# Linked List:

**Node Struct:**

struct Node // Node of the linked list

{

int data;

Node\* next;

Node(int data);

};

Node::Node(int data) // Constructor

{

this->data = data;

this->next = nullptr;

}

**Linked List Class:**

class LinkedList // Linked List

{

Node\* head; // Root of the linked list

void Insert(int data, Node\*& node, int index, int currentindex); // Insert a node in the linked list

void Delete(int data, Node\*& node); // Delete a node from the linked list

void Search(int data, Node\* node); // Search a node in the linked list

void Print(Node\* node); // Print the linked list

Node\* Reverse(Node\* node); // Reverse the linked list

public:

LinkedList(); // Constructor

void Insert(int data, int index); // Insert a node in the linked list to call Recursively

void Delete(int data); // Delete a node from the linked list to call Recursively

void Search(int data); // Search a node in the linked list to call Recursively

void Print(); // Print the linked list to call Recursively

void Reverse(); // Reverse the linked list to call Recursively

};

**Constructor:**

LinkedList::LinkedList() {

this->root = nullptr;

}

**Insertion:**

void LinkedList::Insert(int data, int index)

{

Insert(data, this->head, index, 0);

}

void LinkedList::Insert(int data, Node\*& node, int index, int currentindex)

{

if (index == 0 || node == nullptr)

{

Node\* newnode = new Node(data);

newnode->next = node;

head = newnode;

}

else if (currentindex == index - 1)

{

Node\* newnode = new Node(data);

newnode->next = node->next;

node->next = newnode;

}

else if (node->next != nullptr)

{

Insert(data, node->next, index, currentindex + 1);

}

else

{

cout << "Index out of range" << endl;

}

}

**Deletion:**

void LinkedList::Delete(int data)

{

Delete(data, this->head);

}

void LinkedList::Delete(int data, Node\*& node)

{

if (head == nullptr)

{

cout << "List is empty. Cannot delete." << endl;

}

else if (head->data == data)

{

Node\* temp = head;

head = head->next;

delete temp;

}

else

{

Node\* current = head;

while (current->next != nullptr && current->next->data != data)

{

current = current->next;

}

if (current->next != nullptr)

{

Node\* temp = current->next;

current->next = current->next->next;

delete temp;

}

else

{

cout << "Node with data " << data << " not found. Cannot delete." << endl;

}

}

}

**Print:**

void LinkedList::Print()

{

Print(this->root);

}

void LinkedList::Print(Node\* node)

{

if (node == nullptr)

{

return;

}

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

Print(node->next);

}

**Search:**

void LinkedList::Search(int data)

{

Search(data, this->root);

}

void LinkedList::Search(int data, Node\* node)

{

if (node == nullptr)

{

cout << "Not Found" << endl;

return;

}

if (node->data == data)

{

cout << "Found" << endl;

return;

}

Search(data, node->next);

}

**Reverse with Recursion:**

void LinkedList::Reverse()

{

head = Reverse(this->head);

}

Node\* LinkedList:: Reverse(Node \* node) {

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

return node;

}

// Reverse the rest of the list

Node\* temp = Reverse(node->next);

// Adjust pointers to reverse the current node

node->next->next = node;

node->next = nullptr;

return temp;

}

**Reverse with Loops:**

Node\*LinkedList :: Reverse(Node\* node)

{

Node\* prev = nullptr;

Node\* current = node;

Node\* next = nullptr;

while (current != nullptr) {

next = current->next;

current->next = prev;

prev = current;

current = next;

}

node = prev;

return node;

}

**Time Complexity in Linked List:**

**Insertion:**

1. Inserting at the beginning of the linked list: O(1) time complexity.
2. Inserting at the end of the linked list: O(n) time complexity, where n is the number of nodes in the linked list.
3. Inserting at a specific position (given the pointer to that position): O(1) time complexity.

**Deletion:**

1. Deleting at the beginning of the linked list: O(1) time complexity.
2. Deleting at the end of the linked list: O(n) time complexity, where n is the number of nodes in the linked list.
3. Deleting at a specific position (given the pointer to that position): O(1) time complexity.

**Search:**

1. Searching for a specific element in a linked list: O(n) time complexity in the worst case, where n is the number of nodes in the linked list.
2. Searching for a specific element in a sorted linked list: O(n) time complexity for a simple linear search. For a binary search, the linked list should be doubly linked or circular, and the time complexity would be O(log n).

