|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**EX.NO:1 Write a program to perform the following operations on singly linked list.**

**i) Creation ii) Insertion iii) Deletion iv) Traversal.**

**DATE:**

**Aim:** Write a program that uses functions to perform the following operations on Singly Linked List

(i)Creation (ii) Insertion (iii) Deletion (iv) Traversal.

**Description:**

**Linked List**

When we want to work with an unknown number of data values, we use a linked list data structure to organize that data. The linked list is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called "Node".

Linked List can be implemented as

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

**Single Linked List**

Simply a list is a sequence of data, and the linked list is a sequence of data linked with each other. The formal definition of a single linked list is as follows...

In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data field, and the next field. The data field is used to store actual value of the node and next field is used to store the address of next node in the sequence. The graphical representation of a node in a single linked list is as follows.

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

**1.Creation**

**Step 1** - Define a Node structure with two members data and next

**Step 2** - Define a Node pointer 'head' and set it to NULL.

**2.Insertion**

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

2.1 Inserting At Beginning of the list

2.2 Inserting At End of the list

2.3 Inserting At Specific location in the list

**2.1 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.

**2.2 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.

**2.3 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' **3. Deletion**

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

3.1 Deleting from Beginning of the list

3.2 Deleting from End of the list

3.3 Deleting a Specific Node

**3.1 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.

**3.2 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.

**3.3 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)).

**4. 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).

**Source Code:** To implement Singly Linked List

#include<iostream>

#include<stdlib.h>

void insertAtBeginning(int);

void insertAtEnd(int);

void insertBetween(int,int,int);

void display();

void removeBeginning();

void removeEnd();

void removeSpecific(int);

using namespace std;

struct Node

{

int data;

struct Node \*next;

}\*head = NULL;

int main()

{

int choice,value,choice1,loc1,loc2;

while(1){

mainMenu:

cout<<”\n\n\*\*\*\*\*\* MENU \*\*\*\*\*\*\n1. Insert\n2. Display\n3. Delete\n4. Exit\n Enter your choice: ";

cin>>choice;

switch(choice)

{

case 1: cout<<"Enter the value to be insert: ";

cin>>value;

while(1)

{

cout<<"Where you want to insert: \n1. At Beginning\n2. At End\n3.Between\nEnter your choice: ";

cin>>choice1;

switch(choice1)

{

case 1: insertAtBeginning(value);

break;

case 2: insertAtEnd(value);

break;

case 3: cout<<"Enter the two values where you wanto insert: ";

cin>>loc1>>loc2;

insertBetween(value,loc1,loc2);

break;

default: cout<<"\nWrong Input!! Try again!!!\n\n";

goto mainMenu;

}

goto subMenuEnd;

}

subMenuEnd:

break;

case 2: display();

break;

case 3: cout<<"Ho do you want to Delete: \n1. From Beginning\n2. From End\n3.Spesific\nEnter your choice: ";

cin>>choice1;

switch(choice1)

{

case 1: removeBeginning();

break;

case 2: removeEnd();

break;

case 3: cout<<"Enter the value which you wanto delete: ";

cin>>loc2;

removeSpecific(loc2);

break;

default: cout<<"\nWrong Input!! Try again!!!\n\n";

goto mainMenu;

}

break;

case 4: exit(0);

default: cout<<"\nWrong input!!! Try again!!\n\n";

}

}

}

void insertAtBeginning(int value)

{

struct Node \*newNode;

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

newNode->data = value;

if(head == NULL)

{

newNode->next = NULL;

head = newNode;

}

else

{

newNode->next = head;

head = newNode;

}

cout<<"\nOne node inserted!!!\n";

}

void insertAtEnd(int value)

{

struct Node \*newNode;

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

newNode->data = value;

newNode->next = NULL;

if(head == NULL)

head = newNode;

else

{

struct Node \*temp = head;

while(temp->next != NULL)

temp = temp->next;

temp->next = newNode;

}

cout<<"\nOne node inserted!!!\n";

}

void insertBetween(int value, int loc1, int loc2)

{

struct Node \*newNode;

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

newNode->data = value;

if(head == NULL)

{

newNode->next = NULL;

head = newNode;

}

else

{

struct Node \*temp = head;

while(temp->data != loc1 && temp->data != loc2)

temp = temp->next;

newNode->next = temp->next;

temp->next = newNode;

}

cout<<"\nOne node inserted!!!\n";

}

void removeBeginning()

{

if(head == NULL)

cout<<"\n\nList is Empty!!!";

else

{

struct Node \*temp = head;

if(head->next == NULL)

{

head = NULL;

free(temp);

}

else

{

head = temp->next;

free(temp);

cout<<"\nOne node deleted!!!\n\n";

}

}

}

void removeEnd()

{

if(head == NULL)

{

cout<<"\nList is Empty!!!\n";

}

else

{

struct Node \*temp1 = head,\*temp2;

if(head->next == NULL)

head = NULL;

else

{

while(temp1->next != NULL)

{

temp2 = temp1;

temp1 = temp1->next;

}

temp2->next = NULL;

}

free(temp1);

cout<<"\nOne node deleted!!!\n\n";

}

}

void removeSpecific(int delValue)

{

struct Node \*temp1 = head, \*temp2; while(temp1->data != delValue)

{

if(temp1 -> next == NULL)

{

cout<<"\nGiven node not found in the list!!!";

}

temp2 = temp1;

temp1 = temp1 -> next;

}

temp2 -> next = temp1 -> next;

free(temp1);

cout<<"\nOne node deleted!!!\n\n";

}

void display()

{

if(head == NULL)

{

cout<<"\nList is Empty\n";

}

else

{

struct Node \*temp = head; cout<<"\n\nList elements are - \n"; while(temp->next != NULL)

{

cout<<temp->data<<'\t'; temp = temp->next;

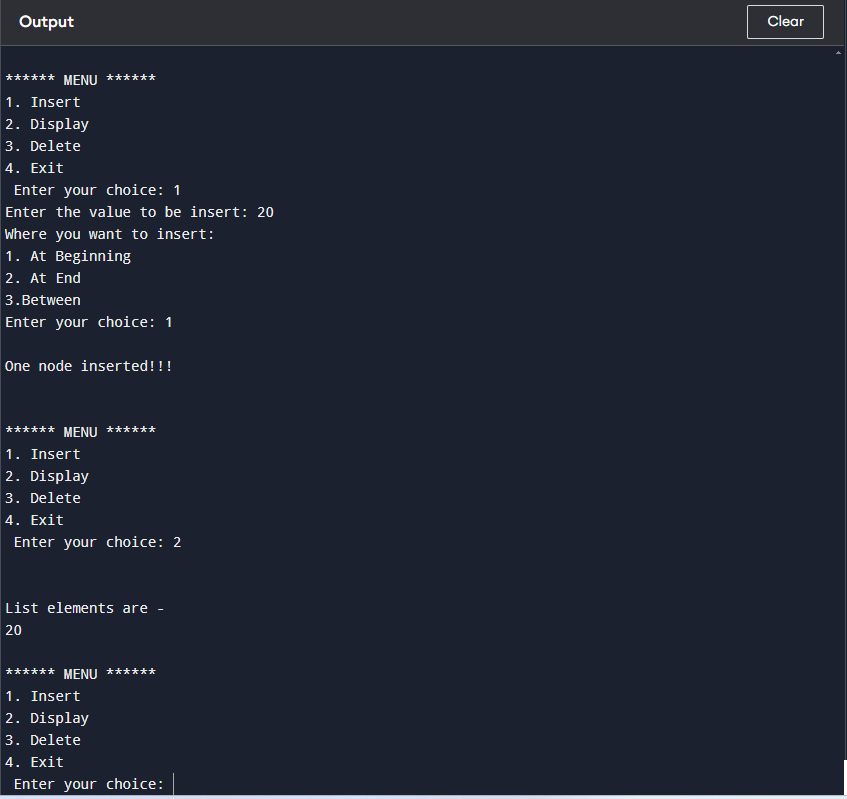
}

cout<<temp->data;

}

}

**2019-2020**



**EX.NO:2 Write a program to perform the following operations on doubly linked list.**

**i) Creation ii) Insertion iii) Deletion iv) Traversal in both ways.**

**DATE:**

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

**Description:**

**Double Linked List**

In a single linked list, every node has a link to its next node in the sequence. So, we can traverse from one node to another node only in one direction and we can not traverse back. We can solve this kind of problem by using a double linked list. A double linked list can be defined as follows...

In a double linked list, every node has a link to its previous node and next node. So, we can traverse forward by using the next field and can traverse backward by using the previous field. Every node in a double linked list contains three fields and they are shown in the following figure...

Here, **'link1'** field is used to store the address of the previous node in the sequence, **'link2'** field is used to store the address of the next node in the sequence and **'data'** field is used to store the actual value of that node.

**Operations on Double Linked List**

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

1. Creation
2. Insertion
3. Deletion
4. Display

**1.Creation**

**Step 1** - Define a Node structure with two members data and next

**Step 2** - Define a Node pointer 'head' and set it to NULL.

**2.Insertion**

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

2.1 Inserting At Beginning of the list

2.2 Inserting At End of the list

2.3 Inserting At Specific location in the list

**2.1 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** . **2.2 Inser ting 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 **Emp ty** , 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 **te mp → next** is equal to **NULL** ).

**Step 6 -** Assign **newNode** to **temp → next** and **temp** to **newNode → previous**.

**2.3 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 both **newNode → previous** &

**newNode → next** and set **newNode** to **head** .

**Step 4 -** If it is **not Em pty** 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.

**Step7 -** Assign **temp1→next** to **temp2** , **newNode** to **temp1 → next**,

**temp1** to **newNode → previous**, **temp2** to

**newNode → next** and **newNode** to **temp2 → previous**.

**3.Deletion**

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

3.1 Deleting from Beginning of the list

3.2 Deleting from End of the list

3.3 Deleting a Specific Node

**3.1 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** .

**3.2 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** .

**3.3 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** ).

**Step 11 -** 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)** ).

**4.Displaying**

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** ).

