**Name:** Eman Akmal

**Roll no:** BSSE-035

**Section:** 3A

**Subject:** DSA

**“LAB 1”**

**Task:**

Create a program that declares an integer variable and a pointer to it. Modify the value of the variable using the pointer and display both the variable and pointer values.

**Code:**

#include <iostream>

using namespace std;

int main(){

int number = 34;

int\* ptr = &number;

cout << " Original Value: " << number << endl;

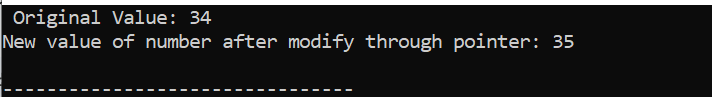
\*ptr = 35;

cout << "New value of number after modify through pointer: " << number << endl;

return 0;

}

**Output:**

****

**HOW?**

* Int number = 34; initializes the variable.
* Int\* ptr = &number creates a pointer that holds the address of the number.
* \*ptr = 35; changes the value through the pointer.
* \*ptr is used to dereference the pointer and get/set the value it points to.

**“LAB 2”**

**Big O Notation (Loops and Arrays)**

**Task:**

Implement a function that finds the maximum value in an array of size n. Determine its time complexity and explain why it is O(n).

**Code:**

#include <iostream>

using namespace std;

int findMax(int arr[], int n){

int maxVal = arr[0];

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

if(arr[i] > maxVal){

maxVal = arr[i];

}

}

return maxVal;

}

int main(){

int arr[] = {45, 87, 92, 87, 23};

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

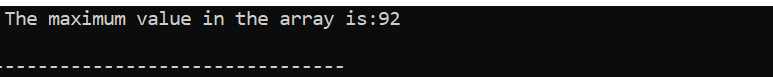
int maxValue = findMax(arr, size);

cout << " The maximum value in the array is:" << maxValue << endl;

return 0;

}

**Output:**

****

**HOW?**

* The function findMax loops through each element once.
* It keeps updating the maxVal if it finds a larger number.
* The loop runs the n times.

### **Time Complexity: O(n)**

* Because the loop runs from index 1 to n-1, checking each element once.
* It performs one comparison per element , so time grows linearly with size of array.
* No nested loops or recursion - just a single pass.

**“LAB 3”**

**Singly Linked List (Insert at End, Insert at Start)**

**Task:**

Implement a singly linked list with functions to insert a node at the start and at the end. Display the list after each insertion.

**Code:**

#include <iostream>

using namespace std;

class Node{

public:

int data;

Node\* next;

Node(int value){

data = value;

next = nullptr;

}

};

class LinkedList {

private:

Node\* head;

public:

LinkedList(){

head = nullptr;

}

void insertAtStart(int value) {

Node\* newNode = new Node(value);

newNode->next = head;

head = newNode;

cout << " Inserted at start: " << value << endl;

display();

}

void insertAtEnd(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 << " Insert at end: " << value << endl;

display();

}

void display(){

Node\* temp = head;

cout << " Current List: ";

while (temp != nullptr){

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

temp = temp-> next;

}

cout << " NULL " << endl << endl;

}

};

int main(){

LinkedList list;

list.insertAtStart(2);

list.insertAtEnd(3);

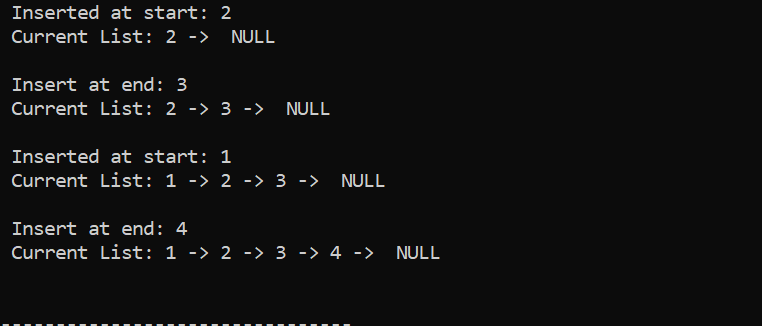
list.insertAtStart(1);

list.insertAtEnd(4);

return 0;

}

**Output:**

****

**HOW?**

* Node class holds one value (data) and a pointer to the next node.
* LinkedList class maintains a pointer to the head of the list.
* insertAtStart adds a node at the beginning by updating the head.
* insertAtEnd traverses to the last node and links a new node at the end.
* Display prints the entire list after each insertion.

**“LAB 4”**

**Singly Linked List (Insert at Specific Location)**

**Task:**

Write a function to insert a node at a specific position in a singly linked list, ensuring valid position handling.

