unordered_map in C++ STL

The **unordered_map** is an associated container that stores elements formed by a combination of key-value pair. The key value is used to uniquely identify the element and mapped value is the content associated with the key. Both key and value can be of any type of predefined or user-defined.

Internally unordered_map is implemented using Hash Table, the key provided to map are hashed into indices of a hash table that is why the performance of data structure depends on hash function a lot but on an average the cost of **search, insert and delete** from hash table is O(1).

The elements in the unordered_map are unordered and are not stored in any specific order like Map.

```

// C++ program to demonstrate functionality of unordered_map

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

int main()
{
    // Declaring umap to be of <string, int> type
    // key will be of string type and mapped value will
    // be of double type
    unordered_map<string, int> umap;

    // inserting values by using [] operator
    umap["GeeksforGeeks"] = 10;
    umap["Practice"] = 20;
    umap["Contribute"] = 30;

    // Traversing an unordered map
    for (auto x : umap)
      cout << x.first << " " << x.second << endl;

}

```

Run

The operators **"["** and **"]"** are called member access operators. In the above program when these operators are used with the map container, it returns a reference to the value associated with the key enclosed within these operators.

If the key does not already exist in the container, it will insert the new key and will return a reference to the newly inserted value.

Searching an element in unordered_map: We can search if an element exists in the unordered_map or not using the find() function. The find() function accepts the key as a parameter and returns an iterator pointing to the key-value pair if it exists otherwise it returns an iterator pointing to the past-the-end element.

Note: Since the find() function returns an iterator pointing to the key-value pair, we can easily access both the key and the value associated to it as:

```
Key = iterator->first;
Value = iterator->second;
```

Below program illustrates this:

```

// C++ program to demonstrate functionality
// of unordered_map

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

int main()
{
    // Declaring umap to be of <string, int> type
    // key will be of string type and mapped value will
    // be of int type
    unordered_map<string, int> umap;

    // inserting values by using [] operator
    umap["GeeksforGeeks"] = 10;
    umap["Practice"] = 20;
    umap["Contribute"] = 30;

    // Searching for the key "Practice"
    if(umap.find("Practice")!=umap.end())
        cout<<"The key Practice Found!\n";
    else
        cout<<"The key Practice Not Found!\n";

    // Accessing key value pair returned by find()
    auto it = umap.find("Practice");
    if(it!=umap.end())
        cout<<"Key is: "<<it->first<<", "
```

Run

Some more Functions of unordered_map

- **begin()**: Returns an iterator pointing to the first element in the container in the unordered_map container.
- **end()**: Returns an iterator pointing to the position past the last element in the container in the unordered_map container.
- **count()**: Counts the number of elements present in an unordered_map with a given key. Since keys are unique in an unordered_map, so it is basically used as a replacement of find() sometimes to check if a key-value pair with a given key exists in the unoredered_map or not.
- **size()**: The size function is used to find the total size of the unorered_map. That is, the total number of elements present in the unordered_map.
- **erase()**: The erase function is used to erase a particular element from the unordered_map. It can also be used to erase a range of elements from the map.

Below program illustrate the working of the above functions:

```

// C++ program to demonstrate functionality
// of unordered_map

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

int main()
{
    // Declaring umap to be of <string, int> type
    // key will be of string type and mapped value
    // will be of int type
    unordered_map<string, int> umap;

    // inserting values by using [] operator
    umap["GeeksforGeeks"] = 10;
    umap["Practice"] = 20;
    umap["Contribute"] = 30;

    // Searching if the key "Practice" exists
    // using count()
    if(umap.count("Practice")>0)
        cout<<"The key Practice Found!\n";
    else
        cout<<"The key Practice Not Found!\n";

    // Printing size of the map before erasing
    cout<<"Size before erasing: "<<umap.size()<<"\n";

```

Run

Time Complexities

The operations such as, begin(), end(), size(), empty() works in **O(1) time in worst case**, where as, the other important functions like count(), find(), erase(key), insert(), [], at() works in **O(1) time in average**.

- Sample Problem: Find frequencies of elements of array

Problem: Given an array, find the frequencies of each element of the given array.

```
Input: arr[] = {10, 20, 20, 10, 10, 5, 15}
Output:
10 - 3
20 - 2
5 - 1
15 - 1

Input: arr[] = {1, 1, 1, 2, 2, 2, 2, 3}
Output:
1 - 3
2 - 4
3 - 1
```

Approach:

Step 1: Create a hashmap or unordered_map.