**Source Code:** To implement Doubly Linked List

#include<iostream.h>

#include<conio.h>

void insertAtBeginning(int);

void insertAtEnd(int);

void insertAtAfter(int,int);

void deleteBeginning();

void deleteEnd();

void deleteSpecific(int);

void display();

struct Node

{

int data;

struct Node \*previous, \*next;

}\*head = NULL;

void main()

{

int choice1, choice2, value, location;

clrscr();

while(1)

{

cout<<"\n\*\*\*\*\*\*\*\*\*\*\* MENU \*\*\*\*\*\*\*\*\*\*\*\*\*\n");

cout<<"1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter your choice: "; cin>>choice1;

switch(choice1)

{

case 1: cout<<"Enter the value to be inserted: ";

cin>>value;

while(1)

{

cout<<"\nSelect from the following Inserting options\n";

cout<<"1. At Beginning\n2. At End\n3. After a Node\n4. Cancel\nEnter

your choice: ";

cin>>choice2;

switch(choice2)

{

case 1: insertAtBeginning(value);

break;

case 2: insertAtEnd(value);

break;

case 3: cout<<"Enter the location after which you want to insert:"; cin>>location;

insertAfter(value,location);

break;

case 4: goto EndSwitch;

default: cout<<"\nPlease select correct Inserting option!!!\n";

}

}

case 2:

while(1)

{

cout<<"\nSelect from the following Deleting options\n";

cout<<"1. At Beginning\n2. At End\n3. Specific Node\n4. Cancel\nEnter your choice: ";

cin>>choice2;

switch(choice2)

{

case 1: deleteBeginning();

break;

case 2: deleteEnd();

break;

case 3: cout<<"Enter the Node value to be deleted: ";

cin>>location;

deleteSpecic(location);

break;

case 4: goto EndSwitch;

default: cout<<"\nPlease select correct Deleting option!!!\n"; }

}

EndSwitch: break;

case 3: display();

break;

case 4: exit(0);

default: cout<<"\nPlease select correct option!!!";

}

}

}

void insertAtBeginning(int value)

{

struct Node \*newNode;

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

newNode -> data = value;

newNode -> previous = NULL;

if(head == NULL)

{

newNode -> next = NULL;

head = newNode;

}

else

{

newNode -> next = head;

head = newNode;

}

cout<<"\nInsertion success!!!";

}

void insertAtEnd(int value)

{

struct Node \*newNode;

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

newNode -> data = value;

newNode -> next = NULL;

if(head == NULL)

{

newNode -> previous = NULL;

head = newNode;

}

else

{

struct Node \*temp = head;

while(temp -> next != NULL)

temp = temp -> next;

temp -> next = newNode;

newNode -> previous = temp;

}

cout<<"\nInsertion success!!!";

}

void insertAfter(int value, int location)

{

struct Node \*newNode;

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

if(head == NULL)

{

newNode -> previous = newNode -> next = NULL; head = newNode;

}

else

{

struct Node \*temp1 = head, temp2;

while(temp1 -> data != location)

{

if(temp1 -> next == NULL)

{

cout<<"Given node is not found in the list!!!"; goto EndFunction;

}

else

{

temp1 = temp1 -> next;

}

}

temp2 = temp1 -> next;

temp1 -> next = newNode;

newNode -> previous = temp1;

newNode -> next = temp2;

temp2 -> previous = newNode;

cout<<"\nInsertion success!!!";

}

EndFunction:

}

void deleteBeginning()

{

if(head == NULL)

cout<<"List is Empty!!! Deletion not possible!!!"; else

{

struct Node \*temp = head;

if(temp -> previous == temp -> next)

{

head = NULL;

free(temp);

}

else

{

head = temp -> next;

head -> previous = NULL;

free(temp);

}

cout<<"\nDeletion success!!!";

}

}

void deleteEnd()

{

if(head == NULL)

cout<<”List is Empty!!! Deletion not possible!!!"; else

{

struct Node \*temp = head;

if(temp -> previous == temp -> next)

{

head = NULL;

free(temp);

}

else

{

while(temp -> next != NULL)

temp = temp -> next;

temp -> previous -> next = NULL;

free(temp);

}

cout<<"\nDeletion success!!!";

}

}

void deleteSpecific(int delValue)

{

if(head == NULL)

cout<<"List is Empty!!! Deletion not possible!!!";

else

{

struct Node \*temp = head;

while(temp -> data != delValue)

{

if(temp -> next == NULL)

{

cout<<"\nGiven node is not found in the list!!!"; goto FuctionEnd;

}

else

{

temp = temp -> next;

}

}

if(temp == head)

{

head = NULL;

free(temp);

}

else

{

temp -> previous -> next = temp -> next; free(temp);

}

cout<<"\nDeletion success!!!";

}

FuctionEnd:

}

void display()

{

if(head == NULL)

cout<<"\nList is Empty!!!";

else

{

struct Node \*temp = head; cout<<"\nList elements are: \n";

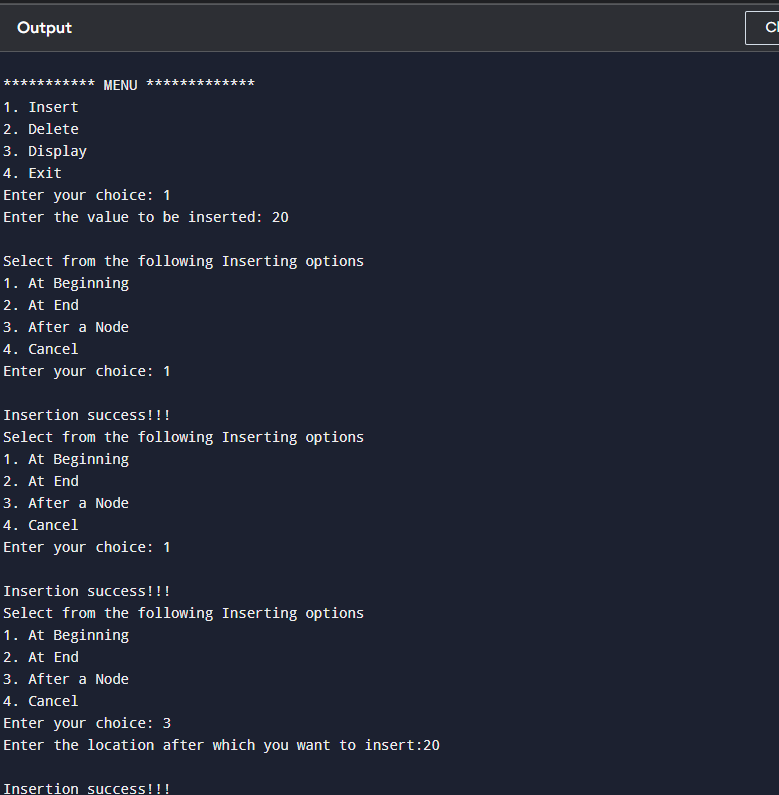
cout<<"NULL <--- "; while(temp -> next != NULL) {

cout<<temp - > data<<”\t ”;

}

cout<<temp -> data; } }

OUTPUT:



**EX.NO:3 Write a program that implements stack (its operations) using i) Arrays ii) linked list.**

**DATE:**

**Aim:** Write a program that implement stack (its operations) using

(i)Arrays (ii)Linked list(Pointers).

**Description:**

**Stack**

1. tack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at a single position which is known as " **top** ".
2. hat means, a new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based

on **LIFO** (Last In First Out) principle.

In a stack, the insertion operation is performed using a function called **"push"** and deletion operation is performed using a function called **"pop"** .

In the figure, PUSH and POP operations are performed at a top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e., at Top).

A stack data structure can be defined as follows...

Stack can also be defined as

**"A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle".**

**Operations on a Stack**

The following operations are performed on the stack...

1. Push (To insert an element on to the stack)
2. Pop (To delete an element from the stack)
3. Display (To display elements of the stack)

Stack data structure can be implemented in two ways. They are as follows...

1. Using Arrays
2. Using Linked List

**Stack Using Arrays**

A stack data structure can be implemented using a one-dimensional array. But stack implemented using array stores only a fixed number of data values. This implementation is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **LIFO principle** with the help of a variable called **'top'** .

Initially, the top is set to -1. Whenever we want to insert a value into the stack, increment

the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

**Stack Operations**

We can Perform the following Operations on Stack

1. Push()
2. Pop()
3. Display()
4. stack can be implemented using array as follows...
5. efore implementing actual operations, first follow the below steps to create an empty stack.

**Step 1 -** Include all the **header files** which are used in the program and define a

constant **'SIZE'** with specific value.

**Step 2 -** Declare all the **functions** used in stack implementation.

**Step 3 -** Create a one dimensional array with fixed size ( **int stack[SIZE]** )

**Step 4 -** Define a integer variable **'top'** and initialize with **'-1'** . ( **int top = -1**)

**Step 5 -** In main method, display menu with list of operations and make suitable function

calls to perform operation selected by the user on the stack.

**1.Push(value) - Inserting value into the stack**

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at **top** position. Push function takes one integer value as parameter and inserts that value into the stack.

We can use the following steps to push an element on to the stack...

**Step 1 -** Check whether **stack** is **FULL** . ( **top == SIZE -1**)

**Step 2 -** If it is **FULL** , then display **"Stack is FULL!!! Insertion is not possible!!!"** and

terminate the function.

**Step 3 -** If it is **NOT FULL** , then increment **top** value by one ( **top++** ) and set stack[top]

to value ( **stack[top] = value** ).

