Sorted arrays only b) Unsorted arrays

c) Arrays with unique elements d) Any type of data structure

Answer: a) Sorted arrays only

3. In the best case, what is the time complexity of a binary search?

a) O(n) b) O(n log n) c) O(1) d) O(log n)

Answer: c) O(1)

4. Which of the following is a characteristic of linear search?

a) Requires sorted data

b) Can work on both sorted and unsorted data

c) Has a logarithmic time complexity

d) Requires a balanced tree

Answer: b) Can work on both sorted and unsorted data

5. What happens when you apply binary search on an unsorted array?

a) It gives incorrect results

b) It works slower

c) It works as efficiently as on sorted arrays

d) It doesn't terminate

Answer: a) It gives incorrect results

6. What is the average-case time complexity of a binary search?

a) O(1) b) O(n) c) O(n log n) d) O(log n)

Answer: d) O(log n)

7. How many comparisons are needed, in the worst case, for a linear search on a list of 8 elements?

a) 2 b) 4 c) 8 d) 10

Answer: c) 8

8. Which of the following is NOT true about binary search?

a) It reduces the search space by half in each step

b) It requires the array to be sorted

c) Its time complexity is O(n)

d) It uses a divide-and-conquer approach

Answer: c) Its time complexity is O(n)

9. How many elements are compared in the first step of a binary search on a sorted array of 100 elements?

a) 1 b) 10 c) 25 d) 50

Answer: a) 1

10. What is the worst-case scenario for binary search in terms of comparisons?

a) The target element is in the middle of the array

b) The target element is not present in the array

c) The array is empty

d) The array contains only one element

Answer: b) The target element is not present in the array

5.2 Sorting

5.2.1 Definition

- **Sorting** is the process of arranging the elements of a collection (such as an array or list) in a specific order (ascending or descending).

5.2.2 Objectives

- **Organization**: Arrange data to make other operations, such as searching and merging, more efficient.
- **Efficiency**: Minimize the time and space complexity of sorting operations.
- **Stability**: Maintain the relative order of equal elements in some sorting algorithms.

5.2.3 Bubble Sort

Definition

- **Bubble Sort** is a simple comparison-based sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

Key Characteristics

- **In-place Sorting**: Does not require additional memory for a separate array.
- **Stable**: Maintains the relative order of equal elements.
- **Simple Implementation**: Easy to understand and implement.

Steps of the Algorithm

STEP 1: **Initialization**: Start from the beginning of the list.

STEP 2: **Comparison**: Compare each pair of adjacent elements.

STEP 3: **Swap**: If a pair is in the wrong order (the first element is greater than the second for ascending order), swap them.

STEP 4: **Repeat**: Continue this process for each element in the list.

STEP 5: **Iterate**: Repeat the entire process for the whole list multiple times until no swaps are needed, indicating the list is sorted.

Pseudocode

```
function bubbleSort(arr):
    n = length(arr)
```

```
for i from 0 to n-1:
    for j from 0 to n-i-2:
        if arr[j] > arr[j+1]:
            swap(arr[j], arr[j+1])
return arr
```

Example

Given an array [5, 1, 4, 2, 8], the bubble sort process is as follows:

- **First Pass**: [1, 5, 4, 2, 8] (swap 5 and 1), [1, 4, 5, 2, 8] (swap 5 and 4), [1, 4, 2, 5, 8] (swap 5 and 2), [1, 4, 2, 5, 8] (no swap needed for 5 and 8).
- **Second Pass**: [1, 4, 2, 5, 8] (no swap needed for 1 and 4), [1, 2, 4, 5, 8] (swap 4 and 2), [1, 2, 4, 5, 8] (no swap needed for 4 and 5), [1, 2, 4, 5, 8] (no swap needed for 5 and 8).
- **Third Pass**: [1, 2, 4, 5, 8] (no swaps needed, array is sorted).

Time Complexity

- **Best Case**: O(n) - Occurs when the array is already sorted (can be achieved by adding a flag to detect no swaps).
- **Average Case**: $O(n^2)$ - Occurs when elements are in random order.
- **Worst Case**: - $O(n^2)$ Occurs when the array is sorted in reverse order.

Space Complexity

- **Space Complexity**: O(1) - Only a constant amount of extra space is required.

Advantages

- **Simplicity**: Very simple to understand and implement.
- **Stable**: Does not change the relative order of elements with equal keys.
- **Adaptive**: With an optimized version (using a flag), it can detect if the array is already sorted and stop early.

Disadvantages

- **Inefficiency**: Poor performance on large datasets due to its $O(n^2)$ time complexity.
- **Not Suitable for Large Datasets**: Generally not used for large data sets due to its inefficiency.

Applications

- **Educational**: Used for teaching purposes to introduce the concept of sorting algorithms.
- **Small Datasets**: Can be useful for sorting small lists or arrays where the simplicity of implementation outweighs the performance concerns.
- **Partially Sorted Data**: Can be efficient if the data is nearly sorted, especially with an optimized version that stops early.

Bubble sort is an introductory algorithm for sorting that highlights basic principles of comparison and swapping, but it is generally impractical for large or complex datasets due to its inefficiency.

