**Data Structures and Algorithms**

**Sorting Techniques:**

**Bubble Sort**

#include <iostream>

using namespace std;

int main() {

int a[20], i, j, n, temp;

cout << "\n Enter the size of array: ";

cin >> n;

cout << "\n Enter the array elements: ";

for (i = 0; i < n; i++)

cin >> a[i];

cout << "\n Array elements before sorting: ";

for (i = 0; i < n; i++)

cout << a[i] << "\t";

// Bubble Sort

for (i = 0; i < n - 1; i++) {

for (j = 0; j < n - i - 1; j++) {

if (a[j] > a[j + 1]) {

temp = a[j];

a[j] = a[j + 1];

a[j + 1] = temp;

}

}

}

cout << "\n Elements after Bubble sort: ";

for (i = 0; i < n; i++)

cout << a[i] << "\t";

return 0;

}

**Insertion Sort**

#include <iostream>

using namespace std;

int main() {

int arr[20], n, i, j, key;

cout << "Enter the size of array: ";

cin >> n;

cout << "Enter the array elements: ";

for (i = 0; i < n; i++)

cin >> arr[i];

// Insertion Sort

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

key = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

// Print sorted array

cout << "Sorted array: ";

for (i = 0; i < n; i++)

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

cout << endl;

return 0;

}

**Selection Sort**

#include <iostream>

using namespace std;

int main() {

int arr[20], n, i, j, min\_index, temp;

cout << "Enter the size of array: ";

cin >> n;

cout << "Enter the array elements: ";

for (i = 0; i < n; i++)

cin >> arr[i];

// Selection Sort

for (i = 0; i < n - 1; i++) {

min\_index = i;

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

if (arr[j] < arr[min\_index]) {

min\_index = j;

}

}

if (min\_index != i) {

// Swap arr[i] and arr[min\_index]

temp = arr[i];

arr[i] = arr[min\_index];

arr[min\_index] = temp;

}

}

// Print sorted array

cout << "Sorted array: ";

for (i = 0; i < n; i++)

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

cout << endl;

return 0;

}

**Shell Sort**

#include <iostream>

using namespace std;

// Function to perform Shell Sort

void shellSort(int arr[], int n) {

// Start with a big gap, then reduce the gap

for (int gap = n / 2; gap > 0; gap /= 2) {

// Perform gapped insertion sort for this gap size.

// The first gap elements arr[0..gap-1] are already in gapped order

// keep adding one more element until the entire array is gap sorted

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

// add arr[i] to the elements that have been gap sorted

// save arr[i] in temp and make a hole at position i

int temp = arr[i];

// shift earlier gap-sorted elements up until the correct location for arr[i] is found

int j;

for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {

arr[j] = arr[j - gap];

}

// put temp (the original arr[i]) in its correct location

arr[j] = temp;

}

}

}

int main() {

int arr[20], n;

cout << "Enter the size of array: ";

cin >> n;

cout << "Enter the array elements: ";

for (int i = 0; i < n; i++)

cin >> arr[i];

// Perform Shell Sort

shellSort(arr, n);

// Print sorted array

cout << "Sorted array: ";

for (int i = 0; i < n; i++)

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

cout << endl;

return 0;

}

**Radix Sort**

#include <iostream>

using namespace std;

// A utility function to get the maximum value in arr[]

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

int mx = arr[0];

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

if (arr[i] > mx)

mx = arr[i];

return mx;

}

// A function to do counting sort of arr[] according to the digit represented by exp.

void countSort(int arr[], int n, int exp) {

int output[n]; // output array

int i, count[10] = {0};

// Store count of occurrences in count[]

for (i = 0; i < n; i++)

count[(arr[i] / exp) % 10]++;

// Change count[i] so that count[i] now contains actual position of this digit in output[]

for (i = 1; i < 10; i++)

count[i] += count[i - 1];

// Build the output array

for (i = n - 1; i >= 0; i--) {

output[count[(arr[i] / exp) % 10] - 1] = arr[i];

count[(arr[i] / exp) % 10]--;

}

// Copy the output array to arr[], so that arr[] now contains sorted numbers according to current digit

for (i = 0; i < n; i++)

arr[i] = output[i];

}

// The main function to that sorts arr[] of size n using Radix Sort

void radixSort(int arr[], int n) {

// Find the maximum number to know number of digits

int m = getMax(arr, n);

// Do counting sort for every digit. Note that instead of passing digit number, exp is passed. exp is 10^i where i is current digit number

for (int exp = 1; m / exp > 0; exp \*= 10)

countSort(arr, n, exp);

}

int main() {

int arr[20], n;

cout << "Enter the size of array: ";

cin >> n;

cout << "Enter the array elements: ";

for (int i = 0; i < n; i++)

cin >> arr[i];

// Perform Radix Sort

radixSort(arr, n);

// Print sorted array

cout << "Sorted array: ";

for (int i = 0; i < n; i++)

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

cout << endl;

return 0;

}

**Searching Techniques:**

**Linear Search**

#include <iostream>

using namespace std;

// Function for linear search

int linearSearch(int arr[], int n, int key) {

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

if (arr[i] == key) {

return i; // Return index if key is found

}

}

return -1; // Return -1 if key is not found

}

int main() {

int arr[20], n, key;

cout << "Enter the size of array: ";

cin >> n;

cout << "Enter the array elements: ";

for (int i = 0; i < n; i++)

cin >> arr[i];

cout << "Enter the element to search: ";

cin >> key;

// Perform linear search

int index = linearSearch(arr, n, key);

if (index != -1) {

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

} else {

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

}

return 0;

}

**Binary Search**

#include <iostream>

using namespace std;

// Function for binary search

int binarySearch(int arr[], int left, int right, int key) {

while (left <= right) {

int mid = left + (right - left) / 2;

// Check if key is present at mid

if (arr[mid] == key)

return mid;

// If key is greater, ignore left half

if (arr[mid] < key)

left = mid + 1;

// If key is smaller, ignore right half

else

right = mid - 1;

}

// If key is not found, return -1

return -1;

}

int main() {

int arr[20], n, key;

cout << "Enter the size of array: ";

cin >> n;

cout << "Enter the array elements (in sorted order): ";

for (int i = 0; i < n; i++)

cin >> arr[i];

cout << "Enter the element to search: ";

cin >> key;

// Perform binary search

int index = binarySearch(arr, 0, n - 1, key);

if (index != -1) {

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

} else {

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

}

return 0;

}

**Recursive binary search**

#include <iostream>

using namespace std;

// Recursive binary search function

int binarySearchRecursive(int arr[], int left, int right, int key) {

if (right >= left) {

int mid = left + (right - left) / 2;

// If the element is present at the middle itself

if (arr[mid] == key)

return mid;

// If the element is smaller than mid, then it can only be present in the left subarray

if (arr[mid] > key)

return binarySearchRecursive(arr, left, mid - 1, key);

// Else the element can only be present in the right subarray

return binarySearchRecursive(arr, mid + 1, right, key);

}

// If the element is not present in the array

return -1;

}

int main() {

int arr[20], n, key;

cout << "Enter the size of array: ";

cin >> n;

cout << "Enter the array elements (in sorted order): ";

for (int i = 0; i < n; i++)

cin >> arr[i];

cout << "Enter the element to search: ";

cin >> key;

