



LAB MANUAL

Name : Muhammad Ahmad Mustafa

Registration : 2023-bsai-016

Submitted to : Mam Irsha Qureshi

Degree : BS-AI- SEC (A)

Data Strucutres

DSA stands for Data Structures and Algorithms. It is about organizing and managing data (data structures) and using step-by-step methods (algorithms) to solve problems efficiently. Data structures like arrays, lists, and trees help store and organize data, while algorithms are used to perform tasks like searching or sorting. Together, DSA is essential for writing better and faster computer programs.

Array

In C++, an array is a collection of elements of the same data type stored in contiguous memory locations. Arrays allow you to store multiple values of the same type in a single variable, which makes it easier to manage large amounts of data.

Functions

Insertion at beginning:

Code:

```
void InsertAtBegining()  
{  
    size++;  
    int val;  
    cout<<"Enter value: ";  
    cin>>val;  
    for (int i = size-1; i > 0; i--)  
    {  
        arr[i] = arr[i - 1];  
    }  
    arr[0]=val;  
}
```

```
Array: 1 2 3 4 5 6 7 8  
Enter value: 0  
Array: 0 1 2 3 4 5 6 7 8
```

Insert At Mid:

Code:

```
void InsertAtMid(int val,int index)
{
    size++;
    for(int i=size-1;i>index;i--)
    {
        arr[i] = arr[i - 1];
    }
    arr[index]=val;
}
```

Output:

```
Array: 1 2 3 4 5 6 7 8
Enter Value: 9
Enter Index: 2
Array: 1 2 9 3 4 5 6 7 8
```

Insert at Last:**Code:**

```
void InsertAtLast(int val)
{
    size++;
    arr[size-1]=val;
}
```

Output:

```
Array: 1 2 3 4 5 6 7 8
Enter Value: 9
Array: 1 2 3 4 5 6 7 8 9
```

Search:

```
void search(int val) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == val) {
```

```

        cout<<"value is at Index: "<<i;
        return;}}
    cout<<"Value not found"<<endl;
    return;
}

```

Output:

```

Array: 1 2 3 4 5 6 7 8
Entere value to find: 8
value is at Index: 7

```

Edit:

Code:

```

void Edit(int index,int newval)
{
    arr[index]=newval;
}

```

Output:

```

Array: 1 2 3 4 5 6 7 8
Enter index to Edit: 2
New value: 9
Array: 1 2 9 4 5 6 7 8

```

Update:

```

void update(int val,int newval)
{
    for(int i=0;i<size;i++)
    {
        if(arr[i]==val)
        {
            arr[i]=newval;
        }
    }
}

```

Output:

```
Array: 1 2 3 2 6 2 8 9
Enter value to update: 2
New value: 0
Array: 1 0 3 0 6 0 8 9
```

Stack

A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle, meaning the last element added (pushed) to the stack is the first one to be removed (popped).

Stack implementation and Infix to Postfix:

Code:

```
#include<iostream>
using namespace std;
class Stack
{
    char array[100];
    int topvalue=-1;
    char topelement;
public:
    void push(char a)
    {
        if(topvalue<=98)
        {
            array[++topvalue]=a;

        }
    }
    void pop()
    {
        if(!empty())
        {
            topvalue--;
        }
    }
}
```

```

}
char top()
{
    if(!empty())
    {topelement=array[topvalue];
    return topelement; }
}
bool empty()
{
    if(topvalue==-1)
    {
        return true;
    }
    else
    {
        return false;
    }
}

};
int precedence(char a);
bool isoperator(char a);
string infixtopostfix(string infix);
int main()
{ string infix,postfix;
  cout<<"Enter an Infix Expression: ";
  cin>>infix;
  postfix=infixtopostfix(infix);
  cout<<"Postfix Expression: "<<postfix;
  return 0 ;
}
int precedence(char a)
{
    if(a=='*' | a=='/')
    {
        return 2;
    }
    if(a=='+' | a=='-')

```

```

    {return 1;}
    else
        return -1;
}
bool isoperator(char a)
{
    if(a=='/' || a=='*' || a=='+' || a=='-')
    {
        return true;
    }
    else
        return false;
}
string infixtopostfix(string infix)
{
    Stack s;
    string postfix;
    for(int i=0;i<infix.length();i++)
    {
        if((infix[i]>='a'CCinfix[i]<='z') || (infix[i]>='A'CCinfix[i]<='Z') || (infix[i]>='0'CCinfix[i]<='9'))
        {
            postfix+=infix[i];
        }
        else if(infix[i]=='(')
        {
            s.push(infix[i]);
        }
        else if(infix[i]==')')
        {
            while(s.top()!='('CC!s.empty())
            {
                char temp=s.top();
                postfix+=temp;
                s.pop();
            }
            s.pop();
        }
        else if(isoperator(infix[i]))

```

```

{
    if(s.empty())
    {
        s.push(infix[i]);
    }
    else if(precedence(infix[i])>precedence(s.top()))
    {
        s.push(infix[i]);
    }
    else{
        while((!s.empty())CC(precedence(infix[i])<=precedence(s.top())))
        {
            char temp=s.top();
            postfix+=temp;
            s.pop();
        }
        s.push(infix[i]);
    }
}

}
while(!s.empty())
{
    postfix+=s.top();
    s.pop();
}
return postfix;

}

```

Output:

```

Enter an Infix Expression: a-b*c+d
Postfix Expression: abc*-d+

```

Single link list

A **singly linked list** is a linear data structure consisting of **nodes**, where each node contains:

1. **Data:** The value stored in the node.
2. **Next:** A pointer to the next node in the sequence.

