**LINKED LISTS**

1. **Singly linked list**

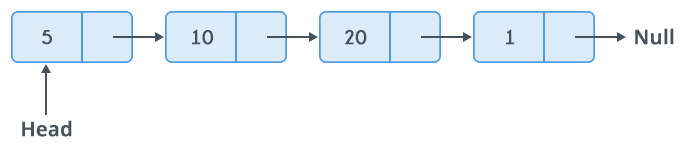
**Aim:** To implement singly linked list using the concept of linked lists.

**Theory:** A linked list is represented by a pointer to the first node of the linked list. The first node is called the head of the linked list. If the linked list is empty, then the value of the head points to NULL.

Each node in a list consists of at least two parts:

* A Data Item (we can store integers, strings, or any type of data).
* Pointer (Or Reference) to the next node (connects one node to another) or An address of another node

In C, we can represent a node using structures.



Uses of Linked List

o The list is not required to be contiguously present in the memory. The node can

reside anywhere in the memory and linked together to make a list. This achieves

optimized utilization of space.

o List size is limited to the memory size and doesn&#39;t need to be declared in advance.

o Empty node cannot be present in the linked list.

o We can store values of primitive types or objects in the singly linked list.

**Program:**

#include<stdio.h>

#include<stdlib.h> //pre-processor directives

struct node //structure definition

{

int data; //data part

struct node \*next; //link part

}; //self referential structure

struct node \*head; //head node declared

void beginsert ();

void lastinsert ();

void randominsert();

void begin\_delete();

void last\_delete();

void random\_delete();

void display();

void search(); //function declaration

void main ()

{

int choice =0;

while(choice != 9)

{

printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from Beginning\n5.Delete from last\n6.Delete node after specified location\n7.Search for an element\n8.Show\n9.Exit\n"); //options available to choose

printf("\nEnter your choice?\n");

scanf("\n%d",&choice);//choice scanned

switch(choice)//switch of choice

{

case 1://case for insertion from begining

beginsert();

break;

case 2://case for insertion from last

lastinsert();

break;

case 3://case for insertion at specified position

randominsert();

break;

case 4://case for deletion from begining

begin\_delete();

break;

case 5://case for deletion from last

last\_delete();

break;

case 6://case for deletion at random position

random\_delete();

break;

case 7://case to search an element

search();

break;

case 8://case to display the elements

display();

break;

case 9://exits from the options

exit(0);

break;

default://wrong choice

printf("Please enter valid choice..");

}

}

}

void beginsert()//function declaration

{

struct node \*ptr;//structure declaration

int item;//variable declaration

ptr = (struct node \*) malloc(sizeof(struct node \*));//dynamic memory allocation

if(ptr == NULL)//condition check

{

printf("\nOVERFLOW");//prints message

}

else

{

printf("\nEnter value\n");

scanf("%d",&item);

ptr->data = item;//data part innitiated

ptr->next = head;//address part initiated

head = ptr;//initialisation

printf("\nNode inserted");//prints message

}

}

void lastinsert()//function defination

{

struct node \*ptr,\*temp;//structure declaration

int item; //variable declaration

ptr = (struct node\*)malloc(sizeof(struct node));//dynamic memory allocation

if(ptr == NULL)//condition check

{

printf("\n OVERFLOW");

}

else

{

printf("\n Enter value?\n");

scanf("%d",&item);

ptr->data = item;

if(head == NULL)//condition check

{

ptr -> next = NULL;//address part initiated

head = ptr;//initialisation

printf("\n Node inserted");//prints message

}

else

{

temp = head;//initialisation

while (temp -> next != NULL)//condition check

{

temp = temp -> next;//accessing next node

}

temp->next = ptr;//address part initiated

ptr->next = NULL;//address part initiated

printf("\nNode inserted");//prints message

}

}

}

void randominsert()//function defination

{

int i,loc,item;//variable declaration

struct node \*ptr, \*temp; //structure declaration

ptr = (struct node \*) malloc (sizeof(struct node));//dynamic memory allocation

if(ptr == NULL)

{

printf("\nOVERFLOW");//prints message

}

else

{

printf("\nEnter element value");

scanf("%d",&item);

ptr->data = item;

printf("\nEnter the location after which you want to insert ");

scanf("\n%d",&loc);

temp=head;//initialisation

for(i=0;i<loc;i++)//condition check

{

temp = temp->next;//accessing next node

if(temp == NULL)//condition check

{

printf("\ncan't insert\n");//prints message

return;