// Perform recursive binary search

int index = binarySearchRecursive(arr, 0, n - 1, key);

if (index != -1) {

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

} else {

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

}

return 0;

}

**Stack:**

**Basic operations on stack**

#include <iostream>

using namespace std;

#define MAX\_SIZE 100 // Maximum size of the stack

class Stack {

private:

int arr[MAX\_SIZE]; // Array to store stack elements

int top; // Index of the top element in the stack

public:

Stack() { top = -1; } // Constructor to initialize an empty stack

// Function to push an element onto the stack

void push(int value) {

if (top == MAX\_SIZE - 1) {

cout << "Stack Overflow! Cannot push more elements.\n";

return;

}

arr[++top] = value;

cout << "Element " << value << " pushed onto the stack.\n";

}

// Function to pop an element from the stack

void pop() {

if (top == -1) {

cout << "Stack Underflow! Cannot pop from an empty stack.\n";

return;

}

int value = arr[top--];

cout << "Element " << value << " popped from the stack.\n";

}

// Function to peep at the top element of the stack

int peep() {

if (top == -1) {

cout << "Stack is empty. Peep operation cannot be performed.\n";

return -1; // Return a sentinel value

}

return arr[top];

}

// Function to change the top element of the stack

void change(int newValue) {

if (top == -1) {

cout << "Stack is empty. Change operation cannot be performed.\n";

return;

}

arr[top] = newValue;

cout << "Top element of the stack changed to " << newValue << ".\n";

}

// Function to count the number of elements in the stack

int count() {

return top + 1;

}

// Function to display all elements of the stack

void display() {

if (top == -1) {

cout << "Stack is empty. Nothing to display.\n";

return;

}

cout << "Stack elements: ";

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

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

}

cout << endl;

}

};

int main() {

Stack stack;

int choice;

int value;

do {

cout << "\nStack Operations:" << endl;

cout << "1. Push" << endl;

cout << "2. Pop" << endl;

cout << "3. Peep" << endl;

cout << "4. Change" << endl;

cout << "5. Count" << endl;

cout << "6. Display" << endl;

cout << "0. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter the value to push: ";

cin >> value;

stack.push(value);

break;

case 2:

stack.pop();

break;

case 3:

value = stack.peep();

if (value != -1)

cout << "Top element of the stack: " << value << endl;

break;

case 4:

cout << "Enter the new value: ";

cin >> value;

stack.change(value);

break;

case 5:

cout << "Number of elements in the stack: " << stack.count() << endl;

break;

case 6:

stack.display();

break;

case 0:

cout << "Exiting program..." << endl;

break;

default:

cout << "Invalid choice! Please enter a valid option." << endl;

}

} while (choice != 0);

return 0;

}

**Evaluation of postfix expression**

#include <iostream>

#include <stack>

#include <string>

using namespace std;

// Function to check if a character is an operator

bool isOperator(char ch) {

return (ch == '+' || ch == '-' || ch == '\*' || ch == '/');

}

// Function to perform arithmetic operation

int performOperation(int operand1, int operand2, char op) {

switch (op) {

case '+':

return operand1 + operand2;

case '-':

return operand1 - operand2;

case '\*':

return operand1 \* operand2;

case '/':

if (operand2 != 0)

return operand1 / operand2;

else {

cerr << "Error: Division by zero" << endl;

return INT\_MIN; // Indicate an error with the result

}

default:

cerr << "Invalid operator: " << op << endl;

return INT\_MIN; // Indicate an error with the result

}

}

// Function to evaluate postfix expression

int evaluatePostfix(string expression) {

stack<int> operandStack;

// Iterate through each character in the expression

for (size\_t i = 0; i < expression.length(); ++i) {

char c = expression[i];

// If the character is a digit, convert it to integer and push it to the stack

if (isdigit(c)) {

operandStack.push(c - '0');

} else if (isOperator(c)) {

// If it's an operator, pop the top two operands from the stack,

// perform the operation, and push the result back to the stack

int operand2 = operandStack.top();

operandStack.pop();

int operand1 = operandStack.top();

operandStack.pop();

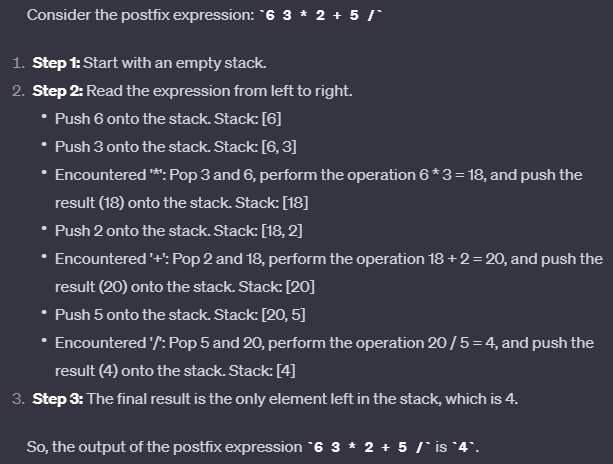
operandStack.push(performOperation(operand1, operand2, c));

}

}

// The result will be the only element left in the stack

return operandStack.top();

}

int main() {

string expression;

cout << "Enter a postfix expression: ";

getline(cin, expression);

int result = evaluatePostfix(expression);

cout << "Result: " << result << endl;

return 0;

}

**Balancing of parenthesis**

#include <iostream>

#include <stack>

#include <string>

using namespace std;

// Function to check if parentheses are balanced

bool areParenthesesBalanced(string expr) {

stack<char> s;

// Iterate through each character in the expression

for (size\_t i = 0; i < expr.length(); ++i) {

char c = expr[i];

if (c == '(' || c == '{' || c == '[') {

// If it's an opening parenthesis, push it onto the stack

s.push(c);

} else if (c == ')' || c == '}' || c == ']') {

// If it's a closing parenthesis, check if stack is empty

// If stack is empty or top of stack doesn't match, return false

if (s.empty() || ((c == ')' && s.top() != '(') || (c == '}' && s.top() != '{') || (c == ']' && s.top() != '['))) {

return false;

}

// Otherwise, pop the top element from stack

s.pop();

}

}

// After iterating through all characters, if stack is empty, return true; otherwise, return false

return s.empty();

}

int main() {

string expression;

cout << "Enter a string containing parentheses: ";

getline(cin, expression);

if (areParenthesesBalanced(expression)) {

cout << "Parentheses are balanced." << endl;

} else {

cout << "Parentheses are not balanced." << endl;

}

return 0;

}

**Queue:**

**Operations on Queue**

#include <iostream>

using namespace std;

#define MAX\_SIZE 100 // Maximum size of the queue

class Queue {

private:

int arr[MAX\_SIZE]; // Array to store queue elements

int front, rear; // Indices for front and rear of the queue

public:

Queue() { front = -1; rear = -1; } // Constructor to initialize an empty queue

// Function to check if the queue is empty

bool isEmpty() {

return front == -1 && rear == -1;

}

// Function to check if the queue is full

bool isFull() {

return (rear + 1) % MAX\_SIZE == front;

}

// Function to enqueue an element into the queue

void enqueue(int value) {

if (isFull()) {

cout << "Queue Overflow! Cannot enqueue more elements.\n";

return;

