01.Linear Search.c

Aim:
To write a C program that performs linear search on an array to find the position of a given element.

Algorithm:

1.Start the program.
2.Read the size of the array (n).
3.Read n elements into the array.
4.Read the element to be searched (key).
5.Traverse the array from index 0 to n–1.
6.If any element matches key, print its position and stop the search.
7.If the loop ends without a match, print “Element not found”.
8.End the program.

Program:

```
#include <stdio.h>
int main()
{
    int n,key,i;
    printf("Enter the size of the array : ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter the elements of the array : ");

    for(i=0;i<n;i++)
    scanf("%d", &arr[i]);
    printf("Enter the element to search : ");
    scanf("%d", &key);

    for(i=0;i<n;i++)
    {
        if(arr[i] == key)
        {
            printf("Element found at position %d \n", i+1);
            return 0;
        }
    }
    printf("Element not found");
}
```

Output:

```
Enter the size of the array : 5
Enter the elements of the array : 10 25 30 45 50
Enter the element to search : 30
Element found at position 3
```

Result:
This program searches an element in an array using linear search and correctly displays whether the element is found or not.

02.Binary Search.c

Aim:
To write a C program to search an element in a sorted array using the Binary Search algorithm.

Algorithm:

```
1.Start the program.
2.Read the number of elements n from the user.
3.Read the n elements of the array in sorted order.
4.Read the element to be searched (key).
5.Initialize
   low = 0,
   high = n - 1.
6.Repeat while low <= high:
a. Find the middle index: mid = (low + high) / 2.
b. If arr[mid] == key,
   Display that the element is found at position mid + 1.
   Set found = 1 and exit the loop.
c. If arr[mid] < key, set low = mid + 1.
d. Else set high = mid - 1.
7.If the loop ends and found is still 0, display “Element not found”.
8.Stop the program.
```

Program:

```c
#include <stdio.h>
int main()
{
   int n,i,mid,low,high,key,found=0;
   printf("Enter the number of elements : ");
   scanf("%d", &n);

   int arr[n];
   printf("Enter the %d elements in sorted order : ",n);
```

```
    for(i=0;i<n;i++)
    {
    scanf("%d", &arr[i]);
    }

    low=0;
    high=n-1;

    printf("Enter the element to search : ");
    scanf("%d", &key);

    while(low <= high)
    {
        mid = (low + high)/2;
        if (arr[mid] == key)
        {
            printf("The Element %d is found at the position %d", key, mid+1);
            found = 1;
            break;
        }
        else if (arr[mid] < key)
        {
            low = mid + 1;
        }
        else
        {
            high = mid - 1;
        }
    }
    if (!found)
    {
        printf("Element not found");
        return 0;
    }
}
```

Output:

```
Enter the number of elements : 5
Enter the 5 elements in sorted order : 10 20 30 40 50
Enter the element to search : 30
The Element 30 is found at position 3
```

Result:

This program uses binary search to quickly find an element in a sorted array and displays whether the element is found or not.

03.Stack Operations Using Array.c

Aim:
To write a C program to implement stack operations such as push, pop, and display using an array.

Algorithm:

```
1. Start the program.
2. Read the stack size (MAX).
Initialize Top = -1.
3. Define stack operations:
Push():
If Top == MAX-1, display "Overflow".
Else increase Top and insert item.
Pop():
If Top == -1, display "Underflow".
Else remove and decrease Top.
Display():
Print all elements from index 0 to Top.
4. In main():
Create stack array.
Display menu:
1 → Push
2 → Pop
3 → Exit
Perform the chosen operation using switch–case.
5. Repeat until user selects Exit.
6. End the program.
```

Program:

```c
#include<stdio.h>
int Top = -1, MAX;

void display(int stack[])
{
   printf("Current stack elements are-\n");
   for(int i=0; i<=Top; i++)
```

```
    {
        printf(">>%d", stack[i]);
    }
    printf("(Top)");
}

void push(int stack[])
{
    int item;
    if(Top == MAX - 1)
    {
        printf("Stack is Overflow");
    }
    else
    {
        printf("Enter the elements to push : ");
        scanf("%d", &item);
        Top = Top+1;
        stack[Top] = item;
    }
}

void pop(int stack[])
{
    int item;
    if(Top == -1)
    {
        printf("Stack is Underflow");
    }
    else
    {
        item = stack[Top];
        Top = Top-1;
    }
}

int main()
{
    int operation = 0, top = -1;
    printf("Enter the size of stack : ");
    scanf ("%d", &MAX);
    int stack[MAX];
    while(operation != 3)
    {
        printf("\n 1.Push \n 2.Pop \n 3.Exit \n");
        printf("\nPlease choose stack operation to perform : ");
        scanf("%d", &operation);
        switch(operation)
        {
            case 1:
            push(stack);
            display(stack);
            break;
            case 2:
            pop(stack);
            display(stack);
            break;
            case 3:
            return 0;
            default:
            printf("Incorrect operation");
            break;
        }
    }
return 0;
}
```

