Data Structures

What is Data Structure?

```
A data structure is defined as a particular way of storing and organizing data in our devices to use the data efficiently and effectively.
```

What is Algorithm?

```
It is a set of operations performed in a step-by-step manner to execute a task.
```

or

```
Data Structures is about how data can be stored in different structures.

Algorithms is about how to solve different problems, often by searching through and manipulating data structures.
```

By understanding DSA, you can:

- Decide which data structure or algorithm is best for a given situation.
- Make programs that run faster or use less memory.
- Understand how to approach complex problems and solve them in a systematic way.

DSA is fundamental in nearly every part of the software world:

- Operating Systems
- Database Systems
- Web Applications
- Machine Learning
- Video Games
- Cryptographic Systems
- Data Analysis
- Search Engines

Types of Complexity:

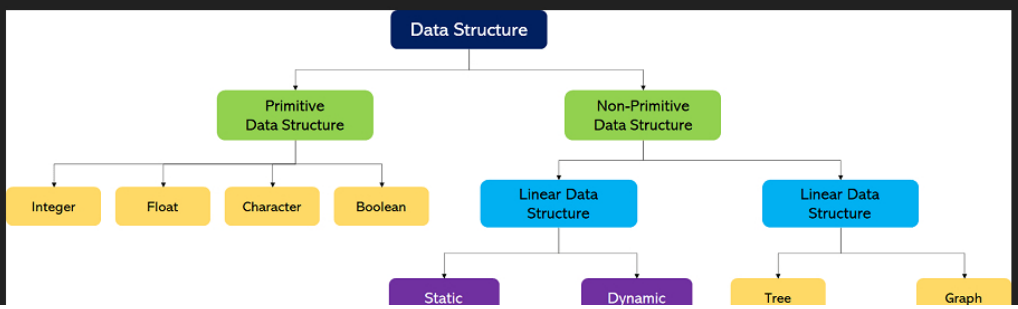
- `Time Complexity:` Time complexity is used to measure the amount of time required to execute the code.
- `Space Complexity:` Space complexity means the amount of space required to execute successfully the functionalities of the code.

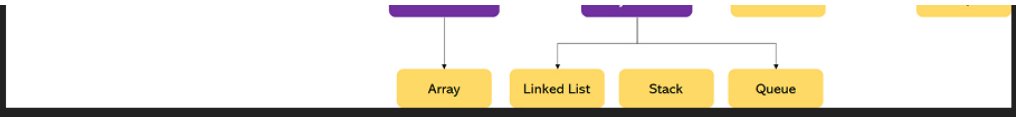
Asymptotic Notations:

- Big-O Notation (O) – Big-O notation specifically describes the worst-case scenario.
- Omega Notation (Ω) – Omega(Ω) notation specifically describes the best-case scenario.
- Theta Notation (θ) – This notation represents the average complexity of an algorithm.

Types of Data Structures

- ### Primitive
  - integer
  - float
  - character
  - boolean
- ### Non-Primitive
  - Linear
    1. Static
       - Array
    2. Dynamic
       - Linked List
       - Stack
       - Queue
  - Non-Linear
    - Tree
    - Graph



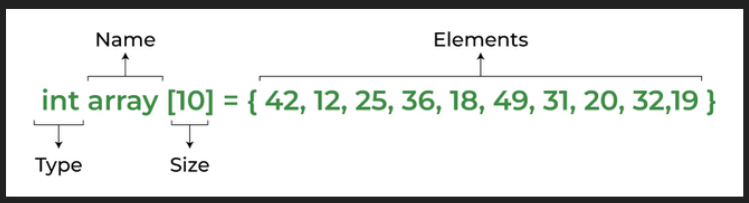


ADT (Abstract Data Type) : combination of Data and its functions

Polish Notation

- infix `a + b`
- prefix `+ab`
- postfix `ab+`

`Array` : An array is a collection of items of same data type stored at contiguous memory locations.



Types of Array operations:

- Traversal: Traverse through the elements of an array.
- Insertion: Inserting a new element in an array. Deletion: Deleting element from the array.
- Searching: Search for an element in the array.
- Sorting: Maintaining the order of elements in the array.

Advantages :

- Arrays allow random access to elements. Arrays represent multiple data items of the same type using a single name.
- Array data structures are used to implement the other data structures like linked lists, stacks, queues, trees, graphs, etc.

Disadvantages :

- An array of fixed size is referred to as a static array.
- Allocating less memory than required to an array leads to loss of data.
- An array is homogeneous in nature so, a single array cannot store values of different data types.

Application of Array:

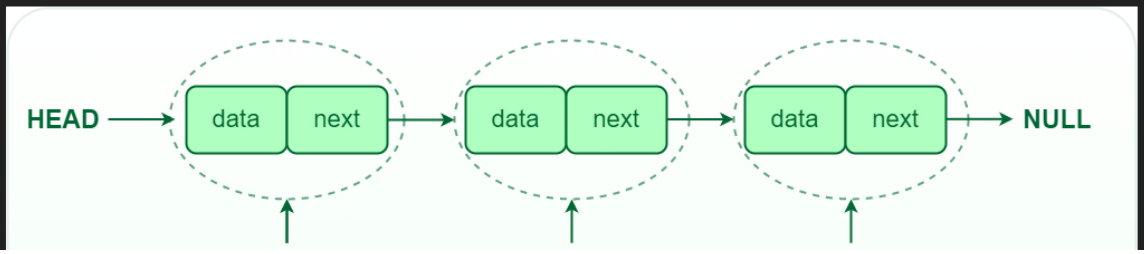
- Database records are usually implemented as arrays.
- It is used for different sorting algorithms such as bubble sort insertion sort, merge sort, and quick sort.

`String` : In some languages, strings are implemented as arrays of characters, making them a derived data type.

Fundamental operations commonly performed on strings in programming.

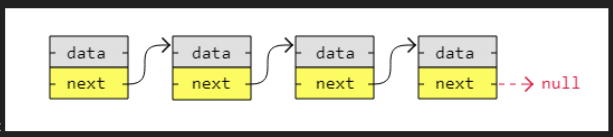
- Concatenation: Combining two strings to create a new string.
- Length: Determining the number of characters in a string.
- Access: Accessing individual characters in a string by index.
- Substring: Extracting a portion of a string.
- Comparison: Comparing two strings to check for equality or order.
- Search: Finding the position of a specific substring within a string.
- Modification: Changing or replacing characters within a string.

`Linked List` : Linked List forms a series of connected nodes, where each node stores the data and the address of the next node.



Node Node Node

Types of Linked List



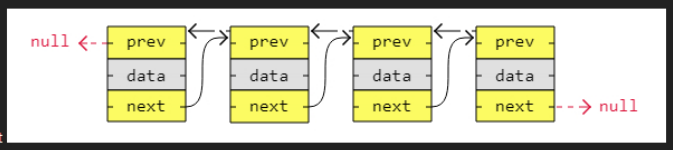
- Singly Linked List

Implementation in cpp

```cpp
#include <iostream>

using namespace std;

// Define a Node structure to represent each element in the list
struct Node {
    int data;
    Node* next;
};

// Define a LinkedList class to manage the list operations
class LinkedList {
private:
    Node* head; // Pointer to the first node in the list

public:
    LinkedList() : head(nullptr) {} // Constructor to initialize an empty list

    // Function to insert a new node at the beginning of the list
    void insertAtBeginning(int value) {
        Node* newNode = new Node;
        newNode->data = value;
        newNode->next = head;
        head = newNode;
        cout << "Inserted " << value << " at the beginning." << endl;
    }

    // Function to insert a new node at the end of the list
    void insertAtEnd(int value) {
        Node* newNode = new Node;
        newNode->data = value;
        newNode->next = nullptr;

        if (head == nullptr) {
            head = newNode;
        } else {
            Node* temp = head;
            while (temp->next != nullptr) {
                temp = temp->next;
            }
            temp->next = newNode;
        }
        cout << "Inserted " << value << " at the end." << endl;
    }

    // Function to display the elements of the list
    void display() {
        Node* temp = head;
        cout << "List: ";
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }

    // Function to search for a value in the list
    bool search(int value) {
        Node* temp = head;
        while (temp != nullptr) {
            if (temp->data == value) {
                return true;
            }
            temp = temp->next;
        }
        return false;
    }

    // Function to delete a node with a given value from the list
    void deleteNode(int value) {
        Node* temp = head;
        Node* prev = nullptr;

        // Traverse the list to find the node to delete
        while (temp != nullptr && temp->data != value) {
            prev = temp;
            temp = temp->next;
        }

        // If the node was found, delete it
        if (temp != nullptr) {
            if (prev == nullptr) {
```

```
                head = temp->next;
            } else {
                prev->next = temp->next;
            }
            delete temp;
            cout << "Deleted " << value << " from the list." << endl;
        } else {
            cout << value << " not found in the list." << endl;
        }
    }

    // Destructor to free memory allocated for nodes
    ~LinkedList() {
        Node* temp = head;
        while (temp != nullptr) {
            Node* nextNode = temp->next;
            delete temp;
            temp = nextNode;
        }
    }
};

int main() {
    LinkedList list;

    list.insertAtEnd(10);
    list.insertAtEnd(20);
    list.insertAtEnd(30);

    list.display(); // Output: List: 10 20 30

    list.insertAtBeginning(5);

    list.display(); // Output: List: 5 10 20 30

    list.deleteNode(20);

    list.display(); // Output: List: 5 10 30

    cout << "Search for 10: " << (list.search(10) ? "Found" : "Not Found") << endl; // Output: Search for 10: Found
    cout << "Search for 40: " << (list.search(40) ? "Found" : "Not Found") << endl; // Output: Search for 40: Not Found

    return 0;
}
```

