Before diving into solving problems on platforms like LeetCode or any other, it's crucial to have a strong grip on your programming language. Think of it as learning to "speak" the language fluently so you can effectively "communicate" with the problems. For instance, in C++, understanding fundamentals like loops, functions, and data types is just the beginning. Mastery of the Standard Template Library (STL), which provides powerful tools like `vectors`, `maps`, and `priority_queues`, is equally important. For example, solving a problem involving sorting could be simplified using `sort()` from STL, which is both efficient and easy to use:

```
vector<int> nums = {5, 3, 8, 1};
sort(nums.begin(), nums.end());
// nums is now {1, 3, 5, 8}
```

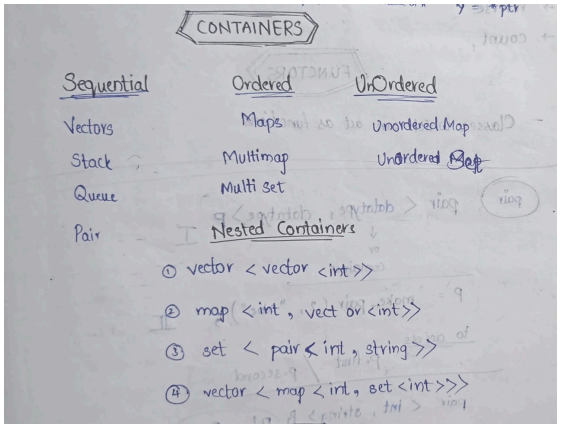
For me, C++ became my language of choice, and focusing on its fundamentals and STL gave me the confidence to approach problems systematically. So, start by mastering your chosen language—it's the key to unlocking your DSA journey!

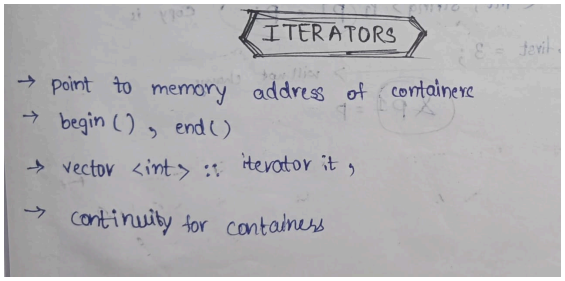
If you're looking for a great resource to learn STL, this YouTube series worked wonders for me:

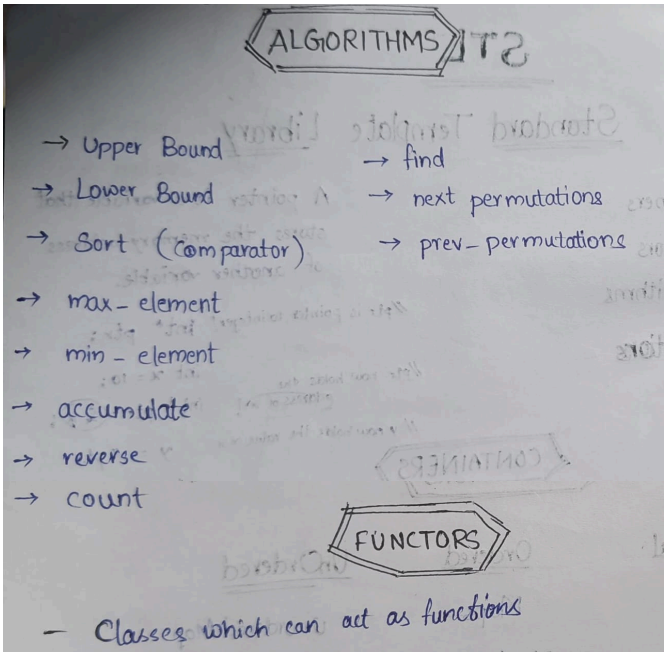
STL Playlist

The key topics covered in the playlist are:

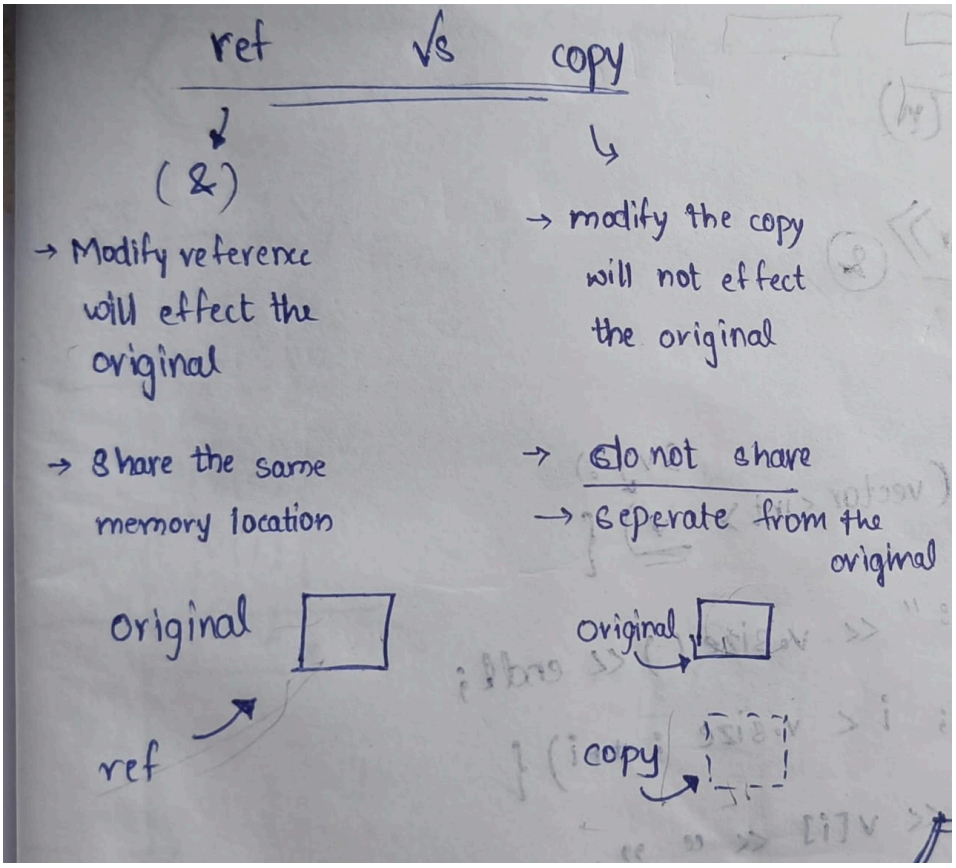
- **Containers**: Learn to use dynamic structures like vectors, maps, sets, etc.
- **Iterators**: Understand how to traverse containers efficiently.
- **Algorithms**: Leverage built-in functions like `sort()`, `lower_bound()`, etc., for optimized solutions.
- **Functors**: Dive into the concept of function objects and their utility in custom operations.

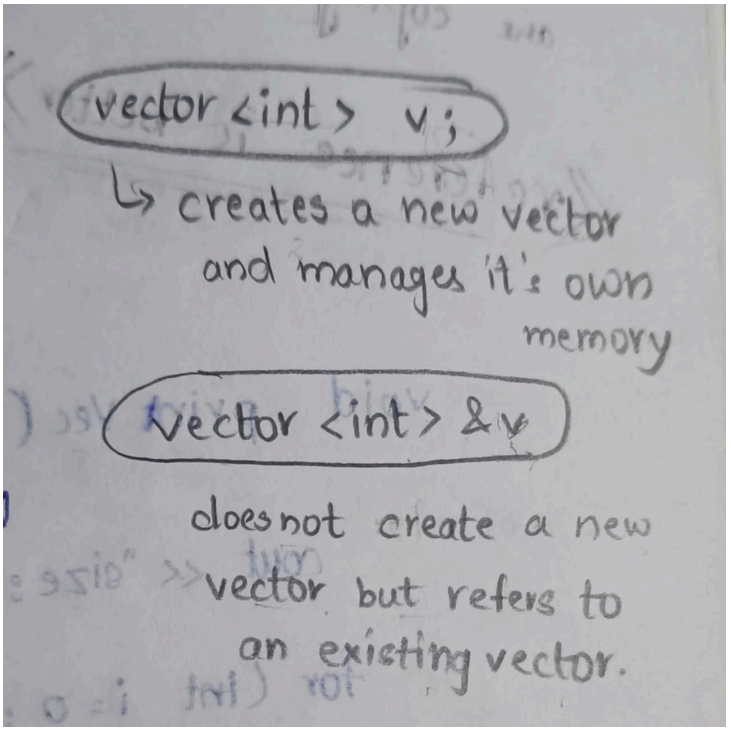






Understanding the difference between **reference** and **copy** is crucial in programming, especially in languages like C++, where memory management plays a vital role.






```cpp
vector<int> v;  // Creates a new vector
v.push_back(1); // Adds 1 to the vector
```

```cpp
void modifyVector(vector<int>& v) {  // Reference to an existing vector
    v.push_back(10);  // Modifies the original vector
}

int main() {
    vector<int> v = {1, 2, 3};
    modifyVector(v);  // Pass by reference
    // v now contains: {1, 2, 3, 10}
}
```

COPYING ARRAYS AND VECTORS?!

