Experiment 1

Aim: Write a C program that uses functions to perform the following operations on singly linked list.: i) Creation ii) Insertion iii) Deletion iv) Traversal

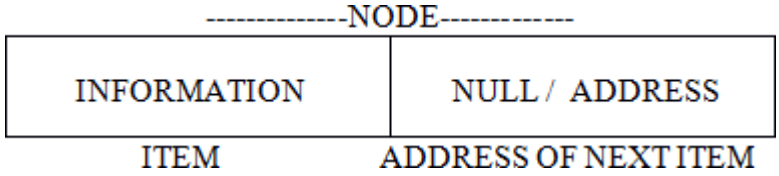
a. *Create a list with single node*
b. *Insert a node at the beginning of the List*
c. *Insert a node at the end of the List*
d. *Insert a node at a given position of the List*
e. *Delete a node at the beginning of the List*
f. *Delete a node at the end of the List*
g. *Delete a node at a given position of the List*
h. *Reverse the List*

Objectives:

- To understand and implement the concept of linked list data structure
- To design, implement, and analyze singly linked list
- To understand different operations on singly linked list
- To understand advantages and disadvantages of using different linked lists

Linked List:

It is a linear collection of data elements called nodes, and node allocate space for each element separately, it contains two fields. The first part contains the information of the element; second part contains the address of next node in the linked list



.

Representation of Linked List using structure:

```
struct node
{
        int  item;
        struct node *next;
}*start;
```

Here start is a node type pointer variable. It can hold the address of another pointer variable. It occupies 2 bytes of memory to point the address of 1^{st} node.

Allocate required memory dynamically for the node

sruct node *temp
temp=(struct node*)malloc(sizeof(struct node));

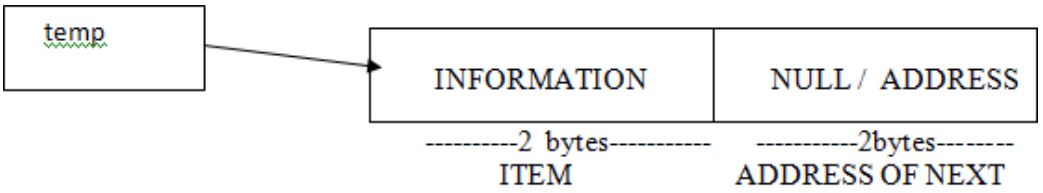
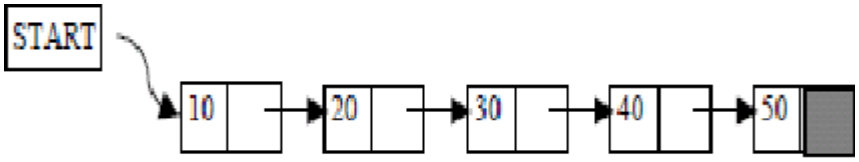


Fig: Example representation of linked list:

temp is a node type pointer variable it can points the dynamically allocated memory i.e. 4 bytes can be accessed by using the following notation.

temp -> item = data;
temp -> next = NULL/ ADDRESS OF NEXT NODE.



Output → 10, 20, 30, 40, 50
Traversing the nodes from left to right

Linked list have many advantages and some of them are:

1. Linked list are dynamic data structure. That is, they can grow or shrink during the execution of a program.

2. Efficient memory utilization: In linked list (or dynamic) representation, memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated (or removed) when it is not needed.

3. Insertion and deletion are easier and efficient. Linked list provides flexibility in inserting a data item at a specified position and deletion of a data item from the given position.

4. Many complex applications can be easily carried out with linked list.

Linked list has following disadvantages

1. More memory: to store an integer number, a node with integer data and address field is allocated. That is more memory space is needed.

2. Access to an arbitrary data item is little bit cumbersome and also time consuming.

The primitive operations performed on the linked list are as follows

Creation operation is used to create a linked list. Once a linked list is created with one node, insertion operation can be used to add more elements in a node.

Insertion operation is used to insert a new node at any specified location in the linked list. A new node may be inserted.

(a) At the beginning of the linked list

(b) At the end of the linked list

(c) At any specified position in between in a linked list

Deletion operation is used to delete an item (or node) from the linked list. A node may be deleted from the

(a) Beginning of a linked list

(b) End of a linked list

(c) Specified location of the linked list

Traversing is the process of going through all the nodes from one end to another end of a linked list. In a singly linked list we can visit from left to right, forward traversing, nodes only. But in doubly linked list forward and backward traversing is possible.

Concatenation is the process of appending the second list to the end of the first list. Consider a list A having n nodes and B with m nodes. Then the operation concatenation will place the 1st node of B in the (n+1)th node in A. After concatenation A will contain

(n+m) nodes

Reverse order of the elements present in the linked list is left to right. Just change this order from right to left will give the all nodes in reverse of the linked list.

Sort – using any one of the sorting method we arranged the all elements present in the linked list are in ascending order.

Types of Linked Lists:

Basically we can divide the linked list into the following three types

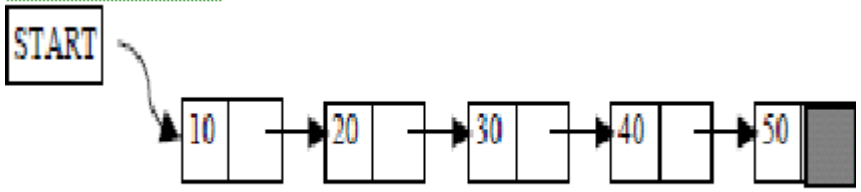
1. Singly linked list
2. Doubly linked list
3. Circular linked list

Singly linked list

All the nodes in a singly linked list are arranged sequentially by linking with a pointer. A singly linked list can grow or shrink, because it is a dynamic data structure.

Following figure explains the different operations on a singly linked list.

Example figure: single linked list:



By traversing the nodes: output- 10, 20, 30, 40, 50

Operations on Linked List:

Insertion

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

Step 1: Create a **newNode** with given value.
Step 2: Check whether list is **Empty** (**head** == **NULL**)
Step 3: If it is **Empty** then, set **newNode→next** = **NULL** and **head** = **newNode**.
Step 4: If it is **Not Empty** then, set **newNode→next** = **head** and **head** = **newNode**.

Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

Step 1: Create a **newNode** with given value and **newNode → next** as **NULL**.
Step 2: Check whether list is **Empty** (**head** == **NULL**).
Step 3: If it is **Empty** then, set **head** = **newNode**.
Step 4: If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
Step 5: Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
Step 6: Set **temp → next** = **newNode**.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list...

Step 1: Create a **newNode** with given value.
Step 2: Check whether list is **Empty** (**head** == **NULL**)
Step 3: If it is **Empty** then, set **newNode → next** = **NULL** and **head** = **newNode**.
Step 4: If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
Step 5: Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
Step 6: Every time check whether **temp** is reached to last node or not. If it is reached to last node then display '**Given node is not found in the list!!! Insertion not possible!!!**' and terminate the function. Otherwise move the **temp** to next node.
Step 7: Finally, Set '**newNode → next** = **temp → next**' and '**temp → next** = **newNode**'

Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

Step 1: Check whether list is **Empty** (**head** == **NULL**)
Step 2: If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
Step 3: If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
Step 4: Check whether list is having only one node (**temp → next** == **NULL**)
Step 5: If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)
Step 6: If it is **FALSE** then set **head** = **temp → next**, and delete **temp**.

Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list...

Step 1: Check whether list is **Empty** (**head** == **NULL**)
Step 2: If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
Step 3: If it is **Not Empty** then, define two Node pointers **'temp1'** and **'temp2'** and initialize '**temp1**' with **head**.
Step 4: Check whether list has only one Node (**temp1 → next** == **NULL**)
Step 5: If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)
Step 6: If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1 → next** == **NULL**)
Step 7: Finally, Set **temp2 → next** = **NULL** and delete **temp1**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list...

Step 1: Check whether list is **Empty** (**head** == **NULL**)
Step 2: If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
Step 3: If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
Step 4: Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
Step 5: If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
Step 6: If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
Step 7: If list has only one node and that is the node to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
Step 8: If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).
Step 9: If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.

Step 10: If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == NULL**).
Step 11: If **temp1** is last node then set **temp2 → next** = **NULL** and delete **temp1** (**free(temp1)**).
Step 12: If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1**(**free(temp1)**).

Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list...

Step 1: Check whether list is **Empty** (**head** == **NULL**)
Step 2: If it is **Empty** then, display **'List is Empty!!!'** and terminate the function.
Step 3: If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
Step 4: Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node
Step 5: Finally display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

SLL CODE:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
	int data;
	struct node *next;
};
struct node* head=NULL;
void printList(struct node* n)
{
	int i=0;
	if(n==NULL)
	{
		printf("List Is Empty");
	}
	while (n != NULL)
	{
	  printf("Data at Node %d: %d\n",i, n->data);
	  n = n->next;
	  i++;
	}
}
void createList(int n)
{
	struct node *newNode;
	struct node *temp;
	int data,i,data1;
	head=(struct node*)malloc(sizeof(struct node));
	if(head==NULL)
	{
		printf("\nUnable to create HeaderNode\n");
	}
```

```
        else
{
                printf("\nEnter Data for Node 0: ");
                scanf("%d",&data);
                head->data=data;
                head->next=NULL;
                temp=head;
                for(i=1;i<n;i++)
                {
                        newNode=(struct node*)malloc(sizeof(struct node));
                        printf("\nEnter Data for Node %d: ",i);
                        scanf("%d",&data1);
                        newNode->data=data1;
                        newNode->next=NULL;
                        temp->next=newNode;
                        temp=temp->next;
                }
        }
        printList(head);
}
void insertAtB(int data)
{
        struct node *newNode;
   newNode = (struct node*)malloc(sizeof(struct node));
   if(newNode == NULL)
   {
      printf("Unable to allocate memory.");
   }
   else
   {
      newNode->data = data; // Link data part
      newNode->next = head; // Link address part
      head = newNode;          // Make newNode as first node
      printf("DATA INSERTED SUCCESSFULLY\n");
   }
}
void insertAtE(int data)
{
        struct node *newNode,*temp;
        newNode = (struct node*)malloc(sizeof(struct node));
   if(newNode == NULL)
   {
      printf("Unable to allocate memory.");
   }
   else
   {
      newNode->data = data; // Link data part
      newNode->next = NULL; // Link address part
      temp = head;          // Make newNode as first node
      while(temp->next != NULL)
```