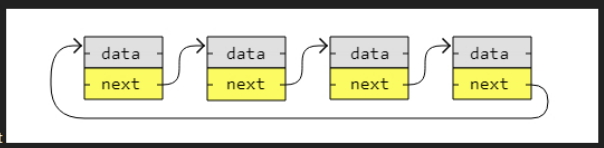
- Doubly Linked List



Implementation in cpp

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

// Node structure for doubly linked list
struct Node {
    int data;
    Node* prev;
    Node* next;
};

class DoublyLinkedList {
private:
    Node* head;
public:
    DoublyLinkedList() : head(nullptr) {}

    // Function to insert a new node at the beginning of the list
    void insertAtBeginning(int value) {
        Node* newNode = new Node;
        newNode->data = value;
        newNode->prev = nullptr;
        newNode->next = head;

        if (head != nullptr) {
            head->prev = newNode;
        }

        head = newNode;
    }

    // Function to insert a new node at the end of the list
    void insertAtEnd(int value) {
        Node* newNode = new Node;
        newNode->data = value;
        newNode->next = nullptr;

        if (head == nullptr) {
            newNode->prev = nullptr;
            head = newNode;
            return;
        }

        Node* temp = head;
```

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

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

    // Function to delete a node by value
    void deleteNode(int value) {
        Node* current = head;

        // Traverse the list to find the node to be deleted
        while (current != nullptr) {
            if (current->data == value) {
                if (current->prev != nullptr) {
                    current->prev->next = current->next;
                } else {
                    head = current->next;
                }

                if (current->next != nullptr) {
                    current->next->prev = current->prev;
                }

                delete current;
                return;
            }
            current = current->next;
        }

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

    // Function to display the elements of the list
    void display() {
        Node* temp = head;

        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }

        cout << endl;
    }
};

int main() {
    DoublyLinkedList dll;

    dll.insertAtEnd(10);
    dll.insertAtBeginning(5);
    dll.insertAtEnd(20);
    dll.insertAtBeginning(2);

    cout << "Doubly Linked List: ";
    dll.display();

    dll.deleteNode(5);
    dll.deleteNode(15); // Element not in the list

    cout << "After deletion: ";
    dll.display();

    return 0;
}
```

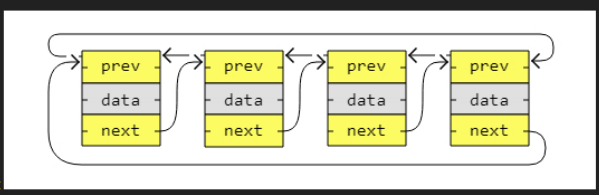
- Circular Linked List 

implementation in cpp

```
#include <iostream>

using namespace std;

// Node structure
struct Node {
    int data;
    Node* next;
};

// Circular linked list class
class CircularLinkedList {
private:
    Node* head;

public:
    // Constructor
    CircularLinkedList() {
        head = nullptr;
    }
```

```
    // Function to insert a node at the beginning of the list
    void insertAtBeginning(int value) {
        Node* newNode = new Node;
        newNode->data = value;
        if (head == nullptr) {
            newNode->next = newNode;  // Point to itself for a single node circular list
        } else {
            Node* temp = head;
            while (temp->next != head) {
                temp = temp->next;
            }
            temp->next = newNode;
            newNode->next = head;
        }
        head = newNode;
    }

    // Function to delete a node from the list
    void deleteNode(int value) {
        if (head == nullptr) {
            cout << "List is empty. Cannot delete." << endl;
            return;
        }

        Node* prev = nullptr;
        Node* curr = head;

        // Find the node to be deleted
        do {
            if (curr->data == value) {
                break;
            }
            prev = curr;
            curr = curr->next;
        } while (curr != head);

        if (curr == head) {
            if (curr->next == head) {  // Only one node in the list
                head = nullptr;
            } else {
                prev->next = curr->next;
                head = curr->next;
            }
            delete curr;
            cout << "Node with value " << value << " deleted successfully." << endl;
        } else {
            cout << "Node with value " << value << " not found." << endl;
        }
    }

    // Function to display the circular linked list
    void displayList() {
        if (head == nullptr) {
            cout << "List is empty." << endl;
            return;
        }

        Node* temp = head;
        do {
            cout << temp->data << " ";
            temp = temp->next;
        } while (temp != head);
        cout << endl;
    }
};

// Main function
int main() {
    CircularLinkedList cll;

    cll.insertAtBeginning(10);
    cll.insertAtBeginning(20);
    cll.insertAtBeginning(30);

    cout << "Circular Linked List: ";
    cll.displayList();

    cll.deleteNode(20);
    cll.deleteNode(40);

    cout << "Updated Circular Linked List: ";
    cll.displayList();

    return 0;
}
```

- Circular Doubly Linked List



implementation in cpp

Implementation in cpp

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

struct Node {
    int data;
    Node* next;
    Node* prev;
};

class CircularDoublyLinkedList {
private:
    Node* head;

public:
    CircularDoublyLinkedList() : head(nullptr) {}

    void insertFront(int value) {
        Node* newNode = new Node;
        newNode->data = value;
        if (head == nullptr) {
            newNode->next = newNode->prev = newNode;
            head = newNode;
        } else {
            Node* lastNode = head->prev;
            newNode->next = head;
            newNode->prev = lastNode;
            lastNode->next = head->prev = newNode;
            head = newNode;
        }
        cout << value << " inserted at the front." << endl;
    }

    void insertEnd(int value) {
        Node* newNode = new Node;
        newNode->data = value;
        if (head == nullptr) {
            newNode->next = newNode->prev = newNode;
            head = newNode;
        } else {
            Node* lastNode = head->prev;
            newNode->next = head;
            newNode->prev = lastNode;
            lastNode->next = head->prev = newNode;
        }
        cout << value << " inserted at the end." << endl;
    }

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

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

    ~CircularDoublyLinkedList() {
        if (head == nullptr) return;

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

int main() {
    CircularDoublyLinkedList list;

    list.insertEnd(10);
    list.insertFront(20);
    list.insertEnd(30);

    list.display();

    return 0;
}
```

Linked List Operations

1. Traversal
2. Remove a node
3. Insert a node
4. Sort

Array V/S Linked List

ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

Hashing

What is a Hash Table?

A hash table is a data structure that implements an associative array abstract data type, where data elements are indexed by a unique key. It uses a technique called hashing to map keys to their associated values, allowing for efficient retrieval and storage of data.

How Does a Hash Table Work?

- Hash Function:

A hash function takes an input (key) and computes a fixed-size output called a hash value or hash code. This hash value is used as the index or address in the hash table where the corresponding data (value) will be stored. A good hash function generates unique hash codes for different keys, aiming to distribute values evenly across the hash table's array.

- Hash Table Structure:

Internally, a hash table typically consists of an array (often called a hash array or buckets) and each slot in the array can hold a key-value pair or a linked list of key-value pairs (in case of collisions). Collisions occur when two different keys hash to the same index. Hash tables handle collisions using techniques like chaining (using linked lists) or open addressing (probing methods).

- Insertion:

To insert a key-value pair into a hash table, the hash function is applied to the key to determine its hash code. The hash code is then mapped to an index in the hash table's array, where the corresponding value is stored. In case of collisions, chaining can be used to append the new key-value pair to the linked list at that index.

- Retrieval:

When retrieving a value associated with a key, the hash function is again applied to the key to compute its hash code. The hash code is used to find the corresponding index in the hash table, and then the value is fetched from that location. If chaining is used and collisions occur, the appropriate linked list is traversed to find the correct key-value pair.

implement in cpp

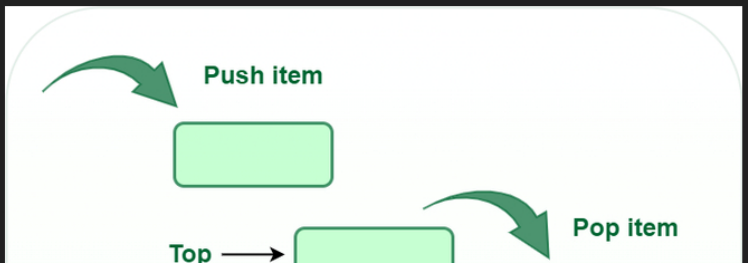
```
#include <iostream>
#include <string>
using namespace std;

