Lesson:

Set in c++

PW SKILLS

Pre-Requisites

- Basic Knowledge of C++ STL (Standard Template Library)
- Vectors in C++ STL

List of Concepts Involved

- Introduction (Understanding the need)
- What is a set in C++ STL?
- Header files required to implement a Set
- Properties of Set
- Advantages of Set
- Disadvantages of Set
- Creating a Set in C++
- MCQs
- Coding Questions
- Unordered set
- Member Functions of unordered_set
- Multiset
- Member Functions of multiset
- Unordered Multiset
- Member Functions of unordered_multiset

Introduction (Understanding the need)

Let's say you have been given the following balls:

Now you are asked to give unique balls. So, you pick up one ball of each color and return it back. The balls you will return are:

This case happens in real life as well. There will be many times when you will be given a lot of repeated values and you would only require unique values. And so in that case we will use SET.
Mathematically,Set is a collection of unique elements and the same thing also holds here.

What is Set?

Sets are containers in C++ STL that are used for storing unique values in a particular order i.e. either ascending or descending.
Each value in a set acts as a key to itself, i.e. the same is used to uniquely identify itself. And hence once a value is added to the set, that particular value can't be modified because they become constant, once they are added into the set.
Let's say 's' is a set, then s[4] = 9 makes no sense and won't work at all.

However, as a whole, values can be added and deleted from the set.

Header files required to implement a Set

Header files required to implement a Set
We mainly need 2 header files to work with sets. These are:
#include<set> (For declaring set)
#include<iterator> (For declaring the iterator)
OR
Instead of using the above 2 header files, we can use:
#include<bits/stdc++.h>

- This header file not only includes these 2 header files but all the other header files that you would ever need in your program. So, it's better to use this one.
- The drawback of using this header file is it increases the average compile time of the program and it's also very readable, but in CP (competitive programming) we can use it.

Properties of Set

- **Storing order –** The set stores the elements in sorted order.
  Although it has the default sorted order and also mention that we can always pass a custom comparator to arrange elements as per our requirements.
- **Values Characteristics –** All the elements in a set have unique values.
- **Values Nature –** The value of the element cannot be modified once it is added to the set, though it is possible to remove and then add the modified value of that element. Thus, the values are immutable. However, set is mutable, i.e. once created, it can be modified as new values can be added or deleted.
- **Search Technique –** Sets follow the Binary search tree implementation.
- **Arranging order –** The values in a set are unindexed.

Advantages of Set

- Set stores unique values, thus there is no scope for having duplicate values.
- Elements are stored in either ascending or descending order, which makes it efficient.
- Sets are dynamic in size, so we don't have to worry about overflowing errors.
- Sets are used to implement many algorithms because it is fast.
- Search in sets happen O(logN) time complexity.
- It is really very fast and efficient to check if the element is present in the set or not as compared to arrays.

Disadvantages of Set

- We can access the elements of arrays by their indexes, but it is not possible in sets. We can access the elements through iterator only.
- Set is not suitable for large data set because it takes O(logN) for basic operations like insertion and deletion, which makes it quite inefficient when we have to add and remove elements quite frequently.
- It is quite difficult to implement a set because of its structure and properties.
- Set uses more memory than arrays or lists because it stores each element in a separate location.

Creating a Set in C++

In Ascending order (by default)
By default, the std::set is sorted in ascending order.

```
set<int>s;
```

In Descending Order

However, we have the option to change the sorting order by using the following syntax.

```
set<int,greater<int>>s;
```

Datatype: A set can take any data type depending on the values, e.g. int, char, float, etc.

Iterating on Set

We can iterate through a set by creating an iterator and pointing it at the start of the set by using the begin() function. After each iteration in the for loop, the iterator will point to the next element in a set and this will continue until we reach the end of the set.

An iterator can be created by:

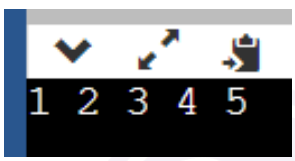
```
set<int>::iterator itr;
```

The elements of the set s can be displayed by:

```
set<int>s = {3, 4, 1, 2, 5};
set<int>::iterator itr;

for(itr = s.begin(); itr != s.end(); itr++){
    cout<<*itr<<" ";
}
```

Output:



But say you have to print the values of the set in reverse order then we need to create a reverse iterator and we will make it point at the last element of the set by using rbegin() function. After each iteration, it will be moving toward the start of the set.