Output:

```
Enter the size of stack : 5

1.Push
2.Pop
3.Exit
Please choose stack operation to perform : 1
Enter the elements to push : 10
Current stack elements are-
>>10(Top)

1.Push
2.Pop
3.Exit
Please choose stack operation to perform : 1
Enter the elements to push : 20
Current stack elements are-
>>10>>20(Top)

1.Push
2.Pop
```

```
3.Exit
Please choose stack operation to perform : 2
Current stack elements are-
>>10(Top)

1.Push
2.Pop
3.Exit
Please choose stack operation to perform : 2
Stack is Underflow
Current stack elements are-
>>10(Top)
```

Result:
This program successfully demonstrates stack operations using an array, allowing the user to push, pop, and display stack elements.

04.Queue Operations Using Array.c

Aim:
To write a C program to implement queue operations using array, including Enqueue, Dequeue, and Display, along with proper handling of overflow and underflow conditions.

Algorithm:

```
1. Start the program
2. Initialize
front = -1
rear = -1
Read MAX (size of the queue)
Create array queue[MAX]
3. Enqueue Operation
Check if rear == MAX - 1
→ If true, display Queue Overflow
Else:
Read the element item
If queue is empty (front == -1 and rear == -1):
→ set front = rear = 0
Else:
```

```
→ rear = rear + 1
Insert item into queue[rear]
4. Deqeueue Operation
Check if queue is empty (front == -1 or front > rear)
→ Display Queue Underflow
Else:
Remove element from queue[front]
Increment front = front + 1
If front > rear after deletion:
→ Reset queue → front = rear = -1
5. Display Operation
If queue is empty: print Queue is Empty
Else:
Print all elements from queue[front] to queue[rear]
6. Repeat operations until user chooses Exit
7. Stop the program
```

Program:

```
#include<stdio.h>

int front = -1, rear = -1, MAX;

void enqueue(int queue[])
{
   int item;
   if(rear == MAX - 1)
   {
      printf("Queue is Overflow\n");
   }
   else
   {
      printf("Enter element to enqueue : ");
      scanf("%d", &item);

      if(front == -1 && rear == -1)   // first element
      {
         front = rear = 0;
      }
      else
```

```
        {
            rear = rear + 1;
        }
        queue[rear] = item;
    }
}

void dequeue(int queue[])
{
    if(front == -1 || front > rear)
    {
        printf("Queue is Underflow\n");
    }
    else
    {
        int item = queue[front];
        front = front + 1;

        // Reset queue if empty
        if(front > rear)
        {
            front = rear = -1;
        }
    }
}

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

int main()
{
    int choice;

    printf("Enter the size of queue: ");
    scanf("%d", &MAX);

    int queue[MAX];

    while(1)
    {
        printf("\n1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\n");
        printf("Choose queue operation: ");
        scanf("%d", &choice);

        switch(choice)
        {
            case 1:
                enqueue(queue);
                break;
            case 2:
                dequeue(queue);
                break;
            case 3:
                display(queue);
                break;
            case 4:
                return 0;
            default:
                printf("Invalid choice\n");
        }
    }
}
```

Output:

```
Enter the size of queue: 5
```

```
1. Enqueue
2. Dequeue
3. Display
4. Exit
Choose queue operation: 1
Enter element to enqueue : 10

1. Enqueue
2. Dequeue
3. Display
4. Exit
Choose queue operation: 1
Enter element to enqueue : 20

Choose queue operation: 3
Current Queue elements: 10 20 <-rear

Choose queue operation: 2
(10 is deleted)

Choose queue operation: 3
Current Queue elements: 20 <-rear

Choose queue operation: 4
```

Result:
The program successfully implements queue operations using an array.
It performs enqueue, dequeue, and display operations correctly and handles overflow and underflow conditions properly.

05.String Reversal (Using Stack).c

Aim:
To write a C program that reverses a string using stack operations (push and pop) implemented with an array.