5.2.4 Selection Sort

Definition

- **Selection Sort** is a comparison-based sorting algorithm that repeatedly selects the smallest (or largest) element from the unsorted portion of the list and moves it to the beginning.

Key Characteristics

- **In-place Sorting**: Operates directly on the original array, requiring no additional storage.
- **Not Stable**: Does not necessarily preserve the relative order of equal elements.
- **Simple Implementation**: Easy to understand and implement.

Steps of the Algorithm

STEP 1: **Initialization**: Start from the first element.

STEP 2: **Find Minimum**: Identify the smallest element in the unsorted portion of the array.

STEP 3: **Swap**: Swap this smallest element with the first unsorted element.

STEP 4: **Repeat**: Move the boundary between sorted and unsorted sections one element to the right.

STEP 5: **Iterate**: Continue until the entire list is sorted.

Pseudocode

```
function selectionSort(arr):
    n = length(arr)
    for i from 0 to n-1:
        minIndex = i
        for j from i+1 to n-1:
            if arr[j] < arr[minIndex]:
                minIndex = j
        swap(arr[i], arr[minIndex])
    return arr
```

Example

Given an array [29, 10, 14, 37, 14]:

1. **First Pass**: Find the smallest element (10) and swap it with the first element. Result: [10, 29, 14, 37, 14].
2. **Second Pass**: Find the smallest element in the remaining unsorted portion (14) and swap it with the second element. Result: [10, 14, 29, 37, 14].
3. **Third Pass**: Find the smallest element in the remaining unsorted portion (14) and swap it with the third element. Result: [10, 14, 14, 37, 29].
4. **Fourth Pass**: Find the smallest element in the remaining unsorted portion (29) and swap it with the fourth element. Result: [10, 14, 14, 29, 37].

Time Complexity

- **Best Case**: $O(n^2)$
- **Average Case**: $O(n^2)$
- **Worst Case**: $O(n^2)$

Space Complexity

- **Space Complexity**: O(1) - Requires a constant amount of extra space.

Advantages

- **Simplicity**: Very simple to understand and implement.
- **Performance on Small Datasets**: Adequate for small datasets or when the simplicity of the algorithm is more important than performance.

Disadvantages

- **Inefficiency**: Poor performance for large datasets due to its O(n2)O(n^2)O(n2) time complexity.
- **Not Stable**: Does not preserve the relative order of elements with equal keys unless modified.

Applications

- **Small Datasets**: Useful for sorting small arrays or lists where ease of implementation outweighs performance concerns.
- **Educational**: Often used for educational purposes to illustrate basic concepts of sorting algorithms.
- **Memory-Limited Environments**: Suitable when memory usage needs to be minimal since it is an in-place sorting algorithm.

Selection sort is a straightforward sorting algorithm suitable for educational purposes and small datasets due to its simplicity and ease of implementation, despite its inefficiency for larger datasets.

5.2.5 Insertion Sort

Definition

- **Insertion Sort** is a simple comparison-based sorting algorithm that builds the sorted array one element at a time by repeatedly taking the next element and inserting it into its correct position among the previously sorted elements.

Key Characteristics

- **In-place Sorting**: Operates directly on the original array, requiring no additional storage.
- **Stable**: Maintains the relative order of equal elements.
- **Adaptive**: Performs efficiently on nearly sorted data or small datasets.

Steps of the Algorithm

1. **Initialization**: Start with the second element, assuming the first element is already sorted.

2. **Insertion**: Compare the current element with the elements in the sorted portion, moving larger elements one position to the right to make room for the current element.
3. **Insert**: Place the current element in its correct position.
4. **Repeat**: Move to the next element and repeat the process until all elements are sorted.

Pseudocode

```
function insertionSort(arr):
    n = length(arr)
    for i from 1 to n:
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j = j - 1
        arr[j + 1] = key
    return arr
```

Example

Given an array [29, 10, 14, 37, 14]:

1. **First Pass**: Compare 10 with 29 and insert before 29. Result: [10, 29, 14, 37, 14].
2. **Second Pass**: Compare 14 with 29, insert before 29. Result: [10, 14, 29, 37, 14].
3. **Third Pass**: Compare 37 with 29, no change. Result: [10, 14, 29, 37, 14].
4. **Fourth Pass**: Compare 14 with 37, move 37, compare 14 with 29, move 29, insert before 29. Result: [10, 14, 14, 29, 37].

Time Complexity

- **Best Case**: O(n) - Occurs when the array is already sorted.
- **Average Case**: $O(n^2)$ - Average performance for randomly ordered elements.
- **Worst Case**: $O(n^2)$ - Occurs when the array is sorted in reverse order.

Space Complexity

- **Space Complexity**: O(1) - Requires a constant amount of extra space.

Advantages

- **Simplicity**: Easy to understand and implement.
- **Efficiency for Small/Nearly Sorted Datasets**: Performs well on small or nearly sorted datasets.
- **Stable**: Maintains the relative order of equal elements.

Disadvantages

- **Inefficiency for Large Datasets**: Poor performance on large datasets due to its $O(n^2)$time complexity.
- **Not Suitable for Large Datasets**: Generally not used for sorting large datasets.