A reverse iterator can be created by:

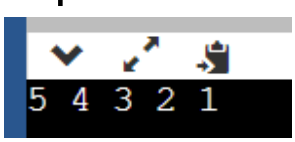
```
set<int>::reverse_iterator ritr;
```

The elements of the set s can be displayed by (in reverse order):

```
set<int>s = {3, 4, 1, 2, 5};
  set<int>::reverse_iterator ritr;

  for(ritr = s.rbegin(); ritr != s.rend(); ritr++){
      cout<<*ritr<<" ";
  }
```

Output:



Insertion and Deletion in Set

The time complexities for doing these operations on sets are:

- Insertion of Elements – O(log N)

- Deletion of Elements – O(log N)

Where N is the number of elements in the set.

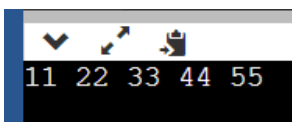
Inserting an element into a Set

For inserting an element insert() is an in-built function, which can insert an element in the set.
Syntax: `set_name.insert(element)`
Return Value: The function returns an iterator pointing to the element that has been inserted.
Code:

```
set<int>s;
 s.insert(55);
 s.insert(44);
 s.insert(11);
 s.insert(33);
 s.insert(22);

 for(auto i : s)
     cout<<i<<" ";
```

Output:



Deleting an element from a Set

To delete one or more elements from the set, erase() function is used.
Syntax:
set_name.erase(element) - Removes the element from the set
set_name.erase(position) - Removes the element at the given position
set_name.erase(start_position, end_position) - Removes elements in specified range (In this case, time complexity becomes O(n), where n is the number of elements in the range)

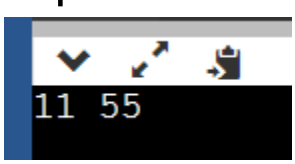
Code:

```
set<int>s = {55, 44, 33, 22, 11};

s.erase(44);        // removing the specified element
s.erase(s.begin());// removing the element at specified position

for(auto i : s)
    cout<<i<<" ";
```

Output:



Explanation:
Here, s = {11, 22, 33, 44, 55}
start points at 11 initially and end points at imaginary element after the last element.
start++ makes it now point at 22 and end-- makes it now point at 55.

So, the deleted elements will be [22, 55) thus deleting 22, 33, 44.

So, final set s = {11, 55}

Functions Associated with Set

Function	Description
size()	Returns the number of elements in the set
max_size()	Returns the maximum number of elements that the set can hold. (a really large number like: 461168601842738790)
empty()	Returns true if set is empty else false
insert()	Inserts one or more elements in a set like `s.insert(10);` `s.insert({20,10,30});`
erase()	Removes one or more elements from the set. Position, element, or a range of elements can be passed
clear()	Removes all the elements from the set
find()	It finds the element that is passed as the parameter. If the element is found, it returns its position else returns the iterator to the end.
count()	Count the number of occurrences of the element passed as a parameter. Since set contains only unique elements so it returns 1 if present else 0

lower_bound()	Returns the value passed if present otherwise returns just greater value.
	`set<int>s = {55, 44, 33, 22, 11};`
	`auto it = s.lower_bound(23);`
	`cout<<*it;`
	Output: 33
	Note:Sometime people use lower_bound(s.begin(), s.end(), value) instead of using s.lower_bound . It's a common mistake as the time complexity of the former one can go upto O(N) as we don't have random access on set and it iterates though it to get the middle element in binary search.

upper_bound()	Returns the next greater value after the value passed as a parameter
	`set<int>s = {55, 44, 33, 22, 11};`
	`auto it = s.upper_bound(33);`
	`cout<<*it;`
	Output: 44

begin()	Returns the iterator pointing to the first element in the set

end()	Returns the iterator pointing at a theoretical element i.e. after the last element of the set
rbegin()	Returns a reverse iterator pointing at the last element of the set
rend()	Returns a reverse iterator pointing at a theoretical element i.e. before the first element of the set

MCQs

1. What will be the output of the following code:

```
int main(){
    set<int> myset = {1,1,1,1,1,1,2,2,2,2,3,3,3,4};

    for(auto it: myset)
        cout<<it<<" ";
    return 0;
}
```

a) Compilation Error
b) Runtime Error
c) 4
d) 1 2 3 4

Answer: d

Explanation: Set only contains unique elements. Here, when we are initializing the set with a lot of duplicate values, these values get discarded. Thus, when we are adding an element to a set and that element is already present in the set, then the set will not change and no matter how many duplicate values you add, set will only contain unique ones. Thus, making 1 2 3 4 as the correct answer in our case.