**Reverse:**

The time complexity for reversing a singly linked list is O(n)

**Types of Linked List:**

* Doubly Linked List
* Circular Linked List

**Doubly Linked List**

**Node Struct:**

struct Node

{

int data;

Node\* next;

Node\* Prev;

Node(int data)

{

this->data = data;

next = nullptr;

Prev = nullptr;

}

};

**Class:**

// List class

class List {

Node\* head;

Node\* tail;

// Recursive print

void print(Node\* current);

// Recursive reverse

void reverse(Node\* current);

// Recursive insert

void InsertRec(Node\* current, int data, int index, int currentIndex);

// Recursive delete

void RecursiceDelete(int data, Node\* current);

public:

// Constructor

List();

// Recursive Print Helper

void printRecursive();

// Recursive insert Helper

void insertRecursive(int data, int index);

// Recursive Delete Helper

void DeleteRecursive(int data);

// Recursive Reverse Helper

void reverseRecursive();

;

// Reverse

void reverse();

// Print

void print();

}

**Constructor:**

// Constructor

List::List() {

head = nullptr;

tail = nullptr;

}

**Insertion:**

// Recursive insert Helper

void List::insertRecursive(int data, int index) {

InsertRec(head, data, index, 0);

}

// Recursive insert

void List::InsertRec(Node\* current, int data, int index, int currentIndex) {

if (index == 0 || current == nullptr) {

Node\* newnode = new Node(data);

newnode->next = current;

if (current != nullptr) {

current->Prev = newnode;

}

head = newnode;

tail = newnode;

}

else if (currentIndex == index - 1) {

Node\* newnode = new Node(data);

newnode->next = current->next;

current->next = newnode;

newnode->Prev = current;

if (newnode->next != nullptr) {

newnode->next->Prev = newnode;

}

else {

tail = newnode;

}

}

else if (current->next != nullptr) {

InsertRec(current->next, data, index, currentIndex + 1);

}

else {

cout << "Index out of range" << endl;

}

}

**Deletion:**

// Recursive Delete Helper

void List::DeleteRecursive(int data) {

RecursiceDelete(data, head);

}

void List::RecursiceDelete(int data, Node\* current) {

if (current == nullptr) {

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

}

else if (current->data == data) {

Node\* temp = current;

head = current->next;

if (head != nullptr) {

head->Prev = nullptr;

}

delete temp;

cout << "Node with data " << data << " deleted" << endl;

}

else if (current->next != nullptr && current->next->data != data) {

RecursiceDelete(data, current->next);

}

else if (current->next != nullptr) {

Node\* temp = current->next;

current->next = current->next->next;

if (current->next != nullptr) {

current->next->Prev = current;

}

else {

tail = current;

}

delete temp;

cout << "Node with data " << data << " deleted" << endl;

}

else {

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

}

}

**Print:**

// Recursive Print Helper

void List::printRecursive() {

print(head);

}

void List::print(Node\* current) {

if (current != nullptr) {

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

print(current->next);

}

else {

cout << endl;

}

}

**Time Complexity in Doubly Linked List:**

**Insertion/Deletion at the Beginning/End:**

Time Complexity: O(1)

Explanation: Inserting or deleting a node at the beginning or end of a doubly linked list involves updating pointers directly, which can be done in constant time.

**Insertion/Deletion at a Specific Position:**

Time Complexity: O(n)

Explanation: Finding the position at which to insert or delete requires traversing the list. Once the position is found, the actual insertion or deletion involves updating pointers, which is a constant-time operation.

**Search (Linear Search):**

Time Complexity: O(n)

Explanation: In the worst case, you may need to traverse the entire list to find the desired element. This is because, unlike arrays, there is no direct access to elements by index.

**Access by Index:**

Time Complexity: O(n)

Explanation: Accessing an element by index in a doubly linked list also requires traversing the list until the desired index is reached. Unlike arrays, where direct indexing provides constant-time access, doubly linked lists require linear traversal.

**Reverse:**

Time Complexity: O(n)

Explanation: Reversing a doubly linked list involves swapping the next and prev pointers for each node in the list. This operation requires traversing the entire list once.

