DS - Onscreen Test

2) Singly Linked List

Aim:
Implementing Singly Linked List (SLL) supporting following operations using menu driven program.
1. Insert at the Begin
2. Insert after the specified existing node
3. Delete before the specified existing node
4. Display all elements in tabular form.

Algorithm :
Program should implement the specified operations strictly in the following manner. Also implement a support method isempty() and make use of it at appropriate places.
1. **createSLL()** – This void function should create a START/HEAD pointer with NULL value as empty SLL.
2. **insertBegin(typedef newelement)** – This void function should take a newelement as an argument to be inserted on an existing SLL and insert it before the element pointed by the START/HEAD pointer.
3. **insertAfter(typedef newelement, typedef existingelement)** – This void function should take two arguments. The function should search for an existingelement on non-empty SLL and insert newelement after this element.
4. **typedef deleteBefore(typedef existingelement)** – This function should search for the existing element passed to the function in the non-empty SLL, delete the node siting before it and return the deleted element.
5. **display( )** – This is a void function which should go through non- empty SLL starting from START/HEAD pointer and display each element of the SLL till the end.

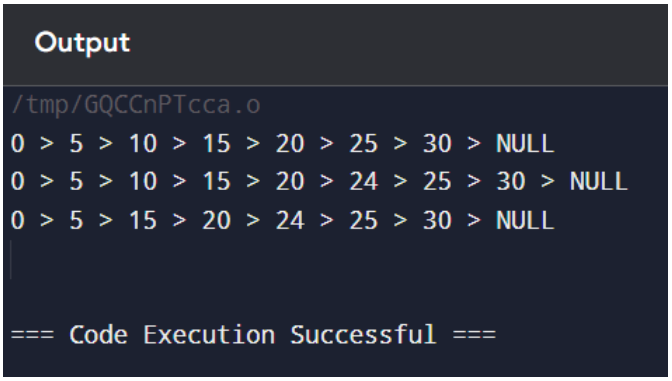
Code :

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int data;
    struct Node* next;
} Node;

Node* START = NULL;

int isempty() {
    return START == NULL;
```

```
}

void createSLL() {
   START = NULL;
}

void insertBegin(int newele) {
   Node* newnode = (Node*)malloc(sizeof(Node));
   newnode->data = newele;
   newnode->next = START;
   START = newnode;
}

void insertAfter(int newele, int existingele) {
   Node* newnode = (Node*)malloc(sizeof(Node));
   newnode->data = newele;
   Node* now = START;
   while (now != NULL && now->data != existingele) {
      now = now->next;
   }
   if (now != NULL) {
      newnode->next = now->next;
      now->next = newnode;
   }
}

int deleteBefore(int existingele) {
   if (isempty()) {
      printf("List is empty. Cannot delete before %d.\n", existingele);
      return -1;
   }
   if (START->next == NULL || START->data == existingele) {
      printf("No element exists before %d to delete.\n", existingele);
      return -1;
   }
   Node* prev = NULL;
   Node* now = START;
   while (now->next != NULL && now->next->data != existingele) {
      prev = now;
      now = now->next;
   }
   if (now->next == NULL) {
      printf("Element %d not found in the list.\n", existingele);
      return -1;
   }
   int deletedData;
   if (prev == NULL) {
      Node* temp = START;
```

```
        START = START->next;
        deletedData = temp->data;
        free(temp);
    } else {
        prev->next = now->next;
        deletedData = now->data;
        free(now);
    }
    return deletedData;
}

void display() {
    if (isempty()) {
        printf("List is empty.\n");
        return;
    }
    Node* now = START;
    while (now != NULL) {
        printf("%d > ", now->data);
        now = now->next;
    }
    printf("NULL\n");
}