2. Consider the set s = {1,2,3,4,5}. Let's say we need to check if 2 is present in the set 's' or not. Which of the following code statements will do the needful?

A) if(s.find(2) != s.end())
B) if(s.count(2))
C) if(s.find(2))
D) if(s.count(2) != s.end())

Choose the correct option:

a) A,B
b) A,B,C,D
c) A,C,D
d) A,D
e) A,B,D
f) A,B,C

Answer: a

Explanation: The set::count() is a built-in function in C++ STL which returns the number of times an element occurs in the set. And since, there are only unique elements in the set, it can either return 1 or 0.

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. If the element is not found, then the iterator points to the position just after the last element in the set which is where the iterator returned by set.end() also points.The time complexity of the count function in the std::set container in C++ is logarithmic, O(log n), where n is the size of the set.

A) `if(s.find(2) =/= s.end())`

Since, find() returns an iterator and so does end(), so we need to make a comparison here, that whether both are equal or not. If they are not equal then element is present else not because then iterator returned by find() will be pointing at theoretical element after the last element in the set.

B) `if(s.count(2))`

Here, `s.count(2)` will be returning either 1 or 0 which can easily be converted into true or false respectively. Thus, it will work perfectly fine.

C) `if(s.find(2))`

Here, `s.find(2)` will be returning an iterator and an iterator inside `if` statement won't return a bool value thus an error will be thrown.

D) `if(s.count(2) =/= s.end())`

`s.count(2)`returns two values either 0 or 1, whereas `s.end()` return an iterator pointing at the theoretical element after the last element in the set. So, there is no point in doing a comparison between the two. Hence, in this case, an error will be thrown.

Coding Questions

Problem: 1

Cherry''s birthday is coming this month! She wants to plan a Birthday party and is preparing an invite list with her friend Aashi. she asks Aashi to tell her names to add to the list.

Aashii is a random guy and keeps coming up with names of people randomly to add to the invite list, even if the name is already on the list! Cherry hates redundancy and hence, enlists the names only once.

Find the final invite-list, that contain names without any repetition.

Input:

First line of each test contains an integer N, the number of names that Aashi pops up with.

Output:

Output the final invite-list with each name in a new line. The names in the final invite-list are sorted lexicographically.

Solution:

Code link: https://pastebin.com/UTPDXyFi

Explanation:

- We read in the number of names n.
- We create an ordered set of strings called names, which automatically sorts the names lexicographically and eliminates duplicates.

- We read in each name from the input and insert it into the set using the insert function.
- Finally, we iterate through the set and print out each name in the set, which represents the final invite list without any repetition.
- **Note:** The use of an ordered set ensures that the names are sorted lexicographically and no duplicates are present.

2. Add the common elements

Given 2 vectors v1 and v2. Find out the common elements between the two and return the sum of them.

Input:

V1 = {1, 1, 2, 3, 3, 3}

V2 = {5, 6, 7, 5, 2, 3, 6}

Output:

5

Explanation:

The values common between V1 and V2 are: 2, 3. So, the sum is 2+3 = 5.

Solution: https://pastebin.com/ZJrVctbj

Solution Explanation: Here, we have two vectors, and we are required to find the common elements between them and return their sum. This can be done easily by a set. First, we will iterate through the first vector and insert all its elements into it. Since the set only stores unique elements, no duplicates will be there in the set. Now, we iterate through the next vector and instead of inserting elements into the set, we now check if that element is already in the set or not. If it is present then we add that element to our ans variable, otherwise, we iterate forward.

3. Check if string has all english alphabets

Given a string. You have to check if it has all english alphabets from a-z irrespective of their case i.e. they can be any, uppercase or lowercase.

Input-1:

abcdEfGHIJKLmnopqrstuvWXYZ

Output-1:

Yes

Input-2:

PhysicsWallah

Output-2:

No

Explanation:

Input-1 has all the alphabets irrespective of upper or lower case, so the output is Yes. But in the case of Input-2, it doesn't contain all the alphabets, hence No.