**Circular Linked List:**

# Stack:

**Array Based Implementation:**

**Stack Class:**

class Stack

{

int\* arr;

int top;

int capacity;

int size;

public:

Stack(int size);

void push(int data);

int pop();

int peek();

bool isEmpty();

bool isFull();

};

**Constructor:**

Stack::Stack(int size)

{

this->size = size;

arr = new int[size];

top = -1;

capacity = 0;

}

**Push:**

void Stack::push(int data)

{

if (!isFull())

{

arr[++top] = data;

capacity++;

}

else

{

cout << "Stack is full" << endl;

}

}

**Empty and Full:**

bool Stack::isEmpty()

{

return capacity == 0;

}

bool Stack::isFull()

{

return capacity == size;}

**Pop:**

int Stack::pop()

{

if (!isEmpty())

{

capacity--;

return arr[top--];

}

else

{

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

return -1;

}

}

**Peek:**

int Stack::peek()

{

if (!isEmpty())

{

return arr[top];

}

else

{

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

return -1;

}

}

**Linked List Base Implementation:**

**Node Struct:**

struct Node {

int data;

Node\* next;

Node(int data);

};

**Stack Class:**

class Stack

{

Node\* top;

public:

Stack();

void push(int data);

int pop();

int peek();

bool isEmpty();

};

**Constructor:**

Stack::Stack()

{

top = nullptr;

}

**Push:**

void Stack::push(int data)

{

Node\* newnode = new Node(data);

newnode->next = top;

top = newnode;

}

**Pop:**

int Stack::pop()

{

if (!isEmpty())

{

Node\* temp = top;

top = top->next;

int data = temp->data;

delete temp;

return data;

}

else

{

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

return -1;

}

}

**Peek:**

int Stack::peek()

{

if (!isEmpty())

{

return top->data;

}

else

{

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

return -1;

}

}

**Is Empty:**

bool Stack::isEmpty()

{

return top == nullptr;

}

**Time Complexity in Stack:**

**Push (Add an element to the top of the stack):**

Time Complexity: O(1)

Explanation: Adding an element to the top of the stack involves incrementing the stack pointer and placing the new element at the corresponding index in the array. This operation takes constant time.

**Pop (Remove the element from the top of the stack):**

Time Complexity: O(1)

Explanation: Removing the element from the top of the stack involves accessing the element at the current stack pointer, decrementing the stack pointer, and updating the value at that index. These operations also take constant time.

**Peek (View the element at the top of the stack without removing it):**

Time Complexity: O(1)

Explanation: Accessing the element at the top of the stack without removing it is a constant-time operation, as it only involves reading the value at the current stack pointer.

**Is Empty (Check if the stack is empty):**

Time Complexity: O(1)

Explanation: Checking whether the stack is empty can be done by comparing the stack pointer to the base index of the array. This comparison takes constant time.

# Queue:

**Array Based Implementation:**

**Queue Class:**

class Queue

{

int\* arr;

int front;

int rear;

int capacity;

int size;

public:

Queue(int size);

void enqueue(int data);

int dequeue();

int peek();

bool isEmpty();

bool isFull();

};

**Constructor:**

Queue::Queue(int size)

{

this->size = size;

arr = new int[size];

front = 0;

rear = -1;

capacity = 0;

}

**Enqueue:**

void Queue::enqueue(int data)

{

if (!isFull())

{

rear = (rear + 1) % size;

arr[rear] = data;

capacity++;

}

else

{

cout << "Queue is full" << endl;

}

}

**Dequeue:**

int Queue::dequeue()

{

if (!isEmpty())

{

capacity--;

int data = arr[front];

front = (front + 1) % size;

return data;

}

else

{

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

return -1;

}

}

**Peek:**

int Queue::peek()

{

if (!isEmpty())

{

return arr[front];

}

else

{

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

return -1;

}

}

**Is Empty and Is Full:**

bool Queue::isEmpty()

{

return capacity == 0;

}

bool Queue::isFull()

{

Return

}

**Linked List Based Implementation:**

**Node Struct:**

struct Node {

int data;

Node\* next;

Node(int data);

};

Node::Node(int data)

{

this->data = data;

next = nullptr;

}

**Queue Class:**

class Queue

{

Node\* front;

Node\* rear;

public:

Queue();

void enqueue(int data);

int dequeue();

int peek();

bool isEmpty();

void makeNull();

};

**Constructor:**

Queue::Queue()

{

front = nullptr;

rear = nullptr;

}

**Enqueue:**

void Queue::enqueue(int data)

{

Node\* newnode = new Node(data);

if (isEmpty())

{

front = rear = newnode;

}

else

{

rear->next = newnode;

rear = newnode;

}

}

**Dequeue:**

int Queue::dequeue()

{

if (!isEmpty())

{

Node\* temp = front;

front = front->next;

int data = temp->data;

delete temp;

return data;

}

else

{

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

return -1;

}

}

**Peek:**

int Queue::peek()

{

if (!isEmpty())

{

return front->data;

}

else

{

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

return -1;

}

}

**Is Empty:**

bool Queue::isEmpty()

{

return front == nullptr;

}

**Make Null:**

void Queue::makeNull()

{

while (!isEmpty())

{

dequeue();

}

}

**Time Complexity in Queue:**

**Enqueue (Add an element to the back of the queue):**

Time Complexity: O(1)

Explanation: In a basic implementation using an array or a linked list, adding an element to the back of the queue involves updating the rear pointer or index. This operation takes constant time.

