A group of cubes connected together

Description automatically generated

Task:01

Domain: Algorithm and Data-structure

Mehjabin Mostafa | Real-Time Processing with Divide and Conquer Algorithms in Big Data Applications

**Arrays and Vectors:**

* **Array:**

Array is a collection of data that is the same type. It means it stores multiple elements in the same kind of container or variable or memory location. It has a unique identifier called index which is used to accessing the elements of a particular array.

Such as:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **10** | **20** | **30** | **40** | **50** | **60** | **70** |

**0 1 2 3 4 5 6**

* It’s a 1D array named A[***8***] where,
* Indexes are defined in **Teal**
* Elements are defined in **Pink**
* Size is defined in **Red**

There’s a difference between **Size** and **Index.** Indexes will always start from **0** and size will start from **1.** Index will be smaller than the size if the array and their difference will be always 1. **Index** is a specific position, but **size** is the total count of positions. Size is the same as elements.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Number= | [ 99, | 77, | 88, | 87 ] |
| Index: | 0 | 1 | 2 | 3 |
| Size: | 4 |  |  |  |

**Declaration and Initialization:**

An Array should be declared by Data Type and the name of the Array and then its size.

Example:

*double Number[****10****];*

Here we are storing the number of 10 students in a Array for an individual course.

**Double:** Type of Data

**Number:** Name of the Array

**10**: Size of the Array.

After declaring the array, the next part is initializing. We can initialize the array by user input, or it can be done by direct initializing within the array.

Example:

**Direct Initializing:**

*double Number[10]={80,98.5,85,76,64,89,56.5,78,99,100};*

**User Input:**

*double Number[10];*

*cin>> Number[10];*

We can also initialize the array by using loops after declaration.

*for (int i = 0; i < N; i++) {  
 Number[i] = value;  
}*

We can also Traverse the Array using for loop.

Example:

*#include <iostream>*

*using namespace std;*

*int main(){*

*double Number [10] = { 80,98.5,85,76,64,89,56.5,78,99,100};*

*for (int i = 0; i < 10; i++) {*

*cout << Number[i] << endl;*

*}*

*return 0;*

*}*

**Size of an Array:**

We can find the size of an array by using some functions named sizeof(). At first, we have to find the whole size of the given array and then divide the size by the element which is stored.

*int size = sizeof(Number) / sizeof(Number [i]);*

**Pointers in Array:**

We can point an Array by using “ *\* ptr* = *name of the array* ” this syntax then it will show us the memory location of the array. It will be easier to access and handle the memory address.

Example:

*#include <iostream>*

*using namespace std;*

*int main()*

*{*

*// Defining an array*

*Double Number [] = { 80,98.5,85,76,64,89,56.5,78,99,100};*

*// Define a pointer*

*double \* ptr= Number;*

*// Printing address of the arrary using array name*

*cout << "Memory address of Number: " << &arr << endl;*

*// Printing address of the array using ptr*

*cout << "Memory address of Number: " << ptr << endl;*

*return 0;*

*}*

**Passing Array to Function:**

Normally we can pass the array to function as usual as variable in C++. But we can access the array by only using pointer. That’s why it’s better to pass the array as pointer to easily modify and access the array. For using pointers, it requires careful handling of array size.

Example:

*void printArray(int\*Number, int n) {*

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

*cout << Number[i] << endl;*

*}*

*}*

*int main() {*

*int Number[10] = {80,98.5,85,76,64,89,56.5,78,99,100};*

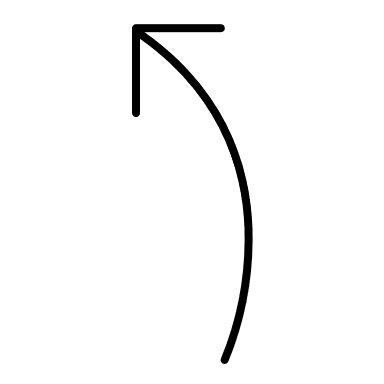
*printArray(numbers, 10); // Passes the array as a pointer*

*return 0;*

*}*

We can also pass the array by reference, but it works for only fixed sized array.

Example:

*void printArray(double (&Number) [10]) { // Array size must be specified*

*(referencing an array named Number)*

*for (int i = 0; i < 10; i++) {*

*cout << Number[i] << " ";*

*}*

*}*

*int main() {*

*double Number[10] = {80,98.5,85,76,64,89,56.5,78,99,100};*

*printArray(Number); // Passes the array by reference*

*return 0;*

*}*

Other than that, for flexibility we can use dynamic array which is called vector, and it works more effectively. But we will discuss this in our vector segment.

All the things that we were discussing previously were for static array which has only one dimension. An array can have many dimensions. But for easier to understand we will discuss 2D (dimension) array.

The basic operations are the same as the 1D array but if the array is bigger than the 1D array we must declare and initialize it first for the 2D array. Then some basic syntax will change. Other than that, all things are kind of the same.

**2D Array:**

Multidimensional arrays are generally represented by the rows and columns. For the declaration and initialization, we must mention how many rows and how many columns the array can have.

|  |  |  |  |
| --- | --- | --- | --- |
| row col | 0 | 1 | 2 |
| 0 | **5** | **7** | **9** |
| 1 | **11** | **13** | **15** |
| 2 | **17** | **19** | **21** |

Here we have standard 3 by 3 matrix as 2D array.

* **White** indicates the index of row and column.
* **Black** indicates the value of the index that are stored in the array.

**Initialization and Declaration:**

*int Number[3][3]= { 5,7,9,*

*11,13,15,*

*17,19,21 };*

We can declare and initialize at the same moment by doing this. If we want to take user input, we can do it separately or combinedly by for loop.