}

if (isEmpty()) {

front = rear = 0; // If queue is empty, set front and rear to 0

} else {

rear = (rear + 1) % MAX\_SIZE; // Circular increment rear index

}

arr[rear] = value; // Enqueue the element

cout << "Element " << value << " enqueued into the queue.\n";

}

// Function to dequeue an element from the queue

void dequeue() {

if (isEmpty()) {

cout << "Queue Underflow! Cannot dequeue from an empty queue.\n";

return;

}

int value = arr[front]; // Get the element at the front of the queue

if (front == rear) {

// If there was only one element in the queue, reset front and rear

front = rear = -1;

} else {

front = (front + 1) % MAX\_SIZE; // Circular increment front index

}

cout << "Element " << value << " dequeued from the queue.\n";

}

// Function to count the number of elements in the queue

int count() {

if (isEmpty()) {

return 0; // If queue is empty, return count as 0

}

// Calculate count based on front and rear indices

return (rear - front + MAX\_SIZE) % MAX\_SIZE + 1;

}

// Function to display all elements of the queue

void display() {

if (isEmpty()) {

cout << "Queue is empty. Nothing to display.\n";

return;

}

cout << "Queue elements: ";

int i = front;

while (i != rear) {

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

i = (i + 1) % MAX\_SIZE; // Circular increment index

}

cout << arr[rear] << endl;

}

};

int main() {

Queue queue;

int choice;

int value;

do {

cout << "\nQueue Operations:" << endl;

cout << "1. Enqueue" << endl;

cout << "2. Dequeue" << endl;

cout << "3. Count" << endl;

cout << "4. Display" << endl;

cout << "0. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter the value to enqueue: ";

cin >> value;

queue.enqueue(value);

break;

case 2:

queue.dequeue();

break;

case 3:

cout << "Number of elements in the queue: " << queue.count() << endl;

break;

case 4:

queue.display();

break;

case 0:

cout << "Exiting program..." << endl;

break;

default:

cout << "Invalid choice! Please enter a valid option." << endl;

}

} while (choice != 0);

return 0;

}

**Circular Queue**

#include <iostream>

using namespace std;

#define MAX\_SIZE 5 // Maximum size of the circular queue

class CircularQueue {

private:

int arr[MAX\_SIZE]; // Array to store circular queue elements

int front, rear; // Indices for front and rear of the circular queue

public:

CircularQueue() { front = -1; rear = -1; } // Constructor to initialize an empty circular queue

// Function to check if the circular queue is empty

bool isEmpty() {

return front == -1 && rear == -1;

}

// Function to check if the circular queue is full

bool isFull() {

return (rear + 1) % MAX\_SIZE == front;

}

// Function to enqueue an element into the circular queue

void enqueue(int value) {

if (isFull()) {

cout << "Circular Queue Overflow! Cannot enqueue more elements.\n";

return;

}

if (isEmpty()) {

front = rear = 0; // If circular queue is empty, set front and rear to 0

} else {

rear = (rear + 1) % MAX\_SIZE; // Circular increment rear index

}

arr[rear] = value; // Enqueue the element

cout << "Element " << value << " enqueued into the circular queue.\n";

}

// Function to dequeue an element from the circular queue

void dequeue() {

if (isEmpty()) {

cout << "Circular Queue Underflow! Cannot dequeue from an empty circular queue.\n";

return;

}

int value = arr[front]; // Get the element at the front of the circular queue

if (front == rear) {

// If there was only one element in the circular queue, reset front and rear

front = rear = -1;

} else {

front = (front + 1) % MAX\_SIZE; // Circular increment front index

}

cout << "Element " << value << " dequeued from the circular queue.\n";

}

// Function to count the number of elements in the circular queue

int count() {

if (isEmpty()) {

return 0; // If circular queue is empty, return count as 0

}

// Calculate count based on front and rear indices

return (rear - front + MAX\_SIZE) % MAX\_SIZE + 1;

}

// Function to display all elements of the circular queue

void display() {

if (isEmpty()) {

cout << "Circular Queue is empty. Nothing to display.\n";

return;

}

cout << "Circular Queue elements: ";

int i = front;

while (true) {

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

if (i == rear) break; // Exit loop when rear is reached

i = (i + 1) % MAX\_SIZE; // Circular increment index

}

cout << endl;

}

};

int main() {

CircularQueue circularQueue;

int choice;

int value;

do {

cout << "\nCircular Queue Operations:" << endl;

cout << "1. Enqueue" << endl;

cout << "2. Dequeue" << endl;

cout << "3. Count" << endl;

cout << "4. Display" << endl;

cout << "0. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter the value to enqueue: ";

cin >> value;

circularQueue.enqueue(value);

break;

case 2:

circularQueue.dequeue();

break;

case 3:

cout << "Number of elements in the circular queue: " << circularQueue.count() << endl;

break;

case 4:

circularQueue.display();

break;

case 0:

cout << "Exiting program..." << endl;

break;

default:

cout << "Invalid choice! Please enter a valid option." << endl;

}

} while (choice != 0);

return 0;

}

**Priority Queue**

#include <iostream>

using namespace std;

#define MAX\_SIZE 100 // Maximum size of the priority queue

class PriorityQueue {

private:

int arr[MAX\_SIZE]; // Array to store priority queue elements

int priorities[MAX\_SIZE]; // Array to store priorities of elements

int size; // Current size of the priority queue

public:

PriorityQueue() { size = 0; } // Constructor to initialize an empty priority queue

// Function to check if the priority queue is empty

bool isEmpty() {

return size == 0;

}

// Function to check if the priority queue is full

bool isFull() {

return size == MAX\_SIZE;

}

// Function to enqueue an element with its priority into the priority queue

void enqueue(int value, int priority) {

if (isFull()) {

cout << "Priority Queue Overflow! Cannot enqueue more elements.\n";

return;

}

// Find the position to insert based on priority

int position = 0;

while (position < size && priority <= priorities[position]) {

position++;

}

// Shift elements to make space for the new element

for (int i = size; i > position; i--) {

arr[i] = arr[i - 1];

priorities[i] = priorities[i - 1];

}

// Insert the new element

arr[position] = value;

priorities[position] = priority;

size++;

cout << "Element " << value << " enqueued into the priority queue with priority " << priority << ".\n";

}

// Function to dequeue the highest priority element from the priority queue

void dequeue() {

if (isEmpty()) {

cout << "Priority Queue Underflow! Cannot dequeue from an empty priority queue.\n";

return;

}

int value = arr[0]; // Get the element with highest priority

// Shift elements to fill the gap

for (int i = 0; i < size - 1; i++) {

arr[i] = arr[i + 1];

priorities[i] = priorities[i + 1];

}

size--;

cout << "Element " << value << " dequeued from the priority queue.\n";

}

// Function to display all elements of the priority queue

void display() {

if (isEmpty()) {

cout << "Priority Queue is empty. Nothing to display.\n";

return;

}

cout << "Priority Queue elements: ";

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

cout << arr[i] << "(" << priorities[i] << ") ";

}

cout << endl;