**Dequeue (Remove the element from the front of the queue):**

Time Complexity: O(1)

Explanation: Removing the element from the front of the queue involves updating the front pointer or index. This operation also takes constant time.

**Peek (View the element at the front of the queue without removing it**):

Time Complexity: O(1)

Explanation: Accessing the element at the front of the queue without removing it is a constant-time operation, as it involves reading the value from the front.

**Is Empty (Check if the queue is empty):**

Time Complexity: O(1)

Explanation: Checking whether the queue is empty can be done by comparing the front and rear pointers or indices. This comparison takes constant time.

**Types of Queues:**

1. Circular Queue
2. Double Ended Queue
3. Priorities Queue

**Circular Queue:**

**Queue Class:**

class CircularQueue

{

int\* arr;

int front;

int rear;

int capacity;

int size;

public:

CircularQueue(int size);

void enqueue(int data);

int dequeue();

int peek();

};

**Constructor:**

CircularQueue::CircularQueue(int size)

{

this->size = size;

arr = new int[size];

front = 0;

rear = -1;

capacity = 0;

}

**Enqueue:**

void CircularQueue::enqueue(int data)

{

rear = (rear + 1) % size;

arr[rear] = data;

capacity++;

}

**Dequeue:**

int CircularQueue::dequeue()

{

int data = arr[front];

front = (front + 1) % size;

capacity--;

return data;

}

**Peek:**

int CircularQueue::peek()

{

return arr[front];

}

**Double Ended Queue:**

**Queue Class:**

class Deque {

private:

int\* arr;

int front, rear, capacity;

public:

Deque(int size);

bool isEmpty();

bool isFull();

void insertFront(int value);

void insertRear(int value);

void deleteFront();

void deleteRear();

int getFront();

int getRear();

void display();

~Deque();

};

**Constructor:**

// Constructor

Deque::Deque(int size) {

capacity = size;

arr = new int[capacity];

front = -1;

rear = 0;

}

**Is Empty and Is Full:**

bool Deque::isEmpty() {

return front == -1;

}

// Check if the deque is full

bool Deque::isFull() {

return (front == 0 && rear == capacity - 1) || front == rear + 1;

}

**Insert Front:**

// Insert element at the front of the deque

void Deque::insertFront(int value) {

if (isFull()) {

cout << "Deque is full. Cannot insert at the front." << endl;

return;

}

if (front == -1)

front = rear = 0;

else if (front == 0)

front = capacity - 1;

else

front--;

arr[front] = value;

cout << "Inserted " << value << " at the front." << endl;

}

**Insert Rear:**

// Insert element at the rear of the deque

void Deque::insertRear(int value) {

if (isFull()) {

cout << "Deque is full. Cannot insert at the rear." << endl;

return;

}

if (front == -1)

front = rear = 0;

else if (rear == capacity - 1)

rear = 0;

else

rear++;

arr[rear] = value;

cout << "Inserted " << value << " at the rear." << endl;

}

**Delete Front:**

// Delete element from the front of the deque

void Deque::deleteFront() {

if (isEmpty()) {

cout << "Deque is empty. Cannot delete from the front." << endl;

return;

}

if (front == rear)

front = rear = -1;

else if (front == capacity - 1)

front = 0;

else

front++;

cout << "Deleted element from the front." << endl;

}

**Delete Rear:**

// Delete element from the rear of the deque

void Deque::deleteRear() {

if (isEmpty()) {

cout << "Deque is empty. Cannot delete from the rear." << endl;

return;

}

if (front == rear)

front = rear = -1;

else if (rear == 0)

rear = capacity - 1;

else

rear--;

cout << "Deleted element from the rear." << endl;

}

**Get Front:**

// Get the front element of the deque

int Deque::getFront() {

if (isEmpty()) {

cout << "Deque is empty. No front element." << endl;

return -1;

}

return arr[front];

}

**Get Rear:**

// Get the rear element of the deque

int Deque::getRear() {

if (isEmpty() || rear < 0) {

cout << "Deque is empty. No rear element." << endl;

return -1;

}

return arr[rear];

}

**Display:**

// Display the elements of the deque

void Deque::display() {

if (isEmpty()) {

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

return;

}

cout << "Deque elements: ";

int i = front;

while (true) {

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

if (i == rear)

break;

i = (i + 1) % capacity;

}

cout << endl;

}

**Note:**

This display can also be used in circular and simple queue.

