**1.STACK**

#include <iostream>

using namespace std;

#define MAX 10

class Stack {

private:

int items[MAX];

int top;

public:

Stack() {

top = -1;

}

bool isFull() {

return top == MAX - 1;

}

bool isEmpty() {

return top == -1;

}

void push(int newItem) {

if (isFull()) {

cout << "Stack is full! Cannot add " << newItem << endl;

} else {

items[++top] = newItem;

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

}

}

void pop() {

if (isEmpty()) {

cout << "Stack is empty! Cannot pop." << endl;

} else {

cout << "Item popped: " << items[top--] << endl;

}

}

void printStack() {

if (isEmpty()) {

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

} else {

cout << "Stack items: ";

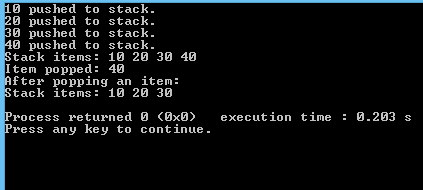
for (int i = 0; i <= top; i++) {

cout << items[i] << " ";

}

cout << endl;

**Output:**



}

}

};

int main() {

Stack s;

s.push(10);

s.push(20);

s.push(30);

s.push(40);

s.printStack();

s.pop();

cout << "After popping an item:" << endl;

s.printStack();

return 0;

}

2 **.Queue**

#include <iostream>

using namespace std;

class MyQueue{

private:

int Queue[100], Front, Rear, MaxSize;

public:

MyQueue(int size) {

MaxSize = size;

Front = -1;

Rear = -1;

}

bool isEmpty(){

if((Front == -1) && (Rear == -1 ) ) {

return true;

}else{

return false;

}

}

bool isFull(){

if(Rear == (MaxSize-1)){

return true;

}else {

return false;

}

}

bool enqueue(int element){

if(isFull()){

cout<<"Queue is Full"<<endl;

return false;

}

else{

if (isEmpty()){

Front = 0;

}

Queue[++Rear] = element;

return true;

}

}

bool dequeue(){

if(isEmpty()){

cout<<"Queue is Empty"<<endl;

return false;

}

else if(Front == Rear){

Front = Rear = -1;

cout<<"Removed Successfully"<<endl;

return true;

}else{

Front ++ ;

cout<<"Removed Successfully"<<endl;

return true;

}

}

void front(){

if(!isEmpty()){

cout<<"Front Element:"<<Queue[Front]<<endl;

}

else{

cout<<"Queue is empty"<<endl;

}

}

void showQueue(){

if(!isEmpty()){

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

for(int i= Front; i<=Rear; i++){

cout<<Queue[i]<<endl;

}

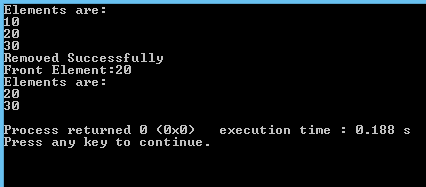
}else{

cout<<"Queue is empty"<<endl;

}

}

**Output:**



};

int main(){

MyQueue q1(100);

q1.enqueue(10);

q1.enqueue(20);

q1.enqueue(30);

q1.showQueue();

q1.dequeue();

q1.front();

q1.showQueue();

}

**3. Dynamic Stack**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

class Stack {

private:

Node\* top;

public:

Stack() {

top = nullptr;

}

void push(int value) {

Node\* newNode = new Node();

newNode->data = value;

newNode->next = top;

top = newNode;

}

int pop() {

if (top == nullptr) {

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

return -1;

}

Node\* temp = top;

int poppedValue = top->data;

top = top->next;

delete temp;

return poppedValue;

}

int peek() {

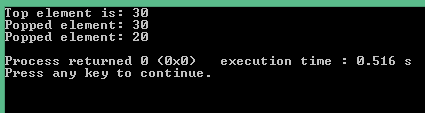
if (top == nullptr) {

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

return -1;

}

**Output:**



return top->data;

}

bool isEmpty() {

return top == nullptr;

}

};

int main() {

Stack stack;

stack.push(10);

stack.push(20);

stack.push(30);

cout << "Top element is: " << stack.peek() << endl;

cout << "Popped element: " << stack.pop() << endl;

cout << "Popped element: " << stack.pop() << endl;

if (stack.isEmpty()) {

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

}

return 0;

}

**4.Circular Queue**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* next;

Node(int value) {

data = value;

next = NULL;