```
        temp=temp->next;
        temp->next=newNode;
        printf("DATA INSERTED SUCCESSFULLY AT END OF THE LIST\n");
    }
}
void insertAtM(int pos,int data)
{
        struct node *newNode,*temp;
        int i;
    newNode = (struct node*)malloc(sizeof(struct node));
            if(newNode == NULL)
    {
        printf("Unable to allocate memory.");
    }
    else
    {
        if(pos==0)
        {
                insertAtB(data);
        }else{
                newNode->data = data; // Link data part
                newNode->next = NULL; // Link address part
        temp = head;          // Make newNode as first node
        for(i=0;i<pos;i++)
        {
        temp=temp->next;
        if(temp == NULL)
        {
                break;
        }
        }
        if(temp!= NULL)
        {
        newNode->next=temp->next;
        temp->next=newNode;
         printf("DATA INSERTED SUCCESSFULLY\n");
        }else{
        printf("Not Possible/Empty List");
        }
        }
    }
}
void deletionAtB()
{
        struct node *delNode,*p;
    int c=0;
                if(head==NULL)
                {
                        printf("Empty List");
                }
```

```
		else
		{
			delNode= head;
		head=head->next;
		free(delNode);
		printf("DATA DELETED SUCCESSFULLY\n");
		}
}
void deletionAtE()
{
	struct node *temp, *temp1;
	if(head== NULL)
	{
		printf("List Already Null\n");
	}
	else
	{
		temp=head;
		temp1=NULL;
		while(temp->next !=NULL)
		{
			temp1=temp;
			temp=temp->next;
		}
		if(temp1!=NULL)
		{
			temp1->next=NULL;
		}
		if(temp==head)
			head=NULL;
		free(temp);
		printf("Deleted\n");
	}
}
void deletionAtM(int pos)
{
	struct node *prev=head;
	struct node *temp=head;
	struct node *next=head;
	struct node *p=head;
	int c=0;
	while(p!=NULL)
	{
		p=p->next;
		c++;
	}
	if(pos==0 && c==0)
		{
			if(head==NULL){
				printf("EmptyList");
```

```
                        }
                        else
                        {
                        head=temp->next;
                        free(temp);
                        printf("Deleted");
                        }
                }
        if(pos < c)
        {
                if(pos==0)
                {
                        if(head==NULL){
                                printf("EmptyList");
                        }
                        else
                        {
                        head=temp->next;
                        free(temp);
                        printf("Deleted");
                        }
                }
                else
                {
                // Find previous node of the node to be deleted
                 for (int i=0; prev!=NULL && i<pos-1; i++)
           prev = prev->next;
        //printf("\nPrevious:prev->data :%d\n",prev->data);
                 for (int i=0; next!=NULL && i<pos+1; i++)
           next = next->next;
        //printf("\nNext:next->data :%d\n",next->data);
                prev->next=next;
   // printList(head);
                printf("\n");
                }
                }
                else
                {
                        printf("Position Not Found");
           }
}
void search(int n)
{
        struct node *temp=NULL;
        int i=0;
        if(head->data==n)
        {
                printf("Element found at head position");
        }
        else
```

```
	{
		temp=head;
		while(temp->next!=NULL)
		{
			temp=temp->next;
			i++;
			if(temp->data == n)
			{
				printf("Element found at %d position ",i);
			}

		}
	}
}
void reverse()
{
	struct node *current, *prev,*next;
	current=head;
	prev=NULL;
	while(current!=NULL)
	{
		next=current->next;
		current->next=prev;
		prev=current;
		current=next;
	}
	head=prev;
}
int main()
{
	int choice,n,data,pos;
	while(1)
	{
		printf("\n0.Exit\n1. Create a List\n2. Insert Node at Beginning\n3. Insert Node at Ending of the List\n");
		printf("4. Delete a Node at Beginning of the List\n5. Delete a Node at Ending of the List\n");
		printf("6. Insert a Node\n7. Delete a Node\n8. Search a Element\n9. Reverse List\n10. Display\n");
		printf("Enter your Choice:\t");
		while(scanf("%d",&choice)==0)
		{
			printf("Only Numbers!!!");
			int c;
			while((c=getchar())!='\n' && c!=EOF); //clear the stdin
		}
		switch(choice)
		{
			case 0:
				exit(0);
```

```
case 1:
        printf("\nEnter Size of the List:\t");
        while(scanf("%d",&n)==0)
        {
                printf("Only Numbers!!!");
                int c;
                while((c=getchar())!='\n' && c!=EOF); //clear the stdin
        }
        if(n > 0)
        {
                createList(n);
        }
        else
        {
                printf("Size in +ve, not -ve");
        }
        break;
case 2:
        printf("Enter Data to add at beginning of the List: ");
        while(scanf("%d",&data)==0)
        {
                printf("Only Numbers!!!");
                int c;
                while((c=getchar())!='\n' && c!=EOF); //clear the stdin
        }
        insertAtB(data);
        break;
case 3:
        printf("Enter Data to add at Ending of the List: ");
        while(scanf("%d",&data)==0)
        {
                printf("Only Numbers!!!");
                int c;
                while((c=getchar())!='\n' && c!=EOF); //clear the stdin
        }
        insertAtE(data);
        break;
case 4:
        deletionAtB();
        break;
case 5:
        deletionAtE();
        break;
case 6:
        printf("Enter Data Position to Insert element: ");
        while(scanf("%d",&pos)==0)
        {
                printf("Only Numbers!!!");
                int c;
                while((c=getchar())!='\n' && c!=EOF); //clear the stdin
```

```
		}
		if(pos >= 0)
		{
			printf("Enter Data to Insert in List: ");
			while(scanf("%d",&data)==0)
			{
				printf("Only Numbers!!!");
				int c;
				while((c=getchar())!='\n' && c!=EOF); //clear the stdin
			}
			insertAtM(pos,data);
		}
		else
		{
				printf("No -ve postions\n");
				exit(0);
			}
		break;
case 7:
		printf("Enter Data Position to Delete element: ");
		while(scanf("%d",&pos)==0)
		{
			printf("Only Numbers!!!");
			int c;
			while((c=getchar())!='\n' && c!=EOF); //clear the stdin
		}
		if(pos >= 0)
		{
			deletionAtM(pos);
		}
		else
		{
				printf("No -ve postions\n");
				exit(0);
			}
		break;
case 8:
		printf("\nEnter Element to Search:\t");
		while(scanf("%d",&data)==0)
		{
			printf("Only Numbers!!!");
			int c;
			while((c=getchar())!='\n' && c!=EOF); //clear the stdin
		}
		search(data);
		break;
case 9:
		reverse();
		break;
case 10:
```