**Priority Queue:**

# Trees:

**Binary Trees:**

**Types:**

* **Full Binary Tree**

1. A tree is called Full Binary Tree if every non leaf node has exactly two children.
2. Other names (Proper Binary Tree, Strictly Binary Tree, 2-Tree)
3. Every node is a full node or a leaf node.

* **Complete Binary Tree**

A perfect/Complete binary tree of height h is a binary tree where:

* All leaf nodes have the same depth or level L
* All other nodes are full-nodes
* Also called as Perfect Binary Tree

**Formulas:**

* A perfect/complete binary tree with height h has 2h leaf nodes.
* A perfect/complete binary tree has 2h+1-1 nodes.
* Number of leaf nodes = 2h
* Number of Internal nodes = 2h-1 nodes
* Total number of nodes = (2 x Number of Leaf Nodes) – 1 or
* Total number of nodes = 2h+1-1
* Height = log2(n+1)–1

**Almost Complete Binary:**

Almost complete binary tree of height h is a binary tree in which.

1. There are 2d nodes at depth d for d = 1, 2..., h−1

• Each leaf in the tree is either at level h or at level h – 1

2. The nodes at depth h are as far left as possible.

**Array Storage:**

**Array Starting from 0th index:**

**Left Child Index:**

Formula: 2 \* i + 1

**Right Child Index:**

Formula: 2 \* i + 2

Where “i” is number of index

**Array Starting from 1st index:**

**Left Child Index:**

Formula: 2 \* i

**Right Child Index:**

Formula: 2 \* i + 1

**Other Types:**

* Binary Search Tree (BST)
* AVL Tree

**Binary Search Tree (BST) :**

**Node Struct:**

struct Node {

int data;

Node\* left;

Node\* right;

Node(int data);

};

Node::Node(int data)

{

this->data = data;

left = nullptr;

right = nullptr;

}

**BST Class:**

class BST {

Node\* root;

void insert(Node\*& Node, int data);

void Delete(Node\*& Node, int data);

bool search(Node\* Node, int data);

void Inorder(Node\* Node);

void Preorder(Node\* Node);

void Postorder(Node\* Node);

Node\* FindLeftMax(Node\* node);

Node\*FindRightMin(Node\* node);

public:

BST();

void insert(int data);

void Delete(int data);

bool search(int data);

void Inorder();

void Preorder();

void Postorder();

};

**Constructor:**

BST::BST()

{

root = nullptr;

}

**Insert:**

void BST::insert(int data)

{

insert(root, data);

}

void BST::insert(Node\*& node, int data)

{

if (node == nullptr)

{

node = new Node(data);

}

else if (data < node->data)

{

insert(node->left, data);

}

else if (data > node->data)

{

insert(node->right, data);

}

else

{

cout << "Duplicate data" << endl;

}

}

**Left Max and Right Min:**

Node\* BST::FindLeftMax(Node\* node)

{

if (node->right == nullptr)

{

return node;

}

else

{

return FindLeftMax(node->right);

}

}

Node\* BST::FindRightMin(Node\* node)

{

if (node->left == nullptr)

{

return node;

}

else

{

return FindRightMin(node->left);

}

}

**Note:**

* Left Max is also called as Inorder Predecessor
* Right Min is also called as Inorder Successor

**Delete:**

void BST::Delete(int data)

{

Delete(root, data);

}

void BST::Delete(Node\*& node, int data)

{

if (node == nullptr)

{

cout << "Data not found" << endl;

}

else if (data < node->data)

{

Delete(node->left, data);

}

else if (data > node->data)

{

Delete(node->right, data);

}

else

{

if (node->left == nullptr && node->right == nullptr)

{

delete node;

node = nullptr;

}

else if (node->left == nullptr) // node has only right child or no child

{

Node\* temp = node;

node = node->right;

delete temp;

}

else if (node->right == nullptr) // node has only left child or no child

{

Node\* temp = node;

node = node->left;

delete temp;

}

else //node has both left and right child

{

Node\* temp = FindLeftMax(node->left); //you can also use FindRightMin(node->right)

node->data = temp->data;

Delete(node->left, temp->data);

}

}

}

**Search:**

bool BST::search(int data)

{

return search(root, data);

}

bool BST::search(Node\* node, int data)

{

if (node == nullptr)

{

return false;

}

else if (data < node->data)

{

return search(node->left, data);

}

else if (data > node->data)

{

return search(node->right, data);

}

else

{

return true;

}

}

**In Order Traversal:**

void BST::Inorder()

{

Inorder(root);

}

void BST::Inorder(Node\* node)

{

if (node != nullptr)

