Set ADT

std::set

std::unordered_set

Introduction

- In computer science, a set is an abstract data type that can store unique values, without any particular order.
- It is a computer implementation of the mathematical concept of a finite set.
- Unlike most other collection types, rather than retrieving a specific element from a set, one typically tests a value for membership in a set.
- Some set data structures are designed for static or frozen sets that do not change after they are constructed.
- Static sets allow only query operations on their elements — such as checking whether a given value is in the set, or enumerating the values in some arbitrary order.
- Other variants, called dynamic or mutable sets, allow also the insertion and deletion of elements from the set.

Source: https://en.wikipedia.org/wiki/Set_(abstract_data_type)

Set implementation

- Sets can be implemented using various data structures, which provide different time and space trade-offs for various operations.
- Some implementations are designed to improve the efficiency of very specialized operations, such as nearest or union.
- Implementations described as "general use" typically strive to optimize the **element_of**, **add**, and **delete** operations.
- A simple implementation is to use a **list**, ignoring the order of the elements and taking care to avoid repeated values. This is simple but inefficient, as operations like set membership or element deletion are O(n), as they require scanning the entire list.
- Sets are often instead implemented using more efficient data structures, particularly various kinds of **trees**, or **hash tables**.

C++ std::set data structure in STL

- Container properties
- Sets are containers that store unique elements following a specific order.
- In a set, the value of an element also identifies it (the value is itself the key, of type T), and each value must be unique. The value of the elements in a set cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.

- Internally, the elements in a set are always sorted following a specific strict weak ordering criterion indicated by its internal comparison object (of type Compare).

- **std::set** containers are generally slower than **std::unordered_set** containers to access individual elements by their key, but they allow the direct iteration on subsets based on their order.

- Sets are typically implemented as binary search trees.

C++ std::set data structure

```cpp
int main() {
// empty set container
set<int> s1;
// insert elements in random order
s1.insert(40);
s1.insert(30);
s1.insert(60);
s1.insert(20);
s1.insert(50);
s1.insert(10);
cout<<"print set:"<<endl;
print(s1);

cout<<"\ninsert(50):"<<endl;
s1.insert(50);
print(s1);

cout<<"\nerase(30):"<<endl;
s1.erase(30);
print(s1);

cout<<"\ncopy set: "<<endl;
set<int> s2(s1.begin(), s1.end());
print(s2);
```

```cpp
void print(set<int> s1){
	// printing set s1
	set<int>::iterator itr;

	cout << "\nThe set is : \n";
	for (itr = s1.begin(); itr != s1.end(); itr+
	+) {
	cout << *itr << " ";
	}
	cout << endl;
	}
```

An element previously added to set is not added again

```
print set:
10 20 30 40 50 60

insert(50):
10 20 30 40 50 60

erase(30):
10 20 40 50 60

copy set:
10 20 40 50 60
```

C++ std::set data structure

```
cout << "\nremove all elements up to 40 in s2"<<endl;
s2.erase(s2.begin(), s2.find(40));
print(s2);
cout << "\ns2.insert(90)"<<endl;
s2.insert(90);
print(s2);
cout<<"\nunion of two sets:"<<endl;
set<int> result=s1;
result.insert(s2.begin(),s2.end());
print(result);

cout<<"\
ns1.erase(s2.lower_bound(20),s1.upper_bound(50)):"<<endl;
s1.erase(s1.lower_bound(20),s1.upper_bound(50));
print(s1);
```

```
remove all elements up to 40 in s2
40 50 60

s2.insert(90)
40 50 60 90

union of two sets:
10 20 40 50 60 90

s1.erase(s2.lower_bound(20),s1.upper_bound(50)):
10 60
```

C++ std::unordered_set data structure

```
//empty unordered_set container
unordered_set<int> u1;

// insert elements in random order
u1.insert(40);
u1.insert(30);
u1.insert(60);
u1.insert(20);
u1.insert(50);
u1.insert(10);

cout<<"\nelements of
unordered_set"<<endl;
print(u1);
```

```
elements of unordered_set
10 30 40 60 20 50
```

set vs unordered_set

	set	unordered_set
Ordering	increasing order (by default)	no ordering
Implementation	Self balancing BST like Red-Black Tree	Hash Table
search time	log(n)	O(1) -> Average O(n) -> Worst Case
Insertion time	log(n) + Rebalance	Same as search
Deletion time	log(n) + Rebalance	Same as search

Use set when

- We need ordered data.
- We would have to print/access the data (in sorted order).
- We need predecessor/successor of elements.
- Since set is ordered, we can use functions like binary_search(), lower_bound() and upper_bound() on set elements. These functions cannot be used on unordered_set().