int main() {
    createSLL();
    insertBegin(30);
    insertBegin(25);
    insertBegin(20);
    insertBegin(15);
    insertBegin(10);
    insertBegin(5);
    insertBegin(0);
    display();
    insertAfter(24, 20);
    display();
    deleteBefore(15);
    display();
    return 0;
}
```

Output :


Output
/tmp/GQCCnPTcca.o
0 > 5 > 10 > 15 > 20 > 25 > 30 > NULL
0 > 5 > 10 > 15 > 20 > 24 > 25 > 30 > NULL
0 > 5 > 15 > 20 > 24 > 25 > 30 > NULL
=== Code Execution Successful ===

3) STACK

a) Aim:
WAP to create a stack using SLL by implementing following operations
1. Create stack,
2. Insert an element,
3. Delete an element,
4. Display top element.

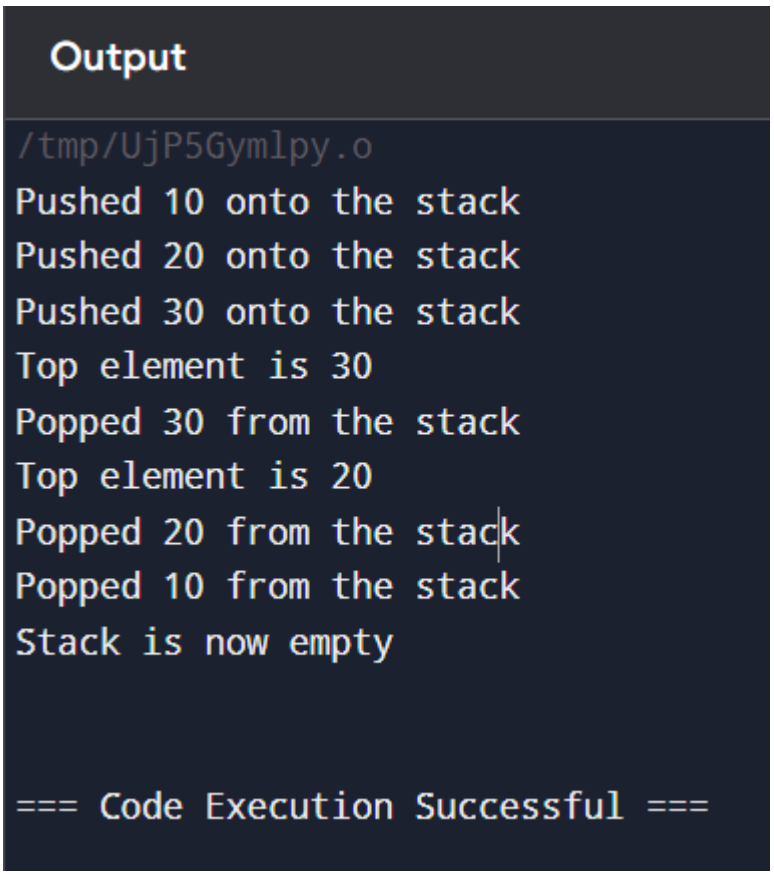
Algorithm :
Implement “dynamic stack” with following functions
1. **void createStack(void)** : Create and initializes the top to NULL, creating the empty stack.
2. **void push(char x)** : Creates a node with value 'x' and inserts on top of the stack.
3. **char pop(void)** : Deletes a node from top of the stack and returns the deleted value.
4. **boolean isEmpty(void)** : Returns “1” for stack empty; “0” otherwise.
5. **char peek(void)** : Return current stack top element.

Code :

```
#include<stdio.h>
#include<stdlib.h>

//declaring Node and stack structure
struct Node {
    int data;
    struct Node* next;
};

struct Stack {
    struct Node* top;
};

//creating the stack
struct Stack* createStack() {
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->top = NULL;
    return stack;
}

//push
void push(struct Stack* stack, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        return;
    }
    newNode->data = data;
```

```
    newNode->next = stack->top;
    stack->top = newNode;
    printf("Pushed %d onto the stack \n", data);
}

//element delete
int pop(struct Stack* stack) {
    if (stack->top == NULL) {
        printf("Stack is now empty\n");
        return -1;
    }
    struct Node* temp = stack->top;
    int poppedData = temp->data;
    stack->top = stack->top->next;
    free(temp);
    printf("Popped %d from the stack\n", poppedData);
    return poppedData;
}

//Top element
void displayTop(struct Stack* stack) {
    if (stack->top == NULL) {
        printf("Stack is empty\n");
    } else {
        printf("Top element is %d \n", stack->top->data);
    }
}