Applications

- **Small Datasets**: Useful for sorting small arrays or lists where simplicity and stability are more important than performance.
- **Nearly Sorted Data**: Efficient for datasets that are already mostly sorted.
- **Online Algorithms**: Can be used for online sorting where elements arrive one at a time.

Insertion sort is a straightforward sorting algorithm that is efficient for small or nearly sorted datasets, making it an excellent choice for specific scenarios where simplicity and stability are prioritized.

5.2.6 Shell Sort

Definition

- **Shell Sort** is an in-place comparison-based sorting algorithm that generalizes insertion sort to allow the exchange of items that are far apart. The idea is to arrange the list of elements so that, starting anywhere, taking every h-th element produces a sorted list.

Key Characteristics

- **In-place Sorting**: Operates directly on the original array without requiring extra storage.

- **Improves Insertion Sort**: By allowing swaps of distant elements, it reduces the number of inversions, making subsequent insertion sorts faster.
- **Non-Stable**: Does not maintain the relative order of equal elements.
- **Adaptive**: Performance depends on the choice of gap sequence.

Steps of the Algorithm

1. **Initialization**: Choose an initial large gap h.
2. **Gap Reduction**: Perform a gapped insertion sort for this gap size. The gap is reduced and the process is repeated until the gap is 1.
3. **Insertion Sort for Gaps**: For each gap hhh, sort the sublists of every hhh-th element.

Pseudocode

```
function shellSort(arr):
    n = length(arr)
    gap = n // 2
    while gap > 0:
        for i from gap to n-1:
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap //= 2
    return arr
```

Example

Given an array [35, 33, 42, 10, 14, 19, 27, 44, 26, 31] and using the gap sequence [5, 3, 1]:

1. **First Gap (5)**:
    - Compare and sort elements 5 positions apart.
    - Result after first pass: [19, 14, 27, 10, 26, 35, 33, 42, 44, 31].
2. **Second Gap (3)**:
    - Compare and sort elements 3 positions apart.

- Result after second pass: [10, 14, 19, 26, 27, 31, 33, 35, 42, 44].

3. **Third Gap (1)**:
   - Perform standard insertion sort.
   - Final sorted array: [10, 14, 19, 26, 27, 31, 33, 35, 42, 44].

Time Complexity

- **Best Case**: O(nlogn) - Occurs with a good choice of gaps.
- **Average Case**: Depends on the gap sequence; typically $O(n^{3/2})$.
- **Worst Case**: $O(n^2)$ - Worst-case scenario for certain gap sequences.

Space Complexity

- **Space Complexity**: O(1) - Requires a constant amount of extra space.

Advantages

- **Efficient for Medium-Sized Arrays**: Faster than simple quadratic algorithms like bubble sort and insertion sort for medium-sized arrays.
- **Improves Insertion Sort**: Reduces the number of swaps required compared to standard insertion sort by addressing far-apart elements early on.
- **Adaptable**: Performance can be significantly improved with an optimal choice of gap sequence.

Disadvantages

- **Non-Stable**: Does not preserve the relative order of equal elements.
- **Complexity in Gap Sequence**: Performance is highly dependent on the choice of gap sequence, making it harder to implement optimally.

Applications

- **Medium-Sized Arrays**: Useful for sorting medium-sized datasets where its improved efficiency over quadratic sorting algorithms is beneficial.
- **Educational**: Often used to teach sorting algorithms and to introduce concepts of gap sequences and hybrid sorting techniques.

Shell sort provides a significant improvement over simple sorting algorithms for medium-sized arrays by using gap sequences to allow for more efficient sorting, though its performance is highly dependent on the choice of gaps.

5.2.7 Radix Sort

Definition

- **Radix Sort** is a non-comparative sorting algorithm that sorts integers by processing individual digits. It sorts the numbers digit by digit, starting from the least significant digit (LSD) to the most significant digit (MSD) or vice versa.

Key Characteristics

- **Non-Comparative**: Unlike comparison-based sorting algorithms, it doesn't compare elements directly.
- **Stable**: Maintains the relative order of equal elements.
- **Efficient for Large Numbers**: Particularly effective for sorting large lists of numbers with fixed-length digits.

Steps of the Algorithm

1. **Initialize**: Determine the maximum number of digits ddd in the largest number.
2. **Sort by Digit**: Starting from the least significant digit, sort the numbers using a stable sorting algorithm (typically counting sort).
3. **Repeat**: Repeat the sorting process for each digit until all digits are processed.

Pseudocode

```
function radixSort(arr):
    maxNumber = findMax(arr)
    numDigits = numberOfDigits(maxNumber)
    for digit from 1 to numDigits:
        arr = countingSortByDigit(arr, digit)
    return arr

function countingSortByDigit(arr, digit):
    n = length(arr)
    output = [0] * n
    count = [0] * 10
```

```
    for i from 0 to n-1:
        digitValue = getDigit(arr[i], digit)
        count[digitValue] += 1
    for i from 1 to 9:
        count[i] += count[i-1]
    for i from n-1 to 0:
        digitValue = getDigit(arr[i], digit)
        output[count[digitValue] - 1] = arr[i]
        count[digitValue] -= 1
    return output

function getDigit(number, digit):
    return (number // 10^(digit-1)) % 10
```

Example

Given an array [170, 45, 75, 90, 802, 24, 2, 66]:

1. **Initial Array**: [170, 45, 75, 90, 802, 24, 2, 66].
2. **Sort by 1st Digit (LSD)**:
    - Result: [170, 90, 802, 2, 24, 45, 75, 66].
3. **Sort by 2nd Digit**:
    - Result: [802, 2, 24, 45, 66, 170, 75, 90].
4. **Sort by 3rd Digit**:
    - Result: [2, 24, 45, 66, 75, 90, 170, 802].

