Set in C++ STL

Sets are a type of associative containers in which each element has to be unique because the value of the element identifies it. The value of the element cannot be modified once it is added to the set, though it is possible to remove and add the modified value of that element.

Sets internally store elements in sorted order, however, you can change the order of elements by passing a comparator function.

Header File: The Set in C++ STL is declared in the header file "set".

Syntax:

```
set set_name;
```

Some basic functions associated with Set:

- **insert()** - This function is used to insert a new element in the Set.
- **begin()** – Returns an iterator to the first element in the set.
- **end()** – Returns an iterator to the theoretical element that follows last element in the set.
- **rbegin()**– Returns a reverse iterator pointing to the last element in the container.
- **rend()**– Returns a reverse iterator pointing to the theoretical element right before the first element in the set container.
- **size()** – Returns the number of elements in the set.
- **empty()** – Returns whether the set is empty.

Note: The Set container in C++ STL does not allows duplicate elements, so upon inserting an element already existing in the Set, it simply ignores the new element.

Below program illustrates the functions discussed above:

```

    // insert elements in random order
    s.insert(40);
    s.insert(30);
    s.insert(60);
    s.insert(20);
    s.insert(50);
    s.insert(50); // only one 50 will be added to the set
    s.insert(10);

    // Traversing set using begin() and end()
    cout << "The set s is :";
    for (auto itr = s.begin(); itr != s.end(); ++itr)
    {
        cout << " " << *itr;
    }
    cout << endl;

    // Traversing set using rbegin() and rend()
    //set <int, greater <int> > :: iterator itr;
    cout << "The set s in reverse order is :";
    for (auto itr = s.rbegin(); itr != s.rend(); ++itr)
    {
        cout << " " << *itr;
    }
    cout << endl;

    return 0;
}

```

Run

Output:

```
The set s is : 60 50 40 30 20 10
The set s in reverse order is : 10 20 30 40 50 60
```

find() and count() functions of Set in C++ STL

- The **set::find()** is a built-in function in C++ STL which returns an iterator to the element which is searched in the set container. It accepts the element to be searched as a parameter and if the element is not found, then it returns an iterator pointing to the position just after the last element in the set otherwise if the element is found, it returns an iterator pointing to that element.

- The **set::count()** function works similar to the find() function. As the name suggests it is supposed to count the number of occurrences of the element present in the set, but as set does not allows duplicate elements, the function count() returns 0 if the element is not present in the set otherwise it returns 1.

Below program illustrate the find() and count() function of Set in C++:

```cpp

// CPP program to illustrate the
// set::find() and set::count() functions

#include <iostream>
#include <set>
#include <algorithm>

using namespace std;

int main()
{
    // Initialize set
    set<int> s;

    s.insert(1);
    s.insert(4);
    s.insert(2);
    s.insert(5);
    s.insert(3);

    // iterator pointing to
    // position where 2 is
    auto pos = s.find(3);

    if(pos != s.end())
        cout<<"Element Found.\n";
    else
        cout<<"Element not Found.\n";

```

Run

Output:

```
Element Found.
The set elements after 3 are: 3 4 5
4 is present
```

set::lower_bound() function in C++ STL

The **set::lower_bound()** is a built-in function in C++ STL which returns an iterator pointing to the element in the container which is equivalent to k passed in the parameter. In case k is not present in the set container, the function returns an iterator pointing to the immediate next element which is just greater than k. If the key passed in the parameter exceeds the maximum value in the container, then the iterator returned points to the last element in the set container.

Below program illustrate the lower_bound() function of Set in C++ STL:

```cpp
    // in the set container
    s.insert(1);
    s.insert(4);
    s.insert(2);
    s.insert(5);
    s.insert(6);

    cout << "The set elements are: ";
    for (auto it = s.begin(); it != s.end(); it++)
        cout << *it << " ";

    // when 2 is present
    auto it = s.lower_bound(2);
    cout << "\nThe lower bound of key 2 is ";
    cout << (*it) << endl;

    // when 3 is not present
    // points to next greater after 3
    it = s.lower_bound(3);
    cout << "The lower bound of key 3 is ";
    cout << (*it) << endl;

    // when 8 exceeds the max element in set
    it = s.lower_bound(8);
    cout << "The lower bound of key 8 is ";
    cout << (*it) << endl;

    return 0;
}

```

Run

Output:

```
The set elements are: 1 2 4 5 6
The lower bound of key 2 is 2
The lower bound of key 3 is 4
The lower bound of key 8 is 5
```

set::upper_bound() function in C++ STL

The **set::upper_bound()** is a built-in function in C++ STL which returns an iterator pointing to the immediate next element which is just greater than k. If the key passed in the parameter exceeds the maximum key in the container, then the iterator returned points to next of last element (which can be identified using set end() function) in the set container.