//main function
int main() {
    struct Stack* stack = createStack();
    push(stack, 10);
    push(stack, 20);
    push(stack, 30);
    displayTop(stack);
    pop(stack);
    displayTop(stack);
    pop(stack);
    pop(stack);
    pop(stack);
    return 0;
}
```

Output :



b) Aim :
WAP to convert a given infix expression into equivalent postfix form using stack implemented in part (a).

Code :

```
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>

//declaring Node and stack structure
struct Node {
    char data;
    struct Node* next;
};

struct Stack {
    struct Node* top;
};

//creating the stack
```

```
struct Stack* createStack() {
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->top = NULL;
    return stack;
}

//push
void push(struct Stack* stack, char data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        return;
    }
    newNode->data = data;
    newNode->next = stack->top;
    stack->top = newNode;
    printf("Pushed %c onto the stack \n", data);
}

//element delete
char pop(struct Stack* stack) {
    if (stack->top == NULL) {
        printf("Stack is now empty\n");
        return -1;
    }
    struct Node* temp = stack->top;
    char poppedData = temp->data;
    stack->top = stack->top->next;
    free(temp);
    printf("Popped %c from the stack\n", poppedData);
    return poppedData;
}

//Top element
char peek(struct Stack* stack) {
    if (stack->top == NULL) {
        printf("Stack is empty\n");
        return -1;
    }
    else {
        return stack->top->data;
    }
}

//check if operator
int isOperator(char ch) {
    return ch == '+' || ch == '-' || ch == '*' || ch == '/';
}
```

```
//check precedence
int precedence(char ch) {
    switch (ch) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
    }
    return 0;
}

//convert infix to postfix
void infixToPostfix(char* exp) {
    struct Stack* stack = createStack();
    int i, k;
    for (i = 0, k = -1; exp[i]; ++i) {
        if (isalnum(exp[i]))
            exp[++k] = exp[i];
        else if (exp[i] == '(')
            push(stack, exp[i]);
        else if (exp[i] == ')') {
            while (stack->top && peek(stack) != '(')
                exp[++k] = pop(stack);
            pop(stack);
        }
        else if (isOperator(exp[i])) {
            while (stack->top && precedence(peek(stack)) >= precedence(exp[i]))
                exp[++k] = pop(stack);
            push(stack, exp[i]);
        }
    }
    while (stack->top)
        exp[++k] = pop(stack);
    exp[++k] = '\0';
    printf("Postfix expression: %s\n", exp);
}

//main function
int main() {
    char exp[] = "a+b*(c^d-e)^(f+g*h)-i";
    printf("Infix expression: %s\n", exp);
    infixToPostfix(exp);
    return 0;
}
```

Output :

```
Output
/tmp/P8iES8eBXV.o
Infix expression: a+b*(c^d-e)^(f+g*h)-i
Pushed + onto the stack
Pushed * onto the stack
Pushed ( onto the stack
Pushed - onto the stack
Popped - from the stack
Popped ( from the stack
Pushed ( onto the stack
Pushed + onto the stack
Pushed * onto the stack
Popped * from the stack
Popped + from the stack
Popped ( from the stack
Popped * from the stack
Popped + from the stack
Pushed - onto the stack
Popped - from the stack
Postfix expression: abcde-fgh*+*+i-