It requires only the initialization of row and column, after that maximum operations are the same. We can traverse the array, find the memory location, use function to pass the elements and many things.

**Vector:**

* Vector is a container object which can dynamically change its size as required.
* It’s basically an array with dynamically resizable memory with some changes in function and syntax.
* The declaration of vector is similar like array, but we must mention the vector in the very first library function.

Example:

*#include<iostream>*

***#include<vector>***

*using namespace std;*

*//function to print vector*

*void printVector(vector<int> P)*

*{*

*cout<<"|";*

*for(int i=0;i<P.size();i++)*

*{*

*cout<<P[i]<<"|";*

*}*

*}*

*int main(){*

***vector<int> V={10,20,30,40};***

*printVector(V);}*

* It can increase itself after adding new elements and can erase.

Example:

*#include<iostream>*

*#include<vector>*

*using namespace std;*

*//function to print vector*

*void printVector(vector<int> P)*

*{*

*cout<<"|";*

*for(int i=0;i<P.size();i++)*

*{*

*cout<<P[i]<<"|";*

*}*

*}*

*int main(){*

*vector<int> M={999,99999,9980,0};*

***M.insert(M.begin(),8);***

***M.insert(M.begin()+1,9);***

*cout<<M[1]<<endl;*

***M.erase(M.begin());***

*cout<<M[0]<<endl;*

*cout<<"-----------------"<<endl;}*

* It’s not always necessary to define its size. It will work fine without mentioning the size. And if anyone wants, they can define the size earlier.
* Interesting thing is, it can shrink its memory if the data no longer exists in that memory location. And it does this to save some space which is no longer needed.

Example:

*#include<iostream>*

*#include<vector>*

*using namespace std;*

*void printVector(vector<int> P)*

*{*

*cout<<"|";*

*for(int i=0;i<P.size();i++)*

*{*

*cout<<P[i]<<"|";*

*}*

*}*

*int main(){*

*vector<int> V={10,20,30,40};*

*cout<< V.size()<<endl;*

*V.push\_back(90);*

*V.pop\_back();*

***V.shrink\_to\_fit();(****if we didn’t use shrink to fit function, the output*

*Would show the capacity of 8bit.)*

*cout<< V.size()<<endl;*

*cout<< V.capacity()<<endl;*

*cout<<"------------------"<<endl;*

*printVector(V);*

*}*

**Output:(with shrink to fit)**

*4*

*4*

**4**

*------------------*

*|10|20|30|40|*

**Output:(without shrink to fit)**

*4*

*4*

***8***

*------------------*

*|10|20|30|40|*

**We can also use this segment of code for updating the value:**

*// Insert an element at the beginning*

*M.insert(M.begin(), 11);*

*// Print the updated vector*

*cout << "\nUpdated Vector after inserting 11 at the beginning:" << endl;*

***for(auto itr = M.begin(); itr != M.end(); itr++)***

*{ cout << \*itr << endl;*

* For an extra element it doubles its size every time.

Example:

*#include<iostream>*

*#include<vector>*

*using namespace std;*

*int main(){*

*vector<int> M={999,99999,9980,0};*

*M.insert(M.begin(),8);*

*M.insert(M.begin()+1,9);*

*cout<<M[1]<<endl;*

*M.erase(M.begin());*

*cout<<M[0]<<endl;*

*cout<<"-----------------"<<endl;*

**Output:**

*4*

***4***

***8***

*------------------*

* It uses some stack functions to add and remove elements like push and pop.

*#include<iostream>*

*#include<vector>*

*using namespace std;*

*void printVector(vector<int> P)*

*{ cout<<"|";*

*for(int i=0;i<P.size();i++)*

*{*

*cout<<P[i]<<"|";*

*}*

*}*

*int main(){*

*vector<int> V={10,20,30,40};*

*cout<< V.size()<<endl;*

***V.push\_back(90);***

***V.pop\_back();***

*cout<< V.size()<<endl;*

*cout<< V.capacity()<<endl;*

*cout<<"------------------"<<endl;*

*printVector(V);}*

**Difference between Array and Vector:**

**(Gist)**

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Operation | **Array Code** | **Vector Code** |
| Declaration | int arr[5] = {1, 2, 3, 4, 5}; | vector<int> vec = {1, 2, 3, 4, 5}; |
| Access Element | cout << arr[2]; | cout << vec[2]; |
| Modify Element | arr[2] = 10; | vec[2] = 10; |
| Traverse Elements | for (int i = 0; i < 5; i++) { cout << arr[i] << " "; } | for (int i = 0; i < vec.size(); i++) { cout << vec[i] << " "; } |
| Add Element | /\* Not supported \*/ | vec.push\_back(6); |
| Remove Element | /\* Not supported \*/ | vec.pop\_back(); |
| Resize | /\* Not supported \*/ | vec.resize(10); |
| Bounds-Checked Access | /\* Not supported \*/ | try { cout << vec.at(3); } catch (out\_of\_range &e) { cout << e.what(); } |
| Size of Collection | cout << sizeof(arr) / sizeof(arr[0]); | cout << vec.size(); |
| Clear Elements | /\* Reset values manually or reinitialize array \*/ | vec.clear(); |

|  |  |  |
| --- | --- | --- |
| Operation | Array | Vector |
| Add Element | Not possible (size is fixed). | push\_back() to add at the end. |
| Remove Element | Not possible. | pop\_back() to remove last e  lement. |
| Access with Bounds | No safety, use arr[index]. | Safe access with vec.at(index). |
| Resize | Not applicable. | Automatic or with resize(). |

**Graph Theory Fundamentals:**

Graph is a network that helps to define and visualize relationships between various components. Graph theory is all about the study of the properties of these types of networks, and how they can be used to model and solve a whole host of interesting problems.