**Code:**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* next;

Node(int value){

data = value;

next = nullptr;

}

};

class LinkedList{

private:

Node\* head;

public:

LinkedList(){

head = nullptr;

}

void insertAtEnd(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;

}

}

void insertAtPosition(int value, int position){

Node\* newNode = new Node(value);

if (position == 1){

newNode->next = head;

head = newNode;

cout << " Inserted " << " "<< value << " at position 1" << endl;

display();

return;

}

Node\* temp = head;

int count = 1;

while(temp != nullptr && count < position - 1){

temp = temp->next;

count++;

}

if (temp == nullptr){

cout << "Invalid position! Insertion failed." << endl;

delete newNode;

return;

}

newNode->next = temp->next;

temp->next = newNode;

cout << "Inserted" << " " << value << " at position " << position << endl;

display();

}

void display(){

Node\* temp = head;

cout << "Current List: ";

while (temp != nullptr) {

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

temp = temp->next;

}

cout << "NULL" << endl << endl;

}

};

int main(){

LinkedList list;

list.insertAtEnd(10);

list.insertAtEnd(20);

list.insertAtEnd(30);

list.display();

list.insertAtPosition(10, 2);

list.insertAtPosition(8, 1);

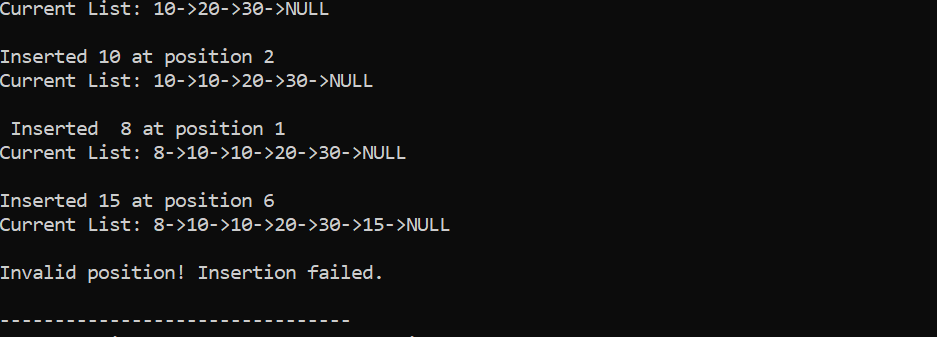
list.insertAtPosition(15, 6);

list.insertAtPosition(80, 10);

return 0;

}

**Output:**

****

**HOW?**

* insertAtPosition(value, position):
* If position == 1, insert at the beginning.
* Else, traverse the to position -1.
* Otherwise, insert the new node at the specified location.

**“LAB 5”**

**Singly Linked List (Display Nodes)**

**Task:**

Implement functions to display the first node, last node, Nth node, and centre node of a singly linked list.

**Code:**

#include <iostream>

using namespace std;

class Node{

public:

int data;

Node\* next;

Node(int value){

data = value;

next = nullptr;

}

};

class LinkedList {

private:

Node\* head;

public:

LinkedList(){

head = nullptr;

}

void insertAtEnd(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;

}

}

void displayList(){

Node\* temp = head;

cout << " Current List: ";

while (temp != nullptr){

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

temp = temp->next;

}

cout << "NULL" << endl << endl;

}

void displayFirst(){

if(head != nullptr)

cout << "First node: " << head->data << endl;

else

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

}

void displayLast(){

if(head == nullptr){

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

return;

}

Node\* temp = head;

while (temp->next != nullptr)

temp = temp->next;

cout << "Last node: " << temp->data << endl;

}

void displayNth(int position){

if (position <= 0){

cout << "Invalid position!" << endl;

return;

}

Node\* temp = head;

int count = 1;

while (temp != nullptr && count < position ){

temp = temp->next;

count++;

}

if(temp != nullptr)

cout << "Node at Position" << position << ":" << temp->data << endl;

else

cout << "Position" << position << " is out of range!" << endl;

}

void displayMiddle(){

if (head == nullptr){

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

return;

}

Node\* slow = head;

Node\* fast = head;

while(fast != nullptr && fast->next != nullptr){

slow = slow->next;

fast = fast->next->next;

}

cout << "Middle node: " << slow->data << endl;

}

};

int main(){

LinkedList list;

list.insertAtEnd(5);

list.insertAtEnd(10);

list.insertAtEnd(15);

list.insertAtEnd(20);

list.insertAtEnd(25);

list.displayList();

list.displayFirst();

list.displayLast();

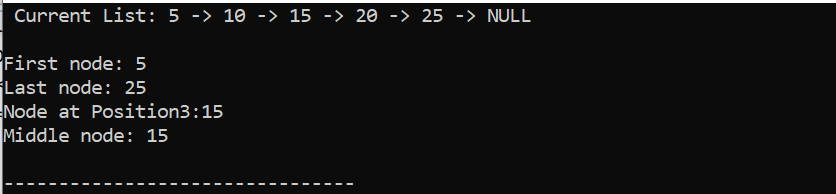
list.displayNth(3);

list.displayMiddle();

return 0;

}

**Output:**

****

**HOW?**

* First Node: Directly from head.
* Last node: Traverse till temp->next == nullptr.
* N-th node: loop till position, check bounce.
* Middle node; use slow and fast pointer method:
* Slow moves one step.
* Fast moves two steps.
* When fast reaches the end, slow is in the middle.

**“LAB 6”**

**Singly Linked List (Delete Nodes)**

**Task:**

Implement functions to delete the first node, last node, Nth node, and centre node of a singly linked list.

