Set

STL Set is an associative container that stores a sorted sequence of unique elements. It is implemented as a self-balancing binary search tree (usually a Red-Black Tree) and provides the following key features:

1. **Ordered Elements**: Set maintains its elements in a specific order defined by the comparison operator. By default, it stores elements in ascending order.
2. **Unique Elements**: Set ensures that all elements are unique. If an element already exists in the set, it will not be inserted again.
3. **Fast Search**: Set provides fast search operations (O(log n)) for finding elements within the container.
4. **Efficient Insertion and Removal**: Insertion and removal of elements in a set have a time complexity of O(log n), where n is the number of elements in the set.
5. **Iterators**: Set supports bidirectional iterators, allowing you to traverse the elements in ascending or descending order.
6. **Range Operations**: Set supports various range-based operations such as finding elements, counting elements, and finding the bounds of a value within the set.
7. **Automatic Sorting**: Set automatically maintains the elements in sorted order, ensuring efficient searching and traversal.
8. **No Random Access**: Set does not provide random access to elements. You can only access elements using iterators.

Use STL Set when:

- You need to maintain a sorted sequence of unique elements.
- You frequently perform searching, insertion, or removal of elements.
- Ordering of elements is important.
- You want to eliminate duplicates automatically.

The key differences between std::set and std::unordered_set in C++ are as follows:

1. **Internal Data Structure**: **std::set** is implemented as a self-balancing binary search tree (usually a Red-Black Tree), while **std::unordered_set** is implemented as a hash table.
2. **Ordering**: **std::set** maintains its elements in a specific order defined by the comparison operator. It provides ordered access to its elements. On the other hand, **std::unordered_set** does not maintain any specific order of elements. The elements

are arranged based on their hash values, resulting in an arbitrary order during iteration.

3. **Performance**: The performance characteristics of **std::set** and **std::unordered_set** differ:

    - **std::set** has a slower average time complexity for insertion, removal, and search operations, typically O(log n), where n is the number of elements in the set.
    - **std::unordered_set** has faster average time complexity for insertion, removal, and search operations, usually O(1) on average, but with a worst-case time complexity of O(n) for certain operations.

The choice between the two containers depends on the specific use case and the importance of performance characteristics.

4. **Duplicates**: **std::set** allows only unique elements in the container. If an element already exists, it will not be inserted again. On the other hand, **std::unordered_set** enforces uniqueness using hash values and equality comparison. Duplicate elements are automatically eliminated.

5. **Iterators**: Both **std::set** and **std::unordered_set** support iterators, but the nature of iteration differs:

    - **std::set** supports bidirectional iterators, allowing traversal of elements in ascending or descending order.
    - **std::unordered_set** supports forward iterators, allowing traversal of elements in an arbitrary order determined by their hash values.

6. **Element Access**: **std::set** allows access to elements through iterators but does not provide direct random access. On the other hand, **std::unordered_set** does not support direct element access by index or position.

7. **Storage Overhead**: **std::unordered_set** may have a higher storage overhead compared to **std::set** due to the additional memory required for the hash table structure.

When to use **std::set**:

- When you need to maintain elements in a sorted order.
- When ordered access to elements is important.
- When you want to eliminate duplicates automatically.
- When you can tolerate slightly slower insertion, removal, and search operations.

When to use **std::unordered_set**:

- When the ordering of elements is not important.

- When you need faster average insertion, removal, and search operations.
- When you don't need automatic sorting or duplicate elimination.
- When performance is crucial and you can tolerate a higher worst-case time complexity for certain operations.

The Time And Space Complexity of the functions used:

1. **printSet**:
   - One-liner: Prints the elements of a set.
   - Time complexity: O(n), where n is the size of the set.
   - Space complexity: O(1).
2. **printSetReverse**:
   - One-liner: Prints the elements of a set in reverse order.
   - Time complexity: O(n), where n is the size of the set.
   - Space complexity: O(1).
3. **main**:
   - One-liner: Demonstrates various operations on a set.
   - Time complexity: Varies based on the operations performed.
   - Space complexity: O(n), where n is the size of the set (due to the storage of elements in the set).
4. **set.empty**:
   - One-liner: Checks if the set is empty.
   - Time complexity: O(1).
   - Space complexity: O(1).
5. **set.insert**:
   - One-liner: Inserts an element into the set.
   - Time complexity: O(log n), where n is the size of the set (due to the self-balancing binary search tree implementation).
   - Space complexity: O(1).
6. **set.count**:
   - One-liner: Returns the number of occurrences of a specific element in the set.

- Time complexity: O(log n), where n is the size of the set.
- Space complexity: O(1).

7. **set.erase**:
    - One-liner: Removes an element from the set.
    - Time complexity: O(log n), where n is the size of the set.
    - Space complexity: O(1).
8. **set.find**:
    - One-liner: Finds the iterator pointing to the first occurrence of a specific element in the set.
    - Time complexity: O(log n), where n is the size of the set.
    - Space complexity: O(1).
9. **set.clear**:
    - One-liner: Removes all elements from the set.
    - Time complexity: O(n), where n is the size of the set.
    - Space complexity: O(1).