Insert at beginning:

void insertAtHead(int val)

```
{
    Node* n=new Node(val);
    if(head==nullptr)
    {
        head=n;
    }
    else
    {
        n->next=head;
        head=n;
    }
}
```

Output:

```
1->2->3->4->5->
Enter value: 0
0->1->2->3->4->5->
```

Insert before value:

void insertBeforeValue(int key,int val)

```
{
    Node* temp=head;
    if(head->data==key)
    {
        insertAtHead(val);
    }
    else
    { while(temp->next->data!=key)
      {
```

```
temp=temp->next;  
}
```

```
Node* n=new Node(val);  
n->next=temp->next;  
temp->next=n;}}
```

Output:

```
1->2->3->4->5->  
Enter value before which you want to insert new value: 5  
Enter value: 0  
1->2->3->4->0->5->
```

Insert at Last:

```
void insertAtTail(int val)  
{  
    if(head==nullptr)  
    {  
        insertAtHead(val);  
    }  
    else  
    {  
  
        Node* temp=head;  
        while(temp->next!=nullptr)  
        {  
            temp=temp->next;  
        }  
        temp->next=new Node(val);  
    }  
}
```

Output:

```
1->2->3->4->5->
Enter value: 6
1->2->3->4->5->6->
```

Delete first:

```
void deletehead()
{
    if(head==nullptr)
    {
        cout<<"List is empty"<<endl;
        return;
    }
    if(head->next==nullptr)
    {
        delete head;
        head=nullptr;
        return;
    }
    Node*temp=head;
    head=head->next;
    delete temp;
    temp=nullptr;
    return;
}
```

Output:

```
1->2->3->4->5->
2->3->4->5->
```

Deletion mid:

```
void deletevalue(int val)
{
    if(head->data==val)
    {
        Node*temp=head;
        head=head->next;
```

```

    delete temp;
    return;

}
Node* temp=head;
while (temp->next != nullptr CC temp->next->data != val)
{
    temp = temp->next;
}
if(temp->next==nullptr)
{
    cout<<val<<" Not found in the list"<<endl;
    return;
}
Node*deletenode=temp->next;

temp->next=temp->next->next;
delete deletenode;
return;

}

```

Output:

```

1->2->3->4->5->
Enter value: 3
1->2->4->5->

```

Delete Last:

```

void deleteTail()
{
    if(head==nullptr)
    {
        cout<<"List is empty"<<endl;
        return;
    }

    if(head->next==nullptr)
    {

```

```

    delete head;
    head=NULLptr;
    return;
}
Node* temp=head;
while(temp->next->next!=NULLptr)
{
    temp=temp->next;
}
delete temp->next;
temp->next=NULLptr;
return;
}

```

Output:

```

1->2->3->4->5->
1->2->3->4->

```

Search:

```

bool search(int key) {
    Node* temp = head;
    while (temp != NULLptr) {
        if (temp->data == key) {
            return true; // Key found
        }
        temp = temp->next;
    }
    return false; // Key not found
}

```

Output:

```

1->2->3->4->5->NULL
Enter value to search: 3
3 is present in the list.
1->2->3->4->NULL

```

Edit:

```

void edit(int oldVal, int newVal) {
    Node* temp = head;
    while (temp != nullptr) {
        if (temp->data == oldVal) {
            temp->data = newVal;
            cout << "Value " << oldVal << " updated to " << newVal << "." << endl;
            return;
        }
        temp = temp->next;
    }
    cout << "Value " << oldVal << " not found in the list." << endl;
}

```

Output:

```

1->2->3->4->5->NULL
Enter the value to edit: 3
Enter the new value: 8
Value 3 updated to 8.
1->2->8->4->5->NULL

```

Update:

```

void update(int position, int newVal) {
    if (position < 0) {
        cout << "Invalid position." << endl;
        return;
    }

    Node* temp = head;
    int index = 0;

    while (temp != nullptr) {
        if (index == position) {
            temp->data = newVal;
            cout << "Value at position " << position << " updated to " << newVal << "." << endl;
            return; }
        temp = temp->next;
        index++; }
    cout << "Position " << position << " not found in the list." << endl;
}

```

```
}
```

Output:

```
1->2->3->4->5->NULL
Enter position to update (0-based): 2
Enter the new value: 10
Value at position 2 updated to 10.
1->2->10->4->5->NULL
```

Double link list

Functions

Insert at first:

```
void insertAtHead(int val) {
    DoublyNode* newNode = new DoublyNode(val);
    if (head == nullptr) {
        head = newNode;
    } else {
        newNode->next = head;
        head->prev = newNode;
        head = newNode;
    }
}
```

Output:

```
Insertion at Head:
After inserting 1: 1->NULL
After inserting 2: 2<->1->NULL
```

Insert at tail:

```
void insertAtTail(int val) {
    DoublyNode* newNode = new DoublyNode(val);
    if (head == nullptr) {
        head = newNode;
    } else {
        DoublyNode* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
    }
}
```

```

temp->next = newNode;
newNode->prev = temp;
}
}

```

Output:

```

Insertion at Tail:
After inserting 1: 1->NULL
After inserting 2: 1->2->NULL
After inserting 3: 1->2->3->NULL

```

Insert before value:

```

void insertBeforeValue(int key, int val) {
    DoublyNode* temp = head;
    while (temp != nullptr) {
        if (temp->data == key) {
            DoublyNode* newNode = new DoublyNode(val);
            newNode->next = temp;
            newNode->prev = temp->prev;
            if (temp->prev != nullptr) {
                temp->prev->next = newNode;
            } else {
                head = newNode; // If inserting before the head
            }
            temp->prev = newNode;
            return;
        }
        temp = temp->next;
    }
    cout << key << " not found in the list." << endl;
}

```

Output:


```

Insertion Before Value:
After inserting 1: 1->NULL
After inserting 2: 1->2->NULL
After inserting 3: 1->2->3->NULL
After inserting 4: 1->2->3->4->NULL
After inserting 5 before 3: 1->2->5->3->4->NULL