Arrays in C++

- **Fixed Size:** Arrays in C++ are static and have a fixed size at compile time.
- **No Built-In Copying:** Arrays cannot be copied directly using assignment (=). The assignment operation between arrays just copies the pointer, not the content. If you want to copy an array, you must do it element by element.

Example:

cpp Copy code

```cpp
int arr1[] = {1, 2, 3};
int arr2[] = arr1;  // Error: Arrays cannot be directly assigned
```

Correct way to copy:

cpp Copy code

```cpp
int arr1[] = {1, 2, 3};
int arr2[3];
for (int i = 0; i < 3; i++) {
    arr2[i] = arr1[i];  // Element-wise copy
}
```

Vectors in C++

- **Dynamic Size:** Vectors are dynamic and can grow or shrink at runtime.
- **Built-In Copying:** Vectors support direct assignment (=) and copying. When you assign one vector to another, the contents are copied, not just the reference.

Example:

cpp Copy code

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

vector<int> vec1 = {1, 2, 3};
vector<int> vec2 = vec1;  // vec2 is a copy of vec1
vec2[0] = 10;             // Modifying vec2 doesn't affect vec1

// vec1 = {1, 2, 3}, vec2 = {10, 2, 3}
```

Copying a vector has a time complexity of O(n), as it duplicates all elements, using more memory and time. References (O(1)) are more efficient since they avoid copying and directly access the original vector. Use references for performance and copying when you need a separate, independent version of the vector.

Array of Vector VS Vector of Vector

Array of Vectors:

- It's an array where each element is a vector. This means that the size of the array is fixed, but each vector can vary in size.

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

int main() {
    int n;
    cin >> n;  // input size of array

    vector<int> arr[n];  // array of n vectors

    for (int i = 0; i < n; i++) {
        int m;
        cin >> m;  // size of the i-th vector
        for (int j = 0; j < m; j++) {
            int x;
            cin >> x;  // element to add
            arr[i].push_back(x);  // add element to the vector
        }
    }

    // Output all vectors
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < arr[i].size(); j++) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Vector of Vectors:

- It's a single vector that holds other vectors as its elements. The size of the outer vector is dynamic, and each inner vector can vary in size.

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

int main() {
    int n;
    cin >> n;  // input number of vectors

    vector<vector<int>> v;  // vector of vectors

