Program-1 :- write a program to simulate stacks with, ~~insert~~, pop, push, overflow and underflow conditions

(LAB-2)

```
#include <stdio.h>

# DEFINE SIZE 10

void push(int);
void pop();
Void display();

int stack[SIZE], top=-1;

void main()
 }
  int value, choice;

  while(1) {

  Printf(" 1. Push, 2. Pop, 3. display, 4. Exit");
  Printf(" Enter choice : ");
  scanf("%d", &choice);

 switch(choice)
  }
   case 1 : Printf(" Enter the value to be inserted : ");
            scanf("%d", &value),
            push(value);
            break;

   case : 2 : pop();
              break;
```

```
case 3: display();
        break;

case 4: exit(0);

default: printf("wrong selection!");

}

}

}

void push (int value)

}
  if( top == SIZE-1)
   }
   printf(" stack is full ");
  else
    }
     top++;
    stack[top] = value;
    printf(" Insertion Done");
 }
}

 void pop()
 }
   if (top == -1)
     }
       printf(" stack is empty");
     }
```

```
else{
  printf(" Element deleted");
  top--;
 }
}
void display()
 {
  if(top == -1)
     printf(" stack is empty");
  else
  {
   int i;
   printf("stack");
   for(i= top; i>=0; i--)
      printf("\n %d \n ,stack[i]);
  }
 }
```

~~Complete output~~ :-

Program-2 :- WAP to convert a given valid parenthesized infix arithmetic expression to postfix expression.

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

#define SIZE 100;

int isOperator(char ch)
  {
    return(ch=="+" || ch=='-' || ch=='*' || ch=='/');
  }

int getprecedence(char operator){
    if(operator=='+' || operator=='-')
     return 1;
    else if(operator=='*' || operator=='/')
     return 2;
   else
     return 0;
  }

void infixtopostfix(char infix[], char postfix[])
{
  int i,j;
  char stack[SIZE];
  int top=-1;
```

```
for(i=0,j=0; infix[i] != '\0'; i++)
  {
  if (isalnum (infix[i])
       postfix[j++] = infix[i];
  elseif (infix[i] == '(')
     stack [++top] = infix[i];
  elseif (infix[i] == ')')
   {
    while (top != -1 && stack[top] != '(')
      postfix[j++] = stack[pop--];

     top--;
   }
  elseif (isoperator(infix[i])) {
     while (top != -1 && getprecedence (stack[top]) >=
          getprecedence (infix[i]))
               postfix[j++] = stack[top]

     stack [++top] = infix[i];

   }

}
while (top != -1) {
        postfix[j++] = stack[top--];

  }
  postfix[j] = '\0';
}
```

```
int main()
}
  char infix[SIZE] = stack[top--];
    }
  postfix[j] = '\0';
}
int main()
    }
     char infix[size], postfix[SIZE];
     printf("Enter ~~prefi~~ infix expression \n");
     scanf("%s", &infix);
     infixTopostfix(infix, postfix);
     printf("Infix expression : %s\n", infix);
     printf("postfix Expression : %s\n", postfix);
     return 0;
}
```

~~example output~~

LAB-3 - 8/1/24

1) WAP to simulate the working of queue of integers using an array. Provide the following operations of insert, delete, display, also print message for overflow and underflow conditions.

```
#include<stdio.h>
#define SIZE 5
int front = -1;
int rear = -1;
int Queue[SIZE]
void insert(){
    if(rear == SIZE - 1)
        printf("The Queue is overflow");
    else{
        if(front == -1)
            front = 0;
        int add;
        printf("Insert an element :");
        scanf("%d", &add);
        rear = rear+1;
        Queue[rear] = add;
    }
}
```

```
void delete ()
{
  if ( front == rear || front > rear )
     printf (" The Queue is in underflow condition ");
  else
    {
     printf (" The deleted item is %d ", queue [front]);
     front = front + 1;
    }

void display ()
  {
    if ( front == -1 )
       printf (" The Queue is empty ! ");
       int i ;
       else{
       for ( i = front ; i <= rear ; i++ )
         printf (" %d \n ", queue [i] );
    }
  }

void main()
 {
   int choice,
   while(1)
    {
      printf (" 1. Insert \n 2. Delete \n 3. Display \n 4.
        quit ");
```

```
printf("Enter a choice:");
scanf("%d", &choice);
switch(choice)
  {
   case 1:
        insert();
        break;

   case 2:
        delete();
        break;

   case 3: display();
        break;

   case 4: exit(1)

    }
  }
}
```

Date :- 8/1/24

2. write a program to simulate the working of a circular queue using an array, provide operations like insert, delete, display, also underflow and overflow conditions should be mentioned

```
#include <stdio.h>

#define SIZE 5

int item[SIZE];
int front = -1, rear = -1;

int isfull(){
    if ((front == rear+1) || (front == 0 && rear
    == SIZE-1));
    return 1;
    }
  return 0;
}

int isEmpty(){
    if (front == -1)
      return 1;
  return 0;
```

```
void enqueue (int element){

    if (isfull())
        {
        printf("\n The queue is full");
        }

    else
      {
       if (front == -1)

          front = 0;

       rear = (rear+1) % SIZE;
       item [rear] = element;
       printf ("\n Inserted %d", element);