**2.Pop() - Delete a value from the Stack**

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

**Step 1 -** Check whether **stack** is **EMPTY** . ( **top == -1**)

**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 delete **stack[top]** and decrement **top** value by one ( **top --**).

**3.Display() - Displays the E lements of a Stack**

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

**Step 1 -** Check whether **stack** is **EMPTY** . ( **top == -1**)

**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 variable '**i**' and initialize with top.

Display **stack[i]** value and decrement **i** value by one ( **i--**).

**Step 4 -** Repeat above step until **i** value becomes '0'.

**Source Code:** To implement Stack Using Arrays

#include<iostream.h>

#include<iostream>

#define SIZE 10

void push(int);

void pop();

void display();

int stack[SIZE], top = -1;

using namespace std;

int main()

{

int value, choice;

while(1){

cout<<"\n\n\*\*\*\*\* MENU \*\*\*\*\*\n";

cout<<"1. Push\n2. Pop\n3. Display\n4. Exit";

cout<<"\nEnter your choice: ";

cin>>choice;

switch(choice)

{

case 1: cout<<"Enter the value to be insert: ";

cin>>value;

push(value);

break;

case 2: pop();

break;

case 3: display();

break;

case 4: exit(0);

default: cout<<"\nWrong selection!!! Try again!!!";

}

}

}

void push(int value)

{

if(top == SIZE-1)

cout<<"\nStack is Full!!! Insertion is not possible!!!";

else

{

top++;

stack[top] = value;

cout<<"\nInsertion success!!!";

}

}

void pop()

{

if(top == -1)

cout<<"\nStack is Empty!!! Deletion is not possible!!!";

else

{

cout<<"\nDeleted :"<<stack[top];

top--;

}

}

void display()

{

if(top == -1)

cout<<"\nStack is Empty!!!";

else

{

int i;

cout<<"\nStack elements are:\n";

for(i=top; i>=0; i--)

cout<<stack[i];

}

}

**Output:**

**(ii) Stack Using Linked Li st**

The major problem with the stack implemented using an arrays is, it works only for a fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself.

1. tack implemented using an array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using a linked list data structure.
2. he stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as ' **top** ' element. That means every newly inserted element is pointed by ' **top** '. Whenever we want to remove an element from the stack, simply remove the node which is pointed by ' **top** ' by moving '**top** ' to its previous node in the list. The **next** field of the first element must be always **NULL** .

**Stack Operations using Linked List**

We can Perform the Following Operations on Stack Using Linked List (i.e)

1. Push()
2. Pop()
3. Display()

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

**Step 1 -** Include all the **header files** which are used in the program. And declare all

the **user defined functions** .

**Step 2 -** Define a '**Node** ' structure with two members **data** and **next** .

**Step 3 -** Define a **Node** pointer '**top** ' and set it to **NULL** .

**Step 4 -** Implement the **main** method by displaying Menu with list of operations and

make suitable function calls in the **main** method.

**1.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** .

**2.Pop() - Deleting an Element from a Stack**

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

**Step 1 -** Check whether **stac k** 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 5 -** Finally, delete '**temp** '. ( **free(temp)** ).

**3.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 5 -** Finally! Display '**temp → data** ---> **NULL** '.

**Source Code:** To Implement Stack Using Linked List

#include<iostream>

struct Node

{

int data;

struct Node \*next;

}\*top = NULL;

void push(int);

void pop();

void display();

using namespace std;

int main()

{

int choice, value;

cout<<"\n:: Stack using Linked List ::\n";

while(1)

{

cout<<"\n\*\*\*\*\*\* MENU \*\*\*\*\*\*\n";

cout<<"1. Push\n2. Pop\n3. Display\n4. Exit\n"; cout<<"Enter your choice: ";

cin>>choice;

switch(choice)

{

case 1: cout<<"Enter the value to be insert: ";

cin>>value;

push(value);

break;

case 2: pop();

break;

case 3: display(); break;

case 4: exit(0);

default: cout<<"\nWrong selection!!! Please try again!!!\n";

}

}

}

void push(int value)

{

struct Node \*newNode;

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

newNode->data = value;

if(top == NULL)

newNode->next = NULL;

else

newNode->next = top;

top = newNode;

cout<<"\nInsertion is Success!!!\n";

}

void pop()

{

if(top == NULL)

cout<<"\nStack is Empty!!!\n";

else

{

struct Node \*temp = top;

cout<<"\nDeleted element:", temp->data;

top = temp->next;

free(temp);

}

}

void display()

{

if(top == NULL)

cout<<"\nStack is Empty!!!\n";

else

{

struct Node \*temp = top;

while(temp->next != NULL)

{

cout<<temp->data; temp = temp -> next;

}

cout<<temp->data;

}

}

OUTPUT:

**EX.NO:4 Write a programs that implements Queue (its operations) using i) Arrays ii) linked list**

**DATE:**

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

**Description:**

**Queue Using Arrays**

A queue data structure can be implemented using one dimensional array. The queue implemented using array stores only fixed number of data values.

The implementation of queue data structure using array is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle** with the help of variables **'front'** and '**rear** '. Initially both '**front** ' and '**rear** ' are set to -1.

Whenever, we want to insert a new value into the queue, increment ' **rear** ' value by one and then insert at that position. Whenever we want to delete a value from the queue, then delete the element which is at 'front' position and increment 'front' value by one.

**Queue Operations using Array**

We can Perform the following operations on Queue

1. enQueue()
2. deQueue()
3. Display()

Before we implement actual operations, first follow the below steps to create an empty queue.

**Step 1 -** Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.

**Step 2 -** Declare all the **user defined functions** which are used in queue

implementation.

**Step 3 -** Create a one dimensional array with above defined SIZE ( **int**

**queue[SIZE]** ).

**Step 4 -** Define two integer variables **'front'** and '**rear** ' and initialize both with **'-1'** .

( **int front = -1, rear = -1**).

**Step 5 -** Then implement main method by displaying menu of operations list and make

suitable function calls to perform operation selected by the user on queue.

**1.enQueue(value) - Inserting value into the queue**

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as a parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

**Step 1 -** Check whether **queue** is **FULL** . ( **rear == SIZE -1**)

**Step 2 -** If it is **FULL** , then display **"Queue is FULL!!! Insertio n is not**

**possible!!!"** and terminate the function.

**Step 3 -** If it is **NOT FULL** , then increment **rear** value by one ( **rear++** ) and

set **queue[rear]** = **value** .

**2.deQueue() - Deleting a value from the Queue**

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from **front** position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

**Step 1 -** Check whether **queue** is **EMPTY** . (**front == rear** )

**Step 2 -** If it is **EMPTY** , then display **"Queue is EMPTY!!! Deletion is not**

**possible!!!"** and terminate the function.

**Step 3 -** If it is **NOT EMPTY** , then increment the **front** value by one ( **fron t ++** ). Then

display **queue[front]** as deleted element. Then check whether

both **front** and **rear** are equal ( **front** == **rear** ), if it **TRUE** , then set

both **front** and **rear** to '**-1**' ( **front** = **rear** = **-1**).

**3.Display() - Displays the elements of a Queue**

We can use the following steps to display the elements of a queue...

**Step 1 -** Check whether **queue** is **EMPTY** . (**front == rear** )

**Step 2 -** If it is **EMPTY** , then display **"Queue is EMPTY!!!"** and terminate the function.

**Step 3 -** If it is **NOT EMPTY** , then define an integer variable '**i** ' and set '**i** = **front+1** '.

**Step 4 -** Display '**queue[i]** ' value and increment '**i**' value by one ( **i++**). Repeat the same

until '**i** ' value reaches to **rear** ( **i** <= **rear** ).

**Source Code:** To implement Queue using Arrays

#include<iostream>

#define SIZE 10

void enQueue(int);

void deQueue();

void display();

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

using namespace std;

int main()

{

int value, choice;

while(1){

cout<<"\n\n\*\*\*\*\* MENU \*\*\*\*\*\n";

cout<<"1. Insertion\n2. Deletion\n3. Display\n4. Exit";

cout<<"\nEnter your choice: ";

cin>>choice;

switch(choice){

case 1: cout<<"Enter the value to be insert: ";

cin>>value;

enQueue(value);

break;

case 2: deQueue();

break;

case 3: display();

break;

default: cout<<"\nWrong selection!!! Try again!!!";

}

}

}

void enQueue(int value){

if(rear == SIZE-1)

cout<<"\nQueue is Full!!! Insertion is not possible!!!";

else{

if(front == -1)

front = 0;

rear++;

queue[rear] = value;

cout<<"\nInsertion success!!!";

}

}

void deQueue()

{

if(front == rear)

cout<<"\nQueue is Empty!!! Deletion is not possible!!!";

else

{

cout<<"\nDeleted : %d", queue[front];

front++;

if(front == rear)

front = rear = -1;

}

}

void display()

{

if(rear == -1)

cout<<"\nQueue is Empty!!!";

else

{

int i;

cout<<"\nQueue elements are:\n";

for(i=front; i<=rear; i++)

cout<<queue[i];

}

}

**Output:**

**II. Queue Using Linked List**

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use.

A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation).

The Queue implemented using linked list can organize as many data values as we want. In linked list implementation of a queue, the last inserted node is always pointed by ' **rear** ' and the first node is always pointed by ' **front** '.

**Queue Operations using Array**

We can Perform the following operations on Queue

1. enQueue()
2. deQueue()
3. Display()

To implement queue using linked list, we need to set the following things before implementing actual operations.

**Step 1 -** Include all the **header files** which are used in the program. And declare

all the **user defined functions** .

**Step 2 -** Define a '**Node** ' structure with two members **data** and **next** .

**Step 3 -** Define two **Node** pointers '**front** ' and '**rear** ' and set both to **NULL** .

**Step 4 -** Implement the **main** method by displaying Menu of list of operations and

make suitable function calls in the **main** method to perform user selected operation.

