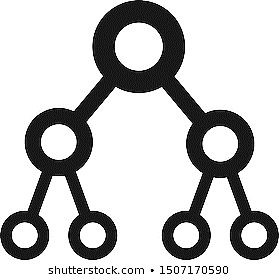
 **Atma Ram Sanatan Dharma College**

University of Delhi



**Data structure practical file**

**Submitted to :--** **Dr. Shalini Gupta**

**Submitted by – Azad**

**Roll no – 18030**

**Semester – III**

**Course – BSc (Hons) Computer Science**

# **INDEX**

|  |  |  |
| --- | --- | --- |
| S No. | Objective |  |
| **1** | **Given a list of N elements, which follows no particular arrangement, you are required to search an element x in the list. The list is stored using array data structure. If the search is successful, the output should be the index at which the element occurs, otherwise returns -1 to indicate that the element is not present in the list. Assume that the elements of the list are all distinct. Write a program to perform the desired task.** |  |
| **2** | **Given a list of N elements, which is sorted in ascending order, you are required to search an element x in the list. The list is stored using array data structure. If the search is successful, the output should be the index at which the element occurs, otherwise returns -1 to indicate that the element is not present in the list. Assume that the elements of the list are all distinct. Write a program to perform the desired task.** |  |
| **3** | **Write a program to implement singly linked list which supports the following operations:**  **(i) Insert an element x at the beginning of the doubly linked list**  **(ii) Insert an element x at position in the doubly linked list**  **(iii) Insert an element x at the end of the doubly linked list**  **(iv) Remove an element from the beginning of the doubly linked list**  **(v) Remove an element from position in the doubly linked list.**  **(vi) Remove an element from the end of the doubly linked list**  **(vii) Search for an element x in the doubly linked list and return its pointer . (viii) Concatenate two doubly linked lists** |  |
| **4** | **Write a program to implement doubly linked list which supports the following operations:**  **(i) Insert an element x at the beginning of the doubly linked list**  **(ii) Insert an element x at position in the doubly linked list**  **(iii) Insert an element x at the end of the doubly linked list**  **(iv) Remove an element from the beginning of the doubly linked list**  **(v) Remove an element from position in the doubly linked list.**  **(vi) Remove an element from the end of the doubly linked list**  **(vii) Search for an element x in the doubly linked list and return its pointer . (viii) Concatenate two doubly linked lists** |  |
| **5** | **Write a program to implement circularly linked list which supports the following operations:**   1. **Insert an element x at the front of the circularly linked list** 2. **Insert an element x after an element y in the circularly linked list** 3. **Insert an element x at the back of the circularly linked list** 4. **Remove an element from the back of the circularly linked list** 5. **Remove an element from the front of the circularly linked list** 6. **Remove the element x from the circularly linked list** 7. **Search for an element x in the circularly linked list and return its pointer** 8. **Concatenate two circularly linked lists** |  |
| **6** | **Implement a Stack using Array representation.** |  |
| **7** | **Implement a Stack using Linked List representation.** |  |
| **8** | **Implement a Queue using Circular Array representation**. |  |
| **9** | **Implement a Queue using Circular Linked List representation**. |  |
| **10** | **Implement Double-ended Queues using Linked List representation**. |  |
| **11** | **Write a program to implement Binary Search Tree which supports the following operations:**   1. **Insert an element x** 2. **Delete an element x** 3. **Search for an element x in the BST and change its value to y and then place the node with value y at its appropriate position in the BST** 4. **Display the elements of the BST in preorder, inorder, and postorder traversal** 5. **Display the elements of the BST in level-by-level traversal** 6. **Display the height of the BST** |  |

**Practical Question 1**

Objective

**Given a list of N elements, which follows no particular arrangement, you are required to search an element x in the list. The list is stored using array data structure. If the search is successful, the output should be the index at which the element occurs, otherwise returns -1 to indicate that the element is not present in the list. Assume that the elements of the list are all distinct. Write a program to perform the desired task.**

**Code :-**

#include <iostream>

using namespace std;

int linearSearch(int \*arr, int size, int el)

{

  for (int i = 0; i < size; i++)

    if (arr[i] == el)

      return i;

  return -1;

}

int main()

{

  int ch = 1, el, res, n, arr[100];

  cout << "Enter Number of Elements: ";

  cin >> n;

  cout << "Enter Array Elements: ";

  for (int i = 0; i < n; i++)

    cin >> arr[i];

  cout << "Enter Search Element: ";

  cin >> el;

  res = linearSearch(arr, n, el);

  if (res != -1)

    cout << "Element found at index "<< res << endl;

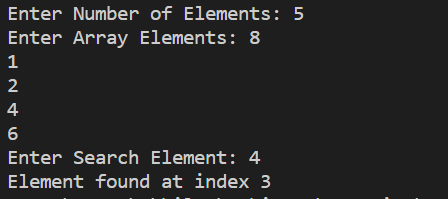
  else

    cout << "Element not found in array"<< endl;

  return 0;

}

Output :



**Practical Question 2**

Objective

**Given a list of N elements, which is sorted in ascending order, you are required to search an element x in the list. The list is stored using array data structure. If the search is successful, the output should be the index at which the element occurs, otherwise returns -1 to indicate that the element is not present in the list. Assume that the elements of the list are all distinct. Write a** program **to perform the desired task**

**Code :-**

#include <iostream>

using namespace std;

int binarySearch(int \*arr, int left, int right, int el)

{

  if (right >= left)

  {

    int mid = (right + left) / 2;

    if (arr[mid] == el)

      return mid;

    if (arr[mid] > el)

      return binarySearch(arr, left, mid - 1, el);

    return binarySearch(arr, mid + 1, right, el);

  }

  return -1;

}

int main(void)

{

  int ch = 1, el, res, n, arr[100];

  cout << "Enter Number of Elements: ";

  cin >> n;

  cout << "Enter Array Elements: ";

  for (int i = 0; i < n; i++)

    cin >> arr[i];

  cout << "Enter Search Element: ";

  cin >> el;

  res = binarySearch(arr, 0, n - 1, el);

  if (res != -1)

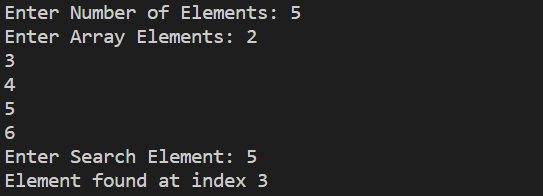
    cout <<"Element found at index "<< res << endl;

  else

    cout << "Element not found in array"<< endl;

  return 0;

}

Output :

**Practical Question 3**

Objective

**Write a program to implement singly linked list which supports the following operations:**

1. **Insert an element x at the beginning of the singly linked list**
2. **Insert an element x at ith position in the singly linked list**
3. **Remove an element from the beginning of the singly linked list**
4. **Remove an element from ith position in the singly linked list**
5. **Search for an element x in the singly linked list and return its pointer**

(f) **Concatenate two singly linked lists**

**Code :-**

#include <iostream>

using namespace std;

struct node

{

    int data;

    struct node \*next;

} \* head;

class singlylink

{

public:

    singlylink()

    {

        head = NULL;

    }

    void creatnode()

    {

        struct node \*temp, \*current = NULL;

        char ch = 'y';

        do

        {

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

            if (temp == NULL)

            {

                cout << "node not available";

                return;

            }

            cout << "Enter node : ";

            cin >> temp->data;

            temp->next = NULL;

            if (head == NULL)

            {

                head = temp;

                current = temp;

            }

            else

            {

                current->next = temp;

                current = temp;

            }

            cout << "Do you want more node Enter y otherwise press any key ";

            cin >> ch;

        } while (ch == 'y');

        Display();

    }

    void inserthead()

    {

        struct node \*temp, \*ptr;

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

        cout << "Enter node to be insert in head ";

        cin >> temp->data;

        if (head == NULL)

        {

            head = temp;

            head->next = NULL;

        }

        else

        {

            ptr = head;

            head = temp;

            head->next = ptr;

        }

        cout<<"Node inserted succesfully \n";

    }

    void insertposition()

    {

        struct node \*temp, \*newnode;

        newnode = (struct node \*)malloc(sizeof(struct node));

        int pos;

        int value;

        cout << "\nEnter node to insert position : \n";

        cin >> value;

        cout << " Enter position : \n";

        cin >> pos;

        if (newnode == NULL)

        {

            cout << "node not found ";

        }

        else

        {

            newnode->data = value;

            newnode->next = NULL;

            temp = head;

            for (int i = 2; i <= pos - 1; i++)

            {

                temp = temp->next;

                if (temp == NULL)

                    break;

            }

            if (temp != NULL)

            {

                newnode->next = temp->next;

                temp->next = newnode;

            }

            else

            {

                cout << "insert not possible ";

            }

        }

    }

    void deletehead()

    {

        struct node \*temp;

        if (head == NULL)

        {

            cout << "List is empty ";

        }

        else

            temp = head;

        head = temp->next;

        free(temp);

        cout << "\n Node deleted from head ";

    }

    void deletepos()

    {

        int pos, i, counter = 0;

        if (head == NULL)

        {

            cout << "List is empty" << endl;

            return;

        }

        cout << "Enter the position of value to be deleted: ";

        cin >> pos;

        struct node \*s, \*ptr;

        s = head;

        if (pos == 1)

        {

            head = s->next;

        }

        else

        {

            while (s != NULL)

            {

                s = s->next;

                counter++;

            }

            if (pos > 0 && pos <= counter)

            {

                s = head;

                for (i = 1; i < pos; i++)

                {

                    ptr = s;

                    s = s->next;

                }

                ptr->next = s->next;