    }

}

void dequeue (){

        int element
        if (isEmpty()){
          printf("\n queue is empty \n");
          return (-1);

      }

      else
        {
         element = item[front];
         if (front == rear){

          front = -1
          rear = -1;

      }
```

```
lse
 front =(front + 1) % SIZE;

 printf ("\n Deleted Element %d \n",element);
 }

void display ()
    {
     int i;
     if ( isEmpty ())
       printf ("\n Empty queue ! ");
     else
       {
        Printf ("\n front position = %d \n", front );
        for (i = front ; i ! = rear ; i = (i+1) % SIZE )
        { printf ("%d \n", item [i]);
          }
        printf ("%d ", item[i]);
     }

void main ()
    int choice, element;
     while (1)
    {
     printf (" 1. Enque \n 2. Dequeue \n 3. Display
              \n Exit);
```

```
printf("Enter the choice now");
scanf("%d",&choice);
switch(choice)
{
case 1: printf("Enter Element");
        scanf("%d" &element);
        enqueue(element);
        break;

case 2:
    element = dequeue()
    break;
    case 3:
        display();
        break;
    case 4:
        exit(0);
    }
}
```

8) WAP to implement single linked list with delete operation at beggining / end / any position and display it aswell

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

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

struct Node* createnode (int value) {
    struct node * newNode = (struct Node*) malloc
    (sizeof (struct node));

    newnode -> data = value;
    newnode -> next = NULL;
    return newnode;
}

void displaylist (struct node* head)
    {
    struct node * current = head;
    while (current != NULL)
    {
    printf ("%d ", current -> data);
    current = current -> next
    }
```

```
struct node* deleteFirst(struct node
{
  if (head == NULL)
    {
      printf("empty!");
    }
  struct node* temp = head;
  head = head -> next;
  free(temp);
  return head;
}

struct node* deleteElement(struct node* head, int val
{
  if (head == NULL)
    {
      printf("empty!");
      return NULL;
    }
  struct node* current value;
  struct node* previous = NULL

  while(current != NULL && current -> data != value
    {
      previous = current;
      current = current -> next;
    }
  if(current == NULL)
    {
      printf("element not found");
      return head;
    }
```

```
if (previous == NULL)
  {
    head = head -> next
  }
else
  {
    previous -> next = current -> next;
  }
return head;
}

struct node * deleteLast (struct *node head)
  {
    if (head == NULL)
      {
        printf (" list is empty");
        return NULL;
      }
    if (head -> next == NULL)
      {
        free(head));
        return NULL;
      }
    struct node * current = head;
    struct node * previous = NULL;
    while (current -> next != NULL)
      {
        previous = current;
        current = previous; current -> next
      }
```

```
if (previous == NULL)
  {
    head = head -> next
  }
else
  {
    previous -> next = current -> next;
  }
return head;
}

struct node * deleteLast (struct node *head)
  {
    if (head == NULL)
      {
        printf (" list is empty");
        return NULL;
      }
    if (head -> next == NULL)
      {
        free(head);
        return NULL;
      }
    struct node * current = head;
    struct node * previous = NULL;
    while (current -> next != NULL)
      {
        previous = current;
        current = previous, current -> next
      }
```

```
previous -> next = NULL;
free (current);
return head;
}

int main()
{
    struct node* head = NULL;
    int choice, value
    printf("enter elements for linked list, enter -1 to stop inserting");
    while (1)
    {
        scanf("%d", &value);
        [redacted]
        }
        if (value == -1)
        {
            break;
        }
    if(head == NULL)
    {
        head = createnode(value)
    }
    else
    {
        struct node* current = head
        while (current->next != NULL)
        {
            current = current->next;
        }
```

```
current -> next = createNode(value);
    }
  }

printf("After creating we have:");
displayList(head);

while(1)
{
printf("choose operation: \n 1: for deletion first \n 2:
        delete specified element \n 3: delete last element
        \n 4: exit);

scanf("%d", choice);

switch(choice)
    {
case 1: head = deleteFirst(head);
        printf("After operation 1")
        displayList(head);
        break;

case 2: printf("enter element to be deleted");
        scanf("%d", &value);
        head = deleteElement(head, value);
        printf("After deleting element");
        displayList(head);

case 3: head: deletelast(head)
        displayList(head);
        break;
```

```
case 4: exit(0);
default: printf("Invalid choice");
}
}
```

Sample Input Output -

Enter elements of the linkedlist :

10 20 30 40

Choose operation

1: Delete First element

2: Delete specified element

3: Delete Last element

4: exit

~~Input~~ - 1

20 30 40

Input : 2

Enter element to be deleted : 30

20 40

Input :- 3

20

(29/11/24)

9) WAP to sort elements in a linked list :- (Bubble sort)

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

struct node {
    int data;
    struct node* next;
};

void swap (struct node *a, struct Node *b) {
    int temp = a->data;
    a->data = b->data;
    b->data = temp;
}

void bubblesort (struct node * head)
{
    int swapped;
    struct node * ptr1;
    struct node* lptr= NULL
    if (head == NULL) {
        return;
    }
    do {
        swapped = 0;
        ptr1 = head;
        while (ptr1->next != lptr)
        {
            if (ptr1->data > ptr1->next->data) {
```

