Map

1. Introduction to STL Map: STL Map is one of the essential components of the C++ Standard Template Library (STL). It is a data structure that represents an associative container that stores elements in a key-value pair. The keys are unique, and they are used to access and retrieve the associated values quickly. The Map is implemented as a balanced binary search tree, typically a Red-Black Tree, which ensures efficient searching, insertion, and deletion operations.

2. Key Features and Characteristics:

- **Associative Container:** STL Map stores elements in a sorted order, based on the keys. This allows for efficient searching and retrieval using keys.
- **Unique Keys:** Each key in the map is unique. Duplicate keys are not allowed, ensuring a one-to-one relationship between keys and values.
- **Balanced Binary Search Tree:** The underlying data structure is usually a balanced binary search tree, ensuring logarithmic complexity for operations.
- **Logarithmic Complexity:** Operations like insertion, deletion, and search have a time complexity of O(log N), where N is the number of elements in the map.
- **Ordered Elements:** Elements in the map are always sorted based on the key's sorting criterion (by default, keys are sorted in ascending order).
- **Iterators:** STL Map supports bidirectional iterators that can be used to traverse the elements in the container.
- **Value Modification:** The value associated with a key can be modified, but the key itself remains constant.

3. Performace Consideration:

- **Search Complexity:** Searching in a map has a time complexity of O(log N) since it uses a balanced binary search tree.
- **Insertion/Deletion Complexity:** Insertion and deletion operations also have a time complexity of O(log N) since the tree structure must be maintained.
- **Memory Overhead:** STL Map has a higher memory overhead compared to simpler containers like **std::unordered_map**.
- **Choosing the Right Container:** Consider using STL Map when you need a sorted associative container with u

Key differences between Map and Unordered_map:

1. **Ordering**: **std::map** is an ordered container where the elements are sorted based on the keys in ascending order by default. On the other hand, **std::unordered_map** is an

unordered container where the elements are not sorted and the order of elements may vary.

2. **Data Structure**: **std::map** typically uses a self-balancing binary search tree (usually a red-black tree) to store its elements, which provides efficient logarithmic time complexity for insertion, deletion, and search operations. In contrast, **std::unordered_map** uses a hash table to store its elements, which provides constant time complexity on average for insertion, deletion, and search operations.
3. **Performance**: **std::unordered_map** generally offers faster average case performance for large datasets compared to **std::map** due to its constant-time complexity for most operations. However, **std::map** might perform better in scenarios where maintaining a sorted order of keys or performing range-based operations is required.
4. **Lookup Time**: **std::unordered_map** provides constant-time complexity (O(1)) for lookup operations, while **std::map** offers logarithmic time complexity (O(log n)).
5. **Iterator Stability**: Iterators of **std::map** remain valid even after modifications to the container (e.g., insertions or deletions). On the other hand, for **std::unordered_map**, modifications to the container might invalidate iterators, as the hash table structure may change during rehashing.
6. **Key Requirements**: **std::map** requires the key type to support comparison operators (**<**, **>**, **==**, etc.), as it relies on ordering. In contrast, **std::unordered_map** requires the key type to have a hash function defined and support equality comparison (**==**).
7. **Memory Overhead**: **std::unordered_map** typically has a higher memory overhead compared to **std::map**. This is because **std::unordered_map** needs additional memory for storing hash table buckets and maintaining the hash function.
8. **Iterator Invalidations**: Insertions and deletions in **std::map** can cause iterators to be invalidated if they point to the modified elements or elements after the modification point. In **std::unordered_map**, insertions and deletions may cause rehashing, which can potentially invalidate all iterators.

The Time and Space complexity of the functions used:

1. **printMap(map<string, int>& ages)**
    - Description: Prints the elements of the map in ascending order of keys.
    - Time Complexity: O(n)
    - Space Complexity: O(1)
2. **printMapReverse(map<string, int>& ages)**
    - Description: Prints the elements of the map in descending order of keys.
    - Time Complexity: O(n)

- Space Complexity: O(1)

3. **map::empty()**
    - Description: Checks if the map is empty.
    - Time Complexity: O(1)
    - Space Complexity: O(1)
4. **map::operator[]**
    - Description: Accesses the value associated with the given key or inserts a new key-value pair if the key doesn't exist.
    - Time Complexity: Average case: O(log n), Worst case: O(log n)
    - Space Complexity: O(log n)
5. **map::insert**
    - Description: Inserts a new key-value pair into the map.
    - Time Complexity: Average case: O(log n), Worst case: O(log n)
    - Space Complexity: O(log n)
6. **map::emplace**
    - Description: Constructs and inserts a new key-value pair into the map using perfect forwarding.
    - Time Complexity: Average case: O(log n), Worst case: O(log n)
    - Space Complexity: O(log n)
7. **map::size()**
    - Description: Returns the number of elements in the map.
    - Time Complexity: O(1)
    - Space Complexity: O(1)
8. **map::count**
    - Description: Counts the number of elements with a given key.
    - Time Complexity: Average case: O(log n), Worst case: O(log n)
    - Space Complexity: O(1)
9. **map::begin()**
    - Description: Returns an iterator to the beginning of the map.
    - Time Complexity: O(1)

- Space Complexity: O(1)

10. **map::end()**
    - Description: Returns an iterator to the end of the map.
    - Time Complexity: O(1)
    - Space Complexity: O(1)
11. **map::erase**
    - Description: Erases an element with the given key from the map.
    - Time Complexity: Average case: O(log n), Worst case: O(log n)
    - Space Complexity: O(1)
12. **map::find**
    - Description: Finds an element with the given key in the map.
    - Time Complexity: Average case: O(log n), Worst case: O(log n)
    - Space Complexity: O(1)
13. **map::clear**
    - Description: Removes all elements from the map.
    - Time Complexity: O(n)
    - Space Complexity: O(1)
14. **Map::at**
    - Description: 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)