{

Inorder(node->left);

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

Inorder(node->right);

}

}

**Pre Order Traversal:**

void BST::Preorder()

{

Preorder(root);

}

void BST::Preorder(Node\* node)

{

if (node != nullptr)

{

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

Preorder(node->left);

Preorder(node->right);

}

}

**Post Order Traversal:**

void BST::Postorder()

{

Postorder(root);

}

void BST::Postorder(Node\* node)

{

if (node != nullptr)

{

Postorder(node->left);

Postorder(node->right);

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

}

}

**AVL Tree:**

**Node Struct:**

struct Node {

int data;

Node\* left;

Node\* right;

Node(int data);

};

Node::Node(int data)

{

this->data = data;

left = nullptr;

right = nullptr;

}

**AVL Class:**

class AVL {

Node\* root;

void insert(Node\*& Node, int data);

void Delete(Node\*& Node, int data);

bool search(Node\* Node, int data);

Node\* FindLeftMax(Node\* node);

Node\*FindRightMin(Node\* node);

int height(Node\* node);

int balanceFactor(Node\* node);

void rotateLeft(Node\*& node);

void rotateRight(Node\*& node);

void balance(Node\*& node);

void Inorder(Node\* Node);

public:

AVL();

void insert(int data);

void Delete(int data);

bool search(int data);

void Inorder();

};

**Constructor:**

AVL::AVL()

{

root = nullptr;

}

**Insert:**

void AVL::insert(int data)

{

insert(root, data);

}

void AVL::insert(Node\*& node, int data)

{

if (node == nullptr)

{

node = new Node(data);

}

else if (data < node->data)

{

insert(node->left, data);

}

else if (data > node->data)

{

insert(node->right, data);

}

else

{

cout << "Duplicate data" << endl;

}

balance(node);

}

**Deletion:**

void AVL::Delete(int data)

{

Delete(root, data);

}

void AVL::Delete(Node\*& node, int data)

{

if (node == nullptr)

{

cout << "Data not found" << endl;

}

else if (data < node->data)

{

Delete(node->left, data);

}

else if (data > node->data)

{

Delete(node->right, data);

}

else

{

if (node->left == nullptr && node->right == nullptr)

{

delete node;

node = nullptr;

}

else if (node->left == nullptr)

{

Node\* temp = node;

node = node->right;

delete temp;

}

else if (node->right == nullptr)

{

Node\* temp = node;

node = node->left;

delete temp;

}

else

{

Node\* temp = FindLeftMax(node->left); // you can also use FindRightMin(node->right)

node->data = temp->data;

Delete(node->left, temp->data);

}

}

balance(node);

}

**Search:**

bool AVL::search(int data)

{

return search(root, data);

}

bool AVL::search(Node\* node, int data)

{

if (node == nullptr)

{

return false;

}

else if (data < node->data)

{

return search(node->left, data);

}

else if (data > node->data)

{

return search(node->right, data);

}

else

{

return true;

}

}

**Height:**

int AVL::height(Node\* node)

{

if (node == nullptr)

{

return -1;

}

else

{

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

}

}

**Balance Factor:**

int AVL::balanceFactor(Node\* node)

{

return height(node->left) - height(node->right);

}

**Rotate Left:**

void AVL::rotateLeft(Node\*& node)

{

Node\* temp = node->right;

node->right = temp->left;

temp->left = node;

node = temp;

}

**Rotate Right:**

void AVL::rotateRight(Node\*& node)

{

Node\* temp = node->left;

node->left = temp->right;

temp->right = node;

node = temp;

}

**Balance:**

void AVL::balance(Node\*& node)

{

if (balanceFactor(node) == 2) // left heavy

{

if (balanceFactor(node->left) == 1) // left left case

{

rotateRight(node);

}

else // left right case

{

rotateLeft(node->left);

rotateRight(node);

}

}

else if (balanceFactor(node) == -2) // right heavy

{

if (balanceFactor(node->right) == -1) // right right case

{

rotateLeft(node);

}

else // right left case

{

rotateRight(node->right);

rotateLeft(node);

}

}

}

**Time Complexity of Binary Trees:**

**Search (in a Binary Search Tree or AVL):**

Average Case: O(log n)

Worst Case: O(n) (if the tree is unbalanced)

**Insertion (in a Binary Search Tree or AVL):**

Average Case: O(log n)

Worst Case: O(n) (if the tree is unbalanced)

**Deletion (in a Binary Search Tree or AVL):**

Average Case: O(log n)

Worst Case: O(n) (if the tree is unbalanced)

**Traversal (Inorder, Preorder, Postorder):**

O(n) for each traversal, as you need to visit every node in the tree.