**1.enQueue(value) - Inserting an element into the Queue**

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

**Step 1 -** Create a **newNode** with given value and set '**newNode → next** ' to **NULL** .

**Step 2 -** Check whether queue is **Empty** ( **rear** == **NULL** )

**Step 3 -** If it is **Empty** then, set **front** = **newNode** and **rear** = **newNode** .

**Step 4 -** If it is **Not Empty** then, set **rear → next** = **newNode** and **rear** = **newNode** . **2.deQueue () - Deleting an Element from Queue**

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

**Step 1 -** Check whether **queue** is **Empty** (**front == NULL** ).

**Step 2 -** If it is **Empty** , then display **"Queue is Empty!!! Deletion is not**

**possible!!!"** and terminate from the function

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

**Step 4 -** Then set '**front** = **front → next**' and delete '**temp** ' ( **free(temp)** ).

**3.Display() - Displaying the elements of Queue**

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

**Step 1 -** Check whether queue is **Empty** (**front** == **NULL** ).

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

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

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

**Step 5 -** Finally! Display '**temp → data** ---> **NULL** '.

**Source Code: To implement Queue using Linked List**

#include<iostream>

struct Node

{

int data;

struct Node \*next;

}\*front = NULL,\*rear = NULL;

void insert(int);

void delete1();

void display();

using namespace std;

int main()

{

int choice, value;

cout<<"\n:: Queue Implementation using Linked List ::\n"; while(1){

cout<<"\n\*\*\*\*\*\* MENU \*\*\*\*\*\*\n";

cout<<"1. Insert\n2. Delete\n3. Display\n4. Exit\n"; cout<<"Enter your choice: ";

cin>>choice;

switch(choice){

case 1: cout<<"Enter the value to be insert: ";

cin>>value;

insert(value);

break;

case 2: delete1();

break;

case 3: display(); break;

case 4: exit(0);

default: cout<<"\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;

}

cout<<"\nInsertion is Success!!!\n";

}

void delete1()

{

if(front == NULL)

cout<<"\nQueue is Empty!!!\n";

else{

struct Node \*temp = front;

front = front -> next;

cout<<"\nDeleted element: %d\n", temp->data;

free(temp);

}

}

void display()

{

if(front == NULL)

cout<<"\nQueue is Empty!!!\n";

else{

struct Node \*temp = front;

while(temp->next != NULL){

cout<<temp->data;

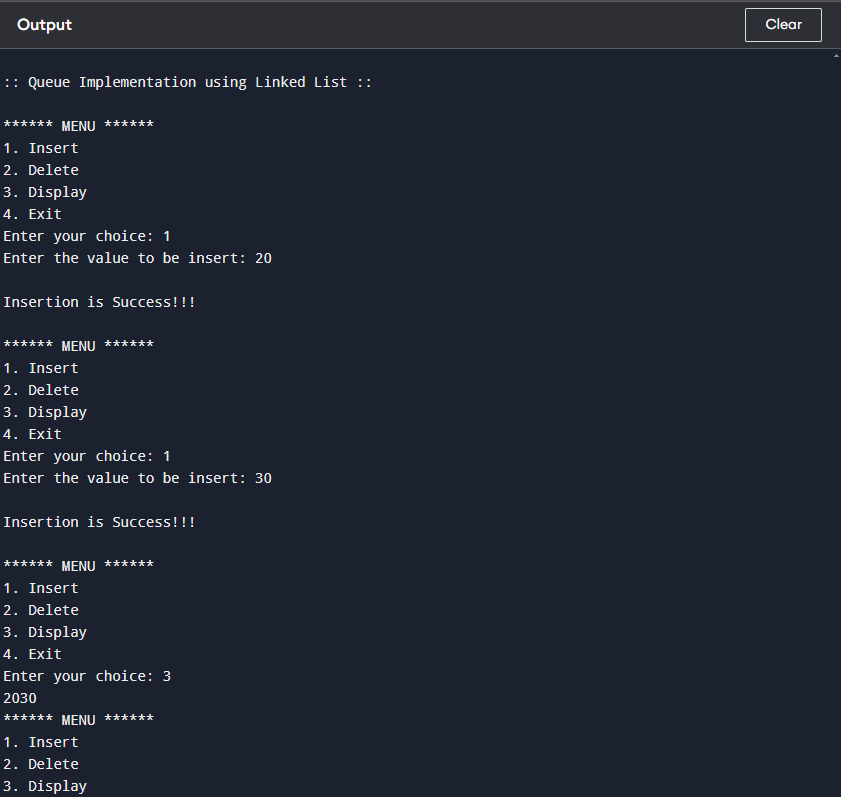
temp = temp -> next;

}

cout<<temp->data;

}

}



**EX.NO:5 Write C program that implements the Quick sort method to sort a given**

**list of integers in ascending order. B-Tree.**

**DATE:**

**Quicksort Algorithm**

1. An array is divided into subarrays by selecting a **pivot element** (element selected from the array).  
   While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

**Source Code:**

// Quick sort in C

#include <stdio.h>

// function to swap elements

void swap(int \*a, int \*b) {

int t = \*a;

\*a = \*b;

\*b = t;

}

// function to find the partition position

int partition(int array[], int low, int high) {

// select the rightmost element as pivot

int pivot = array[high];

// pointer for greater element

int i = (low - 1);

// traverse each element of the array

// compare them with the pivot

for (int j = low; j < high; j++) {

if (array[j] <= pivot) {

// if element smaller than pivot is found

// swap it with the greater element pointed by i

i++;

// swap element at i with element at j

swap(&array[i], &array[j]);

}

}

// swap the pivot element with the greater element at i

swap(&array[i + 1], &array[high]);

// return the partition point

return (i + 1);

}

void quickSort(int array[], int low, int high) {

if (low < high) {

// find the pivot element such that

// elements smaller than pivot are on left of pivot

// elements greater than pivot are on right of pivot

int pi = partition(array, low, high);

// recursive call on the left of pivot

quickSort(array, low, pi - 1);

// recursive call on the right of pivot

quickSort(array, pi + 1, high);

}

}

// function to print array elements

void printArray(int array[], int size) {

for (int i = 0; i < size; ++i) {

printf("%d ", array[i]);

}

printf("\n");

}

// main function

int main() {

int data[] = {8, 7, 2, 1, 0, 9, 6};

int n = sizeof(data) / sizeof(data[0]);

printf("Unsorted Array\n");

printArray(data, n);

// perform quicksort on data

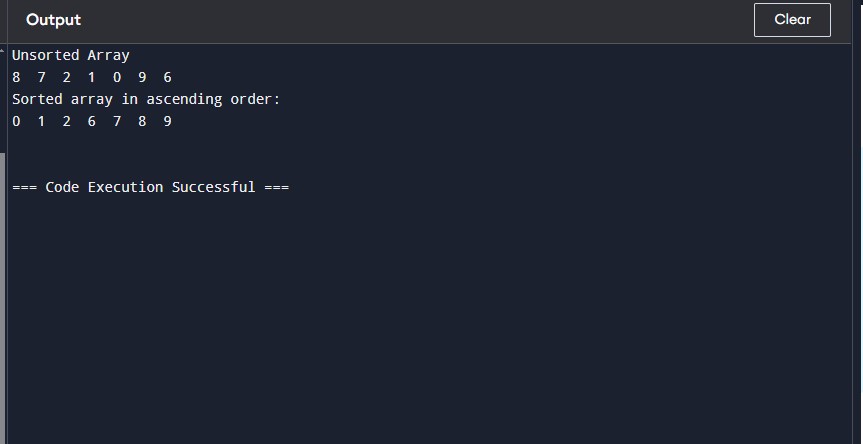
quickSort(data, 0, n - 1);

printf("Sorted array in ascending order: \n");

printArray(data, n);

}

OUTPUT:



**EX.NO:6 Write C program that implement the Merge sort method to sort a given**

**list of integers in ascending order.**

**DATE:**

**Algorithm:**

Using the **Divide and Conquer** technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].

**Divide**

If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

**Conquer**

In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

**Combine**

When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r].

1. Create copies of the subarrays L ← A[p..q] and M ← A[q+1..r].
2. Create three pointers i, j and k
   1. i maintains current index of L, starting at 1
   2. j maintains current index of M, starting at 1
   3. k maintains the current index of A[p..q], starting at p.
3. Until we reach the end of either L or M, pick the larger among the elements from L and M and place them in the correct position at A[p..q]
4. When we run out of elements in either L or M, pick up the remaining elements and put in A[p..q]

**Source Code:**

// Merge sort in C

#include <stdio.h>

// Merge two subarrays L and M into arr

void merge(int arr[], int p, int q, int r) {

// Create L ← A[p..q] and M ← A[q+1..r]

int n1 = q - p + 1;

int n2 = r - q;

int L[n1], M[n2];

for (int i = 0; i < n1; i++)

L[i] = arr[p + i];

for (int j = 0; j < n2; j++)

M[j] = arr[q + 1 + j];

// Maintain current index of sub-arrays and main array

int i, j, k;

i = 0;

j = 0;

k = p;

// Until we reach either end of either L or M, pick larger among

// elements L and M and place them in the correct position at A[p..r]

while (i < n1 && j < n2) {

if (L[i] <= M[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = M[j];

j++;

}

k++;

}

// When we run out of elements in either L or M,

// pick up the remaining elements and put in A[p..r]

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = M[j];

j++;

k++;

}

}

// Divide the array into two subarrays, sort them and merge them

void mergeSort(int arr[], int l, int r) {

if (l < r) {

// m is the point where the array is divided into two subarrays

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

// Merge the sorted subarrays

merge(arr, l, m, r);

}

}

// Print the array

void printArray(int arr[], int size) {

for (int i = 0; i < size; i++)

printf("%d ", arr[i]);

printf("\n");