```
                printList(head);
                break;
            }
        }
        return 0;
}
```

OUTPUT

Experiment 2

Aim: Write a program that uses functions to perform the following operations on doubly linked list.: i) Creation ii) Insertion iii) Deletion iv) Traversal

Objectives:

- To understand and implement the concept of linked list data structure
- To design, implement, and analyze doubly linked lists
- To understand different operations on doubly linked lists
- To understand advantages and disadvantages of using different linked lists

Insertion

In a double linked list, the insertion operation can be performed in three ways as follows...

1. **Inserting At Beginning of the list**
2. **Inserting At End of the list**
3. **Inserting At Specific location in the list**

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list...

Step 1: Create a newNode with given value and newNode → previous as NULL.
Step 2: Check whether list is Empty (head == NULL)
Step 3: If it is Empty then, assign NULL to newNode → next and newNode to head.
Step 4: If it is not Empty then, assign head to newNode → next and newNode to head.

Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list…

Step 1: Create a newNode with given value and newNode → next as NULL.
Step 2: Check whether list is Empty (head == NULL)
Step 3: If it is Empty, then assign NULL to newNode → previous and newNode to head.
Step 4: If it is not Empty, then, define a node pointer temp and initialize with head.
Step 5: Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).
Step 6: Assign newNode to temp → next and temp to newNode → previous.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the double linked list…

Step 1: Create a newNode with given value.

Step 2: Check whether list is Empty (head == NULL)

Step 3: If it is Empty then, assign NULL to newNode → previous & newNode → next and newNode to head.

Step 4: If it is not Empty then, define two node pointers temp1 & temp2 and initialize temp1 with head.

Step 5: Keep moving the temp1 to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).

Step 6: Every time check whether temp1 is reached to the last node. If it is reached to the last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp1 to next node.

Step 7: Assign temp1 → next to temp2, newNode to temp1 → next, temp1 to newNode → previous, temp2 to newNode → next and newNode to temp2 → previous.

Deletion

In a double linked list, the deletion operation can be performed in three ways as follows…

1. **Deleting from Beginning of the list**
2. **Deleting from End of the list**
3. **Deleting a Specific Node**

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the double linked list…

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3: If it is not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4: Check whether list is having only one node (temp → previous is equal to temp → next)

Step 5: If it is TRUE, then set head to NULL and delete temp (Setting Empty list conditions)

Step 6: If it is FALSE, then assign temp → next to head, NULL to head → previous and delete temp.

Deleting from End of the list

We can use the following steps to delete a node from end of the double linked list…

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty, then display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3: If it is not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4: Check whether list has only one Node (temp → previous and temp → next both are NULL)

Step 5: If it is TRUE, then assign NULL to head and delete temp. And terminate from the function. (Setting Empty list condition)

Step 6: If it is FALSE, then keep moving temp until it reaches to the last node in the list. (until temp → next is equal to NULL)

Step 7: Assign NULL to temp → previous → next and delete temp.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the double linked list…

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3: If it is not Empty, then define a Node pointer 'temp' and initialize with head.

Step 4: Keep moving the temp until it reaches to the exact node to be deleted or to the last node.

Step 5: If it is reached to the last node, then display 'Given node not found in the list! Deletion not possible!!!' and terminate the fuction.

Step 6: If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7: If list has only one node and that is the node which is to be deleted then set head to NULL and delete temp(free(temp)).

Step 8: If list contains multiple nodes, then check whether temp is the first node in the list (temp == head).

Step 9: If temp is the first node, then move the head to the next node (head = head → next), set head of previous to NULL(head → previous = NULL) and delete temp.

Step 10: If temp is not the first node, then check whether it is the last node in the list (temp → next == NULL).

Step11: If temp is the last node then set temp of previous of next to NULL (temp → previous → next = NULL) and delete temp (free(temp)).

Step 12: If temp is not the first node and not the last node, then set temp of previous of next to temp of next (temp → previous → next =

temp → next), temp of next of previous to temp of previous (temp → next → previous = temp → previous) and delete temp (free(temp)).

Displaying a Double Linked List

We can use the following steps to display the elements of a double linked list…

Step 1: Check whether list is Empty (head == NULL)

Step 2: If it is Empty, then display 'List is Empty!!!' and terminate the function.

Step 3: If it is not Empty, then define a Node pointer 'temp' and initialize with head.

Step 4: Display 'NULL <- '.

Step 5: Keep displaying temp → data with an arrow (<=) until temp reaches to the last node

Step 6: Finally, display temp → data with arrow pointing to NULL (temp → data -→ NULL).

DLL CODE

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

struct node
{
	int data;
	struct node *next;
	struct node *prev;
};
	struct node *head=NULL;
void printList(struct node* n)
{
	int i=0;
	if(n==NULL)
	{
		printf("List Is Empty");
	}
	while (n != NULL)
	{
	  printf("Data at Node %d: %d\n",i, n->data);
	 n = n->next;
	 i++;
	}
}
void createList(int n)
{
```

```
        struct node* newNode;
        struct node* temp;
        int data,i,data1;
        //int n=3;

        head=(struct node*)malloc(sizeof(struct node));

        if(head==NULL)
        {
                printf("\nUnable to create HeaderNode\n");
        }else{
                printf("\nEnter Data for Node 0: ");
                scanf("%d",&data);

                head->data=data;
                head->next=NULL;
                head->prev=NULL;

                temp=head;

                for(i=1;i<n;i++)
                {
                        newNode=(struct node*)malloc(sizeof(struct node));

                        printf("\nEnter Data for Node %d: ",i);
                        scanf("%d",&data1);

                        newNode->data=data1;
                        newNode->prev=temp;
                        newNode->next=NULL;

                        temp->next=newNode;
                        temp=temp->next;
                }
        }
}
void insertAtB(int data)
{
        struct node *newNode;

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

    if(newNode == NULL)
```

```
    {
       printf("Unable to allocate memory.");
    }
    else
    {
       newNode->data = data; // Link data part
       newNode->next = head; // Link address part
       newNode->prev = NULL;

       head = newNode;          // Make newNode as first node

       printf("DATA INSERTED SUCCESSFULLY\n");
    }
}
void insertAtE(int data)
{
       struct node *newNode,*temp,*prev;

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

    if(head == NULL)
    {
       insertAtB(data);
    }
    else
    {
       newNode->data = data; // Link data part
       newNode->next = NULL; // Link address part
       newNode->prev = NULL;

        temp = head;          // Make newNode as first node

        while(temp->next != NULL)
        {
          //prev=temp;
                temp=temp->next;
        }
        temp->next=newNode;
        newNode->prev=temp;

       printf("DATA INSERTED SUCCESSFULLY AT END OF THE LIST\n");
    }
}
```

```
void insertAtM(int pos,int data)
{
        struct node *newNode,*temp;
        int i;
    newNode = (struct node*)malloc(sizeof(struct node));

    if(head == NULL)
    {
        insertAtB(data);
    }
    else
    {
        if(pos==0)
        {
                insertAtB(data);
        }else{
                newNode->data = data; // Link data part
                newNode->next = NULL; // Link address part

         temp = head;          // Make newNode as first node
         for(i=0;i<pos;i++)
         {
                temp=temp->next;
                if(temp == NULL)
                {
                        break;
                }
         }
         if(temp!= NULL)
         {
                newNode->next=temp->next;
                temp->next=newNode;
                newNode->prev=temp;
                 printf("DATA INSERTED SUCCESSFULLY\n");
         }else{
                printf("Not Possible/Empty List");
         }
        }


    }
}
void deletionAtB()
```

```
{
	struct node *delNode,*p;
   int c=0;

		if(head==NULL)
		{
			printf("Empty List");
		}else{
			delNode= head;
			if(delNode->prev==delNode->next)
			{
				head=NULL;
				free(delNode);
				return;
			}else{
				head=head->next;
				head->prev=NULL;
		free(delNode);
			}
		printf("DATA DELETED SUCCESSFULLY\n");
		}
}
void deletionAtE()
{
	struct node *temp, *temp1;

	if(head== NULL)
	{
		printf("List Already Null\n");
	}else{
		//temp=head;
		if(temp->prev==NULL && temp->next==NULL)
		{
			head=NULL;
			free(temp);
			return;
		}else{
			printf("F");
			while(temp->next != NULL)
		{
			temp=temp->next;
		}
		temp->prev->next=NULL;
```

```
                free(temp);
                        return;
                }
        }
        printList(head);
}
void deletionAtM(int pos)
{

        struct node *prev=head;
        struct node *temp=head;
        struct node *next=head;
        struct node *p=head;
        int c=0;
        while(p!=NULL)
        {
                p=p->next;
                c++;
        }
        if(pos==0 && c==0)
                {
                        if(head==NULL){
                                printf("EmptyList");
                                return;
                        }else{
                                deletionAtB();
                        }
                }
        if(pos < c)
        {
                if(pos==0)
                {
                        if(head==NULL){
                                printf("EmptyList");
                                return;
                        }else{
                                deletionAtB();
                        }
                }else{
                         for (int i=0; i<pos-1; i++)
                         {
                                temp=temp->next;
                         }
```

```
                if(temp->prev==NULL && temp->next==NULL)
                {
                        head=NULL;
                        free(temp);
                        printf("DATA DELETED");
                        return;
                }else if(temp==head){
                        head=head->next;
                        head->prev=NULL;
                        free(temp);
                        printf("DATA DELETED");
                }else if(temp->next==NULL){
                        deletionAtE();
                }else{
                        temp->prev->next=temp->next;
                        temp->next->prev=temp->prev;
                        free(temp);
                        printf("DATA DELETED");
                }
        }
    }else {
                printf("Position Not Found");
    }
}
void search(int n)
{
    struct node *temp=NULL;
    int i=0;
    if(head==NULL)
    {
        printf("EMPTY LIST");
        return;
    }
    if(head->data==n)
    {
        printf("Element found at head position");
    }else{
        temp=head;
        while(temp->next!=NULL)
        {
            temp=temp->next;
            i++;
            if(temp->data == n)
```

```
                {
                        printf("Element found at %d position ",i);
                }else{
                        printf("Element found");
                }

        }
    }
}
void reverse()
{
    struct node *current, *temp,*next;
    current=head;
    temp=NULL;
    while(current!=NULL)
    {
        temp=current->prev;
        current->prev=current->next;
        current->next=temp;
        current=current->prev;
    }
    if(temp!=NULL)
        head=temp->prev;

}
int main()
{
    int choice,n,data,pos;

    while(1)
    {
        printf("\n0.Exit\n1. Create a List\n2. Insert Node at Beginning\n3. Insert Node at Ending of the List\n");
        printf("4. Delete a Node at Beginning of the List\n5. Delete a Node at Ending of the List\n");
        printf("6. Insert a Node\n7. Delete a Node\n8. Search a Element\n9. Reverse List\n10. Display\n");
        printf("Enter your Choice:\t");
        while(scanf("%d",&choice)==0)
        {
            printf("Only Numbers!!!");
            int c;
            while((c=getchar())!='\n' && c!=EOF); //clear the stdin
```

```
}
switch(choice)
{
	case 0:
		exit(0);
	case 1:
		printf("\nEnter Size of the List:\t");
		while(scanf("%d",&n)==0)
		{
			printf("Only Numbers!!!");
			int c;
			while((c=getchar())!='\n' && c!=EOF); //clear the stdin
		}
		if(n > 0)
		{
			createList(n);
		}else{
			printf("Size in +ve, not -ve");
		}
		break;
	case 2:
		printf("Enter Data to add at beginning of the List: ");
		while(scanf("%d",&data)==0)
		{
			printf("Only Numbers!!!");
			int c;
			while((c=getchar())!='\n' && c!=EOF); //clear the stdin
		}
		insertAtB(data);
		break;
	case 3:
		printf("Enter Data to add at Ending of the List: ");
		while(scanf("%d",&data)==0)
		{
			printf("Only Numbers!!!");
			int c;
			while((c=getchar())!='\n' && c!=EOF); //clear the stdin
		}
		insertAtE(data);
		break;
	case 4:
		deletionAtB();
		break;
```

```
case 5:
        deletionAtE();
        break;
case 6:
        printf("Enter Data Position to Insert element: ");
        while(scanf("%d",&pos)==0)
        {
                printf("Only Numbers!!!");
                int c;
                while((c=getchar())!='\n' && c!=EOF); //clear the stdin
        }
        if(pos >= 0)
        {
                printf("Enter Data to Insert in List: ");
                while(scanf("%d",&data)==0)
                {
                        printf("Only Numbers!!!");
                        int c;
                while((c=getchar())!='\n' && c!=EOF); //clear the stdin
                }
                insertAtM(pos,data);
        }else{
                        printf("No -ve postions\n");
                        exit(0);
                }
        break;
case 7:
        printf("Enter Data Position to Delete element: ");
        while(scanf("%d",&pos)==0)
        {
                printf("Only Numbers!!!");
                int c;
                while((c=getchar())!='\n' && c!=EOF); //clear the stdin
        }
        if(pos >= 0)
        {
                deletionAtM(pos);
        }else{
                        printf("No -ve postions\n");
                        exit(0);
                }
        break;
case 8:
```

```
                        printf("\nEnter Element to Search:\t");
                        while(scanf("%d",&data)==0)
                        {
                                printf("Only Numbers!!!");
                                int c;
                                while((c=getchar())!='\n' && c!=EOF); //clear the stdin
                        }
                        search(data);
                        break;
                case 9:
                        reverse();
                        break;
                case 10:
                        printList(head);
                        break;
        }
    }
    return 0;
}
```

OUTPUT

Experiment 3

Aim:Write a program that uses functions to perform the following operations on circular linked list.: i) Creation ii) Insertion iii) Deletion iv) Traversal

Objectives:

- To understand and implement the concept of linked list data structure
- To design, implement, and analyze circular linked lists
- To understand different operations on circular linked lists
- To understand advantages and disadvantages of using different linked lists

Circular Singly Linked List

In a circular linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

Step 1: Include all the **header files** which are used in the program.
Step 2: Declare all the **user defined** functions.
Step 3: Define a **Node** structure with two members **data** and **next**
Step 4: Define a Node pointer '**head**' and set it to **NULL**.
Step 4: Implement the **main** method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the circular linked list...

Step 1: Create a **newNode** with given value.
Step 2: Check whether list is **Empty** (**head** == **NULL**)
Step 3: If it is **Empty** then,
set **head** = **newNode** and **newNode→next** = **head** .
Step 4: If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.
Step 5: Keep moving the '**temp**' to its next node until it reaches to the last

node (until '**temp → next == head**').

Step 6: Set '**newNode → next =head**', '**head = newNode**' and '**temp → next = head**'.

Inserting At End of the list

We can use the following steps to insert a new node at end of the circular linked list...

Step 1: Create a **newNode** with given value.

Step 2: Check whether list is **Empty** (**head == NULL**).

Step 3: If it is **Empty** then, set **head = newNode** and **newNode → next = head**.

Step 4: If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Step 5: Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next == head**).

Step 6: Set **temp → next = newNode** and **newNode → next = head**.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the circular linked list...

Step 1: Create a **newNode** with given value.

Step 2: Check whether list is **Empty** (**head == NULL**)

Step 3: If it is **Empty** then, set **head = newNode** and **newNode → next = head**.

Step 4: If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Step 5: Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).

Step 6: Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.

Step 7: If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).

Step 8: If **temp** is last node then set **temp → next = newNode** and **newNode → next = head**.

Step 8: If **temp** is not last node then set **newNode → next = temp → next** and **temp → next = newNode**.

Deletion

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the circular linked list...