Solution: https://pastebin.com/A0vh2uSs

Solution Explanation: Initially, if the size of the string is less than 26, then it can never have all the alphabets. So, we will keep a check of that. Then, we will transform the entire string into lowercase or uppercase because of the condition of the question. Next, we will insert all the characters of the string into a set of char data type. Since set contains only unique elements, so if the size of the set is less than 26, then it doesn't contain all English alphabets, otherwise, it does and so we print "Yes".

Unordered Set in C++

unordered_set is a container in C++ STL that stores unique elements in no order, and which allow for fast retrieval of individual elements.

In an unordered_set, the value of an element is at the same time its key, which identifies it uniquely.

In unordered_set elements can be added and removed. However, the values can't be changed, because values at the same time are keys and keys are immutable.

Internally, the elements in the unordered_set are not sorted in any particular order, but organized into buckets depending on their hash values.

It allows fast access to individual elements directly by their values (with a constant average time complexity on average).

unordered_set containers are faster than set containers to access individual elements by their key.

They are generally less efficient than set for range iteration through a subset of elements.

Header File Required

To create an unordered_set in C++ we require the `unordered_set` header file. We can include it in the following way:

```
#include<unordered_set>
```

Also, we can use `bits/stdc++.h` as it includes every standard library. But it in itself is a non-standard header file of the GNU C++ library. It doesn't include only `unordered_set` but also other libraries thus it increases the compilation time. We can include that in the following way:

```
#include<bits/stdc++.h>
```

Declaring an unordered_set in C++

An unordered_set in C++ can be created by the following syntax:

```
unordered_set<data_t ype>name;
```

For example:

```
unordered_set<int>s1;
unordered_set<string>s2;
unordered_set<char>s3;
```

Initializing a set

You can also create and initialize a set in the following way:

```
unordered_set<datatype>name = {val1, val2, val3, val4, val5};
```

For example:

```
unordered_set<int>s4 = {{33, 1, 2, 33, 22, 55, 5, 3, 5};
unordered_set<string>s5 = {"Physics", "College", "Wallah"};
unordered_set<char>s6 = {'C','O','L','L','E','G','E'};
```

Inserting into an unordered_set

Let us create a set and then insert elements into it.

```
  unordered_set<string>s;
    s.insert("Apple");
    s.insert("Ball");
    s.insert("College Wallah");
```

Printing an unordered_set

We can print the pairs of the unordered_set using a for each loop.

Say we have to print the unordered_set we created above.

```
for(auto i : s)
  cout<<i<<endl;
```

The **output** will be:

```
College Wallah
Ball
Apple
```

Notice, how no order is there, it is randos.

Member Functions

Say there is an unordered_set<int>s;

We can perform the following functions on it:

Functions	Explanation	Example	Time Complexity

insert()	Inserts element
	Each element is inserted only if it is not already present in the set.
		`s.insert(1);`
		`s.insert({1,2,3});`
		`s.insert(a.begin(),a.end());`
		`s.insert(2+3);`
			If a single element is inserted:
			Average - O(1)
			Worst - O(N)
			
			Multiple elements inserted:
			Average: O(n)
			Worst - O(n*(N+1))
			
			Where n elements are inserted and N stands for the number of elements in the unordered_set (size).
			Explanation:The reason for this is that when the unordered_set is resized due to the number of elements exceeding its current capacity, all the elements need to be rehashed and reinserted into the new larger unordered_set. This results in a worst-case time complexity of O(n*(N+1)), where n is the number of elements being inserted, and N is the size of the unordered_set.
			On average, the time complexity of inserting a single element into an unordered_set is O(1), assuming a good hash function. However, when inserting multiple elements, the average time complexity is O(n), since each element needs to be hashed and inserted into the unordered_set.

erase()	Erases elements by • Iterator • Key • Range	• `s.erase(s.begin());` • `s.erase(2);` • `s.erase(s.begin(),s.end());`	Average: Linear in the number of elements removed i.e. O(n), n is no. of elements removed Worst: O(N) where N stands for the size of unordered_set
swap()	Swaps two unordered_set with the same type, size may differ.	`s1.swap(s2);` `swap(s1, s2);`	Constant
clear()	Removes all the elements	`s.clear();`	O(N), linear is size of unordered_set

