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 LAB

(Candidates admitted from 2024 onwards)

PERIYAR UNIVERSITY

CENTRE FOR DISTANCE AND ONLINE EDUCATION (CDOE)

B.Sc COMPUTER SCIENCE 2024 admission onwards

CORE COURSE – IV
Data Structure and Algorithms Lab

Prepared by:

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

DATA STRUCTURE AND ALGORITHMS LAB

Sl. No	Contents
1	Write a program to implement the List ADT using arrays and linked lists.
2	Write a programs to implement the following using a singly linked list. · Stack ADT · Queue ADT
3	Write a program that reads an infix expression, converts the expression to postfix form and then evaluates the postfix expression (use stack ADT).
4	Write a program to implement priority queue ADT.
5	Write a program to perform the following operations: · Insert an element into a binary search tree. · Delete an element from a binary search tree. · Search for a key element in a binary search tree.
6	Write a program to perform the following operations · Insertion into an AVL-tree · Deletion from an AVL-tree
7	Write a programs for the implementation of BFS and DFS for a given graph.
8	Write a programs for implementing the following searching methods: · Linear search · Binary search.
9	Write a programs for implementing the following sorting methods: · Bubble sort · Selection sort · Insertion sort · Radix sort.

Objective:

- To understand the concepts of ADTs
- To learn linear data structures-lists, stacks, queues
- To learn Tree structures and application of trees
- To learn graph structures and application of graphs
- To understand various sorting and searching

INDEX

PROGRAM -1

AIM:

To write a python program to implement the List ADT using arrays and linked lists.

ALGORITHM:

Step 1: **Define the ArrayList Class:**

- Initialize **ArrayList** class with an empty list **arr**.
- Define **append** method to add elements to the list.
- Define **display** method to print the contents of the list.

Step 2: **Define the Node Class:**

- Initialize **Node** class with **data** and **next** attributes.

Step 3: **Define the LinkedList Class:**

- Initialize **LinkedList** class with **head** attribute pointing to the first node.
- Define **append** method to add elements to the end of the linked list.
- Define **display** method to print the elements of the linked list.

Step 4: **Test List ADT using Arrays:**

- Create an instance of **ArrayList**.
- Append elements to the list.
- Display the contents of the list.

Step 5: **Test List ADT using Linked List:**

- Create an instance of **LinkedList**.

- Append elements to the linked list.
- Display the elements of the linked list.

CODING:

```
class ArrayList:
    def __init__(self):
        self.arr = []

    def append(self, value):
        self.arr.append(value)

    def display(self):
        print("Array List:", self.arr)



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

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, value):
        new_node = Node(value)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
```

```
    def display(self):
        current = self.head
        linked_list_str = ""
        while current:
            linked_list_str += str(current.data) + " "
            current = current.next
        print("Linked List:", linked_list_str)

# Test List ADT using arrays
array_list = ArrayList()
array_list.append(10)
array_list.append(20)
array_list.append(30)
array_list.display()

# Test List ADT using linked list
linked_list = LinkedList()
linked_list.append(10)
linked_list.append(20)
linked_list.append(30)
linked_list.display()
```

OUTPUT

Array List: [10, 20, 30]

Linked List: 10 20 30

Result:

The python program to implement the List ADT using arrays and linked lists has been executed and the results are verified successfully.

PROGRAM -2

AIM:

To write python program to implement Stack ADT and Queue ADT using a singly linked list.

ALGORITHM:

Step 1: **Define the Node Class:**

- Initialize **Node** class with **data** and **next** attributes.

Step 2: **Define the Stack Class:**

- Initialize **Stack** class with **top** attribute pointing to the topmost element in the stack.
- Define **is_empty** method to check if the stack is empty.
- Define **push** method to push elements onto the stack.
- Define **pop** method to pop the top element from the stack.
- Define **peek** method to return the top element of the stack without removing it.
- Define **display** method to print the elements of the stack.