Step 1: Check whether list is **Empty** (**head** == **NULL**)
Step 2: If it is **Empty** then, display '**List is Empty!!! Deletion is not possible'** and terminate the function.
Step 3: If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.
Step 4: Check whether list is having only one node (**temp1** → **next** == **head**)
Step 5: If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)
Step 6: If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1** → **next** == **head**)
Step 7: Then set **head** = **temp2** → **next**, **temp1** → **next** = **head** and delete **temp2**.

Deleting from End of the list

We can use the following steps to delete a node from end of the circular linked list...

Step 1: Check whether list is **Empty** (**head** == **NULL**)
Step 2: If it is **Empty** then, display '**List is Empty!!! Deletion is not possible'** and terminate the function.
Step 3: If it is **Not Empty** then, define two Node pointers '**temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
Step 4: Check whether list has only one Node (**temp1** → **next** == **head**)
Step 5: If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)
Step 6: If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1** → **next** == **head**)
Step 7: Set **temp2** → **next** = **head** and delete **temp1**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the circular linked list...

Step 1: Check whether list is **Empty** (**head** == **NULL**)

Step 2: If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3: If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.

Step 4: Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.

Step 5: If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.

Step 6: If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next** == **head**)

Step 7: If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).

Step 8: If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).

Step 9: If **temp1** is the first node then set **temp2** = **head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → nex**t = **head** and delete **temp1**.

Step 10: If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).

Step 11: If **temp1** is last node then set **temp2 → next** = **head** and delete **temp1** (**free(temp1)**).

Step 12: If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1**(**free(temp1)**).

Displaying a circular Linked List

We can use the following steps to display the elements of a circular linked list...

Step 1: Check whether list is **Empty** (**head** == **NULL**)

Step 2: If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.

Step 3: If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.

Step 4: Keep displaying **temp → data** with an arrow (**--->**)
until **temp** reaches to the last node

Step 5: Finally display **temp → data** with arrow pointing to **head → data**.

CLL Code:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int data;
        struct node *next;
};

struct node *head=NULL;
int getCount()
{
   int i=0;
   struct node* n;
   n=head;
        if(head==NULL)
        {
                return i;
        }else{
        do
        {
          n = n->next;
          i++;
        }while (n->next != head->next);
        }
   return i;
}
void printList(struct node* n)
{
        int i=0;
        if(head==NULL)
        {
                printf("EMPTY LIST");
        }else{
        do
        {
           printf("Node[%d]: %d\n",i, n->data);
          n = n->next;
          i++;
        }while (n->next != head->next);
        }
}
void createList(int n)
```

```
{

        struct node* newNode;
        struct node* temp;
        int data,i,data1;
        head=(struct node*)malloc(sizeof(struct node));
        if(head==NULL)
        {
                printf("\nUnable to create HeaderNode\n");
        }else{
                printf("\nEnter Data for Node 0: ");
                scanf("%d",&data);

                head->data=data;
                head->next=head;
                temp=head;
                for(i=1;i<n;i++)
                {
                        newNode=(struct node*)malloc(sizeof(struct node));

                        printf("\nEnter Data for Node %d: ",i);
                        scanf("%d",&data1);

                        newNode->data=data1;
                        newNode->next=head;

                        temp->next=newNode;
                        temp=temp->next;
                }
        }
}
void insertAtB(int data)
{
        struct node *newNode, *temp;

   if(head == NULL)
   {
      head = (struct node*)malloc(sizeof(struct node));
                head->data=data;
```

```
                head->next=head;
    }
    else
    {
        temp=head;
        while(temp->next!=head)
        {
                temp=temp->next;
        }
        newNode = (struct node*)malloc(sizeof(struct node));

       newNode->data = data; // Link data part
       newNode->next = head; // Link address part
       head=newNode;
       temp->next=head;
       printf("DATA INSERTED SUCCESSFULLY\n");
    }

}
void insertAtE(int data)
{
        struct node *newNode, *temp;

    if(head == NULL)
    {
       head = (struct node*)malloc(sizeof(struct node));
                head->data=data;
                head->next=head;
    }
    else
    {
        temp=head;
        while(temp->next!=head)
        {
                temp=temp->next;
                printf("%d",temp->data);
        }
        newNode = (struct node*)malloc(sizeof(struct node));
        temp->next=newNode;
       newNode->data = data; // Link data part
       newNode->next = head; // Link address part

       printf("DATA INSERTED SUCCESSFULLY\n");
```

```
    }

}
void insertAtM(int pos,int data)
{
        struct node *newNode, *temp,*temp1;
        int i=0,count=1;
        //printf("%d Count\n",count);

        if(head==NULL)
        {
                head = (struct node*)malloc(sizeof(struct node));
                head->data=data;
                head->next=head;
        }else{
        temp1=head;
        temp = head;
        while(temp1->next!=head)
        {
                count++;
                temp1=temp1->next;
        }
                if(pos==0)
                {
                        insertAtB(data);
                }else if(pos==count)
                {
                        insertAtE(data);
                }else if(pos > count)
                {
                        printf("Position Not Found");
                }else{
       for(i=1;i<=pos-1;i++)
       {
        temp=temp->next;
       }
       newNode=(struct node*)malloc(sizeof(struct node));
       newNode->next=temp->next;
       temp->next=newNode;
       newNode->data=data;
       printf("DATA INSERTED SUCCESSFULLY\n");
                }
        }
```

```
}
void deletionAtB()
{
        struct node *nodeToDelete,*current;
   if(head==NULL)
   {
        printf("Empty List");
   }else
   {
                nodeToDelete=head;
                current=head;
                if(current->next == head){
                        printf("List is Empty");
                        head=NULL;
                        //exit(0);
                }
                else{
                        while(current->next!=head)
                        {
                                current=current->next;
                        }
                        head=nodeToDelete->next;
                        current->next=head;
                        free(nodeToDelete);
                }
   }
}
void deletionAtE()
{
        struct node *temp,*temp1;
        if(head==NULL)
        {
                printf("Empty LIST");
        }else{
                temp1=head;
                if(temp1->next==head)
                {
                        head=NULL;
                }else{
                        while(temp1->next!=head)
                        {
                                temp=temp1;
                                temp1=temp1->next;
```

```
                }
                temp->next=head;
                free(temp1);
            }

        }
}
void deletionAtM(int pos)
{

        struct node *prev=head;
        struct node *temp=head;
        struct node *next=head;
        struct node *p=head;
        int c=getCount();
        int i;

        if(pos==0 && c==0)
        {
            deletionAtB();
        }else if(pos==c){
            deletionAtE();
        }
        if(pos < c)
        {
            if(pos==0)
            {
                deletionAtB();
            }else{
                for(i=0;i<=pos-1;i++)
                {
                    prev=temp;
                    temp=temp->next;
            //      j++;
                }
                prev->next=temp->next;
                free(temp);
            //printf("%d\t%d",pos,j);
            }
        }else {
                printf("Position Not Found");
        }
}
```

```
void search(int n)
{
	struct node *temp=NULL;
	int i=0;
	if(head==NULL)
	{
		printf("EMPTY LIST");
	}else{
		if(head->data==n)
		{
			printf("Element found at head position");
		}else{
			temp=head;
			while(temp->next!=head)
			{
				temp=temp->next;
				i++;
				if(temp->data == n)
				{
					printf("Element found at %d position ",i);
				}
			}
		}
	}
}
void reverse()
{
	 if (head == NULL)
	return;

   struct node* prev = NULL;
   struct node* current = head;
   struct node* next;
   do {
      next = current->next;
      current->next = prev;
      prev = current;
      current = next;
   } while (current != head);
   head->next = prev;
   head = prev;
```

```
    printList(head);
}
int main()
{
        int choice,n,data,pos;
        int count=0;

        while(1)
        {
                printf("\n0.Exit\n1. Create a List\n2. Insert Node at Beginning\n3. Insert Node at Ending of the List\n");
                printf("4. Delete a Node at Beginning of the List\n5. Delete a Node at Ending of the List\n");
                printf("6. Insert a Node\n7. Delete a Node\n8. Search a Element\n9. Reverse List\n10. Display\n");
                printf("Enter your Choice:\t");
                while(scanf("%d",&choice)==0)
                {
                        printf("Only Numbers!!!");
                        int c;
                        while((c=getchar())!='\n' && c!=EOF); //clear the stdin
                }
                switch(choice)
                {
                        case 0:
                                exit(0);
                        case 1:
                                printf("\nEnter Size of the List:\t");
                                while(scanf("%d",&n)==0)
                                {
                                        printf("Only Numbers!!!");
                                        int c;
                                while((c=getchar())!='\n' && c!=EOF); //clear the stdin
                                }
                                if(n > 0)
                                {
                                        createList(n);
                                }else{
                                        printf("Size in +ve, not -ve");
                                }
                                break;
                        case 2:
                                printf("Enter Data to add at beginning of the List: ");
```

```
        while(scanf("%d",&data)==0)
        {
                printf("Only Numbers!!!");
                int c;
        while((c=getchar())!='\n' && c!=EOF); //clear the stdin
        }
        insertAtB(data);
        break;
case 3:
        printf("Enter Data to add at Ending of the List: ");
        while(scanf("%d",&data)==0)
        {
                printf("Only Numbers!!!");
                int c;
        while((c=getchar())!='\n' && c!=EOF); //clear the stdin
        }
        insertAtE(data);
        break;
case 4:
        deletionAtB();
        break;
case 5:
        deletionAtE();
        break;
case 6:
        printf("Enter Data Position to Insert element: ");
        while(scanf("%d",&pos)==0)
        {
                printf("Only Numbers!!!");
                int c;
        while((c=getchar())!='\n' && c!=EOF); //clear the stdin
        }
        if(n >= pos)
        {
                printf("Enter Data to Insert in List: ");
                while(scanf("%d",&data)==0)
                {
                        printf("Only Numbers!!!");
                        int c;
        while((c=getchar())!='\n' && c!=EOF); //clear the stdin
                }
                insertAtM(pos,data);
        }else{
```

```
                        printf("No -ve postions/Not found\n");
                }
        break;
case 7:
        printf("Enter Position to Delete element: ");
        while(scanf("%d",&pos)==0)
        {
                printf("Only Numbers!!!");
                int c;
        while((c=getchar())!='\n' && c!=EOF); //clear the stdin
        }
        if(pos >= 0)
        {
                deletionAtM(pos);
        }else{
                printf("No -ve postions/Position Not found\n");

                }
        break;
case 8:
        printf("\nEnter Element to Search:\t");
        while(scanf("%d",&data)==0)
        {
                printf("Only Numbers!!!");
                int c;
        while((c=getchar())!='\n' && c!=EOF); //clear the stdin
        }
        search(data);
        break;
case 9:
        reverse();
        break;
case 10:
        printList(head);
        break;
case 11:
        count=getCount();
        printf("%d",count);
        break;
default:
        printf("Wrong Entry");
        break;
```