            }

            else

            {

                cout << "Position out of range" << endl;

            }

            free(s);

            cout << "Element Deleted" << endl;

        }

    }

    void search()

    {

        struct node \*temp;

        int value, pos = 0;

        bool flag = false;

        if (head == NULL)

        {

            cout << "List is empty" << endl;

            return;

        }

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

        cin >> value;

        struct node \*s;

        s = head;

        while (s != NULL)

        {

            pos++;

            if (s->data == value)

            {

                flag = true;

                cout << "Element " << value << " is found at position " << pos << endl;

            }

            s = s->next;

        }

        if (!flag)

            cout << "Element " << value << " not found in the list" << endl;

    }

    void Display()

    {

        struct node \*temp;

        if (head == NULL)

        {

            cout << "The List is Empty" << endl;

            return;

        }

        temp = head;

        cout << "Elements of list are: " << endl;

        while (temp != NULL)

        {

            cout << temp->data << "->";

            temp = temp->next;

        }

        cout << "NULL" << endl;

    }

};

int main()

{

    singlylink sl;

    int choice;

    do

    {

        cout << "\n\tSingly Linked List\n"

             << "===================================\n"

             << " (1) creat link list (2) InsertBegin\n"

             << " (3) InsertAtLoc (4) DeleteFront\n"

             << " (5) DeletePosition (6) Search Element\n"

             << " (7) Display\n (0) Exit\n\n";

        cout << "Enter your choice : ";

        cin >> choice;

        switch (choice)

        {

        case 1:

            sl.creatnode();

            break;

        case 2:

            sl.inserthead();

            break;

        case 3:

            sl.insertposition();

            break;

        case 4:

            sl.deletehead();

            break;

        case 5:

            sl.deletepos();

            break;

        case 6:

            sl.search();

            break;

        case 7:

            sl.Display();

            break;

        case 0:

        default:

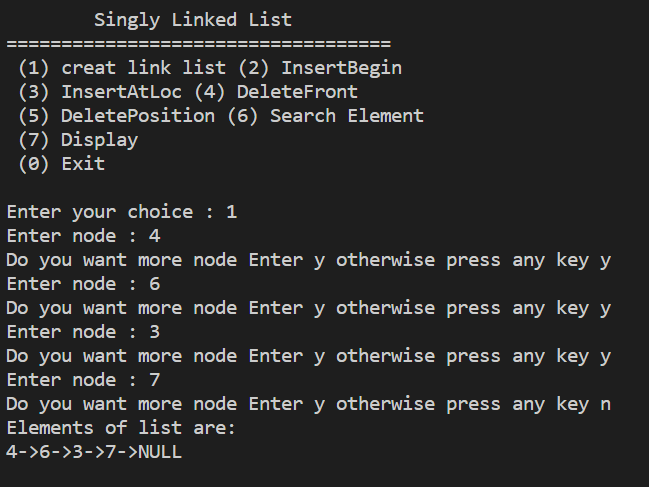
            break;

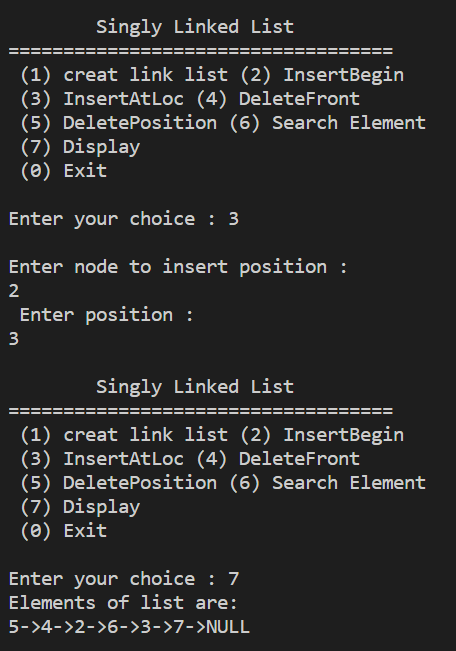
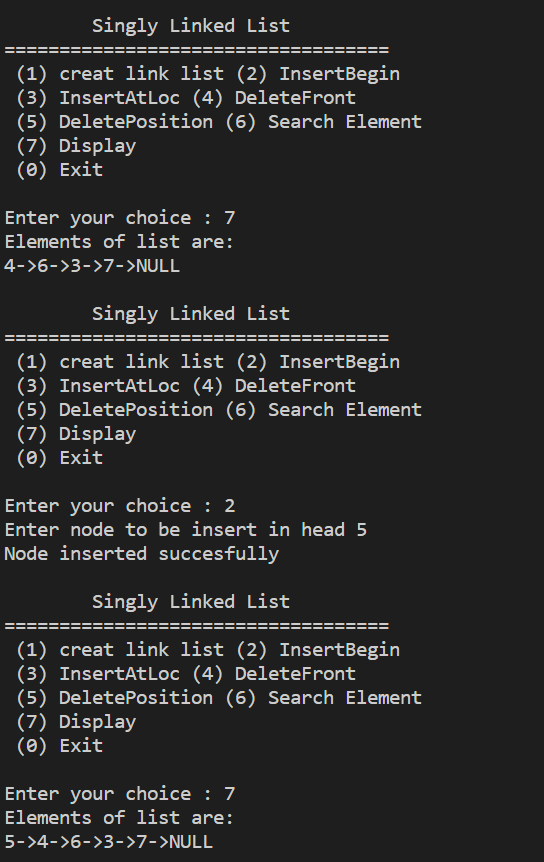
        }

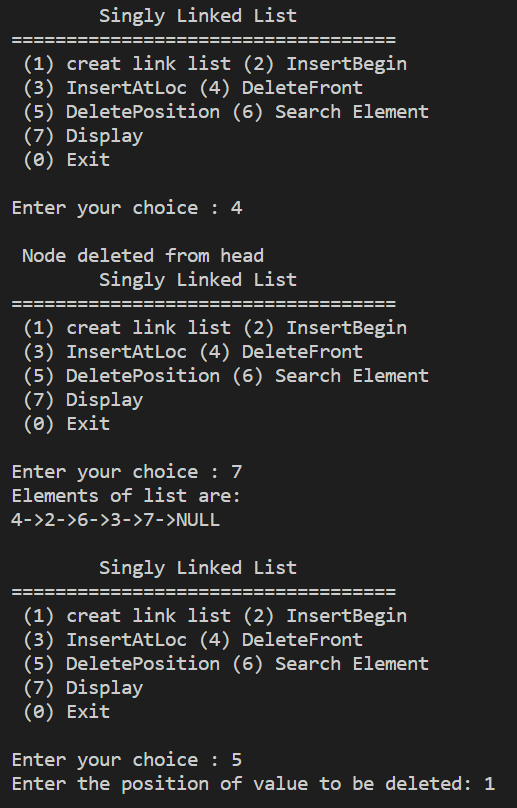
    } while (choice != 0);

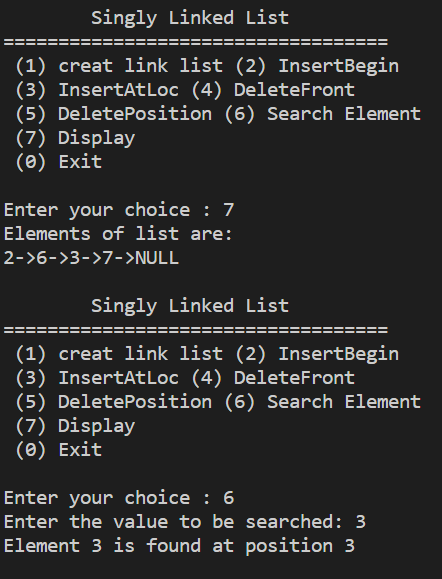
    return 0;

}

Output :







**Practical Question 4**

Objective

**Write a program to implement doubly linked list which supports the following operations:**

1. **Insert an element x at the beginning of the doubly linked list**
2. **Insert an element x at ith position in the doubly linked list**
3. **Insert an element x at the end of the doubly linked list**
4. **Remove an element from the beginning of the doubly linked list**
5. **Remove an element from ith position in the doubly linked list**
6. **Remove an element from the end of the doubly linked list**
7. **Search for an element x in the doubly linked list and return its pointer**
8. **Concatenate two doubly linked lists**

**Code :**

#include <iostream>

using namespace std;

struct node

{

    int data;

    struct node \*next;

    struct node \*prev;

} \* head;

class doublylink

{

public:

    doublylink()

    {

        head = NULL;

    }

    void creatnode()

    {

        struct node \*temp, \*current = NULL;

        char ch = 'y';

        do

        {

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

            if (temp == NULL)

            {

                cout << "node not available";

                return;

            }

            cout << "Enter node : ";

            cin >> temp->data;

            temp->next = NULL;

            temp->prev = NULL;

            if (head == NULL)

            {

                head = temp;

                current = temp;

            }

            else

            {

                current->next = temp;

                temp->prev = current;

                current = temp;

            }

            cout << "do you want more node Enter Yes=Y or No=N ";

            cin >> ch;

        } while (ch == 'y');

        Display();

    }

    void inserthead()

    {

        struct node \*temp, \*ptr;

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

        cout << "Enter node to be insert in head ";

        cin >> temp->data;

        if (head == NULL)

        {

            head = temp;

            head->next = NULL;

            head->prev = NULL;

        }

        else

        {

            head->prev = temp;

            temp->next = head;

            head = temp;

        }

        cout << "Node inserted succesfully \n";

    }

    void insertend()

    {

        struct node \*newnode, \*temp;

        newnode = (struct node \*)malloc(sizeof(struct node));

        cout << "\nEnter node to be insert at end \n";

        cin >> newnode->data;

        newnode->next = NULL;

        newnode->prev = NULL;

        if (head == NULL)