    for (int i = 0; i < n; i++) {
        int m;
        cin >> m;  // size of the i-th vector
        vector<int> temp;  // temporary vector
        for (int j = 0; j < m; j++) {
            int x;
            cin >> x;  // element to add
            temp.push_back(x);  // add element to the temporary vector
        }
        v.push_back(temp);  // add the vector to the vector of vectors
    }

    // Output all vectors
    for (int i = 0; i < v.size(); i++) {
        for (int j = 0; j < v[i].size(); j++) {
            cout << v[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

ITERATORS

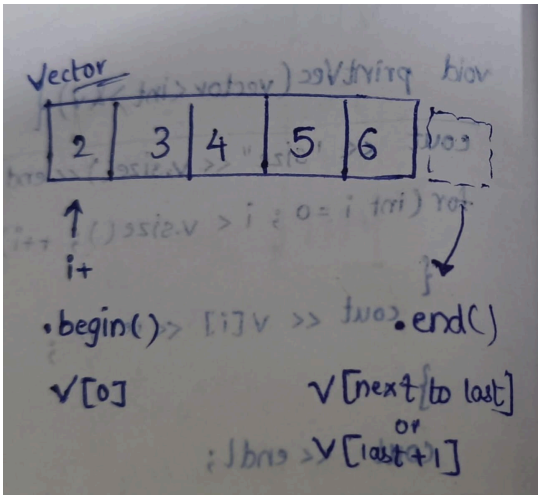
Difference Between Iterators and Indices

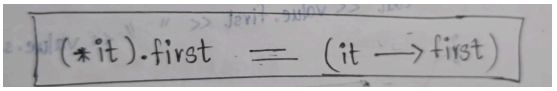
1. **Iterator**:
   - Works with STL containers (`vector`, `set`, `map`, etc.).
   - Allows traversal without relying on index (e.g., `it = v.begin()`).
   - Example:

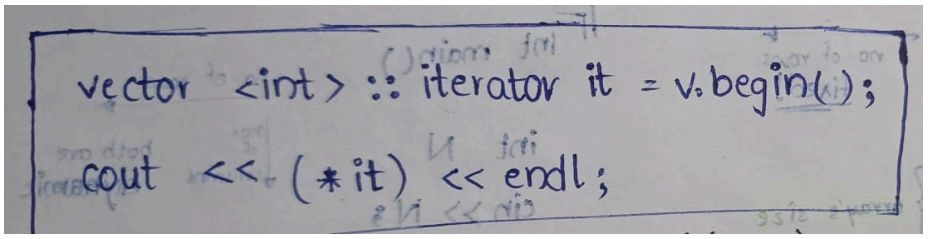
```cpp
for (auto it = v.begin(); it != v.end(); ++it) {
    cout << *it << " "; // Dereference iterator
}
```

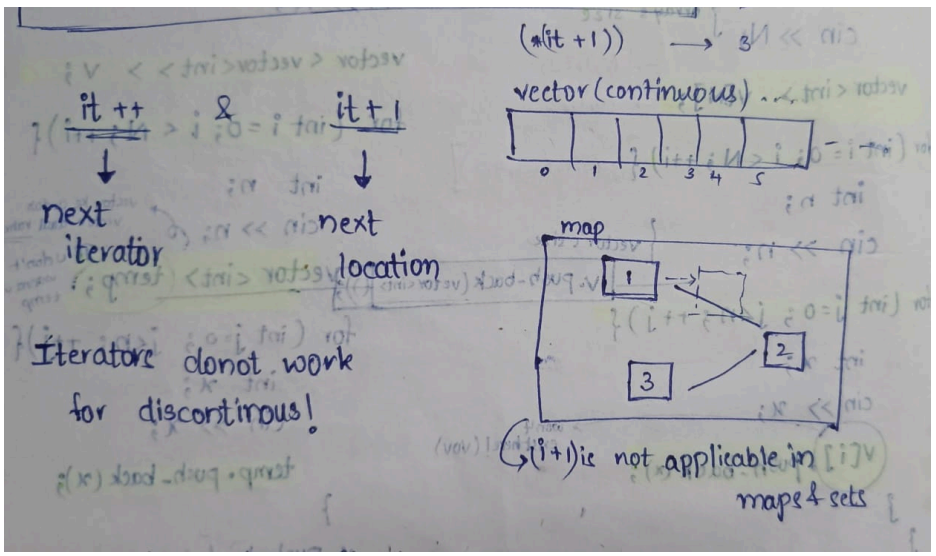
2. **Index (`i`)**:
   - Limited to containers with contiguous storage (like arrays or vectors).
   - Directly uses numerical indexing (e.g., `v[i]`).

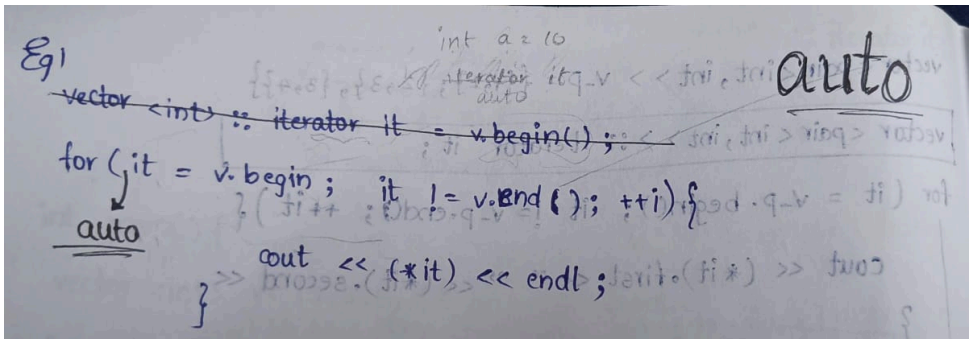
Both are tools for traversal but iterators are more versatile and work seamlessly across all containers in the Standard Template Library.











Writing `vector<int>::iterator` is cumbersome, especially with complex container types like `map<string, vector<int>>`.

Maps in C++

Maps in C++ are part of the STL (Standard Template Library) and are powerful data structures for storing key-value pairs. Here's everything you need to know:

---

1. Ordered Maps (`std::map`)

- **Definition**: Stores key-value pairs in sorted order of keys (ascending by default).
- **Internal Implementation**: Uses a **Balanced Binary Search Tree** (usually a Red-Black Tree).
- **Key Operations**:
    1. **Insert**: Insert a key-value pair.
    2. **Find**: Search for a key.
    3. **Erase**: Delete a key-value pair.

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

int main() {
    map<int, string> m;

    // Insert key-value pairs
    m[1] = "One";
    m[2] = "Two";
    m[3] = "Three";

    // Using m.insert
    m.insert({4, "Four"});

    // Find a key
    if (m.find(2) != m.end()) {
        cout << "Found key 2: " << m[2] << endl;
    }

    // Erase a key
    m.erase(3);

    // Iterating over the map
    for (auto it : m) {
        cout << it.first << ": " << it.second << endl;
    }

    return 0;
}
```