// Define a structure for key-value pairs (student name and roll number)
struct KeyValuePair {
    string key;  // Student name
    int value;   // Roll number
};

// Define the Hash Table class
class HashTable {
private:
    static const int TABLE_SIZE = 10;  // Size of the hash table
    KeyValuePair* table[TABLE_SIZE];   // Array of key-value pairs

    // Hash function to compute hash code
    int hashFunction(const string& key);

public:
    HashTable();               // Constructor
    ~HashTable();              // Destructor
    void insert(string key, int value);
    int get(string key);
};

// Constructor: Initialize the hash table array
HashTable::HashTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        table[i] = nullptr;
    }
}

// Destructor: Free memory allocated for key-value pairs
HashTable::~HashTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        delete table[i];
    }
}

// Hash function: Simple hash function using string length
```

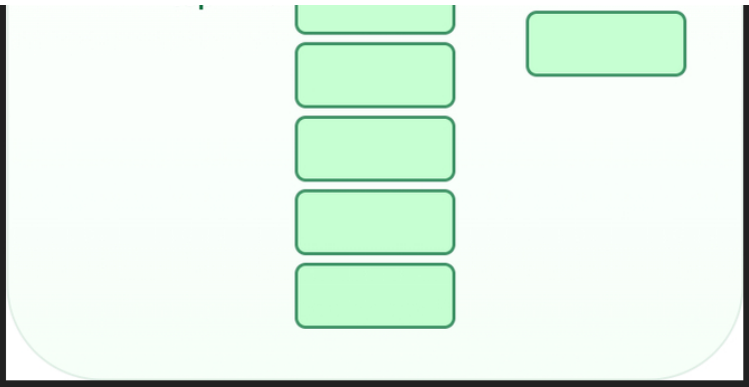
```cpp
int HashTable::hashFunction(const string& key) {
    return key.length() % TABLE_SIZE;
}

// Insert a key-value pair into the hash table
void HashTable::insert(string key, int value) {
    int index = hashFunction(key);
    KeyValuePair* newPair = new KeyValuePair{key, value};

    if (table[index] == nullptr) {
        table[index] = newPair;
    } else {
        // Handle collision using chaining (linked list)
        KeyValuePair* current = table[index];
        while (current->next != nullptr) {
            current = current->next;
        }
        current->next = newPair;
    }
}

// Get the value associated with a key from the hash table
int HashTable::get(string key) {
    int index = hashFunction(key);
    KeyValuePair* current = table[index];

    while (current != nullptr) {
        if (current->key == key) {
            return current->value;
        }
        current = current->next;
    }

    // Key not found
    return -1;
}

// Main function to test the hash table
int main() {
    HashTable ht;

    // Inserting key-value pairs into the hash table
    ht.insert("Alice", 101);
    ht.insert("Bob", 102);
    ht.insert("Charlie", 103);
    ht.insert("David", 104);
    ht.insert("Emma", 105);

    // Retrieving values from the hash table
    cout << "Roll number of Bob: " << ht.get("Bob") << endl;
    cout << "Roll number of Emma: " << ht.get("Emma") << endl;
    cout << "Roll number of Frank: " << ht.get("Frank") << endl;

    return 0;
}
```

Hash Set v/s Hash Map

	Hash Set	Hash Map
Uniqueness and storage	Every element is a unique key.	Every entry is a key-value-pair, with a key that is unique, and a value connected it.
Use case	Checking if an element is in the set, like checking if a name is on a guest list.	Finding information based on a key, like looking up who owns a certain telephone number.
Is it fast to search, add and delete elements?	Yes, average $O(1)$.	Yes, average $O(1)$.
Is there a hash function that takes the key, generates a hash code, and that is the bucket where the element is stored?	Yes	Yes

`Stack` : A stack is a linear data structure in which the insertion of a new element and removal of an existing element takes place at the same end represented as the top of the stack.





Basic Operations on Stack

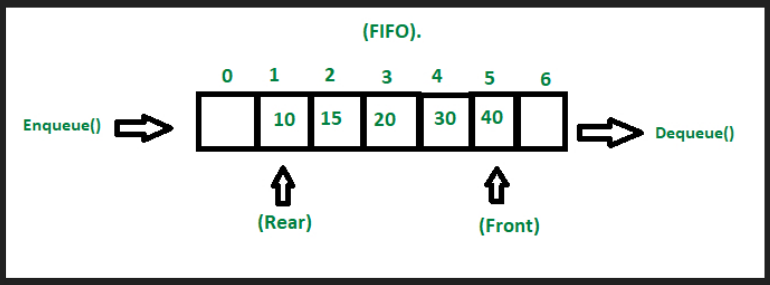
- push() to insert an element into the stack
- pop() to remove an element from the stack
- top() Returns the top element of the stack.
- isEmpty() returns true if stack is empty else false.
- size() returns the size of stack.

implement in cpp

```cpp
#include <iostream>

const int MAX_SIZE = 100; // Maximum size of the stack

class Stack {
private:
    int top;
    int data[MAX_SIZE];

public:
    Stack() {
        top = -1; // Initialize top to -1 (empty stack)
    }

    bool isEmpty() {
        return top == -1;
    }

    bool isFull() {
        return top == MAX_SIZE - 1;
    }

    void push(int value) {
        if (isFull()) {
            std::cout << "Stack overflow! Cannot push more elements.\n";
            return;
        }
        data[++top] = value;
        std::cout << "Pushed: " << value << std::endl;
    }

    void pop() {
        if (isEmpty()) {
            std::cout << "Stack underflow! Cannot pop from an empty stack.\n";
            return;
        }
        std::cout << "Popped: " << data[top--] << std::endl;
    }

    int peek() {
        if (isEmpty()) {
            std::cout << "Stack is empty. Peek operation failed.\n";
            return -1; // Return a default value or throw an exception
        }
        return data[top];
    }

    void display() {
        if (isEmpty()) {
            std::cout << "Stack is empty.\n";
            return;
        }
        std::cout << "Stack elements:\n";
        for (int i = top; i >= 0; i--) {
            std::cout << data[i] << std::endl;
        }
    }
};

int main() {
    Stack stack;

    stack.push(10);
    stack.push(20);
    stack.push(30);
    stack.display();
```

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

    stack.pop();
    stack.display();

    std::cout << "Is stack empty? " << (stack.isEmpty() ? "Yes" : "No") << std::endl;

    return 0;
}
```

Queue : A queue is a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.



implementation in cpp

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

// Node structure for the queue
struct Node {
    int data;
    Node* next;
};

class Queue {
private:
    Node* front;
    Node* rear;

public:
    Queue() {
        front = nullptr;
        rear = nullptr;
    }

    // Function to check if the queue is empty
    bool isEmpty() {
        return front == nullptr;
    }

    // Function to enqueue an element into the queue
    void enqueue(int value) {
        Node* newNode = new Node;
        newNode->data = value;
        newNode->next = nullptr;
        if (isEmpty()) {
            front = rear = newNode;
        } else {
            rear->next = newNode;
            rear = newNode;
        }
        cout << value << " enqueued to the queue." << endl;
    }

    // Function to dequeue an element from the queue
    void dequeue() {
        if (isEmpty()) {
            cout << "Queue is empty. Cannot dequeue." << endl;
        } else {
            Node* temp = front;
            cout << front->data << " dequeued from the queue." << endl;
            front = front->next;
            delete temp;
        }
    }

    // Function to get the front element of the queue
    int peek() {
        if (isEmpty()) {
            cout << "Queue is empty. No front element." << endl;
            return -1;
        }
        return front->data;
    }

    // Function to display the queue
    void display() {
        if (isEmpty()) {
            cout << "Queue is empty." << endl;
```