Use unordered_set when

- We need to keep a set of distinct elements and no ordering is required.
- We need single element access i.e. no traversal.

Source: https://www.geeksforgeeks.org/set-vs-unordered_set-c-stl/

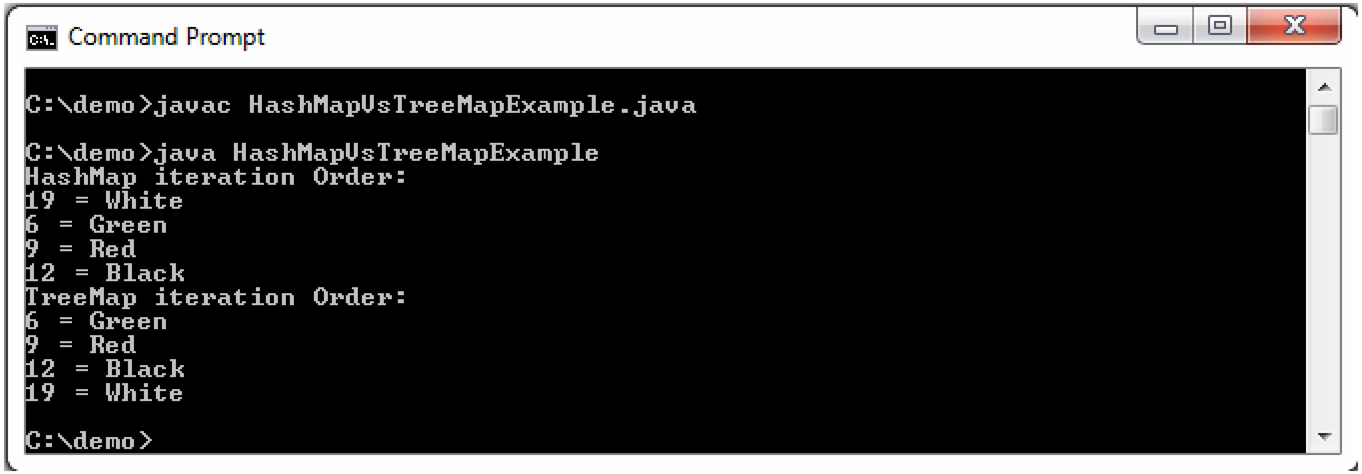
map vs unordered_map

	map	unordered_map
Ordering	increasing order (by default)	no ordering
Implementation	Self balancing BST like Red-Black Tree	Hash Table
search time	log(n)	O(1) -> Average O(n) -> Worst Case
Insertion time	log(n) + Rebalance	Same as search
Deletion time	log(n) + Rebalance	Same as search

https://tutorialspoint.dev/language/cpp/map-vs-unordered_map-c

Java: HashMap vs TreeMap

- The HashMap class uses the hash table.
- TreeMap internally uses a Red-Black tree, which is a self-balancing Binary Search Tree.
- HashMap: it does not maintain any order.
- TreeMap: The elements are sorted in natural order (ascending).



https://www.javatpoint.com/difference-between-hashmap-and-treemap

Java: HashSet vs TreeSet

- For operations like search, insert, and delete HashSet takes constant time for these operations on average. HashSet is faster than TreeSet.
- HashSet is Implemented using a hash table.
- TreeSet takes O(Log n) for search, insert and delete which is higher than HashSet.
- But TreeSet keeps sorted data. Also, it supports operations like higher() (Returns least higher element), floor(), ceiling(), etc.
- These operations are also O(Log n) in TreeSet and not supported in HashSet.
- TreeSet is implemented using a self-balancing binary search tree (Red-Black Tree). TreeSet is backed by TreeMap in Java.

https://www.geeksforgeeks.org/hashset-vs-treeset-in-java/