}

}

ptr ->next = temp ->next;//address part initiated

temp ->next = ptr; //address part initiated

printf("\nNode inserted"); //prints message

}

}

void begin\_delete()//function defination

{

struct node \*ptr;//structure declaration

if(head == NULL)//condition check

{

printf("\nList is empty\n");//prints message

}

else

{

ptr = head;//initialisation

head = ptr->next;//address part initiated

free(ptr);//memory free

printf("\nNode deleted from the begining ...\n");//prints message

}

}

void last\_delete()//function defination

{

struct node \*ptr,\*ptr1;//structure declaration

if(head == NULL)//condition check

{

printf("\nlist is empty");//prints message

}

else if(head -> next == NULL)//condition check

{

head = NULL;//initialisation

free(head);//memory free

printf("\nOnly node of the list deleted ...\n");//prints message

}

else

{

ptr = head;//initialisation

while(ptr->next != NULL)//condition check

{

ptr1 = ptr;//initialisation

ptr = ptr ->next;//accessing next node

}

ptr1->next = NULL;//address part initiated

free(ptr);//memory free

printf("\nDeleted Node from the last ...\n");//prints message

}

}

void random\_delete()//function defination

{

struct node \*ptr,\*ptr1;//structure declaration

int loc,i;//variable declaration

printf("\n Enter the location of the node after which you want to perform deletion \n");

scanf("%d",&loc);//lcation scanned

ptr=head;//initialisation

for(i=0;i<loc;i++)//condition check

{

ptr1 = ptr;//initialisation

ptr = ptr->next//accessing next node ;

if(ptr == NULL)//condition check

{

printf("\nCan't delete");//prints message

return;

}

}

ptr1 ->next = ptr ->next;//address part initiated

free(ptr);//memory free

printf("\nDeleted node %d ",loc+1);//prints message //prints message

}

void search()//function defination

{

struct node \*ptr;//structure declaration

int item,i=0,flag;//variable declaration

ptr = head;//initialisation

if(ptr == NULL)

{

printf("\nEmpty List\n");//prints message

}

else

{

printf("\nEnter item which you want to search?\n");

scanf("%d",&item);//item scanned

while (ptr!=NULL)//condition check

{

if(ptr->data == item)//condition check

{

printf("item found at location %d ",i+1);//printing location

flag=0;//item found

}

else

{

flag=1;//item not found

}

i++;//incrementing variable

ptr = ptr -> next;//accessing next node

}

if(flag==1)

{

printf("Item not found\n");//prints message

}

}

}

void display()//function defination

{

struct node \*ptr;//structure declaration

ptr = head;//initialisation

if(ptr == NULL)//condition check

{

printf("Nothing to print");//prints message

}

else

{

printf("\nprinting values . . . . .\n");//prints message

while (ptr!=NULL)//condition check

{

printf("\n%d",ptr->data);