**Finding Height of Binary Tree:**

O(n) in the worst case, as you may need to visit every node.

**Checking if a Binary Tree is Balanced:**

O(n), as it may require calculating the height of the left and right subtrees.

# Heap

**Types of Heaps:**

1. Max Heap
2. Min Heap

**Max Heap:**

**Class Max Heaps:**

class MaxHeap

{

int\* arr;

int capacity;

int size;

public:

MaxHeap(int capacity);

void insert(int data);

int extractMax();

int getMax();

void heapify(int index);

void display();

};

**Constructor:**

MaxHeap::MaxHeap(int capacity)

{

this->capacity = capacity;

arr = new int[capacity];

size = 0;

}

**Insertion:**

void MaxHeap::insert(int data)

{

if (size == capacity)

{

cout << "Heap is full" << endl;

return;

}

arr[size++] = data;

int i = size - 1;

while (i > 0 && arr[i] > arr[(i - 1) / 2])

{

swap(arr[i], arr[(i - 1) / 2]);

i = (i - 1) / 2;

}

}

**Extract Max:**

int MaxHeap::extractMax()

{

if (size == 0)

{

cout << "Heap is empty" << endl;

return -1;

}

int data = arr[0];

arr[0] = arr[size - 1];

size--;

heapify(0);

return data;

}

**Get Max:**

int MaxHeap::getMax()

{

if (size == 0)

{

cout << "Heap is empty" << endl;

return -1;

}

return arr[0];

}

**Heapify:**

void MaxHeap::heapify(int index)

{

int left = 2 \* index + 1;

int right = 2 \* index + 2;

int largest = index;

if (left < size && arr[left] > arr[largest])

largest = left;

if (right < size && arr[right] > arr[largest])

largest = right;

if (largest != index)

{

swap(arr[index], arr[largest]);

heapify(largest);

}

}

**Display:**

void MaxHeap::display()

{

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

{

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

}

cout << endl;

}

**Min Heap:**

**Class Min Heap:**

class MinHeap

{

int\* arr;

int capacity;

int size;

public:

MinHeap(int capacity);

void insert(int data);

int extractMin();

int getMin();

void heapify(int index);

void display();

};

**Constructor:**

MinHeap::MinHeap(int capacity)

{

this->capacity = capacity;

arr = new int[capacity];

size = 0;

}

**Insertion:**

void MinHeap::insert(int data)

{

if (size == capacity)

{

cout << "Heap is full" << endl;

return;

}

arr[size++] = data;

int i = size - 1;

while (i > 0 && arr[i] < arr[(i - 1) / 2])

{

swap(arr[i], arr[(i - 1) / 2]);

i = (i - 1) / 2;

}

}

**Extract Min:**

int MinHeap::extractMin()

{

if (size == 0)

{

cout << "Heap is empty" << endl;

return -1;

}

int data = arr[0];

arr[0] = arr[size - 1];

size--;

heapify(0);

return data;

}

**Get Min:**

int MinHeap::getMin()

{

if (size == 0)

{

cout << "Heap is empty" << endl;

return -1;

}

return arr[0];

}

**Heapify:**

void MinHeap::heapify(int index)

{

int left = 2 \* index + 1;

int right = 2 \* index + 2;

int smallest = index;

if (left < size && arr[left] < arr[smallest])

{

smallest = left;

}

if (right < size && arr[right] < arr[smallest])

{

smallest = right;

}

if (smallest != index)

{

swap(arr[index], arr[smallest]);

heapify(smallest);

}

}

**Display:**

void MinHeap::display()

{

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

{

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

}

cout << endl;

}

**Time Complexity of Heap:**

**Insertion (Heapify Up):**

Time Complexity: O(log n)

In the worst case, the height of the binary heap is log(n), where 'n' is the number of elements in the heap. Therefore, the time complexity of inserting an element and restoring the heap property (heapify up) is O(log n).

**Deletion (Heapify Down):**

Time Complexity: O(log n)

Similar to insertion, deleting the root element and restoring the heap property (heapify down) has a time complexity of O(log n).

**Building a Heap (Heapify):**

Time Complexity: O(n)

Building a binary heap from an array of elements can be done efficiently in O(n) time. This is often faster than inserting elements one by one.

**Extracting the Minimum/Maximum:**

Time Complexity: O(log n)

Extracting the minimum (for min-heap) or maximum (for max-heap) element involves removing the root and restoring the heap property, resulting in a time complexity of O(log n).