Step 3: **Test Stack ADT:**

- Create an instance of **Stack**.
- Push elements onto the stack.
- Display the contents of the stack.
- Peek the top element of the stack.
- Pop an element from the stack.

- Display the updated contents of the stack.

Algorithm for Queue ADT using a singly linked list:

Step 1: **Define the Node Class:**

- Initialize **Node** class with **data** and **next** attributes.

Step 2: **Define the Queue Class:**

- Initialize **Queue** class with **front** and **rear** attributes.
- Define **is_empty** method to check if the queue is empty.
- Define **enqueue** method to add elements to the rear of the queue.
- Define **dequeue** method to remove elements from the front of the queue.
- Define **display** method to print the elements of the queue.

Step 3: **Test Queue ADT:**

- Create an instance of **Queue**.
- Enqueue elements into the queue.
- Display the contents of the queue.
- Dequeue an element from the queue.
- Display the updated contents of the queue.

CODING:

Stack ADT using a Singly Linked List:

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


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

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

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

    def pop(self):
        if self.is_empty():
            print("Stack is empty")
            return None
        popped_value = self.top.data
        self.top = self.top.next
        return popped_value

    def peek(self):
        if self.is_empty():
            print("Stack is empty")
            return None
```

```
        return self.top.data

    def display(self):
        current = self.top
        stack_str = ""
        while current:
            stack_str += str(current.data) + " "
            current = current.next
        print("Stack:", stack_str)


# Test Stack ADT
stack = Stack()
stack.push(10)
stack.push(20)
stack.push(30)
stack.display()
print("Peek:", stack.peek())
print("Popped:", stack.pop())
stack.display()
```

OUTPUT:

Stack: 30 20 10
Peek: 30
Popped: 30
Stack: 20 10

Queue ADT using a Singly Linked List:

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


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

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

    def enqueue(self, value):
        new_node = Node(value)
        if self.is_empty():
            self.front = new_node
            self.rear = new_node
        else:
            self.rear.next = new_node
            self.rear = new_node

    def dequeue(self):
        if self.is_empty():
            print("Queue is empty")
            return None
        dequeued_value = self.front.data
        if self.front == self.rear:
            self.front = None
            self.rear = None
        else:
```

```
            self.front = self.front.next
        return dequeued_value

    def display(self):
        current = self.front
        queue_str = ""
        while current:
            queue_str += str(current.data) + " "
            current = current.next
        print("Queue:", queue_str)


# Test Queue ADT
queue = Queue()
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)
queue.display()
print("Dequeued:", queue.dequeue())
queue.display()
```

OUTPUT:

Queue: 10 20 30
Dequeued: 10
Queue: 20 30

RESULT:

The python program to implement Stack ADT and Queue ADT using a singly linked list has been executed and the results are verified successfully.

. PROGRAM -3

AIM:

To write a python program that reads an infix expression, converts the expression to postfix form and then evaluates the postfix expression using stack ADT.

ALGORITHM:

Step 1: **Define the Stack Class:**

- Initialize **Stack** class with **items** attribute as an empty list.
- Define **is_empty** method to check if the stack is empty.
- Define **push** method to push items onto the stack.
- Define **pop** method to pop the top item from the stack.
- Define **peek** method to return the top item of the stack without removing it.
- Define __**str**__ method to return a string representation of the stack.

Step 2: **Define infix_to_postfix function:**

- Define a dictionary **precedence** to store operator precedence.
- Initialize an empty string **postfix** to store the postfix expression.
- Initialize a stack.
- Iterate through each character in the infix expression:
  - If the character is an operand, append it to the **postfix** string.
  - If the character is an opening parenthesis, push it onto the stack.
  - If the character is a closing parenthesis, pop operators from the stack and append them to **postfix** until an opening parenthesis is encountered.
  - If the character is an operator, pop operators from the stack and append them to **postfix** if they have higher precedence than the current operator. Then push the current operator onto the stack.
- Pop any remaining operators from the stack and append them to **postfix**.
- Return the **postfix** expression.