=== Code Execution Successful ===
```

4) Static Priority Queue

Aim: Write a menu driven program to implement a static priority queue supporting following operations.
1. Create empty queue,
2. Insert an element on the queue,
3. Delete an element from the queue,
4. Display front, rear element or
5. Display all elements of the queue.

Algorithm :
1. Create Empty Queue:
- Initialize the `front` and `rear` of the queue to `-1`, indicating the queue is empty.

2. Insert Element:
- Check if the queue is full by comparing the rear with `MAX-1`.
- If the queue is empty, insert the element at the front (0th index).- Otherwise, insert the element based on its priority. If an element with a higher priority already exists, shift elements to the right and insert the new element in the correct position.

3. Delete Element:
- Check if the queue is empty.
- Remove the element at the front of the queue and adjust the `front` pointer.
- If after deletion the queue becomes empty, reset `front` and `rear` to `-1`.

4. Display Queue:
- If the queue is empty, print "Queue is empty".
- Otherwise, display the elements along with their priorities from `front` to `rear`.

Code :

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 4

struct PriorityQueue {
    int data[MAX];
    int front, rear;
};

void createQueue(struct PriorityQueue* pq) {
    pq->front = pq->rear = -1;
}

int isEmpty(struct PriorityQueue* pq) {
```

```
    return pq->front == -1;
}

int isFull(struct PriorityQueue* pq) {
    return pq->rear == MAX - 1;
}

void insert(struct PriorityQueue* pq, int value) {
    if (isFull(pq)) {
        printf("Queue is full. Cannot insert %d.\n", value);
        return;
    }
    if (isEmpty(pq)) {
        pq->front = pq->rear = 0;
        pq->data[pq->rear] = value;
    } else {
        int i = pq->rear;
        while (i >= pq->front && pq->data[i] < value) {
            pq->data[i + 1] = pq->data[i];  // Shift the element to make space
            i--;
        }
        pq->data[i + 1] = value;
        pq->rear++;
    }
    printf("Inserted %d\n", value);
}

void delete(struct PriorityQueue* pq) {
    if (isEmpty(pq)) {
        printf("Queue is empty. Cannot delete.\n");
        return;
    }
    printf("Deleted %d\n", pq->data[pq->front]);
    if (pq->front == pq->rear) {
        pq->front = pq->rear = -1;
    } else {
        pq->front++;
    }
}

void display(struct PriorityQueue* pq) {
    if (isEmpty(pq)) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Priority Queue elements:\n");
    for (int i = pq->front; i <= pq->rear; i++) {
        printf("%d ", pq->data[i]);
```

```
    }
    printf("\n");
}

int main() {
    struct PriorityQueue pq;
    createQueue(&pq);
    int choice, value;

    while (1) {
        printf("\nPriority Queue Operations:\n");
        printf("1. Create queue\n");
        printf("2. Insert (enqueue)\n");
        printf("3. Delete (dequeue)\n");
        printf("4. Display all elements\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                createQueue(&pq);
                printf("Queue created.\n");
                break;
            case 2:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insert(&pq, value);
                break;
            case 3:
                delete(&pq);
                break;
            case 4:
                display(&pq);
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }
    return 0;
}
```

Output :

Output

```
/tmp/8AeXeW8EBD.o

Priority Queue Operations:
1. Create queue
2. Insert (enqueue)
3. Delete (dequeue)
4. Display all elements
5. Exit
Enter your choice: 1
Queue created.

Priority Queue Operations:
1. Create queue
2. Insert (enqueue)
3. Delete (dequeue)
4. Display all elements
5. Exit
Enter your choice: 2
Enter value to insert: 8
Inserted 8

Priority Queue Operations:
1. Create queue
2. Insert (enqueue)
3. Delete (dequeue)
4. Display all elements
5. Exit
Enter your choice: 2
Enter value to insert: 5
Inserted 5

Priority Queue Operations:
1. Create queue
2. Insert (enqueue)
3. Delete (dequeue)
4. Display all elements
5. Exit
Enter your choice: 2
Enter value to insert: 2
Inserted 2

Priority Queue Operations:
1. Create queue
2. Insert (enqueue)
3. Delete (dequeue)
4. Display all elements
5. Exit
Enter your choice: 2
Enter value to insert: 2
Inserted 2

Priority Queue Operations:
1. Create queue
2. Insert (enqueue)
3. Delete (dequeue)
4. Display all elements
```

```
Priority Queue Operations:
1. Create queue
2. Insert (enqueue)
3. Delete (dequeue)
4. Display all elements
5. Exit
Enter your choice: 3
Deleted 8

Priority Queue Operations:
1. Create queue
2. Insert (enqueue)
3. Delete (dequeue)
4. Display all elements
5. Exit
Enter your choice: 4
Priority Queue elements:
5 2 2

Priority Queue Operations:
1. Create queue
2. Insert (enqueue)
3. Delete (dequeue)
4. Display all elements
5. Exit
Enter your choice: 5
```

5) Binary Search Tree (BST)

Aim:
Write a program for following operations on Binary Search Tree using Doubly Linked List (DLL).

