



The [Standard Template Library \(STL\)](#) contains in-built data-structures that come in handy while implementing complex algorithms. One of these containers is a “Set”.

Table of Contents

[hide]

- 1 Properties of a Set in C++
- 2 Initializing a Set in C++
- 3 Traversing a Set in C++
- 4 Adding elements to a Set
- 5 Removing elements from a Set
- 6 Searching an element in a Set
- 7 Other commonly used Set functions
- 8 Conclusion

Difference between a set and a priority queue (heap) :
A priority queue only gives you access to one element in sorted order
-- i.e., you can get the highest priority item, and when you remove that, you can get the next highest priority, and so on. A priority queue also allows duplicate elements, so it's more like a multiset than a set.

Properties of a Set in C++

Let us walk through some basic properties of a C++ Set before moving onto its implementation.

ch **Uniqueness** – All elements inside a C++ Set are **unique**. i.e. `setName.insert(value)` only inserts a unique value and discards a duplicate value.

evr ch Sorted – The elements inside a Set are always in a sorted manner.

Immutable – Any element inside the Set cannot be changed. It can only be inserted or deleted.

Unmodifiable – Any element inside the Set cannot be changed.
Unindexed – The STL Set does not support indexing.

Internal Implementation – The Sets in STL are internally implemented by BSTs (Binary Search Trees).

Initializing a Set in C++

The initialization of a Set involves defining the data type for the elements to be stored and the order in which they are to be stored.

```
1 #include<iostream>
2 #include<set>
3 /*
4 Or
5
6 #include<bits/stdc++.h>
7 */
8 using namespace std;
9
10 int main() {
11
12     // Empty Set - Increasing Order (Default)
13     set<int> s1;
14     // s1 = {}
15
16     // Empty Set - Decreasing Order
17     set<int, greater<int>> s2;
18     // s2 = {}
19
20     // Set with values
21     set<int, greater<int>> s3 = {6, 10, 5, 1};
22     // s3 = {10, 6, 5, 1}
23
24     // Initialize Set using other set
25     set<int, greater<int>> s4(s3);
26     // s4 = {10, 6, 5, 1}
27
28     // Print all elements
29 }
```

```

28     // Initializing a set from array
29     int arr[] = {10, 4, 5, 61};
30     set<int> s5(arr, arr+2); // Only two elements
31     // s5 = {4, 10}
32
33     return 1;
34 }
```

The above code snippet explains the methods of initializing a STL Set.

Traversing a Set in C++

C++ has a concept of iterators for each specific STL data structure. Since a Set does not support indexing, we need to define iterators to fetch its elements.

```

1 #include<iostream>
2 #include<set>
3
4 using namespace std;
5
6 int main(){
7
8     // Set with values
9     set<int, greater<int>> s1 = {6, 10, 5, 1};
10
11    // Iterator for the set
12    set<int> :: iterator it;
13
14    // Print the elements of the set
15    for(it=s1.begin(); it != s1.end(); it++)
16        cout<<*it<<" ";
17    cout<<endl;
18 }
```

Output:

```
10 6 5 1
```

Few things to note here are:

ch 'begin()' – This function returns an iterator to the first element.

evr
ch 'end()' – This function returns an iterator past the last element.
on
evr
ch '*it' – The value stored in the position iterator points to.
on
gth
evr
ri
on
_ri

Adding elements to a Set

The trivial task of adding elements into a set is done by `insert()` function. The function takes in the value of the same nature as the elements in the set.

