**Hash Map Using STL**

*“ Hash Map is a Data Structure which Provides Facility to Perform Operations like Insertion, Deletion, Traversal, etc in Constant Time “*

* Hash Map is Like a table that maps a Unique Key to a Value.
* Hash Map can be created using :

1. unordered\_map<key-datatype, value-datatype>
2. map<key-datatype, value-datatype>

Simply,

Hash Map == Unordered\_map

Balanced BST == map

std::unordered\_map and std::map are two commonly used container classes in the C++ Standard Library (STL) for implementing associative arrays.

**Map**

map is an implementation of a balanced binary search tree, also known as a Red-Black tree, that orders its elements based on their keys. This means that elements in a std::map are **stored in a sorted order based on their keys**, which allows for efficient searching and traversal of the elements. However, the time complexity for insertion and deletion operations is logarithmic, which means that they are **slower compared to the unordered version.**

**Unordered\_map**

unordered\_map, on the other hand, is an implementation of a hash table, where the **elements are stored in a random order based on their hash** **values**. The unordered nature of this container makes it **faster for insertion, deletion, and search operations**, with an average time complexity of O(1). However, the **order of the elements is not guaranteed to be sorted.**

**unordered\_map<key-datatype, value-datatype>**

***Creation E.g.***

unordered\_map<string, int> table;

***Insertion Type 1 E.g.***

pair<string, int> p = make\_pair(“C++”, 1);

table.insert(p);

***Insertion Type 2 E.g.***

pair<string, int> p(“C++”,1);

table.insert(p);

***Insertion Type 3 E.g.* (Easy way)**

table[“C++”] = 1;

**NOTE :**

* *Re-Inserting to already available key (with/without value ) will Overwrite the key value*

***Insertion Type 4 E.g (vector<int> as value, int as key)***

unordered\_map<int, vector<int>> hashMap;

hashMap[0].push\_back(1);

hashMap[0].push\_back(2);

hashMap[1].push\_back(3);

…

***Searching Key and Fetching value Type 1 E.g.***

table[“C++”]

***Searching Key and Fetching value Type 2 E.g.***

table.at[“C++”]

**Searching Key (IF Unique Key Exist)**

table.**find**(key) == **table**.end()

**NOTE**

* Here **find** method returns an iterator
* **Reminder**: The following snippet can be useful to remove duplicates from an array/string without affecting the relative ordering and maintaining a proper sequence **without** any **sort** method (Hash-map and arrays can be used to achieve this).

***Obtain the size of the map E.g.***

table.size()

***Check The Presence of the key E.g.***

table.count(“C++”)

***Erase a key from the map E.g.***

table.erase(“C++”)

***Create a map Iterator E.g.***

unordered\_map<string, int> :: iterator it***;***

***Traverse a map using Iterator E.g.***

it = table.begin();

while(it != table.end()){

cout<<”Key = “<<it->first<<”, Value = “<<it->second;

it++;

}

**NOTE :**

* **it->first -** Provides Key
* **it->second -** Provides Key’s - value

**Unordered\_map Key Findings:**

1. When we declare unordered\_map, all **keys** are automatically supplied **value** at beginning based on the **type** of **value** :

* **char, int** - ‘\0’ (equivalent to 0)
* **string** - **“” (**empty string)

So in order to check whether we already got a value in a map or not, we can do something like this (suppose map as mp and key of type **int** being accessed is k):

If ( mp[k] == ‘\0’) {

// do something

}

* Here the if statement will run if map’s specific key is not supplied a value apart from its initialization (beginning) step.
* We don’t have to directly compare with zero as it can create conflict between determining actual value and NULL (not initialized) character.

**HashMap Key value pair as int and vector<int>**

#include <iostream>

#include <vector>

#include <unordered\_map>

using namespace std;

int main() {

unordered\_map<int, vector<int>> hashMap;

hashMap[0].push\_back(1);

hashMap[0].push\_back(2);

hashMap[0].push\_back(3);

for(auto x : hashMap[0]) {

cout << x << " ";

}

return 0;

}

**NOTE**

* map<datatype1, datatype2> can be useful when we want things to be remain sorted based on their lexicographical order which is not a thing in unordered\_map