```
                }
        }
        return 0;
}
```

OUTPUT

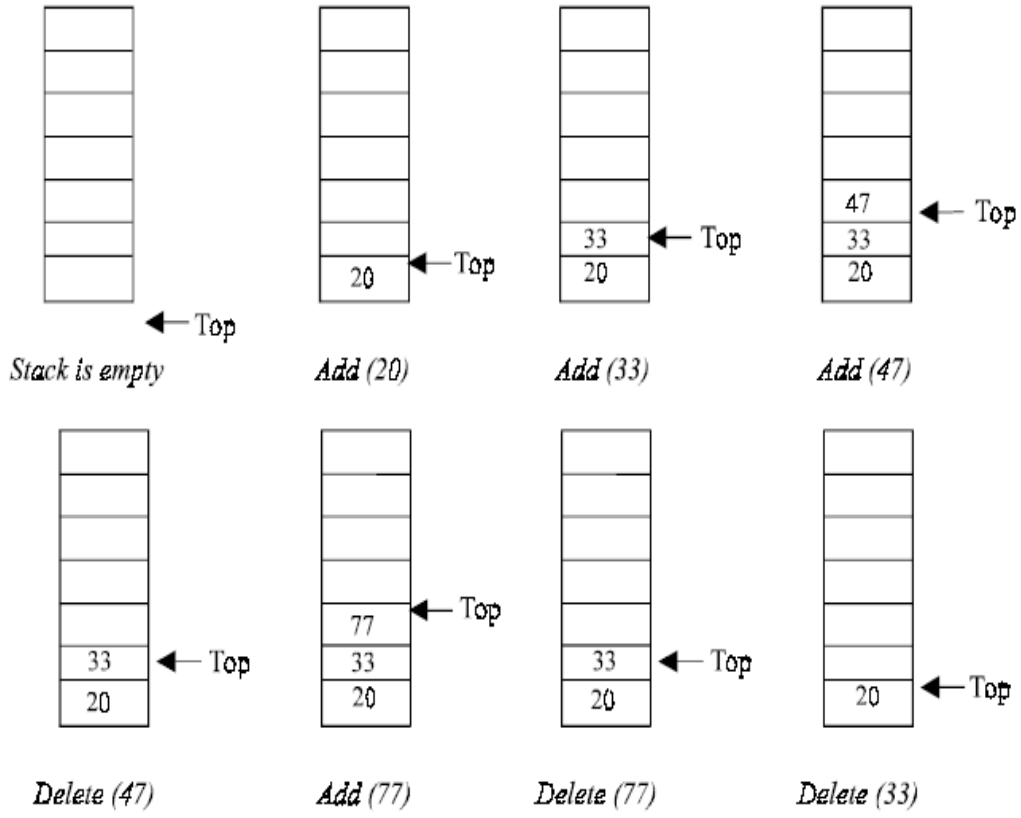
Experiment 4

Aim: Write a program that implement stack (its operations) using i) Arrays ii) Pointers

Objectives:

- To understand and implement the stack data structure
- To design, implement, and analyze stack implanted using array and linked list
- To understand different operations on stacks
- To understand the implementation of stack operations using array and linked list

Stack is a non primitive linear data structure in which a new data element may be added/inserted and deleted at only one end, called top of the stack. the last added element will be first removed from the stack. That is why the stack is also called ***Last-in-First-out (LIFO)***. Note that the most frequently accessible element in the stack is the top most elements, whereas the least accessible element is the bottom of the stack.
The operation of the stack can be illustrated in the following figure:



. Fig 1: **Stack operations**

OPERATIONS PERFORMED ON STACK

The primitive operations performed on the stack are as follows:

PUSH: The process of adding (or inserting) a new element to the top of the stack is called PUSH operation. Pushing an element to a stack will add the new element at the top. After every push operation the top is incremented by one. If the array is full and no new element can be accommodated, then the stack overflow condition occurs.

POP: The process of deleting (or removing) an element from the top of stack is called POP operation. After every pop operation the stack is decremented by one. If there is no element in the stack and the pop operation is performed then the stack underflow condition occurs**Stack using linked list:** Implementation issues of the stack (Last In First Out - LIFO) using linked list is illustrated in following figures.

Stack Implementation

Stack can be implemented in two ways:
1. Static implementation (using arrays)
2. Dynamic implementation (using pointers)

4.1 Static implementation using arrays

Implementation of stack using arrays is a very simple technique. Algorithm for pushing (or add or insert) a new element at the top of the stack and popping (or delete) an element from the stack is given below.

Algorithm:

Push(Stack [MAXSIZE], item)

Let stack[Maxsize]is an array for implementing the stack.
1 [check for stack overflow?]
  If Top=Maxsize-1, then print: overflow and exit.
2 Set TOP=TOP+1 [increase TOP by 1]
3 Set STACK[TOP]:=ITEM [inserts item in new TOP position]
4 EXIT

Pop(Stack [MAXSIZE], item)

1 [check for stack underflow?]
  If Top<0, then print: stack underflow and exit.
  Else [remove the Top element]
  Set item=stack[TOP]
2 Decrement the stack top
  Set Top=Top-1
3 Return the deleted item from the stack
4 EXIT

Implementation:

/* Program of stack using array*/

```
#include<stdio.h>
#include<stdlib.h>
int top=-1;
#define MAX_SIZE 101
int S[MAX_SIZE];
void Push(int x)
{
	if(top == MAX_SIZE -1)
	{
		printf("Stack Overflow");
		return;
	}else{
		S[++top]=x;
	}
}
void Pop()
{
	if(top==-1)
	{
		printf("Stack is EMPTY");
		return;
	}else{
		top--;
	}
}
int Top()
{
	return S[top];
}
void printStack()
{
	int i;
	printf("Stack Elements are:\n");
	for(i=0;i<=top;i++)
	{
		printf("%d\n",S[i]);
	}
}
int main()
{
	int choice,data;

	while(1)
	{
		printf("\n0.Exit\n1. PUSH an element into Stack\n2. Pop an Element from Stack\n3. Top of the Stack\n");
		printf("4. Display\n");
		printf("Enter your Choice:\t");
```

```
		while(scanf("%d",&choice)==0)
		{
			printf("Only Numbers!!!");
			int c;
			while((c=getchar())!='\n' && c!=EOF); //clear the stdin
		}
		switch(choice)
		{
			case 0:
				exit(0);
			case 1:
				printf("\nEnter an Element:\t");
				while(scanf("%d",&data)==0)
				{
					printf("Only Numbers!!!");
					int c;
					while((c=getchar())!='\n' && c!=EOF); //clear the stdin
				}
				Push(data);
				break;
			case 2:
				Pop();
				break;
			case 3:
				printf("Top of the Elements in Stack %d",Top());
				break;
			case 4:
				printStack();
				break;
			default:
				printf("WRONG ENTRY");
				break;
		}
	}
	return 0;
}
```

Output:

Stack implementation using linked list

Algorithm for PUSH operation:
Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. DATA is the data item to be pushed.

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

Step 1: Create a **newNode** with given value.

Step 2: Check whether stack is **Empty (top == NULL)**

Step 3: If it is **Empty**, then set **newNode → next = NULL.**

Step 4: If it is **Not Empty**, then set **newNode → next = top.**

Step 5: Finally, set **top = newNode.**

Algorithm for POP operation
Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. TEMP is pointer variable to hold any nodes address. DATA is the information on the node which is just deleted.

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

Step 1: Check whether **stack** is **Empty (top == NULL).**

Step 2: If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function

Step 3: If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.

Step 4: Then set '**top = top → next**'.

Step 7: Finally, delete '**temp**' (**free(temp)**).

Algorithm for deletion operation

display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

Step 1: Check whether stack is **Empty (top == NULL).**

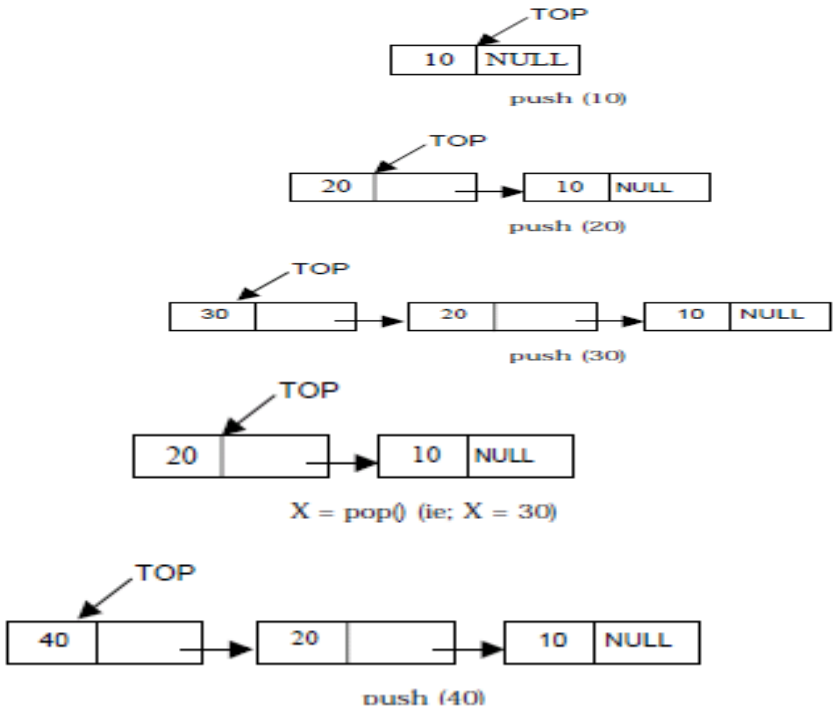
Step 2: If it is **Empty**, then display '**Stack is Empty!!!**' and terminate the function.

Step 3: If it is **Not Empty**, then define a Node pointer '**temp**' and initialize with **top.**

Step 4: Display '**temp → data** --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack (**temp → next != NULL**).

Step 4: Finally! Display '**temp → data** ---> **NULL**'.

Illustration of stack operations using linked list:



Program:

/* Write a program to perform operations on stack using linked list */

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
struct node
{
        int item;
        struct node *next;
}*top;
int count;
int empty()
{
        if(top==NULL)
                return 1;
        else
                return 0;
}
void show()
{
        int i;
        struct node *p;
        if(empty())
        {
                printf("\nNo elements in the stack:");
                return;
        }
```

```
	printf("\nTotal Elements present in the stack  %d :",count);
	p=top;
	while(p!=NULL)
	{
		printf("\n%d",p->item);
		p=p->next;
	}
}
void push()
{
	int x,i;
	struct node *temp;
	printf("\nEnter the element to push into stack: ");
	scanf("%d",&x);
	temp=(struct node*)malloc(sizeof(struct node));
	temp->item=x;
	temp->next=NULL;
	if(top==NULL)
		top=temp;
	else
	{
		temp->next=top;
		top=temp;
	}
	count++;
}
void pop()
{
	struct node *temp;
	if(empty())
	{
		printf("\n NO element in stack to delete:");
		return;
	}
	temp=top;
	top=top->next;
	free(temp);
	count--;
}
void main()
{
	int choice,ch;
	do
	{
		printf("\n1.push\n2.pop\n3.show\nenter choice: ");
		scanf("%d",&choice);
		switch(choice)
		{
			case 1: push();break;
```

```
                case 2: pop();break;
                case 3: show(); break;
                default: printf("\n Wrong Choice: ");
            }
            printf("\n Do You Want To Continue:  (1/0)   : ");
            scanf("%d",&ch);
        }       while(ch==1);
        getch();
}
```

output:

Experiment 5

Aim: Write a program that implement Linear Queue (its operations) using i) Arrays ii) Pointers.