Time Complexity

- **Best Case**: O(nk) - Where nnn is the number of elements and kkk is the number of digits.
- **Average Case**: O(nk) - Performance remains consistent across different inputs.
- **Worst Case**: O(nk) - Consistently linear based on input size and digit count.

Space Complexity

- **Space Complexity**: O(n+k) - Requires additional space for the output array and counting array.

Advantages

- **Efficiency**: Can be more efficient than comparison-based algorithms for large datasets of integers with a fixed number of digits.
- **Stability**: Maintains the relative order of elements with equal keys.

Disadvantages

- **Limited to Integers**: Primarily used for sorting integers and fixed-length strings.
- **Space Usage**: Requires additional space for sorting by digits.

Applications

- **Large Datasets of Integers**: Effective for sorting large arrays of integers, especially when the number of digits is fixed and not excessively large.
- **Sorting Fixed-Length Strings**: Can be adapted to sort fixed-length strings (e.g., sorting dates or words of the same length).

Radix sort is a powerful algorithm for sorting large lists of integers by processing each digit individually, providing efficient and stable sorting without direct element comparisons.

Let us sum up:

- **Definition and Objectives**:
  - Sorting arranges data in a specific order to enhance the efficiency of other operations like searching and merging, focusing on minimizing time and space complexity and ensuring stability.
- **Bubble Sort**:
  - **Definition**: Repeatedly compares adjacent elements and swaps them if needed, sorting the list incrementally.
  - **Time Complexity**: $O(n^2)$ in worst and average cases, $O(n)$ in the best case when already sorted.
  - **Space Complexity**: $O(1)$. Simple and stable but inefficient for large datasets.
- **Selection Sort**:
  - **Definition**: Selects the smallest (or largest) element from the unsorted portion and moves it to the beginning.
  - **Time Complexity**: $O(n^2)$ in all cases.

- **Space Complexity**: O(1). Simple and in-place, but not stable and inefficient for large datasets.

➢ **Insertion Sort**:

- **Definition**: Builds a sorted list one element at a time by inserting each element into its correct position.
- **Time Complexity**: $O(n^2)$ in worst and average cases, O(n) in the best case when already sorted.
- **Space Complexity**: O(1). Stable and efficient for small or nearly sorted datasets.

➢ **Shell Sort**:

- **Definition**: Generalizes insertion sort to allow swaps of distant elements using a sequence of gaps.
- **Time Complexity**: $O(n^3/2)$ on average, $O(n^2)$ in worst cases, O(n log n) in the best cases with optimal gaps.
- **Space Complexity**: O(1). Improves insertion sort performance but is non-stable and dependent on gap sequence.

➢ **Radix Sort**:

- **Definition**: Non-comparative sorting algorithm that sorts numbers digit by digit, starting from the least significant digit.
- **Time Complexity**: O(nk) where n is the number of elements and k is the number of digits.
- **Space Complexity**: O(n+k). Efficient and stable for large integer datasets with fixed digit lengths but limited to integers and requires extra space.

➢ **Applications and Characteristics**:

- **Bubble, Selection, and Insertion Sorts**: Useful for small or educational purposes due to their simplicity and ease of implementation but inefficient for large datasets.
- **Shell Sort**: Suitable for medium-sized arrays and educational use, offering improvements over simple quadratic algorithms.
- **Radix Sort**: Ideal for large datasets of integers or fixed-length strings, providing efficient and stable sorting without direct comparisons.

Check your progress

1. **Which sorting algorithm repeatedly swaps adjacent elements if they are in the wrong order?**

   a) Selection Sort b) Bubble Sort c) Insertion Sort d) Shell Sort

   Answer: b) Bubble Sort

2. **Which sorting algorithm selects the smallest element and places it at the beginning?**

   a) Selection Sort b) Radix Sort c) Shell Sort d) Insertion Sort

   Answer: a) Selection Sort

3. **What is the worst-case time complexity of Bubble Sort?**

   a) O(n log n) b) O(n) c) $O(n^2)$ d) $O(n^3)$

   Answer: c) $O(n^2)$

4. **Which sorting algorithm inserts an element into its proper place by shifting elements?**

   a) Bubble Sort b) Selection Sort c) Insertion Sort d) Radix Sort

   Answer: c) Insertion Sort

5. **What is the average time complexity of Insertion Sort?**

   a) $O(n^2)$ b) O(n log n) c) $O(n^3)$ d) O(n)