Algorithm:

1. Start the program.
2. Read the size of the character stack (MAX).
3. Allocate memory for the stack using realloc.

```
4. Define push(item):
If stack is full, display “Overflow”.
Else increment Top and insert character.
5. Define pop():
If stack is empty, display “Underflow”.
Else return the character at Top and decrement Top.
6. In StringReversal():
Read a string from the user.
Push every character of the string onto the stack.
Pop all characters and print them to get the reversed string.
7. End the program.
```

Program:

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

int Top = -1, MAX;
char *STACK;

void push(char item)
{
if(Top == MAX - 1)
{
printf("Stack is Overflow");
}
else
{
Top = Top + 1;
STACK[Top] = item;
}
}

char pop()
{
char item;
if (Top == -1)
{
printf("Stack is Underflow / Empty \n");
return '\0';
}
else
{
item=STACK[Top];
Top = Top - 1;
return item;
}
}

void StringReversal()
{
char string[MAX];
printf("Please Enter the string of size %d to reverse : ", MAX);
scanf("%s", string);

for(int c=0; c<MAX; c++)
{
push(string[c]);
}

printf("Reversed String : ");

for(int c=0; c<MAX; c++)
{
printf("%c", pop());
}
}

int main()
{
printf("Enter the size of CHAR STACK : ");
scanf("%d", &MAX);
STACK = realloc(STACK, MAX);
StringReversal();
return 0;
}
```

Output:

```
Enter the size of CHAR STACK : 5
Please Enter the string of size 5 to reverse : HELLO
Reversed String : OLLEH
```

Result:
The program successfully reverses a string using stack operations, demonstrating the LIFO (Last In, First Out) property of stacks.Aim:
To write a C program that reverses a string using stack operations (push and pop) implemented with an array.

Algorithm:

1. Start the program.
2. Read the size of the character stack (MAX).
3. Allocate memory for the stack using realloc.
4. Define push(item):
If stack is full, display “Overflow”.
Else increment Top and insert character.
5. Define pop():
If stack is empty, display “Underflow”.
Else return the character at Top and decrement Top.
6. In StringReversal():
Read a string from the user.
Push every character of the string onto the stack.
Pop all characters and print them to get the reversed string.
7. End the program.

Program:

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

int Top = -1, MAX;
char *STACK;

void push(char item)
{
if(Top == MAX - 1)
{
printf("Stack is Overflow");
}
else
{
Top = Top + 1;
STACK[Top] = item;
}
}

char pop()
{
char item;
if (Top == -1)
{
```

```
printf("Stack is Underflow / Empty \n");
return '\0';
}
else
{
item=STACK[Top];
Top = Top - 1;
return item;
}
}

void StringReversal()
{
char string[MAX];
printf("Please Enter the string of size %d to reverse : ", MAX);
scanf("%s", string);

for(int c=0; c<MAX; c++)
{
push(string[c]);
}

printf("Reversed String : ");

for(int c=0; c<MAX; c++)
{
printf("%c", pop());
}
}

int main()
{
printf("Enter the size of CHAR STACK : ");
scanf("%d", &MAX);
STACK = realloc(STACK, MAX);
StringReversal();
return 0;
}
```

Output:

```
Enter the size of CHAR STACK : 5
Please Enter the string of size 5 to reverse : HELLO
Reversed String : OLLEH
```

Result:
The program successfully reverses a string using stack operations, demonstrating the LIFO (Last In, First Out) property of stacks.Aim:
To write a C program that reverses a string using stack operations (push and pop) implemented with an array.

Algorithm:

```
1. Start the program.
2. Read the size of the character stack (MAX).
3. Allocate memory for the stack using realloc.
4. Define push(item):
If stack is full, display “Overflow”.
Else increment Top and insert character.
5. Define pop():
If stack is empty, display “Underflow”.
Else return the character at Top and decrement Top.
6. In StringReversal():
Read a string from the user.
Push every character of the string onto the stack.
Pop all characters and print them to get the reversed string.
7. End the program.
```

Program:

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

int Top = -1, MAX;
char *STACK;

void push(char item)
{
if(Top == MAX - 1)
{
printf("Stack is Overflow");
}
else
{
Top = Top + 1;
STACK[Top] = item;
}
}
```

```
char pop()
{
char item;
if (Top == -1)
{
printf("Stack is Underflow / Empty \n");
return '\0';
}
else
{
item=STACK[Top];
Top = Top - 1;
return item;
}
}

void StringReversal()
{
char string[MAX];
printf("Please Enter the string of size %d to reverse : ", MAX);
scanf("%s", string);

for(int c=0; c<MAX; c++)
{
push(string[c]);
}

printf("Reversed String : ");

for(int c=0; c<MAX; c++)
{
printf("%c", pop());
}
}

int main()
{
printf("Enter the size of CHAR STACK : ");
scanf("%d", &MAX);
STACK = realloc(STACK, MAX);
StringReversal();
return 0;
}
```

Output:

```
Enter the size of CHAR STACK : 5
Please Enter the string of size 5 to reverse : HELLO
Reversed String : OLLEH
```

Result:
The program successfully reverses a string using stack operations, demonstrating the LIFO (Last In, First Out) property of stacks.Aim:
To write a C program that reverses a string using stack operations (push and pop) implemented with an array.

Algorithm:

```
1. Start the program.
2. Read the size of the character stack (MAX).
3. Allocate memory for the stack using realloc.
4. Define push(item):
If stack is full, display “Overflow”.
Else increment Top and insert character.
5. Define pop():
If stack is empty, display “Underflow”.
Else return the character at Top and decrement Top.
6. In StringReversal():
Read a string from the user.
Push every character of the string onto the stack.
Pop all characters and print them to get the reversed string.
7. End the program.
```

Program:

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

int Top = -1, MAX;
char *STACK;

void push(char item)
{
if(Top == MAX - 1)
{
printf("Stack is Overflow");
}
```

```
else
{
Top = Top + 1;
STACK[Top] = item;
}
}

char pop()
{
char item;
if (Top == -1)
{
printf("Stack is Underflow / Empty \n");
return '\0';
}
else
{
item=STACK[Top];
Top = Top - 1;
return item;
}
}

void StringReversal()
{
char string[MAX];
printf("Please Enter the string of size %d to reverse : ", MAX);
scanf("%s", string);

for(int c=0; c<MAX; c++)
{
push(string[c]);
}

printf("Reversed String : ");

for(int c=0; c<MAX; c++)
{
printf("%c", pop());
}
}

int main()
{
```

```
printf("Enter the size of CHAR STACK : ");
scanf("%d", &MAX);
STACK = realloc(STACK, MAX);
StringReversal();
return 0;
}
```

Output:

```
Enter the size of CHAR STACK : 5
Please Enter the string of size 5 to reverse : HELLO
Reversed String : OLLEH
```

Result:
The program successfully reverses a string using stack operations, demonstrating the LIFO (Last In, First Out) property of stacks.

06.Infix To Postfix Conversion.c

Aim:
To convert a given infix expression to its equivalent postfix expression using a stack.

Algorithm:

```
1.Create an empty stack for operators.
2.Read the infix expression from left to right (character by character).
3.If the character is an operand (letter or digit), append it to the postfix output.
4.If the character is '(', push it onto the stack.
5.If the character is ')', pop from the stack and append to postfix until '(' is popped.
6.If the character is an operator (+ - * / ^), then:
  While the stack is not empty and top of stack has operator of higher or equal precedence (for ^ you may treat as right-associative if needed), pop it and append to postfix.
  Push the current operator onto the stack.
  After the entire infix is processed, pop any remaining operators from the stack and append to postfix.
7.Terminate postfix string and display it.
```

Program:

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

```
#define MAX 100

char stack[MAX];
int top = -1;

int isEmpty() {
    return top == -1;
}

void push(char item) {
    if (top >= MAX - 1) {
        printf("stack overflow\n");
        return;
    }
    stack[++top] = item;
}

char pop() {
    if (isEmpty()) {
        printf("stack underflow\n");
        return '\0';
    }
    return stack[top--];
}

char peek() {
    if (isEmpty()) return '\0';
    return stack[top];
}

int precedence(char item) {
    if (item == '^') return 3;
    else if (item == '*' || item == '/') return 2;
    else if (item == '+' || item == '-') return 1;
    else return 0;
}

int main() {
    char infix[MAX], postfix[MAX];
    int i, k = 0;
    char ch, temp;

    printf("Enter Infix expression: ");
    scanf("%s", infix);
    for (i = 0; i < (int)strlen(infix); i++) {
        ch = infix[i];

        if (isalnum(ch)) {              /* operand */
            postfix[k++] = ch;
        } else if (ch == '(') {
            push(ch);
        } else if (ch == ')') {
            while (!isEmpty() && (temp = pop()) != '(') {
                postfix[k++] = temp;
            }
        } else {                        /* operator */
            while (!isEmpty() && precedence(peek()) >= precedence(ch)) {
                postfix[k++] = pop();
            }
            push(ch);
        }
    }

    while (!isEmpty()) {
        postfix[k++] = pop();
    }
    postfix[k] = '\0';

    printf("postfix expression: %s\n", postfix);
    return 0;
}
```

Output:

```
Enter Infix expression: (A+B)^D+E/(F+A*D)+C
postfix expression: AB+D^E F A D * + / + C +
```

Result:
The given infix expression is successfully converted into its equivalent postfix expression using stack operations.
The program correctly applies the rules of operator precedence and parentheses handling, and displays the postfix form of the entered infix expression.