**Key Terminologies:**

|  |  |
| --- | --- |
| Term | Description |
| Vertex | Every individual data element is called a vertex or a node. In the above image, the vertices are A, B, C, D & E. |
| Edge (Arc) | It is a connecting link between two nodes or vertices. Each edge has two ends and is represented as (startingVertex, and endingVertex). |
| Undirected Edge | It is a bidirectional edge. |
| Directed Edge | It is a unidirectional edge. |
| Weighted Edge | An edge with value (cost) on it. |
| Degree | The total number of edges connected to a vertex in a graph. |
| Indegree | The total number of incoming edges connected to a vertex. |
| Outdegree | The total number of outgoing edges connected to a vertex. |
| Self-loop | An edge is called a self-loop if its two endpoints coincide. |
| Adjacency | Vertices are said to be adjacent if an edge is connected. |

**5 6**

**Edge**

**6 9 Fig1: Undirected Graph**

**With weighted value(cyclic)**

**4**

**Vertex**

**5 6**



**6 9 Fig2: Directed Graph**



**With weighted value**



**4**



**Types of graphs:**

|  |  |  |
| --- | --- | --- |
| Graph Type | Description | Example |
| Undirected Graph | Edges have no direction; connections are bidirectional. | Social networks |
| Directed Graph (Digraph) | Edges have direction, represented with arrows. | Traffic systems |
| Weighted Graph | Edges have weights or costs. | Map navigation |
| Tree | A connected acyclic graph. | File systems |
| Cyclic | having a cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle. | Network Topology |
| Bipartite Graph | Vertices can be divided into two disjoint sets, and edges only connect vertices from different sets. | Matching problems |

**Basic Graph representation:**

**1. Adjacency Matrix:**

* Definition: A 2D array A[Vi][Vj] where:
  + A[i][j]=1 (or the weight, in weighted graphs) if there is an edge between vertex i and vertex j.
  + A[i][j]=0 if no edge exists.

A diagram of a graph

Description automatically generated

**2. Adjacency List:**

* Definition: An array (or list) of lists:
  + Each vertex has a list of all its adjacent vertices.
  + For weighted graphs, store tuples or pairs (neighbor,weight)(neighbor, weight)(neighbor,weight).

**A diagram of a number

Description automatically generated with medium confidence**

In this representation, every vertex of graph contains list of its adjacent vertices. The n rows of the adjacency matrix are represented as n chains. The nodes in chain I represent the vertices that are adjacent to vertex i. It can be represented in two forms. In one form, array is used to store n vertices and chain is used to store its adjacencies. So that we can access the adjacency list for any vertex in O(1) time. Adjlist[i] is a pointer to first node in the adjacency list for vertex i.

Structure is:

*#define MAX\_VERTICES 50*

*typedef struct node \*node\_pointer;*

*typedef struct node {*

*int vertex; struct node \*link; };*

*node\_pointer graph[MAX\_VERTICES];*

*int n=0; /\* vertices currently in use \*/*

Another type of representation is given below. example: consider the following directed graph representation implemented using linked list.

A diagram of a diagram

Description automatically generated with medium confidence

This representation can also be implemented using an array.

A diagram of a reference array

Description automatically generated

**Edge List**

* **Description**: A simple list of edges in the form of pairs (u,v)(u, v)(u,v) for unweighted graphs, or triplets (u,v,w)(u, v, w)(u,v,w) for weighted graphs.
  + **For undirected graphs**: Include both (u,v) and (v,u) or treat them as symmetric.
* **Formula**:

EdgeList={(u,v) ∣ (u,v) is an edge}.

For weights:

EdgeList={(u,v,w) ∣ w is the weight of edge (u,v)}.

A diagram of a graph

Description automatically generated

Incidence Matrix

* Description: A 2D array B[V][E], where rows represent vertices, and columns represent edges.
  + B[i][j]= 1 if vertex i is incident to edge j.
  + For directed graphs: Use +1 for the source vertex and −1 for the destination vertex.
* Formula: B[i][j]={1 if vertex i is incident to edge j,

−1 if vertex i is the end of directed edge j,

0 otherwise}

**Comparison Table for remembrance:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Representation** | **Space Complexity** | **Adjacency Check** | **Adding Edge** | **Removing Edge** | **Traversal Ease** |
| Adjacency Matrix | O() | O(1) | O(1) | O(1) | O() |
| Adjacency List | O(V+E) | O(degree) | O(1) | O(degree) | O(V+E) |
| Edge List | O(E) | O(E) | O(1) | O(E) | Depends on sorting |
| Incidence Matrix | O(V×E) | O(E) | O(V) | O(V) | Rarely used |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Criteria | Adjacency List | Adjacency Matrix | Edge list | Instance matrix |
| Advantages | - Simple and intuitive to implement.  - Constant-time adjacency checks O(1).  - Easy edge addition/removal. | - Space-efficient for sparse graphs (O(V+E)  - Easy to traverse neighbors.  - Simple edge/vertex addition. | - Simple and lightweight.  - Space-efficient for sparse graphs.  - Easy edge addition/removal. | - Explicitly represents vertex-edge relationships.  - Useful for complex edge-vertex relationships. |
| Disadvantages | - High space complexity (V^2)  - Inefficient for sparse graphs.  - Expensive to add/remove vertices. | - Checking adjacency is slower (O(degree)O(\text{degree})O(degree)).  - Inefficient for dense graphs. | - Adjacency checks are inefficient O(E).  - Sorting needed for some algorithms. | - High space complexity O(V×E)  - Complex edge/vertex operations. |
| Use cases | - Dense graphs.  - Frequent adjacency queries. | - Sparse graphs.  - Memory-critical applications. | - Edge-centric algorithms like Kruskal's MST.  - Edge traversal problems. | - Network flow problems.  - Niche domains involving edge-vertex relationships. |