   Answer: a) $O(n^2)$

6. **Which sorting algorithm uses gaps to sort elements to improve efficiency over Insertion Sort?**

   a) Radix Sort b) Shell Sort c) Bubble Sort d) Selection Sort

   Answer: b) Shell Sort

7. **Radix Sort is primarily used to sort which type of data?**

   a) Floating point numbers b) Strings c) Integers d) Boolean values

   Answer: c) Integers

8. **Which sorting algorithm does not compare elements directly but sorts them based on digit place?**

a) Shell Sort b) Radix Sort c) Bubble Sort d) Selection Sort

Answer: b) Radix Sort

9. **What is the time complexity of Radix Sort if counting sort is used as a subroutine?**

   a) O(d(n + k)) b) O(n log n) c) $O(n^2)$ d) O(n)

   Answer: a) O(d(n + k)) (where d is the number of digits and k is the range of digit values)

10. **Which of the following sorting algorithms is NOT stable?**

    a) Bubble Sort b) Insertion Sort c) Radix Sort d) Selection Sort

    Answer: d) Selection Sort

5.3 Hashing

Definition

- **Hashing** is a technique used to map data of arbitrary size to fixed-size values, known as hash values or hash codes, which are then used to index data for quick retrieval.

Key Characteristics

- **Efficient Data Retrieval**: Provides fast access to data.
- **Fixed-Size Output**: Maps data to fixed-size hash values.
- **Handles Large Data**: Suitable for managing large datasets efficiently.

Steps in Hashing

1. **Data Input**: Take the input data.
2. **Hash Function Application**: Apply a hash function to generate a hash value.
3. **Index Mapping**: Use the hash value to determine the index in a hash table.
4. **Data Storage**: Store the data at the computed index.

Applications

- **Databases**: For indexing and quick data retrieval.
- **Cryptography**: For secure data transmission.
- **Cache Management**: For fast data lookup and retrieval.

5.3.1 Hash Functions

Definition

- **Hash Functions** are algorithms that take an input (or 'key') and return a fixed-size string of bytes. The output is typically a hash code used to index a hash table.

Key Characteristics

- **Deterministic**: The same input always produces the same output.
- **Fast Computation**: Quickly computes hash values.
- **Uniform Distribution**: Spreads inputs uniformly across the hash table to minimize collisions.
- **Minimizes Collisions**: Reduces the chance that two different inputs produce the same hash value.

Properties of a Good Hash Function

1. **Deterministic**: Same input should always yield the same hash.
2. **Uniformity**: Hash values should be uniformly distributed.
3. **Minimizing Collisions**: Different inputs should yield different hashes as much as possible.
4. **Efficiency**: Quick to compute.

Examples of Hash Functions

- **Division Method**: h(k) = k mod m, where k is the key and m is the size of the hash table.
- **Multiplication Method**: h(k) = floor(m * (k * A mod 1)), where A is a constant (0 < A < 1) and m is the table size.
- **Universal Hashing**: Uses randomization to reduce collisions.

Applications

- **Hash Tables**: Used in hash tables to index and retrieve items quickly.
- **Data Storage**: Efficient data retrieval in databases.
- **Data Validation**: Verifying data integrity with checksums and fingerprints.
- **Cryptography**: Generating secure hash codes for data security.

Example

For a hash table of size 10:

1. **Input Data**: Key = 12345
2. **Hash Function**: h(k) = k mod 10
3. **Hash Value**: h(12345) = 12345 mod 10 = 5
4. **Data Storage**: Store the data at index 5 in the hash table.

In summary, hashing and hash functions are essential tools in data structures for efficient data storage and retrieval, providing fast and reliable access to data through well-designed hash functions.

5.3.2 Separate Chaining in Hash Tables

Definition

- **Separate Chaining** is a collision resolution technique in hash tables where each bucket (or slot) of the hash table is a linked list. When a collision occurs (i.e., two different keys hash to the same index), the collided elements are stored in this linked list.

Key Characteristics

- **Collision Handling**: Handles collisions by storing multiple entries at the same hash table index.
- **Efficiency**: Provides efficient insertion and deletion operations, particularly when the hash function distributes keys uniformly.
- **Space Efficiency**: Does not require additional space proportional to the number of keys stored.

Implementation

- **Insertion**: Compute the hash value of the key. If the bucket is empty, insert the key-value pair. If not, append the pair to the end of the linked list at that index.
- **Deletion**: Locate the key in the linked list at the hashed index and remove it.
- **Search**: Compute the hash value and search for the key in the linked list at the hashed index.

Applications

- **Hash Tables**: Widely used in implementations of hash tables for handling collisions efficiently.
- **Database Indexing**: Useful in database indexing for quick data retrieval.
- **Symbol Tables**: Implementing symbol tables in compilers and interpreters.

5.3.3 Open Addressing:

Definition: Open addressing is a collision resolution technique in hashing where all elements are stored in the hash table itself.

Key Points:

- When a collision occurs (i.e., two keys hash to the same index), the algorithm probes the table to find an empty slot to place the collided key.
- Common probing methods include linear probing, quadratic probing, and double hashing.
- Requires careful handling of deletions to avoid breaking the probing sequence.

5.3.4 Rehashing:

Definition: Rehashing is the process of dynamically adjusting the size of the hash table and redistributing the stored elements to new hash positions.

Key Points:

- Typically triggered when the load factor (ratio of number of elements to table size) exceeds a certain threshold.
- Involves creating a new, larger hash table and re-inserting all existing elements into this new table according to their new hash values.
- Helps reduce collisions and improves the efficiency of hash table operations over time.