Capacity

Functions	Explanation	Example	Time Complexity
empty()	Returns 1 if the unordered_set is empty else 0	`s.empty();`	Constant, O(1)
size()	Returns the size	`s.size();`	Constant, O(1)
max_size()	Returns the maximum size (a really large number)	`s.max_size();`	Constant, O(1)

Operations

Functions	Explanation	Example	Time Complexity
find()	Returns the iterator to the element	`s.find();`	Average: O(1), constant Worst: O(N), N is size of container
count()	Returns the no. of occurrences of the key passed and since the unordered_set has only unique keys so it returns 1 if present else 0	`s.count();`	Average: O(1), constant Worst: O(N), N is the size of container
equal_range()	Returns the bounds of a range that includes all the elements in the container that compare equal to k.	`s.equal_range(k)`	Average: Constant Worst: O(N)

Buckets

Functions	Explanation	Example	Time Complexity
bucket_count()	Returns the number of buckets in the unordered_set container.	`s.bucket_count();`	Constant
max_bucket_count()	Returns the maximum number of buckets that the unordered_set container can have.	`s.max_bucket_count();`	Constant
bucket_size()	Returns the number of elements in bucket x	`s.bucket_size(x);`	O(n) where n is the bucket size
bucket()	Returns the bucket number where the element with value k is located.	`s.bucket(k);`	Constant

Hash policy

Functions	Explanation	Example	Time Complexity
load_factor()	Returns the current load factor in the unordered_set container. load_factor = size of unordered_set / bucket_count	`s.load_factor();`	Constant
max_load_factor()	By default, unordered_set containers have a max_load_factor of 1.0.	`s.max_load_factor();`	Constant
rehash()	Sets the number of buckets in the container to x or more.	`s.rehash(x);`	Average: O(N) Worst: O(N2)
reserve()	Sets the number of buckets in the container (bucket_count) to the most appropriate to contain at least x elements.	`s.reserve(x);`	Average: O(N) Worst: O(N2)

Iterators

Functions	Explanation	Example	Time Complexity
begin()	Returns an iterator to the first element in the unordered_set	`s.begin();`	Constant
end()	Returns an iterator to the theoretical element after the last element	`s.end();`	Constant

multiset in C++ STL

multiset is similar to set except the fact that we can store repetitive elements in it. Also, all the elements are stored in either ascending or descending order.

Header File Required

To create a multiset in C++ we require the `set` header file. We can include it in the following way:

```
#include<set>
```

Also, we can use `bits/stdc++.h` as it includes every standard library. But it in itself is a non-standard header file of the GNU C++ library. It doesn't include only `set` but also other libraries thus it increases the compilation time. We can include that in the following way:

```
#include<bits/stdc++.h>
```

Declaring a multiset in C++

A set in C++ can be created by the following syntax:

In ascending order (by default):

```
multiset<data_type>set_name;
```

In descending order:

```
multiset<data_type, greater<data_type>>set_name;
```

For example:

```
multiset<int>s1;
multiset<string>s2;
multiset<char>s3;
```

Initializing a multiset

You can also create and initialize a set in the following way:

```
multiset<data_type>set_name = {val1, val2, val3, val4, val5};
```

For example:

```
multiset<int>s4 = {1, 11, 2, 2, 3, 33, 3, 33, 2, 55};
multiset<string>s5 = {"Physics", "College", "Wallah"};
multiset<char>s6 = {'C','O','L','L','E','G','E'};
```

Inserting into a multiset

Let us create a multiset and then insert elements into it.

```
multiset<string>s;
s.insert("Apple");
s.insert("Ball");
s.insert("College Wallah");
s.insert("College");
```

```
    s.insert("Cat");
    s.insert("College");
```

Printing a multiset

We can print the pairs of the multiset using a for each loop.

Say we have to print the multiset we created above.

```
for(auto i : s)
        cout<<i<<endl;
```

The output will be:

```
Apple
Ball
Cat
College
College
College Wallah
```

Notice how repetitive keys are stored in the multiset in ascending order (by default).