Step 3: **Define evaluate_postfix function:**

- Initialize a stack.
- Iterate through each character in the postfix expression:
  - If the character is a digit, push it onto the stack.
  - If the character is an operator, pop two operands from the stack, perform the operation, and push the result back onto the stack.
- After iterating through the postfix expression, the result will be the only item left on the stack.
- Return the result.

Step 4: **Define main function:**

- Prompt the user to enter an infix expression.
- Convert the infix expression to postfix using the **infix_to_postfix** function.
- Print the postfix expression.
- Evaluate the postfix expression using the **evaluate_postfix** function.
- Print the result of the evaluation.

CODING:

```
class Stack:
    def __init__(self):
        self.items = []

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

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

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            print("Stack is empty")
```

```
    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            print("Stack is empty")

    def __str__(self):
        return str(self.items)

def infix_to_postfix(expression):
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
    postfix = ""
    stack = Stack()

    for char in expression:
        if char.isalnum():
            postfix += char
        elif char == '(':
            stack.push(char)
        elif char == ')':
            while not stack.is_empty() and stack.peek() != '(':
                postfix += stack.pop()
            stack.pop()  # Remove the opening parenthesis
        else:
          while not stack.is_empty() and precedence.get(stack.peek(), 0) >=
precedence.get(char, 0):
                postfix += stack.pop()
            stack.push(char)

    while not stack.is_empty():
        postfix += stack.pop()
```

```
    return postfix

def evaluate_postfix(postfix):
    stack = Stack()

    for char in postfix:
        if char.isdigit():
            stack.push(int(char))
        else:
            operand2 = stack.pop()
            operand1 = stack.pop()
            if char == '+':
                stack.push(operand1 + operand2)
            elif char == '-':
                stack.push(operand1 - operand2)
            elif char == '*':
                stack.push(operand1 * operand2)
            elif char == '/':
                stack.push(operand1 / operand2)

    return stack.pop()

def main():
    infix_expression = input("Enter the infix expression: ")
    postfix_expression = infix_to_postfix(infix_expression)
    print("Postfix expression:", postfix_expression)
    result = evaluate_postfix(postfix_expression)
    print("Result of evaluation:", result)

if __name__ == "__main__":
    main()
```

OUTPUT:

Enter the infix expression: a+b-c*d/e^f

Postfix expression: ab+cd*ef^/-

RESULT:

The python program that reads an infix expression, converts the expression to postfix form and then evaluates the postfix expression using stack ADT has been executed and the results are verified successfully.

.

PROGRAM -4

AIM:

To write a python program to implement priority queue ADT.

ALGORITHM:

Step 1: Define the PriorityQueue Class:

- Initialize PriorityQueue class with items attribute as an empty list.
- Define is_empty method to check if the priority queue is empty.
- Define enqueue method to add items to the priority queue with a specified priority.
- Define dequeue method to remove and return the item with the highest priority from the priority queue.
- Define peek method to return the item with the highest priority without removing it.
- Define __str__ method to return a string representation of the priority queue.

Step 2: Test the Priority Queue ADT:

- Create an instance of PriorityQueue.
- Enqueue tasks with priorities into the priority queue.
- Print the contents of the priority queue.
- Peek at the task with the highest priority.
- Dequeue the task with the highest priority.
- Print the updated contents of the priority queue.