**Operations on Graph in Data Structure:**

**Basic graph operation:**

* Creating a graph with a specified number of vertices and edges.
* Add/Remove Vertex – Add or remove a vertex in a graph.
* Add/Remove Edge – Add or remove an edge between two vertices.
* Check if the graph contains a given value.
* Find the path from one vertex to another vertex

**Graph Algorithms:**

* **BFS(breadth first search):**

It’s a graph traversal algorithm that explores all neighbors of a vertex before moving to their neighbors. It works in a **layered approach**, visiting vertices level by level based on their distance from the starting node. It uses FIFO (Fast In Fast Out) method to visit all the vertices as queues.

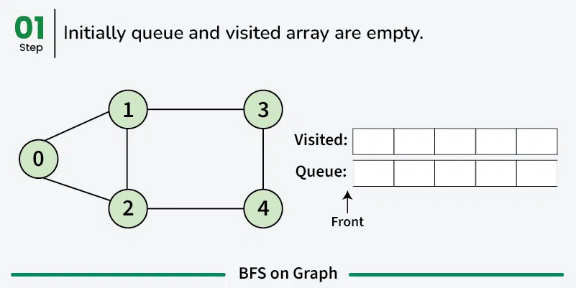
**Characteristics:**

* Traverses the graph in **breadth** (level-wise) rather than depth.
* Ensures all vertices at a given distance from the source are visited before moving further.
* Suitable for finding the **shortest path** in an unweighted graph since it visits vertices in increasing order of distance from the source.

**Applications:**

* Shortest Path
* Connectivity
* Bipartite Graph Detection

**Simulation:**

****

.

**A diagram of a graph

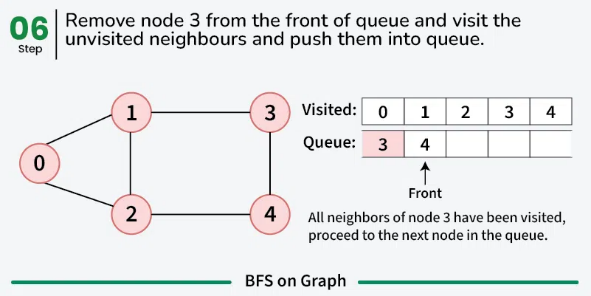
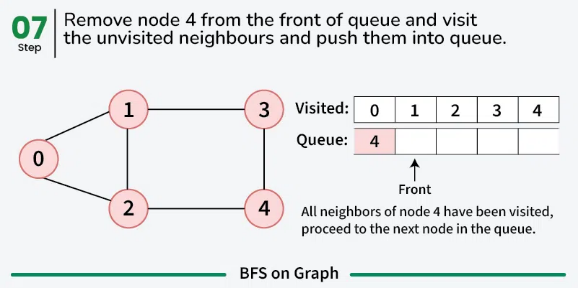
Description automatically generated**

A diagram of a graph

Description automatically generated

**A diagram of a number

Description automatically generated**

A diagram of a graph

Description automatically generated

**Code:**

*#include <iostream>*

*#include <queue>*

*#include <vector>*

*using namespace std;*

*void bfs(vector<vector<int>>& adj, int s)*

*{ queue<int> q;*

*vector<bool> visited(adj.size(), false);*

*visited[s] = true;*

*q.push(s);*

*while (!q.empty()) {*

*int curr = q.front();*

*q.pop();*

*cout << curr << " ";*

*for (int x : adj[curr]) {*

*if (!visited[x]) {*

*visited[x] = true;*

*q.push(x);*

*}}}}*

*// Function to add an edge to the graph*

*void addEdge(vector<vector<int>>& adj,*

*int u, int v)*

*{*

*adj[u].push\_back(v);*

*adj[v].push\_back(u); // Undirected Graph*

*}*

*int main()*

*{ int V = 5;*

*// Adjacency list representation of the graph*

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

*// Add edges to the graph*

*addEdge(adj, 0, 1);*

*addEdge(adj, 0, 2);*

*addEdge(adj, 1, 3);*

*addEdge(adj, 1, 4);*

*addEdge(adj, 2, 4);*

*cout << "BFS starting from 0 : \n";*

*bfs(adj, 0);*

*return 0;}*

**DFS (Depth-First Search):**

It’s a graph traversal algorithm that explores as far as possible along one branch before backtracking. It works recursively or iteratively with a stack (LIFO - Last In, First Out) method to visit vertices, diving deep into the graph before exploring other branches.

**Characteristics:**

* Traverses the graph in depth (deep path exploration) rather than breadth.
* Explores one complete path before backtracking to explore other paths.
* Suitable for exploring all possible paths or searching for specific structures like cycles.

