Maps in C++ are associative containers that store elements in key-value pairs. The two main types of maps in C++ are **std::map** (ordered map) and **std::unordered\_map** (unordered map).

**std::map (Ordered Map)**

**Underlying Data Structure:**

* **Balanced Binary Search Tree** (usually a Red-Black Tree).

**Characteristics:**

* Maintains keys in sorted order.
* Provides logarithmic time complexity for insertions, deletions, and lookups.
* Elements are stored in a specific order based on the keys.

**Efficiency:**

* Efficient for scenarios where ordered traversal of keys is required.
* Suitable for range queries where you need to find elements within a specific range of keys.

**Time Complexities:**

* **Insertion:** O(log n)
* **Deletion:** O(log n)
* **Lookup:** O(log n)

**Algorithms and Data Structures:**

* **Red-Black Tree:** A type of self-balancing binary search tree.
  + Ensures the tree remains balanced during insertions and deletions.
  + Provides guarantees on the maximum height of the tree, ensuring logarithmic time complexity for operations.

**std::unordered\_map (Unordered Map)**

**Underlying Data Structure:**

* **Hash Table**.

**Characteristics:**

* Does not maintain any specific order for keys.
* Provides average constant time complexity for insertions, deletions, and lookups.
* Elements are stored based on their hash values.

**Efficiency:**

* Efficient for scenarios where fast lookups, insertions, and deletions are required.
* Suitable when the order of elements is not important.

**Time Complexities:**

* **Insertion:** O(1) on average, O(n) in the worst case (due to rehashing).
* **Deletion:** O(1) on average, O(n) in the worst case.
* **Lookup:** O(1) on average, O(n) in the worst case.

**Algorithms and Data Structures:**

* **Hash Table:**
  + Uses a hash function to map keys to buckets where values are stored.
  + Provides efficient O(1) average time complexity for operations.
  + In the worst case, when many keys hash to the same bucket, time complexity degrades to O(n).

**Key Concepts in Hash Tables:**

* **Hash Function:** Computes an index (hash value) for each key, determining where the key-value pair is stored.
* **Collision Handling:** Techniques like chaining (using linked lists) or open addressing (probing) to handle cases where multiple keys hash to the same index.
* **Rehashing:** Process of resizing the hash table and redistributing the elements when the load factor (number of elements to the size of the hash table) exceeds a threshold.

**Summary**

* **std::map:**
  + Uses a balanced binary search tree (Red-Black Tree).
  + Maintains elements in sorted order.
  + Provides O(log n) time complexity for operations.
  + Suitable for ordered traversal and range queries.
* **std::unordered\_map:**
  + Uses a hash table.
  + Does not maintain any order.
  + Provides O(1) average time complexity for operations.
  + Suitable for fast lookups, insertions, and deletions without concern for order.

By understanding the underlying data structures and their time complexities, you can choose the appropriate type of map for your specific use case.

BFS VS DFS

**BFS (Breadth-First Search) vs. DFS (Depth-First Search) Algorithms**

**BFS (Breadth-First Search)**

**Overview:**

* BFS is a graph traversal algorithm that starts at a given node and explores all its neighbors at the present depth level before moving on to nodes at the next depth level.
* It uses a queue data structure to keep track of the nodes to be explored.

**Data Structure Used:**

* **Queue:** This is the main data structure used in BFS. It follows the First-In-First-Out (FIFO) principle, which helps in exploring nodes level by level.

**Steps:**

1. Enqueue the starting node.
2. Dequeue a node, process it, and enqueue all its unvisited neighbors.
3. Repeat step 2 until the queue is empty.

**Use Cases:**

* Finding the shortest path in an unweighted graph.
* Level-order traversal of a tree.
* Crawlers in search engines.
* Peer-to-peer networks.

**DFS (Depth-First Search)**

**Overview:**

* DFS is a graph traversal algorithm that starts at a given node and explores as far along each branch as possible before backtracking.
* It uses a stack data structure (either explicit or implicit via recursion) to keep track of the nodes to be explored.

**Data Structure Used:**

* **Stack:** This can be an explicit stack or the call stack used in recursion. It follows the Last-In-First-Out (LIFO) principle, which helps in exploring nodes deeply before backtracking.

| **Criterion** | **BFS** | **DFS** |
| --- | --- | --- |
| Data Structure | Queue | Stack |
| Exploration | Level by level | Depth by depth |
| Memory Usage | Higher (needs to store all nodes at the current level) | Lower (only needs to store nodes along the current path) |
| Path Discovery | Shortest path in unweighted graphs | Not guaranteed to find the shortest path |
| Use Cases | Shortest path, broadcasting | Pathfinding, topological sorting |