```

Delete Head Function:

```

void deleteHead() {
    if (head == nullptr) {
        cout << "List is empty." << endl;
        return;
    }
    if (head->next == nullptr) {
        delete head;
        head = nullptr;
        return;
    }
    DoublyNode* temp = head;
    head = head->next;
    head->prev = nullptr;
    delete temp;
}

```

Output:

```

Delete Head:
After deleting head: 2->3->4->NULL

```

Delete tail :

```

void deleteTail() {
    if (head == nullptr) {
        cout << "List is empty." << endl;
        return;
    }
    if (head->next == nullptr) {
        delete head;
        head = nullptr;
        return;
    }
}

```

```

}
DoublyNode* temp = head;
while (temp->next != nullptr) {
    temp = temp->next;
}
temp->prev->next = nullptr;
delete temp;
}

```

Output:

```

After inserting 4: 1->2->3->4->NULL

Delete Tail:
After deleting tail: 1->2->3->NULL

```

Delete value:

```

void deleteValue(int val) {
    if (head == nullptr) {
        cout << "List is empty." << endl;
        return;
    }

    if (head->data == val) {
        DoublyNode* temp = head;
        head = head->next;
        if (head != nullptr) {
            head->prev = nullptr;
        }
        delete temp;
        return;
    }

    DoublyNode* temp = head;
    while (temp != nullptr) {
        if (temp->data == val) {
            if (temp->next != nullptr) {
                temp->next->prev = temp->prev;
            }
            delete temp;
            return;
        }
        temp = temp->next;
    }
}

```

```

    }
    if (temp->prev != nullptr) {
        temp->prev->next = temp->next;
    }
    delete temp;
    return;
}
temp = temp->next;
}

cout << val << " not found in the list." << endl;
}

```

Output:

```

After inserting 4: 1->2->3->4->NULL

Delete Value:
After deleting 3: 1->2->4->NULL

```

Search:

```

bool search(int val) {
    DoublyNode* temp = head;
    while (temp != nullptr) {
        if (temp->data == val) {
            return true;
        }
        temp = temp->next;
    }
    return false;
}

```

Output:

```

After inserting 4: 1->2->3->4->NULL

Search for 3: Found

```

Edit:

```

void edit(int oldVal, int newVal) {
    DoublyNode* temp = head;
    while (temp != nullptr) {
        if (temp->data == oldVal) {
            temp->data = newVal;
            return;
        }
        temp = temp->next;
    }
    cout << oldVal << " not found in the list." << endl;
}

```

Output:

```

After inserting 4: 1->2->3->4->NULL

Edit:
After editing 3 to 10: 1->2->10->4->NULL

```

Update:

```

void update(int index, int newVal) {
    DoublyNode* temp = head;
    int count = 0;
    while (temp != nullptr) {
        if (count == index) {
            temp->data = newVal;
            return;
        }
        count++;
        temp = temp->next;
    }
    cout << "Index " << index << " out of range." << endl;
}

```

Output:

```

After inserting 4: 1->2->3->4->NULL

Update:
After updating index 2 to 10: 1->2->10->4->NULL

```

Stack

Definition:

A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle. This means that the last element added to the stack is the first one to be removed. It is commonly used in various algorithms and is particularly useful for problems related to recursive calls, parsing expressions, and backtracking.

In C++, the **stack** is part of the **Standard Template Library (STL)** and is defined in the `<stack>` header. The stack allows operations like:

- **Push:** Adds an element to the top of the stack.
- **Pop:** Removes the top element of the stack.
- **Top:** Accesses the top element without removing it.
- **Empty:** Checks if the stack is empty.
- **Size:** Returns the number of elements in the stack.

Syntax:

To use a stack in C++, include the `<stack>` header. The basic syntax for declaring a stack is:

```
cpp
Copy code
#include <stack>
```

```
stack<data_type> stack_name;
```

Where:

- **data_type** is the type of elements the stack will store (e.g., int, string).
- **stack_name** is the name of the stack.

Basic Operations on Stacks:

1. **Push:** `stack_name.push(value)`
2. **Pop:** `stack_name.pop()`
3. **Top:** `stack_name.top()`
4. **Check if empty:** `stack_name.empty()`
5. **Size:** `stack_name.size()`

Recursive Function Call Simulation

```
#include <iostream>
#include <stack>
using namespace std;
void simulateRecursiveCall(int n) {
    stack<int> s;

    while (n > 0) {
```

```

        s.push(n);
        n--;
    }

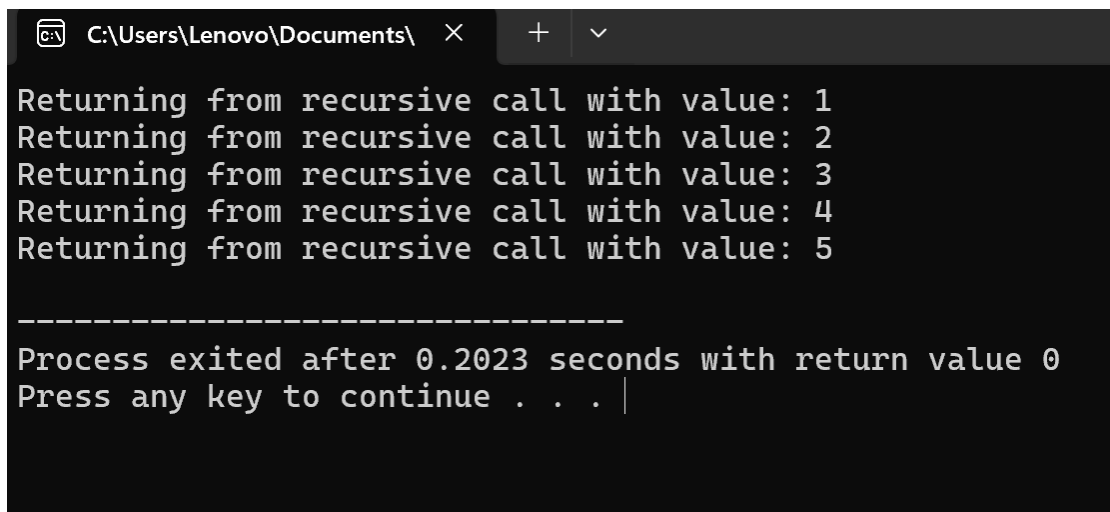
    while (!s.empty()) {
        cout << "Returning from recursive call with value: " << s.top() << endl;
        s.pop();
    }
}

int main() {
    simulateRecursiveCall(5);

    return 0;
}