CODING:

```
class PriorityQueue:
    def __init__(self):
        self.items = []

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

    def enqueue(self, item, priority):
        self.items.append((item, priority))
        self.items.sort(key=lambda x: x[1], reverse=True)

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)[0]
        else:
            print("Priority queue is empty")

    def peek(self):
        if not self.is_empty():
            return self.items[0][0]
        else:
            print("Priority queue is empty")

    def __str__(self):
        return str([item[0] for item in self.items])


# Test the Priority Queue ADT
pq = PriorityQueue()
pq.enqueue("Task 1", 3)
```

```
pq.enqueue("Task 2", 1)
pq.enqueue("Task 3", 2)
print("Priority Queue:", pq)
print("Peek:", pq.peek())
print("Dequeue:", pq.dequeue())
print("Priority Queue:", pq)
```

OUTPUT:

Priority Queue: ['Task 1', 'Task 3', 'Task 2']

Peek: Task 1

Dequeue: Task 1

Priority Queue: ['Task 3', 'Task 2']

RESULT:

The python program to implement priority queue ADT has been executed and the results are verified successfully.

.

PROGRAM -5

AIM:

To write a python program to perform Insertion, Deletion and search operations in a binary search tree

ALGORITHM:

Step 1: **Define the TreeNode Class:**

- Initialize **TreeNode** class with **key**, **left**, and **right** attributes.

Step 2: **Define the insert Function:**

- Define **insert** function to insert a new node with a given key into the BST.
- If the root is None, create a new node and return it.
- If the key is less than the root's key, recursively insert into the left subtree.
- If the key is greater than the root's key, recursively insert into the right subtree.
- Return the root.

Step 3: **Define the minValueNode Function:**

- Define **minValueNode** function to find the node with the minimum key value in a given subtree.
- Traverse the left subtree until a node with a None left child is found.
- Return the current node.

Step 4: **Define the deleteNode Function:**

- Define **deleteNode** function to delete a node with a given key from the BST.
- If the root is None, return None.
- If the key is less than the root's key, recursively delete from the left subtree.
- If the key is greater than the root's key, recursively delete from the right subtree.
- If the key is found, handle three cases:
    - If the node has no left child, replace it with its right child.
    - If the node has no right child, replace it with its left child.
    - If the node has both left and right children, replace it with the node containing the minimum key value from the right subtree.
- Return the root.

Step 5: **Define the search Function:**

- Define **search** function to search for a key in the BST.
- If the root is None or the key is found at the root, return the root.
- If the key is greater than the root's key, recursively search the right subtree.
- Otherwise, recursively search the left subtree.

Step 6: **Define the inorder Function:**

- Define **inorder** function to perform an inorder traversal of the BST.
- Recursively traverse the left subtree.
- Print the key of the current node.

- Recursively traverse the right subtree.

Step 7: **Test the operations:**

- Initialize an empty root node.
- Insert keys into the BST.
- Print the inorder traversal of the BST.
- Delete a key from the BST.
- Print the inorder traversal of the BST after deletion.
- Search for a key in the BST and print whether it's found or not.

CODING:

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

def insert(root, key):
    if root is None:
        return TreeNode(key)
    if key < root.key:
        root.left = insert(root.left, key)
    elif key > root.key:
        root.right = insert(root.right, key)
    return root


def minValueNode(node):
    current = node
    while current.left is not None:
```

```
        current = current.left
    return current


def deleteNode(root, key):
    if root is None:
        return root

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

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

```
def inorder(root):
    if root:
        inorder(root.left)
        print(root.key, end=" ")
        inorder(root.right)


# Test the operations
root = None
keys = [50, 30, 20, 40, 70, 60, 80]

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

print("Inorder traversal of the BST:")
inorder(root)
print("\n")

print("Deleting 20 from the BST:")
root = deleteNode(root, 20)
print("Inorder traversal after deletion:")
inorder(root)
print("\n")

print("Searching for 30 in the BST:")
result = search(root, 30)
if result:
    print("Key found in the BST")
else:
    print("Key not found in the BST")
```

OUTPUT:

Inorder traversal of the BST:

20 30 40 50 60 70 80

Deleting 20 from the BST:

Inorder traversal after deletion:

30 40 50 60 70 80

Searching for 30 in the BST:

Key found in the BST

RESULT:

The python program to perform Insertion, Deletion and search operations in a binary search tree has been executed and the results are verified successfully.