Objectives:

- To understand and implement the queue data structure
- To design, implement, and analyze queue operations implemented using array and linked list
- To understand different operations on queue
- To understand the implementation of queue operations using array and linked list

QUEUES

Queue is a linear data structure.

- In which data will be inserted and deleted through two different ends.
- Follows the mechanism of FIFO (first in first out).
- Insertions are allowed at rear end.
- Deletions are allowed at front end.
- Queue allows insertion, deletion and display operations.
- Insertion operation is also called as enqueue operation.
- The deletion operation is also called as dequeue operation.
- Overflow occurs whenever the queue is full
- Under flow occurs whenever the queue is empty.

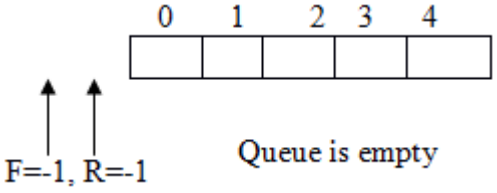
A queue can be represented by two ways.

- Array representation
- Linked list representation.

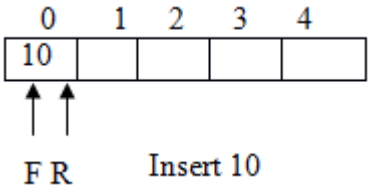
Queue implementation using Array

Following figure will illustrate the basic operations on queue.

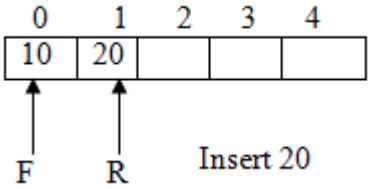
Consider a queue Q with 5 elements Q[5]. Initially there is no element in the queue front and rear pointers pointing to -1.



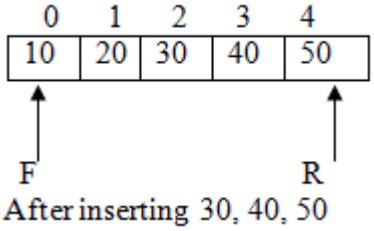
Insert element 10 to the queue, when ever we are inserting element 10 to the queue before that we need to increase front and rear values by one, now front and rear points to array index zero.



Insert element 20 to the queue, when ever we are inserting element 20 to the queue before that we need to increase rear values by one, rear points to array index one.

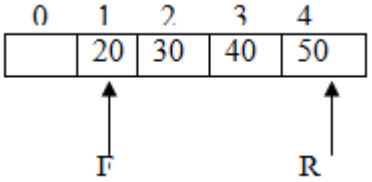


Similarly After inserting 30, 40,50 to the queue front is pointing to zero and rear is pointing to four.

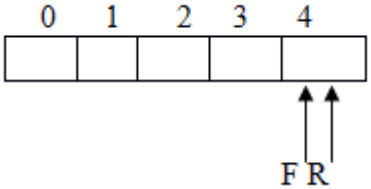


Now queue is full, after this point it is not possible to insert a new element to the queue, this condition is called overflow condition

next if you call the delete operation first inserted element is deleted (i.e. 10) by incrementing front pointer now queue is look like



After deleting remaining elements 20,30,40,50 from the queue



Algorithm:

Let Q be the array of some specified size say SIZE

INSERTING AN ELEMENT INTO THE QUEUE

1. Initialize front=0 rear = –1
2. Input the value to be inserted and assign to variable “data”
3. If (rear >= SIZE)
   (*a*) Display “Queue overflow”
   (*b*) Exit
4. Else
   (*a*) Rear = rear +1
5. Q[rear] = data
6. Exit

DELETING AN ELEMENT FROM QUEUE

1. If (rear< front)
   (*a*) Front = 0, rear = –1
   (*b*) Display “The queue is empty”
   (*c*) Exit
2. Else
   (*a*) Data = Q[front]
3. Front = front +1
4. Exit

Applications of Queues:

a. Round Robin technique for processor scheduling is implemented using queues.
b. All types of customer service center software’s are designed using queues to store customer information.(ex. Train reservation)
c. Printer server routines are designed using queues.

QUEUE implementation using ARRAY

Program:

```
 # include<stdio.h>
# define MAX 5
intqueue_arr[MAX];
int rear = -1;
int front = -1;
main()
{
        int choice;
        while(1)
        {
                printf("1.Insert\n");
                printf("2.Delete\n");
                printf("3.Display\n");
                printf("4.Quit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);

                switch(choice)
                {
                case 1 :
                        insert();
                        break;
                case 2 :
                        del();
                        break;
                case 3:
                        display();
                        break;
                case 4:
                        exit(1);
                default:
                        printf("Wrong choice\n");
                }/*End of switch*/
        }/*End of while*/
}/*End of main()*/

insert()
{
        intadded_item;
        if (rear==MAX-1)
                printf("Queue Overflow\n");
        else
        {
                if (front==-1)  /*If queue is initially empty */
                        front=0;
                printf("Input the element for adding in queue : ");
```

```
                scanf("%d", &added_item);
                rear=rear+1;
                queue_arr[rear] = added_item ;
        }
}/*End of insert()*/

del()
{
        if (front == -1 || front > rear)
        {
                printf("Queue Underflow\n");
                return ;
        }
        else
        {
                printf("Element deleted from queue is : %d\n", queue_arr[front]);
                front=front+1;
        }
}/*End of del() */

display()
{
        int i;
        if (front == -1)
                printf("Queue is empty\n");
        else
        {
                printf("Queue is :\n");
                for(i=front;i<= rear;i++)
                        printf("%d  ",queue_arr[i]);
                printf("\n");
        }
}/*End of display() */
```

Output:

QUEUE implementation using linked list

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
struct Node
{
 int data;
 struct Node *next;
}*front = NULL,*rear = NULL;
void insert(int);
void delete();
void display();
void main()
{
  int choice, value;
  printf("\n:: Queue Implementation using Linked List ::\n");
  while(1){
    printf("\n****** MENU ******\n");
    printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice){
        case 1: printf("Enter the value to be insert: ");
                scanf("%d", &value);
                insert(value);
                break;
        case 2: delete(); break;
        case 3: display(); break;
        case 4: exit(0);
        default: printf("\nWrong selection!!! Please try again!!!\n");
    }
  }
}
void insert(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  newNode -> next = NULL;
  if(front == NULL)
    front = rear = newNode;
  else{
    rear -> next = newNode;
```

```
    rear = newNode;
  }
  printf("\nInsertion is Success!!!\n");
}
void delete()
{
  if(front == NULL)
    printf("\nQueue is Empty!!!\n");
  else{
    struct Node *temp = front;
    front = front -> next;
    printf("\nDeleted element: %d\n", temp->data);
    free(temp);
  }
}
void display()
{
  if(front == NULL)
    printf("\nQueue is Empty!!!\n");
  else{
    struct Node *temp = front;
    while(temp->next != NULL){
        printf("%d--->",temp->data);
        temp = temp -> next;
    }
    printf("%d--->NULL\n",temp->data);
  }  }
```

Output:

Experiment 6

Aim: Write a C program that implement Deque (its operations).

```
#include<stdio.h>
#include<process.h>
#define MAX 30
typedef struct dequeue
{
   int data[MAX];
   int rear,front;
}dequeue;
void initialize(dequeue *p);
int empty(dequeue *p);
int full(dequeue *p);
void enqueueR(dequeue *p,int x);
void enqueueF(dequeue *p,int x);
int dequeueF(dequeue *p);
int dequeueR(dequeue *p);
void print(dequeue *p);
void main()
{
   int i,x,op,n;
   dequeue q;

initialize(&q);
   do
   {
printf("\n1.Create\n2.Insert(rear)\n3.Insert(front)\n4.Delete(rear)\n5.Delete(front");
         printf("\n6.Print\n7.Exitn\n Enter your choice:");

scanf("%d",&op);
switch(op)
         {
 case 1:
printf("/nEnter number of elements:");
scanf("%d",&n);
initialize(&q);
printf("\nEnter the data:");
for(i=0;i<n;i++)
  {
scanf("%d",&x);
if(full(&q))
{
printf("\nQueue is full!!");
exit(0);
}
enqueueR(&q,x);
   }
```

```
break;
case 2:
printf("\nEnter element to be inserted:");
scanf("%d",&x);
if(full(&q))
   {
printf("\nQueue is full!!");
exit(0);
   }
enqueueR(&q,x);
break;
case 3: printf("\nEnter the element to be inserted:");
scanf("%d",&x);
if(full(&q))
{
printf("\nQueue is full!!");
   exit(0);
}
enqueueF(&q,x);
break;
case 4: if(empty(&q))
{
printf("\nQueue is empty!!");
 exit(0);
}
x=dequeueR(&q);
printf("\nElement deleted is %dn",x);
break;
case 5: if(empty(&q))
{
printf("\nQueue isempty!!");
exit(0);

}
x=dequeueF(&q);
printf("\nElement deleted is %dn",x);
break;
case 6: print(&q);
break;
default: break;
          }
   }while(op!=7);
}
void initialize(dequeue *P)
{
   P->rear=-1;
   P->front=-1;
}
int empty(dequeue *P)
{
```

```
   if(P->rear==-1)
        return(1);
   return(0);
}
int full(dequeue *P)
{
if((P->rear+1)%MAX==P->front)
        return(1);
   return(0);
}
void enqueueR(dequeue *P,int x)
{
   if(empty(P))
   {
        P->rear=0;
        P->front=0;

P->data[0]=x;
   }
   else
   {
P->rear=(P->rear+1)%MAX;

P->data[P->rear]=x;
   }
}
void enqueueF(dequeue *P,int x)
{
   if(empty(P))
   {
        P->rear=0;
        P->front=0;
P->data[0]=x;
   }
   else
   {
P->front=(P->front-1+MAX)%MAX;
P->data[P->front]=x;
   }
}
int dequeueF(dequeue *P)
{
   int x;
x=P->data[P->front];
if(P->rear==P->front)
//delete the last element
initialize(P);
   else
P->front=(P->front+1)%MAX;
   return(x);
```