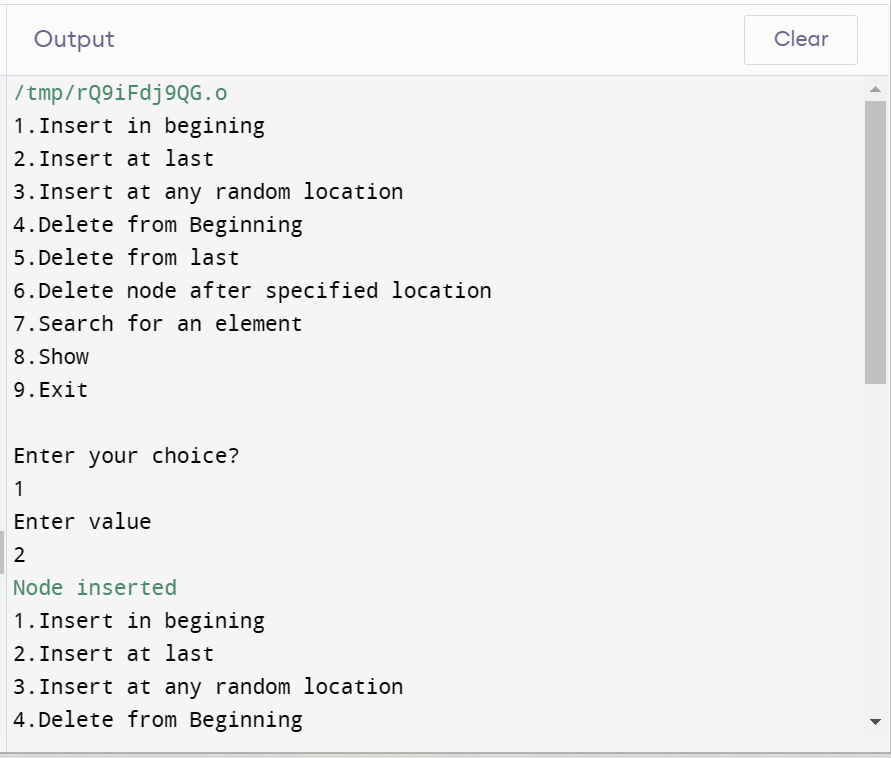
ptr = ptr -> next;//accessing next node

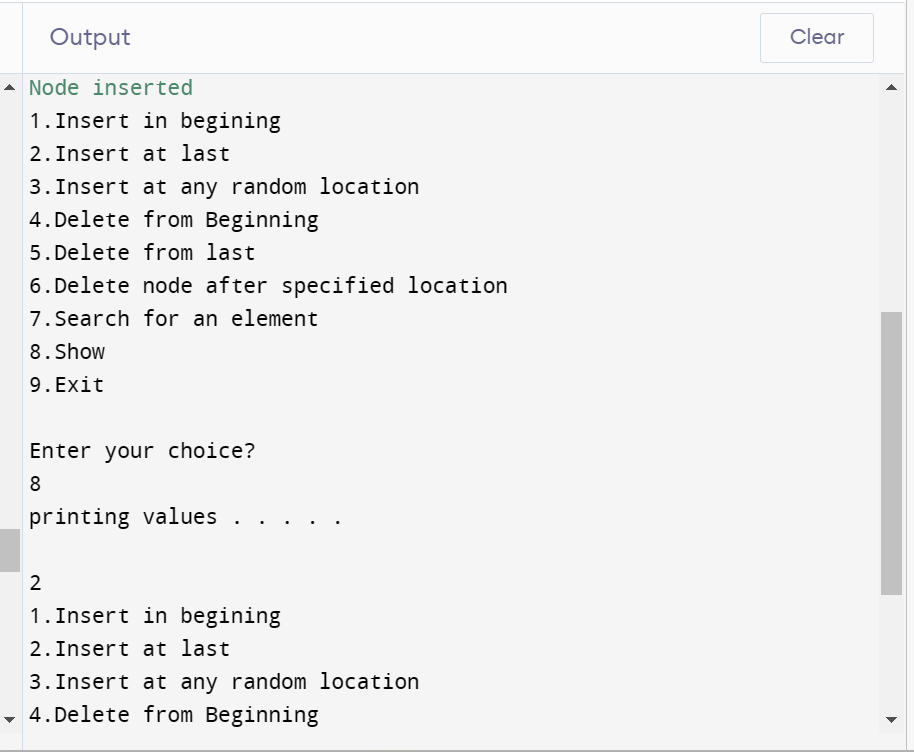
}

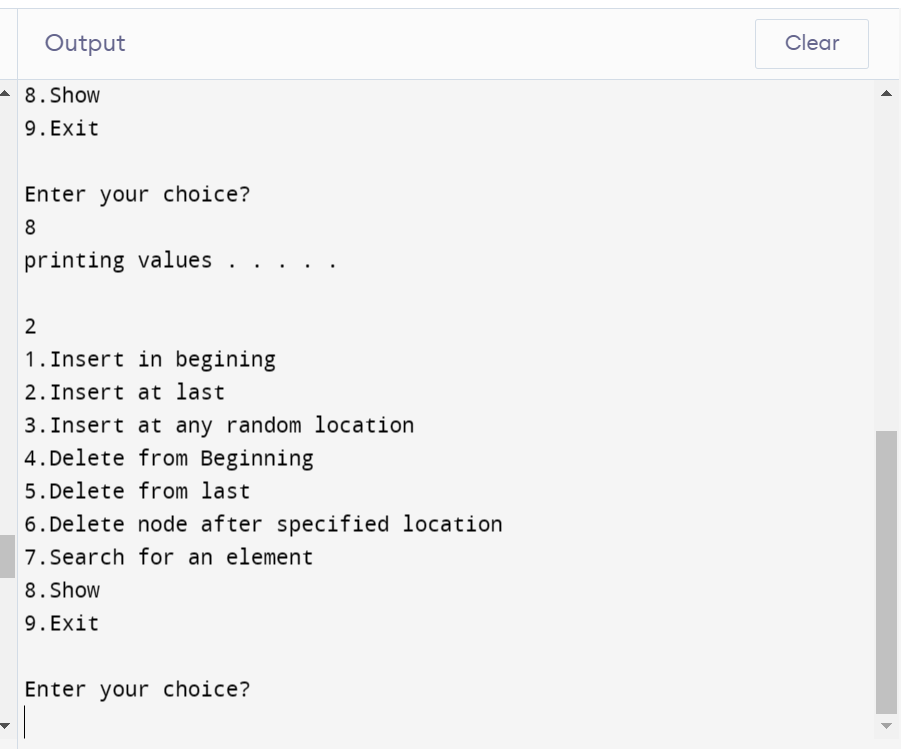
}

}

**Output:**







1. **Doubly linked lists**

**Aim:** To implement doubly linked list.

**Theory:** Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a **doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), pointer to the previous node (previous pointer).** A sample node in a doubly linked list is shown in the figure.



A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



In C, structure of a node in doubly linked list can be given as :

1. struct node
2. {
3. struct node \*prev;
4. **int** data;
5. struct node \*next;
6. }

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

The operations that can be performed using a double linked list is the same as that of a singly linked list.

**Program:**

#include<stdio.h>

#include<stdlib.h>//pre-processor directives

struct node //structure defination

{

struct node \*prev;//previous node address

struct node \*next;//next node address

int data;//data part

};//self referential structure

struct node \*head;//head structure declaration

void insertion\_beginning();

void insertion\_last();

void insertion\_specified();

void deletion\_beginning();

void deletion\_last();

void deletion\_specified();

void display();

void search();//function declaration

void main ()//main function defination

{

int choice =0;//variable declaration

while(choice != 9)//condition check

{

printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from Beginning\n5.Delete from last\n6.Delete node after specified location\n7.Search for an element\n8.Show\n9.Exit\n"); //options available to choose

printf("\nEnter your choice?\n");