```
            return;
        }
        Node* temp = front;
        cout << "Queue elements: ";
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    Queue queue;
    queue.enqueue(10);
    queue.enqueue(20);
    queue.enqueue(30);
    queue.display();
    cout << "Front element: " << queue.peek() << endl;
    queue.dequeue();
    queue.display();
    cout << "Front element: " << queue.peek() << endl;
    queue.dequeue();
    queue.dequeue();
    queue.dequeue(); // Trying to dequeue from an empty queue
    queue.display();
    return 0;
}
```

Types of Queue:

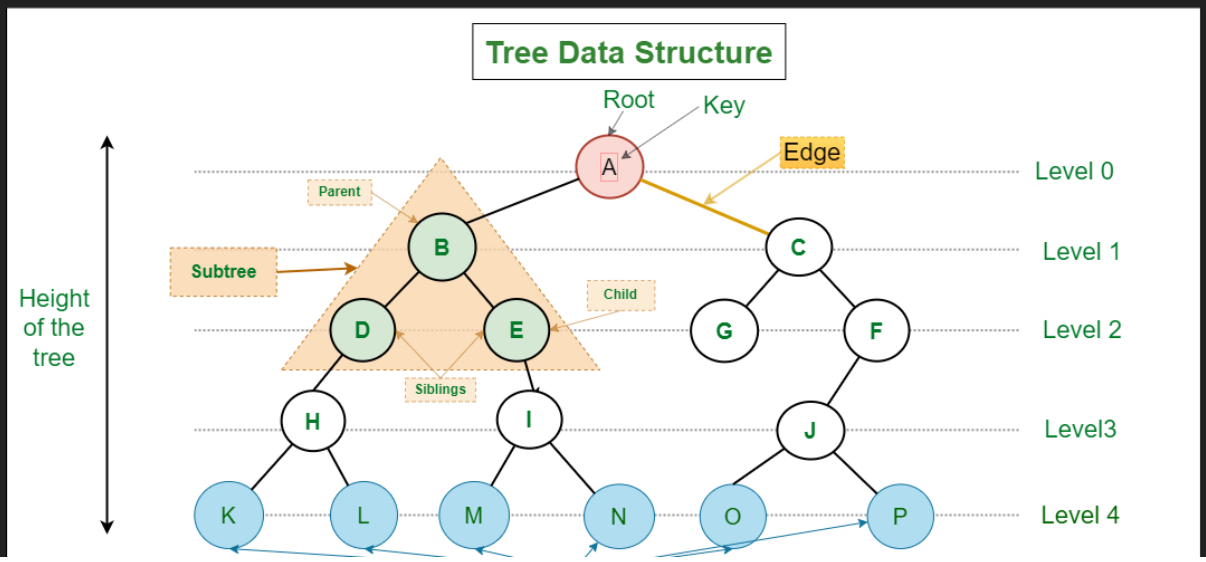
- `Input Restricted Queue` : This is a simple queue. In this type of queue, the input can be taken from only one end but deletion can be done from any of the ends.
- `Output Restricted Queue` : This is also a simple queue. In this type of queue, the input can be taken from both ends but deletion can be done from only one end.
- `Circular Queue` : This is a special type of queue where the last position is connected back to the first position. Here also the operations are performed in FIFO order. To know more refer this.
- `Double-Ended Queue (Dequeue)` : In a double-ended queue the insertion and deletion operations, both can be performed from both ends. To know more refer this.
- `Priority Queue` : A priority queue is a special queue where the elements are accessed based on the priority assigned to them. To know more refer this.

Basic Operations for Queue in Data Structure:

- Enqueue() – Adds (or stores) an element to the end of the queue..
- Dequeue() – Removal of elements from the queue.
- Peek() or front()- Acquires the data element available at the front node of the queue without deleting it.
- rear() – This operation returns the element at the rear end without removing it.
- isFull() – Validates if the queue is full.
- isNull() – Checks if the queue is empty.

`Tree` :

- A tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search.
- It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.
- The topmost node of the tree is called the root, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.



Basic Terminologies In Tree Data Structure:

- `Parent Node` : The node which is a predecessor of a node is called the parent node of that node.
- `Child Node` : The node which is the immediate successor of a node is called the child node of that node.
- `Root Node` : The topmost node of a tree or the node which does not have any parent node is called the root node.
- `Leaf Node or External Node` : The nodes which do not have any child nodes are called leaf nodes.
- `Ancestor of a Node` : Any predecessor nodes on the path of the root to that node are called Ancestors of that node.
- `Descendant` : Any successor node on the path from the leaf node to that node.
- `Sibling` : Children of the same parent node are called siblings.
- `Level of a node` : The count of edges on the path from the root node to that node.
- `Internal node` : A node with at least one child is called Internal Node.
- `Neighbour of a Node` : Parent or child nodes of that node are called neighbors of that node.
- `Subtree` : Any node of the tree along with its descendant.

Basic Operation Of Tree Data Structure:

- **Create – create a tree in the data structure.**
- **Insert – Inserts data in a tree.**
- **Search – Searches specific data in a tree to check whether it is present or not.**
- **Traversal:**

Depth-First Traversals:

- `Preorder Traversal` : In preorder traversal, you visit the root node first, then recursively traverse the left subtree, followed by the right subtree.
- `Inorder Traversal` : In inorder traversal, you recursively traverse the left subtree, visit the root node, and then traverse the right subtree. In binary search trees (BSTs), inorder traversal visits nodes in sorted order.
- `Postorder Traversal` : In postorder traversal, you recursively traverse the left and right subtrees before visiting the root node.

Breadth-First Traversal:

- `Level Order Traversal` : Level order traversal visits nodes level by level, from left to right. It starts at the root, then visits all nodes at depth 1, followed by nodes at depth 2, and so on.

```
Example:-
          A
         / \
        B   C
       / \   \
      D   E   F

Preorder Traversal: A - B - D - E - C - F

Inorder Traversal: D - B - E - A - C - F

Postorder Traversal: D - E - B - F - C - A

Level Order Traversal: A - B - C - D - E - F
```

Types of Tree

1. Binary Tree : A Binary Tree is a type of tree data structure where each node can have a maximum of two child nodes, a left child node and a right child node.

- A `balanced Binary` Tree has at most 1 in difference between its left and right subtree heights, for each node in the tree.
- A `complete Binary` Tree has all levels full of nodes, except the last level, which is can also be full, or filled from left to right. The properties of a complete Binary Tree means it is also balanced.
- A `full Binary` Tree is a kind of tree where each node has either 0 or 2 child nodes.
- A `perfect Binary` Tree has all leaf nodes on the same level, which means that all levels are full of nodes, and all internal nodes have two child nodes.The properties of a perfect Binary Tree means it is also full, balanced, and complete.

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

// Define the structure for a binary tree node
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int val) : data(val), left(nullptr), right(nullptr) {}
};

// Binary Tree ADT
class BinaryTree {
private:
    TreeNode* root;

public:
    BinaryTree() : root(nullptr) {}

    // Function to insert a value into the binary tree
    void insert(int val) {
```

```
    root = insertRecursive(root, val);
}

// Function to perform recursive insertion
TreeNode* insertRecursive(TreeNode* node, int val) {
    if (node == nullptr) {
        return new TreeNode(val);
    }

    if (val < node->data) {
        node->left = insertRecursive(node->left, val);
    } else if (val > node->data) {
        node->right = insertRecursive(node->right, val);
    }

    return node;
}

// Function to search for a value in the binary tree
bool search(int val) {
    return searchRecursive(root, val);
}

// Function to perform recursive search
bool searchRecursive(TreeNode* node, int val) {
    if (node == nullptr) {
        return false;
    }

    if (val == node->data) {
        return true;
    } else if (val < node->data) {
        return searchRecursive(node->left, val);
    } else {
        return searchRecursive(node->right, val);
    }
}

// Function to perform inorder traversal of the binary tree
void inorderTraversal() {
    inorderRecursive(root);
    cout << endl;
}

// Function to perform recursive inorder traversal
void inorderRecursive(TreeNode* node) {
    if (node == nullptr) {
        return;
    }

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

// Function to delete a value from the binary tree
void remove(int val) {
    root = removeRecursive(root, val);
}

// Function to perform recursive deletion
TreeNode* removeRecursive(TreeNode* node, int val) {
    if (node == nullptr) {
        return node;
    }

    if (val < node->data) {
        node->left = removeRecursive(node->left, val);
    } else if (val > node->data) {
        node->right = removeRecursive(node->right, val);
    } else {
        if (node->left == nullptr) {
            TreeNode* temp = node->right;
            delete node;
            return temp;
        } else if (node->right == nullptr) {
            TreeNode* temp = node->left;
            delete node;
            return temp;
        }

        TreeNode* temp = minValueNode(node->right);
        node->data = temp->data;
        node->right = removeRecursive(node->right, temp->data);
    }

    return node;
}

// Function to find the node with the minimum value in a subtree
TreeNode* minValueNode(TreeNode* node) {
    TreeNode* current = node;
    while (current && current->left != nullptr) {
        current = current->left;
    }
    return current;
}

// Function to check if the binary tree is
```

```
    // Function to check if the binary tree is empty
    bool isEmpty() const {
        return root == nullptr;
    }

    // Function to get the height of the binary tree
    int height() const {
        return heightRecursive(root);
    }

    // Function to perform recursive height calculation
    int heightRecursive(TreeNode* node) const {
        if (node == nullptr) {
            return -1;
        }

        int leftHeight = heightRecursive(node->left);
        int rightHeight = heightRecursive(node->right);

        return 1 + max(leftHeight, rightHeight);
    }
};

int main() {
    BinaryTree tree;

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

    cout << "Inorder traversal: ";
    tree.inorderTraversal();

    cout << "Height of the tree: " << tree.height() << endl;

    int searchVal = 40;
    if (tree.search(searchVal)) {
        cout << searchVal << " is present in the tree." << endl;
    } else {
        cout << searchVal << " is not present in the tree." << endl;
    }

    int deleteVal = 30;
    tree.remove(deleteVal);
    cout << "Inorder traversal after removing " << deleteVal << ": ";
    tree.inorderTraversal();

    return 0;
}
```

Implement binary tree using array

Binary Tree can be stored in an Array starting with the root node R on index 0. The rest of the tree can be built by taking a node stored on index i , and storing its left child node on index $2 \cdot i + 1$, and its right child node on index $2 \cdot i + 2$.