        {

            head = newnode;

        }

        else

        {

            temp = head;

            while (temp->next != NULL)

                temp = temp->next;

            temp->next = newnode;

            newnode->prev = temp;

            cout << "\n Node inserted succesfully";

        }

    }

    void insertposition()

    {

        struct node \*temp, \*newnode;

        newnode = (struct node \*)malloc(sizeof(struct node));

        int pos;

        int value;

        cout << "\nEnter node to insert position : \n";

        cin >> value;

        cout << " Enter position : \n";

        cin >> pos;

        if (newnode == NULL)

        {

            cout << "node not found ";

        }

        else

        {

            temp = head;

            for (int i = 2; i <= pos - 1; i++)

            {

                temp = temp->next;

                if (temp == NULL)

                    break;

            }

            newnode->data = value;

            newnode->next = temp->next;

            newnode->prev = temp;

            temp->next = newnode;

            temp->next->prev = newnode;

            cout << "Node inserted \n";

        }

    }

    void delhead()

    {

        struct node \*temp;

        if (head == NULL)

        {

            cout << "List is empty ";

        }

        else

            temp = head;

        head = temp->next;

        temp->prev = NULL;

        free(temp);

        cout << "\n Node deleted from head ";

    }

    void delend()

    {

        struct node \*temp, \*temp1;

        if (head == NULL)

        {

            cout << "\nlist is empty\n";

        }

        else if (head->next == NULL)

        {

            head = NULL;

            free(head);

        }

        else

        {

            temp = head;

            while (temp->next != NULL)

            {

                temp = temp->next;

            }

            temp->prev->next = NULL;

            free(temp);

            cout << "\nNode deleted  from end succesfully\n";

        }

    }

    void delpos()

    {

        struct node \*temp, \*current;

        int pos, i, counter = 0;

        if (head == NULL)

        {

            cout << "List is empty" << endl;

            return;

        }

        cout << "\nEnter the position of value to be deleted: ";

        cin >> pos;

        if (pos == 1)

        {

            temp = head;

            head = head->next;

            if (head->next)

                head->prev = NULL;

            free(temp);

            return;

        }

        else

        {

            temp = head;

            int count = 1;

            while (count < pos && temp != NULL)

            {

                count++;

                temp = temp->next;

            }

            if (temp == NULL)

            {

                cout << "Position does not exist ";

                return;

            }

            current = temp->prev;

            current->next = temp->next;

            if (temp->next)

                temp->next->prev = current;

            free(temp);

            cout << "\nNode deleted ";

        }

    }

    void search()

    {

        int value;

        struct node \*temp = head;

        int position = 0;

        int flag = 0;

        cout << "Enter the value to find the position : ";

        cin >> value;

        while (temp != NULL)

        {

            position += 1;

            if (temp->data == value)

            {

                flag = 1;

                break;

            }

            temp = temp->next;

        }

        if (flag == 0)

        {

            cout << "\nNode with data was not found " << value;

        }

        else

        {

            cout << "\nNode found at position : " << position;

        }

    }

    void Display()

    {

        struct node \*temp;

        if (head == NULL)

        {

            cout << "The List is Empty" << endl;

            return;

        }

        temp = head;

        cout << "Elements of list are: " << endl;

        while (temp != NULL)

        {

            cout << temp->data << "<->";

            temp = temp->next;

        }

        cout << "NULL" << endl;

    }

};

int main()

{

    doublylink dl;

    int choice;

    do

    {

        cout << "\n\tDoubly Linked List\n"

             << "===================================\n"

             << " (1) creat link list (2) InsertBegin\n"

             << " (3) InsertBack      (4) InsertAtLoc\n"

             << " (5) DeleteFront     (6) DeleteBack\n"

             << " (7) DeletePosition  (8) Search Element\n"

             << " (9)Display          (0) Exit\n\n";

        cout << "Enter your choice : ";

        cin >> choice;

        switch (choice)

        {

        case 1:

            dl.creatnode();

            break;

        case 2:

            dl.inserthead();

            break;

        case 3:

            dl.insertend();

            break;

        case 4:

            dl.insertposition();

            break;

        case 5:

            dl.delhead();

            break;

        case 6:

            dl.delend();

            break;

        case 7:

            dl.delpos();

            break;

        case 8:

            dl.search();

            break;

        case 9:

            dl.Display();

            break;

        case 0:

        default:

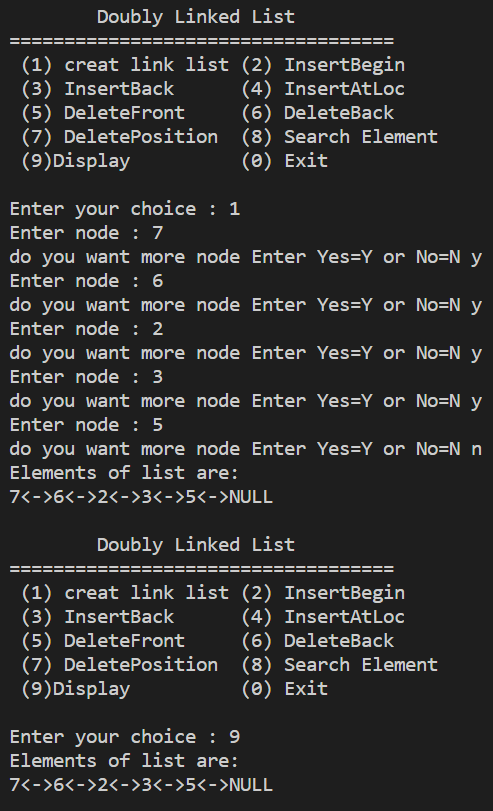
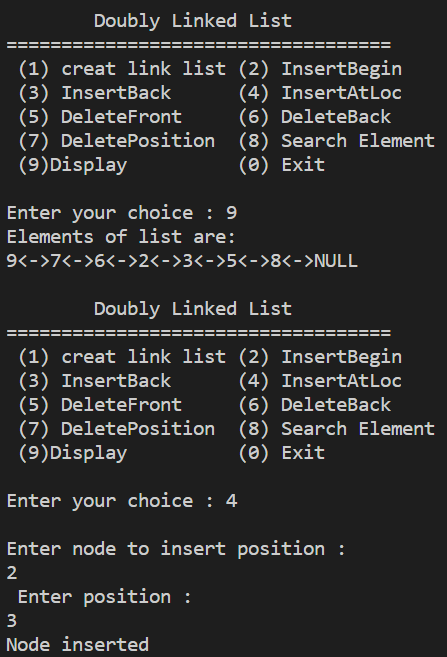
            break;

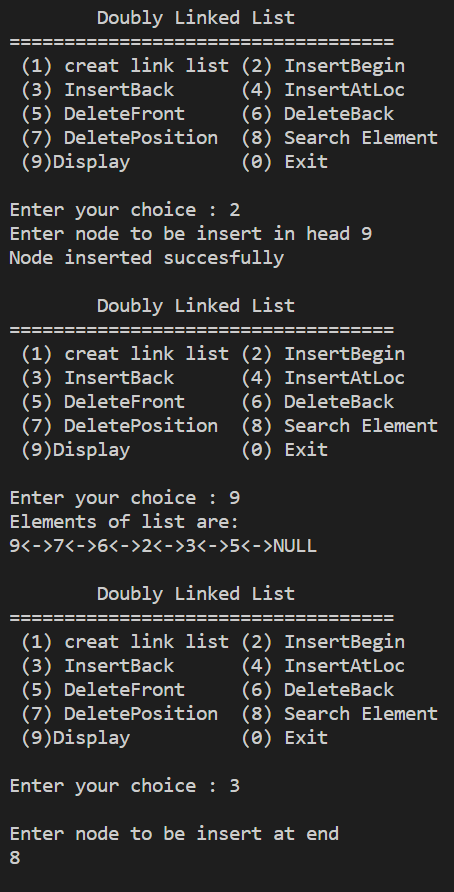
        }

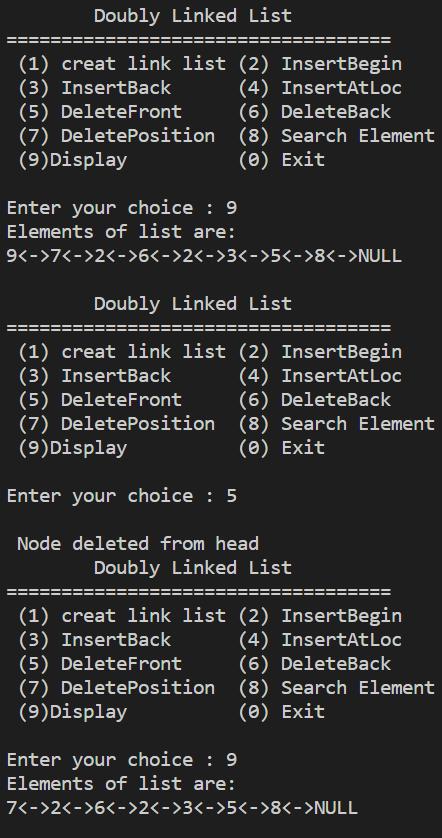
    } while (choice != 0);