PROGRAM -6

AIM:

To write a python program to perform insertion and deletion in into an AVL-tree

ALGORITHM:

Step 1: **Define the TreeNode Class:**

- Initialize **TreeNode** class with **key**, **left**, **right**, and **height** attributes.

Step 2: **Define the height Function:**

- Define **height** function to return the height of a node.
- If the node is None, return 0.
- Otherwise, return the height attribute of the node.

Step 3: **Define the get_balance Function:**

- Define **get_balance** function to calculate the balance factor of a node.
- If the node is None, return 0.
- Otherwise, return the difference in heights between the left and right subtrees.

Step 4: **Define the right_rotate Function:**

- Define **right_rotate** function to perform a right rotation on a given node.
- Store the left child and the right child of the given node.
- Update the pointers to perform the rotation.
- Update the heights of the rotated nodes.
- Return the new root after rotation.

Step 5: **Define the left_rotate Function:**

- Define **left_rotate** function to perform a left rotation on a given node.
- Store the left child and the right child of the given node.
- Update the pointers to perform the rotation.
- Update the heights of the rotated nodes.
- Return the new root after rotation.

Step 6: **Define the insert Function:**

- Define **insert** function to insert a new key into the AVL tree.
- If the root is None, create a new node with the key and return it.
- If the key is less than the root's key, recursively insert into the left subtree.
- If the key is greater than the root's key, recursively insert into the right subtree.
- Update the height of the current node.
- Check the balance factor of the current node and perform rotations if necessary.
- Return the root after insertion.

Step 7: **Define the minValueNode Function:**

- Define **minValueNode** function to find the node with the minimum key value in a given subtree.
- Traverse the left subtree until a node with a None left child is found.
- Return the current node.

Step 8: **Define the deleteNode Function:**

- Define **deleteNode** function to delete a node with a given key from the AVL tree.
- Follow the standard procedure for deleting a node in a BST.
- After deletion, update the heights of the ancestors of the deleted node.
- Check the balance factor of the ancestors and perform rotations if necessary.
- Return the root after deletion.

Step 9: **Define the pre_order_traversal Function:**

- Define **pre_order_traversal** function to perform a preorder traversal of the AVL tree.
- Print the key of the current node.
- Recursively traverse the left subtree.
- Recursively traverse the right subtree.

Step 10: **Test the AVL tree operations:**

- Initialize an empty root node.
- Insert keys into the AVL tree.
- Print the preorder traversal of the AVL tree after insertion.
- Delete a key from the AVL tree.
- Print the preorder traversal of the AVL tree after deletion.

CODING:

```
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1

def height(node):
    if node is None:
        return 0
    return node.height

def get_balance(node):
    if node is None:
        return 0
    return height(node.left) - height(node.right)

def right_rotate(y):
    x = y.left
    T2 = x.right

    x.right = y
    y.left = T2

    y.height = 1 + max(height(y.left), height(y.right))
    x.height = 1 + max(height(x.left), height(x.right))

    return x

def left_rotate(x):
    y = x.right
    T2 = y.left
```

```
    y.left = x
    x.right = T2

    x.height = 1 + max(height(x.left), height(x.right))
    y.height = 1 + max(height(y.left), height(y.right))

    return y


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

    root.height = 1 + max(height(root.left), height(root.right))

    balance = get_balance(root)

    if balance > 1 and key < root.left.key:
        return right_rotate(root)

    if balance < -1 and key > root.right.key:
        return left_rotate(root)

    if balance > 1 and key > root.left.key:
        root.left = left_rotate(root.left)
        return right_rotate(root)

    if balance < -1 and key < root.right.key:
```

```
        root.right = right_rotate(root.right)
        return left_rotate(root)

    return root

def minValueNode(node):
    current = node
    while current.left is not None:
        current = current.left
    return current

def deleteNode(root, key):
    if root is None:
        return root

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

    if root is None:
```

```
        return root

    root.height = 1 + max(height(root.left), height(root.right))

    balance = get_balance(root)

    if balance > 1 and get_balance(root.left) >= 0:
        return right_rotate(root)

    if balance < -1 and get_balance(root.right) <= 0:
        return left_rotate(root)

    if balance > 1 and get_balance(root.left) < 0:
        root.left = left_rotate(root.left)
        return right_rotate(root)

    if balance < -1 and get_balance(root.right) > 0:
        root.right = right_rotate(root.right)
        return left_rotate(root)

    return root

def pre_order_traversal(root):
    if root:
        print(root.key, end=" ")
        pre_order_traversal(root.left)
        pre_order_traversal(root.right)

# Test the AVL tree operations
root = None
keys = [9, 5, 10, 0, 6, 11, -1, 1, 2]
for key in keys:
    root = insert(root, key)
```

```
print("AVL tree after insertion:")
pre_order_traversal(root)
print("\n")

print("AVL tree after deleting 10:")
root = deleteNode(root, 10)
pre_order_traversal(root)
```