Member Functions

Say we have a multiset<int>s;

Now we can perform the following functions on it:

Functions	Explanation	Example	Time Complexity
insert()	Inserts element	• s.insert(1); • s.insert(position, val); • s.insert(first, last);	• Logarithmic • Logarithmic • O(Nlog(size+N)) where N is the distance between first and last and size is the size of multiset before insertion
erase()	Erases elements by • Iterator • Value • Range	• s.erase(s.find(1)); • Note:s.erase(s.find(1)) will only erase one occurrence of 1. • s.erase(2); • Note:it will delete all the occurrences of 2.it can cause random behavior and severe bugs in the code. • s.erase(s.begin(),s.end());	• Amortized Constant • O(logN) (N: the size of multiset) • Linear in the distance between the range
swap()	Swaps 2 sets with the same type, size may differ.	m1.swap(m2); swap(m1, m2);	Constant
clear()	Removes all the elements	s.clear();	O(N), linear is size of multiset

Capacity

Functions	Explanation	Example	Time Complexity
empty()	Returns 1 if the multiset is empty else 0	`s.empty();`	Constant, O(1)
size()	Returns the size	`s.size();`	Constant, O(1)
max_size()	Returns the maximum size (a really large number)	`s.max_size();`	Constant, O(1)

Operations

Functions	Explanation	Example	Time Complexity
find()	Returns the iterator to the element	`s.find(val);`	O(logN)
count()	Returns the no. of occurrences of the key passed	`s.count(val);`	O(logN) + Linear in no. of matches
upper_bound()	Returns the iterator to the next greater element	`s.upper_bound(4)`	O(logN)
lower_bound()	Returns the iterator to the first occurrence of that element (if present) otherwise the next greater element	`s.lower_bound(4)`	O(logN), where N stands for the size of set
equal_range()	Returns the bounds of a range that includes all the elements in the container which have a value equivalent to val.	`s.equal_range(val)`	O(logN)

Iterators

Functions	Explanation	Example	Time Complexity
begin()	Returns an iterator to the first element in the multiset	`s.begin();`	Constant
end()	Returns an iterator to the theoretical element after the last element	`s.end();`	Constant
rbegin()	Returns a reverse iterator pointing to the last element of the multiset	`s.rbegin();`	Constant
rend()	Returns a reverse iterator pointing to the theoretical element just before the first element in the multiset	`s.rend();`	Constant

Unordered multiset in C++ STL

Header File Required

To create an unordered_multiset in C++ we require the `unordered_set` header file. We can include it in the following way:

```
#include<unordered_set>
```

Also, we can use `bits/stdc++.h` as it includes every standard library. But it in itself is a non-standard header file of the GNU C++ library. It doesn't include only `unordered_multiset` but also other libraries thus it increases the compilation time. We can include that in the following way:

```
#include<bits/stdc++.h>
```

Declaring an unordered_multiset in C++

A set in C++ can be created by the following syntax:

```
unordered_multiset<data_type>name;
```

For example:

```
unordered_multiset<int>s1;
unordered_multiset<string>s2;
unordered_multiset<char>s3;
```

Initializing an unordered_multiset

You can also create and initialize a set in the following way:

```
unordered_multiset<datatype>name={val1, val2, val3, val4, val5};
```

For example:

```
unordered_multiset<int>s4 = {1, 11, 2, 2, 3, 33, 3, 33, 2, 55};
unordered_multiset<string>s5 = {"Physics", "College", "Wallah",
                                "Wallah", "Wallah", "Wallah"};
unordered_multiset<char>s6 = {'C','O','L','L','E','G','E'};
```

Inserting into an unordered_multiset

Let us create an unordered_multiset and then insert elements into it.

```
  unordered_multiset<string>s;
    s.insert("Apple");
    s.insert("Ball");
    s.insert("College Wallah");
    s.insert("College");
    s.insert("Cat");
    s.insert("College");
```

Printing a set

We can print the pair of the unordered_multiset using a for each loop.

Say we have to print the unordered_multiset we created above.

```
for(auto i : s)
        cout<<i<<endl;
```

The **output** will be:

```
Cat
College
College
College Wallah
Ball
Apple
```

Notice repetitive keys for values. Also, no specific order is maintained.