scanf("\n%d",&choice);//choice scanned

switch(choice)//switch of choice

{

case 1://case for insertion from begining

beginsert();

break;

case 2://case for insertion from last

lastinsert();

break;

case 3://case for insertion at specified position

randominsert();

break;

case 4://case for deletion from begining

begin\_delete();

break;

case 5://case for deletion from last

last\_delete();

break;

case 6://case for deletion at random position

random\_delete();

break;

case 7://case to search an element

search();

break;

case 8://case to display the elements

display();

break;

case 9://exits from the options

exit(0);

break;

default://wrong choice

printf("Please enter valid choice..");

}

}

}

void insertion\_beginning()//function defination

{

struct node \*ptr;//structure declaration

int item;//variable declaration

ptr = (struct node \*)malloc(sizeof(struct node));//dynamic memory allocation

if(ptr == NULL) //condition check

{

printf("\nOVERFLOW");//prints message

}

else

{

printf("\nEnter Item value");

scanf("%d",&item);//item scanned

if(head==NULL)//condition check

{

ptr->next = NULL;

ptr->prev=NULL;//initialisation

ptr->data=item;//data initiation

head=ptr;//initialisation

}

else

{

ptr->data=item;//data initiation

ptr->prev=NULL;

ptr->next = head;

head->prev=ptr;//prev part of the node initialised

head=ptr;//initialisation

}

printf("\nNode inserted\n");//prints message

}

}

void insertion\_last()//function defination

{

struct node \*ptr,\*temp;//structure declaration

int item;//variable declaration

ptr = (struct node \*) malloc(sizeof(struct node));//dynamic memory allocation

if(ptr == NULL)//condition check

{

printf("\nOVERFLOW");//prints message

}

else

{

printf("\nEnter value");