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

const int MAX_SIZE = 100;

class BinaryTree {
private:
    int arr[MAX_SIZE];
    int root;

public:
    BinaryTree() {
        root = 0;
        for (int i = 0; i < MAX_SIZE; i++) {
            arr[i] = -1;  // Initialize all elements to -1 (empty)
        }
    }

    void insert(int data) {
        if (arr[root] == -1) {
            arr[root] = data;  // Insert data at root if it's empty
        } else {
            int current = root;
            bool inserted = false;
            while (!inserted) {
                if (data <= arr[current]) {
                    if (arr[2 * current + 1] == -1) {
                        arr[2 * current + 1] = data;
                        inserted = true;
                    } else {
                        current = 2 * current + 1;
                    }
                } else {
                    if (arr[2 * current + 2] == -1) {
                        arr[2 * current + 2] = data;
                        inserted = true;
                    } else {
                        current = 2 * current + 2;
```

```
                }
            }
        }
    }
}

void display() {
    cout << "Binary Tree (Array representation):\n";
    for (int i = 0; i < MAX_SIZE; i++) {
        if (arr[i] != -1) {
            cout << arr[i] << " ";
        }
    }
    cout << "\n";
}
};

int main() {
    BinaryTree tree;
    tree.insert(5);
    tree.insert(3);
    tree.insert(8);
    tree.insert(2);
    tree.insert(4);
    tree.insert(7);
    tree.insert(9);

    tree.display();

    return 0;
}
```

2. Binary-Search Tree : A Binary Search Tree is a Binary Tree where every node's left child has a lower value, and every node's right child has a higher value.

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

// Define the structure of a tree node
struct TreeNode {
    int key;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int k) : key(k), left(nullptr), right(nullptr) {}
};

// Binary Search Tree ADT
class BinarySearchTree {
private:
    TreeNode* root;

    // Helper function to insert a key into the tree
    TreeNode* insertRec(TreeNode* root, int key) {
        if (root == nullptr) {
            return new TreeNode(key);
        }
        if (key < root->key) {
            root->left = insertRec(root->left, key);
        } else if (key > root->key) {
            root->right = insertRec(root->right, key);
        }
        return root;
    }

    // Helper function to perform an inorder traversal
    void inorderRec(TreeNode* root) {
        if (root != nullptr) {
            inorderRec(root->left);
            cout << root->key << " ";
            inorderRec(root->right);
        }
    }

public:
    BinarySearchTree() : root(nullptr) {}

    // Public method to insert a key into the tree
    void insert(int key) {
        root = insertRec(root, key);
    }

    // Public method to perform an inorder traversal of the tree
    void inorder() {
        inorderRec(root);
        cout << endl;
    }
};

// Main function to test the Binary Search Tree ADT
int main() {
    BinarySearchTree bst;

    // Insert keys into the tree
    bst.insert(50);
    bst.insert(30);
    bst.insert(20);
    bst.insert(40);
```

```
    bst.insert(70);
    bst.insert(60);
    bst.insert(80);

    // Perform an inorder traversal to print the keys in sorted order
    cout << "Inorder traversal of the BST:" << endl;
    bst.inorder();

    return 0;
}
```

3. AVL Tree : AVL trees are self-balancing, which means that the tree height is kept to a minimum so that a very fast runtime is guaranteed for searching, inserting and deleting nodes, with time complexity O(logn).

- **where the balance factor of each node (the height difference between its left and right subtrees) is limited to {-1, 0, 1} to maintain balance.**
- **The Balance Factor (BF) for a node (X) is the difference in height between its right and left subtrees.**

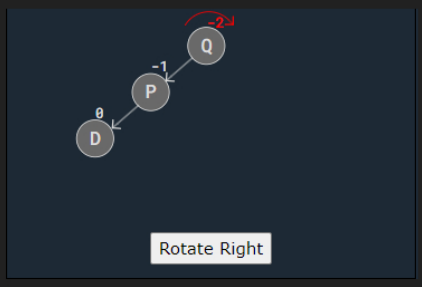
`BF(X) = height(rightSubtree(X))-height(leftSubtree(X))`

The Four "out-of-balance" Cases

- When the balance factor of just one node is less than -1, or more than 1, the tree is regarded as out of balance, and a rotation is needed to restore balance.
- There are four different ways an AVL Tree can be out of balance, and each of these cases require a different rotation operation.

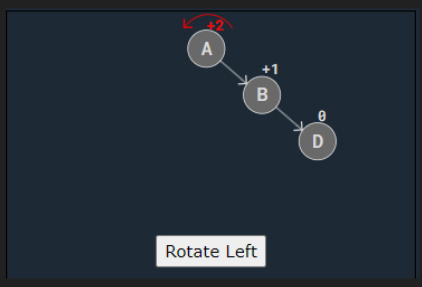
Case	Description	Rotation to Restore Balance
Left-Left (LL)	The unbalanced node and its left child node are both left-heavy.	A single right rotation.
Right-Right (RR)	The unbalanced node and its right child node are both right-heavy.	A single left rotation.
Left-Right (LR)	The unbalanced node is left heavy, and its left child node is right heavy.	First do a left rotation on the left child node, then do a right rotation on the unbalanced node.
Right-Left (RL)	The unbalanced node is right heavy, and its right child node is left heavy.	First do a right rotation on the right child node, then do a left rotation on the unbalanced node.

LL rotation



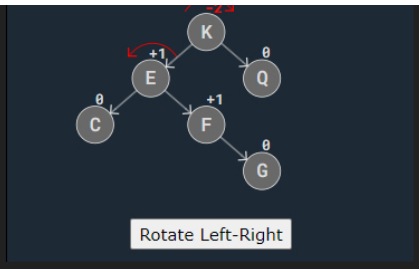

```
TreeNode* leftRotate(TreeNode* node) {
    TreeNode* newRoot = node->right;
    node->right = newRoot->left;
    newRoot->left = node;
    return newRoot;
}
```

RR rotation



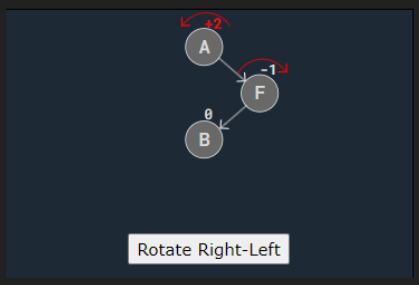

```
TreeNode* rightRotate(TreeNode* node) {
    TreeNode* newRoot = node->left;
    node->left = newRoot->right;
    newRoot->right = node;
    return newRoot;
}
```

LR rotation




```
TreeNode* leftRightRotate(TreeNode* node) {
    node->left = leftRotate(node->left);
    return rightRotate(node);
}
```

RL rotation




```
TreeNode* rightLeftRotate(TreeNode* node) {
    node->right = rightRotate(node->right);
    return leftRotate(node);
}
```

4. n-ary Tree : An N-ary tree is a tree data structure where each node can have at most N children. The value of N determines the degree of the tree.

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

// Define the structure of an N-ary tree node
struct NaryTreeNode {
    char data;
    vector<NaryTreeNode*> children;
    NaryTreeNode(char d) : data(d) {}
};

// Function to create a sample N-ary tree
NaryTreeNode* createNaryTree() {
    NaryTreeNode* root = new NaryTreeNode('A');
    root->children.push_back(new NaryTreeNode('B'));
    root->children.push_back(new NaryTreeNode('C'));
    root->children.push_back(new NaryTreeNode('D'));
    root->children[0]->children.push_back(new NaryTreeNode('E'));
    root->children[0]->children.push_back(new NaryTreeNode('F'));
    root->children[2]->children.push_back(new NaryTreeNode('G'));
    return root;
}

// Function to perform a preorder traversal of the N-ary tree
void preorderTraversal(NaryTreeNode* root) {
    if (root == nullptr) return;
    cout << root->data << " ";
    for (NaryTreeNode* child : root->children) {
        preorderTraversal(child);
    }
}

int main() {
    NaryTreeNode* root = createNaryTree();
    cout << "Preorder traversal of the N-ary tree: ";
    preorderTraversal(root);
    cout << endl;
    return 0;
}
```

5. Red-Black Tree : A red-black tree is a balanced binary search tree where each node is either red or black.

It maintains five properties to ensure balance:

- Every node is either red or black.
- The root is always black.
- Red nodes cannot have red children.
- Every path from a node to its descendant null nodes must have the same number of black nodes.
- Null nodes are considered black.

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

```
using namespace std;

enum Color { RED, BLACK };