Key Points about `std::map`:

1. **Time Complexity**:
    - Insert/Find/Erase: **O(log n)** (due to balanced BST).
2. **Order**: Maintains keys in sorted order.
3. **Valid Key Datatypes**:
    - Any datatype with a comparison operator defined (`<` or `>`).

2. Unordered Maps (`std::unordered_map`)

- **Definition**: Stores key-value pairs without any specific order.
- **Internal Implementation**: Uses a **Hash Table**.
    1. Every key has a hash value calculated using a hash function.
    2. The hash value determines the bucket where the key-value pair is stored.
- **Key Operations**:
    1. **Insert**: Add a key-value pair.
    2. **Find**: Search for a key in **O(1)** average time.
    3. **Erase**: Remove a key-value pair.

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

int main() {
    unordered_map<int, string> um;

    // Insert key-value pairs
    um[1] = "One";
    um[2] = "Two";

    // Using m.insert (pair can't be inserted directly into unordered_map without a defin
    um.insert({3, "Three"});

    // Find a key
    if (um.find(2) != um.end()) {
        cout << "Found key 2: " << um[2] << endl;
    }

    // Erase a key
    um.erase(1);

    // Iterating over the unordered_map
    for (auto it : um) {
        cout << it.first << ": " << it.second << endl;
    }

    return 0;
}
```

Key Considerations for Unordered Maps

1. **Inbuilt Implementation**:
   Hash Tables are used internally.
2. **Time Complexity**:
   - **Average**: O(1) for insert, find, and erase.
   - **Worst-case**: O(n) (when hash collisions occur).
3. **Valid Key Datatypes**:
   - Datatypes like `int`, `long`, and `double` work because hash functions are defined for them internally.
   - Datatypes like `pair`, `set`, or custom types **don't work** directly because no hash function is defined for them by default.

Comparison: Ordered vs Unordered Maps

Feature	`std::map`	`std::unordered_map`
Internal Structure	Balanced BST (Red-Black Tree)	Hash Table
Order	Keys are sorted	No specific order
Time Complexity	O(log n)	O(1) on average
Insertion	Direct	Cannot insert custom pairs directly unless the hash function is defined
Valid Key Types	Any type with comparison operators	Limited to types with defined hash functions (`int`, `double`, etc.)

Important Notes:

1. **Inserting Custom Data Types in `std::unordered_map`:**
   - You need to define a hash function and an equality operator for custom types like `pair` or `set`.

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

struct CustomHash {
    size_t operator()(const pair<int, int>& p) const {
        return hash<int>()(p.first) ^ hash<int>()(p.second);
    }
};

int main() {
    unordered_map<pair<int, int>, string, CustomHash> um;
    um[make_pair(1, 2)] = "Pair";
    cout << um[make_pair(1, 2)] << endl;
    return 0;
}
```

2. When to Use `std::unordered_map`:

- When order of keys doesn't matter, and you need the fastest access.
- Example: Storing frequency counts of elements.

3. When to Use `std::map`:

- When you need sorted keys or want range-based queries.