}

};

int main() {

PriorityQueue priorityQueue;

int choice;

int value, priority;

do {

cout << "\nPriority Queue Operations:" << endl;

cout << "1. Enqueue" << endl;

cout << "2. Dequeue" << endl;

cout << "3. Display" << endl;

cout << "0. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter the value to enqueue: ";

cin >> value;

cout << "Enter the priority of the value: ";

cin >> priority;

priorityQueue.enqueue(value, priority);

break;

case 2:

priorityQueue.dequeue();

break;

case 3:

priorityQueue.display();

break;

case 0:

cout << "Exiting program..." << endl;

break;

default:

cout << "Invalid choice! Please enter a valid option." << endl;

}

} while (choice != 0);

return 0;

}

**Double ended queue**

#include <iostream>

using namespace std;

#define MAX\_SIZE 100 // Maximum size of the deque

class Deque {

private:

int arr[MAX\_SIZE]; // Array to store deque elements

int front, rear; // Indices for front and rear of the deque

public:

Deque() { front = -1; rear = -1; } // Constructor to initialize an empty deque

// Function to check if the deque is empty

bool isEmpty() {

return front == -1 && rear == -1;

}

// Function to check if the deque is full

bool isFull() {

return (rear + 1) % MAX\_SIZE == front;

}

// Function to insert an element at the front of the deque

void insertFront(int value) {

if (isFull()) {

cout << "Deque Overflow! Cannot insert more elements.\n";

return;

}

if (isEmpty()) {

front = rear = 0; // If deque is empty, set front and rear to 0

} else {

front = (front - 1 + MAX\_SIZE) % MAX\_SIZE; // Circular decrement front index

}

arr[front] = value; // Insert the element at the front

cout << "Element " << value << " inserted at the front of the deque.\n";

}

// Function to insert an element at the rear of the deque

void insertRear(int value) {

if (isFull()) {

cout << "Deque Overflow! Cannot insert more elements.\n";

return;

}

if (isEmpty()) {

front = rear = 0; // If deque is empty, set front and rear to 0

} else {

rear = (rear + 1) % MAX\_SIZE; // Circular increment rear index

}

arr[rear] = value; // Insert the element at the rear

cout << "Element " << value << " inserted at the rear of the deque.\n";

}

// Function to delete an element from the front of the deque

void deleteFront() {

if (isEmpty()) {

cout << "Deque Underflow! Cannot delete from an empty deque.\n";

return;

}

int value = arr[front]; // Get the element at the front

if (front == rear) {

// If there was only one element in the deque, reset front and rear

front = rear = -1;

} else {

front = (front + 1) % MAX\_SIZE; // Circular increment front index

}

cout << "Element " << value << " deleted from the front of the deque.\n";

}

// Function to delete an element from the rear of the deque

void deleteRear() {

if (isEmpty()) {

cout << "Deque Underflow! Cannot delete from an empty deque.\n";

return;

}

int value = arr[rear]; // Get the element at the rear

if (front == rear) {

// If there was only one element in the deque, reset front and rear

front = rear = -1;

} else {

rear = (rear - 1 + MAX\_SIZE) % MAX\_SIZE; // Circular decrement rear index

}

cout << "Element " << value << " deleted from the rear of the deque.\n";

}

// Function to display all elements of the deque

void display() {

if (isEmpty()) {

cout << "Deque is empty. Nothing to display.\n";

return;

}

cout << "Deque elements: ";

int i = front;

while (true) {

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

if (i == rear) break; // Exit loop when rear is reached

i = (i + 1) % MAX\_SIZE; // Circular increment index

}

cout << endl;

}

};

int main() {

Deque deque;

int choice;

int value;

do {

cout << "\nDeque Operations:" << endl;

cout << "1. Insert at Front" << endl;

cout << "2. Insert at Rear" << endl;

cout << "3. Delete from Front" << endl;

cout << "4. Delete from Rear" << endl;

cout << "5. Display" << endl;

cout << "0. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter the value to insert at the front: ";

cin >> value;

deque.insertFront(value);

break;

case 2:

cout << "Enter the value to insert at the rear: ";

cin >> value;

deque.insertRear(value);

break;

case 3:

deque.deleteFront();

break;

case 4:

deque.deleteRear();

break;

case 5:

deque.display();

break;

case 0:

cout << "Exiting program..." << endl;

break;

default:

cout << "Invalid choice! Please enter a valid option." << endl;

}

} while (choice != 0);

return 0;

}

**Linked List:**

**Singly Linked List**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

class LinkedList {

private:

Node\* first;

public:

LinkedList() : first(NULL) {}

void insertFront(int item) {

Node\* newNode = new Node();

newNode->data = item;

newNode->next = first;

first = newNode;

}

void insertEnd(int item) {

Node\* newNode = new Node();

newNode->data = item;

newNode->next = NULL;

if (first == NULL) {

first = newNode;

return;

}

Node\* temp = first;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

void deleteFront() {

if (first == NULL) {

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

return;

}

Node\* temp = first;

first = first->next;

cout << "Deleted element: " << temp->data << endl;

delete temp;

}

void deleteEnd() {

if (first == NULL) {

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

return;

}

Node\* temp = first;

Node\* prev = NULL;

while (temp->next != NULL) {

prev = temp;

temp = temp->next;

}

cout << "Deleted element: " << temp->data << endl;

if (prev != NULL)

prev->next = NULL;

else

first = NULL;

delete temp;

}

bool search(int item) {

Node\* temp = first;

while (temp != NULL) {

if (temp->data == item)

return true;

temp = temp->next;

}

return false;

}

void display() {

if (first == NULL) {

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

return;

}

Node\* temp = first;

while (temp != NULL) {

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

temp = temp->next;

}

cout << endl;

}

int countNodes() {

int count = 0;

Node\* temp = first;

while (temp != NULL) {

count++;

temp = temp->next;

}

return count;

}

void reverse() {

Node\* prev = NULL;

Node\* current = first;

Node\* nextNode = NULL;

while (current != NULL) {

nextNode = current->next;

current->next = prev;

prev = current;

current = nextNode;

}

first = prev;

}

};

int main() {

LinkedList list;

while (true) {

cout << "\nSelect an operation:\n";

cout << "1. Insert at front\n";

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

cout << "3. Delete from front\n";

cout << "4. Delete from end\n";

cout << "5. Search list\n";

cout << "6. Display\n";

cout << "7. Count nodes\n";

cout << "8. Reverse\n";

cout << "9. Exit\n";

int choice, item;

cin >> choice;

switch (choice) {

case 1:

cout << "Enter item to insert at front: ";

cin >> item;

list.insertFront(item);

break;

case 2:

cout << "Enter item to insert at end: ";

cin >> item;

list.insertEnd(item);

break;

case 3:

list.deleteFront();

break;

case 4:

list.deleteEnd();

break;

case 5:

cout << "Enter item to search: ";

cin >> item;

if (list.search(item))

cout << "Element found" << endl;

else

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

break;

case 6:

cout << "List: ";

list.display();

break;

case 7:

cout << "Number of nodes: " << list.countNodes() << endl;

break;

case 8:

list.reverse();

cout << "List reversed." << endl;

break;

case 9:

return 0;

default:

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

break;

}

}

return 0;

}

**Circular Linked List**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* next;

};

class CircularLinkedList {

private:

Node\* first;

public:

CircularLinkedList() : first(NULL) {}

void insertFront(int item) {

Node\* newNode = new Node();

newNode->data = item;

newNode->next = NULL;

if (first == NULL) {

first = newNode;

first->next = first;

} else {

Node\* temp = first;

while (temp->next != first) {

temp = temp->next;

}

newNode->next = first;

temp->next = newNode;

first = newNode;

}

cout<<"Element entered at the front"<<endl;

}

void insertEnd(int item) {

Node\* newNode = new Node();

newNode->data = item;

newNode->next = NULL;

if (first == NULL) {

first = newNode;

first->next = first;

} else {

Node\* temp = first;

while (temp->next != first) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = first;

}

cout<<"Element entered at the end"<<endl;

}

void deleteFront() {

if (first == NULL) {

cout << "Underflow" << endl;

return;

}

Node\* temp = first;

if (temp->next == first) {

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

first = NULL;

} else {

Node\* move = first;

while (move->next != first) {

move = move->next;

}

cout << "Deleted Front: " << temp->data << endl;

first = first->next;

move->next = first;

delete temp;

}

}

void deleteEnd() {

if (first == NULL) {

cout << "Underflow" << endl;

return;

}

Node\* move = first;

Node\* pred = NULL;

while (move->next != first) {

pred = move;

move = move->next;

}

cout << "Deleted End : " << move->data << endl;

if (pred != NULL) {

pred->next = first;

} else {

first = NULL;

}

delete move;

}

bool search(int item) {

if (first == NULL) {

cout << "Underflow" << endl;

return false;

}

Node\* save = first;

int count = 1;

while (save->next != first) {

count++;

if (save->data == item) {

cout << "Element found at position: " << count << endl;

return true;

}

save = save->next;

}

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

return false;

}

void display() {

if (first == NULL) {

cout << "Underflow" << endl;

return;

}

Node\* save = first;

do {

cout << save->data << " ";

save = save->next;

} while (save != first);

cout << endl;

}

int count() {

if (first == NULL) {

cout << "Underflow" << endl;

return 0;

}

Node\* save = first;

int count = 1;

while (save->next != first) {

count++;

save = save->next;

}

return count;

}

};

int main() {

CircularLinkedList cll;

int choice, item;

while (true) {

cout << "Circular Linked List Menu:" << endl;

cout << "1. Insert at the front" << endl;

cout << "2. Insert at the end" << endl;

cout << "3. Delete from the front" << endl;

cout << "4. Delete from the end" << endl;

cout << "5. Search for an element" << endl;

cout << "6. Display the list" << endl;

cout << "7. Count the number of nodes" << endl;

cout << "8. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter element to insert at the front: ";

cin >> item;

cll.insertFront(item);

break;

case 2:

cout << "Enter element to insert at the end: ";

cin >> item;

cll.insertEnd(item);

break;

case 3:

cll.deleteFront();

break;

case 4:

cll.deleteEnd();

break;

case 5:

cout << "Enter element to search: ";

cin >> item;

cll.search(item);

break;

case 6:

cout << "Circular Linked List: ";

cll.display();

break;

case 7:

cout << "Number of nodes: " << cll.count() << endl;

break;

case 8:

return 0;

default:

cout << "Invalid choice. Please try again." << endl;

}

}

return 0;

}

**Doubly Linked List**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* prev;

Node\* next;

};

class DoublyLinkedList {

private:

Node\* first;

public:

DoublyLinkedList() {

first = NULL;

}

void insertFront(int x) {

Node\* newNode = new Node;

newNode->data = x;

newNode->prev = NULL;

newNode->next = first;

if (first != NULL) {

first->prev = newNode;

}

first = newNode;

cout<<"Inserted at front\n";

}

void insertEnd(int x) {

Node\* newNode = new Node;

newNode->data = x;

newNode->next = NULL;

if (first == NULL) {

newNode->prev = NULL;

first = newNode;

return;

}

Node\* move = first;

while (move->next != NULL) {

move = move->next;

}

move->next = newNode;

newNode->prev = move;

cout<<"Inserted at end\n";

}

void deleteFront() {

if (first == NULL) {

cout << "Empty linked list." << endl;

return;

}

if (first->next == NULL) {

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

delete first;

first = NULL;

} else {

Node\* temp = first;

first = first->next;

first->prev = NULL;

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

delete temp;

}

cout<<"Deleted at front\n";

}

void deleteEnd() {

if (first == NULL) {

cout << "Empty linked list." << endl;

return;

}

if (first->next == NULL) {

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

delete first;

first = NULL;

} else {

Node\* move = first;

while (move->next != NULL) {

move = move->next;

}

cout << "Deleted: " << move->data << endl;

move->prev->next = NULL;

delete move;

}

cout<<"Deleted at end\n";

}

void display() {

if (first == NULL) {

cout << "Empty linked list." << endl;

return;

}

Node\* move = first;

while (move != NULL) {

cout << move->data << " ";

move = move->next;

}

cout << endl;

}

int countNodes() {

int count = 0;

Node\* move = first;

while (move != NULL) {

count++;

move = move->next;

}

return count;

}

bool search(int x) {

Node\* move = first;

while (move != NULL) {

if (move->data == x) {

return true;

}

move = move->next;

}

return false;

}

};

int main() {

DoublyLinkedList dll;

int choice, data;

while (true) {

cout << "1. Insert from Front" << endl;

cout << "2. Insert from End" << endl;

cout << "3. Delete from Front" << endl;

cout << "4. Delete from End" << endl;

cout << "5. Search" << endl;

cout << "6. Display" << endl;

cout << "7. Count Nodes" << endl;

cout << "8. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter data: ";

cin >> data;

dll.insertFront(data);

break;

case 2:

cout << "Enter data: ";

cin >> data;

dll.insertEnd(data);

break;

case 3:

dll.deleteFront();

break;

case 4:

dll.deleteEnd();

break;

case 5:

cout << "Enter data to search: ";

cin >> data;

if (dll.search(data)) {

cout << "Data found in the linked list." << endl;

} else {

cout << "Data not found in the linked list." << endl;

}

break;

case 6:

cout << "Doubly Linked List: ";

dll.display();

break;

case 7:

cout << "Number of nodes: " << dll.countNodes() << endl;

break;

case 8:

exit(0);

break;

default:

cout << "Invalid choice. Please try again." << endl;

}

}

return 0;

}

**Stack and Linear Queue using Linked List**

#include <iostream>

using namespace std;

// Node structure for linked list

struct Node {

int data;

Node\* next;

};

// Stack class

class Stack {

private:

Node\* top; // Pointer to the top of the stack

public:

Stack() : top(NULL) {} // Constructor to initialize an empty stack

// Function to check if the stack is empty

bool isEmpty() {

return top == NULL;

}

// Function to push an element onto the stack

void push(int value) {

Node\* newNode = new Node;

newNode->data = value;

newNode->next = top;

top = newNode;

cout << "Element " << value << " pushed onto the stack." << endl;

}

// Function to pop an element from the stack

void pop() {

if (isEmpty()) {

cout << "Stack Underflow! Cannot pop from an empty stack." << endl;

return;

}

Node\* temp = top;

top = top->next;

cout << "Element " << temp->data << " popped from the stack." << endl;

delete temp;