```

Output:



```

C:\Users\Lenovo\Documents\ × + ▾
Returning from recursive call with value: 1
Returning from recursive call with value: 2
Returning from recursive call with value: 3
Returning from recursive call with value: 4
Returning from recursive call with value: 5

-----
Process exited after 0.2023 seconds with return value 0
Press any key to continue . . . |

```

Queue

Definition:

A **queue** is a linear data structure that follows the **First In, First Out (FIFO)** principle. In a queue, the element that is inserted first is the one that gets removed first. This is like a queue at a movie theater where the first person in line is the first one to get a ticket. It is used in scenarios like task scheduling, handling requests in servers, and simulating real-life queues.

In C++, the **queue** is part of the **Standard Template Library (STL)** and is defined in the `<queue>` header. A queue supports several operations:

- **Enqueue (push)**: Adds an element to the back of the queue.
- **Dequeue (pop)**: Removes an element from the front of the queue.
- **Front**: Retrieves the element at the front without removing it.
- **Back**: Retrieves the element at the back without removing it.
- **Empty**: Checks if the queue is empty.
- **Size**: Returns the number of elements in the queue.

Syntax:

cpp

Copy code

```
#include <queue>
```

```
queue<data_type> queue_name;
```

Where:

- **data_type** is the type of elements the queue will store (e.g., int, string).
- **queue_name** is the name of the queue.

Basic Operations on Queues:

1. **Enqueue**: `queue_name.push(value)`
2. **Dequeue**: `queue_name.pop()`
3. **Front**: `queue_name.front()`
4. **Back**: `queue_name.back()`
5. **Check if empty**: `queue_name.empty()`
6. **Size**: `queue_name.size()`

Print Job Scheduling

```
#include <iostream>
```

```
#include <queue>
```

```
#include <string>
```

```
using namespace std;
```

```
int main() {
```

```
    queue<string> printQueue;
```

```
    // Add print jobs
```

```
    printQueue.push("Document 1");
```

```
    printQueue.push("Document 2");
```

```
    printQueue.push("Document 3");
```

```
    // Print jobs in order
```

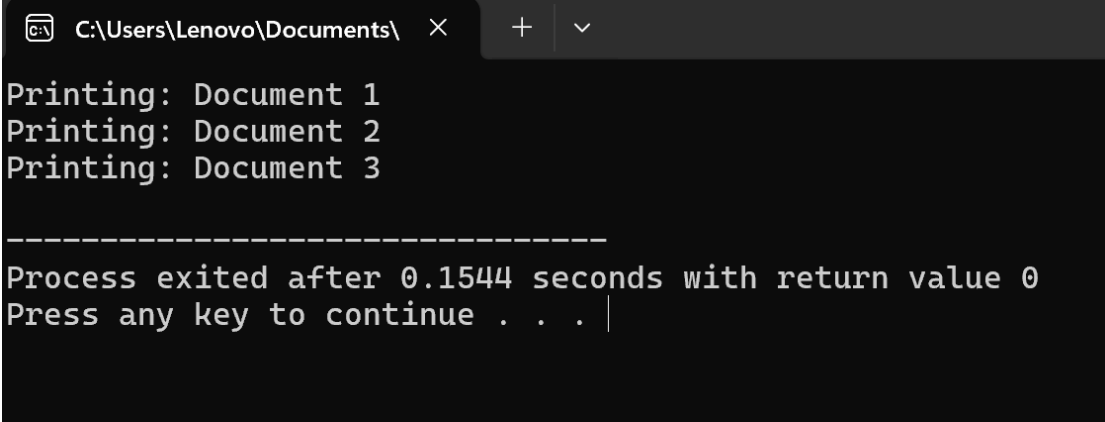
```
    while (!printQueue.empty()) {
```

```
        cout << "Printing: " << printQueue.front() << endl;
```

```
        printQueue.pop();
```

```
}  
  
return 0;  
}
```

Output:



A screenshot of a Windows command prompt window. The title bar shows the file path 'C:\Users\Lenovo\Documents\' and standard window controls. The output of the program is displayed in a monospaced font. It shows three lines of 'Printing: Document' followed by a dashed line separator and a message about the process exiting.

```
C:\Users\Lenovo\Documents\  X  +  v  
Printing: Document 1  
Printing: Document 2  
Printing: Document 3  
  
-----  
Process exited after 0.1544 seconds with return value 0  
Press any key to continue . . . |
```

