### PRACTICAL 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.

#### Code

#include <iostream>

#define MAX\_SIZE 100

using *namespace* std;

*template* <*class* T>

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

{

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

        if (*arr*[i] == *el*)

            return i;

    return -1;

}

*int* main(*void*)

{

*int* ch = 1, el, res, N, arr[MAX\_SIZE];

    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<*int*>(arr, N, el);

    if (res != -1)

        cout << "FOUND: Element found at index "

             << res << endl;

    else

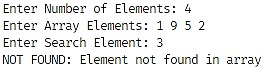
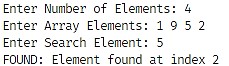
        cout << "NOT FOUND: Element not found in array"

             << endl;

    return 0;

}

#### Output



### PRACTICAL 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.

#### Code

#include <iostream>

#define MAX\_SIZE 100

using *namespace* std;

*template* <*class* T>

*int* binarySearch(T \**arr*, *int* *left*, *int* *right*, T *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[MAX\_SIZE];

    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<*int*>(arr, 0, N - 1, el);

    if (res != -1)

        cout << "FOUND: Element found at index "

             << res << endl;

    else

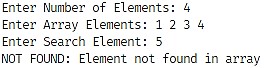
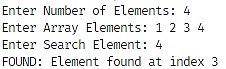
        cout << "NOT FOUND: Element not found in array"

             << endl;

    return 0;

}

#### Output



### PRACTICAL 3

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
6. Concatenate two singly linked lists

#### Code

#include <iostream>

using *namespace* std;

*void* getch();

*void* clrscr();

*template* <*class* T>

*class* Node

{

*public:*

    T info;

    Node \*ptr;

};

*template* <*class* T>

*class* SinglyLinkedList

{

*protected:*

    Node<T> \*head, \*tail;

*public:*

    // Constructor

    SinglyLinkedList()

    {

        head = tail = NULL;

    }

    // Destructor

    ~SinglyLinkedList()

    {

        if (this->isEmpty())

            return;

        Node<T> \*ptr, \*temp = head;

        while (temp != NULL)

        {

            ptr = temp->ptr;

            delete temp;

            temp = ptr;

        }

        head = tail = NULL;

        return;

    }

    // Checks if the list is empty - O(1)

*bool* isEmpty()

    {

        return (head == NULL || tail == NULL);

    }

    // Inserts a node at the beginning - O(1)

*void* insertFront(T *info*)

    {

        Node<T> \*temp = new Node<T>();

        temp->info = *info*;

        temp->ptr = head;

        if (this->isEmpty())

            tail = temp;

        head = temp;

        cout << "Inserted " << *info* << " at front...";

        this->display();

        return;

    }

*void* insertAtLoc(*int* *loc*, T *info*)

    {

        if (*loc* == 1)

        {

            this->insertFront(*info*);

            return;

        }

        Node<T> \*temp = head;

        for (*int* i = 1; temp != NULL && i < *loc* - 1; i++)

            temp = temp->ptr;

        if (temp == NULL)

        {

            cout << "Invalid location...\n";

            return;

        }

        if (temp == tail)

        {

            this->insertBack(*info*);

            return;

        }

        Node<T> \*node = new Node<T>();

        node->info = *info*;

        node->ptr = temp->ptr;

        temp->ptr = node;

        cout << "Inserted node " << *info* << " at location " << *loc* << "...";

        this->display();

        return;

    }

    // Inserts a node at the end - O(1)

*void* insertBack(T *info*)

    {

        Node<T> \*temp = new Node<T>();

        temp->info = *info*;

        temp->ptr = NULL;

        if (this->isEmpty())

            head = tail = temp;

        else

            tail->ptr = temp;

        tail = temp;

        cout << "Inserted " << *info* << " at back...";

        this->display();

        return;

    }

    // Removes a node from the beginning - O(1)

*void* deleteFront()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        Node<T> \*temp = head;

        head = temp->ptr;

        delete temp;

        if (this->isEmpty())

            tail = NULL;

        cout << "\nDeleted node at front...";

        this->display();

        return;

    }

    // Removes a node at a specified location - O(n)

*void* deleteAtLoc(*int* *loc*)

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        if (*loc* == 1)

        {

            this->deleteFront();

            return;

        }

        Node<T> \*node, \*temp = head;

        for (*int* i = 1; temp != NULL && i < *loc* - 1; i++)

            temp = temp->ptr;

        if (temp == NULL || temp->ptr == NULL)

        {

            cout << "Invalid location...\n";

            return;

        }

        if (temp == tail)

        {

            this->deleteBack();

            return;

        }

        node = temp->ptr->ptr;

        delete temp->ptr;

        temp->ptr = node;

        cout << "Deleted node "

             << "at location " << *loc* << "...";

        this->display();

        return;

    }

    // Removes a node at the end - O(n)

*void* deleteBack()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        if (head == tail)

        {

            this->deleteFront();

            return;

        }

        else

        {

            Node<T> \*temp = head;

            while (temp->ptr->ptr != NULL)

                temp = temp->ptr;

            delete temp->ptr;

            temp->ptr = NULL;

            tail = temp;

        }

        cout << "\nDeleted node at back...";

        this->display();

        return;

    }

    // Reverses the linked list - O(n)

*void* reverse()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        Node<T> \*temp = head,

                \*prev = NULL,

                \*next = NULL;

        tail = temp;

        while (temp != NULL)

        {

            next = temp->ptr;

            temp->ptr = prev;

            prev = temp;

            temp = next;

        }

        head = prev;

        cout << "\nList reversed...";

        this->display();

        return;

    }

    // Concatenates two lists - O(n)

*void* concat(SinglyLinkedList<T> &*list*)

    {

        if (!*list*.isEmpty() && !this->isEmpty())

        {

            Node<T> \*node,

                \*temp = tail,

                \*temp1 = *list*.head;

            while (temp1 != NULL)

            {

                node = new Node<T>();

                node->info = temp1->info;

                node->ptr = NULL;

                temp->ptr = node;

                temp = temp->ptr;

                temp1 = temp1->ptr;

            }

            tail = node;

            cout << "Concatenated two lists...\n";

            this->display();

        }

        else

            cout << "\nOne of the lists is empty...\n";

        return;

    }

    // Overloads the + operator - O(n)

*void* operator+(SinglyLinkedList<T> &*list*)

    {

        this->concat(*list*);

        return;

    }

    // Searches for an element - O(n)

    Node<T> \*search(T *ele*)

    {

        if (this->isEmpty())

            return nullptr;

        Node<T> \*temp = head;

        while (temp != NULL)

        {

            if (temp->info == *ele*)

                return temp;

            temp = temp->ptr;

        }

        return nullptr;

    }

    // Calculates the number of nodes - O(n)

*int* count()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return -1;

        }

*int* count = 0;

        Node<T> \*temp;

        for (temp = head; temp != NULL;

             temp = temp->ptr, count++)

            ;

        return count;

    }

    // Traverses the list and prints all nodes - O(n)

*void* display()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        Node<T> \*temp = head;

        cout << "\nList: ";

        while (temp->ptr != NULL)

        {

            cout << temp->info << " -> ";

            temp = temp->ptr;

        }

        cout << temp->info << endl;

        return;

    }

};

*int* main(*void*)

{

*int* choice, ele, info, loc, count;

    SinglyLinkedList<*int*> list, list2;

    do

    {

        cout << "\tSingly Linked List\n"

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

             << " (1) Search (2) InsertFront\n"

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

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

             << " (7) DeleteAtLoc (8) Display\n"

             << " (9) Count (10) Reverse\n"

             << " (11) Concat (0) Exit\n\n";

        cout << "Enter Choice: ";

        cin >> choice;

        switch (choice)

        {

        case 1:

            cout << "\nEnter Search Element: ";

            cin >> ele;

            if (list.search(ele) != nullptr)

                cout << "Element " << ele << " found...\n";

            else

                cout << "Element not found or List is Empty...\n";

            break;

        case 2:

            cout << "\nEnter Element: ";

            cin >> info;

            list.insertFront(info);

            break;

        case 3:

            cout << "\nEnter Element: ";

            cin >> info;

            list.insertBack(info);

            break;

        case 4:

            cout << "\nEnter Location: ";

            cin >> loc;

            cout << "Enter Element: ";

            cin >> info;

            list.insertAtLoc(loc, info);

            break;

        case 5:

            list.deleteFront();

            break;

        case 6:

            list.deleteBack();

            break;

        case 7:

            cout << "\nEnter Location: ";

            cin >> loc;

            list.deleteAtLoc(loc);

            break;

        case 8:

            list.display();

            break;

        case 9:

            count = list.count();

            if (count != -1)

                cout << "\nNumber of Nodes: " << count << endl;

            break;

        case 11:

            if (!list2.isEmpty())

            {

                cout << "\nList B:";

                list2.display();

