DSA LAB MANUAL PROGRAMS

1. PROGRAM TO IMPLEMENT STACK ADT:

```
class Stack:
    def __init__(self, max_size):
        self.stack = []
        self.max_size = max_size

    def push(self, data):
        if len(self.stack) == self.max_size:
            print("Stack Overflow")
        else:
            self.stack.append(data)

    def pop(self):
        if self.stack:
            print(f"Popped: {self.stack.pop()}")
        else:
            print("Stack Underflow")

    def peek(self):
        print(self.stack[-1] if self.stack else "Stack is empty")

    def display(self):
        print(self.stack if self.stack else "Stack is empty")

if __name__ == "__main__":
    stack = Stack(int(input("Enter max size of the stack: ")))
    while True:
        choice = int(input("\n 1. Push\n 2. Pop\n 3. Peek\n 4. Display\n 5. Exit\nEnter choice: "))
        if choice == 1:
            stack.push(int(input("Enter data: ")))
        elif choice == 2:
            stack.pop()
        elif choice == 3:
            stack.peek()
        elif choice == 4:
            stack.display()
        elif choice == 5:
            print("Exiting....")
            break
        else:
            print("Invalid choice")
```

OUTPUT:

```
Enter max size of the stack: 3

 1. Push
 2. Pop
 3. Peek
 4. Display
 5. Exit
Enter choice: 1
Enter data: 3

 1. Push
 2. Pop
 3. Peek
 4. Display
 5. Exit
Enter choice: 1
Enter data: 4

 1. Push
 2. Pop
 3. Peek
 4. Display
 5. Exit
Enter choice: 1
Enter data: 5

 1. Push
 2. Pop
 3. Peek
 4. Display
 5. Exit
Enter choice: 2
Popped: 5

 1. Push
 2. Pop
 3. Peek
 4. Display
 5. Exit
Enter choice: 3
4

 1. Push
 2. Pop
 3. Peek
 4. Display
 5. Exit
Enter choice: 4
[3, 4]

 1. Push
 2. Pop
 3. Peek
 4. Display
 5. Exit
Enter choice: 5
Exiting....
```

2. PROGRAM TO IMPLEMENT QUEUE ADT:

```
class Queue:
  def __init__(self, max_size):
    self.queue = []
    self.max_size = max_size

  def enqueue(self, data):
    if len(self.queue) == self.max_size:
      print("Queue Overflow")
    else:
      self.queue.append(data)

  def dequeue(self):
    if self.queue:
      print(f"Dequeued: {self.queue.pop(0)}")
    else:
      print("Queue Underflow")

  def front(self):
    print(self.queue[0] if self.queue else "Queue is empty")

  def display(self):
    print(self.queue if self.queue else "Queue is empty")

if __name__ == "__main__":
  queue = Queue(int(input("Enter max size of the queue: ")))
  while True:
    choice = int(input("\n1. Enqueue\n2. Dequeue\n3. Front\n4.
Display\n5. Exit\nEnter choice: "))
    if choice == 1:
      queue.enqueue(int(input("Enter data: ")))
    elif choice == 2:
      queue.dequeue()
    elif choice == 3:
      queue.front()
    elif choice == 4:
      queue.display()
    elif choice == 5:
      print("Exiting……")
      break
    else:
      print("Invalid choice")
```

OUTPUT:

```
Enter max size of the queue: 3

1. Enqueue
2. Dequeue
3. Front
4. Display
5. Exit
Enter choice: 1
Enter data: 5

1. Enqueue
2. Dequeue
3. Front
4. Display
5. Exit
Enter choice: 1
Enter data: 8

1. Enqueue
2. Dequeue
3. Front
4. Display
5. Exit
Enter choice: 2
Dequeued: 5

1. Enqueue
2. Dequeue
3. Front
4. Display
5. Exit
Enter choice: 3
8

1. Enqueue
2. Dequeue
3. Front
4. Display
5. Exit
Enter choice: 4
[8]

1. Enqueue
2. Dequeue
3. Front
4. Display
5. Exit
Enter choice: 5
Exiting……
```