}

// Driver program

int main() {

int arr[] = {6, 5, 12, 10, 9, 1};

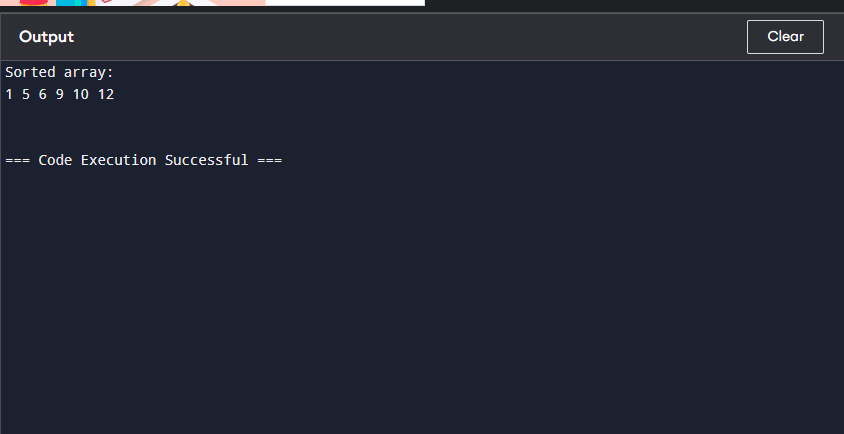
int size = sizeof(arr) / sizeof(arr[0]);

mergeSort(arr, 0, size - 1);

printf("Sorted array: \n");

printArray(arr, size);

}



**EX.NO:7 Write C program that implement the SHELL sort method to sort a given list**

**of integers in ascending order..**

**DATE:**

**Algorithm:**

The interval between the elements is reduced based on the sequence used. Some of the optimal sequences that can be used in the shell sort algorithm are:

* **Shell's original sequence**: N/2 , N/4 , …, 1
* **Knuth's increments**: 1, 4, 13, …, (3k – 1) / 2
* **Sedgewick's increments**: 1, 8, 23, 77, 281, 1073, 4193, 16577...4j+1+ 3·2j+ 1
* **Hibbard's increments**: 1, 3, 7, 15, 31, 63, 127, 255, 511…
* **Papernov & Stasevich increment**: 1, 3, 5, 9, 17, 33, 65,...
* **Pratt**: 1, 2, 3, 4, 6, 9, 8, 12, 18, 27, 16, 24, 36, 54, 81....

**Source Code:**

// Shell Sort in C programming

#include <stdio.h>

// Shell sort

void shellSort(int array[], int n) {

// Rearrange elements at each n/2, n/4, n/8, ... intervals

for (int interval = n / 2; interval > 0; interval /= 2) {

for (int i = interval; i < n; i += 1) {

int temp = array[i];

int j;

for (j = i; j >= interval && array[j - interval] > temp; j -= interval) {

array[j] = array[j - interval];

}

array[j] = temp;

}

}

}

// Print an array

void printArray(int array[], int size) {

for (int i = 0; i < size; ++i) {

printf("%d ", array[i]);

}

printf("\n");

}

// Driver code

int main() {

int data[] = {9, 8, 3, 7, 5, 6, 4, 1};

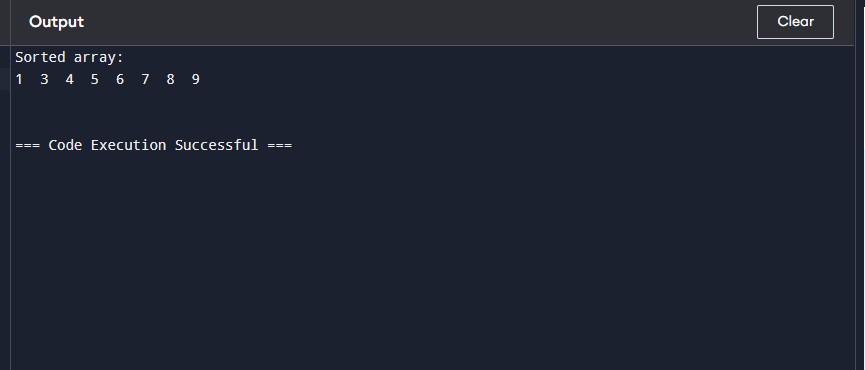
int size = sizeof(data) / sizeof(data[0]);

shellSort(data, size);

printf("Sorted array: \n");

printArray(data, size);

}



**EX.NO:8 Write a program to perform the following:**

**i) Creating a Binary Tree of integers**

**ii) Traversing the above binary tree in preorder, inorder and postorder..**

**DATE:**

**i)**

**Algorithm:**

1. Declare and initialize necessary variables  
2. Read the data item to be inserted in the tree say x.  
3. Create a new node with its left and right pointers to null.  
4. Assign data x to info field of new node.  
5. If (tree == NULL)  
then tree = address of new node  
else if (x < tree -> info)  
if(tree->left == NULL)  
then tree->left = new node  
else  
tree = tree->left  
repeat step 5.  
else if(x>tree->info)  
if(tree->right == NULL)  
then tree->right = new node  
else  
tree = tree->right  
repeat step 5  
else if(x == tree->info)  
print “Duplicated data” and exit  
6. For next insertion, goto step 5.

**ii)**

## **In-order Traversal**

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

***D → B → E → A → F → C → G***

### **Algorithm**

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

## **Pre-order Traversal**

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

***A → B → D → E → C → F → G***

### **Algorithm**

Until all nodes are traversed −

**Step 1** − Visit root node.

**Step 2** − Recursively traverse left subtree.

**Step 3** − Recursively traverse right subtree.

## **Post-order Traversal**

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

***D → E → B → F → G → C → A***

### **Algorithm**

Until all nodes are traversed −

**Step 1** − Recursively traverse left subtree.

**Step 2** − Recursively traverse right subtree.

**Step 3** − Visit root node.

**Source Code:**

#include <iostream>

#include <cstdlib>

using namespace std;

struct tree {

int info;

tree \*Left, \*Right;

};

tree\* root;

class Binary\_tree {

public:

Binary\_tree();

void insert1(int);

tree\* insert2(tree\*, tree\*);

void Delete(int);

void pretrav(tree\*);

void intrav(tree\*);

void posttrav(tree\*);

};

Binary\_tree::Binary\_tree()

{

root = NULL;

}

tree\* Binary\_tree::insert2(tree\* temp, tree\* newnode)

{

if (temp == NULL) {

temp = newnode;

}

else if (temp->info < newnode->info) {

insert2(temp->Right, newnode);

if (temp->Right == NULL)

temp->Right = newnode;

}

else {

insert2(temp->Left, newnode);

if (temp->Left == NULL)

temp->Left = newnode;

}

return temp;

}

void Binary\_tree::insert1(int n)

{

tree \*temp = root, \*newnode;

newnode = new tree;

newnode->Left = NULL;

newnode->Right = NULL;

newnode->info = n;

root = insert2(temp, newnode);

}

void Binary\_tree::pretrav(tree\* t = root)

{

if (root == NULL) {

cout << "Nothing to display";

}

else if (t != NULL) {

cout << t->info << " ";

pretrav(t->Left);

pretrav(t->Right);

}

}

void Binary\_tree::intrav(tree\* t = root)

{

if (root == NULL) {

cout << "Nothing to display";

}

else if (t != NULL) {

intrav(t->Left);

cout << t->info << " ";

intrav(t->Right);

}

}

void Binary\_tree::posttrav(tree\* t = root)

{

if (root == NULL) {

cout << "Nothing to display";

}

else if (t != NULL) {

posttrav(t->Left);

posttrav(t->Right);

cout << t->info << " ";

}

}

void Binary\_tree::Delete(int key)

{

tree \*temp = root, \*parent = root, \*marker;

if (temp == NULL)

cout << "The tree is empty" << endl;

else {

while (temp != NULL && temp->info != key) {

parent = temp;

if (temp->info < key) {

temp = temp->Right;

}

else {

temp = temp->Left;

}

}

}

marker = temp;

if (temp == NULL)

cout << "No node present";

else if (temp == root) {

if (temp->Right == NULL && temp->Left == NULL) {

root = NULL;

}

else if (temp->Left == NULL) {

root = temp->Right;

}

else if (temp->Right == NULL) {

root = temp->Left;

}

else {

tree\* temp1;

temp1 = temp->Right;

while (temp1->Left != NULL) {

temp = temp1;

temp1 = temp1->Left;

}

if (temp1 != temp->Right) {

temp->Left = temp1->Right;

temp1->Right = root->Right;

}

temp1->Left = root->Left;

root = temp1;

}

}

else {

if (temp->Right == NULL && temp->Left == NULL) {

if (parent->Right == temp)

parent->Right = NULL;

else

parent->Left = NULL;

}

else if (temp->Left == NULL) {

if (parent->Right == temp)

parent->Right = temp->Right;

else

parent->Left = temp->Right;

}

else if (temp->Right == NULL) {

if (parent->Right == temp)

parent->Right = temp->Left;

else

parent->Left = temp->Left;

}

else {

tree\* temp1;

parent = temp;

temp1 = temp->Right;

while (temp1->Left != NULL) {

parent = temp1;

temp1 = temp1->Left;

}

if (temp1 != temp->Right) {

temp->Left = temp1->Right;

temp1->Right = parent->Right;

}

temp1->Left = parent->Left;

parent = temp1;

}

}

delete marker;

}

int main()

{

Binary\_tree bt;

int choice, n, key;

while (1) {

cout << "\n\t1. Insert\n\t2. Delete\n\t3. Preorder Traversal\n\t4. Inorder Treversal\n\t5. Postorder Traversal\n\t6. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter item: ";

cin >> n;

bt.insert1(n);

break;

case 2:

cout << "Enter element to delete: ";

cin >> key;

bt.Delete(key);

break;

case 3:

cout << endl;

bt.pretrav();

break;

case 4:

cout << endl;

bt.intrav();

break;

case 5:

cout << endl;

bt.posttrav();

break;

case 6:

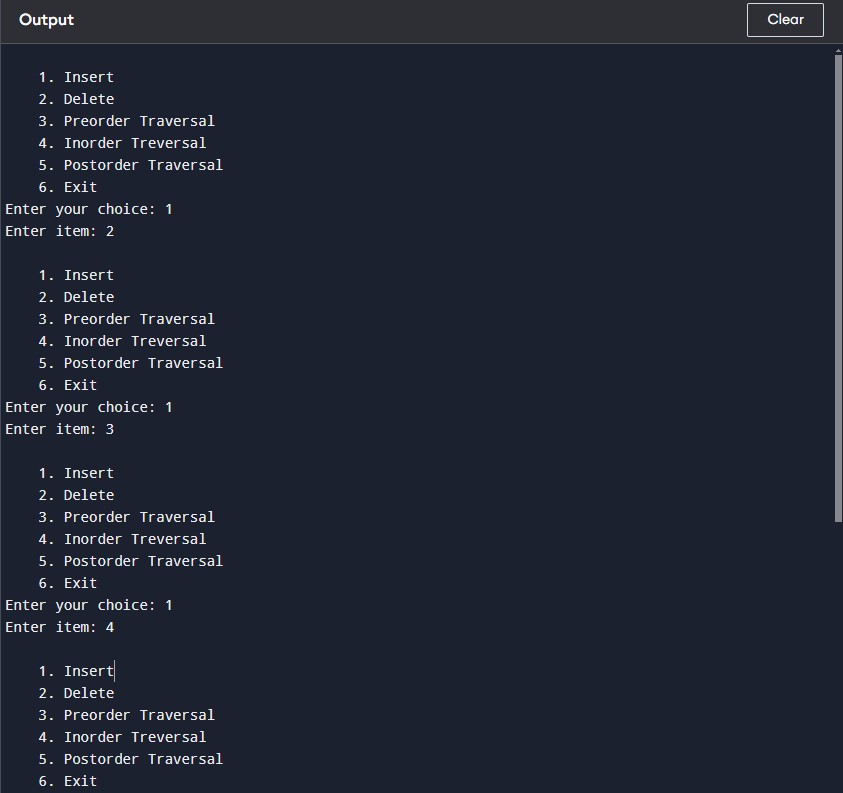
exit(0);

}

}

return 0;

}



**EX.NO:9 Write a program to perform the following:**

**i) Creating a AVL Tree of integers**

**ii) Traversing the above AVL tree in preorder, inorder and postorder.**

**DATE:**

**Algorithm:**

1. **and (ii)**

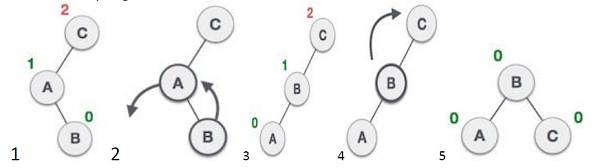
**height(avl \*)** : It calculate the height of the given AVL tree.

**difference(avl \*)**: It calculate the difference between height of sub trees of given tree

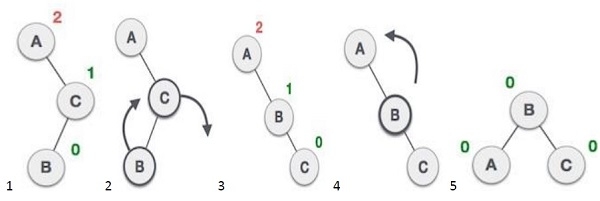
**avl \*rr\_rotat(avl \*)**: A right-right rotation is a combination of right rotation followed by right rotation.

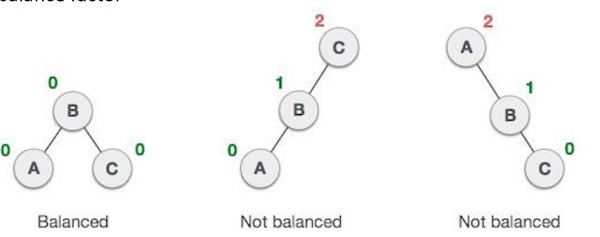
**avl \*ll\_rotat(avl \*)**: A left-left rotation is a combination of left rotation followed by left rotation.

**avl \*lr\_rotat(avl\*)**: A left-right rotation is a combination of left rotation followed by right rotation.



**avl \*rl\_rotat(avl \*)**: It is a combination of right rotation followed by left rotation.



avl \* balance(avl \*): It perform balance operation to the tree by getting balance factor

**avl \* insert(avl\*, int)**: It perform insert operation. Insert values in the tree using this function.

**show(avl\*, int):** It display the values of the tree.

**inorder(avl \*)**: Traverses a tree in an in-order manner.

**preorder(avl \*)**: Traverses a tree in a pre-order manner.

**postorder(avl\*)**: Traverses a tree in a post-order manner

**Source Code:**

#include<iostream>

#include<cstdio>

#include<sstream>

#include<algorithm>

#define pow2(n) (1 << (n))

using namespace std;

struct avl {

   int d;

   struct avl \*l;

   struct avl \*r;

}\*r;

class avl\_tree {

   public:

      int height(avl \*);

      int difference(avl \*);

      avl \*rr\_rotat(avl \*);

      avl \*ll\_rotat(avl \*);

      avl \*lr\_rotat(avl\*);

      avl \*rl\_rotat(avl \*);

      avl \* balance(avl \*);

      avl \* insert(avl\*, int);

      void show(avl\*, int);

      void inorder(avl \*);

      void preorder(avl \*);

      void postorder(avl\*);

      avl\_tree() {

         r = NULL;

      }

};

int avl\_tree::height(avl \*t) {

   int h = 0;

   if (t != NULL) {

      int l\_height = height(t->l);

      int r\_height = height(t->r);

      int max\_height = max(l\_height, r\_height);

      h = max\_height + 1;

   }

   return h;

}

int avl\_tree::difference(avl \*t) {

   int l\_height = height(t->l);

   int r\_height = height(t->r);

   int b\_factor = l\_height - r\_height;

   return b\_factor;

}

avl \*avl\_tree::rr\_rotat(avl \*parent) {

   avl \*t;

   t = parent->r;

   parent->r = t->l;

   t->l = parent;

   cout<<"Right-Right Rotation";

   return t;

}

avl \*avl\_tree::ll\_rotat(avl \*parent) {

   avl \*t;

   t = parent->l;

   parent->l = t->r;

   t->r = parent;

   cout<<"Left-Left Rotation";

   return t;

}

avl \*avl\_tree::lr\_rotat(avl \*parent) {

   avl \*t;

   t = parent->l;

   parent->l = rr\_rotat(t);

   cout<<"Left-Right Rotation";

   return ll\_rotat(parent);

}

avl \*avl\_tree::rl\_rotat(avl \*parent) {

   avl \*t;

   t = parent->r;

   parent->r = ll\_rotat(t);

   cout<<"Right-Left Rotation";

   return rr\_rotat(parent);

}

avl \*avl\_tree::balance(avl \*t) {

   int bal\_factor = difference(t);

   if (bal\_factor > 1) {

      if (difference(t->l) > 0)

         t = ll\_rotat(t);

      else

         t = lr\_rotat(t);

   } else if (bal\_factor < -1) {

      if (difference(t->r) > 0)

         t = rl\_rotat(t);

      else

         t = rr\_rotat(t);

   }

   return t;

}

avl \*avl\_tree::insert(avl \*r, int v) {

   if (r == NULL) {

      r = new avl;

      r->d = v;

      r->l = NULL;

      r->r = NULL;

      return r;

   } else if (v< r->d) {

      r->l = insert(r->l, v);

      r = balance(r);

   } else if (v >= r->d) {

      r->r = insert(r->r, v);

      r = balance(r);

   } return r;

}

void avl\_tree::show(avl \*p, int l) {

   int i;

   if (p != NULL) {

      show(p->r, l+ 1);

      cout<<" ";

      if (p == r)

         cout << "Root -> ";

      for (i = 0; i < l&& p != r; i++)

         cout << " ";

         cout << p->d;

         show(p->l, l + 1);

   }

}

void avl\_tree::inorder(avl \*t) {

   if (t == NULL)

      return;

      inorder(t->l);

      cout << t->d << " ";

      inorder(t->r);

}

void avl\_tree::preorder(avl \*t) {

   if (t == NULL)

      return;

      cout << t->d << " ";

      preorder(t->l);

      preorder(t->r);

}

void avl\_tree::postorder(avl \*t) {

   if (t == NULL)

      return;

      postorder(t ->l);

      postorder(t ->r);

      cout << t->d << " ";

}

int main() {

   int c, i;

   avl\_tree avl;

   while (1) {

      cout << "1.Insert Element into the tree" << endl;

      cout << "2.show Balanced AVL Tree" << endl;

      cout << "3.InOrder traversal" << endl;

      cout << "4.PreOrder traversal" << endl;

      cout << "5.PostOrder traversal" << endl;

      cout << "6.Exit" << endl;

      cout << "Enter your Choice: ";

      cin >> c;

      switch (c) {

         case 1:

            cout << "Enter value to be inserted: ";

            cin >> i;

            r = avl.insert(r, i);

         break;

         case 2:

            if (r == NULL) {

               cout << "Tree is Empty" << endl;

               continue;

            }

            cout << "Balanced AVL Tree:" << endl;

            avl.show(r, 1);

            cout<<endl;

         break;

         case 3:

            cout << "Inorder Traversal:" << endl;

            avl.inorder(r);

            cout << endl;

         break;

         case 4:

            cout << "Preorder Traversal:" << endl;

            avl.preorder(r);

            cout << endl;

         break;

         case 5:

            cout << "Postorder Traversal:" << endl;

            avl.postorder(r);

            cout << endl;

         break;

         case 6:

            exit(1);

         break;

         default:

            cout << "Wrong Choice" << endl;