5.3.5 Extendible Hashing:

Definition: Extendible hashing is a dynamic hashing technique that adapts to the number of records in the hash table without requiring extensive data movement during insertions and deletions.

Key Points:

- Organizes the hash table into a directory of buckets, each containing a fixed number of slots.
- Directory size grows dynamically as needed, based on the hash values of inserted keys.
- Provides efficient search, insert, and delete operations, typically with minimal data movement.
- Ideal for applications where the size of the hash table is expected to grow or shrink dynamically.

Comparison:

- **Open Addressing vs. Chaining:** Open addressing directly stores all elements in the hash table, whereas chaining uses linked lists or other structures at each hash table slot to handle collisions.
- **Rehashing vs. Resizing:** Rehashing involves creating a new hash table and redistributing elements, whereas resizing simply increases or decreases the size of the existing hash table.
- **Extendible Hashing vs. Linear Hashing:** Extendible hashing uses a global directory to manage buckets, whereas linear hashing uses a dynamic splitting technique to split overflowing buckets into two.

These techniques are fundamental in designing efficient hash tables that can handle dynamic data and minimize collisions, each with its own advantages and trade-offs depending on the specific application requirements.

Let us Sum up:

- **Definition of Hashing**:
  - Hashing maps data of arbitrary size to fixed-size hash values for quick retrieval, using a hash function to generate indexes in a hash table.
- **Key Characteristics of Hashing**:
  - Provides efficient data retrieval, generates fixed-size hash values, and manages large datasets effectively.

- **Hash Functions**:
  - Algorithms that produce a fixed-size output (hash code) from an input key, aiming for determinism, fast computation, uniform distribution, and minimizing collisions.
- **Separate Chaining**:
  - A collision resolution technique where each hash table bucket is a linked list. Collisions are handled by storing multiple entries in the linked list at the same index.
- **Open Addressing**:
  - A collision resolution technique where all elements are stored directly in the hash table. When a collision occurs, the table is probed to find an empty slot using methods like linear, quadratic, or double hashing.
- **Rehashing**:
  - The process of resizing a hash table and redistributing elements to reduce collisions. It is triggered when the load factor exceeds a threshold.
- **Extendible Hashing**:
  - A dynamic hashing technique using a directory of buckets that grows as needed. It efficiently handles insertions and deletions with minimal data movement and adapts to changes in the number of records.

Summary:

- **Linear Search:** A straightforward searching algorithm that sequentially checks each element in a list until a match is found or the entire list has been searched. Time complexity is O(n).
- **Binary Search:** A more efficient searching algorithm for sorted arrays, dividing the search interval in half repeatedly until the target element is found or determined to be absent. Time complexity is O(log n).
- **Bubble Sort:** A simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. Time complexity is O(n^2).
- **Selection Sort:** A sorting algorithm that repeatedly selects the smallest (or largest) element from the unsorted portion of the array and swaps it with the first unsorted element. Time complexity is O(n^2).

- **Insertion Sort:** A sorting algorithm where each element is taken from the unsorted portion and inserted into its correct position in the sorted portion of the list. Time complexity is O(n^2), but it can be efficient for small data sets.
- **Shell Sort:** An extension of insertion sort that allows the exchange of items that are far apart to produce partially sorted arrays that can be efficiently sorted, eventually by insertion sort. Time complexity varies but is generally better than O(n^2).
- **Radix Sort:** A non-comparative sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value. Time complexity is O(nk) where n is the number of keys and k is the average length of the keys.
- **Hashing:** A technique that maps keys to indices of a hash table using hash functions. Enables efficient retrieval, insertion, and deletion operations.
- **Hash Functions:** Functions that map data of arbitrary size to data of a fixed size. They are essential for the implementation and performance of hashing algorithms.
- **Separate Chaining:** A collision resolution technique where each bucket of the hash table is independent and stores a linked list of entries that hash to the same index.
- **Open Addressing:** A collision resolution technique where all elements are stored within the hash table itself, typically by probing or searching through alternative locations.
- **Rehashing:** The process of creating a new hash table and moving all the elements from the current hash table into the new one, typically when the load factor exceeds a specified threshold.
- **Extendible Hashing:** A dynamic hashing technique where the hash table grows and shrinks dynamically as the data size changes, using directory-based and bucket-based structures to manage collisions.

Check your progress

1. **What is the primary purpose of a hash function in hashing?**

   a) To store data sequentially

   b) To map a given input to a unique fixed-size output

   c) To perform data compression

   d) To encrypt data for security

   Answer: b) To map a given input to a unique fixed-size output

2. **In separate chaining, collisions are handled by:**

   a) Linear probing

   b) Using a linked list at each index

   c) Using double hashing

   d) Rehashing

   Answer: b) Using a linked list at each index

3. **Which of the following is a drawback of Open Addressing?**

   a) Requires additional memory for linked lists

   b) Needs secondary hash functions

   c) Clustering of elements in the table

   d) Unbounded insertion time

   Answer: c) Clustering of elements in the table

4. **What is the key benefit of using Rehashing?**

   a) Decreases the load factor by increasing table size

   b) Increases collision rate

   c) Reduces memory usage

   d) Helps in sorting elements faster

   Answer: a) Decreases the load factor by increasing table size

5. **Which technique is employed in Open Addressing to resolve collisions?**

   - a) Linear probing
   - b) Separate chaining
   - c) Radix sort
   - d) Binary search