**Code:**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* next;

Node(int value){

data = value;

next = nullptr;

}

};

class LinkedList {

private:

Node\* head;

public:

LinkedList() {

head = nullptr;

}

void insertAtEnd(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;

}

}

void displayCurrent(){

Node\* temp = head;

cout << "Current List: ";

while (temp != nullptr){

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

temp = temp->next;

}

cout << "NULL" << endl << endl;

}

void deleteAtFirst(){

if(head == nullptr){

cout << "List is empty. Nothing to delete." << endl;

return;

}

Node\* temp = head;

head = head->next;

cout << "Deleted first node: " << temp->data << endl;

delete temp;

displayCurrent();

}

void deleteAtLast(){

if (head == nullptr) {

cout << "List is empty. Nothing to delete." << endl;

return;

}

if(head->next == nullptr) {

cout << "Deleted ladt Node:" << head->data << endl;

delete head;

head = nullptr;

displayCurrent();

return;

}

Node\* temp = head;

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

temp = temp->next;

cout << "Deleted last node: " << temp->next->data << endl;

delete temp->next;

temp->next = nullptr;

displayCurrent();

}

void deleteNthNode(int position) {

if (head == nullptr || position <= 0){

cout << "Invalid Position or Empty list." << endl;

return;

}

if (position == 1){

deleteAtFirst();

return;

}

Node\* temp = head;

int count = 1;

while (temp != nullptr && count < position - 1){

temp = temp->next;

count++;

}

if(temp == nullptr || temp->next == nullptr){

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

return;

}

Node\* toDelete = temp->next;

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

cout<< "Deleted node at posistion " << position << " : " << toDelete->data << endl;

delete toDelete;

displayCurrent();

}

void deleteAtMiddle(){

if( head == nullptr) {

cout << "List id empty. Nothing to delete. " << endl;

return;

}

if (head->next == nullptr) {

cout << "Deleted middle node (only node):" << head->data << endl;

delete head;

head = nullptr;

displayCurrent();

return;

}

Node\* slow = head;

Node\* fast = head;

Node\* prev = nullptr;

while (fast != nullptr && fast->next != nullptr ) {

fast = fast->next->next;

prev = slow;

slow = slow->next;

}

cout << " Deleted middle node: " << slow->data << endl;

prev->next = slow->next;

delete slow;

displayCurrent();

}

};

int main(){

LinkedList list;

list.insertAtEnd(2);

list.insertAtEnd(4);

list.insertAtEnd(6);

list.insertAtEnd(8);

list.insertAtEnd(10);

list.displayCurrent();

list.deleteAtFirst();

list.deleteAtLast();

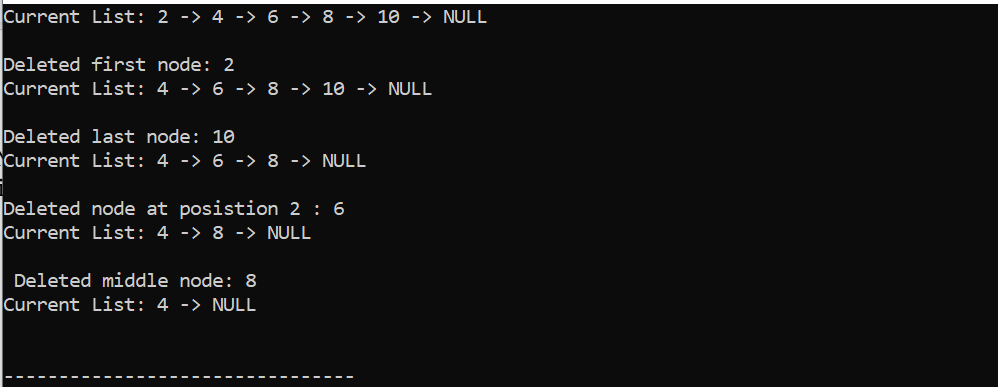
list.deleteNthNode(2);

list.deleteAtMiddle();

return 0 ;

}

**Ouput:**

****

**HOW?**

* deleteFirst : removes the head and moves it to next node.
* deleteLast: traverses to the second-last node and deletes the last one.
* deleteNth: Traverses to (N-1)th node and skips over the nth node.
* deleteMiddle: uses slow and fast pointers to find the middle node and deletes it.

**“LAB 7”**

**Doubly Linked List (Insert & Display Nodes)**

**Task:**

Implement functions to insert node at first, last, Nth location, and centre of a doubly linked list. And display in order and display in reverse order.

**Code:**

#include <iostream>

using namespace std;

class Node{

public:

int data;

Node\* next;

Node\* prev;

Node(int value) {

data = value;

next = nullptr;

prev = nullptr;

}

};

class DoublyLinkedList {

private:

Node\* head;

Node\* tail;

public:

DoublyLinkedList(){

head = nullptr;

tail = nullptr;

}

void insertAtStart(int value) {

Node\* newNode = new Node(value);

if(head == nullptr) {

head = tail = newNode;

}

else{

newNode->next = head;

head->prev = newNode;

head = newNode;

}

cout << " Insert at start: " << " " << value << endl;

}

void insertAtEnd(int value) {

Node\* newNode = new Node(value);

if(tail == nullptr) {

head = tail = newNode;

}

else{

tail->next = newNode;

newNode->prev = tail;

tail = newNode;

}

cout << "Insert at end: " << " " << value << endl;

