Unordered_Set

The **unordered_set** container in C++ is an implementation of an unordered associative container that stores unique elements in no particular order. It is part of the Standard Template Library (STL) and is defined in the **<unordered_set>** header file.

Key Features:

- **Unique Elements: unordered_set** only allows unique elements. It automatically removes duplicate elements when inserted.
- **Fast Lookup:** The main advantage of **unordered_set** is its fast average constant-time lookup. It uses a hash-based data structure, which provides efficient element retrieval.
- **No Ordering:** Unlike **set**, **unordered_set** does not maintain the elements in a specific order. Elements are stored based on their hash values, resulting in a random order of elements.
- **Dynamic Size:** The size of an **unordered_set** can dynamically change as elements are inserted or erased.
- **Iterators: unordered_set** provides iterators for traversing the elements. However, note that the order of elements is not guaranteed.
- **No Duplicate Elements: unordered_set** enforces uniqueness of elements. If an element already exists in the set, it will not be inserted again.
- **Hash Function: unordered_set** uses a hash function to determine the storage position of each element. The default hash function for basic types is provided, but custom hash functions can be used for user-defined types.
- **No Element Modification:** Elements in an **unordered_set** cannot be modified directly. You need to remove an element and insert a modified version if needed.
- **No Reverse Iteration:** Unlike **set**, **unordered_set** does not provide reverse iterators (**rbegin** and **rend**) to iterate in reverse order.
- **Space Overhead: unordered_set** uses additional memory to store hash tables, resulting in a higher space overhead compared to **set**.

Use Cases:

- **unordered_set** is useful when fast element lookup is required and the order of elements is not important.
- It is commonly used for implementing algorithms that require checking for existence or uniqueness of elements efficiently.
- It can be used in various scenarios such as implementing hash tables, frequency counting, removing duplicates, etc.

Note: If you require the elements to be sorted or need reverse iteration, consider using the **set** container instead.

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

1. **Ordering**: **unordered_set** does not maintain any particular order of elements, while **set** stores elements in a specific sorted order based on a comparison function or the default less-than operator. In **unordered_set**, the elements are stored based on their hash values, resulting in a random order.
2. **Lookup Time Complexity**: **unordered_set** provides faster average case constant-time lookup (O(1)) for operations like insertion, deletion, and search. On the other hand, **set** has a logarithmic time complexity (O(log n)) for these operations due to its ordered nature.
3. **Duplicate Elements**: **unordered_set** only allows unique elements. If an element already exists, it will not be inserted again. In contrast, **set** allows only unique elements by default, and inserting a duplicate element has no effect.
4. **Iterators**: **unordered_set** provides forward iterators to traverse the elements. However, the order of elements is not guaranteed. On the other hand, **set** provides bidirectional iterators, allowing both forward and reverse iteration over the elements, maintaining their sorted order.
5. **Space Overhead**: **unordered_set** uses additional memory to store hash tables, resulting in a higher space overhead compared to **set**.
6. **Element Modification**: Elements in both **unordered_set** and **set** are immutable. You cannot modify an element directly. To modify an element, you need to remove it from the container and insert the modified version.
7. **Implementation**: **unordered_set** uses a hash-based data structure to store elements, while **set** typically uses a balanced binary search tree (such as a red-black tree) for efficient ordering and retrieval.

Choosing Between unordered_set and set:

- Use **unordered_set** when you need fast average case constant-time lookup, and the order of elements is not important.
- Use **set** when you require elements to be sorted in a specific order and need operations like range queries or ordered iteration.
- If you need both fast lookup and ordered iteration, you can consider using an **unordered_set** for fast lookup and copying its elements to a **set** for ordered iteration when needed.

Time and Space Complexity of the functions used:

1. **unordered_set::empty()** - Check if the unordered_set is empty.
    - Time Complexity: O(1)
    - Space Complexity: O(1)
2. **unordered_set::insert()** - Insert elements into the unordered_set.
    - Time Complexity: Average-case O(1), Worst-case O(n) due to hash collisions.
    - Space Complexity: O(1) per element inserted.
3. **printUnorderedSet()** - Print the elements of the unordered_set.
    - Time Complexity: O(n)
    - Space Complexity: O(1)
4. **unordered_set::size()** - Get the number of elements in the unordered_set.
    - Time Complexity: O(1)
    - Space Complexity: O(1)
5. **unordered_set::count()** - Check if an element is present in the unordered_set.
    - Time Complexity: Average-case O(1), Worst-case O(n) due to hash collisions.
    - Space Complexity: O(1)
6. **unordered_set::erase()** - Remove an element from the unordered_set.
    - Time Complexity: Average-case O(1), Worst-case O(n) due to hash collisions.
    - Space Complexity: O(1)
7. **unordered_set::begin()** and **unordered_set::end()** - Retrieve iterators to the beginning and end of the unordered_set.
    - Time Complexity: O(1)
    - Space Complexity: O(1)
8. **unordered_set::find()** - Search for an element in the unordered_set.
    - Time Complexity: Average-case O(1), Worst-case O(n) due to hash collisions.
    - Space Complexity: O(1)
9. **unordered_set::clear()** - Remove all elements from the unordered_set.
    - Time Complexity: O(n)
    - Space Complexity: O(1)