**Applications:**

* Cycle Detection
* Topological Sorting
* Strongly Connected Components
* Maze Solving

*Code:*

*#include <iostream>*

*#include <vector>*

*using namespace std;*

*void DFS(vector<vector<int>> &adj, int start) {*

*vector<bool> visited(adj.size(), false); // Visited array*

*vector<int> container;*

*container.push\_back(start);*

*while (!container.empty()) {*

*int curr = container.back();*

*container.pop\_back();*

*if (!visited[curr]) {*

*visited[curr] = true;*

*cout << curr << " "; // Print the current vertex*

*for (int neighbor : adj[curr]) {*

*if (!visited[neighbor]) {*

*container.push\_back(neighbor);}}}}}*

*int main() {*

*int V = 5; // Number of vertices*

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

*vector<vector<int>> edges = {{1, 2}, {1, 0}, {2, 0}, {2, 3}, {2, 4}};*

*for (auto &e : edges) {*

*adj[e[0]].push\_back(e[1]);*

*adj[e[1]].push\_back(e[0]); // Since it's an undirected graph*

*}*

*int start = 1; // Starting vertex*

*cout << "DFS starting from vertex " << start << ":\n";*

*DFS(adj, start); // Perform iterative DFS*

*return 0;}*

**Dijkstra's Algorithm:**

Dijkstra's algorithm is a greedy algorithm used to find the shortest paths from a source vertex to all other vertices in a graph with non-negative edge weights. The algorithm selects the vertex with the smallest tentative distance and updates its neighbors' distances.

**Characteristics:**

* Finds the shortest path in a weighted graph.
* Works with graphs having non-negative edge weights.
* Uses a priority queue (min-heap) to efficiently retrieve the next vertex with the smallest tentative distance.
* The algorithm terminates when the shortest path to every vertex has been found.

**Applications:**

* Shortest Path in Weighted Graphs
* Network Routing
* GPS Navigation Systems

**Bellman-Ford Algorithm:**

Bellman-Ford is a dynamic programming algorithm that calculates the shortest path from a source vertex to all other vertices in a graph. It can handle graphs with negative edge weights and detect negative weight cycles.

**Characteristics:**

* Can handle graphs with negative edge weights.
* Slower than Dijkstra’s algorithm, with a time complexity of O(VE).
* Can detect negative weight cycles in the graph.
* Repeatedly relaxes edges for all vertices up to (V-1) times.

**Applications:**

* Shortest Path in Graphs with Negative Edge Weights
* Detecting Negative Weight Cycles
* Used in various financial models for detecting arbitrage

**Kruskal's Algorithm:**

Kruskal’s algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) of a graph. It adds the smallest edge to the MST that doesn’t form a cycle, repeating this process until all vertices are connected.

**Characteristics:**

* Focuses on edges, not vertices.
* Always adds the smallest edge that doesn't form a cycle.
* Uses a disjoint-set (union-find) data structure to check for cycles.
* Has a time complexity of O(E log E) due to sorting edges.

**Applications:**

* Minimum Spanning Tree (MST)
* Network Design (e.g., in cable installations or road systems)

**Prim's Algorithm:**

Prim’s algorithm is another greedy algorithm used to find the Minimum Spanning Tree (MST). Unlike Kruskal’s, it grows the MST from a single vertex by repeatedly adding the smallest edge that connects a vertex in the MST to a vertex outside it.

**Characteristics:**

* Starts with a single vertex and grows the MST one edge at a time.
* Uses a priority queue to efficiently select the next smallest edge.
* Works well for dense graphs, where there are many edges.
* Time complexity is O(E log V) with an efficient priority queue.

**Applications:**

* Minimum Spanning Tree (MST)
* Designing Efficient Networks

**Topological Sort:**

Topological sort is a linear ordering of the vertices in a Directed Acyclic Graph (DAG) such that for every directed edge (u, v), vertex u comes before vertex v in the ordering. This is often used to represent dependencies between tasks or processes.

**Characteristics:**

* Only applicable to Directed Acyclic Graphs (DAGs).
* Produces a linear ordering of vertices.
* Can be implemented using either DFS or Kahn’s Algorithm (BFS).
* If a graph has a cycle, no topological sort is possible.

**Applications:**

* Task Scheduling
* Compilation Order in Programming
* Project Planning and Dependencies

**Stack and Queue:**

**Stack:**

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle, where the last element added to the stack is the first to be removed. It allows insertion and deletion of elements only from one end, called the top.

**Characteristics:**

* Operates on the LIFO principle.
* Allows two primary operations:
  + Push: Add an element to the top of the stack.
  + Pop: Remove the topmost element.
* Supports additional operations like Peek/Top (view the top element without removing it).
* It can be implemented using arrays or linked lists.

**Applications:**

* Function Call Stack
* Backtracking (solving mazes, puzzles)
* Undo Mechanisms
* Expression Evaluation.

**Operations:**

**Push**

* **Description**: Adds an element to the top of the stack.
* **Effect**: Increases the stack size.
* **Example**:
  + Initial Stack: [10, 20]
  + Push 30
  + Final Stack: [10, 20, 30]

**2. Pop**

* **Description**: Removes and returns the topmost element from the stack.
* **Effect**: Decreases the stack size.
* **Example**:
  + Initial Stack: [10, 20, 30]
  + Pop → 30
  + Final Stack: [10, 20]
* **Note**: If the stack is empty, a pop operation usually results in an **underflow error**.