}

void insertAtPosition(int value, int position) {

if(position <= 0){

cout << "Invalid position!" << endl;

return;

}

if(position == 1){

insertAtStart(value);

return;

}

Node\* newNode = new Node(value);

Node\* temp = head;

int count = 1;

while (temp != nullptr && count < position - 1){

temp = temp->next;

count++;

}

if(temp == nullptr){

cout << "Posiiton out of range. " << endl;

delete newNode;

return;

}

newNode->next = temp->next;

newNode->prev = temp;

if(temp->next != nullptr)

temp->next->prev = newNode;

temp->next = newNode;

if(newNode->next == nullptr)

tail = newNode;

cout << " Inserted " << " " << value << " at position " << " " << position << endl;

}

void insertAtCenter(int value){

int count = 0;

Node\* temp = head;

while (temp != nullptr) {

count++;

temp = temp->next;

}

int midPos = (count / 2) + 1;

insertAtPosition(value, midPos);

cout << "Inserted" << value << "at center position" << midPos << endl;

}

void displayForward(){

Node\* temp = head;

cout << "List (forward):" ;

while(temp != nullptr) {

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

temp = temp->next;

}

cout << "NULL" << endl;

}

void displayBackward(){

Node\* temp = tail;

cout << "List (reverse): ";

while (temp != nullptr) {

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

temp = temp->prev;

}

cout << "NULL" << endl;

}

};

int main (){

DoublyLinkedList list;

list.insertAtStart(22);

list.insertAtEnd(24);

list.insertAtEnd(26);

list.insertAtPosition(23, 2);

list.insertAtCenter(25);

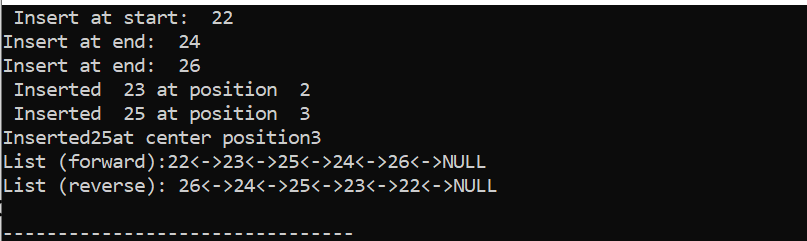
list.displayForward();

list.displayBackward();

return 0;

}

**Output:**

****

**HOW?**

* Doubly Linked List has prev and next pointers.
* Tail is tracked to make reverse display easy.
* insertAtStart() updates head and fixes.
* insertAtEnd() updates tail and fixes links.
* insertAtPosition() handles middle insertions with pointer adjustments.
* insertAtCenter() counts nodes and inserts ata the middle position.
* displayForward() starts from head; displayBackward starts from tail.

**“LAB 8”**

**Merge two LinkedLists**

**Task:**

1. Create 2 Singly LinkedLists and Merge them and display them.

**Code:**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* next;

Node(int val){

data = val;

next = nullptr;

}

};

class SinglyLinkedList {

public:

Node\* head;

SinglyLinkedList(){

head = nullptr;

}

void insertAtEnd(int val){

Node\* newNode = new Node(val);

if (head == nullptr)

head == newNode;

else{

Node\* temp = head;

while (temp->next != nullptr)

temp = temp->next;

temp->next = newNode;

}

}

void display() {

Node\* temp = head;

cout << "Singly Linked List: ";

while (temp != nullptr){

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

temp = temp->next;

}

cout << "Null\n";

}

void merge (SinglyLinkedList& other){

if(head == nullptr){

head = other.head;

}

else{

Node\* temp = head;

while (temp->next != nullptr)

temp = temp->next;

temp->next = other.head;

}

other.head = nullptr;

}

};

int main (){

SinglyLinkedList list1, list2;

list1.insertAtEnd(1);

list1.insertAtEnd(12);

list1.insertAtEnd(3);

list1.insertAtEnd(4);

cout << "Before marge(Singly):\n";

list1.display();

list2.display();

list1.merge(list2);

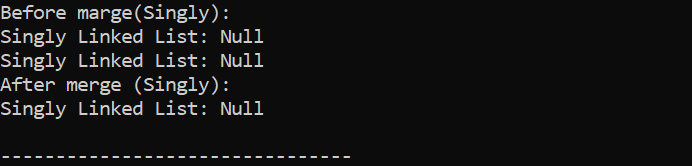
cout << "After merge (Singly): \n";

list1.display();

return 0;

}

**Output:**

****

**HOW?**

Singly List: Traverse to end of list1 and link to list2.