Linear:

```
#include <iostream>  
using namespace std;
```

```
class LinearQueue {  
private:  
    int front, rear, size;  
    int* queue;
```

```
public:  
    LinearQueue(int s) {  
        size = s;  
        front = -1;  
        rear = -1;  
        queue = new int[size];  
    }
```

```
    ~LinearQueue() {  
        delete[] queue;  
    }
```

```
    bool isFull() {  
        return rear == size - 1;  
    }
```



```

bool isEmpty() {
    return front == -1 || front > rear;
}

void enqueue(int value) {
    if (isFull()) {
        cout << "Queue Overflow!" << endl;
        return;
    }
    if (front == -1) front = 0; // Initialize front if adding the first element
    queue[++rear] = value;
    cout << "Enqueued: " << value << endl;
}

void dequeue() {
    if (isEmpty()) {
        cout << "Queue Underflow!" << endl;
        return;
    }
    cout << "Dequeued: " << queue[front++] << endl;
}

void display() {
    if (isEmpty()) {
        cout << "Queue is Empty!" << endl;
        return;
    }
    cout << "Queue Elements: ";
    for (int i = front; i <= rear; i++) {
        cout << queue[i] << " ";
    }
    cout << endl;
}

};

int main() {
    LinearQueue q(5);

    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.display();

    q.dequeue();
    q.display();
}

```

```
    return 0;
}
```

Output:

```
D:\University\3rd smester\DS. X + v
Enqueued: 10
Enqueued: 20
Enqueued: 30
Queue Elements: 10 20 30
Dequeued: 10
Queue Elements: 20 30

-----
Process exited after 0.1948 seconds with return value 0
Press any key to continue . . .
```

Circular:

```
D:\University\3rd smester\DS. X + v
Enqueued: 10
Enqueued: 20
Enqueued: 30
Queue Elements: 10 20 30
Dequeued: 10
Queue Elements: 20 30

-----
Process exited after 0.222 seconds with return value 0
Press any key to continue . . .
```

Single Link List

(Insertion(Front, mid, last), deletion(Front, mid, last)):

```
#include <iostream>
using namespace std;
```

```
class Node {
public:
    int data;
    Node* next;
```

```
Node(int value) {
    data = value;
    next = nullptr;
```

```
    }  
};
```

```
class SinglyLinkedList {
```

```
private:
```

```
    Node* head;
```

```
public:
```

```
    SinglyLinkedList() {
```

```
        head = nullptr;
```

```
    }
```

```
    // Insert at the front
```

```
    void insertFront(int value) {
```

```
        Node* newNode = new Node(value);
```

```
        newNode->next = head;
```

```
        head = newNode;
```

```
        cout << "Inserted at Front: " << value << endl;
```

```
    }
```

```
    // Insert at the end
```

```
    void insertLast(int value) {
```

```
        Node* newNode = new Node(value);
```

```
        if (head == nullptr) {
```

```
            head = newNode;
```

```
        } else {
```

```
            Node* temp = head;
```

```
            while (temp->next != nullptr) {
```

```
                temp = temp->next;
```

```
            }
```

```
            temp->next = newNode;
```

```
        }
```

```
        cout << "Inserted at Last: " << value << endl;
```

```
    }
```

```
    // Insert in the middle
```

```
    void insertMid(int value, int position) {
```

```
        if (position <= 0) {
```

```
            cout << "Position must be greater than 0!" << endl;
```

```
            return;
```

```
        }
```

```
        Node* newNode = new Node(value);
```

```
        if (position == 1) { // Insert at the head
```

```
            newNode->next = head;
```

```
            head = newNode;
```

```
        } else {
```

```
            Node* temp = head;
```

```

    for (int i = 1; temp != nullptr && i < position - 1; i++) {
        temp = temp->next;
    }
    if (temp == nullptr) {
        cout << "Position out of range!" << endl;
        return;
    }
    newNode->next = temp->next;
    temp->next = newNode;
}
cout << "Inserted at Position " << position << ": " << value << endl;
}

```

```

// Delete from the front
void deleteFront() {
    if (head == nullptr) {
        cout << "List is empty, nothing to delete!" << endl;
        return;
    }
    Node* temp = head;
    head = head->next;
    cout << "Deleted from Front: " << temp->data << endl;
    delete temp;
}

```

```

// Delete from the end
void deleteLast() {
    if (head == nullptr) {
        cout << "List is empty, nothing to delete!" << endl;
        return;
    }
    if (head->next == nullptr) { // Only one element
        cout << "Deleted from Last: " << head->data << endl;
        delete head;
        head = nullptr;
        return;
    }
    Node* temp = head;
    while (temp->next->next != nullptr) {
        temp = temp->next;
    }
    cout << "Deleted from Last: " << temp->next->data << endl;
    delete temp->next;
    temp->next = nullptr;
}

```

```

// Delete from the middle

```

```

void deleteMid(int position) {
    if (head == nullptr) {
        cout << "List is empty, nothing to delete!" << endl;
        return;
    }
    if (position <= 0) {
        cout << "Position must be greater than 0!" << endl;
        return;
    }
    if (position == 1) { // Delete the head
        Node* temp = head;
        head = head->next;
        cout << "Deleted from Position " << position << ": " << temp->data << endl;
        delete temp;
        return;
    }
    Node* temp = head;
    for (int i = 1; temp != nullptr && i < position - 1; i++) {
        temp = temp->next;
    }
    if (temp == nullptr || temp->next == nullptr) {
        cout << "Position out of range!" << endl;
        return;
    }
    Node* toDelete = temp->next;
    temp->next = toDelete->next;
    cout << "Deleted from Position " << position << ": " << toDelete->data << endl;
    delete toDelete;
}

// Display the linked list
void display() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
    Node* temp = head;
    cout << "Linked List: ";
    while (temp != nullptr) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL" << endl;
}

};

int main() {