scanf("%d",&item);//item scanned

ptr->data=item;//data initiation

if(head == NULL) //condition check

{

ptr->next = NULL;//next part of the node initialised

ptr->prev = NULL;//prev part of the node initialised

head = ptr;//initialisation

}

else

{

temp = head;//initialisation

while(temp->next!=NULL)//condition check

{

temp = temp->next;//next part of the node initialised

}

temp->next = ptr;//next part of the node initialised

ptr ->prev=temp; //prev part of the node initialised

ptr->next = NULL;//next part of the node initialised

}

}

printf("\nnode inserted\n");//prints message

}

}

void insertion\_specified() //function defination

{

struct node \*ptr,\*temp;//structure declaration

int item,loc,i;//variable declaration

ptr = (struct node \*)malloc(sizeof(struct node));//dynamic memory allocation

if(ptr == NULL) //condition check

{

printf("\n OVERFLOW");//prints message

}

else

{

temp=head;//initialisation

printf("Enter the location");

scanf("%d",&loc);

for(i=0;i<loc;i++)//condition check

{

temp = temp->next;//acessing next node

if(temp == NULL)//condition check

{

printf("\n There are less than %d elements", loc);

return;

}

}

printf("Enter value");

scanf("%d",&item);//item scanned

ptr->data = item;//data initiation

ptr->next = temp->next;//next part of the node initialised

ptr -> prev = temp;

temp->next = ptr;//next part of the node initialised

temp->next->prev=ptr;//next part of the node initialised

printf("\nnode inserted\n");//prints message

}

}

void deletion\_beginning()//function defination

{

struct node \*ptr;//structure declaration

if(head == NULL)//condition check

{

printf("\n UNDERFLOW");//prints message

}

else if(head->next == NULL)

{

head = NULL;//initialisation

free(head);//memory free

printf("\nnode deleted\n");//prints message

}

else

{

ptr = head;//initialisation

head = head -> next;//next part of the node initialised

head -> prev = NULL;

free(ptr);//memory free

printf("\nnode deleted\n");//prints message

}

}

void deletion\_last()//function defination

{

struct node \*ptr;//structure declaration

if(head == NULL)//condition check

{

printf("\n UNDERFLOW");//prints message

}

else if(head->next == NULL)//condition check

{

head = NULL;//initialisation

free(head);//memory free

printf("\nnode deleted\n");//prints message

}

else

{

ptr = head;//initialisation

if(ptr->next != NULL)//condition check

{

ptr = ptr -> next;//next part of the node initialised

}

ptr -> prev -> next = NULL;

free(ptr);//memory free

printf("\nnode deleted\n");//prints message

}

}

void deletion\_specified()//function defination

{

struct node \*ptr, \*temp;//structure declaration

int val;//variable declaration

printf("\n Enter the data after which the node is to be deleted : ");

scanf("%d", &val);//item scanned

ptr = head;//initialisation

while(ptr -> data != val)//condition check

ptr = ptr -> next;//acessing next node

if(ptr -> next == NULL)//condition check

{

printf("\nCan't delete\n");//prints message

}

else if(ptr -> next -> next == NULL)//condition check

{

ptr ->next = NULL;//next part of the node initialised

}

else

{

temp = ptr -> next;//next part of the node initialised

ptr -> next = temp -> next;//next part of the node initialised

temp -> next -> prev = ptr;//next part of the node initialised

free(temp);//memory free

printf("\nnode deleted\n");//prints message

}

}

void display()//function defination

{

struct node \*ptr;//structure declaration

printf("\n printing values...\n");//prints message

ptr = head;//initialisation

while(ptr != NULL)//condition check

{

printf("%d\n",ptr->data);

ptr=ptr->next;//acessing next node

}

}

void search()//function defination

{

struct node \*ptr;//structure declaration

int item,i=0,flag;//variable declaration

ptr = head; //initialisation

if(ptr == NULL)//condition check

{

printf("\nEmpty List\n");//prints message

}

else

{

printf("\nEnter item which you want to search?\n");

scanf("%d",&item);//item scanned

while (ptr!=NULL)//condition check

{

if(ptr->data == item)//condition check

{

printf("\nitem found at location %d ",i+1);