            }

            cout << "\nNumber of Nodes to add in List B: ";

            cin >> count;

            if (count)

            {

                cout << "Enter Elements to List B: ";

                for (*int* i = 0; i < count; i++)

                {

                    cin >> info;

                    list2.insertBack(info);

                }

                list + list2;

            }

            break;

        case 10:

            list.reverse();

            break;

        case 0:

        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\_\_

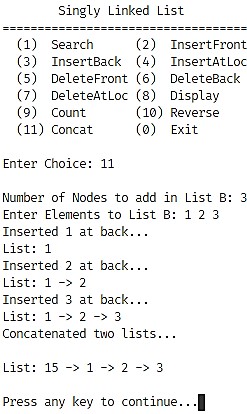
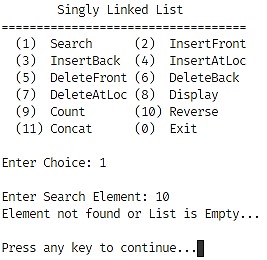
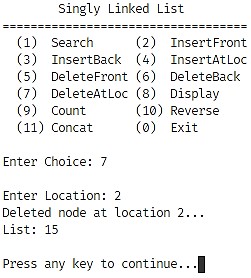
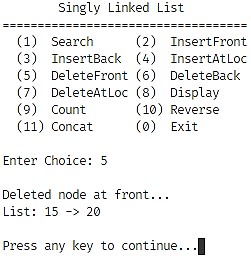
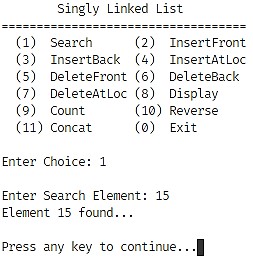
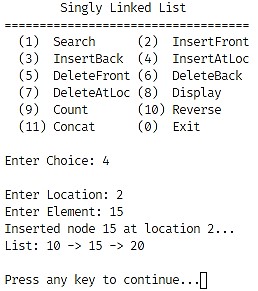
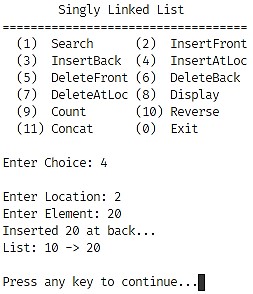
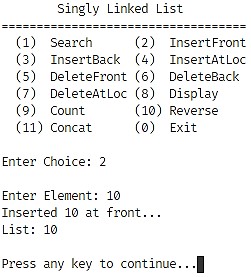
    system("clear");

#endif

    return;

}

#### Output



### PRACTICAL 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;

*void* getch();

*void* clrscr();

*template* <*class* T>

*class* Node

{

*public:*

    T info;

    Node \*prev;

    Node \*next;

};

*template* <*class* T>

*class* DoublyLinkedList

{

*protected:*

    Node<T> \*head, \*tail;

*public:*

    // Constructor

    DoublyLinkedList()

    {

        head = tail = NULL;

    }

        // Destructor

        ~DoublyLinkedList()

    {

        if (this->isEmpty())

            return;

        Node<T> \*ptr;

        for (; !isEmpty();)

        {

            ptr = head->next;

            delete head;

            head = ptr;

        }

        head = tail = ptr;

        return;

    }

    // Checks if the list is empty - O(1)

*bool* isEmpty()

    {

        return (head == NULL || tail == NULL);

    }

    // Inserts a node at the beginning - O(1)

*void* insertFront(T *info*)

    {

        Node<T> \*temp = new Node<T>();

        temp->info = *info*;

        temp->next = head;

        temp->prev = NULL;

        if (this->isEmpty())

            tail = temp;

        else

            head->prev = temp;

        head = temp;

        cout << "Inserted " << *info* << " at front...";

        this->display();

        return;

    }

    // Inserts a node at a specified location - O(n)

*void* insertAtLoc(*int* *loc*, T *info*)

    {

        if (*loc* == 1)

        {

            this->insertFront(*info*);

            return;

        }

        Node<T> \*temp = head;

        Page 21 of 166 for (*int* i = 1; temp != NULL && i < loc - 1; i++)

            temp = temp->next;

        if (temp == NULL)

        {

            cout << "Invalid location...\n";

            return;

        }

        if (temp == tail)

        {

            this->insertBack(*info*);

            return;

        }

        Node<T> \*node = new Node<T>();

        node->info = *info*;

        node->next = temp->next;

        node->prev = temp;

        temp->next->prev = node;

        temp->next = node;

        cout << "Inserted node " << *info* << " at location " << *loc* << "...";

        this->display();

        return;

    }

    // Inserts a node at the end - O(1)

*void* insertBack(T *info*)

    {

        Node<T> \*temp = new Node<T>();

        temp->info = *info*;

        temp->next = NULL;

        temp->prev = tail;

        if (this->isEmpty())

            head = tail = temp;

        else

            tail->next = temp;

        tail = temp;

        cout << "Inserted " << *info* << " at back...";

        this->display();

        return;

    }

    // Removes a node from the beginning - O(1)

*void* deleteFront()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        Page 22 of 166 Node<T> \*temp = head;

        head = temp->next;

        if (this->isEmpty())

            tail = NULL;

        else

            head->prev = NULL;

        delete temp;

        cout << "\nDeleted node at front...";

        this->display();

        return;

    }

    // Removes a node at a specified location - O(n)

*void* deleteAtLoc(*int* *loc*)

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        if (*loc* == 1)

        {

            this->deleteFront();

            return;

        }

        Node<T> \*node, \*temp = head;

        for (*int* i = 1; temp != NULL && i < *loc* - 1; i++)

            temp = temp->next;

        if (temp == NULL || temp->next == NULL)

        {

            cout << "Invalid location...\n";

            return;

        }

        if (temp->next == tail)

        {

            this->deleteBack();

            return;

        }

        node = temp->next->next;

        node->prev = temp;

        delete temp->next;

        temp->next = node;

        cout << "Deleted node "

             << "at location " << *loc* << "...";

        this->display();

        return;

    }

        // Removes a node at the end - O(1)

*void*

        deleteBack()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        Node<T> \*temp = tail;

        tail = temp->prev;

        if (this->isEmpty())

            head = NULL;

        else

            tail->next = NULL;

        delete temp;

        cout << "\nDeleted node at back...";

        this->display();

        return;

    }

    // Reverses the linked list - O(n)

*void* reverse()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        Node<T> \*temp = head,

                \*temp1 = NULL;

        tail = temp;

        while (temp != NULL)

        {

            temp1 = temp->prev;

            temp->prev = temp->next;

            temp->next = temp1;

            temp = temp->prev;

        }

        if (temp1 != NULL)

            head = temp1->prev;

        cout << "\nList reversed...";

        this->display();

        return;

    }

    // Concatenates two lists - O(n)

*void* concat(DoublyLinkedList<T> &*list*)

    {

        Page 24 of 166 if (!list.isEmpty() && !this->isEmpty())

        {

            Node<T> \*node,

                \*temp = tail,

                \*temp1 = list.head;

            while (temp1 != NULL)

            {

                node = new Node<T>();

                node->info = temp1->info;

                node->next = NULL;

                node->prev = temp;

                temp->next = node;

                temp = temp->next;

                temp1 = temp1->next;

            }

            tail = node;

            cout << "Concatenated two lists...\n";

            this->display();

        }

        else cout << "\nOne of the lists is empty...\n";

        return;

    }

    // Overloads the + operator - O(n)

*void* operator+(DoublyLinkedList<T> &*list*)

    {

        this->concat(*list*);

        return;

    }

    // Searches for an element - O(n)

    Node<T> \*search(T *ele*)

    {

        if (this->isEmpty())

            return nullptr;

        Node<T> \*temp = head;

        while (temp != NULL)

        {

            if (temp->info == *ele*)

                return temp;

            temp = temp->next;

        }

        return nullptr;

    }

    // Calculates the number of nodes - O(n)

*int* count()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return -1;

        }

*int* count = 0;

        Node<T> \*temp;

        for (temp = head; temp != NULL;

             temp = temp->next, count++)

            ;

        return count;

    }

    // Traverses the list and prints all nodes - O(n)

*void* display()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        Node<T> \*temp = head;

        cout << "\nList: ";

        while (temp->next != NULL)

        {

            cout << temp->info << " -> ";

            temp = temp->next;

        }

        cout << temp->info << endl;

        return;

    }

};

*int* main(*void*)

{

*int* info, ele, choice, loc, count;

    DoublyLinkedList<*int*> list, list2;

    do

    {

        cout << "\tDoubly Linked List\n"

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

             << " (1) Search (2) InsertFront\n"

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

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

             << " (7) DeleteAtLoc (8) Display\n"

             << " (9) Count (10) Reverse\n"