```

```

SinglyLinkedList list;

list.insertFront(10);
list.insertFront(20);
list.display();

list.insertLast(30);
list.insertLast(40);
list.display();

list.insertMid(25, 2);
list.display();

list.deleteFront();
list.display();

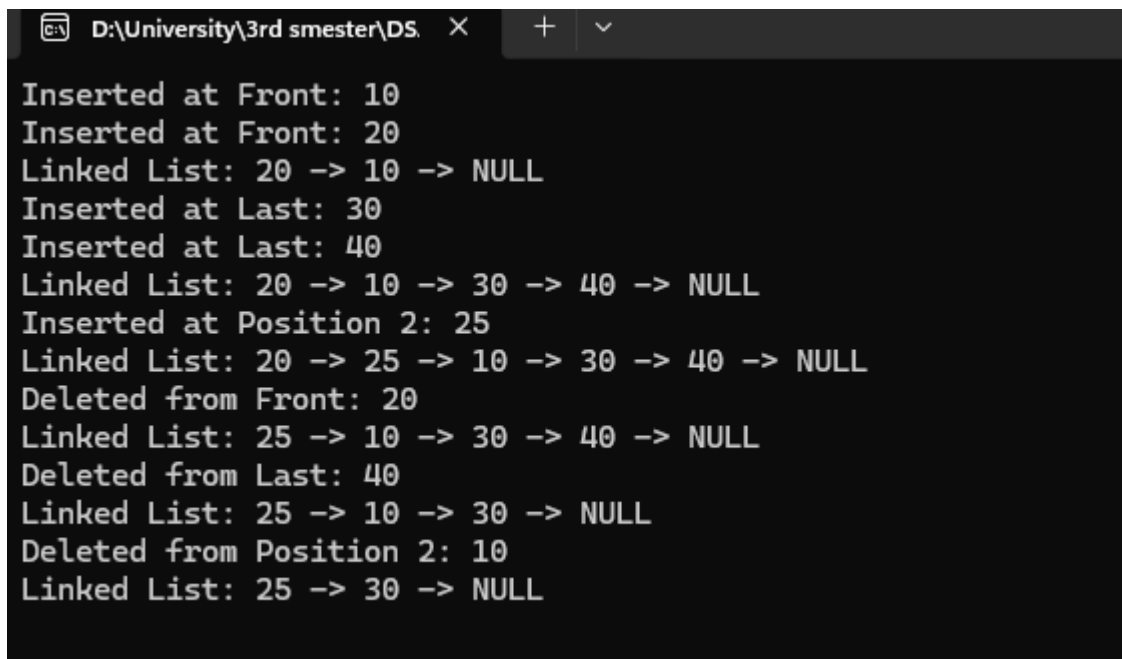
list.deleteLast();
list.display();

list.deleteMid(2);
list.display();

return 0;
}

```

Output:



```

D:\University\3rd smester\DS. x
Inserted at Front: 10
Inserted at Front: 20
Linked List: 20 -> 10 -> NULL
Inserted at Last: 30
Inserted at Last: 40
Linked List: 20 -> 10 -> 30 -> 40 -> NULL
Inserted at Position 2: 25
Linked List: 20 -> 25 -> 10 -> 30 -> 40 -> NULL
Deleted from Front: 20
Linked List: 25 -> 10 -> 30 -> 40 -> NULL
Deleted from Last: 40
Linked List: 25 -> 10 -> 30 -> NULL
Deleted from Position 2: 10
Linked List: 25 -> 30 -> NULL

```

Double Link List

Double Link list (Insertion(Front, mid, last), deletion(Front, mid, last))

```
#include <iostream>
using namespace std;

class Node {
public:
    int data;
    Node* prev;
    Node* next;

    Node(int value) {
        data = value;
        prev = nullptr;
        next = nullptr;
    }
};

class DoublyLinkedList {
private:
    Node* head;
    Node* tail;

public:
    DoublyLinkedList() {
        head = nullptr;
        tail = nullptr;
    }

    // Insert at the front
    void insertFront(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            head = tail = newNode;
        } else {
            newNode->next = head;
            head->prev = newNode;
            head = newNode;
        }
        cout << "Inserted at Front: " << value << endl;
    }

    // Insert at the end
```

```

void insertLast(int value) {
    Node* newNode = new Node(value);
    if (tail == nullptr) {
        head = tail = newNode;
    } else {
        tail->next = newNode;
        newNode->prev = tail;
        tail = newNode;
    }
    cout << "Inserted at Last: " << value << endl;
}

// Insert at the middle
void insertMid(int value, int position) {
    if (position <= 0) {
        cout << "Position must be greater than 0!" << endl;
        return;
    }
    Node* newNode = new Node(value);
    if (position == 1) { // Insert at the head
        insertFront(value);
        return;
    }
    Node* temp = head;
    for (int i = 1; temp != nullptr && i < position - 1; i++) {
        temp = temp->next;
    }
    if (temp == nullptr) {
        cout << "Position out of range!" << endl;
        return;
    }
    newNode->next = temp->next;
    if (temp->next != nullptr) {
        temp->next->prev = newNode;
    }
    newNode->prev = temp;
    temp->next = newNode;
    cout << "Inserted at Position " << position << ": " << value << endl;
}

// Delete from the front
void deleteFront() {
    if (head == nullptr) {
        cout << "List is empty, nothing to delete!" << endl;
        return;
    }
    Node* temp = head;

```



```

head = head->next;
if (head != nullptr) {
    head->prev = nullptr;
} else {
    tail = nullptr;
}
cout << "Deleted from Front: " << temp->data << endl;
delete temp;
}

```

```

// Delete from the end
void deleteLast() {
    if (tail == nullptr) {
        cout << "List is empty, nothing to delete!" << endl;
        return;
    }
    Node* temp = tail;
    tail = tail->prev;
    if (tail != nullptr) {
        tail->next = nullptr;
    } else {
        head = nullptr;
    }
    cout << "Deleted from Last: " << temp->data << endl;
    delete temp;
}