Feature	`std::map` (Ordered Map)	`std::multimap` (Multi-key Map)	`std::unordered_map` (Hash Map)
Key Uniqueness	**Unique keys** only.	**Allows duplicate keys.**	**Unique keys** only.
Ordering	**Sorted order** (based on the key).	**Sorted order** (based on the key).	**Unordered** (no specific order).
Internal Structure	**Red-Black Tree** (Balanced BST).	**Red-Black Tree** (Balanced BST).	**Hash Table.**
Insertion Time Complexity	$O(\log n)$	$O(\log n)$	$O(1)$ on average, $O(n)$ in worst case (due to collisions).
Access Time Complexity	$O(\log n)$	$O(\log n)$	$O(1)$ on average, $O(n)$ in worst case.
Duplicate Keys	**No** (Overwrites existing value).	**Yes** (Multiple elements with the same key).	**No** (Overwrites existing value).
Direct Access to Elements	Yes, using `map[key]`.	No direct access, use iterators or `equal_range()`.	Yes, using `unordered_map[key]`.
Memory Usage	Moderate (due to tree structure).	Moderate (due to tree structure).	Low (due to hash table, but depends on load factor).
Key Data Types	Any type that supports comparison (`<`).	Any type that supports comparison (`<`).	Any type that supports hashing (`std::hash` must be defined).
Use Case	When unique keys are needed, and sorting is required.	When duplicate keys are needed.	When fast access and order don't matter.
Example Code	`map<string, int> m; m["apple"] = 5;`	`multimap<string, int> m; m.insert({"apple", 5});`	`unordered_map<string, int> m; m["apple"] = 5;`

Sets in C++

1. `std::set` (Ordered Set)

- **Definition**: A `set` is a collection of unique elements, stored in **sorted order** based on the key.
- **Internal Structure**: It is typically implemented as a **red-black tree** (balanced binary search tree).
- **Key**: Elements in a set are unique, and they are stored in sorted order.
- **Operations**:
    - `s.find(key)`: Searches for the key. Returns an iterator to the element if found, or `s.end()` if not found.
    - `s.insert(value)`: Inserts the element in sorted order. If the element already exists, the insertion is ignored (since sets do not allow duplicates).
    - `s.erase(key)`: Removes the element with the given key. It returns the number of elements erased (either 0 or 1 for a set).

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

int main() {
    set<int> s;
    s.insert(10);
    s.insert(20);
    s.insert(30);

    // Find an element
    if (s.find(20) != s.end()) {
        cout << "20 is present in the set." << endl;
    }

    // Erase an element
    s.erase(10);

    for (int elem : s) {
        cout << elem << " ";
    }

    return 0;
}
```

```
20 is present in the set.
20 30
```

2. `std::unordered_set` (Unordered Set)

- **Definition**: An `unordered_set` is a collection of unique elements, but unlike `set`, the elements are **not sorted**. It is based on **hash tables**.
- **Internal Structure**: Internally implemented using a **hash table**, providing fast average time complexity for operations.
- **Key**: Elements in an unordered set are unique, but they are stored in an **unordered manner**.
- **Operations**:
    - `s.find(key)`: Searches for the key. Returns an iterator to the element if found, or `s.end()` if not found.
    - `s.insert(value)`: Inserts the element. If the element already exists, it will not be inserted again.
    - `s.erase(key)`: Removes the element with the given key.

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

int main() {
    unordered_set<int> s;
    s.insert(10);
    s.insert(20);
    s.insert(30);

    // Find an element
    if (s.find(20) != s.end()) {
        cout << "20 is present in the unordered set." << endl;
    }

    // Erase an element
    s.erase(10);

    for (int elem : s) {
        cout << elem << " ";
    }

    return 0;
}
```

```
20 is present in the unordered set.
20 30
```

3. `std::multiset` (Multi-key Set)

- **Definition**: A `multiset` is a set that allows **duplicate elements** but stores them in **sorted order** (like `std::set`).
- **Internal Structure**: It is also implemented as a **red-black tree** (balanced binary search tree), but it allows multiple occurrences of the same element.
- **Key**: Elements can appear multiple times.
- **Operations**:
    - `s.find(key)`: Finds and returns an iterator to the first occurrence of the key (if found).
    - `s.insert(value)`: Inserts the element in sorted order, even if it is a duplicate (unlike a `set`).
    - `s.erase(key)`: Removes all occurrences of the element with the given key.