```
swap(ptr1, ptr1->next)
swapped = 1;
}
ptr1 = ptr1->next;
}
lptr = ptr1;
} while(swapped)
}

void insert(struct node **head, int data) {
struct Node* newnode = (struct node*) malloc
(sizeof(struct Node));
newNode -> data = data;
newNode -> next = *head;
*head = newNode;

void printlist(struct node* node)
{
while(node != NULL)
{
printf("%d", node->data);
node = node->next;
}
}
```

```
int main()
{
struct Node* head = NULL
insert(&head, 5);
insert(&head, 2);
insert(&head, 9);
insert(&head, 6);
printf("linked list before sorting");
printlist(head);
bubblesort(head);
printf("sorted linked list");
printlist(head);
}
```

output

linked list before sorting:

5 2 9 6

linked list after sorting:

2 5 6 9

sorting

NLP

29/11/24

10) WAP to concatenate two linked list

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

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

void insertatend(struct node** head, int data){
    struct node* newNode = (struct node*) malloc(
    sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    if(*head == NULL)
        *head = newNode;
    return;

    struct Node* temp = *head;
    while(temp->next != NULL){
        temp = temp->next
    }
    temp->next = newNode;

void concatenatelist(struct Node** list1, struct node*
    list2){
    if(*list1 == NULL){
        *list1 = list2;
        return;
    }
    struct node* temp = *list1;
    while(temp->next != NULL){
        temp = temp->next;
    }
    temp->next = list2;
}
void printlist(struct node* node){
    while(node != NULL){
        printf("%d", node->data);
        node = node->next;
    }
    printf("\n");
}
```

evaluated NJ 29/11/24

```
int main(){
    struct node* list1 = NULL;
    struct node* list2 = NULL;
    insertend(&list1, 1);
    insertend(&list1, 2);
    insertend(&list2, 3);
    insertend(&list2, 4);
```

```
printf(" Before concatenation ");
printlist(list1);
printlist(list2);
printf(" After concatenation");
printlist(list1);
}
```

output :-

list1 : 1 2 3

list2 : 4 5 6

concatenated list : 1 2 3 4 5 6

Continuation program

a) WAP to reverse elements in a single linked list

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

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

struct Node* insertAtBeginning(
        struct node * head, int newData) {

struct node * newNode = (struct node*) malloc(sizeof(
                        struct Node));

  newNode -> data = NewData
  newNode -> next = head
  return newNode;
}

struct node * reverselinkedlist(struct node * head)

  struct Node * prev = NULL;
  struct Node * current = head;
  struct Node * nextNode = NULL;

  while (current != NULL)
    {
      nextNode = current -> next;
      current -> next = prev;
      prev = current;
      current = nextNode; }
```

```
return prev;

}

void printList(struct Node* head)
    {
     struct Node* current = head;
     while (current != NULL){
        printf("%d", current -> data);
        current = current -> next;
      }
     printf("\n");

}

int main(){

     struct node * head = NULL;
     head = insertAtBeginning(head, 3);
     head = insertAtBeginning(head, 4);
     head = insertAtBeginning(head, 6);
     printf("LinkedList before Reverse");
     printList(head);
     head= reverseLinkedList(head);
     printf("Reversed LinkedList");
     printList(head);

}
```

Output :-

Linked list before reversing : 3 4 6

Reversed linked list : 6 4 3

---

12) WAP to implement stack using single linked list

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

struct Node{
  int Data;
  struct Node* next;
};

struct stack
  {
    struct Node* top;
  };

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

int isEmpty(struct stack * stack)
   {
     return (stack -> top == NULL);
   }
```

```
void push(struct stack* stack, int newData) {
 struct Node* newNode = (struct Node*)malloc(sizeof
 (struct Node));
 newNode -> data = newData;
 newNode -> next = stack -> top;
 stack -> top = newNode;
 printf("%d pushed to the current stack\n", newData);
}

void pop(struct stack* stack)
 {
  if (isEmpty(stack)) {
  printf("Stack is Empty, cannot pop anymore");
  return;
  }
  struct node *temp = stack -> top;
  stack -> top = temp -> next;
  printf("%d popped from stack\n", temp -> data);
}

void printstack(struct stack* stack)
 {
if (isEmpty(stack))
  {
  printf("Stack is empty");
  return;
 }
```

```
printf("Printing stacks");
struct node* current = stack -> top;
while (current != NULL)
{
 printf("%d", current -> data);
 current = current -> next;
}
printf("\n");
}

int main()
 {
  struct stack * stack = initializestack();
  push(stack, 3);
  push(stack, 7);
  push(stack, 5);
  printstack(stack);
  pop(stack);
  pop(stack);
  pop(stack);
printstack(stack);
return 0;
}
```

output :-

3 pushed to stack

7 pushed to stack

5 pushed to stack

stack elements : 5 7 3

5 popped from stack

7 popped from stack

3 popped from stack , stack is empty