3. PROGRAM TO IMPLEMENT SINGLY LINKED LIST:	OUTPUT:

<pre>class Node:
  def __init__(self, data):
    self.data = data
    self.next = None
 	
class SinglyLinkedList:
  def __init__(self):
    self.head = None
 	
  def insert(self, data):
    new_node = Node(data)
    new_node.next = self.head
    self.head = new_node
 	
  def display(self):
    temp = self.head
    print("Nodes of singly linked list:")
    while temp:
      print(temp.data, end="\n" if temp.next else " -> None\n")
      temp = temp.next
 	
sll = SinglyLinkedList()
sll.insert(1)
sll.insert(2)
sll.insert(3)
sll.insert(4)
sll.display()</pre>
	Nodes of singly linked list:
	4
	3
	2
	1 -> None

4. PROGRAM TO IMPLEMENT DOUBLY LINKED LIST:

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

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

   def insert(self, data):
      new_node = Node(data)
      new_node.next = self.head
      if self.head:
         self.head.prev = new_node
      self.head = new_node

   def display(self):
      temp = self.head
      print("Nodes of doubly linked list:")
      while temp:
         print(temp.data, end=" <-> " if temp.next else " ->
None\n")
         temp = temp.next


dll = DoublyLinkedList()
dll.insert(1)
dll.insert(2)
dll.insert(3)
dll.insert(4)
dll.display()
```

OUTPUT:

```
Nodes of doubly linked list:
4 <-> 3 <-> 2 <-> 1 -> None
```

5. PROGRAM TO IMPLEMENT CIRCULAR LINKED LIST:

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

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

    def insert(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            new_node.next = new_node
        else:
            temp = self.head
            while temp.next != self.head:
                temp = temp.next
            temp.next = new_node
            new_node.next = self.head

    def display(self):
        if not self.head:
            print("List is empty")
            return
        temp = self.head
        print("Circular Linked List:\n ", end="")
        while True:
            print(temp.data, end=" -> ")
            temp = temp.next
            if temp == self.head:
                print(f"{self.head.data} (back to head)")
                break


cll = CircularLinkedList()
for i in [1, 2, 3, 4]:
    cll.insert(i)
cll.display()
```

OUTPUT:

```
Circular Linked List:
 1 -> 2 -> 3 -> 4 -> 1 (back to head)
```

6. PROGRAM TO EVALUATE POSTFIX EXPRESSION:	OUTPUT:

def evaluate_postfix(expression):
  stack = []
  for char in expression:
    if char.isdigit():
      stack.append(int(char))
    else:
      b = stack.pop()
      a = stack.pop()
      if char == '+':
        stack.append(a + b)
      elif char == '-':
        stack.append(a - b)
      elif char == '*':
        stack.append(a * b)
      elif char == '/':
        stack.append(a // b)
  return stack.pop()
 	
postfix_expression = "562+*31-/"
result = evaluate_postfix(postfix_expression)
print(f"Result of postfix expression '{postfix_expression}' is {result}")
	Result of postfix expression '562+*31-/' is 20

7. PROGRAM TO DESIGN AND IMPLEMENT THE BINARY TREE:

```
class Node:
   def __init__(self, val):
      self.val = val
      self.left = self.right = None

class Tree:
   def __init__(self):
      self.root = None

   def add(self, val):
      if not self.root:
         self.root = Node(val)
      else:
         self._add_node(self.root, val)

   def _add_node(self, node, val):
      if node.left is None:
         node.left = Node(val)
      elif node.right is None:
         node.right = Node(val)
      else:
         self._add_node(node.left, val) if node.left else
self._add_node(node.right, val)

   def show(self, node, lvl=0):
      if node:
         self.show(node.right, lvl + 1)
         print(' ' * 4 * lvl + '->', node.val)
         self.show(node.left, lvl + 1)


t = Tree()
for v in [7, 5, 4, 3, 2, 1]:
   t.add(v)

print("Constructed Tree:")
t.show(t.root)
```

OUTPUT:

```
Constructed Tree:
   -> 4
-> 7
      -> 2
   -> 5
      -> 3
         -> 1
```

8. PROGRAM TO DESIGN AND IMPLEMENT BINARY SEARCH TREE:

```
class Node:
    def __init__(self, val):
        self.val = val
        self.left = self.right = None

class BST:
    def __init__(self):
        self.root = None

    def add(self, val):
        if not self.root:
            self.root = Node(val)
        else:
            self._add_node(self.root, val)

    def _add_node(self, node, val):
        if val < node.val:
            if node.left:
                self._add_node(node.left, val)
            else:
                node.left = Node(val)
        else:
            if node.right:
                self._add_node(node.right, val)
            else:
                node.right = Node(val)

    def show(self, node, lvl=0):
        if node:
            self.show(node.right, lvl + 1)
            print(' ' * 4 * lvl + '->', node.val)
            self.show(node.left, lvl + 1)


bst = BST()
for v in [7, 5, 4, 3, 2, 1]:
    bst.add(v)

print("Constructed BST:")
bst.show(bst.root)
```