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

int main() {
    multiset<int> ms;
    ms.insert(10);
    ms.insert(20);
    ms.insert(10);
    ms.insert(30);

    // Find an element
    if (ms.find(10) != ms.end()) {
        cout << "10 is present in the multiset." << endl;
    }

    // Erase an element (removes all occurrences of 10)
    ms.erase(10);

    for (int elem : ms) {
        cout << elem << " ";
    }

    return 0;
}
```

```
10 is present in the multiset.
20 30
```

Comparison Summary (Set, Unordered Set, Multiset)

Feature	`std::set` (Ordered Set)	`std::unordered_set` (Unordered Set)	`std::multiset` (Multi-key Set)
Element Uniqueness	Unique elements only.	Unique elements only.	Allows duplicate elements.
Ordering	Sorted order (based on key).	Unordered (based on hash).	Sorted order (based on key).
Internal Structure	Red-Black Tree (Balanced BST).	Hash Table.	Red-Black Tree (Balanced BST).
Insert Complexity	$O(\log n)$	$O(1)$ average, $O(n)$ worst case	$O(\log n)$
Search Complexity	$O(\log n)$	$O(1)$ average, $O(n)$ worst case	$O(\log n)$
Remove Complexity	$O(\log n)$	$O(1)$ average, $O(n)$ worst case	$O(\log n)$
Duplicate Elements	No duplicates.	No duplicates.	Allows duplicates.
Example Use Case	When elements need to be unique and ordered.	When elements need to be unique, but order doesn't matter.	When you need duplicate elements in a sorted collection.

Although I've covered key topics like maps, sets, vectors, and more, I highly recommend exploring this playlist to truly understand the nuances. The explanations are clear, concise, and provide real-world problem-solving approaches. While learning, don't just copy; think critically and implement your own ideas. Experiment with variations, challenge yourself with new problems, and make mistakes—that's where real learning happens. This journey is about building confidence and a deep understanding of concepts that you can apply effectively. Stay consistent, and you'll master DSA in no time! Keep solving and growing.

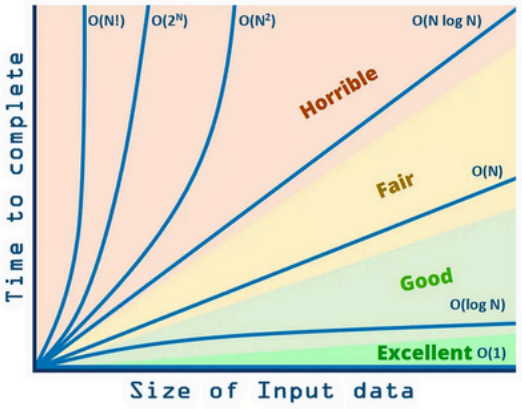
Time and Space Complexity:

Algorithm Efficiency:

- Algorithms are the heart of data structures, helping solve problems systematically.
- The efficiency of an algorithm is measured in two key aspects:
    - **Time Complexity:** How much time an algorithm takes based on the input size.
    - **Space Complexity:** How much memory an algorithm uses during its execution.

Time Complexity:

- Time complexity measures the time an algorithm takes to complete, depending on the size of the input.
- It is commonly expressed using Big-O notation (e.g., O(1), O(logN), O(N), etc.), which describes the algorithm's growth rate as input increases.



Space Complexity:

- Space complexity measures the amount of memory used by the algorithm.
- It includes:
    1. **Auxiliary Space:** Extra space used by the algorithm (temporary variables, function calls, etc.).
    2. **Input Space:** Memory needed to store the input data.

Tip: While time complexity often gets more attention, space efficiency is equally important for handling larger inputs or limited resources.

Types of Time Analysis:

1. **Worst Case:**
    - Consider the maximum time an algorithm might take for any input.
    - Example: For Linear Search, the worst case is when the target is the last element or not present at all.
2. **Best Case:**
    - Consider the minimum time an algorithm might take for any input.

    - Example: For Linear Search, the best case is when the target is the first element.
3. **Average Case:**
    - Estimates the average time the algorithm will take across all possible inputs.
    - This is useful for understanding the overall performance in realistic scenarios.

Beginner Tips for Time and Space Complexity:

1. **Focus on Big-O:** Learn to identify and compare different complexities like O(1), O(N), O(N^2), etc. It's key to writing efficient code.
2. **Trade-offs Exist:** Sometimes, reducing time complexity might increase space usage and vice versa.
3. **Practice Makes Perfect:** Analyzing algorithms becomes easier as you practice solving problems and implementing data structures.