07.Single Linked List All Operations Using Switch Case.c

Aim:
To write a C program to perform various operations on a singly linked list such as insertion at beginning, insertion at end, insertion at a specific position, deletion of a node, and displaying the list.

Algorithm:

```
1. Start the program.
2. Create a structure Node
Contains data
Contains pointer next
3. Create functions:
CreateNode() → Allocates memory and creates a new node
InsertAtBeginning() → Inserts new node at the start
InsertAtEnd() → Inserts new node at the end
InsertAtPosition() → Inserts a node at a given position
DeleteNode() → Deletes a node by value
DisplayList() → Prints all nodes
4. In main():
Initialize head = NULL
Display menu
Based on user choice, call the respective linked list function
Continue until user selects Exit
5. End the program.
```

Program:

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

struct Node
{
    int data;
    struct Node *next; // To hold the address of next node
```

```c
};

struct Node *CreateNode(int data)
{
    // STACK = (char *)malloc(MAX * sizeof(char));
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    if (!newNode)
    {
        printf("Memory allocation failed!\n");
        return NULL;
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

void InsertAtBeginning(struct Node **head, int data)
{
    struct Node *newNode = CreateNode(data);
    newNode->next = *head;
    *head = newNode;
    printf("Node with data %d inserted at beginning successfully  .\n", data);
}

void InsertAtEnd(struct Node **head, int data)
{
    struct Node *newNode = CreateNode(data);
    if (*head == NULL)
    {
        *head = newNode;
    }
    else
    {
        struct Node *temp = *head;
        while (temp->next != NULL)
            temp = temp->next;
        temp->next = newNode;
    }
    printf("Node with data %d inserted at the end successfully.\n", data);
}

void InsertAtPosition(struct Node **head, int data, int position)
{
    // Considering that the position starts from 1 (Head at 1)
    if (position < 1)
    {
        printf("Invalid position!\n");
        return;
    }

    if (position == 1)
    {
        InsertAtBeginning(head, data);
        return;
    }

    struct Node *prev = *head;
    for (int k = 1; (k < position - 1 && prev != NULL); k++)
    {
        prev = prev->next;
    }

    if (prev == NULL)
    {
        printf("Given position is out of range!\n");
        return;
    }

    struct Node *newNode = CreateNode(data);
    newNode->next = prev->next;
    prev->next = newNode;

    printf("Node with data %d inserted at position %d successfully.\n", data, position);
}

void DeleteNode(struct Node **head, int valueToDelete)
{
    if (*head == NULL)
    {
        printf("Linked List is empty, deletio operation can't be performed");
        return;
    }

    struct Node *temp = *head;

    if (temp->data == valueToDelete)
    {
        *head = temp->next;
        free(temp);
        printf("Data %d deleted from list.\n", valueToDelete);
        return;
    }

    /* while (temp != NULL && temp->data != valueToDelete)
    {
        prev = temp;
        temp = temp->next;
    }*/

    struct Node *prev = *head;
    while (prev->next != NULL)
    {
        if (prev->next->data == valueToDelete)
        {
            temp = prev->next;
            prev->next = temp->next;
            free(temp);
            printf("Data %d deleted from list.\n", valueToDelete);
            return;
        }
        prev = prev->next;
    }

    // If key not found
    if (prev->next == NULL)
    {
        printf("Element %d not found.\n", valueToDelete);
        return;
    }
}

void DisplayList(struct Node *head)
{
    if (head == NULL)
    {
        printf("List is empty.\n");
        return;
    }

    struct Node *temp;
    temp = head;
    printf("\nLinked List Nodes: ");
    while (temp != NULL)
    {
        // if(temp == head)
        // {
        //     printf("|head|%p| ->", temp);
        // }
        printf(" |At=%p|%d|Next=%p| -> ", temp, temp->data, temp->next);
        temp = temp->next;
    }
   // printf("NULL\n");
}

// Main function
int main()
{
    struct Node *head = NULL;
    int choice, data, pos;

    while (1)
    {
        printf("\n--- Linked List Menu ---\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert at End\n");
        printf("3. Insert at Position\n");
        printf("4. Delete by Value\n");
        printf("5. Display List\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            printf("Enter data to insert: ");
            scanf("%d", &data);
            InsertAtBeginning(&head, data);
            break;
        case 2:
            printf("Enter data to insert: ");
            scanf("%d", &data);
            InsertAtEnd(&head, data);
            break;
        case 3:
            printf("Enter data and position to insert: ");
            scanf("%d %d", &data, &pos);
            InsertAtPosition(&head, data, pos);
            break;
        case 4:
            printf("Enter value to delete: ");
            scanf("%d", &data);
            DeleteNode(&head, data);
            break;
        case 5:
            DisplayList(head);
            break;
        case 6:
            printf("Exiting...\n");
            exit(0);
        default:
            printf("Invalid choice! Try again.\n");
        }
    }
    return 0;
}
```

Output:

```
--- Linked List Menu ---
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete by Value
5. Display List
6. Exit
Enter your choice: 1
Enter data to insert: 10
Node with data 10 inserted at beginning successfully.