**Task:**

2. Create 2 Double LinkedLists and Merge them and display them.

**Code:**

#include <iostream>

using namespace std;

class DNode {

public:

int data;

DNode\* prev;

DNode\* next;

DNode(int val) {

data = val;

prev = nullptr;

next = nullptr;

}

};

class DoublyLinkedList {

public:

DNode\* head;

DNode\* tail;

DoublyLinkedList(){

head = nullptr;

tail = nullptr;

}

void insertAtEnd(int val) {

DNode\* newNode = new DNode(val);

if (head == nullptr)

head = tail = newNode;

else{

tail->next = newNode;

newNode->prev = tail;

tail = newNode;

}

}

void displayForward (){

DNode\* temp = head;

cout << "Doubly Linked List (forward): ";

while (temp != nullptr){

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

temp = temp->next;

}

cout << " NULL \n ";

}

void merge(DoublyLinkedList& other){

if(head == nullptr) {

head = other.head;

tail = other.tail;

}

else if (other.head != nullptr) {

tail->next = other.head;

other.head->prev = tail;

tail = other.tail;

}

other.head = other.tail = nullptr;

}

};

int main(){

DoublyLinkedList dlist1, dlist2;

dlist1.insertAtEnd(5);

dlist1.insertAtEnd(6);

dlist1.insertAtEnd(7);

dlist1.insertAtEnd(8);

cout << "\n Before merge (Doubly): \n";

dlist1.displayForward();

dlist2.displayForward();

dlist1.merge(dlist2);

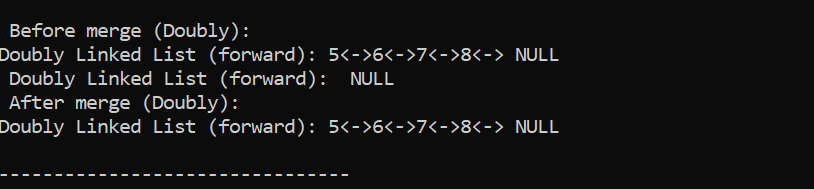
cout << "After merge (Doubly): \n";

dlist1.displayForward();

return 0;

}

**Output:**

****

**HOW?**

Doubly List: Link tail->next of list1 to head of list2, and fix prev.

**“LAB 9”**

**Circular Linked List (Insert & Display Nodes)**

**Task:**

Implement functions to insert node at first, last, Nth location, and centre of a circular linked list. And display in order and display in reverse order.

**Code:**

#include <iostream>

using namespace std;

class Node{

public:

int data;

Node\* next;

Node(int val) {

data = val;

next = nullptr;

}

};

class CircularLinkedList{

private:

Node\* head;

public:

CircularLinkedList(){

head = nullptr;

}

void insertAtStart(int val){

Node\* newNode = new Node(val);

if (head == nullptr) {

head = newNode;

newNode->next = head;

}

else {

Node\* temp = head;

while (temp->next != head)

temp = temp -> next;

newNode->next = head;

temp->next = newNode;

head = newNode;

}

cout << " Inserted at Start: " << val << endl;

}

void insertAtEnd(int val) {

Node\* newNode = new Node(val);

if (head == nullptr){

head = newNode;

newNode->next = head;

}

else{

Node\* temp = head;

while (temp->next != head)

temp = temp->next;

temp->next = newNode;

newNode->next = head;

}

cout << "Inserted at end: " << val << endl;

}

void insertAtPosition(int val, int pos){

if (pos <= 0){

cout << "Invalid position!" << endl;

return;

}

if(pos == 1){

insertAtStart(val);

return;

}

Node\* newNode = new Node(val);

Node\* temp = head;

int count = 1;

while(count < pos - 1 && temp->next != head){

temp = temp ->next;

count++;

}

if (count < pos - 1){

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

return;

}

newNode->next = temp->next;

temp->next = newNode;

cout << "Inserted " << val << "at position " << pos << endl;

}

void insertAtCenter(int val){

int len = 0;

Node\* temp = head;

if (head == nullptr){

insertAtStart(val);

return;

}

do {

len++;

temp = temp->next;

}

while (temp != head);

int mid = (len / 2) + 1;

insertAtPosition(val, mid) ;

cout << "Inserted at center position: " << mid << endl;

}

void didsplayForward(){

if (head == nullptr) {

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

return;

}

Node\* temp = head;

cout << "Circular List (forward): ";

do{

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

temp = temp ->next;

} while (temp != head);

cout << "(back to head)" << endl;

}

void displayReverseHelper (Node\* current, Node\* start){

if(current->next != start)

displayReverseHelper(current->next, start);

cout << current->data << "<-" ;

}

void displayReverse(){

if (head == nullptr){

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

return;

}

cout << "Circular List (reverse): ";

displayReverseHelper(head, head);

cout << "(back to back)" << endl;

}

};

int main (){

CircularLinkedList clist;

clist.insertAtStart(10);

clist.insertAtEnd(20);

clist.insertAtEnd(30);

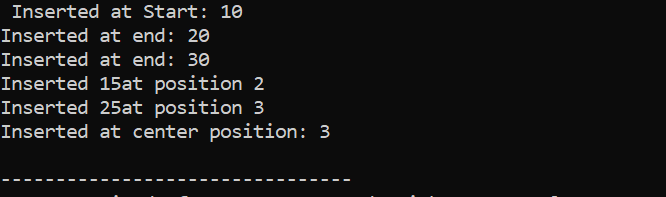
clist.insertAtPosition(15, 2);

clist.insertAtCenter(25);

return 0;

}

**Output:**

****

**HOW?**

* Circular nature is handled by making the last node’s next = head.
* Display reverse is done recursively since we don’t have prev pointers.
* Insertion at position/center works based on counting nodes in a loop.