   Answer: a) Linear probing

6. **Double Hashing, as a collision resolution method, uses:**

a) Two hash functions to resolve collisions

b) Separate chaining combined with probing

c) Linked lists at each table index

d) Rehashing to increase table size

Answer: a) Two hash functions to resolve collisions

7. **In Extendible Hashing, the size of the hash table:**

a) Is fixed and never changes

b) Doubles dynamically based on load factor

c) Reduces when elements are deleted

d) Stays fixed but collisions are handled externally

Answer: b) Doubles dynamically based on load factor

8. **What is a primary concern when choosing a hash function?**

a) Making the hash table size very small

b) Ensuring high clustering of elements

c) Minimizing collisions

d) Maximizing the hash value output size

Answer: c) Minimizing collisions

9. **Which collision resolution strategy ensures that no linked lists are used?**

a) Separate chaining

b) Open addressing

c) Double hashing

d) Rehashing

Answer: b) Open addressing

10. **In rehashing, when is the hash table size increased?**

a) When the load factor exceeds a threshold

b) After every insertion

c) When the hash function fails

d) Every time a collision occurs

Answer: a) When the load factor exceeds a threshold

Activities:

Activity 1 : Implement Linear Search

- **Description:** Write a function in your preferred programming language (e.g., Python, Java, JavaScript) to perform linear search on an array of integers. Ensure it handles edge cases like the element not being found.
- **Objective:** To understand the basic mechanics of linear search and practice implementing simple search algorithms.

Activity 2 : Debugging Binary Search

- **Description:** Given a binary search implementation that contains bugs (e.g., incorrect mid-point calculation or handling of edge cases), debug and correct the code to ensure it functions correctly.
- **Objective:** Improve understanding of binary search by identifying common implementation errors and practicing debugging techniques.

Activity 3 : Visualize Bubble Sort

- **Description:** Create a visualization tool (using a GUI framework or online coding platform) that demonstrates how bubble sort works step-by-step on an array of integers. Highlight each comparison and swap operation visually.
- **Objective:** Gain a deeper understanding of bubble sort's iterative nature and visualize its inefficiency for large datasets.

Activity 4 : Sorting Algorithm Comparison

- **Description:** Implement selection sort alongside bubble sort and insertion sort. Use a benchmarking tool or timing functions to compare their performance on arrays of varying sizes (e.g., 1000, 5000, 10000 elements).
- **Objective:** Compare the time complexity and practical performance differences between sorting algorithms, emphasizing selection sort's simplicity and its role in algorithmic efficiency.

Activity 5: Recursive Insertion Sort

- **Description:** Implement insertion sort using both iterative and recursive approaches. Compare their implementations and discuss the advantages and disadvantages of each method.
- **Objective:** Explore different implementation strategies for insertion sort and deepen understanding of recursion in sorting algorithms.

Glossary:

- **Searching:** The process of finding a specific item (or record) in a collection of items (or records).
- **Linear Search:** A sequential searching algorithm that checks each element in a list until the target element is found or the end of the list is reached.
- **Binary Search:** A searching algorithm for sorted arrays that repeatedly divides the search interval in half, eliminating half of the remaining elements at each step.
- **Sorting:** Arranging items in a specific order, typically numerical or lexicographical (alphabetical).
- **Bubble Sort:** A simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
- **Selection Sort:** A sorting algorithm that divides the input list into two parts: the sorted part at the left end and the unsorted part at the right end. It repeatedly selects the smallest (or largest) element from the unsorted part and swaps it with the first unsorted element.
- **Insertion Sort:** A sorting algorithm that builds the final sorted array (or list) one item at a time by inserting each item into its correct position within the sorted part of the array.

- **Shell Sort:** An extension of insertion sort that allows the exchange of items that are far apart to produce partially sorted arrays that can be efficiently sorted, eventually by insertion sort.

- **Radix Sort:** A non-comparative sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.

- **Hashing:** The process of mapping data of arbitrary size to data of a fixed size using a hash function.

- **Hash Functions:** Functions that map data of arbitrary size to data of a fixed size (hash value), typically used in hash tables to locate data quickly.

- **Separate Chaining:** A collision resolution technique in hash tables where each bucket (or slot) of the hash table is independent and can store multiple entries in a linked list or similar data structure.

- **Open Addressing:** A collision resolution technique in hash tables where all elements are stored within the hash table itself, typically by probing or searching through alternative locations.

- **Rehashing:** The process of creating a new hash table and moving all the elements from the current hash table into the new one, usually triggered by exceeding a load factor threshold.

- **Extendible Hashing:** A dynamic hashing technique where the hash table grows and shrinks dynamically as the data size changes, using directory-based and bucket-based structures to manage collisions and facilitate efficient operations.

Questions:

- **Searching:**

    1. What are the fundamental operations involved in searching algorithms?
    2. Compare and contrast linear search and binary search in terms of time complexity and suitability for different types of data.
    3. Describe scenarios where each of the searching algorithms (linear search and binary search) would be most effective.