--- Linked List Menu ---
Enter your choice: 2
Enter data to insert: 20
Node with data 20 inserted at the end successfully.

--- Linked List Menu ---
Enter your choice: 3
Enter data and position to insert: 15 2
Node with data 15 inserted at position 2 successfully.

--- Linked List Menu ---
```

```
Enter your choice: 5
Linked List Nodes:  |At=0x1234|10|Next=0x2345| ->  |At=0x2345|15|Next=0x3456| ->
|At=0x3456|20|Next=0x0| ->

--- Linked List Menu ---
Enter your choice: 4
Enter value to delete: 15
Data 15 deleted from list.

--- Linked List Menu ---
Enter your choice: 5
Linked List Nodes:  |At=0x1234|10|Next=0x3456| ->  |At=0x3456|20|Next=0x0| ->
```

Result:
The program successfully performs all basic operations on a singly linked list, including insertion, deletion, and displaying the nodes.

08.Doubly Linked List All Operations Using Switch Case.c

Aim:
To write a C program to perform various operations on a Doubly Linked List, such as inserting nodes at the beginning and end, deleting nodes from the beginning and end, and displaying the list using a menu-driven (switch case) approach.

Algorithm:

```
1. Start the program
2. Define a structure NODE
data (stores value)
prev (pointer to previous node)
next (pointer to next node)
3. Initialize
head = NULL
INSERT AT BEGINNING
Create a new node using malloc()
Read the value and store it in newNode->data
If list is empty (head == NULL):
Set head = newNode
Else:
Set newNode->next = head
Set head->prev = newNode
```

```
Update head = newNode
INSERT AT END
Create a new node
If list is empty:
Set head = newNode
Else:
Traverse the list to reach last node
Set last->next = newNode
Set newNode->prev = last
DELETE FROM BEGINNING
If list is empty:
→ Display “List is empty”
Else:
Store head in temp
Move head = head->next
If head ≠ NULL → set head->prev = NULL
Free temp
DELETE FROM END
If list is empty: display “List empty”
If only one node:
Set head = NULL
Free that node
Else:
Traverse to last node
Set last->prev->next = NULL
Free last node
DISPLAY
If list is empty → print “List is empty”
Else:
Start from head
Print each node using:
data <-> data <-> ... NULL
6. Repeat operations until user selects EXIT
7. Stop the program
```

Program:

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

struct Node {
   int data;
   struct Node *prev, *next;
};
```

```
struct Node *head = NULL;

// Create new node
struct Node* createNode(int value) {
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

// Insert at beginning
void insertBeginning() {
    int value;
    printf("Enter value to insert: ");
    scanf("%d", &value);

    struct Node *newNode = createNode(value);

    if (head == NULL) {
        head = newNode;
    } else {
        newNode->next = head;
        head->prev = newNode;
        head = newNode;
    }
    printf("Node inserted at beginning.\n");
}

// Insert at end
void insertEnd() {
    int value;
    printf("Enter value to insert: ");
    scanf("%d", &value);

    struct Node *newNode = createNode(value);

    if (head == NULL) {
        head = newNode;
        return;
    }

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

    temp->next = newNode;
    newNode->prev = temp;

    printf("Node inserted at end.\n");
}

// Delete from beginning
void deleteBeginning() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node *temp = head;
    head = head->next;

    if (head != NULL)
        head->prev = NULL;

    free(temp);
    printf("Node deleted from beginning.\n");
}

// Delete from end
void deleteEnd() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node *temp = head;

    if (temp->next == NULL) {  // only one node
        head = NULL;
        free(temp);
        printf("Last node deleted.\n");
        return;
    }

    while (temp->next != NULL)
        temp = temp->next;
    temp->prev->next = NULL;
    free(temp);

    printf("Node deleted from end.\n");
}

// Display list
void display() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node *temp = head;
    printf("Doubly Linked List: ");

    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }

    printf("NULL\n");
}

int main() {
    int choice;

    while (1) {
        printf("\n--- DOUBLY LINKED LIST MENU ---\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert at End\n");
        printf("3. Delete from Beginning\n");
        printf("4. Delete from End\n");
        printf("5. Display\n");
        printf("6. Exit\n");

        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1: insertBeginning(); break;
            case 2: insertEnd(); break;
            case 3: deleteBeginning(); break;
            case 4: deleteEnd(); break;
            case 5: display(); break;
            case 6: return 0;
            default: printf("Invalid choice! Try again.\n");
        }
    }
}
```

