Unordered_Map

unordered_map is an associative container available in the C++ Standard Template Library (STL). It is implemented as a hash table, which provides fast access to elements based on their keys. **unordered_map** allows for efficient insertion, deletion, and retrieval of elements. It is commonly used when fast lookup based on keys is required, and the order of elements is not important.

Key Features of unordered_map:

1. **Fast Lookup**: **unordered_map** provides fast access to elements based on their keys. The average time complexity for insertion, deletion, and retrieval operations is constant on average, i.e., O(1), making it suitable for scenarios where fast lookup is required.
2. **Unique Keys**: **unordered_map** stores elements as key-value pairs, where each key is unique. Duplicate keys are not allowed. If an element with an existing key is inserted, it will replace the existing element.
3. **Hashing**: **unordered_map** uses a hash function to map keys to indices in the underlying hash table. This enables efficient lookup and retrieval of elements.
4. **Iterators**: **unordered_map** provides iterators to traverse through its elements. Iterators allow you to access, modify, and remove elements in the container.
5. **Automatic Memory Management**: **unordered_map** manages memory automatically. When elements are inserted or removed, the container takes care of memory allocation and deallocation.
6. **Dynamic Size**: **unordered_map** can dynamically grow and shrink to accommodate the number of elements inserted or removed. It efficiently manages memory to optimize space usage.
7. **No Ordering**: Elements in **unordered_map** are not stored in any particular order. If you need to maintain a specific order, you should consider using **std::map** instead.

Tips for Using unordered_map:

1. **Choose the Right Hash Function**: If the default hash function provided by C++ is not sufficient for your specific use case, you can define a custom hash function to improve performance.
2. **Consider Load Factor**: The load factor is the ratio of the number of elements to the number of buckets in the hash table. A higher load factor can result in more collisions, impacting performance. You can adjust the load factor by specifying a different initial number of buckets during **unordered_map** construction.
3. **Beware of Hash Collisions**: Hash collisions occur when different keys produce the same hash value. Although **unordered_map** handles collisions internally, a high

number of collisions can degrade performance. Choosing a good hash function and an appropriate initial number of buckets can help mitigate collisions.

4. **Avoid Modifying Keys**: Modifying keys while they are stored in the **unordered_map** can lead to undefined behavior. If you need to update a key, it's recommended to remove the element and insert a new one with the updated key.
5. **Prefer at() over [] for Error Handling**: When accessing elements, using **at(key)** provides bounds checking and throws an exception if the key is not found. On the other hand, using **[]** to access an element with a non-existing key will insert a new element with a default value.

Here are the key differences between unordered_map and map containers in C++:

1. **Ordering of Elements**: The elements in a **map** are ordered based on their keys, using a comparison function by default. On the other hand, an **unordered_map** does not maintain any particular order of elements.
2. **Lookup Time Complexity**: The time complexity for insertion, deletion, and search operations in **map** is logarithmic (O(log n)) due to the underlying balanced binary search tree structure. In contrast, **unordered_map** provides constant-time complexity (O(1)) for these operations on average, making it faster in most cases.
3. **Hashing and Equality Comparison**: **unordered_map** uses a hash function to compute the hash value of keys and an equality comparison to handle collisions. This allows for efficient lookup based on hashed values. In contrast, **map** relies on the comparison function to maintain the ordering of elements.
4. **Storage Overhead**: **unordered_map** typically requires more memory compared to **map**. This is because **unordered_map** needs additional space for maintaining the hash table and handling collisions. **map** uses a balanced binary search tree, which requires less memory overhead.
5. **Iterators**: Iterating through elements in a **map** follows the order determined by the keys. In contrast, the iteration order of an **unordered_map** is not defined as it depends on the hash values and the internal structure of the hash table.
6. **Key Type Requirements**: **map** requires the key type to support comparison operations (either using a comparison function or the less-than operator). **unordered_map** additionally requires the key type to support hashing and equality comparison.
7. **Performance Trade-offs**: The choice between **map** and **unordered_map** depends on the specific requirements of the application. If the order of elements or efficient range-based iteration is important, **map** is a good choice. On the other hand, if fast lookup and insertion times are critical, and the order of elements is not a concern, **unordered_map** provides better performance.

Time and Space Complexity of the functions used:

1. **at()**: Accesses the element with a given key and throws an exception if the key does not exist.
    - Time Complexity: O(1)
    - Space Complexity: O(1)
2. **find()**: Searches the container for an element with a specific key and returns an iterator to it. If the element is not found, it returns an iterator to **end()**.
    - Time Complexity: O(1) on average
    - Space Complexity: O(1)
3. **count()**: Returns the number of elements with a specific key. Since **unordered_map** allows only unique keys, the return value will be either 0 or 1.
    - Time Complexity: O(1)
    - Space Complexity: O(1)
4. **erase()**: Removes an element from the container with the specified key.
    - Time Complexity: O(1)
    - Space Complexity: O(1)
5. **clear()**: Removes all elements from the container.
    - Time Complexity: O(N)
    - Space Complexity: O(1)
6. **empty()**: Checks if the container is empty.
    - Time Complexity: O(1)
    - Space Complexity: O(1)
7. **size()**: Returns the number of elements in the container.
    - Time Complexity: O(1)
    - Space Complexity: O(1)
8. **printMap():** Prints the elements of the map in ascending order of keys.
    - Time Complexity: O(n)
    - Space Complexity: O(1)