```

```

// Delete from the middle
void deleteMid(int position) {
    if (head == nullptr) {
        cout << "List is empty, nothing to delete!" << endl;
        return;
    }
    if (position <= 0) {
        cout << "Position must be greater than 0!" << endl;
        return;
    }
    if (position == 1) { // Delete the head
        deleteFront();
        return;
    }
    Node* temp = head;
    for (int i = 1; temp != nullptr && i < position; i++) {
        temp = temp->next;
    }
    if (temp == nullptr) {
        cout << "Position out of range!" << endl;
    }
}

```

```

        return;
    }
    if (temp->next != nullptr) {
        temp->next->prev = temp->prev;
    }
    if (temp->prev != nullptr) {
        temp->prev->next = temp->next;
    }
    cout << "Deleted from Position " << position << ": " << temp->data << endl;
    delete temp;
}

// Display the linked list
void display() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
    Node* temp = head;
    cout << "Doubly Linked List: ";
    while (temp != nullptr) {
        cout << temp->data << " <-> ";
        temp = temp->next;
    }
    cout << "NULL" << endl;
}

};

int main() {
    DoublyLinkedList list;

    list.insertFront(10);
    list.insertFront(20);
    list.display();

    list.insertLast(30);
    list.insertLast(40);
    list.display();

    list.insertMid(25, 2);
    list.display();

    list.deleteFront();
    list.display();

    list.deleteLast();
    list.display();
}

```

```

list.deleteMid(2);
list.display();

return 0;
}

```

Output:

```

Inserted at Front: 10
Inserted at Front: 20
Doubly Linked List: 20 <--> 10 <--> NULL
Inserted at Last: 30
Inserted at Last: 40
Doubly Linked List: 20 <--> 10 <--> 30 <--> 40 <--> NULL
Inserted at Position 2: 25
Doubly Linked List: 20 <--> 25 <--> 10 <--> 30 <--> 40 <--> NULL
Deleted from Front: 20
Doubly Linked List: 25 <--> 10 <--> 30 <--> 40 <--> NULL
Deleted from Last: 40
Doubly Linked List: 25 <--> 10 <--> 30 <--> NULL
Deleted from Position 2: 10
Doubly Linked List: 25 <--> 30 <--> NULL

```

Circular Link List

Circular Link List (Insertion(Front, mid, last), deletion(Front, mid, last))

```

#include <iostream>
using namespace std;

class Node {
public:
    int data;
    Node* next;

    Node(int value) {

```

```

        data = value;
        next = nullptr;
    }
};

class CircularLinkedList {
private:
    Node* tail; // The last node in the circular list

public:
    CircularLinkedList() {
        tail = nullptr;
    }

    // Insert at the front
    void insertFront(int value) {
        Node* newNode = new Node(value);
        if (tail == nullptr) {
            tail = newNode;
            tail->next = tail;
        } else {
            newNode->next = tail->next;
            tail->next = newNode;
        }
        cout << "Inserted at Front: " << value << endl;
    }

    // Insert at the end
    void insertLast(int value) {
        Node* newNode = new Node(value);
        if (tail == nullptr) {
            tail = newNode;
            tail->next = tail;
        } else {
            newNode->next = tail->next;
            tail->next = newNode;
            tail = newNode;
        }
        cout << "Inserted at Last: " << value << endl;
    }

    // Insert in the middle
    void insertMid(int value, int position) {
        if (position <= 0) {
            cout << "Position must be greater than 0!" << endl;
            return;
        }
    }

```

```

Node* newNode = new Node(value);
if (tail == nullptr || position == 1) { // Insert at the front if the list is empty or position is 1
    insertFront(value);
    return;
}
Node* temp = tail->next;
for (int i = 1; temp != tail && i < position - 1; i++) {
    temp = temp->next;
}
if (temp == tail) { // Position out of range
    cout << "Position out of range! Inserting at the end." << endl;
    insertLast(value);
} else {
    newNode->next = temp->next;
    temp->next = newNode;
    cout << "Inserted at Position " << position << ": " << value << endl;
}
}

```

```

// Delete from the front
void deleteFront() {
    if (tail == nullptr) {
        cout << "List is empty, nothing to delete!" << endl;
        return;
    }
    Node* temp = tail->next;
    if (tail == temp) { // Only one node
        tail = nullptr;
    } else {
        tail->next = temp->next;
    }
    cout << "Deleted from Front: " << temp->data << endl;
    delete temp;
}

```

```

// Delete from the end
void deleteLast() {
    if (tail == nullptr) {
        cout << "List is empty, nothing to delete!" << endl;
        return;
    }
    Node* temp = tail->next;
    if (tail == temp) { // Only one node
        cout << "Deleted from Last: " << temp->data << endl;
        delete temp;
        tail = nullptr;
    } else {

```

```

        while (temp->next != tail) {
            temp = temp->next;
        }
        temp->next = tail->next;
        cout << "Deleted from Last: " << tail->data << endl;
        delete tail;
        tail = temp;
    }
}

// Delete from the middle
void deleteMid(int position) {
    if (tail == nullptr) {
        cout << "List is empty, nothing to delete!" << endl;
        return;
    }
    if (position <= 0) {
        cout << "Position must be greater than 0!" << endl;
        return;
    }
    Node* temp = tail->next;
    if (position == 1) { // Delete the head
        deleteFront();
        return;
    }
    for (int i = 1; temp != tail && i < position - 1; i++) {
        temp = temp->next;
    }
    if (temp->next == tail->next || temp == tail) { // Position out of range
        cout << "Position out of range!" << endl;
        return;
    }
    Node* toDelete = temp->next;
    temp->next = toDelete->next;
    if (toDelete == tail) {
        tail = temp;
    }
    cout << "Deleted from Position " << position << ": " << toDelete->data << endl;
    delete toDelete;
}

// Display the circular linked list
void display() {
    if (tail == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
}