Below program illustrate the upper_bound() function of Set in C++ STL:

```
{
    set<int> s;

    // Function to insert elements
    // in the set container
    s.insert(1);
    s.insert(4);
    s.insert(2);
    s.insert(5);
    s.insert(6);

    cout << "The set elements are: ";
    for (auto it = s.begin(); it != s.end(); it++)
        cout << *it << " ";

    // when 2 is present
    // points to next element after 2
    auto it = s.upper_bound(2);
    cout << "\nThe upper bound of key 2 is ";
    cout << (*it) << endl;

    // when 3 is not present
    // points to next greater after 3
    it = s.upper_bound(3);
    cout << "The upper bound of key 3 is ";
    cout << (*it) << endl;

    return 0;
}

```

Run

Output:

```
The set elements are: 1 2 4 5 6
The upper bound of key 2 is 4
The upper bound of key 3 is 4
```

Time Complexities and Internal Working

Set in C++ STL stores data in sorted order internally. Also, it uses self-balancing BSTs or Red-Black tree for implementation internally. Thus, it provides some useful operations like search(), insert() and delete() in O(LogN) time complexity.

```
begin() , end()   --------
rbegin() , rend() -------- \  _____    O(1) Time Complexity
cbegin() , cend() ---------/
size() , empty()  --------

insert()         --------
count()          --------  \
find()           --------    ---- >  O(logN) Time Complexity
lower_bound()    --------  /
upper_bound()    --------
```

Sample Problem: Design a Data Structure that supports insert, delete, search, NextGreater and getRank operations

Problem: The aim is to design a data structure that supports the following operations,

- **insert():** Inserts an element into the Data Structure
- **delete():** Delete an element from the Data Structure
- **search():** Search for an element in the Data Structure
- **NextGreater():** Given an element, finding the just next greater element
- **getRank():** When sorted in an ascending manner, the position of the given element.

Examples:

```
Inserting elements into the DS:

insert(1)

insert(37)

insert(40)

insert(45)

insert(2)

insert(18)



DS: {1, 37, 40, 45, 2, 18}



Deleting elements:

delete(1)

delete(37)



DS: {40, 45, 2, 18}



Inserting back elements:

insert(1)

insert(37)



DS: {40, 45, 2, 18, 1, 37}
```

```
Searching elements:

search(2) : TRUE

search(3) : FALSE



Finding next greater element:

NextGreater(18) : 37



Finding the rank:

getRank(18) : 3

For this, the DS is sorted in an ascending manner.
```

Approach: Though this problem can be solved by using various data structures and for some reason use of unordered set would seem more convincing as the operation of insertion, deletion and searching could be done in a constant or O(1) time complexity, but the other operations like NextGreater() and getRank() will consume O(n) time. Therefore, a more efficient solution would be to use **Set** which is an Ordered Data Structure using a red-black tree, resulting in O(logN) time complexity for all the mentioned operations, hence better.

- insert(): Just create a set and start inserting elements using insert operation.
- delete(): Delete the required elements using erase() operation.
- search(): Search for an element using find() function.
- NextGreater(): This operation can be done using the upper_bound() function, which returns an iterator to the next element.
- getRank(): This operation can be also be done using the upper_bound() function and subtracting the position of the first element to get the rank of the given element.

Time Complexity: For every function of the STL we have used the compiler will take O(logN) time complexity which is better than the O(N) time complexity used in the unordered set.

```cpp


// C++ program to illustrate the
// insert, delete, search, finding
// greater element and rank
#include <bits/stdc++.h>
using namespace std;
set<int> s;

// Function to insert elements
void sInsert(int x)
{
    s.insert(x);
}

// Function to delete elements
void sDelete(int x)
{
    cout << "Deleted element:" << x << endl;
    s.erase(x);
}

// Function to search for an element
bool sSearch(int x)
{
    cout << "Searching for " << x << ": ";
    if (s.find(x) != s.end())
        return true;
    return false;
}
```

Run

Output:

```
The set is:
1 2 18 37 40 45
Deleted element:1
Deleted element:37
The set is:
2 18 40 45
The new set is:
1 2 18 37 40 45
Searching for 2: 1
Searching for 3: 0
Element greater to 18: 37
Rank of 18: 3
```

multiset in C++ STL

Multisets are a type of associative containers similar to set, with an exception that multiple elements can have the same values. Like Set, Multiset also uses Red-Black trees internally for implementation and hence support the operations like search, insert and delete in O(logN) time.

Multiset also stores data in an ordered way, By default, it stores data in sorted order.

Since multiset is similar to set except the fact that it can also store duplicate elements, most of the functions available in multiset behave in the same way as that of a set.

However, some functions like erase, count, lower_bound, upper_bound behaves in a slightly different way. Let's see how:

erase() and count() in multiset

- The **multiset::erase()** function in multiset is used to erase all occurrences of a particular element from the multiset. It can accept the value of the element to be erased or an iterator pointing to the element to be erased.

  It can also be used to erase a range of elements by providing the start and end iterator of the range of elements.
- The **multiset::count()** function is a built-in function in C++ STL which searches for a specific element in the multiset container and returns the number of occurrence of that element.