OUTPUT:

AVL tree after insertion:

9 1 0 -1 5 2 6 10 11

AVL tree after deleting 10:

1 0 -1 9 5 2 6 11 >

RESULT:

The python program to perform insertion and deletion in into an AVL-tree has been executed and the results are verified successfully.

PROGRAM -7

AIM:

To write a python program for the implementation of BFS and DFS for a given graph.

ALGORITHM:

Step 1: **Import Required Modules:**

- Import the defaultdict class from the collections module.

Step 2: **Define the Graph Class:**

- Initialize the Graph class with a graph attribute, which is a defaultdict of lists.

Step 3: **Define the add_edge Method:**

- Define the add_edge method to add an edge between two vertices.
- Append the second vertex to the list corresponding to the first vertex in the graph.

Step 4: **Define the bfs Method:**

- Define the bfs method to perform breadth-first search traversal of the graph.
- Initialize a list called visited to keep track of visited vertices.
- Initialize a queue to store vertices to be visited.
- Append the start vertex to the queue and mark it as visited.
- While the queue is not empty, pop a vertex from the queue, print it, and mark its neighbors as visited by adding them to the queue.

Step 5: **Define the dfs_util Method:**

- Define the dfs_util method to perform depth-first search traversal of the graph recursively.
- Mark the current vertex as visited and print it.
- Recursively call the dfs_util method for each unvisited neighbor of the current vertex.

Step 6: **Define the dfs Method:**

- Define the dfs method to initiate depth-first search traversal of the graph.
- Initialize a list called visited to keep track of visited vertices.
- Call the dfs_util method for the start vertex.

Step 7: **Test the BFS and DFS Algorithms:**

- Create an instance of the Graph class.
- Add edges to the graph.
- Print the BFS traversal starting from a specific vertex.
- Print the DFS traversal starting from a specific vertex.

CODING:

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

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def bfs(self, start):
        visited = [False] * len(self.graph)
        queue = []
        queue.append(start)
        visited[start] = True

        while queue:
            vertex = queue.pop(0)
            print(vertex, end=" ")

            for neighbor in self.graph[vertex]:
                if not visited[neighbor]:
                    queue.append(neighbor)
                    visited[neighbor] = True

    def dfs_util(self, vertex, visited):
        visited[vertex] = True
        print(vertex, end=" ")

        for neighbor in self.graph[vertex]:
            if not visited[neighbor]:
                self.dfs_util(neighbor, visited)

    def dfs(self, start):
```

```
        visited = [False] * len(self.graph)
        self.dfs_util(start, visited)