**Heapify Operation (Bottom-Up Construction):**

Time Complexity: O(n)

Constructing a heap bottom-up from an array has a time complexity of O(n). This is because you can start happifying elements from the last non-leaf node and move towards the root, ensuring the heap property.

# Graphs:

* Directed Graphs
* Undirected Graphs

**Adjacency Matrix:**

**Undirected Graphs:**

#include<iostream>

using namespace std;

class AdjacencyMatrix

{

int\*\* arr;

int size;

public:

AdjacencyMatrix(int size);

void addEdge(int source, int destination);

void display();

void removeEdge(int source, int destination);

};

AdjacencyMatrix::AdjacencyMatrix(int size)

{

this->size = size;

arr = new int\* [size];

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

{

arr[i] = new int[size];

for (int j = 0; j < size; j++)

{

arr[i][j] = 0;

}

}

}

void AdjacencyMatrix::addEdge(int source, int destination)

{

if (source >= 0 && source < size && destination >= 0 && destination < size)

{

arr[source][destination] = 1;

arr[destination][source] = 1;

}

else

{

cout << "Invalid edge" << endl;

}

}

void AdjacencyMatrix::display()

{

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

{

cout << i << " : ";

for (int j = 0; j < size; j++)

{

if (arr[i][j] == 1)

{

cout << j << " ";

}

}

cout << endl;

}

}

void AdjacencyMatrix::removeEdge(int source, int destination)

{

if (source >= 0 && source < size && destination >= 0 && destination < size)

{

arr[source][destination] = 0;

arr[destination][source] = 0;

}

else

{

cout << "Invalid edge" << endl;

}

}

**Directed Graphs:**

void AdjacencyMatrix::addEdge(int source, int destination)

{

if (source >= 0 && source < size && destination >= 0 && destination < size)

{

arr[source][destination] = 1;

}

else

{

cout << "Invalid edge" << endl;

}

}

void AdjacencyMatrix::removeEdge(int source, int destination)

{

if (source >= 0 && source < size && destination >= 0 && destination < size)

{

arr[source][destination] = 0;

}

else

{

cout << "Invalid edge" << endl;

}

}

**Note:**

Only these functions will change.

**Adjacency List:**

# Traversals:

**BFS:**

void BFS(int source)

{

if (source >= 0 && source < size)

{

queue<int> q;

bool\* visited = new bool[size];

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

{

visited[i] = false;

}

q.push(source);

visited[source] = true;

while (!q.empty())

{

int current = q.front();

q.pop();

cout << current << " ";

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

{

if (arr[current][i] == 1 && visited[i] == false)

{

q.push(i);

visited[i] = true;

}

}

}

cout << endl;

}

else

{

cout << "Invalid source" << endl;

}

}

**DFS:**

void AdjacencyMatrix::DFS(int source)

{

if (source >= 0 && source < size)

{

stack<int> s;

bool\* visited = new bool[size];

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

{

visited[i] = false;

}

s.push(source);

visited[source] = true;

while (!s.empty())

{

int current = s.top();

s.pop();

cout << current << " ";

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

{

if (arr[current][i] == 1 && visited[i] == false)

{

s.push(i);

visited[i] = true;

}

}

}

cout << endl;

}

else

{

cout << "Invalid source" << endl;

}

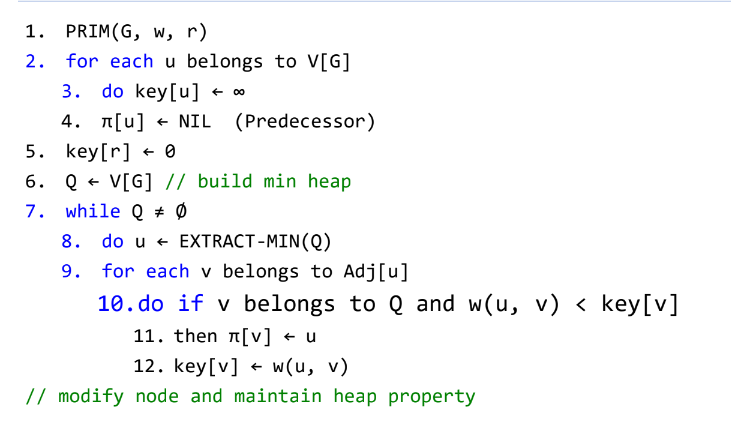
}

**Uses of Adjacency Matrix and List:**

* Matrix is use when graph is denser and it’s order O(n2)
* List is use when graph is sparse and it’s order is O(n+e)
* Where e is number of edges.

# Minimum spanning Tree:

**Prims Algorithm:**



**Dijakstra’s Algorithm:**

A screenshot of a computer code

Description automatically generated