1) Create empty BST,
2) Insert a new element on the BST
3) Search for an element on the BST
4) Delete an element from the BST
5) Display all elements of the BST

Algorithm :
1. createBST() – This function should create a ROOT pointer with NULL value as empty BST.
2. insert(typedef newelement) – This void function should take a newelement as an argument to be inserted on an existing BST following the BST property.
3. typedef search(typedef element) – This function should search for specified element on the non-empty BST and return a pointer to it.
4. **typedef delete(typedef element)** – This function searches for an element and if found deletes it from the BST and returns the same.
5. **typedef getParent(typedef element) -** This function searches for an element and if found, returns its parent element.
6. DisplayInorder( ) – This is a void function which should go through non- empty BST, traverse it in inorder fashion and print the elements on the way.

Code :

```
#include <stdio.h>
#include <stdlib.h>

// Node structure for BST
typedef struct Node {
    int data;
    struct Node *left, *right, *parent;
} Node;

// Function to create a new node
Node* createNode(int data, Node* parent) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    newNode->parent = parent;
    return newNode;
}

// Function to create an empty BST
Node* createBST() {
    return NULL;  // Initially the root is NULL
}

// Function to insert a new element into the BST
```

```
Node* insert(Node* root, int data) {
    if (root == NULL) {
        return createNode(data, NULL);  // Create root if tree is empty
    }
    if (data < root->data) {
        if (root->left == NULL) {
            root->left = createNode(data, root);
        } else {
            insert(root->left, data);
        }
    } else if (data > root->data) {
        if (root->right == NULL) {
            root->right = createNode(data, root);
        } else {
            insert(root->right, data);
        }
    }
    return root;
}

// Search for an element in the BST
Node* search(Node* root, int data) {
    if (root == NULL || root->data == data) {
        return root;
    }
    if (data < root->data) {
        return search(root->left, data);
    } else {
        return search(root->right, data);
    }
}

// Find the minimum node in a subtree
Node* findMin(Node* root) {
    while (root->left != NULL) {
        root = root->left;
    }
    return root;
}

// Delete an element from the BST
Node* delete(Node* root, int data) {
    if (root == NULL) {
        return root;
    }
    if (data < root->data) {
        root->left = delete(root->left, data);
    } else if (data > root->data) {
        root->right = delete(root->right, data);
    } else {
```

```
        // Node found
        if (root->left == NULL) {
            Node* temp = root->right;
            if (temp) temp->parent = root->parent;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            Node* temp = root->left;
            if (temp) temp->parent = root->parent;
            free(root);
            return temp;
        }

        // Node with two children
        Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = delete(root->right, temp->data);
    }
    return root;
}

// Function to get the parent of an element
Node* getParent(Node* root, int data) {
    Node* node = search(root, data);
    if (node && node->parent) {
        return node->parent;
    }
    return NULL;  // No parent (either root or element not found)
}

// Inorder display (Left, Root, Right)
void DisplayInorder(Node* root) {
    if (root != NULL) {
        DisplayInorder(root->left);
        printf("%d ", root->data);
        DisplayInorder(root->right);
    }
}

int main() {
    Node* root = createBST();

    // Insert elements into the BST
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);
```

```
    // Display BST in Inorder
    printf("Inorder traversal: ");
    DisplayInorder(root);
    printf("\n");

    // Search for an element
    int search_val = 40;
    Node* found = search(root, search_val);
    if (found) {
        printf("Element %d found in the BST\n", search_val);
    } else {
        printf("Element %d not found in the BST\n", search_val);
    }

    // Delete an element
    root = delete(root, 50);
    printf("Inorder after deleting 50: ");
    DisplayInorder(root);
    printf("\n");

    // Get parent of an element
    Node* parent = getParent(root, 60);
    if (parent) {
        printf("Parent of 60 is %d\n", parent->data);
    } else {
        printf("Parent not found or element is root\n");
    }

    return 0;
}
```

Output :