Output:

```
--- DOUBLY LINKED LIST MENU ---
1. Insert at Beginning
2. Insert at End
3. Delete from Beginning
4. Delete from End
5. Display
6. Exit
Enter your choice: 1
Enter value to insert: 10
Node inserted at beginning.

Enter your choice: 2
Enter value to insert: 20
Node inserted at end.

Enter your choice: 5
Doubly Linked List: 10 <-> 20 <-> NULL

Enter your choice: 3
Node deleted from beginning.

Enter your choice: 5
Doubly Linked List: 20 <-> NULL

Enter your choice: 6
```

Result:
The program successfully implements all basic operations of a Doubly Linked List using an interactive menu. It demonstrates insertion, deletion, and traversal in both directions, showing the working of linked list manipulation in C.

09.Stack Operations Using Linked List.c

Aim:

To write a C program to implement basic stack operations such as Push, Pop, and Display using a linked list, where insertion and deletion are performed at the top of the list.

Algorithm:

```
1. Start the program
2. Define a structure NODE
Contains data
Contains pointer next
3. Initialize
top = NULL
PUSH (Insert at top)
Read the value
Create a new node
Set newNode->data = value
Set newNode->next = top
Update top = newNode
POP (Delete from top)
If top == NULL
→ Print “Stack Underflow”
Else:
Store top in temp
Update top = top->next
Free temp
DISPLAY
If stack is empty → print message
Else traverse from top to NULL
Print each node's data
4.Repeat using switch case until user chooses EXIT
5.Stop the program
```

Program:

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

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

struct Node *top = NULL;

// PUSH operation
```

```
void push() {
    int value;
    printf("Enter value to push: ");
    scanf("%d", &value);

    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = top;
    top = newNode;

    printf("Element pushed.\n");
}

// POP operation
void pop() {
    if (top == NULL) {
        printf("Stack Underflow.\n");
        return;
    }

    struct Node *temp = top;
    top = top->next;
    free(temp);

    printf("Element popped.\n");
}

// DISPLAY operation
void display() {
    if (top == NULL) {
        printf("Stack is empty.\n");
        return;
    }

    struct Node *temp = top;
    printf("Stack elements (top to bottom): ");

    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }

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

```
int main() {
    int choice;

    while (1) {
        printf("\n--- STACK USING LINKED LIST ---\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: push(); break;
            case 2: pop(); break;
            case 3: display(); break;
            case 4: return 0;
            default: printf("Invalid choice!\n");
        }
    }
}
```

Output:

```
--- STACK USING LINKED LIST ---
1. Push
2. Pop
3. Display
4. Exit
Enter choice: 1
Enter value to push: 15
Element pushed.

Enter choice: 1
Enter value to push: 25
Element pushed.

Enter choice: 3
Stack elements (top to bottom): 25 -> 15 -> NULL

Enter choice: 2
Element popped.

Enter choice: 3
```

```
Stack elements (top to bottom): 15 -> NULL

Enter choice: 4
```

Result:
The program successfully demonstrates stack operations using a linked list. It performs push, pop, and display efficiently by inserting and deleting elements at the top of the linked list.

10.Circular Queue Operations Using Linked List.c

Aim:
To implement Circular Queue operations such as enqueue, dequeue, and display using a linked list, where the last node links back to the first node.

Algorithm:

```
1. Start the program
2. Define a NODE structure
data
next pointer
ENQUEUE Algorithm
Read value
Create a new node
If queue is empty:
front = rear = newNode
rear->next = front (circular link)
Else:
rear->next = newNode
rear = newNode
rear->next = front
DEQUEUE Algorithm
If queue empty → Underflow
If only one node:
front = rear = NULL
Else:
Move front = front->next
Update rear->next = front
Free deleted node
```

DISPLAY Algorithm
If queue empty → print message
Start with temp = front
Traverse using:
do { print data; temp = temp->next; } while(temp != front)
Stop when returned to start
3.Repeat until user chooses
4.EXIT

Program:

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

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

struct Node *front = NULL, *rear = NULL;

// ENQUEUE operation
void enqueue() {
    int value;
    printf("Enter value to enqueue: ");
    scanf("%d", &value);