OUTPUT:

```
Constructed BST:
-> 7
    -> 5
        -> 4
            -> 3
                -> 2
                    -> 1
```

9. PROGRAM TO DESIGN AND IMPLEMENT DIJKSTRA'S ALGORITHM:

```
import heapq

def dijkstra(graph, start):
    dist = {node: float('inf') for node in graph}
    dist[start] = 0
    prev = {node: None for node in graph}
    pq = [(0, start)]

    while pq:
        curr_dist, node = heapq.heappop(pq)
        if curr_dist > dist[node]:
            continue
        for neighbor, weight in graph[node]:
            new_dist = curr_dist + weight
            if new_dist < dist[neighbor]:
                dist[neighbor] = new_dist
                prev[neighbor] = node
                heapq.heappush(pq, (new_dist, neighbor))

    def path(node):
        p = []
        while node:
            p.insert(0, node)
            node = prev[node]
        return p

    return {node: (dist[node], path(node)) for node in graph}

graph = {
    'A': [('B', 1), ('C', 4)],
    'B': [('C', 2), ('D', 5)],
    'C': [('D', 1)],
    'D': []
}

result = dijkstra(graph, 'A')

for node, (distance, p) in result.items():
    print(f"{node}: Distance = {distance}, Path = {' -> '.join(p)}")
```

OUTPUT:

```
A: Distance = 0, Path = A
B: Distance = 1, Path = A -> B
C: Distance = 3, Path = A -> B -> C
D: Distance = 4, Path = A -> B -> C -> D
```

## 10. PROGRAM TO SOLVE MERGE SORT.:	## OUTPUT:

```
print('Merge Sort: ')
A = []
n = int(input('Enter Number of Elements in the List: '))
for i in range(0, n):
   x = int(input('Enter the Element %d :' %(i+1)))
   A.append(x)
print('Original List: ')
print(A)

def Merge(left, right, A):
   i = j = k = 0
   while(i < len(left) and j < len(right)):
      if(left[i] < right[j]):
         A[k] = left[i]
         i = i + 1
      else:
         A[k] = right[j]
         j += 1
      k += 1

   while(i < len(left)):
      A[k] = left[i]
      i += 1
      k += 1

   while(j < len(right)):
      A[k] = right[j]
      j += 1
      k += 1
   print('Merging', A)

def MergeSort(A):
   print('Splitting', A)
   n = len(A)
   if(n > 1):
      mid = n // 2
      left = A[:mid]
      right = A[mid:]
      MergeSort(left)
      MergeSort(right)
      Merge(left, right, A)

MergeSort(A)
print("Sorted List:\n")
print(A)
```

OUTPUT:

```
Merge Sort:
Enter Number of Elements in the List: 3
Enter the Element 1 :5
Enter the Element 2 :2
Enter the Element 3 :6
Original List:
[5, 2, 6]
Splitting [5, 2, 6]
Splitting [5]
Splitting [2, 6]
Splitting [2]
Splitting [6]
Merging [2, 6]
Merging [2, 5, 6]
Sorted List:

[2, 5, 6]
```

11. PROGRAM TO SOLVE INSERTION SORT:	OUTPUT:

<pre>def insertionSort(arr):
   for i in range(1, len(arr)):
      key = arr[i]
      j = i - 1
      while j >= 0 and key < arr[j]:
         arr[j + 1] = arr[j]
         j -= 1
      arr[j + 1] = key
 	
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
 	
lst = []
print("Sorted array is : ")
for i in range(len(arr)):
   lst.append(arr[i])
print(lst)</pre>
	Sorted array is :
	[5, 6, 11, 12, 13]

12.A)PROGRAM TO IMPLEMENT LINEAR SEARCH:	**OUTPUT:**

<pre>n = int(input("Enter the number of elements:"))
a = []
 	
for i in range(n):
   x = int(input("Enter the elements:"))
   a.append(x)
 	
def linear(e):
   for i in range(n):
      if a[i] == e:
         print("Element found")
         break
   else:
      print("Element not found")
 	
e = int(input("Enter the element to search:"))
linear(e)</pre>
	Enter the number of elements:4
	Enter the elements:1
	Enter the elements:3
	Enter the elements:4
	Enter the elements:6
	Enter the element to search:4
	Element found