             << " (11) Concat (0) Exit\n\n";

        cout << "Enter Choice: ";

        cin >> choice;

        switch (choice)

        {

        case 1:

            cout << "\nEnter Search Element: ";

            cin >> ele;

            if (list.search(ele) != nullptr)

                cout << "Element " << ele << " found...\n";

            else

                cout << "Element not found or List is Empty...\n";

            break;

        case 2:

            cout << "\nEnter Element: ";

            cin >> info;

            list.insertFront(info);

            break;

        case 3:

            cout << "\nEnter Element: ";

            cin >> info;

            list.insertBack(info);

            break;

        case 4:

            cout << "\nEnter Location: ";

            cin >> loc;

            cout << "Enter Element: ";

            cin >> info;

            list.insertAtLoc(loc, info);

            break;

        case 5:

            list.deleteFront();

            break;

        case 6:

            list.deleteBack();

            break;

        case 7:

            cout << "\nEnter Location: ";

            cin >> loc;

            list.deleteAtLoc(loc);

            break;

        case 8:

            list.display();

            break;

        case 9:

            count = list.count();

            if (count != -1)

                cout << "\nNumber of Nodes: " << count << endl;

            break;

        case 10 : list.reverse();

            break;

        case 11:

            if (!list2.isEmpty())

            {

                cout << "\nList B:";

                list2.display();

            }

            cout << "\nNumber of Nodes to add in List B: ";

            cin >> count;

            if (count)

            {

                cout << "Enter Elements to List B: ";

                for (*int* i = 0; i < count; i++)

                {

                    cin >> info;

                    list2.insertBack(info);

                }

                list + list2;

            }

            break;

        case 0:

        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\_\_

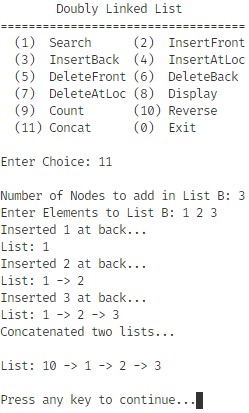
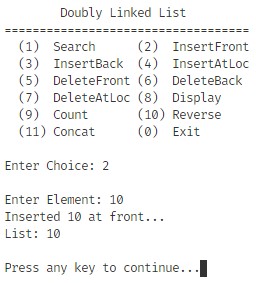
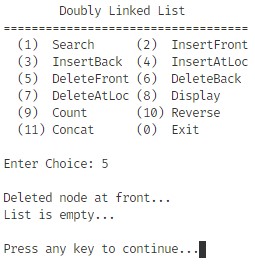
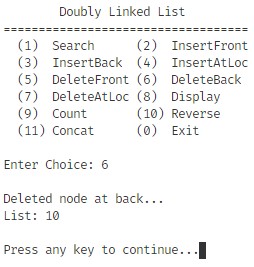
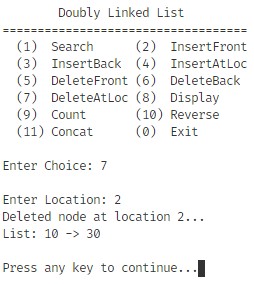
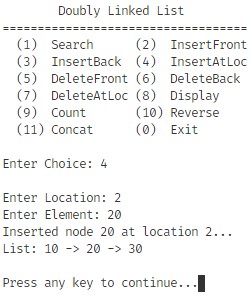
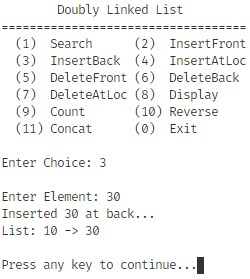
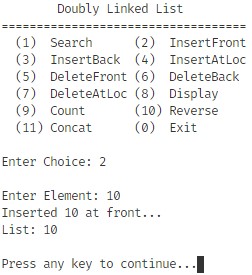
    system("clear");

#endif

    return;

}

#### Output



### PRACTICAL 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>

using *namespace* std;

*void* getch();

*void* clrscr();

*template* <*class* T>

*class* Node

{

*public:*

    T info;

    Node \*prev;

    Node \*next;

};

*template* <*class* T>

*class* CircularDoublyLinkedList

{

*protected:*

    Node<T> \*tail;

*public:*

    // Constructor

    CircularDoublyLinkedList()

    {

        tail = NULL;

    }

    // Destructor

    ~CircularDoublyLinkedList()

    {

        if (this->isEmpty())

            return;

        Node<T> \*ptr, \*temp = tail->next;

        while (temp != tail)

        {

            ptr = temp;

            temp = ptr->next;

            delete ptr;

        }

        delete temp;

        tail = NULL;

        return;

    }

    // Checks if the list is empty - O(1)

*bool* isEmpty()

    {

        return tail == NULL;

    }

    // Inserts a node at the beginning - O(1)

*void* insertFront(T *info*)

    {

        Node<T> \*temp = new Node<T>();

        temp->info = *info*;

        if (this->isEmpty())

        {

            temp->next = temp;

            temp->prev = temp;

            tail = temp;

        }

        else

        {

            temp->prev = tail;

            temp->next = tail->next;

            tail->next->prev = temp;

            tail->next = temp;

        }

        cout << "Inserted " << *info* << " at front...";

        this->display();

        return;

    }

    // Inserts a node at a specified location - O(n)

*void* insertAtLoc(T *searchEle*, T *info*)

    {

*int* loc = 0;

        if (this->isEmpty())

        {

            cout << "List Empty...\n";

            return;

        }

*int* i = 0;

        Node<T> \*temp = tail->next;

        do

        {

            ++i;

            if (temp->info == *searchEle*)

                loc = i;

            temp = temp->next;

        } while (temp != tail->next);

        if (loc == 0)

        {

            cout << "Search Element Not Found...\n";

            return;

        }

        loc++;

        if (loc == 1)

        {

            this->insertFront(*info*);

            return;

        }

*int* size = this->count();

        if (loc > size + 1 || loc < 1)

        {

            cout << "Invalid location...\n";

            return;

        }

        if (loc == size + 1)

        {

            this->insertBack(*info*);

            return;

        }

        temp = tail->next;

        for (*int* i = 1; temp->next != tail && i < loc - 1; i++)

            temp = temp->next;

        Node<T> \*node = new Node<T>();

        node->info = *info*;

        node->next = temp->next;

        temp->next->prev = node;

        node->prev = temp;

        temp->next = node;

        cout << "Inserted node " << *info* << " at location " << loc << "...";

        this->display();

        return;

    }

    // Inserts a node at the end - O(1)

*void* insertBack(T *info*)

    {

        Node<T> \*temp = new Node<T>();

        temp->info = *info*;

        if (this->isEmpty())

        {

            temp->next = temp;

            temp->prev = temp;

        }

        else

        {

            temp->next = tail->next;

            temp->prev = tail;

            tail->next = temp;

            temp->next->prev = temp;

        }

        tail = temp;

        cout << "Inserted " << *info* << " at back...";

        this->display();

        return;

    }

    // Removes a node from the beginning - O(1)

*void* deleteFront()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        if (tail->next == tail)

        {

            delete tail;

            tail = NULL;

        }

        else

        {

            Node<T> \*temp = tail->next;

            tail->next = temp->next;

            temp->next->prev = tail;

            delete temp;

        }

        cout << "\nDeleted node at front...";

        this->display();

        return;

    }

    // Removes a node at a specified location - O(n)

*void* deleteAtLoc(T *ele*)

    {

*int* loc = 0;

        if (this->isEmpty())

        {

            cout << "List Empty...\n";

            return;

        }

*int* i = 0;

        Node<T> \*temp = tail->next;

        do

        {

            ++i;

            if (temp->info == *ele*)

                loc = i;

            temp = temp->next;

        } while (temp != tail->next);

        if (loc == 0)

        {

            cout << "Search Element Not Found...\n";

            return;

        }

*int* size = this->count();

        if (loc > size || loc < 1)

        {

            cout << "Invalid location...\n";

            return;

        }

        if (loc == size)

        {

            this->deleteBack();

            return;

        }

        temp = tail->next;

        for (*int* i = 1; temp->next != tail && i < loc; i++)

            temp = temp->next;

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

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

        delete temp;

        cout << "Deleted node "

             << "at location " << loc << "...";

        this->display();

        return;

    }

    // Removes a node at the end - O(1)

*void* deleteBack()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        if (tail->next == tail)

        {

            delete tail;

            tail = NULL;

        }

        else

        {

            Node<T> \*temp = tail;

            tail = temp->prev;

            temp->next->prev = tail;

            tail->next = temp->next;

            delete temp;

        }

        cout << "\nDeleted node at back...";

        this->display();

        return;

    }

    // Reverses the linked list - O(n)

*void* reverse()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        Node<T> \*temp = tail->next,