}

};

class Stack {

private:

Node\* top;

public:

Stack() {

top = NULL;

}

void push(int value) {

Node\* newNode = new Node(value);

newNode->next = top;

top = newNode;

}

void pop() {

if (top == NULL) {

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

return;

}

Node\* temp = top;

top = top->next;

delete temp;

}

int peek() {

if (top == NULL) {

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

return -1;

}

return top->data;

}

void display() {

Node\* temp = top;

if (top == NULL) {

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

return;

}

while (temp != NULL) {

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

temp = temp->next;

}

cout << endl;

}

};

int main() {

Stack stack;

int choice, value;

while (true) {

cout << "\nMenu:\n";

cout << "1. Push\n";

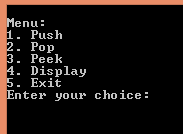
cout << "2. Pop\n";

cout << "3. Peek\n";

cout << "4. Display\n";

cout << "5. Exit\n";

**Output:**



cout << "Enter your choice: ";

cin >> choice;

if (choice == 1) {

cout << "Enter value to push: ";

cin >> value;

stack.push(value);

} else if (choice == 2) {

stack.pop();

} else if (choice == 3) {

int topValue = stack.peek();

if (topValue != -1) {

cout << "Top element: " << topValue << endl;

}

} else if (choice == 4) {

stack.display();

} else if (choice == 5) {

break;

} else {

cout << "Invalid choice. Try again.\n";

}

}

return 0;

}

**5. Singly Linked List**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* next;

Node(int value) {

data = value;

next = NULL;

}

};

class SimpleList {

public:

Node\* head;

SimpleList() {

head = NULL;

}

void addStart(int value) {

Node\* newNode = new Node(value);

newNode->next = head;

head = newNode;

}

void addEnd(int value) {

Node\* newNode = new Node(value);

if (head == NULL) {

head = newNode;

} else {

Node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

}

void show() {

Node\* temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

cout << endl;

}

};

int main() {

SimpleList list;

int choice, value;

while (true) {

cout << "\nMenu:\n";

cout << "1. Add at start\n";

cout << "2. Add at end\n";

cout << "3. Show list\n";

cout << "4. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

if (choice == 1) {

cout << "Enter value to add at start: ";

cin >> value;

list.addStart(value);

}

else if (choice == 2) {

cout << "Enter value to add at end: ";

cin >> value;

list.addEnd(value);

}

else if (choice == 3) {

cout << "The list is: ";

list.show();

}

else if (choice == 4) {

cout << "Exiting program. Goodbye!\n";

break;

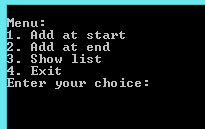
} else {

cout << "Invalid choice. Please try again.\n";

}

}

**Output:**



return 0;

}

**6. Doubly Linked List**

#include <iostream>

using namespace std;

class Node{

public:

int data;

Node\* next;

Node\* prev;

Node(int value) {

data = value;

next = NULL;

prev = NULL;

}

};

class DoublyLinkedList {

private:

Node\* head;

public:

DoublyLinkedList() {

head = NULL;

}

void addAtStart(int value) {

Node\* newNode = new Node(value);

if (head == NULL) {

head = newNode;

} else {

newNode->next = head;

head->prev = newNode;

head = newNode;

}

}

void addAtEnd(int value) {

Node\* newNode = new Node(value);

if (head == NULL) {

head = newNode;

} else {

Node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

}

void deleteValue(int value) {

if (head == NULL) return;

if (head->data == value) {

Node\* temp = head;

head = head->next;

if (head) head->prev = NULL;

delete temp;

return;

}

Node\* temp = head;

while (temp != NULL && temp->data != value) {

temp = temp->next;

}

if (temp == NULL) return;

if (temp->next) temp->next->prev = temp->prev;

if (temp->prev) temp->prev->next = temp->next;

delete temp;

}

bool search(int value) {

Node\* temp = head;

while (temp != NULL) {

if (temp->data == value) return true;

temp = temp->next;

}

return false;

}

void displayForward() {

Node\* temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

cout << endl;

}

void displayBackward() {

if (head == NULL) return;

Node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

while (temp != NULL) {

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

temp = temp->prev;

}

cout << endl;