}

// Function to display all elements of the stack

void display() {

if (isEmpty()) {

cout << "Stack is empty. Nothing to display." << endl;

return;

}

cout << "Stack elements: ";

Node\* temp = top;

while (temp != NULL) {

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

temp = temp->next;

}

cout << endl;

}

};

// Queue class

class Queue {

private:

Node\* front; // Pointer to the front of the queue

Node\* rear; // Pointer to the rear of the queue

public:

Queue() : front(NULL), rear(NULL) {} // Constructor to initialize an empty queue

// Function to check if the queue is empty

bool isEmpty() {

return front == NULL;

}

// Function to enqueue an element into the queue

void enqueue(int value) {

Node\* newNode = new Node;

newNode->data = value;

newNode->next = NULL;

if (isEmpty()) {

front = rear = newNode;

} else {

rear->next = newNode;

rear = newNode;

}

cout << "Element " << value << " enqueued into the queue." << endl;

}

// Function to dequeue an element from the queue

void dequeue() {

if (isEmpty()) {

cout << "Queue Underflow! Cannot dequeue from an empty queue." << endl;

return;

}

Node\* temp = front;

front = front->next;

cout << "Element " << temp->data << " dequeued from the queue." << endl;

delete temp;

if (front == NULL) {

rear = NULL; // If queue becomes empty after dequeue, reset rear pointer

}

}

// Function to display all elements of the queue

void display() {

if (isEmpty()) {

cout << "Queue is empty. Nothing to display." << endl;

return;

}

cout << "Queue elements: ";

Node\* temp = front;

while (temp != NULL) {

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

temp = temp->next;

}

cout << endl;

}

};

int main() {

Stack stack;

Queue queue;

int choice;

int value;

do {

cout << "\nMenu:" << endl;

cout << "1. Push Element onto Stack" << endl;

cout << "2. Pop Element from Stack" << endl;

cout << "3. Enqueue Element into Queue" << endl;

cout << "4. Dequeue Element from Queue" << endl;

cout << "5. Display Stack" << endl;

cout << "6. Display Queue" << endl;

cout << "0. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter the value to push onto the stack: ";

cin >> value;

stack.push(value);

break;

case 2:

stack.pop();

break;

case 3:

cout << "Enter the value to enqueue into the queue: ";

cin >> value;

queue.enqueue(value);

break;

case 4:

queue.dequeue();

break;

case 5:

stack.display();

break;

case 6:

queue.display();

break;

case 0:

cout << "Exiting program..." << endl;

break;

default:

cout << "Invalid choice! Please enter a valid option." << endl;

}

} while (choice != 0);

return 0;

}

**Graph:**

**Graph using adjacency matrix**

#include <iostream>

using namespace std;

class adjmatrix {

private:

int totalnode;

char nodes[10];

int adjmat[10][10];

public:

adjmatrix();

void acceptedge();

void print();

};

adjmatrix::adjmatrix() {

cout << "How many nodes are there less than 10 ";

cin >> totalnode;

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

cout << "Enter name of nodes " << i << " ";

cin >> nodes[i];

}

}

void adjmatrix::acceptedge() {

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

char ans;

for (int j = 0; j < totalnode; j++) {

cout << "Is there an Edge between " << nodes[i] << " -->" << nodes[j] << " (Y/N)";

cin >> ans;

if (ans == 'Y' || ans == 'y')

adjmat[i][j] = 1;

else

adjmat[i][j] = 0;

}

}

}

void adjmatrix::print() {

cout << " ";

for (int i = 0; i < totalnode; i++)

cout << " " << nodes[i];

cout << endl;

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

cout << nodes[i];

for (int j = 0; j < totalnode; j++) {

cout << " " << adjmat[i][j];

}

cout << endl;

}

}

int main() {

adjmatrix adj;

adj.acceptedge();

adj.print();

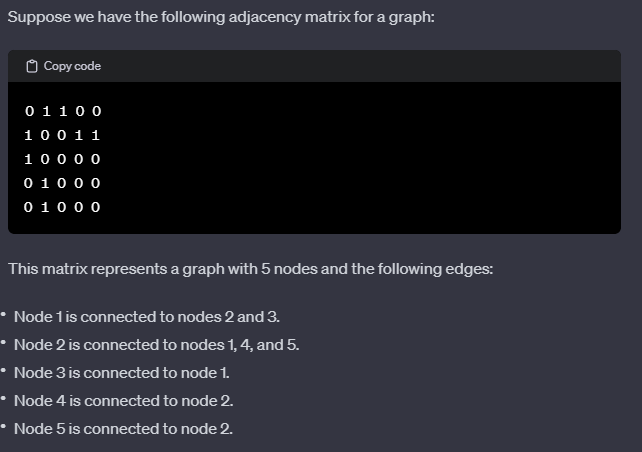
return 0;

}

**Depth First Search (DFS)**

#include <iostream>

using namespace std;

****class dfstree {

private:

int a[20][20];

int visited[20];

int n;

public:

void dfs(int);

void get();

};

void dfstree::get() {

cout << "\nEnter the number of nodes: ";

cin >> n;

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

visited[i] = 0;

}

cout << "\nEnter the adjacency matrix:\n";

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

for (int j = 0; j < n; j++) {

cin >> a[i][j];

}

}

dfs(0);

}

void dfstree::dfs(int v) {

int k;

visited[v] = 1;

cout << "\t" << v + 1;

for (k = 0; k < n; k++) {

if (a[v][k] == 1 && visited[k] == 0) {

dfs(k);

}

}

}

int main() {

dfstree d;

d.get();

return 0;

}

**Breadth First Search (BFS)**

#include <iostream>

using namespace std;

class bfstraval {

private:

int reach[20];

int a[20][20];

int q[20];

int n, i, j, f, r, index;

public:

bfstraval() {

f = r = 0;

index = 1;

}

void get();

void bfs();

};

void bfstraval::get() {

cout << "\nEnter number of vertices: ";

cin >> n;

cout << "\nEnter adjacency matrix:\n";

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

for (j = 1; j <= n; j++) {

reach[i] = 0;

cin >> a[i][j];

}

}

}

void bfstraval::bfs() {

reach[1] = 1;

f++;

r++;

q[r] = index;

cout << "\nBFS is ";

while (f <= r) {

index = q[f];

f++;

cout << index << "\t";

for (j = 1; j <= n; j++) {

if (a[index][j] == 1 && reach[j] != 1) {

reach[j] = 1;

r++;

q[r] = j;

}

}

}

}

int main() {

bfstraval b;

b.get();

b.bfs();

return 0;

}

**Minimum Spanning Tree:**

**Prims Algorithm**

#include<iostream>

using namespace std;

class single {

private:

int v, cost[10][10], i, j, s[10], e[10], near1[10], t[10][3], m, minedge, k, l, mincost;

int jindex;

float dist[10];

public:

void get();

void prim();

void display();

};

void single::get() {

m = 1;

minedge = 9999;

cout << "\nEnter the no. of vertices\n";

cin >> v;

cout << "\nEnter the Adjacency matrix\n";

for (i = 1; i <= v; i++) {

for (j = 1; j <= v; j++) {

cin >> cost[i][j];

if (cost[i][j] == 0)

cost[i][j] = 9999;

else {

e[m] = cost[i][j];

if (e[m] < minedge) {

minedge = e[m];

k = i;

l = j;