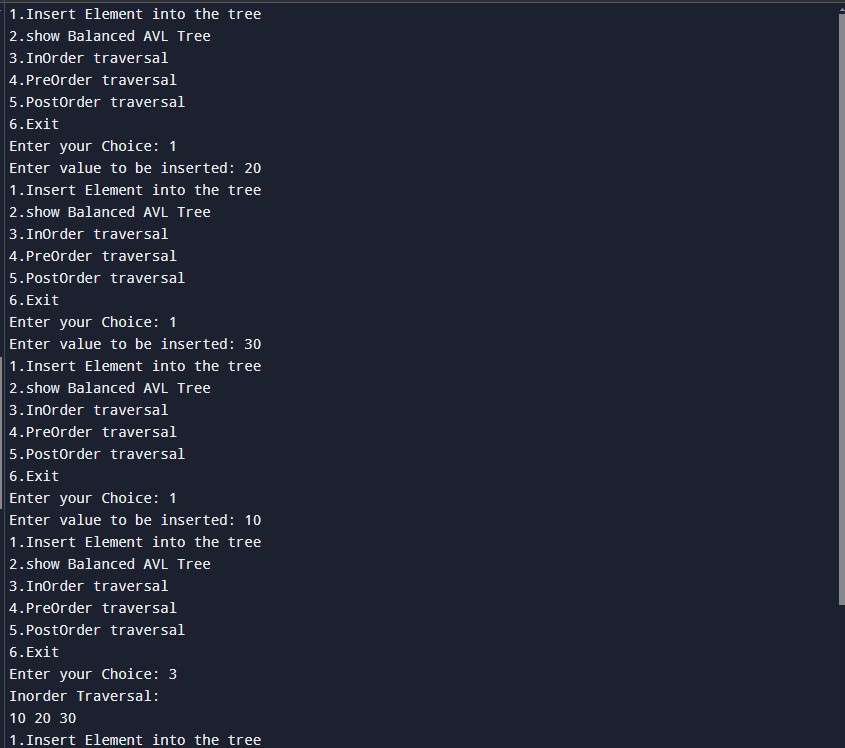
      }

   }

   return 0;

}

OUTPUT:



**EX.NO:10 Write a program that uses functions to perform the following:**

**i) Creating a SplayTree of integers**

**ii) Traversing the above Splaytree in preorder, inorder and postorder.**

**DATE:**

**Note: Pseudo code given below, convert this to Algorithm:**

Begin

   Create a structure s to declare variable k and left child pointer lch and right child pointer rch.

   Create a class SplayTree :

      Create a function RR\_Rotate to rotate to the right.

      Create a function LL\_Rotate to rotate to the left.

      Create a function Splay to implement top-down splay tree.

      Here head.rch points to the Left tree and head.lch

      points to the right tree.

      Create a link to Right tree.

      Create a link to Left tree.

      Assemble left, middle and right tree.

   Create a function New\_Node() to create nodes in the tree.

   Create a function Insert() to insert nodes into the tree.

      If root→k >= all keys will be the root→lch

      Else if

      root->k >=all keys will be the root→rch

      Else

      Return root.

   Create a function Delete() to delete nodes from the tree.

   Create a function Search() to search the nodes in the tree.

   Create a function InOrder() for InOrder traversal of the tree.

   Create a function main(), and perform selective function calls as per choice.

End

**Source Code:**

#include <iostream>

#include <cstdio>

#include <cstdlib>

using namespace std;

struct s//node declaration

{

   int k;

   s\* lch;

   s\* rch;

};

class SplayTree

{

   public:

   s\* RR\_Rotate(s\* k2)

   {

      s\* k1 = k2->lch;

      k2->lch = k1->rch;

      k1->rch = k2;

      return k1;

   }

   s\* LL\_Rotate(s\* k2)

   {

      s\* k1 = k2->rch;

      k2->rch = k1->lch;

      k1->lch = k2;

      return k1;

   }

   s\* Splay(int key, s\* root)

   {

      if (!root)

      return NULL;

      s header;

      header.lch= header.rch = NULL;

      s\* LeftTreeMax = &header;

      s\* RightTreeMin = &header;

      while (1)

      {

         if (key < root->k)

         {

            if (!root->lch)

            break;

            if (key< root->lch->k)

            {

               root = RR\_Rotate(root);

               if (!root->lch)

               break;

            }

            RightTreeMin->lch= root;

            RightTreeMin = RightTreeMin->lch;

            root = root->lch;

            RightTreeMin->lch = NULL;

         }

         else if (key> root->k)

         {

            if (!root->rch)

            break;

            if (key > root->rch->k)

            {

               root = LL\_Rotate(root);

               if (!root->rch)

               break;

            }

            LeftTreeMax->rch= root;

            LeftTreeMax = LeftTreeMax->rch;

            root = root->rch;

            LeftTreeMax->rch = NULL;

         }

         else

         break;

      }

      LeftTreeMax->rch = root->lch;

      RightTreeMin->lch = root->rch;

      root->lch = header.rch;

      root->rch = header.lch;

      return root;

   }

   s\* New\_Node(int key)

   {

      s\* p\_node = new s;

      if (!p\_node)

      {

         fprintf(stderr, "Out of memory!\n");

         exit(1);

      }

      p\_node->k = key;

      p\_node->lch = p\_node->rch = NULL;

      return p\_node;

   }

   s\* Insert(int key, s\* root)

   {

      static s\* p\_node = NULL;

      if (!p\_node)

      p\_node = New\_Node(key);

      else

      p\_node->k = key;

      if (!root)

      {

         root = p\_node;

         p\_node = NULL;

         return root;

      }

      root = Splay(key, root);

      if (key < root->k)

      {

         p\_node->lch= root->lch;

         p\_node->rch = root;

         root->lch = NULL;

         root = p\_node;

      }

      else if (key > root->k)

      {

         p\_node->rch = root->rch;

         p\_node->lch = root;

         root->rch = NULL;

         root = p\_node;

      }

      else

      return root;

      p\_node = NULL;

      return root;

   }

   s\* Delete(int key, s\* root)//delete node

   {

      s\* temp;

      if (!root)//if tree is empty

      return NULL;

      root = Splay(key, root);

      if (key != root->k)//if tree has one item

      return root;

      else

      {

         if (!root->lch)

         {

            temp = root;

            root = root->rch;

         }

         else

         {

            temp = root;

            root = Splay(key, root->lch);

            root->rch = temp->rch;

         }

         free(temp);

         return root;

      }

   }

   s\* Search(int key, s\* root)//seraching

   {

      return Splay(key, root);

   }

   void InOrder(s\* root)//inorder traversal

   {

      if (root)

      {

         InOrder(root->lch);

         cout<< "key: " <<root->k;

         if(root->lch)

         cout<< " | left child: "<< root->lch->k;

         if(root->rch)

         cout << " | right child: " << root->rch->k;

         cout<< "\n";

         InOrder(root->rch);

      }

   }

};

int main()

{

   SplayTree st;

   s \*root;

   root = NULL;

   st.InOrder(root);

   int i, c;

   while(1)

   {

      cout<<"1. Insert "<<endl;

      cout<<"2. Delete"<<endl;

      cout<<"3. Search"<<endl;

      cout<<"4. Exit"<<endl;

      cout<<"Enter your choice: ";

      cin>>c;

      switch( c )//perform switch operation

      {

         case 1:

            cout<<"Enter value to be inserted: ";

            cin>>i;

            root = st.Insert(i, root);

            cout<<"\nAfter Insert: "<<i<<endl;

            st.InOrder(root);

            break;

         case 2:

            cout<<"Enter value to be deleted: ";

            cin>>i;

            root = st.Delete(i, root);

            cout<<"\nAfter Delete: "<<i<<endl;

            st.InOrder(root);

            break;

         case 3:

            cout<<"Enter value to be searched: ";

            cin>>i;

            root = st.Search(i, root);

            cout<<"\nAfter Search "<<i<<endl;

            st.InOrder(root);

            break;

         case 4:

            exit(1);

         default:

            cout<<"\nInvalid type! \n";

      }

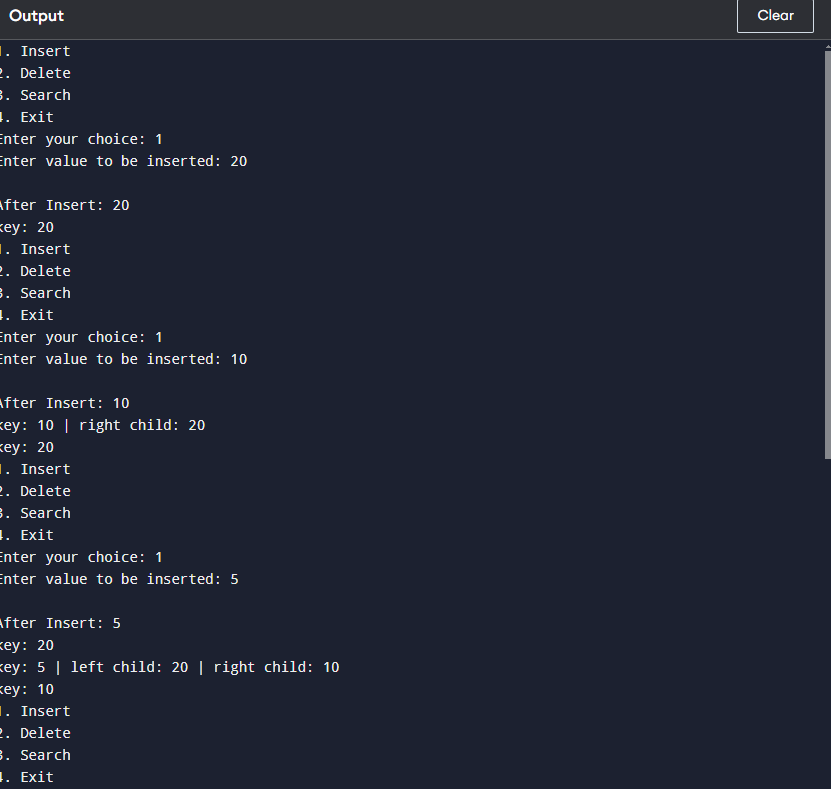
   }

   cout<<"\n";

   return 0;

}

OUTPUT:



**EX.NO:11 Write a program to perform the following:**

**i) Creating a B-Tree of integers**

**ii) Traversing the above B-Tree in preorder, inorder and postorder .**

**DATE:**

**Algorithm:**

1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.
   * Insert the new element in the increasing order of elements.
   * Split the node into the two nodes at the median.
   * Push the median element upto its parent node.
   * If the parent node also contain m-1 number of keys, then split it too by following the same steps.