```

```

        Node* temp = tail->next;
        cout << "Circular Linked List: ";
        do {
            cout << temp->data << " -> ";
            temp = temp->next;
        } while (temp != tail->next);
        cout << "(back to head)" << endl;
    }
};

```

```

int main() {
    CircularLinkedList list;

    list.insertFront(10);
    list.insertFront(20);
    list.display();

    list.insertLast(30);
    list.insertLast(40);
    list.display();

    list.insertMid(25, 2);
    list.display();

    list.deleteFront();
    list.display();

    list.deleteLast();
    list.display();

    list.deleteMid(2);
    list.display();

    return 0;
}

```

Output:

```
D:\University\3rd smester\DS. X + v
Inserted at Front: 10
Inserted at Front: 20
Circular Linked List: 20 -> 10 -> (back to head)
Inserted at Last: 30
Inserted at Last: 40
Circular Linked List: 20 -> 10 -> 30 -> 40 -> (back to head)
Inserted at Position 2: 25
Circular Linked List: 20 -> 25 -> 10 -> 30 -> 40 -> (back to head)
Deleted from Front: 20
Circular Linked List: 25 -> 10 -> 30 -> 40 -> (back to head)
Deleted from Last: 40
Circular Linked List: 25 -> 10 -> 30 -> (back to head)
Deleted from Position 2: 10
Circular Linked List: 25 -> 30 -> (back to head)

-----
Process exited after 0.224 seconds with return value 0
```

Binary Search Tree

Binary Search Tree(Insertion, Deletion, Searching, Traversal(In-order, pre-order, post-order))

```
#include <iostream>
using namespace std;
```

```
class Node {
public:
    int data;
    Node* left;
    Node* right;

    Node(int value) {
```



```

        data = value;
        left = nullptr;
        right = nullptr;
    }
};

class BST {
private:
    Node* root;

    // Helper function for insertion
    Node* insert(Node* node, int value) {
        if (node == nullptr) {
            return new Node(value);
        }
        if (value < node->data) {
            node->left = insert(node->left, value);
        } else if (value > node->data) {
            node->right = insert(node->right, value);
        }
        return node;
    }

    // Helper function for searching
    bool search(Node* node, int value) {
        if (node == nullptr) {
            return false;
        }
        if (value == node->data) {
            return true;
        }
        if (value < node->data) {
            return search(node->left, value);
        }
        return search(node->right, value);
    }

    // Helper function to find the minimum node
    Node* findMin(Node* node) {
        while (node && node->left != nullptr) {
            node = node->left;
        }
        return node;
    }

    // Helper function for deletion
    Node* deleteNode(Node* node, int value) {

```

```

if (node == nullptr) {
    return nullptr;
}

if (value < node->data) {
    node->left = deleteNode(node->left, value);
} else if (value > node->data) {
    node->right = deleteNode(node->right, value);
} else {
    // Node with one child or no child
    if (node->left == nullptr) {
        Node* temp = node->right;
        delete node;
        return temp;
    } else if (node->right == nullptr) {
        Node* temp = node->left;
        delete node;
        return temp;
    }

    // Node with two children
    Node* temp = findMin(node->right);
    node->data = temp->data;
    node->right = deleteNode(node->right, temp->data);
}
return node;
}

// Helper function for in-order traversal
void inOrder(Node* node) {
    if (node == nullptr) return;
    inOrder(node->left);
    cout << node->data << " ";
    inOrder(node->right);
}

// Helper function for pre-order traversal
void preOrder(Node* node) {
    if (node == nullptr) return;
    cout << node->data << " ";
    preOrder(node->left);
    preOrder(node->right);
}

// Helper function for post-order traversal
void postOrder(Node* node) {
    if (node == nullptr) return;

```

```
    postOrder(node->left);
    postOrder(node->right);
    cout << node->data << " ";
}
```

public:

```
    BST() {
        root = nullptr;
    }
```

```
// Public method for insertion
void insert(int value) {
    root = insert(root, value);
    cout << "Inserted: " << value << endl;
}
```

```
// Public method for searching
void search(int value) {
    if (search(root, value)) {
        cout << "Found: " << value << endl;
    } else {
        cout << "Not Found: " << value << endl;
    }
}
```

```
// Public method for deletion
void deleteNode(int value) {
    root = deleteNode(root, value);
    cout << "Deleted: " << value << endl;
}
```

```
// Public method for in-order traversal
void inOrder() {
    cout << "In-order Traversal: ";
    inOrder(root);
    cout << endl;
}
```

```
// Public method for pre-order traversal
void preOrder() {
    cout << "Pre-order Traversal: ";
    preOrder(root);
    cout << endl;
}
```

```
// Public method for post-order traversal
void postOrder() {
```

```

        cout << "Post-order Traversal: ";
        postOrder(root);
        cout << endl;
    }
};

int main() {
    BST tree;

    tree.insert(50);
    tree.insert(30);
    tree.insert(70);
    tree.insert(20);
    tree.insert(40);
    tree.insert(60);
    tree.insert(80);

    tree.inOrder();
    tree.preOrder();
    tree.postOrder();

    tree.search(40);
    tree.search(100);

    tree.deleteNode(50);
    tree.inOrder();

    return 0;
}

```

Output:

```

D:\University\3rd smester\DS
Inserted: 50
Inserted: 30
Inserted: 70
Inserted: 20
Inserted: 40
Inserted: 60
Inserted: 80
In-order Traversal: 20 30 40 50 60 70 80
Pre-order Traversal: 50 30 20 40 70 60 80
Post-order Traversal: 20 40 30 60 80 70 50
Found: 40
Not Found: 100
Deleted: 50
In-order Traversal: 20 30 40 60 70 80

-----
Process exited after 0.2279 seconds with return value 0
Press any key to continue . . .

```