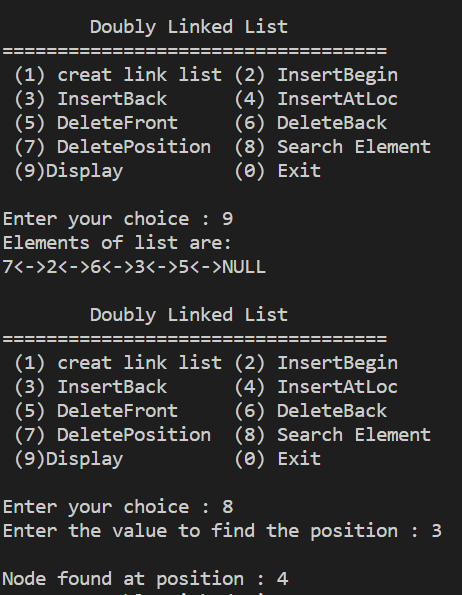
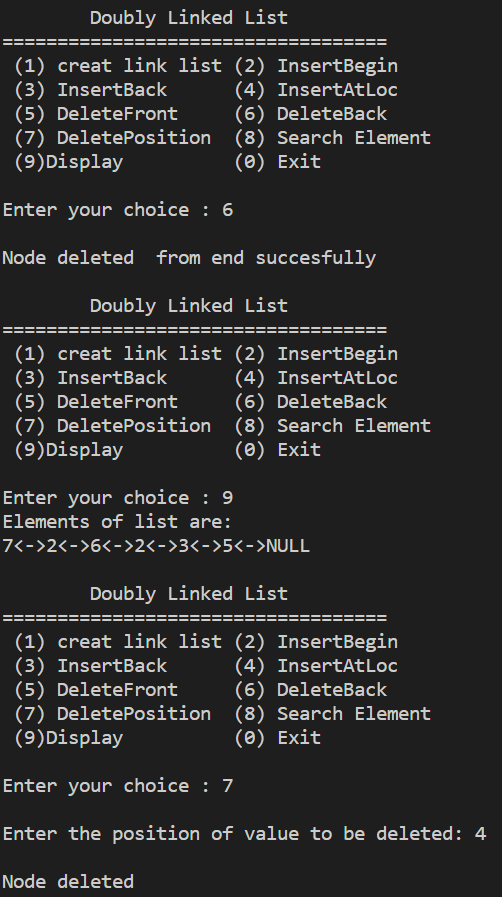
    return 0;

}

Output :





****

**Practical Question 5**

Objective

**Write a program to implement circularly linked list which supports the following operations:**

1. **Insert an element x at the front of the circularly linked list**
2. **Insert an element x after an element y in the circularly linked list**
3. **Insert an element x at the back of the circularly linked list**
4. **Remove an element from the back of the circularly linked list**
5. **Remove an element from the front of the circularly linked list**
6. **Remove the element x from the circularly linked list**
7. **Search for an element x in the circularly linked list and return its pointer**
8. **Concatenate two circularly linked lists**

**Code :-**

#include<iostream>

#include<cstdio>

#include<cstdlib>

using namespace std;

struct node

{

    int info;

    struct node \*next;

}\*last;

class circular\_llist

{

    public:

        void create\_node();

        void add\_begin(int value);

        void add\_after(int value, int position);

        void delete\_element(int value);

        void search\_element(int value);

        void display\_list();

        circular\_llist()

        {

            last = NULL;

        }

};

int main()

{

    int choice, element, position;

    circular\_llist cl;

    while (1)

    {

        cout<<"\nCircular singly linked list\n"

        <<"\n---------------------------\n"

        <<" (1) Create Node (2) Add at beginning\n"

        <<" (3) Add after   (4) Delete position\n"

        <<" (5) Search      (6) Display\n"

        <<" (7)Quit\n";

        cout<<"Enter your choice : ";

        cin>>choice;

        switch(choice)

        {

        case 1:

            cl.create\_node();

            cout<<endl;

            break;

        case 2:

            cout<<"Enter the element: ";

            cin>>element;

            cl.add\_begin(element);

            cout<<endl;

            break;

        case 3:

            cout<<"Enter the element: ";

            cin>>element;

            cout<<"Insert element after position: ";

            cin>>position;

            cl.add\_after(element, position);

            cout<<endl;

            break;

        case 4:

            if (last == NULL)

            {

                cout<<"List is empty, nothing to delete"<<endl;

                break;

            }

            cout<<"Enter the element for deletion: ";

            cin>>element;

            cl.delete\_element(element);

            cout<<endl;

            break;

        case 5:

            if (last == NULL)

            {

                cout<<"List Empty!! Can't search"<<endl;

                break;

            }

            cout<<"Enter the element to be searched: ";

            cin>>element;

            cl.search\_element(element);

            cout<<endl;

            break;

        case 6:

            cl.display\_list();

            break;

        case 7:

            exit(1);

            break;

        default:

            cout<<"Wrong choice"<<endl;

        }

    }

    return 0;

}

void circular\_llist::create\_node()

{

struct node \*temp, \*current = NULL;

    char ch = 'y';

    do

    {

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

        if (temp == NULL)

        {

            cout << "node not available";

            return;

        }

        cout << "Enter node : ";

        cin >> temp->info;

        temp->next = NULL;

        if (last == NULL)

        {

            last = temp;

            temp->next = last;

            current = temp;

        }

        else

        {

            current->next = temp;

            temp->next = last;

            current = temp;

        }

        cout << "do you want more node Enter Yes=Y or No=N ";

        cin >> ch;

    } while (ch == 'y');

}

void circular\_llist::add\_begin(int value)

{

    if (last == NULL)

    {

        cout<<"First Create the list."<<endl;

        return;

    }

    struct node \*temp;

    temp = new(struct node);

    temp->info = value;

    temp->next = last->next;

    last->next = temp;

}

void circular\_llist::add\_after(int value, int pos)

{

    if (last == NULL)

    {

        cout<<"First Create the list."<<endl;

        return;

    }

    struct node \*temp, \*s;

    s = last->next;

    for (int i = 0;i < pos-1;i++)

    {

        s = s->next;

        if (s == last->next)

        {

            cout<<"There are less than ";

            cout<<pos<<" in the list"<<endl;

            return;

        }

    }

    temp = new(struct node);

    temp->next = s->next;

    temp->info = value;

    s->next = temp;

    if (s == last)

    {

        last=temp;

    }

}

void circular\_llist::delete\_element(int value)

{

    struct node \*temp, \*s;

    s = last->next;

    if (last->next == last && last->info == value)

    {

        temp = last;

        last = NULL;

        free(temp);

        return;

    }

    if (s->info == value)

    {

        temp = s;

        last->next = s->next;

        free(temp);

        return;

    }

    while (s->next != last)

    {

        if (s->next->info == value)

        {

            temp = s->next;

            s->next = temp->next;

            free(temp);

            cout<<"Element "<<value;

            cout<<" deleted from the list"<<endl;

            return;

        }

        s = s->next;

    }

    if (s->next->info == value)

    {

        temp = s->next;

        s->next = last->next;

        free(temp);

        last = s;

        return;

    }

    cout<<"Element "<<value<<" not found in the list"<<endl;

}

void circular\_llist::search\_element(int value)

{

    struct node \*s;

    int counter = 0;

    s = last->next;

    while (s != last)

    {

        counter++;

        if (s->info == value)

        {

            cout<<"Element "<<value;

            cout<<" found at position "<<counter<<endl;

            return;

        }

        s = s->next;

    }

    if (s->info == value)

    {

        counter++;

        cout<<"Element "<<value;

        cout<<" found at position "<<counter<<endl;

        return;

    }

    cout<<"Element "<<value<<" not found in the list"<<endl;

}

void circular\_llist::display\_list()

{

    struct node \*s;

    if (last == NULL)

    {

        cout<<"List is empty, nothing to display"<<endl;

        return;

    }

    s = last->next;

    cout<<"Circular Link List: "<<endl;

    while (s != last)

    {

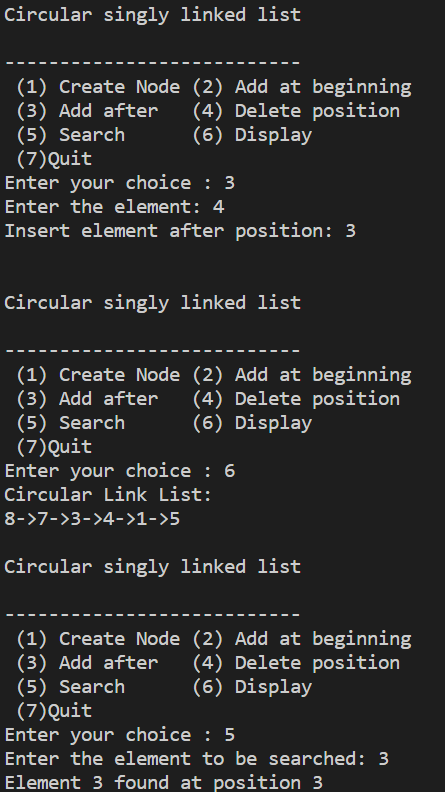
        cout<<s->info<<"->";

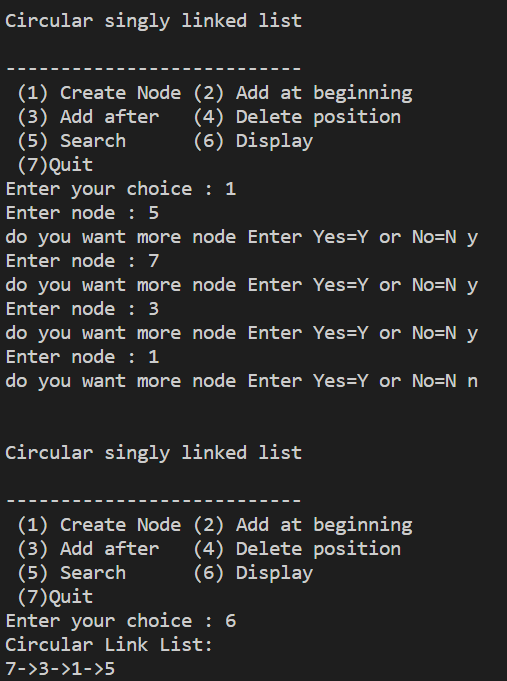
        s = s->next;