# Test the BFS and DFS algorithms
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 0)
g.add_edge(2, 3)
g.add_edge(3, 3)

print("BFS starting from vertex 2:")
g.bfs(2)
print("\n")

print("DFS starting from vertex 2:")
g.dfs(2)
```

OUTPUT:

BFS starting from vertex 2:

2 0 3 1

DFS starting from vertex 2:

2 0 1 3 >

RESULT:

The python program for the implementation of BFS and DFS for a given graph has been executed and the results are verified successfully.

.

PROGRAM -8

AIM:

To write a python programs for implementing Linear search and Binary search.

ALGORITHM:

Linear search

Step 1: **Define the linear_search Function:**

- Define the linear_search function to search for a target element in an array.
- Iterate over each element in the array using a loop.
- If the current element matches the target, return its index.
- If the target is not found after iterating through the entire array, return -1.

Step 2: **Test the Linear Search:**

- Initialize an array with elements.
- Define the target element to be searched.
- Call the linear_search function with the array and target as arguments.
- If the result is not -1, print the index at which the target is found.
- If the result is -1, print a message indicating that the target is not found.

- **Binary search.**

Step 1: **Define the binary_search Function:**

- Define the binary_search function to search for a target element in a sorted array using binary search.

- Initialize left as 0 and right as the index of the last element in the array.
- While the left index is less than or equal to the right index:
    - Calculate the mid index as the average of left and right.
    - If the element at the mid index equals the target, return the mid index.
    - If the element at the mid index is less than the target, update the left index to mid + 1.
    - If the element at the mid index is greater than the target, update the right index to mid - 1.
- If the target is not found after the loop, return -1.

Step 2: **Test the Binary Search:**

- Initialize a sorted array arr.
- Define the target element to be searched.
- Call the binary_search function with the array and target as arguments.
- If the result is not -1, print the index at which the target is found.
- If the result is -1, print a message indicating that the target is not found.

CODING:

Linear Search:

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

# Test the linear search
arr = [3, 5, 7, 2, 8, 4]
target = 7
result = linear_search(arr, target)
if result != -1:
    print(f"Element {target} found at index {result}")
else:
    print(f"Element {target} not found")
```

output:

Element 7 found at index 2

Binary Search (for sorted arrays):