**“LAB 10”**

**Stack with Linkedlist and Array**

**Task:**

1. With Array; Push, Pop, Display

**Code:**

#include <iostream>

using namespace std;

class StackArray {

private:

int arr[100];

int top;

public:

StackArray(){

top = -1;

}

void push(int val) {

if(top >= 99){

cout << "Stack Overflow! " << endl;

return;

}

arr[++top] = val;

cout << val << " " << "pushed to stack." << endl;

}

void pop(){

if (top == -1){

cout << "Stack Underflow! " << endl;

return;

}

cout << arr[top--] << " popped from stack. " << endl;

}

void display() {

if(top == -1){

cout << " Stack is Empty! " << endl;

return;

}

cout << "Stack ( Top to Bottom): " ;

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

cout << arr[i] << " " ;

cout << endl;

}

};

int main (){

cout << " Stack using Array " << endl;

StackArray s1;

s1.push(10);

s1.push(20);

s1.push(30);

s1.display();

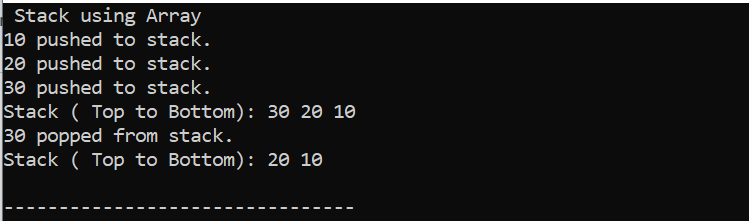
s1.pop();

s1.display();

return 0 ;

}

**Output:**

****

**Task:**

2. With Linkedlist; Push, Pop, Display

**Code:**

#include <iostream>

using namespace std;

class Node{

public:

int data;

Node\* next;

Node(int val){

data = val;

next = nullptr;

}

};

class SLinkedList{

private:

Node\* top;

public:

SLinkedList(){

top = nullptr;

}

void push(int val){

Node\* newNode = new Node(val);

newNode->next = top;

top = newNode;

cout << val << " Pushed to stack." << endl;

}

void pop (){

if (top == nullptr){

cout << " Stack Underflow! " << endl;

return;

}

Node\* temp = top;

cout << top->data << " Popped from stack." << endl;

top = top->next;

}

void display(){

if ( top == nullptr) {

cout << "Stack is empty! " << endl;

return;

}

Node\* temp = top;

cout << " Stack ( Top to Bottom): " ;

while (temp != nullptr){

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

temp = temp -> next;

}

cout << endl;

}

};

int main(){

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

SLinkedList s;

s.push(20);

s.push(25);

s.push(30);

s.display();

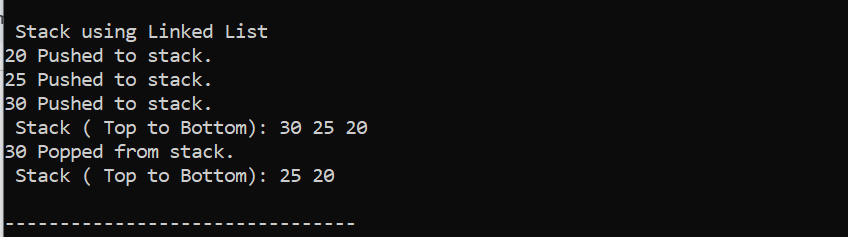
s.pop();

s.display();

return 0;

}

**Output:**

****

**“LAB 11”**

**Queue with Linkedlist and Array**

**Task:**

1. With Array; Enqueue, Dequeue, Display

**Code:**

#include <iostream>

using namespace std;

class ArrayQueue {

int arr[100];

int front, rear;

public:

ArrayQueue(){

front = rear = -1;

}

void enqueue(int val){

if(rear == 99){

cout << " Queue overflow \n ";

return;

}

if(front == -1) front = 0;

arr[++rear] = val;

cout << val << " Enqueue to queue." << endl;

}

void dequeue(){

if(front == -1 || front > rear){

cout << " Queue underflow \n ";

return;

}

cout << arr[front++] << " Dequeue from queue." << endl;

}

void display(){

if(front == -1 || front > rear)

cout << "Queue is empty! \n";

return;

cout << " Queue: (Front to rear): ";

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

cout << arr[i] << " ";

cout << endl;

}

};

int main(){

cout << " Queue with Aarry \n" ;

ArrayQueue q;

q.enqueue(18);

q.enqueue(20);

q.enqueue(22);

q.display();

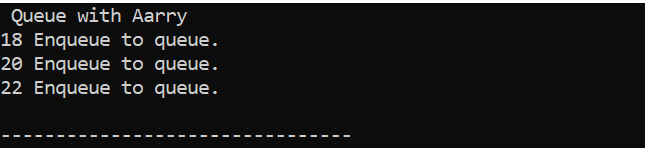
q.display();

q.display();

return 0;

}

**Output:**

****

**Task:**

2. With Linkedlist; Enqueue, Dequeue, Display

**Code:**

#include <iostream>

using namespace std;

class Node{

public:

int data;

Node\* next;

Node(int value){

data = value;

next = nullptr;

}

};

class LinkedListQueue {

Node\* front;

Node\* rear;

public:

LinkedListQueue(){

front = rear = nullptr;