// Define the structure of a Red-Black tree node
struct RedBlackTreeNode {
    int data;
    Color color;
    RedBlackTreeNode* left;
    RedBlackTreeNode* right;
    RedBlackTreeNode* parent;
    RedBlackTreeNode(int d) : data(d), color(RED), left(nullptr), right(nullptr), parent(nullptr) {}
};

// Function to perform a left rotation in a Red-Black tree
void leftRotate(RedBlackTreeNode*& root, RedBlackTreeNode* x) {
    RedBlackTreeNode* y = x->right;
    x->right = y->left;
    if (y->left != nullptr) {
        y->left->parent = x;
    }
    y->parent = x->parent;
    if (x->parent == nullptr) {
        root = y;
    } else if (x == x->parent->left) {
        x->parent->left = y;
    } else {
        x->parent->right = y;
    }
    y->left = x;
    x->parent = y;
}

// Function to perform a right rotation in a Red-Black tree
void rightRotate(RedBlackTreeNode*& root, RedBlackTreeNode* y) {
    RedBlackTreeNode* x = y->left;
    y->left = x->right;
    if (x->right != nullptr) {
        x->right->parent = y;
    }
    x->parent = y->parent;
    if (y->parent == nullptr) {
        root = x;
    } else if (y == y->parent->left) {
        y->parent->left = x;
    } else {
        y->parent->right = x;
    }
    x->right = y;
    y->parent = x;
}

// Function to fix violations after inserting a node in a Red-Black tree
void fixInsert(RedBlackTreeNode*& root, RedBlackTreeNode* z) {
    while (z != root && z->parent->color == RED) {
        if (z->parent == z->parent->parent->left) {
            RedBlackTreeNode* y = z->parent->parent->right;
            if (y != nullptr && y->color == RED) {
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->right) {
                    z = z->parent;
                    leftRotate(root, z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                rightRotate(root, z->parent->parent);
            }
        } else {
            RedBlackTreeNode* y = z->parent->parent->left;
            if (y != nullptr && y->color == RED) {
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->left) {
                    z = z->parent;
                    rightRotate(root, z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                leftRotate(root, z->parent->parent);
            }
        }
    }
    root->color = BLACK;
}

// Function to insert a node into a Red-Black tree
void insertNode(RedBlackTreeNode*& root, int data) {
    RedBlackTreeNode* z = new RedBlackTreeNode(data);
    RedBlackTreeNode* y = nullptr;
    RedBlackTreeNode* x = root;
    while (x != nullptr) {
```

```
        if (z->data < x->data) {
            x = x->left;
        } else {
            x = x->right;
        }
    }
    z->parent = y;
    if (y == nullptr) {
        root = z;
    } else if (z->data < y->data) {
        y->left = z;
    } else {
        y->right = z;
    }
    z->left = nullptr;
    z->right = nullptr;
    z->color = RED;
    fixInsert(root, z);
}

// Function to perform an inorder traversal of a Red-Black tree
void inorderTraversal(RedBlackTreeNode* root) {
    if (root != nullptr) {
        inorderTraversal(root->left);
        cout << root->data << " ";
        inorderTraversal(root->right);
    }
}

int main() {
    RedBlackTreeNode* root = nullptr;
    insertNode(root, 10);
    insertNode(root, 20);
    insertNode(root, 30);
    insertNode(root, 40);
    insertNode(root, 50);

    cout << "Inorder traversal of the Red-Black tree: ";
    inorderTraversal(root);
    cout << endl;

    return 0;
}
```

6. B-trees : B-trees are balanced search trees that generalize binary search trees to allow for multiple keys and children per node. `They are commonly used in databases and file systems due to their ability to handle large amounts of data efficiently.`

A B-tree of order M has the following properties:

- Each node can have at most M children.
- Each internal node (except the root) has at least ⌈M/2⌉ children.
- All leaves are at the same level.

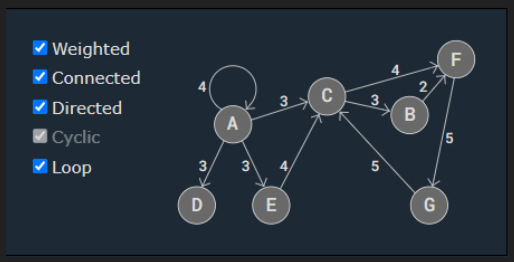
```
#include <iostream>
#include <vector>
using namespace std;

const int MAX_KEYS = 3; // Order of the B-tree

// Define the structure of a B-tree node
struct BTreeNode {
    vector<int> keys;
    vector<BTreeNode*> children;
    BTreeNode* parent;
    BTreeNode() {
        parent = nullptr;
        keys.resize(MAX_KEYS, 0);
        children.resize(MAX_KEYS + 1, nullptr);
    }
};

// Function to create a sample B-tree
BTreeNode* createBTree() {
    BTreeNode* root = new BTreeNode();
    root->keys = {10, 20};
    root->children[0] = new BTreeNode();
    root->children[0]->keys = {5, 8};
    root->children[1] = new BTreeNode();
    root->children[1]->keys = {12, 15};
    root->children[2] = new BTreeNode();
    root->children[2]->keys = {25, 30};
    return root;
}

// Function to perform an inorder traversal of a B-tree
void inorderTraversal(BTreeNode* root) {
    if (root != nullptr) {
        for (int i = 0; i < root->keys.size(); ++i) {
            inorderTraversal(root->children[i]);
            cout << root->keys[i] << " ";
        }
        inorderTraversal(root->children[root->keys.size()]);
    }
}
```

```
int main() {
    BTreeNode* root = createBTree();
    cout << "Inorder traversal of the B-tree: ";
    inorderTraversal(root);
    cout << endl;
    return 0;
}
```

8. Huffman Trees: Huffman trees are used in data compression algorithms, such as Huffman coding. They are a type of binary tree used for encoding characters based on their frequency of occurrence, with more frequent characters having shorter codes.

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

// Define the structure of a Huffman tree node
struct HuffmanTreeNode {
    char data;
    int frequency;
    HuffmanTreeNode *left, *right;
    HuffmanTreeNode(char d, int freq) : data(d), frequency(freq), left(nullptr), right(nullptr) {}
};

// Comparison function for the priority queue
struct Compare {
    bool operator()(HuffmanTreeNode* a, HuffmanTreeNode* b) {
        return a->frequency > b->frequency;
    }
};

// Function to build a Huffman tree from given character frequencies
HuffmanTreeNode* buildHuffmanTree(vector<char> characters, vector<int> frequencies) {
    priority_queue<HuffmanTreeNode*, vector<HuffmanTreeNode*>, Compare> pq;
    for (int i = 0; i < characters.size(); ++i) {
        pq.push(new HuffmanTreeNode(characters[i], frequencies[i]));
    }
    while (pq.size() > 1) {
        HuffmanTreeNode* left = pq.top(); pq.pop();
        HuffmanTreeNode* right = pq.top(); pq.pop();
        HuffmanTreeNode* parent = new HuffmanTreeNode('$', left->frequency + right->frequency);
        parent->left = left;
        parent->right = right;
        pq.push(parent);
    }
    return pq.top();
}

// Function to perform a preorder traversal of the Huffman tree
void preorderTraversal(HuffmanTreeNode* root) {
    if (root != nullptr) {
        cout << root->data << " ";
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}

int main() {
    vector<char> characters = {'A', 'B', 'C', 'D', 'E', 'F'};
    vector<int> frequencies = {5, 9, 12, 13, 16, 45};

    HuffmanTreeNode* root = buildHuffmanTree(characters, frequencies);

    cout << "Preorder traversal of the Huffman tree: ";
    preorderTraversal(root);
    cout << endl;

    return 0;
}
```

`Graph`: A Graph is a non-linear data structure consisting of vertices and edges. The graph is denoted by G(V,E).



Types Of Graph

```
1. Undirected Graph : In an undirected graph, edges have no direction. The connection between nodes is bidirectional. If there's an edge between node A and node
B, it implies that there is a relationship between A and B, and vice versa.

2. Directed Graph (Digraph) : In a directed graph, edges have a direction. This means the relationship between nodes is one-way. If there's an edge from node A
to node B, it doesn't necessarily imply the existence of an edge from B to A.
```

3. Weighted Graph : Weighted graphs assign a numerical value (weight) to each edge. These weights often represent the cost, distance, or any other relevant measure associated with traversing from one node to another.

4. Unweighted Graph : In contrast to weighted graphs, unweighted graphs do not assign any numerical value to the edges. They only represent the presence or absence of a connection between nodes.

5. Cyclic Graph : A cyclic graph contains at least one cycle, meaning there is a path that starts and ends at the same node.Cycles can be of varying lengths and can involve multiple nodes.

6. Eulerian Graphs: An Eulerian graph is a graph that contains a cycle that traverses every edge exactly once.

7. Acyclic Graph : An acyclic graph does not contain any cycles. This means there are no paths that loop back to the same node.