flag=0;//item found

break;//breaks out

}

else

{

flag=1;//item nor fouond

}

i++;//iterative element incremented

ptr = ptr -> next;//acessing next node

}

if(flag==1)//condition check

{

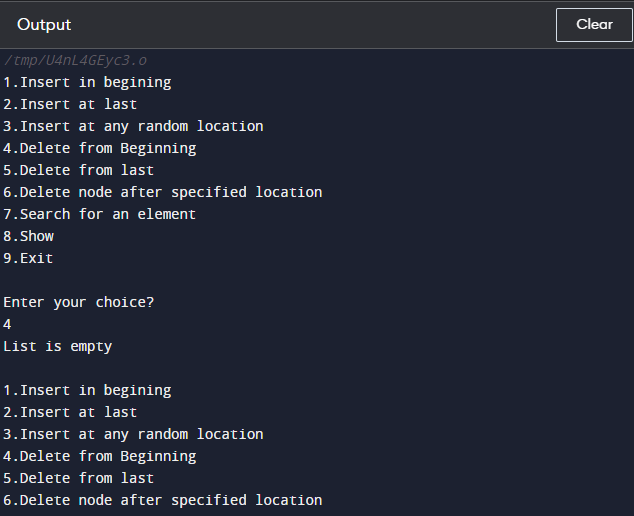
printf("\nItem not found\n");//prints message

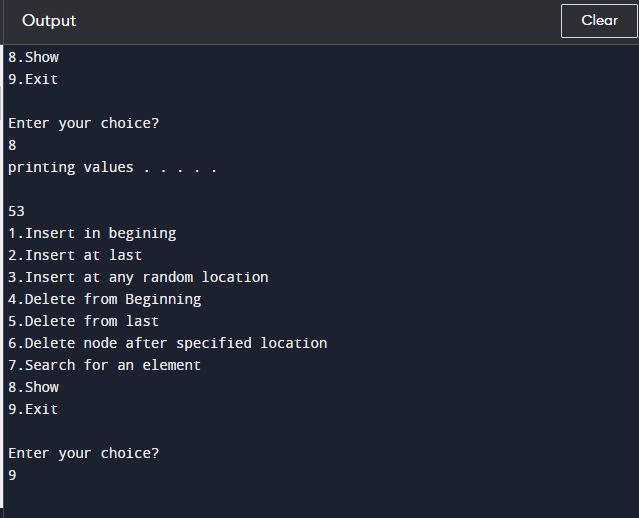
}

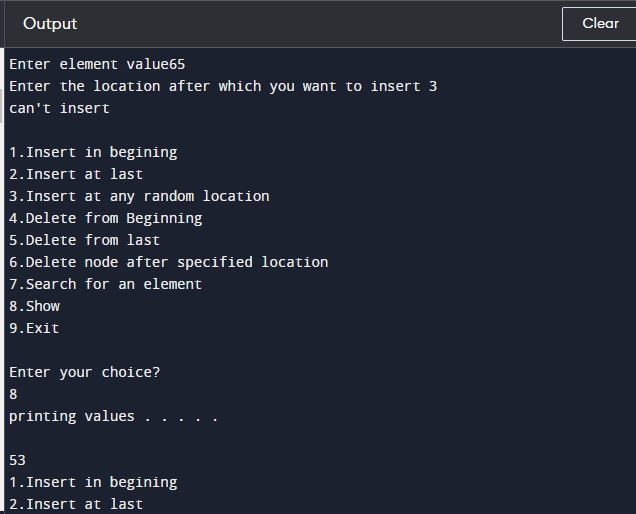
}

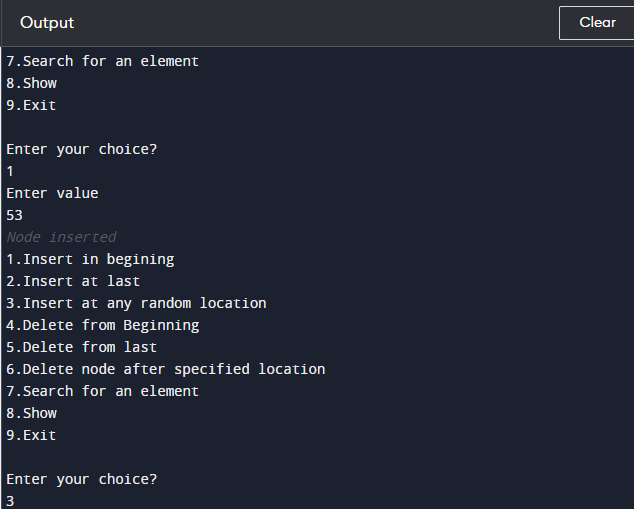
}

**Output:**

****

****

****

****