Below program illustrate the above two functions:

```

// CPP program to illustrate the
// erase() and count() function
// in multiset

#include <iostream>
#include <algorithm>
#include <set>

using namespace std;
int main()
{
    // Create a multiset
    multiset<int> s = { 15, 10, 15, 11, 10, 18, 18, 20, 20 };

    // Use count() to count number of occurrences of 15
    cout <<"15 occurs " << s.count(15)
        << " times in container.\n";

    // Erase all occurrences of 15
    s.erase(15);

    // Print again count of 15
    cout <<"15 occurs " << s.count(15)
        << " times in container.\n";

    // Erase a range using erase() function
    s.erase(s.begin(), s.end());

    // Print Size of multiset
```

Run

Output:

```
15 occurs 2 times in container.
15 occurs 0 times in container.
Size of multiset: 0
```

lower_bound() in multiset

The **multiset::lower_bound()** is a built-in function in C++ STL which returns an iterator pointing to the first element in the container which is equivalent to k passed in the parameter. In case k is not present in the set container, the function returns an iterator pointing to the immediate next element which is just greater than k. If the key passed in the parameter exceeds the maximum value in the container, then the iterator returned prints the number of elements in the container.

```cpp

// CPP program to demonstrate the
// multiset::lower_bound() function
#include <bits/stdc++.h>
using namespace std;
int main()
{

    multiset<int> s;

    // Function to insert elements
    // in the multiset container
    s.insert(1);
    s.insert(2);
    s.insert(2);
    s.insert(1);
    s.insert(4);

    cout << "The multiset elements are: ";
    for (auto it = s.begin(); it != s.end(); it++)
        cout << *it << " ";

    // when 2 is present
    auto it = s.lower_bound(2);
    cout << "\nThe lower bound of key 2 is ";
    cout << (*it) << endl;

    // when 3 is not present
    // points to next greater after 3
    it = s.lower_bound(3);
```

Run

Output:

```
The multiset elements are: 1 1 2 2 4
The lower bound of key 2 is 2
The lower bound of key 3 is 4
The lower bound of key 7 is 5
```

upper_bound() in multiset

The **multiset::upper_bound()** is a built-in function in C++ STL which returns an iterator pointing to the immediate next element which is just greater than k. If the key passed in the parameter exceeds the maximum key in the container, then the iterator returned points an element that points to the position after the last element in the container.

```cpp
    // in the multiset container
    s.insert(1);
    s.insert(3);
    s.insert(3);
    s.insert(5);
    s.insert(4);

    cout << "The multiset elements are: ";
    for (auto it = s.begin(); it != s.end(); it++)
        cout << *it << " ";

    // when 3 is present
    auto it = s.upper_bound(3);
    cout << "\nThe upper bound of key 3 is ";
    cout << (*it) << endl;

    // when 2 is not present
    // points to next greater after 2
    it = s.upper_bound(2);
    cout << "The upper bound of key 2 is ";
    cout << (*it) << endl;

    // when 10 exceeds the max element in multiset
    it = s.upper_bound(10);
    cout << "The upper bound of key 10 is ";
    cout << (*it) << endl;

    return 0;
}

```

Run

Output:

```
The multiset elements are: 1 3 3 4 5
The upper bound of key 3 is 4
The upper bound of key 2 is 3
The upper bound of key 10 is 5
```

equal_range() in multiset

The **multiset::equal_range()** is a built-in function in C++ STL which returns an iterator of pairs. The pair refers to the range that includes all the elements in the container which have a key equivalent to k. The lower bound will be the element itself and the upper bound will point to the next element after key k. If there are no elements matching key K, the range returned is of length 0 with both iterators pointing to the first element which is greater than k according to the container's internal comparison object (key_comp). If the key exceeds the maximum element in the set container, it returns an iterator pointing to the past the last element in the multiset container.

```

// CPP program to demonstrate the
// multiset::equal_range() function

#include <bits/stdc++.h>
using namespace std;

int main()
{
    multiset<int> s;

    // Inserts elements
    s.insert(1);
    s.insert(6);
    s.insert(2);
    s.insert(5);
    s.insert(3);
    s.insert(3);
    s.insert(5);
    s.insert(3);

    // prints the multiset elements
    cout << "The multiset elements are: ";
    for (auto it = s.begin(); it != s.end(); it++)
        cout << *it << " ";

    // Function returns lower bound and upper bound
    auto it = s.equal_range(3);
    cout << "\nThe lower bound of 3 is " << *it.first;
    cout << "\nThe upper bound of 3 is " << *it.second;
```

Run

Output:

```
The multiset elements are: 1 2 3 3 3 5 5 6
The lower bound of 3 is 3
The upper bound of 3 is 5
The lower bound of 10 is 8
The upper bound of 10 is 8
The lower bound of 0 is 1
The upper bound of 0 is 1
```