}

}

}

}

}

void single::prim() {

t[1][1] = k;

t[1][2] = l;

mincost = cost[k][l];

for (i = 1; i <= v; i++) {

if (cost[i][l] < cost[i][k])

near1[i] = l;

else

near1[i] = k;

}

near1[k] = near1[l] = 0;

int minj = 9999;

for (i = 2; i <= v - 1; i++) {

minj = 9999;

for (j = 1; j <= v; j++) {

if (near1[j] != 0) {

if (cost[j][near1[j]] < minj) {

minj = cost[j][near1[j]];

jindex = j;

}

}

}

t[i][1] = jindex;

t[i][2] = near1[jindex];

mincost = mincost + cost[jindex][near1[jindex]];

near1[jindex] = 0;

for (int k1 = 1; k1 <= v; k1++) {

if (near1[k1] != 0 && cost[k1][near1[k1]] > cost[k1][jindex])

near1[k1] = jindex;

}

}

cout << "\n Mincost =" << mincost;

}

void single::display() {

cout << endl;

cout << "\nMinimum Spanning Tree Path as follows\n";

cout << t[1][1] << "->" << t[1][2];

for (i = 2; i < v; i++) {

cout << "->";

cout << t[i][1];

}

}

int main() {

single d;

d.get();

d.prim();

d.display();

return 0;

}

**Kruskal’s Algorithm**

#include <iostream>

using namespace std;

int parent[10];

class kruskal {

private:

int mincost;

int i, j, min, k, n, ne, cost[10][10], a, b, u, v;

public:

kruskal() {

ne = 1;

mincost = 0;

}

void get();

int find(int);

int uni(int, int);

};

int kruskal::find(int i) {

while (parent[i])

i = parent[i];

return i;

}

int kruskal::uni(int i, int j) {

if (i != j) {

parent[j] = i;

return 1;

}

return 0;

}

void kruskal::get() {

cout << "\n enter no of vertices";

cin >> n;

cout << "\n enter cost matrix";

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

for (j = 1; j <= n; j++) {

cin >> cost[i][j];

if (cost[i][j] == 0)

cost[i][j] = 999;

}

}

cout << "\nThe edges of Minimum Cost Spanning Tree are\n\n";

while (ne < n) {

for (i = 1, min = 999; i <= n; i++) {

for (j = 1; j <= n; j++) {

if (cost[i][j] < min) {

min = cost[i][j];

a = u = i;

b = v = j;

}

}

}

u = find(u);

v = find(v);

if (uni(u, v)) {

cout << "\n" << ne++ << " edge (" << a << "," << b << ") =" << min << "\n";

mincost += min;

}

cost[a][b] = cost[b][a] = 999;

}

cout << "\n\tMinimum cost = " << mincost << "\n";

}

int main() {

kruskal k;

k.get();

return 0;

}

**Trees:**

**Binary Search Tree**

#include <iostream>

using namespace std;

// Node structure for BST

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val) : data(val), left(NULL), right(NULL) {}

};

// Binary Search Tree class

class BST {

private:

Node\* root;

// Private helper functions

Node\* insertRecursive(Node\* root, int val);

bool searchRecursive(Node\* root, int val);

void inorderRecursive(Node\* root);

void preorderRecursive(Node\* root);

void postorderRecursive(Node\* root);

int countNodesRecursive(Node\* root);

int findLargestRecursive(Node\* root);

int findSmallestRecursive(Node\* root);

public:

BST() : root(NULL) {}

// Public interface functions

void insert(int val);

bool search(int val);

void inorder();

void preorder();

void postorder();

int countNodes();

int findLargest();

int findSmallest();

};

// Insertion operation

void BST::insert(int val) {

root = insertRecursive(root, val);

}

Node\* BST::insertRecursive(Node\* root, int val) {

if (root == NULL) {

return new Node(val);

}

if (val < root->data) {

root->left = insertRecursive(root->left, val);

} else if (val > root->data) {

root->right = insertRecursive(root->right, val);

}

return root;

}

// Search operation

bool BST::search(int val) {

return searchRecursive(root, val);

}

bool BST::searchRecursive(Node\* root, int val) {

if (root == NULL) {

return false;

}

if (root->data == val) {

return true;

} else if (val < root->data) {

return searchRecursive(root->left, val);

} else {

return searchRecursive(root->right, val);

}

}

// Inorder traversal

void BST::inorder() {

inorderRecursive(root);

cout << endl;

}

void BST::inorderRecursive(Node\* root) {

if (root == NULL) {

return;

}

inorderRecursive(root->left);

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

inorderRecursive(root->right);

}

// Preorder traversal

void BST::preorder() {

preorderRecursive(root);

cout << endl;

}

void BST::preorderRecursive(Node\* root) {

if (root == NULL) {

return;

}

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

preorderRecursive(root->left);

preorderRecursive(root->right);

}

// Postorder traversal

void BST::postorder() {

postorderRecursive(root);

cout << endl;

}

void BST::postorderRecursive(Node\* root) {

if (root == NULL) {

return;

}

postorderRecursive(root->left);

postorderRecursive(root->right);

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

}

// Count nodes operation

int BST::countNodes() {

return countNodesRecursive(root);

}

int BST::countNodesRecursive(Node\* root) {

if (root == NULL) {

return 0;

}

return 1 + countNodesRecursive(root->left) + countNodesRecursive(root->right);

}

// Find largest element operation

int BST::findLargest() {

return findLargestRecursive(root);

}

int BST::findLargestRecursive(Node\* root) {

if (root == NULL) {

return INT\_MIN;

}

while (root->right != NULL) {

root = root->right;

}

return root->data;

}

// Find smallest element operation

int BST::findSmallest() {

return findSmallestRecursive(root);

}

int BST::findSmallestRecursive(Node\* root) {

if (root == NULL) {

return INT\_MAX;

}

while (root->left != NULL) {

root = root->left;

}

return root->data;

}

// Main function

int main() {

BST tree;

int numElements;

cout << "Enter the number of elements to insert: ";

cin >> numElements;

cout << "Enter " << numElements << " elements:\n";

int val;

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

cin >> val;

tree.insert(val);

}

cout << "Inorder traversal: ";

tree.inorder();

cout << "Preorder traversal: ";

tree.preorder();

cout << "Postorder traversal: ";

tree.postorder();

cout << "Number of nodes in the tree: " << tree.countNodes() << endl;

cout << "Largest element in the tree: " << tree.findLargest() << endl;

cout << "Smallest element in the tree: " << tree.findSmallest() << endl;

int key;

cout << "Enter a value to search in the tree: ";

cin >> key;

if (tree.search(key)) {

cout << key << " found in the tree" << endl;

} else {

cout << key << " not found in the tree" << endl;

}

return 0;

}

**Heap Tree**

#include<iostream>

#include<cstdlib>

#include<conio.h>

using namespace std;

class heaptree {

int n, heap[20], i, parent, data, last;

public:

heaptree() {

last = -1;

}

void get() {

cout << "\nEnter Range: ";

cin >> n;

for (i = 0; i < n; i++)

heap[i] = 0;

}

void insertheap() {

if (last >= n - 1)

cout << "\nHeap is full.";

else {

cout << "\nEnter the element: ";

cin >> data;

last++;

heap[last] = data;

reheapup(last);

cout << "Element inserted successfully.";