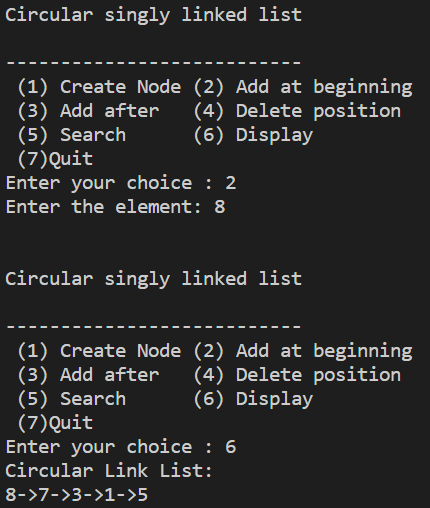
    }

    cout<<s->info<<endl;

}

Output :





**Practical Question 6**

Objective

**Implement a Stack using Array representation**

**Code :-**

#include<iostream>

using namespace std;

class Stack{

    int arr[100];

    int n = 100;

    int top = -1;

    public:

     void push(int);

     int pop();

     void display();

};

void Stack::push(int item)

{

    if(top == n-1)

    {

        cout<<"Stack is full";

    }

    else

    {

        arr[++top] = item;

    }

}

int Stack::pop()

{

    int item;

    if(top < -1)

    {

        cout<<"Stack is underflow ";

    }

    else{

        item = arr[top--];

        return item;

    }

}

void Stack::display()

{

    if(top == -1)

    {

        cout<<"nothing can be displayed ";

    }

    else

    {

        cout<<"\n Element are :\n";

        for(int i = top;i>=0;i--)

        {

            cout<<arr[i]<<" ";

        }

    }

}

int main() {

    Stack s;

   int ch, val;

   cout<<"1) Push in stack"<<endl;

   cout<<"2) Pop from stack"<<endl;

   cout<<"3) Display stack"<<endl;

   cout<<"4) Exit"<<endl;

   do {

      cout<<"\nEnter choice: "<<endl;

      cin>>ch;

      switch(ch) {

         case 1: {

            cout<<"Enter value to be pushed:"<<endl;

            cin>>val;

            s.push(val);

            break;

         }

         case 2: {

            s.pop();

            break;

         }

         case 3: {

            s.display();

            break;

         }

         case 4: {

            cout<<"Exit"<<endl;

            break;

         }

         default: {

            cout<<"Invalid Choice"<<endl;

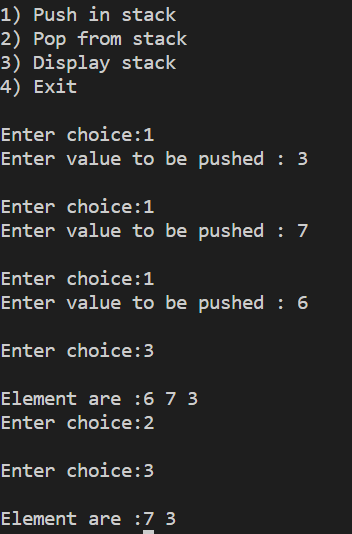
         }

      }

   }while(ch!=4);

   return 0;

}

Output :

**Practical Question 7**

Objective

**Implement a Stack using Linked List representation.**

**Code :**

#include <stdio.h>

#include <stdlib.h>

struct node

{

    int info;

    struct node \*ptr;

}\*top,\*top1,\*temp;

int topelement();

void push(int data);

void pop();

void empty();

void display();

void destroy();

void stack\_count();

void create();

int count = 0;

void main()

{

    int no, ch, e;

    printf("\n 1 - Push");

    printf("\n 2 - Pop");

    printf("\n 3 - Top");

    printf("\n 4 - Empty");

    printf("\n 5 - Exit");

    printf("\n 6 - Dipslay");

    printf("\n 7 - Stack Count");

    printf("\n 8 - Destroy stack");

    create();

    while (1)

    {

        printf("\n Enter choice : ");

        scanf("%d", &ch);

        switch (ch)

        {

        case 1:

            printf("Enter data : ");

            scanf("%d", &no);

            push(no);

            break;

        case 2:

            pop();

            break;

        case 3:

            if (top == NULL)

                printf("No elements in stack");

            else

            {

                e = topelement();

                printf("\n Top element : %d", e);

            }

            break;

        case 4:

            empty();

            break;

        case 5:

            exit(0);

        case 6:

            display();

            break;

        case 7:

            stack\_count();

            break;

        case 8:

            destroy();

            break;

        default :

            printf(" Wrong choice, Please enter correct choice  ");

            break;

        }

    }

}

void create()

{

    top = NULL;

}

void stack\_count()

{

    printf("\n No. of elements in stack : %d", count);

}

void push(int data)

{

    if (top == NULL)

    {

        top =(struct node \*)malloc(1\*sizeof(struct node));

        top->ptr = NULL;

        top->info = data;

    }

    else

    {

        temp =(struct node \*)malloc(1\*sizeof(struct node));

        temp->ptr = top;

        temp->info = data;

        top = temp;

    }

    count++;

}

void display()

{

    top1 = top;

    if (top1 == NULL)

    {

        printf("Stack is empty");

        return;

    }

    while (top1 != NULL)

    {

        printf("%d ", top1->info);

        top1 = top1->ptr;

    }

 }

void pop()

{

    top1 = top;

    if (top1 == NULL)

    {

        printf("\n Error : Trying to pop from empty stack");

        return;

    }

    else

        top1 = top1->ptr;

    printf("\n Popped value : %d", top->info);

    free(top);

    top = top1;

    count--;

}

 int topelement()

{

    return(top->info);

}

void empty()

{

    if (top == NULL)

        printf("\n Stack is empty");

    else

        printf("\n Stack is not empty with %d elements", count);

}

void destroy()

{

    top1 = top;

    while (top1 != NULL)

    {

        top1 = top->ptr;

        free(top);

        top = top1;

        top1 = top1->ptr;

    }

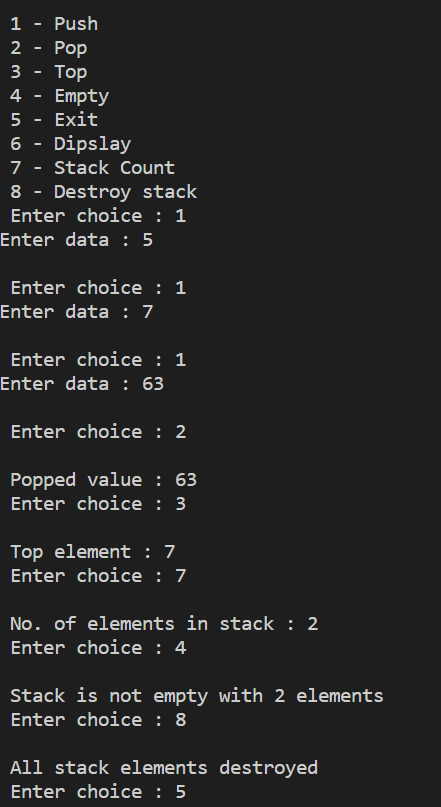
    free(top1);

    top = NULL;

    printf("\n All stack elements destroyed");

    count = 0;

}

Output :

**Practical Question 8**

Objective

**Implement a Queue using Circular Array representation**.

**Code :-**

#include <iostream>

#define MAX\_SIZE 100

using namespace std;

void getch();

void clrscr();

class Queue

{

protected:

  int arr[MAX\_SIZE];

  int front, rear, size;

public:

  Queue(int size = 5)

  {

    this->front = -1;

    this->rear = -1;

    this->size = size;

  }

  bool enqueue(int ele)

  {

    if (this->isFull())

    {

      cerr << "ERROR: Queue Filled\n";

      return false;

    }

    else

    {

      if (this->rear == this->size - 1 ||

          this->rear == -1)

      {

        this->arr[0] = ele;

        this->rear = 0;

        if (this->isEmpty())

          this->front = 0;

      }

      else

        this->arr[++(this->rear)] = ele;

      return true;

    }

  }

  int dequeue()

  {

    if (this->isEmpty())

    {

      cout << "ERROR: Queue Empty\n";

      return (int)(NULL);

    }

    else

    {

      int temp = this->arr[this->front];

      if (this->front == this->rear)

        this->clear();

      else if (this->front == this->size - 1)

        this->front = 0;

      else

        this->front++;

      return temp;

    }

  }

  int frontEl()

  {

    if (this->isEmpty())

    {

      cout << "Queue Empty";

      return (int)(NULL);

    }

    return this->arr[this->front];

  }

  bool isFull()

  {

    return this->front == 0 &&

               this->rear == this->size - 1 ||

           this->front == this->rear + 1;

  }

  bool isEmpty()

  {

    return this->front == -1;

  }

  void clear()

  {

    this->front = this->rear = -1;

  }

  void display()

  {

    if (this->isEmpty())

    {

      cout << "Queue Empty";

      return;

    }

    int i;

    if (this->rear >= this->front)

    {

      for (i = this->front; i < this->rear; i++)

        cout << this->arr[i] << " <- ";

      cout << this->arr[i] << endl;

    }

    else

    {

      for (i = this->front; i < this->size; i++)

        cout << this->arr[i] << " <- ";

for (i = 0; i < this->rear; i++)

        cout << this->arr[i] << " <- ";

      cout << this->arr[i] << endl;

    }

    return;

  }

};

int main()