                \*headRef = tail->next,

                \*temp1 = NULL;

        do

        {

            temp1 = temp->prev;

            temp->prev = temp->next;

            temp->next = temp1;

            temp = temp->prev;

        } while (temp != headRef);

        tail = headRef;

        cout << "\nList reversed...";

        this->display();

        return;

    }

    // Concatenates two lists - O(n)

*void* concat(CircularDoublyLinkedList<T> &*list*)

    {

        if (!*list*.isEmpty() && !this->isEmpty())

        {

            tail->next->prev = *list*.tail;

            Node<T> \*temp = tail->next;

            tail->next = *list*.tail->next;

*list*.tail->next = temp;

            tail = *list*.tail;

            cout << "Concatenated two lists...\n";

            this->display();

        }

        else

            cout << "\nOne of the lists is empty...\n";

        return;

    }

    // Overloads the + operator - O(n)

*void* operator+(CircularDoublyLinkedList<T> &*list*)

    {

        this->concat(*list*);

        return;

    }

    // Searches for an element - O(n)

    Node<T> \*search(T *ele*)

    {

        if (this->isEmpty())

            return nullptr;

        Node<T> \*temp = tail->next;

        do

        {

            if (temp->info == *ele*)

                return temp;

            temp = temp->next;

        } while (temp != tail->next);

        return nullptr;

    }

    // Calculates the number of nodes - O(n)

*int* count()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return -1;

        }

*int* count = 0;

        Node<T> \*temp = tail->next;

        do

        {

            temp = temp->next;

            count++;

        } while (temp != tail->next);

        return count;

    }

    // Traverses the list and prints all nodes - O(n)

*void* display()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        Node<T> \*temp = tail->next;

        cout << "\nList: ";

        while (temp != tail)

        {

            cout << temp->info << " -> ";

            temp = temp->next;

        }

        cout << temp->info << endl;

        return;

    }

};

*int* main(*void*)

{

*int* info, ele, choice, loc, count;

    CircularDoublyLinkedList<*int*> list, list2;

    do

    {

        cout << "\tCircular Doubly Linked List\n"

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

             << " (1) Search (2) InsertFront\n"

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

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

             << " (7) DeleteAtLoc (8) Display\n"

             << " (9) Count (10) Reverse\n"

             << " (11) Concat (0) Exit\n\n";

        cout << "Enter Choice: ";

        cin >> choice;

        switch (choice)

        {

        case 1:

            cout << "\nEnter Search Element: ";

            cin >> ele;

            if (list.search(ele) != nullptr)

                cout << "Element " << ele << " found...\n";

            else

                cout << "Element not found or List is Empty...\n";

            break;

        case 2:

            cout << "\nEnter Element: ";

            cin >> info;

            list.insertFront(info);

            break;

        case 3:

            cout << "\nEnter Element: ";

            cin >> info;

            list.insertBack(info);

            break;

        case 4:

            cout << "\nInsert After: ";

            cin >> ele;

            cout << "Enter Element: ";

            cin >> info;

            list.insertAtLoc(ele, info);

            break;

        case 5:

            list.deleteFront();

            break;

        case 6:

            list.deleteBack();

            break;

        case 7:

            cout << "\nEnter Element: ";

            cin >> ele;

            list.deleteAtLoc(ele);

            break;

        case 8:

            list.display();

            break;

        case 9:

            count = list.count();

            if (count != -1)

                cout << "\nNumber of Nodes: " << count << endl;

            break;

        case 10:

            list.reverse();

            break;

        case 11:

            if (!list2.isEmpty())

            {

                cout << "\nList B:";

                list2.display();

            }

            cout << "\nNumber of Nodes to add in List B: ";

            cin >> count;

            if (count)

            {

                cout << "Enter Elements to List B: ";

                for (*int* i = 0; i < count; i++)

                {

                    cin >> info;

                    list2.insertBack(info);

                }

                list + list2;

            }

            break;

        case 0:

        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\_\_

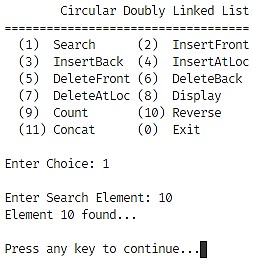
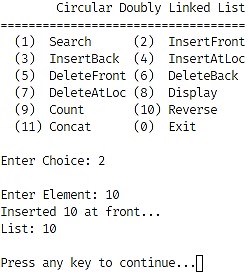
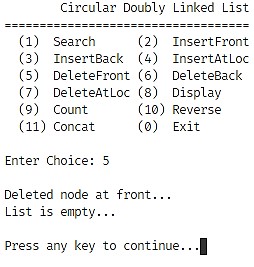
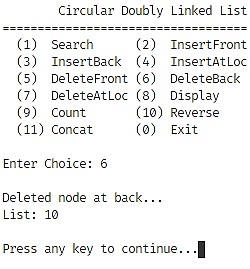
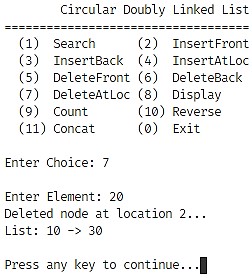
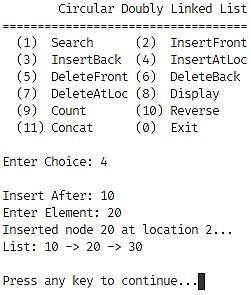
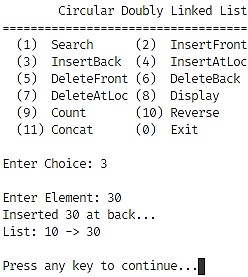
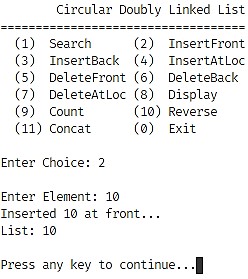
    system("clear");

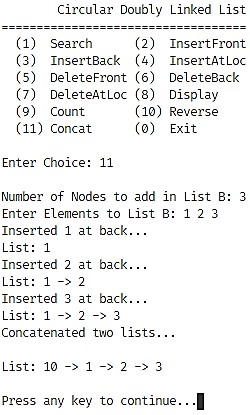
#endif

    return;

}

#### Output





### PRACTICAL 6

Objective

Implement a Stack using Array representation.

#### Code

#include <iostream>

#define MAX\_SIZE 100

using *namespace* std;

*void* getch();

*void* clrscr();

*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();

    }

*void* display()

    {

        if (this->isEmpty())

        {

            cout << "Stack Empty";

            return;

        }

*int* i;

        cout << "Stack: ";

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

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

        cout << this->arr[i] << endl;

        return;

    }

};

*int* main(*void*)

{

*int* n, el, res, choice;

    cout << "Enter size of stack: ";

    cin >> n;

    Stack<*int*> stack(n);

    do

    {

        cout << "\tStack - Arrays\n"

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

             << " (1) Push (2) Pop\n"

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

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

        cout << "Enter Choice: ";

        cin >> choice;

        switch (choice)

        {

        case 1:

            cout << "\nEnter Element: ";

            cin >> el;

            res = stack.push(el);

            if (res)

            {

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

                stack.display();

            }

            break;

        case 2:

            res = stack.pop();

            if (res)

            {

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

                stack.display();

            }

            break;

        case 3:

            cout << "\nTop Element: "

                 << stack.top() << endl;

            break;

        case 4:

            stack.clear();

            break;

        case 5:

            stack.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\_\_

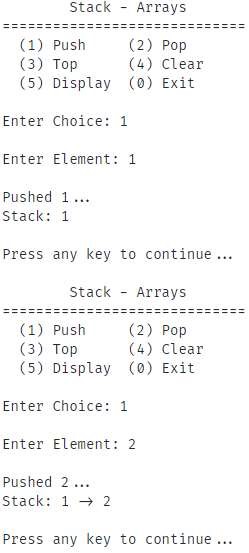
    system("clear");

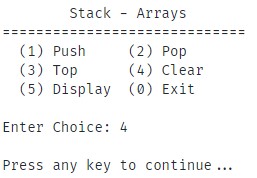
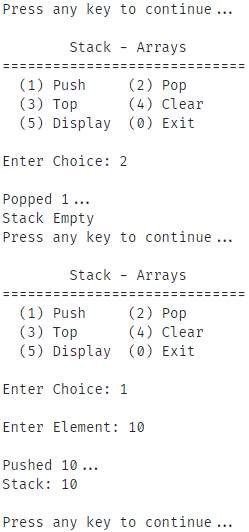
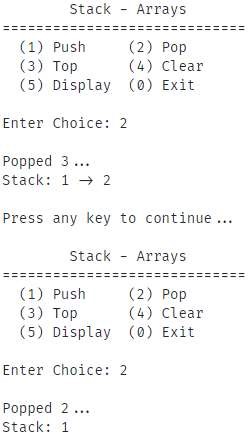
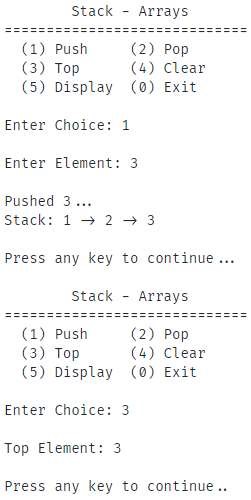
#endif

    return;

}

#### Output





### PRACTICAL 7

Objective

Implement a Stack using Linked List representation.