}

void enqueue (int val){

Node\* newNode = new Node(val);

if(rear == nullptr){

front = rear = newNode;

}

else{

rear->next = newNode;

rear = newNode;

}

cout << val << " Enqueue to queue. " << endl;

}

void dequeue(){

if(front == nullptr){

cout << " Queue Undeflow\n ";

return;

}

Node\* temp = front;

cout << front->data << " Dequeued: " << endl;

front = front->next;

if (front == nullptr) rear = nullptr;

delete temp;

}

void display(){

if (front == nullptr) {

cout << " Queue is empty \n ";

return;

}

Node\* temp = front;

cout << "Queue: ";

while(temp != nullptr){

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

temp = temp->next;

}

cout << endl;

}

};

int main (){

cout << " Queue using Linked List ";

LinkedListQueue q1;

q1.enqueue(25);

q1.enqueue(30);

q1.enqueue(35);

q1.display();

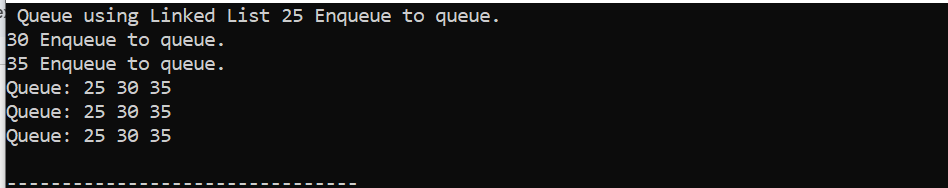
q1.display();

q1.display();

return 0;

}

**Output:**

****

**“LAB 12”**

**BST and AVL**

**Task:**

1. Insert and Traverse for BST

**Code:**

#include <iostream>

using namespace std;

// LAB 12

class BSTNode {

public:

int data;

BSTNode\* left;

BSTNode\* right;

BSTNode(int val){

data = val;

left = right = nullptr;

}

};

class BST{

public:

BSTNode\* root;

BST(){

root = nullptr;

}

BSTNode\* insert(BSTNode\* node, int val){

if(node == nullptr)

return new BSTNode(val);

if(val < node->data)

node->left = insert(node->left, val);

else if(val > node->data)

node->right = insert(node->right, val);

return node;

}

void inorder(BSTNode\* node) {

if (node == nullptr) return;

inorder(node->left);

cout << node->data << " ";

inorder(node->right);

}

};

int main(){

cout << " Binary Search Tree: " << endl;

BST bst;

bst.root = bst.insert(bst.root, 50);

bst.insert(bst.root, 10);

bst.insert(bst.root, 20);

bst.insert(bst.root, 30);

bst.insert(bst.root, 40);

bst.insert(bst.root, 50);

bst.insert(bst.root, 60);

cout << " Inorder Traversal (BST): ";

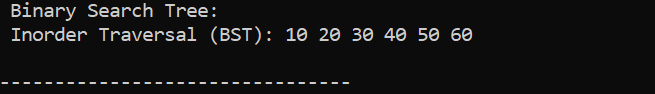
bst.inorder(bst.root);

cout << endl;

return 0;

}

**Output:**

****

**Task:**

2. Insert and Traverse for AVL

**Code:**

#include <iostream>

using namespace std;

class AVLNode {

public:

int data;

AVLNode\* left;

AVLNode\* right;

int height;

AVLNode(int val){

data = val;

left = right = nullptr;

height = 1;

}

};

class AVL {

public:

AVLNode\* root;

AVL(){

root = nullptr;

}

int getHeight(AVLNode\* node){

return node ? node->height : 0;

}

int getBalance(AVLNode\* node){

return node ? getHeight(node->left) - getHeight(node->right) : 0;

}

AVLNode\* rightRotate(AVLNode\* y){

AVLNode\* x = y->left;

AVLNode\* T2 = x->right;

x->right = y;

y->left = T2;

y->height = 1 + max(getHeight(y->left), getHeight(y->right));

x->height = 1 + max(getHeight(x->left), getHeight(x->right));

return x;

}

AVLNode\* leftRotate(AVLNode\* x){

AVLNode\* y = x->right;

AVLNode\* T2 = y->left;

y->left = x;

x->right = T2;

x->height = 1 + max(getHeight(x->left), getHeight(x->right));

y->height = 1 + max(getHeight(y->left), getHeight(y->right));

return y;

}

AVLNode\* insert(AVLNode\* node, int val){

if (node == nullptr)

return new AVLNode(val);

if(val < node->data)

node->left = insert(node->left, val);

else if (val > node->data)

node->right = insert(node->right, val);

else

return node;

node->height = 1 + max(getHeight(node->left), getHeight(node->right));

int balance = getBalance(node);

if(balance > 1 && val < node->left->data)

return rightRotate(node);

if(balance > -1 && val < node->right->data)

return leftRotate(node);

if(balance > 1 && val > node->left->data){

node->left = leftRotate(node->left);

return rightRotate(node);

}

if(balance < -1 && val < node->right->data){

node->right = rightRotate(node->right);

return leftRotate(node);

}

return node;

}

void inorder (AVLNode\* node){

if (node == nullptr) return;

inorder (node->left);

cout << node->data << " ";

inorder(node->right);