{

  int n, el, res, choice;

  cout << "Enter Size of Queue: ";

  cin >> n;

  Queue q(n);

  do

  {

    cout << "\tCircular Queue - Array\n"

         << "===================================\n"

         << "  (1) Enqueue  (2) Dequeue\n"

         << "  (3) Front    (4) Clear\n"

         << "  (5) Display  (0) Exit\n\n";

    cout << "Enter Choice: ";

    cin >> choice;

    switch (choice)

    {

    case 1:

      cout << "\nEnter Element: ";

      cin >> el;

      res = q.enqueue(el);

      if (res)

      {

        cout << "\nEnqueued " << el << "...\n";

        cout << "Queue: ";

        q.display();

      }

      break;

    case 2:

      res = q.dequeue();

      if (res)

      {

        cout << "\nDequeued " << res << "...\n";

        cout << "Queue: ";

        q.display();

      }

      break;

    case 3:

      cout << "\nFront Element: "

           << q.frontEl() << endl;

      break;

    case 4:

      q.clear();

      break;

    case 5:

      cout << "\nQueue: ";

      q.display();

    default:

      break;

    }

    getch();

    clrscr();

  } while (choice != 0);

  return 0;

}

void getch()

{

  cout << "\nPress any key to continue...";

  cin.ignore();

  cin.get();

  return;

}

void clrscr()

{

#ifdef \_WIN32

  system("cls");

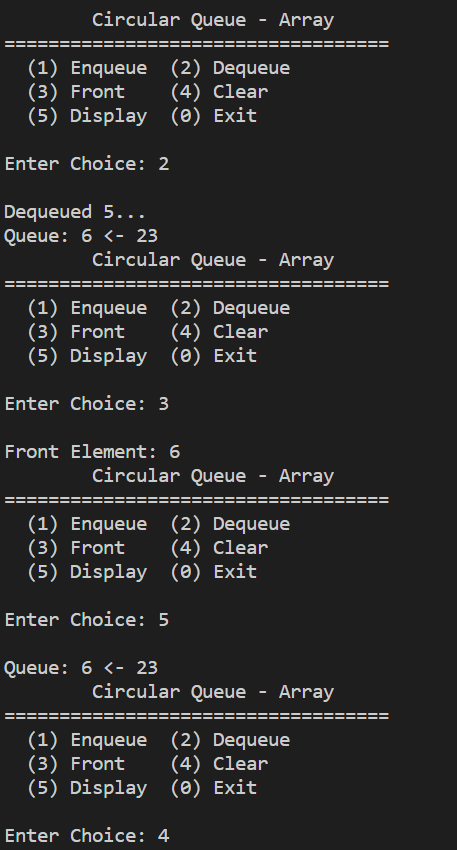
#elif \_\_unix\_\_

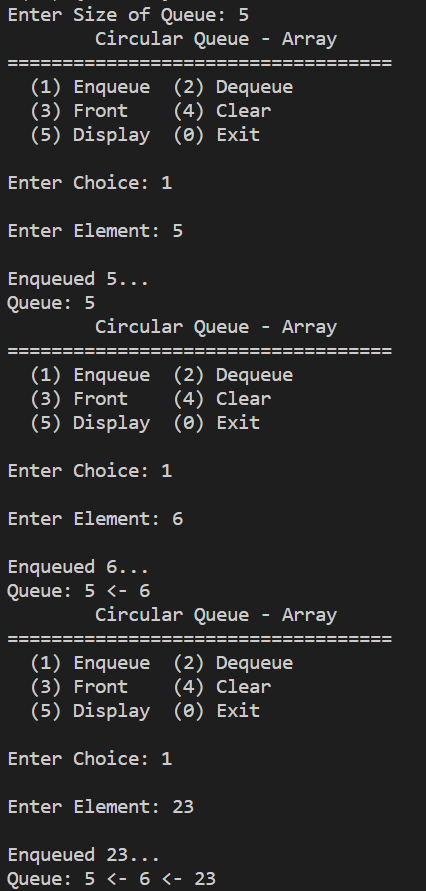
  system("clear");

#endif

  return;

}

Output :



**Practical Question 9**

Objective

**Implement a Queue using Circular Linked List representation**.

**Code :**

#include<stdio.h>

#include<stdlib.h>

struct node

{

        int info;

        struct node \*link;

}\*rear=NULL;

void insert(int item);

int del();

void display();

int isEmpty();

int peek();

int main()

{

        int choice,item;

        while(1)

        {

                printf("\n1.Insert\n");

                printf("2.Delete\n");

                printf("3.Peek\n");

                printf("4.Display\n");

                printf("5.Quit\n");

                printf("\nEnter your choice : ");

                scanf("%d",&choice);

                switch(choice)

                {

                 case 1:

                        printf("\nEnter the element for insertion : ");

                        scanf("%d",&item);

                        insert(item);

                        break;

                 case 2:

                        printf("\nDeleted element is %d\n",del());

                        break;

                 case 3:

                        printf("\nItem at the front of queue is %d\n",peek());

                        break;

                 case 4:

                        display();

                        break;

                 case 5:

                        exit(1);

                 default:

                        printf("\nWrong choice\n");

                }

        }

}

void insert(int item)

{

        struct node \*tmp;

        tmp=(struct node \*)malloc(sizeof(struct node));

        tmp->info=item;

        if(tmp==NULL)

        {

                printf("\nMemory not available\n");

                return;

        }

        if( isEmpty() )

        {

                rear=tmp;

                tmp->link=rear;

        }

        else

        {

                tmp->link=rear->link;

                rear->link=tmp;

                rear=tmp;

        }

}

del()

{

        int item;

        struct node \*tmp;

        if( isEmpty() )

        {

                printf("\nQueue underflow\n");

                exit(1);

        }

        if(rear->link==rear)

        {

                tmp=rear;

                rear=NULL;

        }

        else

        {

                tmp=rear->link;

                rear->link=rear->link->link;

        }

        item=tmp->info;

        free(tmp);

        return item;

}

int peek()

{

        if( isEmpty() )

        {

                printf("\nQueue underflow\n");

                exit(1);

        }

        return rear->link->info;

}

int isEmpty()

{

        if( rear == NULL )

                return 1;

        else

                return 0;

}

void display()

{

        struct node \*p;

        if(isEmpty())

        {

                printf("\nQueue is empty\n");

                return;

        }

        printf("\nQueue is :\n");

        p=rear->link;

        do

        {

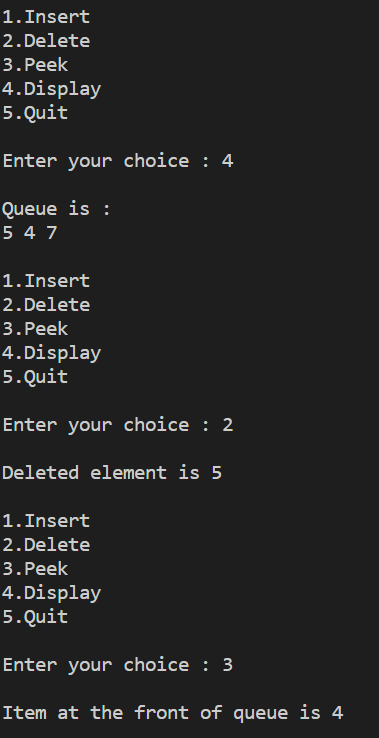
                printf("%d ",p->info);

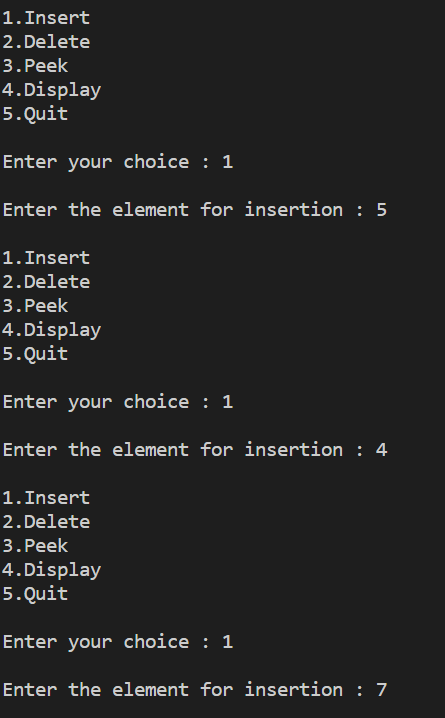
                p=p->link;

        }while(p!=rear->link);

        printf("\n");

}

Output :



**Practical Question 10**

Objective

**Implement Double-ended Queues using Linked List representation**.

**Code :-**

#include <stdio.h>

  #include <stdlib.h>

  struct node {

        int data;

        struct node \*prev, \*next;

  };

  struct node \*head = NULL, \*tail = NULL;

  struct node \* createNode(int data) {

        struct node \*newnode = (struct node \*)malloc(sizeof (struct node));

        newnode->data = data;

        newnode->next = newnode->prev = NULL;

        return (newnode);

  }

   /\*

    \* create sentinel(dummy head & tail) that

    \* helps us to do insertion and deletion

    \* operation at front and rear so easily.  And

    \* these dummy head and tail wont get deleted

    \* till the end of execution of this program

    \*/

  void createSentinels() {

        head = createNode(0);

        tail = createNode(0);

        head->next = tail;

        tail->prev = head;

  }

  /\* insertion at the front of the queue \*/

  void enqueueAtFront(int data) {

        struct node \*newnode, \*temp;

        newnode = createNode(data);

        temp = head->next;

        head->next = newnode;

        newnode->prev = head;

        newnode->next = temp;

        temp->prev = newnode;