#### Code

#include <iostream>

using *namespace* std;

*void* getch();

*void* clrscr();

*template* <*class* T>

*class* Node

{

*public:*

    T info;

    Node \*ptr;

};

*template* <*class* T>

*class* SinglyLinkedList

{

*protected:*

    Node<T> \*head, \*tail;

*public:*

    // Constructor

    SinglyLinkedList()

    {

        head = tail = NULL;

    }

    // Destructor

    ~SinglyLinkedList()

    {

        if (this->isEmpty())

            return;

        Node<T> \*ptr, \*temp = head;

        while (temp != NULL)

        {

            ptr = temp->ptr;

            delete temp;

            temp = ptr;

        }

        head = tail = NULL;

        return;

    }

    // Returns the data on the head of the list - O(1)

    T getHead()

    {

        return this->isEmpty() ? (T)(NULL) : head->info;

    }

    // Checks if the list is empty - O(1)

*bool* isEmpty()

    {

        return (head == NULL || tail == NULL);

    }

    // Inserts a node at the beginning - O(1)

*void* insertFront(T *info*)

    {

        Node<T> \*temp = new Node<T>();

        temp->info = *info*;

        temp->ptr = head;

        if (this->isEmpty())

            tail = temp;

        head = temp;

        return;

    }

    // Inserts a node at a specified location - O(n)

*void* insertAtLoc(*int* *loc*, T *info*)

    {

        if (*loc* == 1)

        {

            this->insertFront(*info*);

            return;

        }

        Node<T> \*temp = head;

        for (*int* i = 1; temp != NULL && i < *loc* - 1; i++)

            temp = temp->ptr;

        if (temp == NULL)

        {

            cout << "Invalid location...\n";

            return;

        }

        if (temp == tail)

        {

            this->insertBack(*info*);

            return;

        }

        Node<T> \*node = new Node<T>();

        node->info = *info*;

        node->ptr = temp->ptr;

        temp->ptr = node;

        return;

    }

    // Inserts a node at the end - O(1)

*void* insertBack(T *info*)

    {

        Node<T> \*temp = new Node<T>();

        temp->info = *info*;

        temp->ptr = NULL;

        if (this->isEmpty())

            head = tail = temp;

        else

            tail->ptr = temp;

        tail = temp;

        return;

    }

    // Removes a node from the beginning - O(1)

*void* deleteFront()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        Node<T> \*temp = head;

        head = temp->ptr;

        delete temp;

        if (this->isEmpty())

            tail = NULL;

        return;

    }

    // Removes a node at a specified location - O(n)

*void* deleteAtLoc(*int* *loc*)

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        if (*loc* == 1)

        {

            this->deleteFront();

            return;

        }

        Node<T> \*node, \*temp = head;

        for (*int* i = 1; temp != NULL && i < *loc* - 1; i++)

            temp = temp->ptr;

        if (temp == NULL || temp->ptr == NULL)

        {

            cout << "Invalid location...\n";

            return;

        }

        if (temp == tail)

        {

            this->deleteBack();

            return;

        }

        node = temp->ptr->ptr;

        delete temp->ptr;

        temp->ptr = node;

        return;

    }

    // Removes a node at the end - O(n)

*void* deleteBack()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        if (head == tail)

        {

            this->deleteFront();

            return;

        }

        else

        {

            Node<T> \*temp = head;

            while (temp->ptr->ptr != NULL)

                temp = temp->ptr;

            delete temp->ptr;

            temp->ptr = NULL;

            tail = temp;

        }

        return;

    }

    // Reverses the linked list - O(n)

*void* reverse()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        Node<T> \*temp = head,

                \*prev = NULL,

                \*next = NULL;

        tail = temp;

        while (temp != NULL)

        {

            next = temp->ptr;

            temp->ptr = prev;

            prev = temp;

            temp = next;

        }

        head = prev;

        return;

    }

    // Concatenates two lists - O(n)

*void* concat(SinglyLinkedList<T> &*list*)

    {

        if (!*list*.isEmpty() && !this->isEmpty())

        {

            Node<T> \*node,

                \*temp = tail,

                \*temp1 = *list*.head;

            while (temp1 != NULL)

            {

                node = new Node<T>();

                node->info = temp1->info;

                node->ptr = NULL;

                temp->ptr = node;

                temp = temp->ptr;

                temp1 = temp1->ptr;

            }

            tail = node;

        }

        return;

    }

    // Overloads the + operator - O(n)

*void* operator+(SinglyLinkedList<T> &*list*)

    {

        this->concat(*list*);

        return;

    }

    // Searches for an element - O(n)

*bool* search(T *ele*)

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return false;

        }

        Node<T> \*temp = head;

        while (temp != NULL)

        {

            if (temp->info == *ele*)

                return true;

            temp = temp->ptr;

        }

        return false;

    }

    // Calculates the number of nodes - O(n)

*int* count()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return -1;

        }

*int* count = 0;

        Node<T> \*temp;

        for (temp = head; temp != NULL;

             temp = temp->ptr, count++)

            ;

        return count;

    }

    // Traverses the list and prints all nodes - O(n)

*void* display()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        Node<T> \*temp = head;

        while (temp->ptr != NULL)

        {

            cout << temp->info << " <- ";

            temp = temp->ptr;

        }

        cout << temp->info << endl;

        return;

    }

};

*template* <*class* T>

*class* Stack

{

*protected:*

    SinglyLinkedList<T> list;

*public:*

*bool* push(T *ele*)

    {

        this->list.insertFront(*ele*);

        return true;

    }

    T pop()

    {

        if (this->isEmpty())

        {

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

            return (T)(NULL);

        }

        T ele = this->list.getHead();

        this->list.deleteFront();

        return ele;

    }

    T top()

    {

        if (this->isEmpty())

        {

            cout << "Stack Empty";

            return (T)(NULL);

        }

        return this->list.getHead();

    }

*bool* isEmpty()

    {

        return this->list.isEmpty();

    }

*void* clear()

    {

        while (!this->isEmpty())

            this->pop();

    }

*void* display()

    {

        if (this->isEmpty())

        {

            cout << "Stack Empty";

            return;

        }

*int* i;

        cout << "Stack: ";

        this->list.display();

        return;

    }

};

*int* main(*void*)

{

*int* el, res, choice;

    Stack<*int*> stack;

    do

    {

        cout << "\tStack - SLList\n"

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

             << " (1) Push (2) Pop\n"

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

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

        cout << "Enter Choice: ";

        cin >> choice;

        switch (choice)

        {

        case 1:

            cout << "\nEnter Element: ";

            cin >> el;

            res = stack.push(el);

            if (res)

            {

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

                stack.display();

            }

            break;

        case 2:

            res = stack.pop();

            if (res)

            {

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

                stack.display();

            }

            break;

        case 3:

            cout << "\nTop Element: "

                 << stack.top() << endl;

            break;

        case 4:

            stack.clear();

            break;

        case 5:

            stack.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\_\_

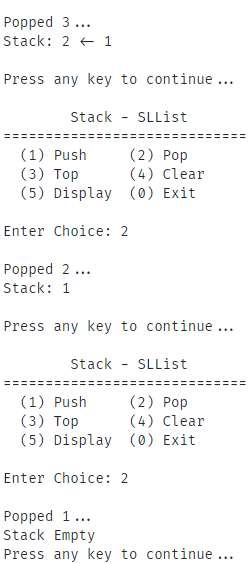
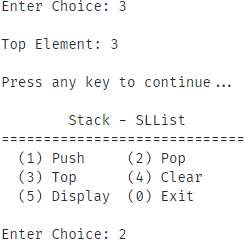
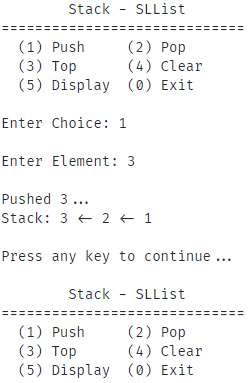
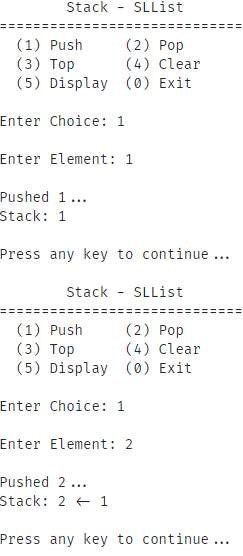
 system("clear");

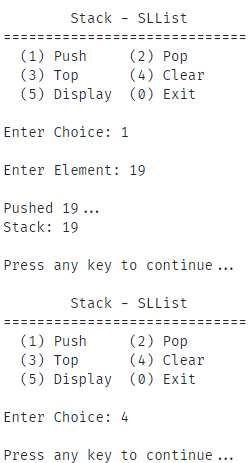
#endif

    return;

}

#### Output





### PRACTICAL 8

Objective

Implement a Queue using Circular Array representation.