}

}

void reheapup(int newindex) {

if (newindex != 0) {

parent = (newindex - 1) / 2;

if (heap[newindex] >= heap[parent]) {

int temp;

temp = heap[newindex];

heap[newindex] = heap[parent];

heap[parent] = temp;

reheapup(parent);

}

else

return;

}

return;

}

void display() {

cout << "\nHeap Tree: ";

for (i = 0; i < n; i++)

cout << heap[i] << " ";

cout << endl;

}

};

int main() {

int ch;

heaptree h;

h.get();

cout << "\n1: Insert\n2: Display\n3: Exit";

do {

cout << "\nEnter choice: ";

cin >> ch;

switch (ch) {

case 1: h.insertheap(); break;

case 2: h.display(); break;

case 3: exit(0);

default: cout << "Invalid choice. Please try again.";

}

} while (ch != 3);

getch();

return 0;

}

**Linked List Application Polynomials**

#include<iostream>

#include<conio.h>

using namespace std;

class polyadd {

private:

struct node {

float coeff;

int exp;

node \*link;

} \*p;

public:

polyadd() {

p = NULL;

}

void poly\_append(float c, int e) {

node \*temp, \*temp1;

temp1 = p;

temp = new node;

temp->coeff = c;

temp->exp = e;

temp->link = NULL;

if (p == NULL) {

p = temp;

} else {

while (temp1->link != NULL)

temp1 = temp1->link;

temp1->link = temp;

}

}

void display() {

node \*temp = p;

int f = 0;

while (temp != NULL) {

if (f != 0) {

if (temp->coeff > 0)

cout << "+";

else

cout << " ";

}

if (temp->exp != 0)

cout << temp->coeff << "x^" << temp->exp;

else

cout << temp->coeff;

temp = temp->link;

f = 1;

}

}

void add(polyadd &l1, polyadd &l2) {

node \*z;

if (l1.p == NULL && l2.p == NULL)

return;

node \*temp1, \*temp2;

temp1 = l1.p;

temp2 = l2.p;

while (temp1 != NULL && temp2 != NULL) {

if (p == NULL) {

p = new node;

z = p;

} else {

z->link = new node;

z = z->link;

}

if (temp1->exp < temp2->exp) {

z->coeff = temp2->coeff;

z->exp = temp2->exp;

temp2 = temp2->link;

} else {

if (temp1->exp > temp2->exp) {

z->coeff = temp1->coeff;

z->exp = temp1->exp;

temp1 = temp1->link;

} else {

if (temp1->exp == temp2->exp) {

z->coeff = temp1->coeff + temp2->coeff;

z->exp = temp1->exp;

temp1 = temp1->link;

temp2 = temp2->link;

}

}

}

}

while (temp1 != NULL) {

if (p == NULL) {

p = new node;

z = p;

} else {

z->link = new node;

z = z->link;

}

z->coeff = temp1->coeff;

z->exp = temp1->exp;

temp1 = temp1->link;

}

while (temp2 != NULL) {

if (p == NULL) {

p = new node;

z = p;

} else {

z->link = new node;

z = z->link;

}

z->coeff = temp2->coeff;

z->exp = temp2->exp;

temp2 = temp2->link;

}

z = z->link;

}

};

int main() {

polyadd p1, p2, p3;

int n, exp;

float coeff;

// Input for first polynomial (p1)

cout << "Enter the number of terms for the first polynomial: ";

cin >> n;

cout << "Enter the coefficients and exponents for the first polynomial:" << endl;

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

cout << "Coefficient for term " << i+1 << ": ";

cin >> coeff;

cout << "Exponent for term " << i+1 << ": ";

cin >> exp;

p1.poly\_append(coeff, exp);

}

// Input for second polynomial (p2)

cout << "Enter the number of terms for the second polynomial: ";

cin >> n;

cout << "Enter the coefficients and exponents for the second polynomial:" << endl;

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

cout << "Coefficient for term " << i+1 << ": ";

cin >> coeff;

cout << "Exponent for term " << i+1 << ": ";

cin >> exp;

p2.poly\_append(coeff, exp);

}

cout << "\nFirst polynomial is: ";

p1.display();

cout << "\nSecond polynomial is: ";

p2.display();

p3.add(p1, p2);

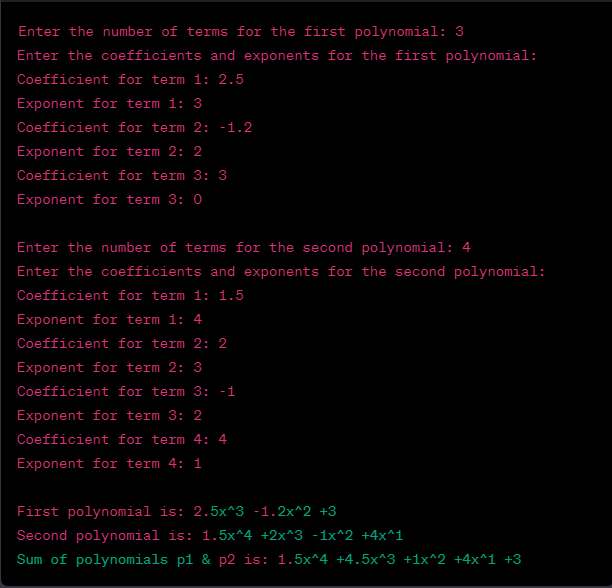
cout << "\nSum of polynomials p1 & p2 is: ";

p3.display();

getch();

return 0;

}

****

**Hashing Techniques**

#include <iostream>

#include <cstdlib>

using namespace std;

const int SIZE = 10;

class HashTable {

private:

int table[SIZE];

public:

HashTable() {

for (int i = 0; i < SIZE; ++i)

table[i] = -1; // Initialize all elements to -1 (indicating empty)

}

// Hash function using Modulo Division

int hashModuloDivision(int key) {

return key % SIZE;

}

// Hash function using Digit Extraction

int hashDigitExtraction(int key) {

return key % 10;

}

// Hash function using Fold Shift

int hashFoldShift(int key) {

int sum = 0;

while (key > 0) {

sum += key % 10;

key /= 10;

}

return sum % SIZE;

}

// Hash function using Fold Boundary

int hashFoldBoundary(int key) {

int sum = 0;

while (key > 0) {

sum += key % 1000; // Extract three digits at a time

key /= 1000;

}

return sum % SIZE;

}

// Linear probing for collision resolution

void insertLinearProbe(int key) {

int index = hashModuloDivision(key);

while (table[index] != -1) {

index = (index + 1) % SIZE; // Linear probing

}

table[index] = key;

}

// Display the hash table

void display() {

cout << "Hash Table:" << endl;

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

cout << "[" << i << "]: ";

if (table[i] != -1)

cout << table[i];

cout << endl;

}

}

};

int main() {

HashTable ht;

int keys[SIZE];

cout << "Enter " << SIZE << " keys:" << endl;

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

cout << "Key " << i + 1 << ": ";

cin >> keys[i];

}

// Insert elements into the hash table using linear probe

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

ht.insertLinearProbe(keys[i]);

}

// Display the hash table

ht.display();

return 0;

}