**3. Peek (or Top)**

* **Description**: Returns the topmost element of the stack without removing it.
* **Effect**: Does not modify the stack.
* **Example**:
  + Stack: [10, 20, 30]
  + Peek → 30

**4. IsEmpty**

* **Description**: Checks whether the stack is empty.
* **Effect**: Returns true if the stack has no elements, otherwise false.
* **Example**:
  + Stack: [10, 20]
  + IsEmpty → false

**5. IsFull (in case of a bounded stack)**

* **Description**: Checks whether the stack has reached its maximum capacity.
* **Effect**: Returns true if the stack is full, otherwise false.
* **Example**:
  + Stack: [10, 20, 30] (Capacity: 3)
  + IsFull → true

**6. Size**

* **Description**: Returns the number of elements in the stack.
* **Effect**: Does not modify the stack.
* **Example**:
  + Stack: [10, 20, 30]
  + Size → 3

Example:

*#include <iostream>*

*using namespace std;*

*// Declare stack and variables*

*int stack[MAX\_SIZE];*

*int top = -1; // Initialize the stack as empty*

*void push(int item) {*

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

*cout << "Stack Overflow! Cannot add more items." << endl;*

*} else {*

*top++;*

*stack[top] = item;*

*cout << "Pushed: " << item << endl;*

*}*

*}*

*void pop() {*

*if (top == -1) {*

*cout << "Stack Underflow! No items to remove." << endl;*

*} else {*

*cout << "Popped: " << stack[top] << endl;*

*top--;*

*}}*

*void peek() {*

*if (top == -1) {*

*cout << "Stack is empty. Nothing to peek." << endl;*

*} else {*

*cout << "Top Element: " << stack[top] << endl;*

*}}*

*bool isEmpty() {*

*return top == -1;*

*}*

*bool isFull() {*

*return top == MAX\_SIZE - 1;*

*}*

*int size() {*

*return top + 1;*

*}*

*// Main function to demonstrate stack operations*

*int main() {*

*// Push elements*

*push(10);*

*push(20);*

*push(30);*

*peek();*

*cout << "Is Full? " << (isFull() ? "Yes" : "No") << endl;*

*push(40);*

*push(50);*

*push(60); // Should trigger overflow*

*cout << "Current Stack Size: " << size() << endl;*

*pop();*

*pop();*

*cout << "Is Empty? " << (isEmpty() ? "Yes" : "No") << endl;*

*peek();*

*cout << "Final Stack Size: " << size() << endl;*

*return 0;}*

**Queue:**

A queue is a linear data structure that follows the First In, First Out (FIFO) principle, where the first element added is the first to be removed. It supports operations at two ends:

* **Enqueue**: Add an element to the rear.
* **Dequeue**: Remove an element from the front.

**Characteristics:**

* Operates on the FIFO principle.
* Allows insertion at the rear and removal from the front.
* Variants include:
  + **Circular Queue**: The rear wraps around to the front when the queue becomes full.
  + **Priority Queue**: Elements are dequeued based on priority, not arrival order.
  + **Double-ended Queue (Deque)**: Insertion and deletion allowed at both ends.

**Applications:**

* Process Scheduling
* Breadth-First Search (BFS)
* Buffer Management
* Simulation Systems

**Operations:**

**Enqueue**

* **Description**: Adds an element to the rear (end) of the queue.
* **Effect**: Increases the size of the queue.
* **Example**:
  + Initial Queue: [10, 20]
  + Enqueue 30
  + Final Queue: [10, 20, 30]

**2. Dequeue**

* **Description**: Removes and returns the front element of the queue.
* **Effect**: Decreases the size of the queue.
* **Example**:
  + Initial Queue: [10, 20, 30]
  + Dequeue → 10
  + Final Queue: [20, 30]
* **Note**: If the queue is empty, this operation may result in an **underflow error**.

**3. Peek (or Front)**

* **Description**: Returns the front element of the queue without removing it.
* **Effect**: Does not modify the queue.
* **Example**:
  + Queue: [10, 20, 30]
  + Peek → 10

**4. IsEmpty**

* **Description**: Checks whether the queue is empty.
* **Effect**: Returns true if there are no elements in the queue, otherwise false.
* **Example**:
  + Queue: []
  + IsEmpty → true

**5. IsFull (in case of a bounded queue)**

* **Description**: Checks whether the queue has reached its maximum capacity.
* **Effect**: Returns true if the queue is full, otherwise false.
* **Example**:
  + Queue: [10, 20, 30, 40] (Capacity: 4)
  + IsFull → true

**6. Size**

* **Description**: Returns the number of elements currently in the queue.
* **Effect**: Does not modify the queue.
* **Example**:
  + Queue: [10, 20, 30]
  + Size → 3

**Code:**

#include <iostream>

using namespace std;

int queue[MAX\_SIZE];

int front = -1; // Points to the front of the queue

int rear = -1; // Points to the rear of the queue

void enqueue(int item) {

if ((rear + 1) % MAX\_SIZE == front) {

cout << "Queue Overflow! Cannot add more items." << endl;

} else {

if (front == -1) {

// First element being added

front = 0;

}

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

queue[rear] = item;

cout << "Enqueued: " << item << endl;

}}

void dequeue() {

if (front == -1) {

cout << "Queue Underflow! No items to remove." << endl;

} else {

cout << "Dequeued: " << queue[front] << endl;

if (front == rear) {

// Queue becomes empty

front = rear = -1;

} else {

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

}}}

void peek() {

if (front == -1) {

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

} else {

cout << "Front Element: " << queue[front] << endl;

}}

bool isEmpty() {

return front == -1;

}

bool isFull() {

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

}

// Size operation: Get the current size of the queue

int size() {

if (front == -1) {

return 0;

} else if (rear >= front) {

return rear - front + 1;

} else {

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

int main() {

// Enqueue elements

enqueue(10);

enqueue(20);

enqueue(30);

peek();

cout << "Is Full? " << (isFull() ? "Yes" : "No") << endl;

enqueue(40);

enqueue(50);

enqueue(60); // Should trigger overflow

cout << "Current Queue Size: " << size() << endl;

dequeue();

dequeue();

cout << "Is Empty? " << (isEmpty() ? "Yes" : "No") << endl;

peek();

cout << "Final Queue Size: " << size() << endl;

return 0;