#### Code

#include <iostream>

#define MAX\_SIZE 100

using *namespace* std;

*void* getch();

*void* clrscr();

*template* <*class* T>

*class* Queue

{

*protected:*

    T arr[MAX\_SIZE];

*int* front, rear, size;

*public:*

    Queue(*int* *size* = 5)

    {

        this->front = -1;

        this->rear = -1;

        this->size = *size*;

    }

*bool* enqueue(T *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;

        }

    }

    T dequeue()

    {

        if (this->isEmpty())

        {

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

            return (T)(NULL);

        }

        else

        {

            T 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;

        }

    }

    T frontEl()

    {

        if (this->isEmpty())

        {

            cout << "Queue Empty";

            return (T)(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(*void*)

{

*int* n, el, res, choice;

    cout << "Enter Size of Queue: ";

    cin >> n;

    Queue<*int*> 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");

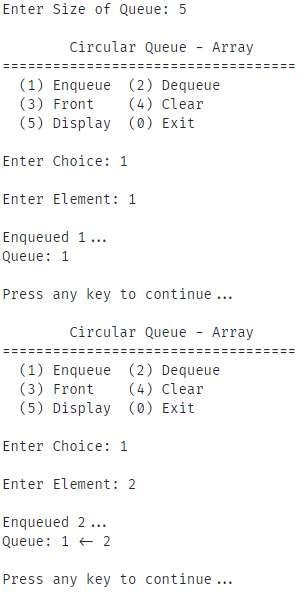
#elif \_\_unix\_\_

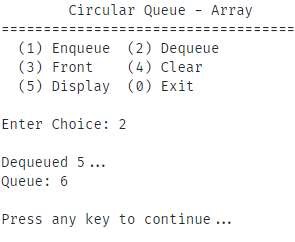
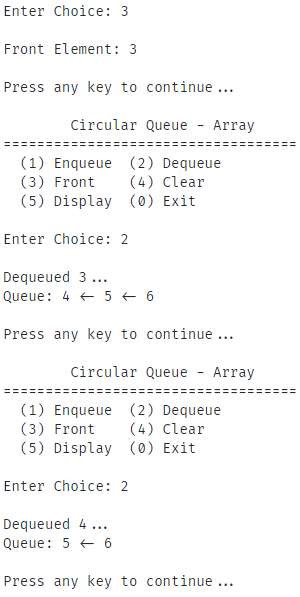
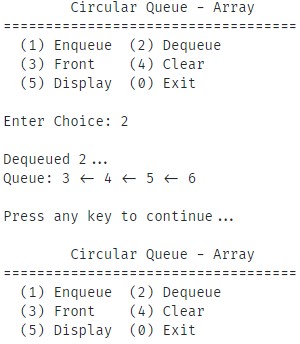
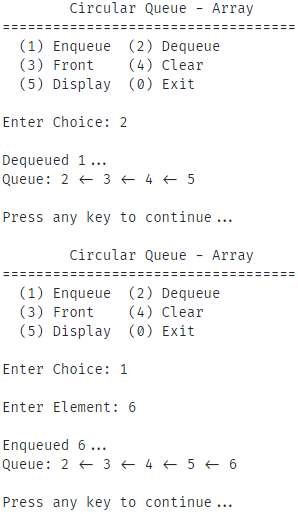
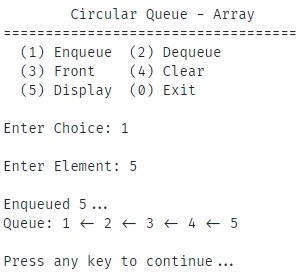
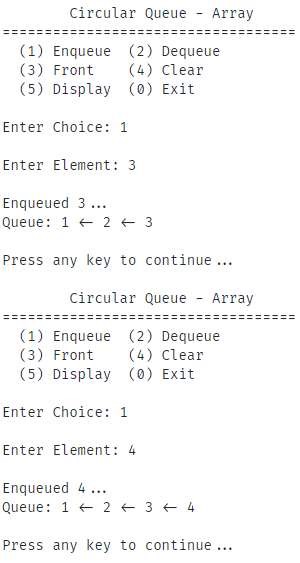
    system("clear");

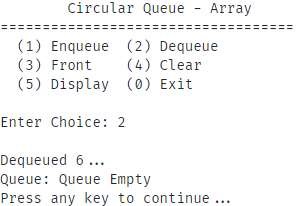
#endif

    return;

}Output







### PRACTICAL 9

Objective

Implement a Queue using Circular Linked List representation.

#### Code

#include <iostream>

using *namespace* std;

*template* <*class* T>

*class* Node

{

*public:*

    T info;

    Node \*ptr;

};

*template* <*class* T>

*class* CircularSinglyLinkedList

{

*public:*

    Node<T> \*tail;

    // Constructor

    CircularSinglyLinkedList()

    {

        tail = NULL;

    }

    // Destructor

    ~CircularSinglyLinkedList()

    {

        if (this->isEmpty())

            return;

        Node<T> \*ptr, \*temp = tail->ptr;

        while (temp != tail)

        {

            ptr = temp;

            temp = ptr->ptr;

            delete ptr;

        }

        delete temp;

        tail = NULL;

        return;

    }

    // Checks if the list is empty - O(1)

*bool* isEmpty()

    {

        return tail == NULL;

    }

    // Inserts a node at the beginning - O(1)

*void* insertFront(T *info*)

    {

        Node<T> \*temp = new Node<T>();

        temp->info = *info*;

        if (this->isEmpty())

        {

            temp->ptr = temp;

            tail = temp;

        }

        else

        {

            temp->ptr = tail->ptr;

            tail->ptr = temp;

        }

        return;

    }

    // Inserts a node at a specified location - O(n)

*void* insertAtLoc(*int* *loc*, T *info*)

    {

        if (*loc* == 1)

        {

            this->insertFront(*info*);

            return;

        }

*int* size = this->count();

        if (*loc* > size + 1 || *loc* < 1)

        {

            cout << "Invalid location...\n";

            return;

        }

        if (*loc* == size + 1)

        {

            this->insertBack(*info*);

            return;

        }

        Node<T> \*temp = tail->ptr;

        for (*int* i = 1; temp->ptr != tail && i < *loc* - 1; i++)

            temp = temp->ptr;

        Node<T> \*node = new Node<T>();

        node->info = *info*;

        node->ptr = temp->ptr;

        temp->ptr = node;

        return;

    }

    // Inserts a node at the end - O(1)

*void* insertBack(T *info*)

    {

        Node<T> \*temp = new Node<T>();

        temp->info = *info*;

        if (this->isEmpty())

            temp->ptr = temp;

        else

        {

            temp->ptr = tail->ptr;

            tail->ptr = temp;

        }

        tail = temp;

        return;

    }

    // Removes a node from the beginning - O(1)

*void* deleteFront()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        else if (tail->ptr == tail)

        {

            delete tail;

            tail = NULL;

        }

        else

        {

            Node<T> \*temp;

            temp = tail->ptr->ptr;

            delete tail->ptr;

            tail->ptr = temp;

        }

        return;

    }

    // Removes a node at a specified location - O(n)

*void* deleteAtLoc(*int* *loc*)

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

*int* size = this->count();

        if (*loc* > size || *loc* < 1)

        {

            cout << "Invalid location...\n";

            return;

        }

        if (*loc* == size)

        {

            this->deleteBack();

            return;

        }

        Node<T> \*node, \*temp = tail->ptr;

        for (*int* i = 1; temp->ptr != tail && i < *loc* - 1; i++)

            temp = temp->ptr;

        node = temp->ptr->ptr;

        delete temp->ptr;

        temp->ptr = node;

        return;

    }

    // Removes a node at the end - O(n)

*void* deleteBack()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        else if (tail->ptr == tail)

        {

            delete tail;

            tail = NULL;

        }

        else

        {

            Node<T> \*temp = tail->ptr;

            while (temp->ptr != tail)

                temp = temp->ptr;

            temp->ptr = tail->ptr;

            delete tail;

            tail = temp;

        }

        return;

    }

    // Traverses the list and prints all nodes - O(n)

*void* display()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        Node<T> \*temp = tail->ptr;

        while (temp != tail)