- **Linear Search:**
    1. Explain the basic steps involved in implementing linear search.
    2. What is the time complexity of linear search? How does it perform on large datasets?
    3. Discuss advantages and disadvantages of using linear search compared to other search algorithms.

- **Binary Search:**
    1. How does binary search work? Describe its algorithmic approach.
    2. What are the prerequisites for using binary search on a dataset?
    3. Explain the time complexity of binary search. Under what conditions is binary search most efficient?
- **Sorting:**

    1. Why is sorting important in computer science and real-world applications?
    2. Compare and contrast different sorting algorithms based on their time complexity and stability.
    3. How does the choice of sorting algorithm depend on the characteristics of the dataset (e.g., size, initial order)?

- **Bubble Sort:**
    1. Describe the basic operation of bubble sort. How does it work?

2. What is the worst-case time complexity of bubble sort? How can this be improved?
3. Provide an example of when bubble sort might be a suitable choice of algorithm.

- **Selection Sort:**
    1. Explain the selection sort algorithm step by step.
    2. How does selection sort perform in terms of time complexity? Is it stable?
    3. Discuss scenarios where selection sort is preferred over other sorting algorithms.

- **Insertion Sort:**
    1. How does insertion sort operate? Describe its key steps.
    2. What is the best-case time complexity of insertion sort? How does it compare to its average and worst cases?
    3. Provide examples of practical applications where insertion sort is beneficial.

- **Shell Sort:**
    1. What are the main principles behind shell sort?
    2. How does the choice of gap sequence impact the efficiency of shell sort?
    3. Compare shell sort with insertion sort in terms of time complexity and practical performance.

- **Radix Sort:**
    1. How does radix sort differ from comparison-based sorting algorithms?
    2. Explain the concept of significant digits in radix sort. How does it affect its time complexity?
    3. Provide examples of when radix sort is advantageous over other sorting algorithms.

- **Hashing:**
    1. What is hashing? How is it used to store and retrieve data efficiently?
    2. Describe the components of a hash function. What makes a good hash function?
    3. Compare direct addressing with collision resolution techniques in hashing.

- **Hash Functions:**
    1. What are the properties of a good hash function?
    2. Explain collision resolution strategies used with hash functions.
    3. How can hash functions be applied in data security and cryptography?

- **Separate Chaining:**
    1. How does separate chaining handle collisions in hash tables?
    2. Discuss the trade-offs of using linked lists versus other data structures for separate chaining.
    3. Provide an example scenario where separate chaining is advantageous.

- **Open Addressing:**
    1. What is open addressing? How does it differ from separate chaining?
    2. Describe the different probing techniques used in open addressing.
    3. Discuss the challenges and benefits of open addressing compared to separate chaining.

- **Rehashing:**
    1. When and why is rehashing necessary in hash tables?
    2. Explain how load factor influences the decision to rehash.
    3. What are the steps involved in rehashing a hash table?

- **Extendible Hashing:**
    1. How does extendible hashing dynamically resize hash tables?
    2. Describe the structure of extendible hashing using directory-based and bucket-based approaches.

3. Compare extendible hashing with other dynamic hashing techniques in terms of performance and scalability.

Further Reading and References

Books:

1. **Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein**
2. **Data Structures and Algorithm Analysis in C++ by Mark A. Weiss**
3. **Algorithms by Robert Sedgewick and Kevin Wayne**
4. **The Algorithm Design Manual" by Steven S. Skiena**

Online Resources:

1. **GeeksforGeeks**
   - Detailed tutorials and explanations of all algorithms mentioned (Searching, Sorting, and Hashing).
   - URL: GeeksforGeeks
   - Topics:
     - Linear Search
     - Binary Search
     - Bubble Sort
     - Selection Sort
     - Insertion Sort
     - Shell Sort
     - Radix Sort
     - Hashing
     - Separate Chaining
     - Open Addressing
2. **Tutorialspoint**
   - Free online tutorials for searching and sorting algorithms.
   - URL: Tutorialspoint - Sorting
   - URL: Tutorialspoint - Hashing

3. **Khan Academy**
    - Interactive tutorials on algorithms and data structures, including searching and sorting algorithms.
    - URL: Khan Academy
4. **Coursera**
    - Offers online courses on algorithms by top universities.
    - URL: Coursera - Algorithms Specialization by Stanford

Video Resources:

1. **YouTube Channels:**
    - **mycodeschool**:
        - Comprehensive explanations of searching, sorting, and hashing algorithms with animations.
        - Linear Search & Binary Search
        - Sorting Algorithms
        - Hashing and Hash Functions
    - **Abdul Bari**:
        - Easy-to-follow video tutorials on various algorithms, including step-by-step explanations of searching, sorting, and hashing.
        - Binary Search
        - Sorting Algorithms Playlist
        - Hashing & Separate Chaining
    - **CS50 by Harvard University**:
        - High-quality lectures covering searching, sorting, and hashing techniques.
        - Searching and Sorting
2. **Udemy**
    - Online courses for in-depth learning of algorithms and data structures, with practical coding exercises.
    - Mastering Data Structures & Algorithms using C and C++
    - Data Structures and Algorithms Bootcamp