}

**Gist:**

|  |  |  |
| --- | --- | --- |
| Feature | Stack | Queue |
| Definition | Follows LIFO (Last In, First Out). | Follows FIFO (First In, First Out). |
| Primary Operation | Push (add), Pop (remove), Peek (view top). | Enqueue (add), Dequeue (remove), Peek (view front). |
| Insertion | Happens at the top of the stack. | Happens at the rear of the queue. |
| Removal | Happens from the top of the stack. | Happens from the front of the queue. |
| Usage | Used for backtracking, undo mechanisms, and DFS. | Used for scheduling, buffering, and BFS. |
| IsEmpty | Checks if top == -1. | Checks if front == -1. |
| IsFull | Checks if top == maxSize - 1. | Checks if (rear + 1) % maxSize == front (in circular queue). |
| Size | Returns top + 1. | Computes size based on front and rear. |
| Advantages | Simple to implement, requires less memory. | Handles tasks efficiently in order, good for dynamic scenarios. |
| Disadvantages | Not suitable for sequential processing. | Limited flexibility for reverse traversal. |
| Real-world examples | Call stack, undo/redo, parsing. | Task scheduling, printer queues, data buffers. |

**Matrices and Large Integers:**

**Matrices**

* A **matrix** is a rectangular array of numbers arranged in rows and columns. It is widely used in mathematics, physics, computer science, and engineering for various computations.

**Characteristics:**

* Represented as a 2D grid: m x n, where m = rows, n = columns.
* Elements are accessed using indices: matrix[i][j].
* Can perform operations like addition, subtraction, multiplication, and transposition.

**Common Operations:**

1. **Addition**:
   * Two matrices can be added if they have the same dimensions.
   * Formula: C[i][j] = A[i][j] + B[i][j].
2. **Subtraction**:
   * Similar to addition; subtract corresponding elements.
   * Formula: C[i][j] = A[i][j] - B[i][j].
3. **Multiplication**:
   * Two matrices can be multiplied if the number of columns in the first matrix equals the number of rows in the second.
   * Formula: C[i][j] = Σ(A[i][k] \* B[k][j]).
4. **Transpose**:
   * Flips a matrix over its diagonal.
   * Formula: T[j][i] = A[i][j].
5. **Determinant** (Square Matrix Only):
   * A scalar value representing the matrix's properties.
   * Used to check invertibility: det(A) ≠ 0 → invertible.
6. **Inverse**:
   * Only for square matrices with non-zero determinant.
   * Formula: A⁻¹ = adj(A) / det(A).

**Advantages:**

* Compact representation for linear equations.
* Efficient for numerical calculations.
* Foundation for advanced concepts like eigenvalues and singular value decomposition (SVD).

**Disadvantages:**

* Operations can become computationally expensive for large matrices.
* Requires specific conditions (e.g., dimensions for multiplication).