```
}
int dequeueR(dequeue *P)
{
   int x;
x=P->data[P->rear];
if(P->rear==P->front)
          initialize(P);
   else
P->rear=(P->rear-1+MAX)%MAX;
   return(x);
}
void print(dequeue *P)
{
   if(empty(P))
   {
printf("\nQueue is empty!!");
          exit(0);
   }
   {
     int i;
   i=P->front;
while(i!=P->rear)
   {
printf("\n%d",P->data[i]);
i=(i+1)%MAX;
   }
   }
printf("\n%dn",P->data[P->rear]);
}
```

OUTPUT:

Write a program to implement all the functions of a dictionary using hashing.

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

struct set
{
  int key;
  int data;
};
struct set *array;
int capacity = 10;
int size = 0;

int hashFunction(int key)
{
  return (key % capacity);
}
int checkPrime(int n)
{
  int i;
  if (n == 1 || n == 0)
  {
  return 0;
  }
  for (i = 2; i < n / 2; i++)
  {
  if (n % i == 0)
  {
    return 0;
  }
  }
  return 1;
}
int getPrime(int n)
{
  if (n % 2 == 0)
  {
  n++;
```

```
  }
  while (!checkPrime(n))
  {
  n += 2;
  }
  return n;
}
void init_array()
{
  capacity = getPrime(capacity);
  array = (struct set *)malloc(capacity * sizeof(struct set));
  for (int i = 0; i < capacity; i++)
  {
  array[i].key = 0;
  array[i].data = 0;
  }
}

void insert(int key, int data)
{
  int index = hashFunction(key);
  if (array[index].data == 0)
  {
  array[index].key = key;
  array[index].data = data;
  size++;
  printf("\n Key (%d) has been inserted \n", key);
  }
  else if (array[index].key == key)
  {
  array[index].data = data;
  }
  else
  {
  printf("\n Collision occured  \n");
  }
}

void remove_element(int key)
{
```

```
  int index = hashFunction(key);
  if (array[index].data == 0)
  {
  printf("\n This key does not exist \n");
  }
  else
  {
  array[index].key = 0;
  array[index].data = 0;
  size--;
  printf("\n Key (%d) has been removed \n", key);
  }
}
void display()
{
  int i;
  for (i = 0; i < capacity; i++)
  {
  if (array[i].data == 0)
  {
    printf("\n array[%d]: / ", i);
  }
  else
  {
    printf("\n key: %d array[%d]: %d \t", array[i].key, i, array[i].data);
  }
  }
}

int size_of_hashtable()
{
  return size;
}

int main()
{
  int choice, key, data, n;
  int c = 0;
  init_array();
```

```
do
{
printf("1.Insert item in the Hash Table"
   "\n2.Remove item from the Hash Table"
   "\n3.Check the size of Hash Table"
   "\n4.Display a Hash Table"
   "\n\n Please enter your choice: ");

scanf("%d", &choice);
switch (choice)
{
case 1:

  printf("Enter key -:\t");
  scanf("%d", &key);
  printf("Enter data -:\t");
  scanf("%d", &data);
  insert(key, data);

  break;

case 2:

  printf("Enter the key to delete-:");
  scanf("%d", &key);
  remove_element(key);

  break;

case 3:

  n = size_of_hashtable();
  printf("Size of Hash Table is-:%d\n", n);

  break;

case 4:

  display();
```

```
    break;

  default:

    printf("Invalid Input\n");
  }

  printf("\nDo you want to continue (press 1 for yes): ");
  scanf("%d", &c);

  } while (c == 1);
}
```

OUTPUT:

```
1.Insert item in the Hash Table
2.Remove item from the Hash Table
3.Check the size of Hash Table
4.Display a Hash Table

 Please enter your choice: 1
Enter key -:    20
Enter data -:   34

 Key (20) has been inserted

Do you want to continue (press 1 for yes): 1
1.Insert item in the Hash Table
2.Remove item from the Hash Table
3.Check the size of Hash Table
4.Display a Hash Table

 Please enter your choice: 1
Enter key -:    66
Enter data -:   49

 Key (66) has been inserted

Do you want to continue (press 1 for yes): 1
1.Insert item in the Hash Table
```

2.Remove item from the Hash Table
3.Check the size of Hash Table
4.Display a Hash Table

Please enter your choice: 4

key: 66 array[0]: 49
array[1]: /
array[2]: /
array[3]: /
array[4]: /
array[5]: /
array[6]: /
array[7]: /
array[8]: /
key: 20 array[9]: 34
array[10]: /
Do you want to continue (press 1 for yes): 0

Write a program that implement Binary Search Trees to perform the following operations i) Creation ii) Insertion iii) Deletion iv) Traversal

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

struct btnode
{
   int value;
   struct btnode *l;
   struct btnode *r;
}*root = NULL, *temp = NULL, *t2, *t1;

void delete1();
void insert();
void delete();
void inorder(struct btnode *t);
void create();
void search(struct btnode *t);
```

```
void search1(struct btnode *t,int data);
int smallest(struct btnode *t);
int largest(struct btnode *t);

int flag = 1;

void main()
{
   int ch;

   printf("\nOPERATIONS ---");
   printf("\n1 - Insert an element into tree\n");
   printf("2 - Delete an element from the tree\n");
   printf("3 - Inorder Traversal\n");
   printf("4 - Exit\n");
   while(1)
   {
      printf("\nEnter your choice : ");
      scanf("%d", &ch);
      switch (ch)
      {
      case 1:
         insert();
         break;
      case 2:
         delete();
         break;
      case 3:
         inorder(root);
         break;
      case 4:
         exit(0);
      default :
         printf("Wrong choice, Please enter correct choice  ");
         break;
      }
   }
}

/* To insert a node in the tree */
```

```
void insert()
{
   create();
   if (root == NULL)
      root = temp;
   else
      search(root);
}

/* To create a node */
void create()
{
   int data;

   printf("Enter data of node to be inserted : ");
   scanf("%d", &data);
   temp = (struct btnode *)malloc(1*sizeof(struct btnode));
   temp->value = data;
   temp->l = temp->r = NULL;
}

/* Function to search the appropriate position to insert the new node */
void search(struct btnode *t)
{
   if ((temp->value > t->value) && (t->r != NULL))       /* value more than root node value
insert at right */
      search(t->r);
   else if ((temp->value > t->value) && (t->r == NULL))
      t->r = temp;
   else if ((temp->value < t->value) && (t->l != NULL))    /* value less than root node value
insert at left */
      search(t->l);
   else if ((temp->value < t->value) && (t->l == NULL))
      t->l = temp;
}

/* recursive function to perform inorder traversal of tree */
void inorder(struct btnode *t)
{
   if (root == NULL)
```

```
    {
        printf("No elements in a tree to display");
        return;
    }
    if (t->l != NULL)
        inorder(t->l);
    printf("%d -> ", t->value);
    if (t->r != NULL)
        inorder(t->r);
}

/* To check for the deleted node */
void delete()
{
    int data;

    if (root == NULL)
    {
        printf("No elements in a tree to delete");
        return;
    }
    printf("Enter the data to be deleted : ");
    scanf("%d", &data);
    t1 = root;
    t2 = root;
    search1(root, data);
}

/* Search for the appropriate position to insert the new node */
void search1(struct btnode *t, int data)
{
    if ((data>t->value))
    {
        t1 = t;
        search1(t->r, data);
    }
    else if ((data < t->value))
    {
        t1 = t;
        search1(t->l, data);
```

```
    }
    else if ((data==t->value))
    {
        delete1(t);
    }
}

/* To delete a node */
void delete1(struct btnode *t)
{
    int k;

    /* To delete leaf node */
    if ((t->l == NULL) && (t->r == NULL))
    {
        if (t1->l == t)
        {
            t1->l = NULL;
        }
        else
        {
            t1->r = NULL;
        }
        t = NULL;
        free(t);
        return;
    }

    /* To delete node having one left hand child */
    else if ((t->r == NULL))
    {
        if (t1 == t)
        {
            root = t->l;
            t1 = root;
        }
        else if (t1->l == t)
        {
            t1->l = t->l;
```

```
    }
    else
    {
        t1->r = t->l;
    }
    t = NULL;
    free(t);
    return;
}

/* To delete node having right hand child */
else if (t->l == NULL)
{
    if (t1 == t)
    {
        root = t->r;
        t1 = root;
    }
    else if (t1->r == t)
        t1->r = t->r;
    else
        t1->l = t->r;
    t == NULL;
    free(t);
    return;
}

/* To delete node having two child */
else if ((t->l != NULL) && (t->r != NULL))
{
    t2 = root;
    if (t->r != NULL)
    {
        k = smallest(t->r);
        flag = 1;
    }
    else
    {
        k =largest(t->l);
        flag = 2;
```

```
        }
        search1(root, k);
        t->value = k;
    }

}

/* To find the smallest element in the right sub tree */
int smallest(struct btnode *t)
{
    t2 = t;
    if (t->l != NULL)
    {
        t2 = t;
        return(smallest(t->l));
    }
    else
        return (t->value);
}

/* To find the largest element in the left sub tree */
int largest(struct btnode *t)
{
    if (t->r != NULL)
    {
        t2 = t;
        return(largest(t->r));
    }
    else
        return(t->value);
}
```

OUTPUT:

```
OPERATIONS ---
1 - Insert an element into tree
2 - Delete an element from the tree
3 - Inorder Traversal
4 - Exit
```

```
Enter your choice : 1
Enter data of node to be inserted : 12

Enter your choice : 1
Enter data of node to be inserted : 5

Enter your choice : 1
Enter data of node to be inserted : 34

Enter your choice : 1
Enter data of node to be inserted : 88

Enter your choice : 1
Enter data of node to be inserted : 2

Enter your choice : 3
2 -> 5 -> 12 -> 34 -> 88 ->
Enter your choice : 2
Enter the data to be deleted : 12

Enter your choice : 3
2 -> 5 -> 34 -> 88 ->
Enter your choice : 2
Enter the data to be deleted : 34