        {

            cout << temp->info << " -> ";

            temp = temp->ptr;

        }

        cout << temp->info << endl;

        return;

    }

};

*template* <*class* T>

*class* Queue

{

*protected:*

    Node<T> \*front, \*rear;

    CircularSinglyLinkedList<T> list;

*public:*

    Queue()

    {

        this->front = this->list.tail;

        this->rear = this->list.tail;

    }

*bool* enqueue(T *ele*)

    {

        this->list.insertBack(*ele*);

        this->front = this->list.tail->ptr;

        this->rear = this->list.tail;

        return true;

    }

    T dequeue()

    {

        if (this->isEmpty())

        {

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

            return (T)(NULL);

        }

        T temp = this->front->info;

        this->list.deleteFront();

        if (this->isEmpty())

            this->front = this->list.tail;

        else

            this->front = this->list.tail->ptr;

        this->rear = this->list.tail;

        return temp;

    }

    T frontEl()

    {

        if (this->isEmpty())

        {

            cout << "Queue Empty";

            return (T)(NULL);

        }

        return this->front->info;

    }

*bool* isEmpty()

    {

        return this->list.isEmpty();

    }

*void* clear()

    {

        while (!this->isEmpty())

            this->dequeue();

    }

*void* display()

    {

        if (this->isEmpty())

        {

            cout << " Queue Empty ";

                return;

        }

        this->list.display();

        return;

    }

};

*int* main(*void*)

{

*int* el, res, choice;

    Queue<*int*> q;

    do

    {

        cout << "\tCircular Queue - CSLList\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\_\_

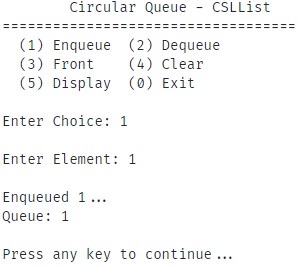
    system("clear");

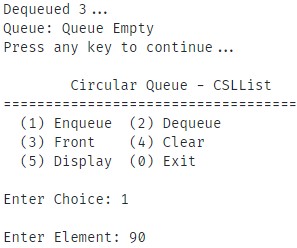
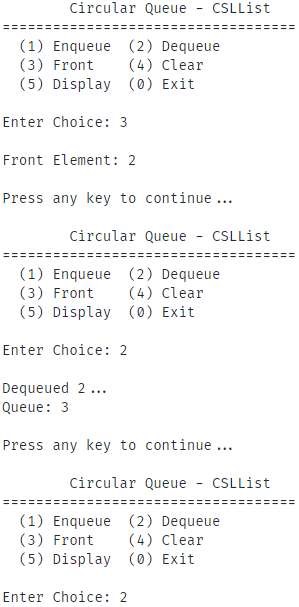
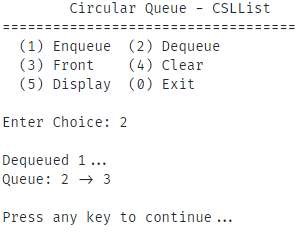
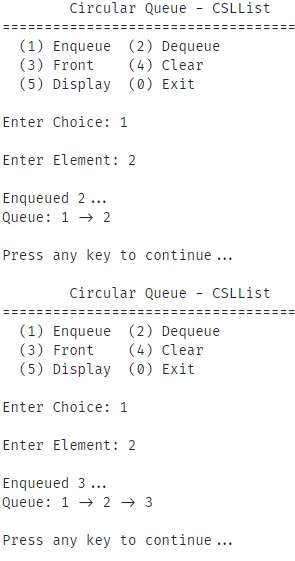
#endif

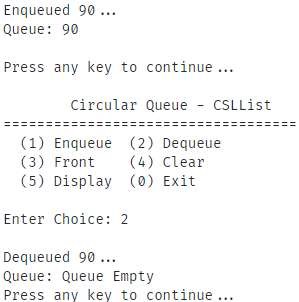
    return;

}

#### Output







### PRACTICAL 10

Objective

Implement Double-ended Queues using Linked List representation.

#### Code

#include <iostream>

using *namespace* std;

*void* getch();

*void* clrscr();

*template* <*class* T>

*class* Node

{

*public:*

    T info;

    Node \*prev;

    Node \*next;

};

*template* <*class* T>

*class* DoublyLinkedList

{

*public:*

    Node<T> \*head, \*tail;

    // Constructor

    DoublyLinkedList()

    {

        head = tail = NULL;

    }

    // Destructor

    ~DoublyLinkedList()

    {

        if (this->isEmpty())

            return;

        Node<T> \*ptr;

        for (; !isEmpty();)

        {

            ptr = head->next;

            delete head;

            head = ptr;

        }

        head = tail = ptr;

        return;

    }

    // Checks if the list is empty - O(1)

*bool* isEmpty()

    {

        return (head == NULL || tail == NULL);

    }

    // Inserts a node at the beginning - O(1)

*void* insertFront(T *info*)

    {

        Node<T> \*temp = new Node<T>();

        temp->info = *info*;

        temp->next = head;

        temp->prev = NULL;

        if (this->isEmpty())

            tail = temp;

        else

            head->prev = temp;

        head = temp;

        return;

    }

    // Inserts a node at the end - O(1)

*void* insertBack(T *info*)

    {

        Node<T> \*temp = new Node<T>();

        temp->info = *info*;

        temp->next = NULL;

        temp->prev = tail;

        if (this->isEmpty())

            head = tail = temp;

        else

            tail->next = temp;

        tail = temp;

        return;

    }

    // Removes a node from the beginning - O(1)

*void* deleteFront()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        Node<T> \*temp = head;

        head = temp->next;

        if (this->isEmpty())

            tail = NULL;

        else

            head->prev = NULL;

        delete temp;

        return;

    }

    // Removes a node at the end - O(1)

*void* deleteBack()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        Node<T> \*temp = tail;

        tail = temp->prev;

        if (this->isEmpty())

            head = NULL;

        else

            tail->next = NULL;

        delete temp;

        return;

    }

    // Traverses the list and prints all nodes - O(n)

*void* display()

    {

        if (this->isEmpty())

        {

            cout << "\nList is empty...\n";

            return;

        }

        Node<T> \*temp = head;

        while (temp->next != NULL)

        {

            cout << temp->info << " -> ";

            temp = temp->next;

        }

        cout << temp->info << endl;

        return;

    }

};

*template* <*class* T>

*class* DoublyEndedQueue

{

*protected:*

    Node<T> \*front, \*rear;

    DoublyLinkedList<T> list;

*public:*

    DoublyEndedQueue()

    {

        this->front = this->list.head;

        this->rear = this->list.tail;

    }

*void* enqueueFront(T *ele*)

    {

        this->list.insertFront(*ele*);

        this->front = this->list.head;

        this->rear = this->list.tail;

    }

*void* enqueueRear(T *ele*)

    {

        this->list.insertBack(*ele*);

        this->front = this->list.head;

        this->rear = this->list.tail;

    }

    T dequeueFront()

    {

        if (this->isEmpty())

        {

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

            return (T)(NULL);

        }

        T temp = this->front->info;

        this->list.deleteFront();

        this->front = this->list.head;

        this->rear = this->list.tail;

        return temp;

    }

    T dequeueRear()

    {

        if (this->isEmpty())

        {

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

            return (T)(NULL);

        }

        T temp = this->rear->info;

        this->list.deleteBack();

        this->front = this->list.head;

        this->rear = this->list.tail;

        return temp;

    }

    T frontEl()

    {

        if (this->isEmpty())

        {

            cout << "Queue Empty";

            return (T)(NULL);

        }

        return this->front->info;

    }

*bool* isEmpty()

    {

        return this->list.isEmpty();

    }

*void* clear()

    {

        while (!this->isEmpty())

            this->dequeue();

    }

*void* display()

    {

        if (this->isEmpty())

        {

            cout << "Queue Empty";

            return;

        }

        this->list.display();

        return;

    }

};

*int* main(*void*)

{

*int* el, res, choice;

    DoublyEndedQueue<*int*> q;

    do

    {

        cout << "\tDoubly Ended Queue - Deque\n"

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

             << " (1) EnqueueBack (2) DequeueRear\n"

             << " (3) EnqueueFront (4) DequeueFront\n"

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

             << " (0) Exit\n\n";

        cout << "Enter Choice: ";

        cin >> choice;

        switch (choice)

        {

        case 1:

            cout << "\nEnter Element: ";

            cin >> el;

            q.enqueueRear(el);

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

            cout << "Queue: ";

            q.display();

            break;

        case 2:

            res = q.dequeueRear();

            if (res)

            {

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

                cout << "Queue: ";

                q.display();

            }

            break;

        case 3:

            cout << "\nEnter Element: ";

            cin >> el;

            q.enqueueFront(el);

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

            cout << "Queue: ";

            q.display();

            break;

        case 4:

            res = q.dequeueFront();

            if (res)