**Applications:**

* Computer Graphics
* Physics
* Machine Learning
* Graph Theory

**Code:**

***#include <iostream>***

***#include <iomanip> // For better formatting***

***using namespace std;***

***void displayMatrix(int matrix[3][3], int rows, int cols) {***

***for (int i = 0; i < rows; i++) {***

***for (int j = 0; j < cols; j++) {***

***cout << setw(5) << matrix[i][j];***

***}***

***cout << endl;***

***}***

***}***

***void addMatrices(int A[3][3], int B[3][3], int C[3][3], int rows, int cols) {***

***for (int i = 0; i < rows; i++) {***

***for (int j = 0; j < cols; j++) {***

***C[i][j] = A[i][j] + B[i][j];***

***}***

***}***

***}***

***void subtractMatrices(int A[3][3], int B[3][3], int C[3][3], int rows, int cols) {***

***for (int i = 0; i < rows; i++) {***

***for (int j = 0; j < cols; j++) {***

***C[i][j] = A[i][j] - B[i][j];***

***}***

***}***

***}***

***void multiplyMatrices(int A[3][3], int B[3][3], int C[3][3], int rowsA, int colsA, int colsB) {***

***for (int i = 0; i < rowsA; i++) {***

***for (int j = 0; j < colsB; j++) {***

***C[i][j] = 0;***

***for (int k = 0; k < colsA; k++) {***

***C[i][j] += A[i][k] \* B[k][j];***

***}***

***}***

***}***

***}***

***void transposeMatrix(int A[3][3], int T[3][3], int rows, int cols) {***

***for (int i = 0; i < rows; i++) {***

***for (int j = 0; j < cols; j++) {***

***T[j][i] = A[i][j];***

***}***

***}***

***}***

***int determinant2x2(int matrix[2][2]) {***

***return (matrix[0][0] \* matrix[1][1]) - (matrix[0][1] \* matrix[1][0]);***

***}***

***void inverse2x2(int matrix[2][2], double inverse[2][2]) {***

***int det = determinant2x2(matrix);***

***if (det == 0) {***

***cout << "Matrix is not invertible!" << endl;***

***return;***

***}***

***inverse[0][0] = matrix[1][1] / static\_cast<double>(det);***

***inverse[0][1] = -matrix[0][1] / static\_cast<double>(det);***

***inverse[1][0] = -matrix[1][0] / static\_cast<double>(det);***

***inverse[1][1] = matrix[0][0] / static\_cast<double>(det);***

***}***

***int main() {***

***int A[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};***

***int B[3][3] = {{9, 8, 7}, {6, 5, 4}, {3, 2, 1}};***

***int C[3][3]; // Resultant matrix***

***int T[3][3]; // Transposed matrix***

***cout << "Matrix A:" << endl;***

***displayMatrix(A, 3, 3);***

***cout << "\nMatrix B:" << endl;***

***displayMatrix(B, 3, 3);***

***addMatrices(A, B, C, 3, 3);***

***cout << "\nAddition of A and B:" << endl;***

***displayMatrix(C, 3, 3);***

***subtractMatrices(A, B, C, 3, 3);***

***cout << "\nSubtraction of A and B:" << endl;***

***displayMatrix(C, 3, 3);***

***multiplyMatrices(A, B, C, 3, 3, 3);***

***cout << "\nMultiplication of A and B:" << endl;***

***displayMatrix(C, 3, 3);***

***transposeMatrix(A, T, 3, 3);***

***cout << "\nTranspose of A:" << endl;***

***displayMatrix(T, 3, 3);***

***int D[2][2] = {{4, 7}, {2, 6}};***

***cout << "\nDeterminant of matrix D:" << determinant2x2(D) << endl;***

***double inv[2][2];***

***cout << "\nInverse of matrix D:" << endl;***

***inverse2x2(D, inv);***

***for (int i = 0; i < 2; i++) {***

***for (int j = 0; j < 2; j++) {***

***cout << setw(10) << inv[i][j];***

***}***

***cout << endl;***

***}***

***return 0;***

***}***

**BigInt (Large Integers)**

**Definition**:  
A **BigInt** represents integers of arbitrary size, overcoming the limitation of fixed-size integer types like int and long in programming. It is commonly used when computations involve numbers larger than the maximum value of standard data types.

**Characteristics:**

* Can handle **very large integers** without overflow.
* Stored and processed as **strings or arrays of digits** internally.
* Operations mimic normal arithmetic but require custom algorithms for addition, subtraction, multiplication, etc.

**Common Operations:**

1. **Addition**:
   * Perform digit-wise addition from least significant to most significant digit.
   * Carry-over is handled explicitly.
2. **Subtraction**:
   * Digit-wise subtraction with borrowing when needed.
   * Ensure the larger number is the minuend to avoid negative results (unless signed is supported).
3. **Multiplication**:
   * Multiply each digit of one number with every digit of the other.
   * Use position shifts (powers of 10) for correct placement.
4. **Division**:
   * Use long-division-like algorithms to find the quotient and remainder.
5. **Modulo**:
   * Compute the remainder of a division operation.
6. **Power**:
   * Efficient algorithms like **Exponentiation by Squaring** handle large exponents.
7. **Comparison**:
   * Compare numbers digit by digit for equality, greater, or lesser.

**Applications:**

* Cryptography
* Factorial Calculation
* Astronomy
* Finance

**Code:**

#include <iostream>

using namespace std;

void reverseArray(char num[], int len) {

for (int i = 0; i < len / 2; ++i) {

char temp = num[i];

num[i] = num[len - 1 - i];

num[len - 1 - i] = temp;

}

}

void addBigInt(char num1[], char num2[], char result[]) {

int len1 = 0, len2 = 0;

while (num1[len1] != '\0') len1++;

while (num2[len2] != '\0') len2++;

reverseArray(num1, len1);

reverseArray(num2, len2);

int carry = 0, i = 0;

while (i < len1 || i < len2 || carry) {

int digit1 = i < len1 ? num1[i] - '0' : 0;

int digit2 = i < len2 ? num2[i] - '0' : 0;

int sum = digit1 + digit2 + carry;

result[i] = (sum % 10) + '0';

carry = sum / 10;

i++;

}

result[i] = '\0';

reverseArray(result, i);

}

void subtractBigInt(char num1[], char num2[], char result[]) {

int len1 = 0, len2 = 0;

while (num1[len1] != '\0') len1++;

while (num2[len2] != '\0') len2++;

reverseArray(num1, len1);

reverseArray(num2, len2);

int borrow = 0, i = 0;

while (i < len1 || i < len2) {

int digit1 = i < len1 ? num1[i] - '0' : 0;

int digit2 = i < len2 ? num2[i] - '0' : 0;

int diff = digit1 - digit2 - borrow;

if (diff < 0) {

diff += 10;

borrow = 1;

} else {

borrow = 0;

}

result[i] = diff + '0';

i++;

}

result[i] = '\0';

while (i > 1 && result[i - 1] == '0') i--;

result[i] = '\0';

reverseArray(result, i);

}

void multiplyBigInt(char num1[], char num2[], char result[]) {

int len1 = 0, len2 = 0;

while (num1[len1] != '\0') len1++;

while (num2[len2] != '\0') len2++;

int tempResult[1000] = {0};

reverseArray(num1, len1);

reverseArray(num2, len2);

for (int i = 0; i < len1; ++i) {

int carry = 0;

for (int j = 0; j < len2 || carry; ++j) {

int digit1 = num1[i] - '0';

int digit2 = j < len2 ? num2[j] - '0' : 0;

int product = digit1 \* digit2 + tempResult[i + j] + carry;

tempResult[i + j] = product % 10;

carry = product / 10;

}

}

int k = len1 + len2;

while (k > 0 && tempResult[k - 1] == 0) k--;

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

result[i] = tempResult[i] + '0';

}

result[k] = '\0';

reverseArray(result, k);

}

int main() {

char num1[1000], num2[1000], result[2000];

cout << "Enter the first BigInt: ";

cin >> num1;

cout << "Enter the second BigInt: ";

cin >> num2;

addBigInt(num1, num2, result);

cout << "Addition: " << result << endl;

// Subtraction (assumes num1 >= num2)

if (num1[0] != '-' && num2[0] != '-') { // Only for positive numbers

subtractBigInt(num1, num2, result);

cout << "Subtraction: " << result << endl;

} else {

cout << "Subtraction not supported for negative numbers in this example." << endl;

}

multiplyBigInt(num1, num2, result);

cout << "Multiplication: " << result << endl;

return 0;}