  }

  /\*insertion at the rear of the queue \*/

  void enqueueAtRear(int data) {

        struct node \*newnode, \*temp;

        newnode = createNode(data);

        temp = tail->prev;

        tail->prev = newnode;

        newnode->next = tail;

        newnode->prev = temp;

        temp->next = newnode;

  }

  /\* deletion at the front of the queue \*/

  void dequeueAtFront() {

        struct node \*temp;

        if (head->next == tail) {

                printf("Queue is empty\n");

        } else {

                temp = head->next;

                head->next = temp->next;

                temp->next->prev = head;

                free(temp);

        }

        return;

  }

  /\* deletion at the rear of the queue \*/

  void dequeueAtRear()  {

        struct node \*temp;

        if (tail->prev == head) {

                printf("Queue is empty\n");

        } else {

                temp = tail->prev;

                tail->prev = temp->prev;

                temp->prev->next = tail;

                free(temp);

        }

        return;

  }

  /\* display elements present in the queue \*/

  void display() {

        struct node \*temp;

        if (head->next == tail) {

                printf("Queue is empty\n");

                return;

        }

        temp = head->next;

        while (temp != tail) {

                printf("%-3d", temp->data);

                temp = temp->next;

        }

        printf("\n");

  }

  int main() {

        int data, ch;

        createSentinels();

        while (1) {

                printf("1. Enqueue at front\n2. Enqueue at rear\n");

                printf("3. Dequeue at front\n4. Dequeue at rear\n");

                printf("5. Display\n6. Exit\n");

                printf("Enter your choice:");

                scanf("%d", &ch);

                switch (ch) {

                        case 1:

                                printf("Enter the data to insert:");

                                scanf("%d", &data);

                                enqueueAtFront(data);

                                break;

                        case 2:

                                printf("Enter ur data to insert:");

                                scanf("%d", &data);

                                enqueueAtRear(data);

                                break;

                        case 3:

                                dequeueAtFront();

                                break;

                        case 4:

                                dequeueAtRear();

                                break;

                        case 5:

                                display();

                                break;

                        case 6:

                                exit(0);

                        default:

                                printf("Pls. enter correct option\n");

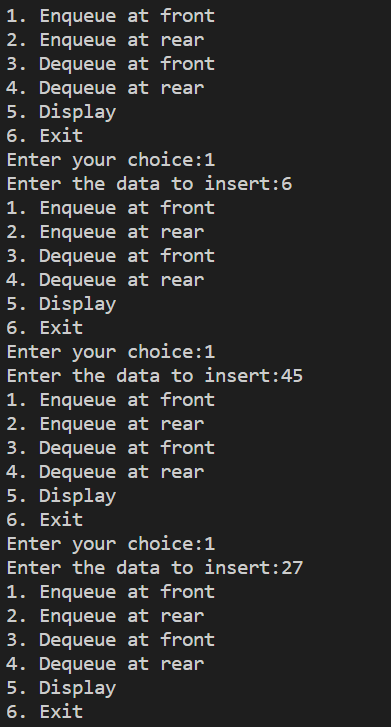
                                break;

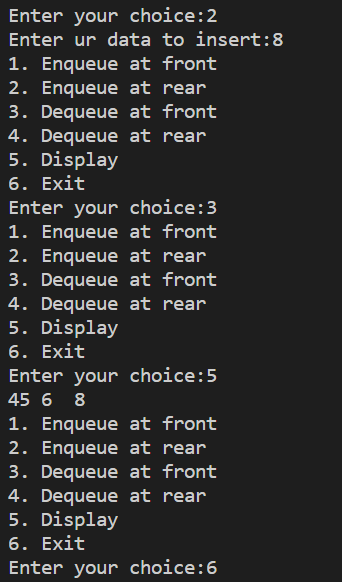
                }

        }

        return 0;

  }

**Output :**



Practical Question 11

Objective

Code :

#include <iostream>

#define MAX\_SIZE 100

using namespace std;

template <class T>

class Stack

{

protected:

  int tos, size;

  T arr[MAX\_SIZE];

public:

  Stack(int size = 30)

  {

    this->tos = -1;

    this->size = size;

  }

  bool push(T ele)

  {

    if (this->tos >= (this->size - 1))

    {

      cerr << "ERROR: Stack Overflow\n";

      return false;

    }

    this->arr[++(this->tos)] = ele;

    return true;

  }

  T pop()

  {

    if (this->isEmpty())

    {

      cout << "ERROR: Stack Underflow\n";

      return (T)(NULL);

    }

    return this->arr[(this->tos)--];

  }

  T top()

  {

    if (this->isEmpty())

    {

      cout << "Stack Empty";

      return (T)(NULL);

    }

    return this->arr[this->tos];

  }

  bool isEmpty()

  {

    return this->tos == -1;

  }

  void clear()

  {

    while (!this->isEmpty())

      this->pop();

  }

};

// queue.hpp

#include <iostream>

#define MAX\_SIZE 100

using namespace std;

template <class T>

class Queue

{

protected:

  T arr[MAX\_SIZE];

  int front, rear, size;

public:

  Queue(int size = 100)

  {

    this->front = -1;

    this->rear = -1;

    this->size = size;

  }

  bool enqueue(T ele)

  {

    if (this->rear >= (this->size - 1))

    {

      cerr << "ERROR: Queue Filled\n";

      return false;

    }

    else if (this->isEmpty())

    {

      this->rear++;

      this->front++;

      this->arr[this->front] = ele;

    }

    else

      this->arr[++(this->rear)] = ele;

    return true;

  }

  T dequeue()

  {

    if (this->front >= this->size)

    {

      cout << "ERROR: Queue Finished\n";

      return (T)(NULL);

    }

    else if (this->isEmpty())

    {

      cout << "ERROR: Queue Empty\n";

      return (T)(NULL);

    }

    else if (this->front == this->rear)

    {

      T temp = this->arr[this->front];

      this->clear();

      return temp;

    }

    return this->arr[(this->front)++];

  }

  T frontEl()

  {

    if (this->isEmpty())

    {

      cout << "Queue Empty";

      return (T)(NULL);

    }

    return this->arr[this->front];

  }

  bool isEmpty()

  {

    return this->front == -1;

  }

  void clear()

  {

    this->front = this->rear = -1;

  }

  void display()

  {

    if (this->isEmpty())

    {

      cout << "Queue Empty";

      return;

    }

    int i;

    for (i = this->front; i < this->rear; i++)

      cout << this->arr[i] << " <- ";

    cout << this->arr[i] << endl;

    return;

  }

};

// main.cpp

#include "stack.hpp"

#include "queue.hpp"

void getch();

void clrscr();

template <class T>

class Node

{

public:

  T data;

  Node \*left, \*right;

  Node()

  {

    left = nullptr;

    right = nullptr;

  }

};

class BinarySearchTree

{

public:

  Node<int> \*root;

  Stack<Node<int> \*> stack;

  Queue<Node<int> \*> queue;

  int countLeaf, countNonLeaf;

  BinarySearchTree()

  {

    root = nullptr;

  }

  void insert(int data, Node<int> \*current)

  {

    Node<int> \*temp;

    if (root == nullptr)

    {

      root = new Node<int>;

      root->data = data;

      root->left = root->right = nullptr;

    }

    else

    {

      if ((data < current->data) &&

          (current->left == nullptr))

      {

        temp = new Node<int>;

        temp->data = data;

        temp->left = temp->right = nullptr;

        current->left = temp;

      }

      else if ((data >= current->data) &&

               (current->right == nullptr))

      {

        temp = new Node<int>;

        temp->data = data;

        temp->left = temp->right = nullptr;

        current->right = temp;

      }

      else

      {

        if (data < current->data)

          insert(data, current->left);

        else

          insert(data, current->right);

      }

    }

  }

  bool search(Node<int> \*node, int key)

  {

    if (node == nullptr)

      return false;

    if (node->data == key)

      return true;

    bool left = search(node->left, key);

    if (left)

      return true;

    bool right = search(node->right, key);

    return right;

  }

  void inOrderRecursive(Node<int> \*root)

  {

    if (root != nullptr)

    {

      inOrderRecursive(root->left);

      cout << root->data << " ";

      inOrderRecursive(root->right);

    }

  }

  void preOrderRecursive(Node<int> \*root)

  {

    if (root != nullptr)

    {

      cout << root->data << " ";

      preOrderRecursive(root->left);

      preOrderRecursive(root->right);

    }

  }

  void postOrderRecursive(Node<int> \*root)

  {

    if (root != nullptr)

    {

      postOrderRecursive(root->left);

      postOrderRecursive(root->right);

      cout << root->data << " ";

    }

  }

  void inOrderIterative()

  {

    Node<int> \*current = root;

    while (current != nullptr ||

           stack.isEmpty() == false)

    {

      while (current != nullptr)

      {

        stack.push(current);

        current = current->left;

      }

      current = stack.pop();

      cout << current->data << " ";

      current = current->right;

    }

  }

  void preOrderIterative()

  {

    Node<int> \*node, \*temp = root;

    if (temp == nullptr)

      return;

    stack.push(temp);

    while (!stack.isEmpty())

    {

      node = stack.pop();

      cout << node->data << " ";

      if (node->right)

        stack.push(node->right);

      if (node->left)

        stack.push(node->left);

    }

  }

  void postOrderIterative()

  {

    Node<int> \*temp = root;

    if (temp == nullptr)

      return;

    do

    {

      while (temp)

      {

        if (temp->right)

          stack.push(temp->right);

        stack.push(temp);

        temp = temp->left;

      }

      temp = stack.pop();

      if (temp->right && !stack.isEmpty() &&

          stack.top() == temp->right)

      {

        stack.pop();

        stack.push(temp);

        temp = temp->right;