12.B)PROGRAM TO IMPLEMENT BINARY SEARCH:	OUTPUT:

<pre>def binary_search(a, target):
    left = 0
    right = len(a) - 1
    found = False
 	
    while left <= right:
        mid = (right + left) // 2
        if a[mid] == target:
            found = True
            break
        elif a[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
 	
    if found:
        print("Element found at index ", mid)
    else:
        print("Element not found")
 	
a = [1, 3, 5, 7, 9, 11, 13, 15]
target = int(input("Enter the target number: "))
binary_search(a, target)</pre>
	Enter the target number: 11
	Element found at index 5

ALGORITHMS

1. STACK ADT

- Initialize an empty stack.
- Push element: Add the element to the stack.
- Pop element: Remove and return the top element of the stack.
- Peek: Return the top element without removing it.
- Check if the stack is empty.
- Display all elements in the stack.

2. QUEUE ADT

- Initialize an empty queue.
- Enqueue element: Add the element to the rear of the queue.
- Dequeue element: Remove and return the front element of the queue.
- Peek: Return the front element without removing it.
- Check if the queue is empty.
- Display all elements in the queue.

3. SINGLY LINKED LIST

- Initialize an empty linked list.
- Insert element: Add a node at the end or beginning.
- Delete element: Remove a node by value.
- Search element: Traverse and find a node by value.
- Display elements: Traverse and print all node values.
- Check if the list is empty.

4. DOUBLY LINKED LIST

- Initialize an empty doubly linked list.
- Insert element: Add a node at the beginning, middle, or end.
- Delete element: Remove a node by value.
- Search element: Traverse and find a node.
- Traverse: Move forward or backward through nodes.
- Display all elements from both directions.

5. CIRCULAR LINKED LIST

- Initialize an empty circular linked list.
- Insert element: Add a node and make it point to the head.
- Delete element: Remove a node by value.
- Search element: Traverse circularly to find a node.
- Display elements: Traverse circularly and print all nodes.
- Check if the list has only one node.

6. EVALUATE POSTFIX EXPRESSION

- Initialize an empty stack.
- For each token in the expression:
  - If it's an operand, push it to the stack.
  - If it's an operator, pop two operands from the stack, apply the operator, and push the result.
- After processing, the stack contains the final result.
- Return the result.

7. BINARY TREE

- Initialize an empty tree with a root node.
- Insert element: Add a node based on binary tree rules (left or right).
- Search element: Recursively search for a node by value.
- Traverse tree: Perform in-order, pre-order, or post-order traversal.
- Delete element: Remove a node while maintaining binary tree properties.
- Display all nodes in a specific order.

8. BINARY SEARCH TREE

- Initialize an empty binary search tree (BST).
- Insert element: Add nodes recursively by comparing values.
- Search element: Find a node by recursively comparing values.
- Delete element: Remove a node and ensure the BST properties.
- Traverse tree: Perform in-order traversal to get sorted elements.
- Display elements in sorted order.

9. DIJKSTRA'S ALGORITHM

- Initialize a set of visited nodes and a distance map.
- Set the initial node distance to 0, and others to infinity.
- For the current node, check its neighbors and update their distances.
- Mark the current node as visited and move to the next closest unvisited node.
- Repeat until all nodes are visited.
- Return the shortest distance to each node.

10. MERGE SORT

- Divide the list into two halves recursively.
- Recursively sort each half.
- Merge the two sorted halves into a single sorted list.
- Repeat the process until all sublists are merged.
- Return the sorted list.

11. INSERTION SORT

- Traverse the list from the second element.
- For each element, insert it into the correct position in the sorted portion of the list.
- Shift elements to the right to make room for the current element.
- Continue until all elements are sorted.
- Return the sorted list.

12. SEARCH ALGORITHMS

- **LINEAR SEARCH:**
  - Traverse through each element in the list.
  - Compare each element with the target.
  - If found, return the index; otherwise, return -1.
- **BINARY SEARCH:**
  - Start with the middle element.
  - If the target is smaller, search the left half; if larger, search the right half.
  - Repeat until the element is found or the sublist is empty.
  - Return the index or -1 if not found.