* Note - `setName.insert(value)` only inserts a unique value and discards a duplicate value.

```
1 #include<iostream>
2 #include<set>
3
4 using namespace std;
5
6 int main() {
7
8     // Set with values
9     set<int, greater<int>> s1 = {6, 10, 5, 1};
10    // s1 = {10, 6, 5, 1}
11
12    // Inserting elements in the set
13    s1.insert(12);
14    s1.insert(20);
15    s1.insert(3);
16
17    // Iterator for the set
18    set<int> :: iterator it;
19
20    // Print the elements of the set
21    for(it=s1.begin(); it != s1.end(); it++)
22        cout<<*it<<" ";
23    cout<<endl;
24
25 }
```

Output:

```
20 12 10 6 5 3 1
```

One thing to keep in mind is that `elements added are placed according to the arrangement` defined at the time of the creation of the set. Therefore, the time complexity of inserting an element into a set is

$O(\log N)$, where **N** is the size of the set.

The size of the set can be retrieved by '`size()`' function.

Removing elements from a Set

The STL set supports the `erase()` function, that can take either the value or the iterator to the value to be removed from the set.

```
1 #include<iostream>
2 #include<set>
3
4 using namespace std;
5
6 int main() {
7
8     // Set with values
9     set<int, greater<int>> s1 = {6, 10, 5, 1};
10    // s1 = {10, 6, 5, 1}
11
12    // Erasing element value = 1
13    s1.erase(1);
14    // s1 = {10, 6, 5}
15
16    // Erasing the first element
17    s1.erase(s1.begin());
18    // s1 = {6, 5}
19
20    // Iterator for the set
21    set<int> :: iterator it;
22
23    // Print the elements of the set
24    for(it=s1.begin(); it != s1.end();it++)
25        cout<<*it<<" ";
26    cout<<endl;
27
28 }
```

Output:

6 5

The C++ code does not prompt an error if the element to be removed is not present in the set. In

case, an invalid iterator is passed, there is a runtime error raised.

Similar to the `insert()` function, `erase()` function takes $O(\log N)$ time to remove an element.

Searching an element in a Set

There is a predefined function `find()` that does the job of searching an element within a set. The function returns an iterator to the value if it is present otherwise returns an iterator past the last element (`end()`).

```
1 #include<iostream>
2 #include<set>
3
4 using namespace std;
5
6 int main() {
7
8     // Set with values
9     set<int, greater<int>> s1 = {6, 10, 5, 1};
10    // s1 = {10, 6, 5, 1}
11
12    // The value to be searched
13    int val = 5;
14
15    // Check if the iterator returned is not the ending of set
16    if(s1.find(val) != s1.end())
17        cout<<"The set contains "<<val<<endl;
18    else
19        cout<<"The set does not contains "<<val<<endl;
20
21    // The value to be searched
22    val = 11;
23
24    // Check if the iterator returned is not the ending of set
25    if(s1.find(val) != s1.end())
26        cout<<"The set contains "<<val<<endl;
27    else
28        cout<<"The set does not contains "<<val<<endl;
29
30 }
```

Output:

```
The set contains 5
The set does not contains 11
```

The code is self-explanatory as it prints the message according to the presence or absence of the value.

Other commonly used Set functions

The following are a few Set functions that may come in handy while working with STL Sets.

`ch size()` – Returns the size of the Set

`evr ch empty()` – Returns true if the Set is empty, otherwise false.

`on
evr
ch rbegin()` – Returns a reverse iterator pointing to the last element of the Set.

`on
gvt
ch rend()` – Returns a reverse iterator pointing before the first element of the Set.

`on
gvt
ch clear()` – Empties the entire Set.

`on
gvt
ch count(value)` – Returns 1 if the ‘value’ is present in the Set, otherwise 0.

`on
gvt
ch swap(set1, set2)` – Swaps the contents of both sets.

`on
gvt
ch emplace(value)` – Inserts the ‘value’ inside the set, if it is not present.

`This` winds up the tutorial on STL Sets in C++.

`on
gvt
ri`

Conclusion

The Standard Template Library has several important containers that must be in every experienced C++ programmer’s arsenal.

We hope this article on STL Sets, was easy to follow. If it was, be sure to go through other [C++](#)

articles on [Journaldev](#). Feel free to comment below for any kind of feedback.