8. Connected Graph : A connected graph is one in which there is a path between every pair of nodes. In other words, there are no isolated nodes or subgraphs within the graph.

9. Disconnected Graph : A disconnected graph consists of two or more connected components (subgraphs), where there is no path between nodes in different components.

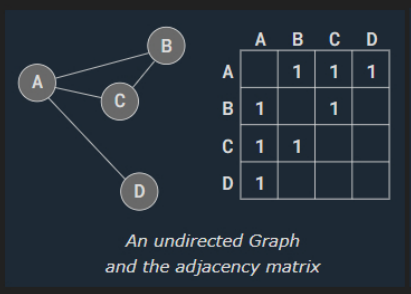
10. Complete Graph : In a complete graph, every pair of distinct nodes is connected by a unique edge. This means there is an edge between every pair of nodes in the graph.

11. Bipartite Graph : A bipartite graph is one whose vertices can be divided into two disjoint sets such that every edge connects a vertex from one set to a vertex in the other set. There are no edges connecting vertices within the same set.

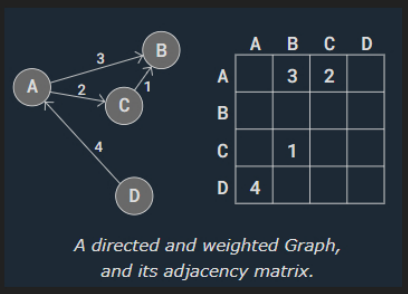
Representation of Graphs

There are two ways to store a graph:

- Adjacency Matrix



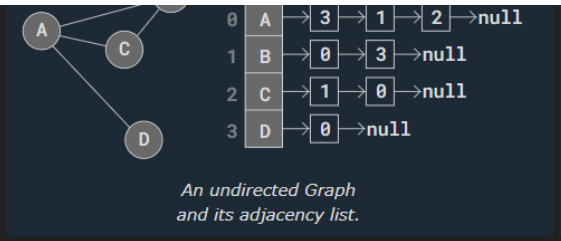
An undirected Graph and the adjacency matrix



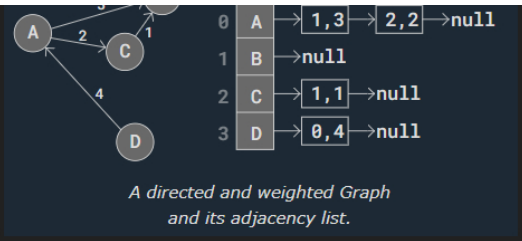
A directed and weighted Graph, and its adjacency matrix.

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

class GraphAdjMatrix {
private:
    int vertices;
    vector<vector<int>> adjMatrix;

public:
    GraphAdjMatrix(int V) : vertices(V) {
        adjMatrix.resize(vertices, vector<int>(vertices, 0));
    }

    void addEdge(int src, int dest) {
        adjMatrix[src][dest] = 1;
        adjMatrix[dest][src] = 1; // Uncomment for undirected graph
    }

    void display() {
        for (int i = 0; i < vertices; ++i) {
            for (int j = 0; j < vertices; ++j) {
                cout << adjMatrix[i][j] << " ";
            }
            cout << endl;
        }
    }
};

int main() {
    int V = 5; // Number of vertices
    GraphAdjMatrix graph(V);

    graph.addEdge(0, 1);
    graph.addEdge(0, 4);
    graph.addEdge(1, 2);
    graph.addEdge(1, 3);
    graph.addEdge(1, 4);
    graph.addEdge(2, 3);
    graph.addEdge(3, 4);

    cout << "Adjacency Matrix:" << endl;
    graph.display();

    return 0;
}
```

- Adjacency List



An undirected Graph and its adjacency list.



A directed and weighted Graph and its adjacency list.

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

class GraphAdjList {
private:
    int vertices;
    vector<vector<int>> adjList;

public:
    GraphAdjList(int V) : vertices(V) {
        adjList.resize(vertices);
    }

    void addEdge(int src, int dest) {
        adjList[src].push_back(dest);
        adjList[dest].push_back(src); // Uncomment for undirected graph
    }

    void display() {
        for (int i = 0; i < vertices; ++i) {
            cout << "Vertex " << i << " -> ";
            for (int j = 0; j < adjList[i].size(); ++j) {
                cout << adjList[i][j] << " ";
            }
            cout << endl;
        }
    }
};

int main() {
    int V = 5; // Number of vertices
    GraphAdjList graph(V);

    graph.addEdge(0, 1);
    graph.addEdge(0, 4);
    graph.addEdge(1, 2);
    graph.addEdge(1, 3);
    graph.addEdge(1, 4);
    graph.addEdge(2, 3);
    graph.addEdge(3, 4);

    cout << "Adjacency List:" << endl;
    graph.display();

    return 0;
}
```

Graph Traversals

- Breadth-First Search :
  - Breadth-First Search is an algorithm used for traversing or searching tree or graph data structures.
  - BFS works by visiting nodes level by level. It uses a queue data structure to keep track of nodes to be visited next.
  - The algorithm starts at the root node (or a specified starting node) and visits its neighbors first, then the neighbors' neighbors, and so on.

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

void bfs(vector<vector<int>>& graph, int startNode) {
    int numNodes = graph.size();
    vector<bool> visited(numNodes, false);
    queue<int> q;

    visited[startNode] = true;
    q.push(startNode);

    while (!q.empty()) {
        int currentNode = q.front();
        q.pop();
        cout << currentNode << " ";

        for (int neighbor : graph[currentNode]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}
```

```
int main() {
    int numNodes = 6;
    vector<vector<int>> graph(numNodes);
    graph[0] = {1, 2};
    graph[1] = {0, 3, 4};
    graph[2] = {0, 4};
    graph[3] = {1, 5};
    graph[4] = {1, 2, 5};
    graph[5] = {3, 4};

    cout << "BFS traversal starting from node 0: ";
    bfs(graph, 0);
    cout << endl;

    return 0;
}
```

- Depth-First Search :
  - Depth-First Search is another algorithm used for traversing or searching tree or graph data structures.
  - Unlike BFS, which explores the neighbor nodes first, DFS explores as far as possible along each branch before backtracking.
  - DFS works by visiting nodes depth by depth. It uses a stack data structure (or recursion) to keep track of nodes to be visited next.
  - The algorithm starts at the root node (or a specified starting node) and explores as far as possible along each branch before backtracking.

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

void dfsUtil(vector<vector<int>>& graph, vector<bool>& visited, int currentNode) {
    visited[currentNode] = true;
    cout << currentNode << " ";

    for (int neighbor : graph[currentNode]) {
        if (!visited[neighbor]) {
            dfsUtil(graph, visited, neighbor);
        }
    }
}

void dfs(vector<vector<int>>& graph, int startNode) {
    int numNodes = graph.size();
    vector<bool> visited(numNodes, false);
    dfsUtil(graph, visited, startNode);
}

int main() {
    int numNodes = 6;
    vector<vector<int>> graph(numNodes);
    graph[0] = {1, 2};
    graph[1] = {0, 3, 4};
    graph[2] = {0, 4};
    graph[3] = {1, 5};
    graph[4] = {1, 2, 5};
    graph[5] = {3, 4};

    cout << "DFS traversal starting from node 0: ";
    dfs(graph, 0);
    cout << endl;

    return 0;
}
```

- `Cycle Detection` : **It is important to be able to detect cycles in Graphs because cycles can indicate problems or special conditions in many applications like networking, scheduling, and circuit design.**

The two most common ways to detect cycles are:

- Depth First Search (DFS): DFS traversal explores the Graph and marks vertices as visited. A cycle is detected when the current vertex has an adjacent vertex that has already been visited.