}

};

int main (){

cout << " \n AVL TREE " << endl;

AVL avl;

avl.root = avl.insert(avl.root, 10);

avl.root = avl.insert(avl.root, 15);

avl.root = avl.insert(avl.root, 20);

avl.root = avl.insert(avl.root, 25);

avl.root = avl.insert(avl.root, 30);

avl.root = avl.insert(avl.root, 35);

avl.root = avl.insert(avl.root, 40);

cout << " Inorder Traversal (Avl): ";

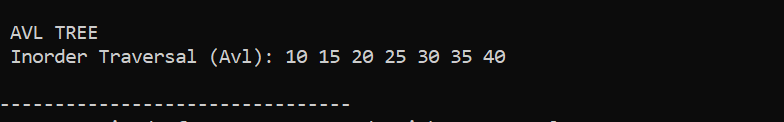
avl.inorder(avl.root);

cout << endl;

return 0;

}

**Output:**

****

**“LAB 13”**

**DFS and BFS**

**Task:**

1. Insert and Traverse for DFS in tree

**Code:**

#include <iostream>

using namespace std;

class Tree\_Node {

public:

int data;

Tree\_Node\* left;

Tree\_Node\* right;

Tree\_Node(int value) {

data = value;

left = right = NULL;

}

};

void DFS\_Tree(Tree\_Node\* root){

if (root == NULL) return;

cout << root->data << " ";

DFS\_Tree(root->left);

DFS\_Tree(root->right);

}

int main (){

Tree\_Node\* root = new Tree\_Node(1);

root->left = new Tree\_Node(2);

root->right = new Tree\_Node(3);

root->left->left = new Tree\_Node(4);

root->left->right = new Tree\_Node(5);

cout << " DFS in Tree: " ;

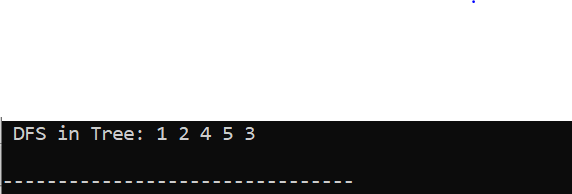
DFS\_Tree(root);

cout << endl;

return 0;

}

**Output:**

****

**Task:**

2. Insert and Traverse for DFS in graph

**Code:**

#include <iostream>

#include <vector>

using namespace std;

void DFSGraph(int node, vector<vector<int>> &adj, vector<bool> &visited){

visited[node] = true;

cout << node << " ";

for(int neighbor : adj[node]){

if(!visited[neighbor])

DFSGraph(neighbor, adj, visited);

}

}

int main(){

int V = 5;

vector<vector<int>> adj(V);

adj[0] = {1,2};

adj[1] = {0,3};

adj[2] = {0,4};

adj[3] = {1};

adj[4] = {2};

vector<bool> visited(V, false);

cout << " DFS in Graph: " ;

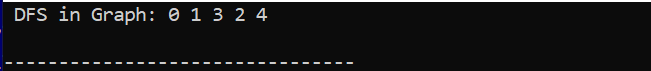
DFSGraph(0, adj, visited);

cout << endl;

return 0;

}

**Output:**

****

**Task:**

3. Insert and Traverse for BFS in tree

**Code:**

#include <iostream>

#include <queue>

using namespace std;

class Tree\_Node{

public:

int data;

Tree\_Node\* left;

Tree\_Node\* right;

Tree\_Node(int value){

data = value;

left = right = NULL;

}

};

void BFS\_Tree(Tree\_Node\* root){

if (root == NULL) return;

queue<Tree\_Node\*> q;

q.push(root);

while(!q.empty()){

Tree\_Node\* current = q.front();

q.pop();

cout << current->data << " ";

if(current->left)q.push(current->right);

if(current->right)q.push(current->right);

}

}

int main(){

Tree\_Node\* root = new Tree\_Node(1);

root->left = new Tree\_Node(2);

root->right = new Tree\_Node(3);

root->left->right = new Tree\_Node(4);

root->left->right = new Tree\_Node(5);

cout << " BFS in Tree: ";

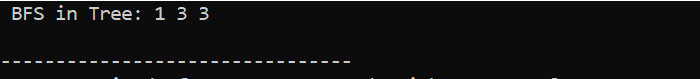
BFS\_Tree(root);

cout << endl;

return 0;

}

**Output:**

****

**Task:**

4. Insert and Traverse for BFS in graph

**Code:**

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

void BFS\_Graph(int start, vector<vector<int>> &adj, vector<bool> &visited){

queue<int> q;

q.push(start);

visited[start] = true;

while(!q.empty()){

int node = q.front();

q.pop();

cout << node << " ";

for (int neighbor : adj[node]) {

if(!visited[neighbor]){

visited[neighbor] = true;

q.push(neighbor);

}

}

}

}

int main (){

int V = 5;

vector<vector<int>> adj(V);

adj[0] = {1, 2};

adj[1] = {1, 3};

adj[2] = {0, 4};

adj[3] = {1};

adj[4] = {2};

vector<bool> visited(V, false);

cout << "BFS IN Graph: ";

return 0;

**Output:**