}

};

int main() {

DoublyLinkedList list;

int choice, value;

while (true) {

cout << "\nMenu:\n";

cout << "1. Add at start\n";

cout << "2. Add at end\n";

cout << "3. Delete a value\n";

cout << "4. Search for a value\n";

cout << "5. Display forward\n";

cout << "6. Display backward\n";

cout << "7. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

if (choice == 1) {

cout << "Enter value to add at start: ";

cin >> value;

list.addAtStart(value);

} else if (choice == 2) {

cout << "Enter value to add at end: ";

cin >> value;

list.addAtEnd(value);

} else if (choice == 3) {

cout << "Enter value to delete: ";

cin >> value;

list.deleteValue(value);

} else if (choice == 4) {

cout << "Enter value to search: ";

cin >> value;

if (list.search(value)) {

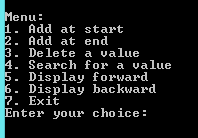
cout << "Value found in the list.\n";

} else {

cout << "Value not found in the list.\n";

}

**Output:**



} else if (choice == 5) {

list.displayForward();

} else if (choice == 6) {

list.displayBackward();

} else if (choice == 7) {

break;

} else {

cout << "Invalid choice. Try again.\n";

}

}

return 0;

}

**7. APPLY SINGLY LINKED LIST TO SOLVE A SOTRING ALGORITHM**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

void appendNode(Node\*& head, int data) {

Node\* newNode = new Node();

newNode->data = data;

newNode->next = nullptr;

if (head == nullptr) {

head = newNode;

} else {

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

}

}

void sortList(Node\* head) {

if (head == nullptr) return;

for (Node\* i = head; i != nullptr; i = i->next) {

for (Node\* j = i->next; j != nullptr; j = j->next) {

if (i->data > j->data) {

int temp = i->data;

i->data = j->data;

j->data = temp;

}

}

}

}

void printList(Node\* head) {

Node\* temp = head;

while (temp != nullptr) {

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

temp = temp->next;

}

cout << endl;

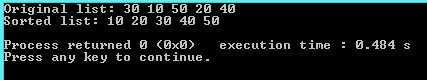
}

int main() {

Node\* head = nullptr;

appendNode(head, 30);

appendNode(head, 10);



appendNode(head, 50);

appendNode(head, 20);

appendNode(head, 40);

cout << "Original list: ";

printList(head);

sortList(head);

cout << "Sorted list: ";

printList(head);

return 0;

}

**8. BST BASIC OPERATION**

#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 {

public:

Node\* root;

BST() {

root = nullptr;

}

void insert(int value) {

Node\* newNode = new Node(value);

if (root == nullptr) {

root = newNode;

} else {

Node\* current = root;

while (true) {

if (value < current->data) {

if (current->left == nullptr) {

current->left = newNode;

break;

}

current = current->left;

} else {

if (current->right == nullptr) {

current->right = newNode;

break;

}

current = current->right;

}

}

}

}

void printInOrder(Node\* node) {

if (node != nullptr) {

printInOrder(node->left);

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

printInOrder(node->right);

}

}

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 {

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\* temp = findMin(node->right);

node->data = temp->data;

node->right = deleteNode(node->right, temp->data);

}

return node;

}

Node\* findMin(Node\* node) {

while (node->left != nullptr) {

node = node->left;

}

return node;

}

bool search(Node\* node, int value) {

if (node == nullptr) {

return false;

}

if (node->data == value) {

return true;

}

if (value < node->data) {

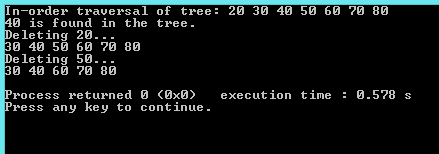
return search(node->left, value);

} else {

return search(node->right, value);

}

}



};

int main() {

BST tree;

tree.insert(50);

tree.insert(30);

tree.insert(20);

tree.insert(40);

tree.insert(70);

tree.insert(60);

tree.insert(80);

cout << "In-order traversal of tree: ";

tree.printInOrder(tree.root);

cout << endl;

int searchValue = 40;

if (tree.search(tree.root, searchValue)) {

cout << searchValue << " is found in the tree." << endl;

} else {

cout << searchValue << " is not found in the tree." << endl;

}

cout << "Deleting 20..." << endl;

tree.root = tree.deleteNode(tree.root, 20);

tree.printInOrder(tree.root);

cout << endl;

cout << "Deleting 50..." << endl;

tree.root = tree.deleteNode(tree.root, 50);

tree.printInOrder(tree.root);

cout << endl;

return 0;

}