            {

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

                cout << "Queue: ";

                q.display();

            }

            break;

        case 5:

            cout << "\nFront Element: "

                 << q.frontEl() << endl;

            break;

        case 6:

            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\_\_

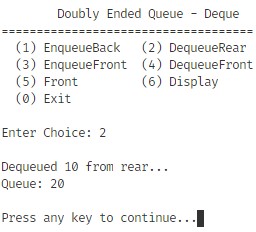
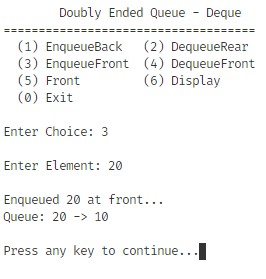
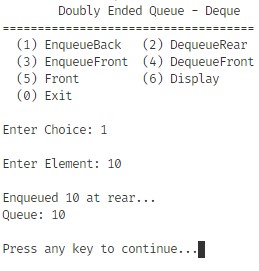
    system("clear");

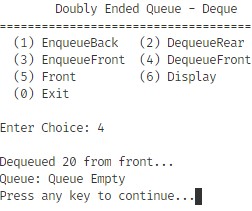
#endif

    return;

}

#### Output





### PRACTICAL 11

#### Objective

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

#### 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;

    }

};

*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;

    }

    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

MENU

================

1. Insertion
2. Searching a node
3. Display its preorder, postorder and inorder traversals. (recursive)
4. Display its preorder, postorder and inorder traversals. (iterative)
5. Display level-by-level traversal. (BFS)
6. Create a mirror image of tree
7. Count the non-leaf, leaf and total number of nodes
8. 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
9. Display height of tree
10. Perform deletion by merging
11. Perform deletion by copying
12. Exit

Enter Choice: 1

Enter Node Data: 10

Press any key to continue...

MENU

================

1. Insertion
2. Searching a node
3. Display its preorder, postorder and inorder traversals. (recursive)
4. Display its preorder, postorder and inorder traversals. (iterative)
5. Display level-by-level traversal. (BFS)
6. Create a mirror image of tree
7. Count the non-leaf, leaf and total number of nodes
8. 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
9. Display height of tree
10. Perform deletion by merging
11. Perform deletion by copying
12. Exit

Enter Choice: 1

Enter Node Data: 5

Press any key to continue...

MENU

================

1. Insertion
2. Searching a node
3. Display its preorder, postorder and inorder traversals. (recursive)
4. Display its preorder, postorder and inorder traversals. (iterative)
5. Display level-by-level traversal. (BFS)
6. Create a mirror image of tree
7. Count the non-leaf, leaf and total number of nodes
8. 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
9. Display height of tree
10. Perform deletion by merging
11. Perform deletion by copying
12. Exit

Enter Choice: 1

Enter Node Data: 14

Press any key to continue...

MENU

================

1. Insertion
2. Searching a node
3. Display its preorder, postorder and inorder traversals. (recursive)
4. Display its preorder, postorder and inorder traversals. (iterative)
5. Display level-by-level traversal. (BFS)
6. Create a mirror image of tree
7. Count the non-leaf, leaf and total number of nodes
8. 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
9. Display height of tree
10. Perform deletion by merging
11. Perform deletion by copying
12. Exit

Enter Choice: 1

Enter Node Data: 0

Press any key to continue...

MENU

================

1. Insertion
2. Searching a node
3. Display its preorder, postorder and inorder traversals. (recursive)
4. Display its preorder, postorder and inorder traversals. (iterative)
5. Display level-by-level traversal. (BFS)
6. Create a mirror image of tree
7. Count the non-leaf, leaf and total number of nodes
8. 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
9. Display height of tree
10. Perform deletion by merging
11. Perform deletion by copying
12. Exit

Enter Choice: 1

Enter Node Data: 6

Press any key to continue...

MENU

================

1. Insertion
2. Searching a node
3. Display its preorder, postorder and inorder traversals. (recursive)
4. Display its preorder, postorder and inorder traversals. (iterative)
5. Display level-by-level traversal. (BFS)
6. Create a mirror image of tree
7. Count the non-leaf, leaf and total number of nodes
8. 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
9. Display height of tree
10. Perform deletion by merging
11. Perform deletion by copying
12. Exit

Enter Choice: 1

Enter Node Data: 10

Press any key to continue...

MENU

================

1. Insertion
2. Searching a node
3. Display its preorder, postorder and inorder traversals. (recursive)
4. Display its preorder, postorder and inorder traversals. (iterative)
5. Display level-by-level traversal. (BFS)
6. Create a mirror image of tree
7. Count the non-leaf, leaf and total number of nodes
8. 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

1. Display height of tree
2. Perform deletion by merging
3. Perform deletion by copying
4. Exit

Enter Choice: 1

Enter Node Data: 14

Press any key to continue...

MENU

================

1. Insertion
2. Searching a node
3. Display its preorder, postorder and inorder traversals. (recursive)
4. Display its preorder, postorder and inorder traversals. (iterative)
5. Display level-by-level traversal. (BFS)
6. Create a mirror image of tree
7. Count the non-leaf, leaf and total number of nodes
8. 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
9. Display height of tree
10. Perform deletion by merging
11. Perform deletion by copying
12. Exit

Enter Choice: 2

Enter Search Data: 14

Search Result: Found

Press any key to continue...

MENU

================

1. Insertion
2. Searching a node
3. Display its preorder, postorder and inorder traversals. (recursive)
4. Display its preorder, postorder and inorder traversals. (iterative)
5. Display level-by-level traversal. (BFS)
6. Create a mirror image of tree
7. Count the non-leaf, leaf and total number of nodes
8. 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
9. Display height of tree
10. Perform deletion by merging
11. Perform deletion by copying
12. Exit

Enter Choice: 2

Enter Search Data: 2

Search Result: Not Found

Press any key to continue...

MENU

================

1. Insertion
2. Searching a node
3. Display its preorder, postorder and inorder traversals. (recursive)
4. Display its preorder, postorder and inorder traversals. (iterative)
5. Display level-by-level traversal. (BFS)
6. Create a mirror image of tree
7. Count the non-leaf, leaf and total number of nodes
8. 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
9. Display height of tree
10. Perform deletion by merging
11. Perform deletion by copying
12. Exit

Enter Choice: 3

In-Order Recursive Traversal: 0 5 6 10 10 14 14

Pre-Order Recursive Traversal: 10 5 0 6 14 10 14

Post-Order Recursive Traversal: 0 6 5 10 14 14 10

Press any key to continue...

MENU

================

1. Insertion
2. Searching a node
3. Display its preorder, postorder and inorder traversals. (recursive)
4. Display its preorder, postorder and inorder traversals. (iterative)
5. Display level-by-level traversal. (BFS)
6. Create a mirror image of tree
7. Count the non-leaf, leaf and total number of nodes
8. 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
9. Display height of tree
10. Perform deletion by merging
11. Perform deletion by copying
12. Exit

Enter Choice: 4

In-Order Iterative Traversal: 0 5 6 10 10 14 14

Pre-Order Iterative Traversal: 10 5 0 6 14 10 14

Post-Order Iterative Traversal: 0 6 5 10 14 14 10

Press any key to continue...

MENU

================

1. Insertion
2. Searching a node
3. Display its preorder, postorder and inorder traversals. (recursive)
4. Display its preorder, postorder and inorder traversals. (iterative)
5. Display level-by-level traversal. (BFS)
6. Create a mirror image of tree
7. Count the non-leaf, leaf and total number of nodes
8. 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
9. Display height of tree
10. Perform deletion by merging
11. Perform deletion by copying

(0) Exit

Enter Choice: 5

Level-by-level Traversal:

10 5 14 0 6 10 14

Press any key to continue...

MENU

================

1. Insertion
2. Searching a node
3. Display its preorder, postorder and inorder traversals. (recursive)
4. Display its preorder, postorder and inorder traversals. (iterative)
5. Display level-by-level traversal. (BFS)
6. Create a mirror image of tree
7. Count the non-leaf, leaf and total number of nodes
8. 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
9. Display height of tree
10. Perform deletion by merging
11. Perform deletion by copying
12. Exit

Enter Choice: 7

Leaf Nodes: 4

Non-Leaf Nodes: 3

Total Nodes: 7

Press any key to continue...

MENU

================

1. Insertion
2. Searching a node
3. Display its preorder, postorder and inorder traversals. (recursive)
4. Display its preorder, postorder and inorder traversals. (iterative)
5. Display level-by-level traversal. (BFS)
6. Create a mirror image of tree
7. Count the non-leaf, leaf and total number of nodes
8. 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
9. Display height of tree
10. Perform deletion by merging
11. Perform deletion by copying

(0) Exit

Enter Choice: 0

Press any key to continue...