**Source Code:**

#include<iostream>

using namespace std;

struct BTree

{

int \*d;

BTree \*\*child\_ptr;

bool l;

int n;

}\*r = NULL, \*np = NULL, \*x = NULL;

BTree\* init()

{

int i;

np = new BTree;

np->d = new int[6];//order 6

np->child\_ptr = new BTree \*[7];

np->l = true;

np->n = 0;

for (i = 0; i < 7; i++) {

np->child\_ptr[i] = NULL;

}

return np;

}

void traverse(BTree \*p)

{

cout<<endl;

int i;

for (i = 0; i < p->n; i++) {

if (p->l == false) {

traverse(p->child\_ptr[i]);

}

cout << " " << p->d[i];

}

if (p->l == false) {

traverse(p->child\_ptr[i]);

}

cout<<endl;

}

void sort(int \*p, int n)

{

int i, j, t;

for (i = 0; i < n; i++) {

for (j = i; j <= n; j++) {

if (p[i] >p[j]) {

t = p[i];

p[i] = p[j];

p[j] = t;

}

}

}

}

int split\_child(BTree \*x, int i) {

int j, mid;

BTree \*np1, \*np3, \*y;

np3 = init();//create new node

np3->l = true;

if (i == -1) {

mid = x->d[2];//find mid

x->d[2] = 0;

x->n--;

np1 = init();

np1->l= false;

x->l= true;

for (j = 3; j < 6; j++) {

np3->d[j - 3] = x->d[j];

np3->child\_ptr[j - 3] = x->child\_ptr[j];

np3->n++;

x->d[j] = 0;

x->n--;

}

for (j = 0; j < 6; j++) {

x->child\_ptr[j] = NULL;

}

np1->d[0] = mid;

np1->child\_ptr[np1->n] = x;

np1->child\_ptr[np1->n + 1] = np3;

np1->n++;

r = np1;

} else {

y = x->child\_ptr[i];

mid = y->d[2];

y->d[2] = 0;

y->n--;

for (j = 3; j <6 ; j++) {

np3->d[j - 3] = y->d[j];

np3->n++;

y->d[j] = 0;

y->n--;

}

x->child\_ptr[i + 1] = y;

x->child\_ptr[i + 1] = np3;

}

return mid;

}

void insert(int a) {

int i, t;

x = r;

if (x == NULL) {

r = init();

x = r;

} else {

if (x->l== true && x->n == 6) {

t = split\_child(x, -1);

x = r;

for (i = 0; i < (x->n); i++) {

if ((a >x->d[i]) && (a < x->d[i + 1])) {

i++;

break;

} else if (a < x->d[0]) {

break;

} else {

continue;

}

}

x = x->child\_ptr[i];

} else {

while (x->l == false) {

for (i = 0; i < (x->n); i++) {

if ((a >x->d[i]) && (a < x->d[i + 1])) {

i++;

break;

} else if (a < x->d[0]) {

break;

} else {

continue;

}

}

if ((x->child\_ptr[i])->n == 6) {

t = split\_child(x, i);

x->d[x->n] = t;

x->n++;

continue;

} else {

x = x->child\_ptr[i];

}

}

}

}

x->d[x->n] = a;

sort(x->d, x->n);

x->n++;

}

int main() {

int i, n, t;

cout<<"enter the no of elements to be inserted\n";

cin>>n;

for(i = 0; i < n; i++) {

cout<<"enter the element\n";

cin>>t;

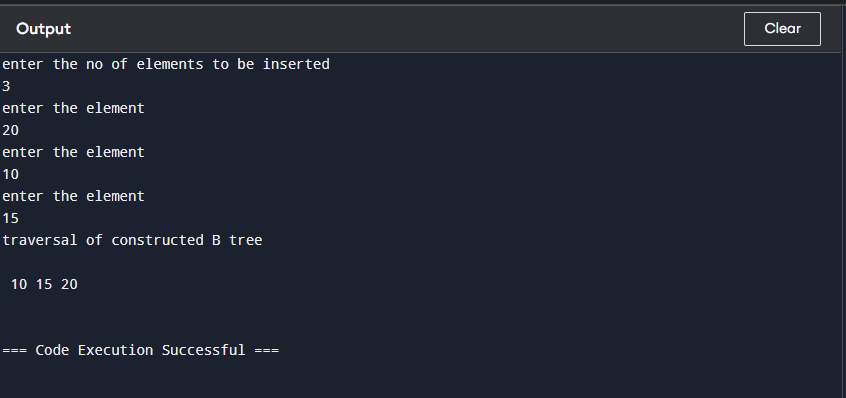
insert(t);

}

cout<<"traversal of constructed B tree\n";

traverse(r);

}



**EX.NO:12 Write a program to simulate various graph traversing algorithms.**

**DATE:**

**BFS Algorithm:**

A standard BFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

**Source Code:**

// BFS algorithm in C++

#include <iostream>

#include <list>

using namespace std;

class Graph {

int numVertices;

list<int>\* adjLists;

bool\* visited;

public:

Graph(int vertices);

void addEdge(int src, int dest);

void BFS(int startVertex);

};

// Create a graph with given vertices,

// and maintain an adjacency list

Graph::Graph(int vertices) {

numVertices = vertices;

adjLists = new list<int>[vertices];

}

// Add edges to the graph

void Graph::addEdge(int src, int dest) {

adjLists[src].push\_back(dest);

adjLists[dest].push\_back(src);

}

// BFS algorithm

void Graph::BFS(int startVertex) {

visited = new bool[numVertices];

for (int i = 0; i < numVertices; i++)

visited[i] = false;

list<int> queue;

visited[startVertex] = true;

queue.push\_back(startVertex);

list<int>::iterator i;

while (!queue.empty()) {

int currVertex = queue.front();

cout << "Visited " << currVertex << " ";

queue.pop\_front();

for (i = adjLists[currVertex].begin(); i != adjLists[currVertex].end(); ++i) {

int adjVertex = \*i;

if (!visited[adjVertex]) {

visited[adjVertex] = true;

queue.push\_back(adjVertex);

}

}

}

}

int main() {

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

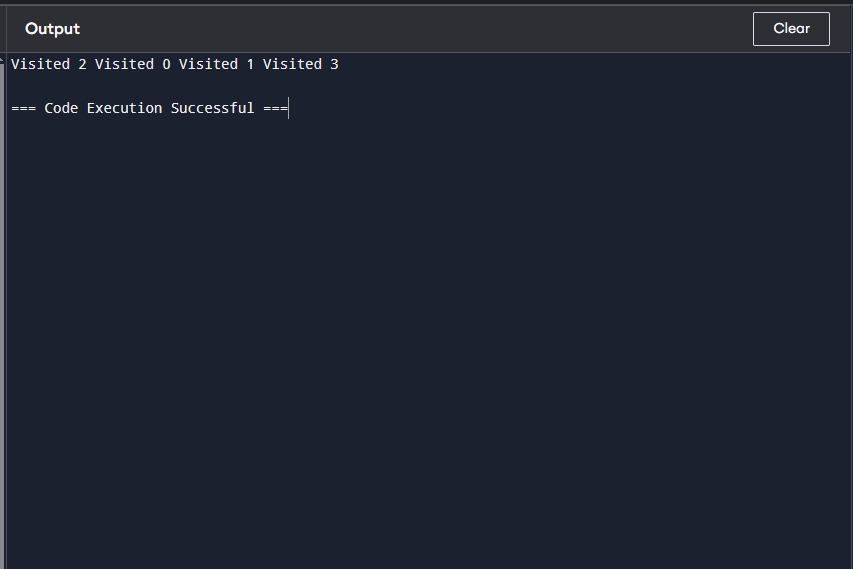
g.addEdge(2, 3);

g.addEdge(3, 3);

g.BFS(2);

return 0;

}



## **Depth First Search Algorithm**

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

**Source Code:**

// DFS algorithm in C++

#include <iostream>

#include <list>

using namespace std;

class Graph {

int numVertices;

list<int> \*adjLists;

bool \*visited;

public:

Graph(int V);

void addEdge(int src, int dest);

void DFS(int vertex);

};

// Initialize graph

Graph::Graph(int vertices) {

numVertices = vertices;

adjLists = new list<int>[vertices];

visited = new bool[vertices];

}

// Add edges

void Graph::addEdge(int src, int dest) {

adjLists[src].push\_front(dest);

}

// DFS algorithm

void Graph::DFS(int vertex) {

visited[vertex] = true;

list<int> adjList = adjLists[vertex];

cout << vertex << " ";

list<int>::iterator i;

for (i = adjList.begin(); i != adjList.end(); ++i)

if (!visited[\*i])

DFS(\*i);

}

int main() {

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 3);

g.DFS(2);

return 0;

}