Enter your choice :
3
2 -> 5 -> 88 ->
Enter your choice : 4
```

Write a program to implement the tree traversal methods using recursion.

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

struct btnode
```

```
{
   int value;
   struct btnode *l;
   struct btnode *r;
}*root = NULL, *temp = NULL, *t2, *t1;

void insert();
void inorder(struct btnode *t);
void create();
void search(struct btnode *t);
void preorder(struct btnode *t);
void postorder(struct btnode *t);

int flag = 1;

void main()
{
   int ch;

   printf("\n TREE TRAVERSAL METHODS USING RECURSION");
   printf("\n1 - Insert an element into tree\n");
   printf("2 - Inorder Traversal\n");
   printf("3 - Preorder Traversal\n");
   printf("4 - Postorder Traversal\n");
   printf("5 - Exit\n");
   while(1)
   {
      printf("\nEnter your choice : ");
      scanf("%d", &ch);
      switch (ch)
      {
      case 1:
         insert();
         break;
      case 2:
         inorder(root);
         break;
      case 3:
         preorder(root);
         break;
```

```
        case 4:
            postorder(root);
            break;
        case 5:
            exit(0);
        default :
            printf("Wrong choice, Please enter correct choice  ");
            break;
        }
    }
}

/* To insert a node in the tree */
void insert()
{
    create();
    if (root == NULL)
        root = temp;
    else
        search(root);
}

/* To create a node */
void create()
{
    int data;

    printf("Enter data of node to be inserted : ");
    scanf("%d", &data);
    temp = (struct btnode *)malloc(1*sizeof(struct btnode));
    temp->value = data;
    temp->l = temp->r = NULL;
}

/* Function to search the appropriate position to insert the new node */
void search(struct btnode *t)
{
    if ((temp->value > t->value) && (t->r != NULL))       /* value more than root node value
insert at right */
        search(t->r);
```

```
    else if ((temp->value > t->value) && (t->r == NULL))
        t->r = temp;
    else if ((temp->value < t->value) && (t->l != NULL))    /* value less than root node value
insert at left */
        search(t->l);
    else if ((temp->value < t->value) && (t->l == NULL))
        t->l = temp;
}

/* recursive function to perform inorder traversal of tree */
void inorder(struct btnode *t)
{
    if (root == NULL)
    {
        printf("No elements in a tree to display");
        return;
    }
    if (t->l != NULL)
        inorder(t->l);
    printf("%d -> ", t->value);
    if (t->r != NULL)
        inorder(t->r);
}

/* To find the preorder traversal */
void preorder(struct btnode *t)
{
    if (root == NULL)
    {
        printf("No elements in a tree to display");
        return;
    }
    printf("%d -> ", t->value);
    if (t->l != NULL)
        preorder(t->l);
    if (t->r != NULL)
        preorder(t->r);
}

/* To find the postorder traversal */
```

```
void postorder(struct btnode *t)
{
   if (root == NULL)
   {
      printf("No elements in a tree to display ");
      return;
   }
   if (t->l != NULL)
      postorder(t->l);
   if (t->r != NULL)
      postorder(t->r);
   printf("%d -> ", t->value);
}
```

OUTPUT:

```
 TREE TRAVERSAL METHODS USING RECURSION
1 - Insert an element into tree
2 - Inorder Traversal
3 - Preorder Traversal
4 - Postorder Traversal
5 - Exit

Enter your choice : 1
Enter data of node to be inserted : 12

Enter your choice : 1
Enter data of node to be inserted : 5

Enter your choice : 34
Wrong choice, Please enter correct choice
Enter your choice : 1
Enter data of node to be inserted : 34

Enter your choice : 1
Enter data of node to be inserted : 2

Enter your choice : 1
Enter data of node to be inserted : 88
```

Enter your choice : 2
2 -> 5 -> 12 -> 34 -> 88 ->
Enter your choice : 3
12 -> 5 -> 2 -> 34 -> 88 ->
Enter your choice : 4
2 -> 5 -> 88 -> 34 -> 12 ->
Enter your choice : 5

10. Write a program that implements the following sorting methods to sort a given list of integers in ascending order

i) Heap sort

ii) Merge sort

```
// Heap Sort in C

  #include <stdio.h>
#include <conio.h>
#define MAX 10
void RestoreHeapUp(int *,int);
void RestoreHeapDown(int*,int,int);
int main()
{
int Heap[MAX],n,i,j;
//clrscr();
printf("\n Enter the number of elements : ");
scanf("%d",&n);
printf("\n Enter the elements : ");
for(i=1;i<=n;i++)
{
scanf("%d",&Heap[i]);
RestoreHeapUp(Heap, i); // Heapify
}
// Delete the root element and heapify the heap
j=n;
for(i=1;i<=j;i++)
{
int temp;
temp=Heap[1];
Heap[1]= Heap[n];
Heap[n]=temp;
n = n-1; // The element Heap[n] is supposed to be deleted
RestoreHeapDown(Heap,1,n); // Heapify
}
n=j;
printf("\n The sorted elements are: ");
for(i=1;i<=n;i++)
printf("%4d",Heap[i]);
```

```
return 0;
}
void RestoreHeapUp(int *Heap,int index)
{
int val = Heap[index];
while( (index>1) && (Heap[index/2] < val) )// Check parent's value
{
Heap[index]=Heap[index/2];
index /= 2;
}
Heap[index]=val;
}
void RestoreHeapDown(int *Heap,int index,int n)
{
int val = Heap[index];
int j=index*2;
while(j<=n)
{
if( (j<n) && (Heap[j] < Heap[j+1]))// Check sibling's value
j++;
if(Heap[j] < Heap[j/2]) // Check parent's value
break;
Heap[j/2]=Heap[j];
j=j*2;
}
Heap[j/2]=val;
}
```

Output:

```
 Enter the number of elements : 8

 Enter the elements : 23
44
1
67
22
12
66
99
```

The sorted elements are: 1 12 22 44 23 66 67 99

MERGE SORT

```
#include <stdio.h>
#include <conio.h>
#define size 100
void merge(int a[], int, int, int);
void merge_sort(int a[],int, int);
void main()
{
int arr[size], i, n;
printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
printf("\n Enter the elements of the array: ");
for(i=0;i<n;i++)
{
 scanf("%d", &arr[i]);
}
merge_sort(arr, 0, n-1);
printf("\n The sorted array is: \n");
for(i=0;i<n;i++)
printf(" %d\t", arr[i]);
getch();
}
void merge(int arr[], int beg, int mid, int end)
{
int i=beg, j=mid+1, index=beg, temp[size], k;
while((i<=mid) && (j<=end))
{
 if(arr[i] < arr[j])
 {
temp[index] = arr[i];
i++;
 }
 else
 {
temp[index] = arr[j];
```

```
j++;
 }
index++;
}
if(i>mid)
{
 while(j<=end)
 {
    temp[index] = arr[j];
j++;
index++;
 }
}
else
{
 while(i<=mid)
 {
temp[index] = arr[i];
i++;
index++;
 }
}
for(k=beg;k<index;k++)
arr[k] = temp[k];
}
void merge_sort(int arr[], int beg, int end)
{
int mid;
if(beg<end)
{
mid = (beg+end)/2;
 merge_sort(arr, beg, mid);
merge_sort(arr, mid+1, end);
 merge(arr, beg, mid, end);
}
}
```

Output:

```
Enter the number of elements in the array : 6

Enter the elements of the array: 88
12
33
18
85
22

The sorted array is:
12      18      22      33      85      88
```

11. Write a program to implement the graph traversal methods such as BFS and DFS.

```
#include
#define MAX 10
void breadth_first_search(int adj[][MAX],int visited[],int start)
{
 int queue[MAX],rear = –1,front =– 1, i;
queue[++rear] = start;
 visited[start] = 1;
 while(rear != front)
 {
 start = queue[++front];
if(start == 4)
printf("5\t");
 else
 printf("%c \t",start + 65);
 for(i = 0; i < MAX; i++)
 {
 if(adj[start][i] == 1 && visited[i] == 0)
 {
 queue[++rear] = i; visited[i] = 1;
}
 }
 }
 }
 int main()
```

```
{
int visited[MAX] = {0};
        int adj[MAX][MAX], i, j;
printf("\n Enter the adjacency matrix: ");
        for(i = 0; i < MAX; i++)
                for(j = 0; j < MAX; j++)
 scanf("%d", &adj[i][j]);
breadth_first_search(adj,visited,0);
        return 0;
}
```

Output:

Output

Enter the adjacency matrix:

0 1 0 1 0

1 0 1 1 0

0 1 0 0 1

1 1 0 0 1

0 0 1 1 0

A B D C E

```
#include <stdio.h>
#define MAX 5
void depth_first_search(int adj[][MAX],int visited[],int start)
{
        int stack[MAX];
int top = –1, i;
printf("%c–",start + 65);
visited[start] = 1;
stack[++top] = start;
while(top ! = –1)
{
 start = stack[top];
                for(i = 0; i < MAX; i++)
 {
        if(adj[start][i] && visited[i] == 0)
 {
```

```
 stack[++top] = i;
printf("%c–", i + 65);
visited[i] = 1;
break;
 }
 }
                    if(i == MAX)
 top––;
}
}
int main()
{
          int adj[MAX][MAX];
          int visited[MAX] = {0}, i, j;
printf("\n Enter the adjacency matrix: ");
          for(i = 0; i < MAX; i++)
                    for(j = 0; j < MAX; j++)
 scanf("%d", &adj[i][j]);
printf("DFS Traversal: ");
          depth_first_search(adj,visited,0);
printf("\n");
          return 0;
}
```

Output

```
Enter the adjacency matrix:
0 1 0 1 0
1 0 1 1 0
0 1 0 0 1
1 1 0 0 1
0 0 1 1 0
DFS Traversal: A –> C –> E –>
```

12. Write a program to implement the Knuth-Morris-Pratt pattern matching algorithm.

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

void computeLPSArray(char* pat, int M, int* lps);

// Prints occurrences of txt[] in pat[]
void KMPSearch(char* pat, char* txt)
{
	int M = strlen(pat);
	int N = strlen(txt);

	// create lps[] that will hold the longest prefix suffix
	// values for pattern
	int lps[M];

	// Preprocess the pattern (calculate lps[] array)
	computeLPSArray(pat, M, lps);

	int i = 0; // index for txt[]
	int j = 0; // index for pat[]
	while (i < N) {
		if (pat[j] == txt[i]) {
			j++;
			i++;
		}

		if (j == M) {
			printf("Found pattern at index %d \n", i - j );
			j = lps[j - 1];
		}

		// mismatch after j matches
		else if (i < N && pat[j] != txt[i]) {
			// Do not match lps[0..lps[j-1]] characters,
```

```
                        // they will match anyway
                        if (j != 0)
                                j = lps[j - 1];
                        else
                                i = i + 1;
                }
        }
}

// Fills lps[] for given pattern pat[0..M-1]
void computeLPSArray(char* pat, int M, int* lps)
{
        // length of the previous longest prefix suffix
        int len = 0;

        lps[0] = 0; // lps[0] is always 0

        // the loop calculates lps[i] for i = 1 to M-1
        int i = 1;
        while (i < M) {
                if (pat[i] == pat[len]) {
                        len++;
                        lps[i] = len;
                        i++;
                }
                else // (pat[i] != pat[len])
                {
                        // This is tricky. Consider the example.
                        // AAACAAAA and i = 7. The idea is similar
                        // to search step.
                        if (len != 0) {
                                len = lps[len - 1];

                                // Also, note that we do not increment
                                // i here
                        }
                        else // if (len == 0)
                        {
                                lps[i] = 0;
                                i++;
```

```
                        }
                }
        }
}

// Driver program to test above function
int main()
{
        char txt[] = "AABAACAADAABAABA";
        char pat[] = "AABA";
        KMPSearch(pat, txt);
        return 0;
}
```

OUTPUT:

```
Found pattern at index 0
Found pattern at index 9
Found pattern at index 12
```