      }

      else

      {

        cout << temp->data << " ";

        temp = nullptr;

      }

    } while (!stack.isEmpty());

  }

  void levelByLevelTraversal()

  {

    Node<int> \*current = root;

    if (current == nullptr)

      return;

    queue.enqueue(current);

    while (!queue.isEmpty())

    {

      current = queue.dequeue();

      cout << current->data << " ";

      if (current->left)

        queue.enqueue(current->left);

      if (current->right)

        queue.enqueue(current->right);

    }

    cout << endl;

  }

  void mirror(Node<int> \*current)

  {

    if (current == nullptr)

      return;

    else

    {

      mirror(current->left);

      mirror(current->right);

      Node<int> \*temp = current->left;

      current->left = current->right;

      current->right = temp;

    }

  }

  int height(Node<int> \*current)

  {

    if (current == nullptr)

      return 0;

    else

    {

      int leftHeight = height(current->left);

      int rightHeight = height(current->right);

      if (leftHeight > rightHeight)

        return (leftHeight + 1);

      else

        return (rightHeight + 1);

    }

  }

  void countNodes(Node<int> \*current)

  {

    if (current == nullptr)

      return;

    if (current->left != nullptr ||

        current->right != nullptr)

      countNonLeaf++;

    if (current->left == nullptr &&

        current->right == nullptr)

      countLeaf++;

    countNodes(current->left);

    countNodes(current->right);

  }

  void deleteByMerging(Node<int> \*temp, int key)

  {

    Node<int> \*prev = nullptr;

    while (temp != nullptr)

    {

      if (temp->data == key)

        break;

      prev = temp;

      if (temp->data < key)

        temp = temp->right;

      else

        temp = temp->left;

    }

    if (temp != nullptr && temp->data == key)

    {

      if (temp == root)

        mergeHelper(root);

      else if (prev->left == temp)

        mergeHelper(prev->left);

      else

        mergeHelper(prev->right);

    }

    else if (root != nullptr)

      cout << "\nNode Not Found...";

    return;

  }

  void mergeHelper(Node<int> \*&node)

  {

    Node<int> \*temp = node;

    if (node == nullptr)

      return;

    // no right child - single child

    if (node->right == nullptr)

      node = node->left;

    // no left child - single chold

    else if (node->left == nullptr)

      node = node->right;

    // node has both children

    else

    {

      // find in-order predecessor

      temp = node->left;

      while (temp->right != nullptr)

        temp = temp->right;

      // merge subtree to predecessor

      temp->right = node->right;

      temp = node;

      node = node->left;

    }

    // delete the node

    delete temp;

    return;

  }

  void deleteByCopying(Node<int> \*temp, int key)

  {

    Node<int> \*prev = nullptr;

    while (temp != nullptr && temp->data != key)

    {

      prev = temp;

      if (temp->data < key)

        temp = temp->right;

      else

        temp = temp->left;

    }

    if (temp != nullptr && temp->data == key)

    {

      if (temp == root)

        copyHelper(root);

      else if (prev->left == temp)

        copyHelper(prev->left);

      else

        copyHelper(prev->right);

    }

    else if (root != nullptr)

      cout << "\nNode Not Found...";

    return;

  }

  void copyHelper(Node<int> \*&node)

  {

    Node<int> \*prev, \*temp = node;

    // no right child - single child

    if (node->right == nullptr)

      node = node->left;

    // no left child - single chold

    else if (node->left == nullptr)

      node = node->right;

    // node has both children

    else

    {

      prev = node;

      // find the in-order predecessor

      temp = node->left;

      while (temp->right != nullptr)

      {

        prev = temp;

        temp = temp->right;

      }

      // copy the prdecessor key

      node->data = temp->data;

      // handle dangling subtrees

      if (prev == node)

        prev->left = temp->left;

      else

        prev->right = temp->left;

    }

    // delete the node

    delete temp;

    return;

  }

  void searchAndReplace(int key, int newKey)

  {

    if (search(root, key))

    {

      deleteByMerging(root, key);

      insert(newKey, root);

    }

    else

    {

      cout << "Node Not Found...";

    }

  }

};

int main(void)

{

  BinarySearchTree tree;

  int choice, data, data2;

  do

  {

    cout << "      MENU      \n"

         << "================\n"

         << "(1) Insertion\n"

         << "(2) Searching a node\n"

         << "(3) Display its preorder, postorder and inorder traversals. (recursive)\n"

         << "(4) Display its preorder, postorder and inorder traversals. (iterative)\n"

         << "(5) Display level-by-level traversal. (BFS)\n"

         << "(6) Create a mirror image of tree\n"

         << "(7) Count the non-leaf, leaf and total number of nodes  \n"

         << "(8) Search for an element x in the BST and change its value to y \n"

         << "    and then place the node with value y at its appropriate position\n"

         << "(9) Display height of tree\n"

         << "(10) Perform deletion by merging\n"

         << "(11) Perform deletion by copying\n"

         << "(0) Exit\n\n";

    cout << "Enter Choice: ";

    cin >> choice;

    switch (choice)

    {

    case 1:

      cout << "\nEnter Node Data: ";

      cin >> data;

      tree.insert(data, tree.root);

      break;

    case 2:

      cout << "\nEnter Search Data: ";

      cin >> data;

      cout << "Search Result: ";

      if (tree.search(tree.root, data))

        cout << "Found";

      else

        cout << "Not Found";

      cout << endl;

      break;

    case 3:

      cout << endl;

      cout << "In-Order Recursive Traversal: ";

      tree.inOrderRecursive(tree.root);

      cout << endl;

      cout << "Pre-Order Recursive Traversal: ";

      tree.preOrderRecursive(tree.root);

      cout << endl;

      cout << "Post-Order Recursive Traversal: ";

      tree.postOrderRecursive(tree.root);

      cout << endl;

      break;

    case 4:

      cout << endl;

      cout << "In-Order Iterative Traversal: ";

      tree.inOrderIterative();

      cout << endl;

      cout << "Pre-Order Iterative Traversal: ";

      tree.preOrderIterative();

      cout << endl;

      cout << "Post-Order Iterative Traversal: ";

      tree.postOrderIterative();

      cout << endl;

      break;

    case 5:

      cout << endl;

      cout << "Level-by-level Traversal: \n";

      tree.levelByLevelTraversal();

      break;

    case 6:

      cout << endl;

      tree.mirror(tree.root);

      cout << "Tree converted to its Mirror Tree..."

           << endl;

      break;

    case 7:

      tree.countLeaf = tree.countNonLeaf = 0;

      tree.countNodes(tree.root);

      cout << endl;

      cout << "Leaf Nodes: "

           << tree.countLeaf << endl;

      cout << "Non-Leaf Nodes: "

           << tree.countNonLeaf << endl;

      cout << "Total Nodes: "

           << tree.countNonLeaf +

                  tree.countLeaf

           << endl;

      break;

    case 8:

      cout << "\nEnter Search Data: ";

      cin >> data;

      cout << "Enter Replacement: ";

      cin >> data2;

      tree.searchAndReplace(data, data2);

      break;

    case 9:

      cout << endl;

      cout << "Height of Tree: "

           << tree.height(tree.root)

           << endl;

      break;

    case 10:

      cout << "\nEnter Node to Delete: ";

      cin >> data;

      tree.deleteByMerging(tree.root, data);

      break;

    case 11:

      cout << "\nEnter Node to Delete: ";

      cin >> data;

      tree.deleteByCopying(tree.root, data);

      break;

    case 0:

    default:

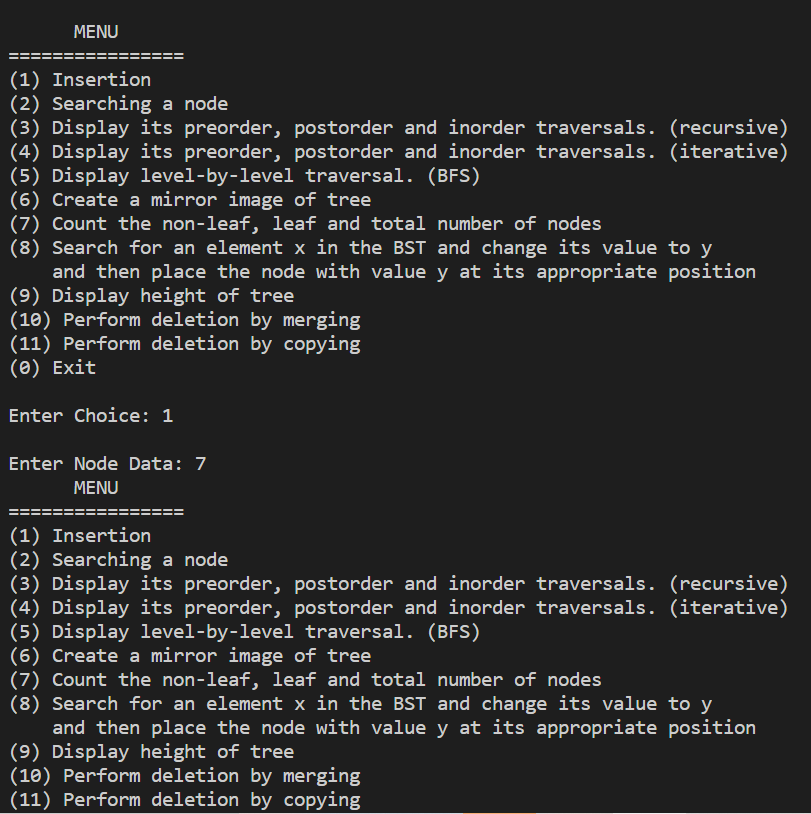
      break;

    }

  } while (choice != 0);

  return 0;

}

Output :