Step 2: Traverse the array to store the frequency of each element of the array in map.

Step 3: Finally traverse the map and print the map and print the key and its value one by one

Time Complexity: Time taken to access the individual frequency is O(1). Since the map does not accept duplicate elements so the time taken will be equal to the size of hashmap. In the worst case, it can be O(N) if all the elements of an array are distinct.

```cpp

// Program to find the frequency
// of each element of an array
#include <bits/stdc++.h>
using namespace std;

// Method to find the frequency
// of each element
void printFrequencies(int arr[], int n)
{
    // Creating an unordered_map
    unordered_map<int, int> mp;

    // Counting the frequency
    for (int i = 0; i < n; i++)
        mp[arr[i]]++;

    // Printing the frequency of
    // each element
    for (auto x : mp)
        cout
            << x.first << "-" << x.second << endl;
}

// Drivers method
int main()
{
    int arr[] = { 10, 20, 20, 10, 10, 5, 15 };
    int n = 8;

```

Run

Output:

```
10 20 20 10 10 5 15 0
15-1
5-1
10-3
20-2
```

Sample Problem: Find the winner of election

Sample Problem: Find the winner of election

Problem: Given an array of names of candidates in an election, the task is to print the candidates receiving the maximum vote. A candidate name in the array represents a vote cast to the candidate. Assume that there are no ties.

```
Input: names[] = {"John", "Rohan",
" Jackie", "Rohan", "Raju"}
Output: Rohan
Explanation:  Rohan is the winner, as his
name appears twice in the array.

Input: names[] = {"Akbar", "Babur",
 "Birbal", "Ram", "Ram", }
Output: Ram
Explanation:  Ram is the winner, as his
name appears twice in the array.
```

Approach: We insert all votes in a hash map and keep track of counts of different names. Finally, we traverse the map and print the person with maximum votes.

Time Complexity: Time taken for inserting elements to map is O(N) and for traversing map is also O(N). This overall time complexity will be **O(n)**.

```cpp

// C++ program to find the winner
// of election
#include <bits/stdc++.h>
using namespace std;

// Function to find the candidate
// receiving the maximum vote
string findWinner(string votes[], int n)
{
    unordered_map<string, int> mp;
    string winner = " ";

    // Inserting the name and occurences
    for (int i = 0; i < n; i++) {
        mp[votes[i]]++;
        int mx = 0;

        // Comparing the vote ount
        for (
            auto it = mp.begin();
            it != mp.end(); it++) {
            if (it->second > mx) {
                mx = it->second;
                winner = it->first;
            }
        }
    }
    return winner;
}
```

Run

Output:

```
Rohan
```

Sample Problem: Find the winner of election if there is a tie

Problem: Given an array of names of candidates in an election, the task is to print the candidates receiving the maximum vote. A candidate name in the array represents a vote cast to the candidate. If there is a tie, print the lexicographically smaller name.

```
Input: names[] = {"John", "Rohan",
 "Jackie", "Rohan", "Raju", "Jackie"}
Output: Jackie
Explanation:  Tie occurs between Rohan and
```

```
Jackie but Jackie is the winner, since J comes before R in a
dictionary.

Input: names[] = {"Akbar", "Babur",
 "Birbal", "Ram", "Ram", "Birbal"}
Output: Birbal
Explanation:  Tie occurs between Ram and
Birbal but Birbal is the winner, since B comes before R in a
dictionary.
```

Approach: We insert all the votes in a hash map and keep track of counts of different names. Finally, we traverse the map and print the person with maximum votes. Since we have to handle the case of a tie, we would also have to compare the names. In case of same maximum votes print the name which comes first lexicographically.

Time Complexity: Time taken for inserting elements to map is O(N) and for traversing map is also O(N). This overall time complexity will be **O(n)**.

```cpp

// C++ program to find the winner
// of election and handle the tie case
#include <bits/stdc++.h>
using namespace std;

// Function to find the candidate
// receiving the maximum vote
string findWinner(string votes[], int n)
{
    // Creating an unordered_map
    unordered_map<string, int> mp;

    // Inserting the name and occurences
    for (int i = 0; i < n; i++) {
        mp[votes[i]]++;
    }
    string winner = "";
    int mx = -1;

    // Finding the winner with maximum
    // votes
    for (
        auto it = mp.begin();
        it != mp.end(); it++) {
        // Handling the tie case
        if (
            (
                it->second > mx)
            || ((it->second == mx)
```

Run