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

# Test the binary search
arr = [2, 4, 6, 8, 10, 12, 14, 16]
target = 10
result = binary_search(arr, target)
if result != -1:
    print(f"Element {target} found at index {result}")
else:
    print(f"Element {target} not found")
```

output:

Element 10 found at index 4

RESULT:

The python program for implementing Linear search and Binary search has been executed and the results are verified successfully.

.PROGRAM -9

AIM:

To write a programs for implementing Bubble sort , Selection sort, Insertion sort and Radix sort.

ALGORITHM:

- **Bubble sort**

Step 1: Define the bubble_sort Function:

- Define the bubble_sort function to sort an array using the bubble sort algorithm.
- Initialize n as the length of the array.
- Iterate over the array using two nested loops:
  - The outer loop runs from 0 to n-1 and represents the pass number.
  - The inner loop runs from 0 to n-i-1 and represents the comparisons within each pass.
  - Compare adjacent elements and swap them if they are in the wrong order.
- After completing the inner loop for each pass, the largest element will be placed at its correct position.
- Repeat this process for all elements by decrementing the range of the inner loop in each iteration.

Step 2: Test the Bubble Sort:

- Initialize an array arr with elements.
- Call the bubble_sort function with the array arr as an argument to sort it.
- Print the sorted array.

- Selection sort

Step 1: Define the selection_sort Function:

- Define the selection_sort function to sort an array using the selection sort algorithm.
- Initialize n as the length of the array.
- Iterate over the array using a loop from 0 to n-1:
    - Set min_index as the current index i.
    - Iterate over the unsorted portion of the array using a nested loop from i+1 to n:
        - If the element at index j is smaller than the element at min_index, update min_index to j.
    - Swap the element at index i with the element at min_index, effectively placing the smallest unsorted element in its correct position.

Step 2: Test the Selection Sort:

- Initialize an array arr with elements.
- Call the selection_sort function with the array arr as an argument to sort it.
- Print the sorted array.

Insertion sort

Here's an algorithm to represent the given code for insertion sort:

Step 1: **Define the insertion_sort Function:**

- Define the insertion_sort function to sort an array using the insertion sort algorithm.
- Initialize n as the length of the array.
- Iterate over the array using a loop from the second element (index 1) to the last element (index n-1):
    - Set key as the current element at index i.

- Set j as the index preceding i.
- While j is greater than or equal to 0 and the element at index j is greater than key:
    - Move the element at index j one position ahead to make space for key.
    - Decrement j to continue checking elements to the left.
- Place key in its correct sorted position after the while loop.

Step 2: **Test the Insertion Sort:**

- Initialize an array arr with elements.
- Call the insertion_sort function with the array arr as an argument to sort it.
- Print the sorted array.

- **Radix sort.**

Step 1: **Define the counting_sort Function:**

- Define the counting_sort function to sort an array using the counting sort algorithm.
- Initialize an output array of size n with all elements as 0.
- Initialize a count array of size 10 with all elements as 0.
- Iterate over the array arr:
    - Calculate the index based on the current element and the exp value.
    - Increment the count at the calculated index.

- Update the count array to store the cumulative count of elements.
- Iterate over the array arr in reverse order:
    - Calculate the index based on the current element and the exp value.
    - Place the element at the correct position in the output array based on the count array.
    - Decrement the count at the calculated index.
- Update the original array arr with the sorted output array.

Step 2: **Define the radix_sort Function:**

- Define the radix_sort function to sort an array using the radix sort algorithm.
- Find the maximum value in the array to determine the number of digits.
- Initialize exp as 1.
- While the maximum value divided by exp is greater than 0:
    - Call the counting_sort function with the array and exp as arguments.
    - Multiply exp by 10 for the next iteration.

Step 3: **Test the Radix Sort:**

- Initialize an array arr with elements.
- Call the radix_sort function with the array arr as an argument to sort it.
- Print the sorted array.

CODING:

- **Bubble sort**

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

# Test the Bubble Sort
arr = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(arr)
print("Sorted array using Bubble Sort:", arr)
```

OUTPUT:

Sorted array using Bubble Sort: [11, 12, 22, 25, 34, 64, 90]

- **Selection sort**

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]

# Test the Selection Sort
arr = [64, 34, 25, 12, 22, 11, 90]
selection_sort(arr)
print("Sorted array using Selection Sort:", arr)
```

OUTPUT:

Sorted array using Selection Sort: [11, 12, 22, 25, 34, 64, 90]

- **Insertion sort:**

```
def insertion_sort(arr):
    n = len(arr)
    for i in range(1, n):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

# Test the Insertion Sort
arr = [64, 34, 25, 12, 22, 11, 90]
insertion_sort(arr)
print("Sorted array using Insertion Sort:", arr)
```

output:

Sorted array using Insertion Sort: [11, 12, 22, 25, 34, 64, 90]

- **Radix Sort:**

```
def counting_sort(arr, exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10

    for i in range(n):
        index = arr[i] // exp
        count[index % 10] += 1

    for i in range(1, 10):
        count[i] += count[i - 1]

    i = n - 1
    while i >= 0:
        index = arr[i] // exp
        output[count[index % 10] - 1] = arr[i]
        count[index % 10] -= 1
        i -= 1

    i = 0
    for i in range(n):
        arr[i] = output[i]

def radix_sort(arr):
    max_value = max(arr)
    exp = 1
    while max_value // exp > 0:
        counting_sort(arr, exp)
        exp *= 10
```

```
# Test the Radix Sort
arr = [170, 45, 75, 90, 802, 24, 2, 66]
radix_sort(arr)
print("Sorted array using Radix Sort:", arr)
```

output:

Sorted array using Radix Sort: [2, 24, 45, 66, 75, 90, 170, 802]

RESULT:

The python program for implementing Bubble sort , Selection sort, Insertion sort and Radix sort has been executed and the results are verified successfully.