    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;

    if (front == NULL) {     // Queue empty
        front = rear = newNode;
        rear->next = front;  // Circular link
    } else {
        rear->next = newNode;
        rear = newNode;
        rear->next = front;
    }

    printf("Element enqueued.\n");
}

// DEQUEUE operation
void dequeue() {
    if (front == NULL) {
        printf("Queue Underflow.\n");
        return;
    }

    struct Node *temp = front;

    if (front == rear) {  // Only one node
        front = rear = NULL;
    } else {
        front = front->next;
        rear->next = front;  // Maintain circular link
    }

    free(temp);
    printf("Element dequeued.\n");
}

// DISPLAY operation
void display() {
    if (front == NULL) {
        printf("Queue is empty.\n");
        return;
    }

    struct Node *temp = front;

    printf("Circular Queue elements: ");
    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != front);

    printf("(back to front)\n");
}

int main() {
    int choice;

    while (1) {
        printf("\n--- CIRCULAR QUEUE USING LINKED LIST ---\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1: enqueue(); break;
            case 2: dequeue(); break;
            case 3: display(); break;
            case 4: return 0;
            default: printf("Invalid choice!\n");
        }
    }
}
```

Output:

```
--- CIRCULAR QUEUE USING LINKED LIST ---
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter choice: 1
Enter value to enqueue: 10
Element enqueued.

Enter choice: 1
Enter value to enqueue: 20
Element enqueued.

Enter choice: 3
Circular Queue elements: 10 -> 20 -> (back to front)

Enter choice: 2
Element dequeued.

Enter choice: 3
Circular Queue elements: 20 -> (back to front)

Enter choice: 4
```

Result:
The program successfully implements a Circular Queue using a Linked List.
It performs enqueue, dequeue, and display efficiently, while maintaining the circular connection from rear to front.Aim:

To implement Circular Queue operations such as enqueue, dequeue, and display using a linked list, where the last node links back to the first node.

Algorithm:

```
1. Start the program
2. Define a NODE structure
data
next pointer
ENQUEUE Algorithm
Read value
Create a new node
If queue is empty:
front = rear = newNode
rear->next = front (circular link)
Else:
rear->next = newNode
rear = newNode
rear->next = front
DEQUEUE Algorithm
If queue empty → Underflow
If only one node:
front = rear = NULL
Else:
Move front = front->next
Update rear->next = front
Free deleted node
DISPLAY Algorithm
If queue empty → print message
Start with temp = front
Traverse using:
do { print data; temp = temp->next; } while(temp != front)
Stop when returned to start
3.Repeat until user chooses
4.EXIT
```

Program:

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

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

struct Node *front = NULL, *rear = NULL;

// ENQUEUE operation
void enqueue() {
    int value;
    printf("Enter value to enqueue: ");
    scanf("%d", &value);

    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;

    if (front == NULL) {     // Queue empty
        front = rear = newNode;
        rear->next = front;  // Circular link
    } else {
        rear->next = newNode;
        rear = newNode;
        rear->next = front;
    }

    printf("Element enqueued.\n");
}

// DEQUEUE operation
void dequeue() {
    if (front == NULL) {
        printf("Queue Underflow.\n");
        return;
    }

    struct Node *temp = front;

    if (front == rear) {  // Only one node
        front = rear = NULL;
    } else {
        front = front->next;
        rear->next = front;  // Maintain circular link
    }

    free(temp);
    printf("Element dequeued.\n");
}
```

```
// DISPLAY operation
void display() {
    if (front == NULL) {
        printf("Queue is empty.\n");
        return;
    }

    struct Node *temp = front;

    printf("Circular Queue elements: ");
    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != front);

    printf("(back to front)\n");
}

int main() {
    int choice;

    while (1) {
        printf("\n--- CIRCULAR QUEUE USING LINKED LIST ---\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1: enqueue(); break;
            case 2: dequeue(); break;
            case 3: display(); break;
            case 4: return 0;
            default: printf("Invalid choice!\n");
        }
    }
}
```

Output:

```
--- CIRCULAR QUEUE USING LINKED LIST ---
1. Enqueue
```

```
2. Dequeue
3. Display
4. Exit
Enter choice: 1
Enter value to enqueue: 10
Element enqueued.

Enter choice: 1
Enter value to enqueue: 20
Element enqueued.

Enter choice: 3
Circular Queue elements: 10 -> 20 -> (back to front)

Enter choice: 2
Element dequeued.

Enter choice: 3
Circular Queue elements: 20 -> (back to front)

Enter choice: 4
```

Result:
The program successfully implements a Circular Queue using a Linked List.
It performs enqueue, dequeue, and display efficiently, while maintaining the circular connection from rear to front.