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

class Graph {
private:
    int V; // Number of vertices
    vector<vector<int>> adj; // Adjacency list

    bool isCyclicUtil(int v, vector<bool>& visited, vector<bool>& recStack) {
        visited[v] = true;
        recStack[v] = true;

        for (int neighbor : adj[v]) {
            if (!visited[neighbor] && isCyclicUtil(neighbor, visited, recStack)) {
                return true;
            } else if (recStack[neighbor]) {
                return true;
            }
        }

        recStack[v] = false;
        return false;
    }
```

```
public:
    Graph(int vertices) : V(vertices), adj(vertices) {}

    void addEdge(int u, int v) {
        adj[u].push_back(v);
    }

    bool isCyclic() {
        vector<bool> visited(V, false);
        vector<bool> recStack(V, false);

        for (int i = 0; i < V; ++i) {
            if (!visited[i] && isCyclicUtil(i, visited, recStack)) {
                return true;
            }
        }

        return false;
    }
};

int main() {
    Graph g(4); // Create a graph with 4 vertices
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    if (g.isCyclic()) {
        cout << "Graph contains cycle\n";
    } else {
        cout << "Graph does not contain cycle\n";
    }

    return 0;
}
```

- Union-Find: This works by initially defining each vertex as a group, or a subset. Then these groups are joined for every edge. Whenever a new edge is explored, a cycle is detected if two vertices already belong to the same group.

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

class UnionFind {
private:
    vector<int> parent;
    vector<int> rank;

public:
    UnionFind(int n) {
        parent.resize(n);
        rank.resize(n);
        for (int i = 0; i < n; ++i) {
            parent[i] = i;
            rank[i] = 0;
        }
    }

    int find(int u) {
        if (u != parent[u]) {
            parent[u] = find(parent[u]);
        }
        return parent[u];
    }

    void unionSets(int u, int v) {
        int rootU = find(u);
        int rootV = find(v);
        if (rootU != rootV) {
            if (rank[rootU] < rank[rootV]) {
                parent[rootU] = rootV;
            } else if (rank[rootU] > rank[rootV]) {
                parent[rootV] = rootU;
            } else {
                parent[rootV] = rootU;
                rank[rootU]++;
            }
        }
    }
};

bool isCyclic(vector<pair<int, int>>& edges, int n) {
    UnionFind uf(n);
    for (auto edge : edges) {
        int u = edge.first;
        int v = edge.second;
        int rootU = uf.find(u);
        int rootV = uf.find(v);
        if (rootU == rootV) {
            return true; // Cycle detected
        }
        uf.unionSets(u, v);
    }
    return false; // No cycle found
```

```
}

int main() {
    int n = 4; // Number of vertices
    vector<pair<int, int>> edges = {{0, 1}, {1, 2}, {2, 3}, {3, 0}}; // Example edges

    if (isCyclic(edges, n)) {
        cout << "Graph contains cycle\n";
    } else {
        cout << "Graph does not contain cycle\n";
    }

    return 0;
}
```

Shortest Path Algorithms : the shortest path problem means to find the shortest possible route or path between two vertices (or nodes) in a Graph.

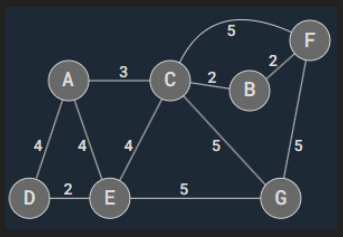
- `Dijkstra's Algorithm` :
  - Dijkstra's algorithm is used for solving single-source shortest path problems for directed or undirected paths. Single-source means that one vertex is chosen to be the start, and the algorithm will find the shortest path from that vertex to all other vertices.
  - Dijkstra's algorithm does not work for graphs with negative edges. For graphs with negative edges,

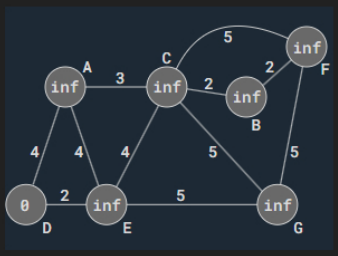
How it works:

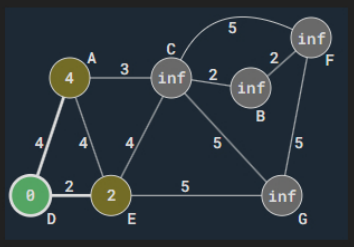
1. Set initial distances for all vertices: 0 for the source vertex, and infinity for all the other.
2. Choose the unvisited vertex with the shortest distance from the start to be the current vertex. So the algorithm will always start with the source as the current vertex.
3. For each of the current vertex's unvisited neighbor vertices, calculate the distance from the source and update the distance if the new, calculated, distance is lower.
4. We are now done with the current vertex, so we mark it as visited. A visited vertex is not checked again.
5. Go back to step 2 to choose a new current vertex, and keep repeating these steps until all vertices are visited.
6. In the end we are left with the shortest path from the source vertex to every other vertex in the graph.

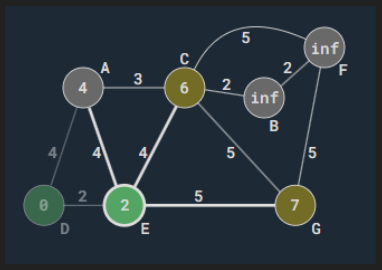
visually implementation

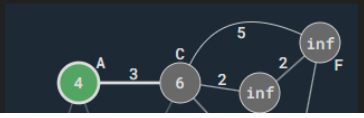
Given graph

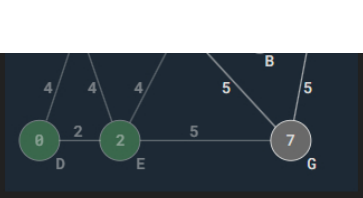


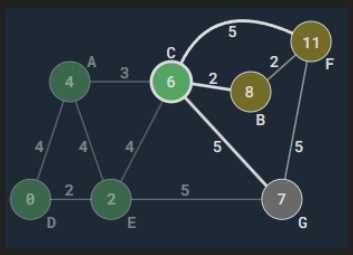


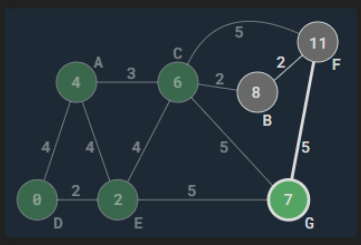


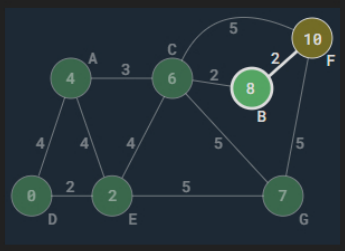












implement in cpp

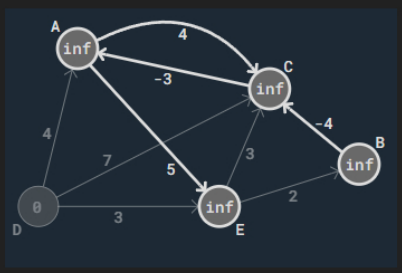
- `Bellman-Ford Algorithm` :
  - The Bellman-Ford algorithm is best suited to find the shortest paths in a directed graph, with one or more negative edge weights, from the source vertex to all other vertices.

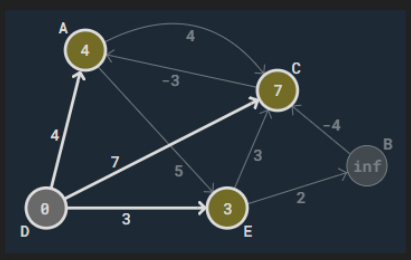
How it works:

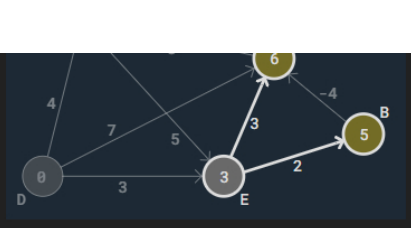
1. Set initial distance to zero for the source vertex, and set initial distances to infinity for all other vertices.
2. For each edge, check if a shorter distance can be calculated, and update the distance if the calculated distance is shorter.
3. Check all edges (step 2) V−1 times. This is as many times as there are vertices (V), minus one.
4. Optional: Check for negative cycles. This will be explained in better detail later.

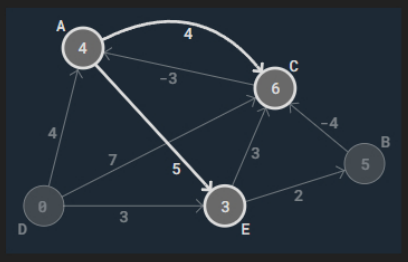
Visually Implementation Steps:

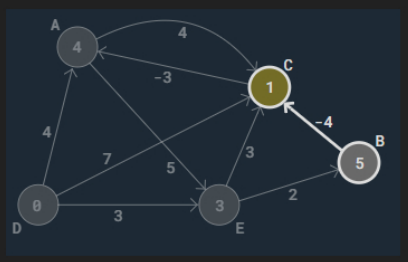
staring vertex is D

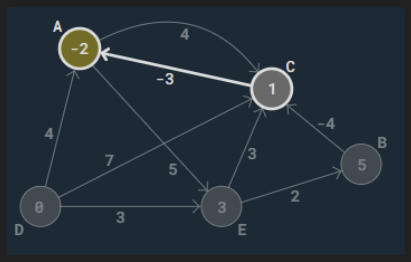












- `Floyd Warshall` :
  - The Floyd Warshall Algorithm is an all pair shortest path algorithm unlike Dijkstra and Bellman Ford which are single source shortest path algorithms. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

Minimum Spanning Tree

- Prim's Algorithm
- Kruskal's Algorithm

Topological Sorting.

Graph Coloring and Matching.

Network Flow Algorithms

- Ford-Fulkerson Algorithm
- Edmonds-Karp Algorithm

Sorting :

- Bubble Sort
- Selection Sort
- Insertion Sort.
- Merge Sort
- Quick Sort
- Heap Sort.

Searching :

- Binary Search.
- Linear Search.