Member Functions

Say there is an unordered_set<int>s;

We can perform the following functions on it:

Functions	Explanation	Example	Time Complexity

insert()	Inserts element	`s.insert(1);`
		`s.insert({1,2,3});`
		`s.insert(a.begin(),a.end());`
		`s.insert(2+3);`
			If a single element is inserted:
			Average - O(1)
			Worst - O(N)
			
			Multiple elements inserted:
			Average: O(n)
			Worst - O(n*(N+1))
			
			Where n elements are inserted and N stands for the number of elements in the unordered_multiset (size)

erase()	Erases elements by • Iterator • Value • Range	• `s.erase(s.find(1));` • `s.erase(2);` • `s.erase(s.begin(),s.end());`	Average: Linear in the number of elements removed i.e. O(n), n is no. of elements removed Worst: O(N) where N stands for the size of unordered_multiset
swap()	Swaps 2 sets with the same type, size may differ.	`m1.swap(m2);` `swap(m1, m2);`	Constant
clear()	Removes all the elements	`s.clear();`	O(N), linear is size of unordered_multiset

Capacity

Functions	Explanation	Example	Time Complexity
empty()	Returns 1 if the unordered_multiset is empty else 0	`s.empty();`	Constant, O(1)
size()	Returns the size	`s.size();`	Constant, O(1)
max_size()	Returns the maximum size (a really large number)	`s.max_size();`	Constant, O(1)

Operations

Functions	Explanation	Example	Time Complexity
find()	Returns the iterator to the element	`s.find(x);`	Average: O(1), constant Worst: O(N), N is the size of container
count()	Returns the no. of occurrences of the key passed	`s.count(x);`	Average: O(n), where n is no. of elements counted Worst: O(N), N is the size of container
equal_range()	Returns the bounds of a range that includes all the elements in the container which have a key equivalent to k.	`s.equal_range(k)`	Average: Constant Worst: O(N)

Buckets

Functions	Explanation	Example	Time Complexity
bucket_count()	Returns the number of buckets in the unordered_multiset container.	`s.bucket_count();`	Constant

max_bucket_count()	Returns the maximum number of buckets that the unordered_multiset container can have.	`s.max_bucket_count();`	Constant
bucket_size()	Returns the number of elements in bucket x	`s.bucket_size(x);`	O(n) where n is the bucket size
bucket()	Returns the bucket number where the element with value k is located.	`s.bucket(k);`	Constant

Hash policy

Functions	Explanation	Example	Time Complexity
load_factor()	Returns the current load factor in the unordered_multiset container. load_factor = size of unordered_multiset / bucket_count	`s.load_factor();`	Constant
max_load_factor()	By default, unordered_multiset containers have a max_load_factor of 1.0.	`s.max_load_factor();`	Constant
rehash()	Sets the number of buckets in the container to x or more.	`s.rehash(x);`	Average: O(N) Worst: O(N2)
reserve()	Sets the number of buckets in the container (bucket_count) to the most appropriate to contain at least x elements.	`s.reserve(x);`	Average: O(N) Worst: O(N2)

Iterators

Functions	Explanation	Example	Time Complexity
begin()	Returns an iterator to the first element in the set	`s.begin();`	Constant
end()	Returns an iterator to the theoretical element after the last element	`s.end();`	Constant

MCQs

1. What will be the output of the following code?

```
multiset<string>s;
s.insert("Apple");
s.insert("Ball");
s.insert("College Wallah");
s.insert("College");
s.insert("Cat");
s.insert("College");

s.erase(s.find("College"), s.end());

for(auto i : s)
    cout<<i<<endl;
```

a) Apple
Ball
College Wallah

b) Apple
Ball
c) Apple
Ball
Cat
d) Apple
Ball
Cat
College Wallah

Answer: c
Explanation: In a multiset repetitive values are stored in ascending order by default.
So, multiset s will be:
Apple
Ball
Cat
College
College
College Wallah

According to this line, `s.erase(s.find("College"),s.end());` from the first occurrence of "College" till end of multiset s everything will be erased. So, s will be:
Apple
Ball
Cat

2. What will be the output of the following code?

```
set<int>s;
 int n = 0;
 for(int i = 0; i <= 3; i++){
     for(int j = 0; j <= 3; j++){
         if(n % 2 == 0)
             s.insert(i);
         else
             s.insert(j);
         n++;
     }
 }

 for(auto i : s)
     cout<<i<<" ";
```

a) 0 0 1 1 2 2 3 3
b) 0 1 2 3
c) 0 1 1 2 3 3
d) Compilation Error

Answer: b

Explanation: Irrespective of the value of n, the value of i and j are getting inserted into s. So, s will contain the values: 0, 1, 2, 3. Now since, a set store only unique values in ascending order by default, so there is no point of repetitive values. Hence, b is the correct answer.

Coding Questions

1. Given the number of questions as n, and marks for the correct answer as p and q marks for the incorrect answer. One can either attempt to solve the question in an examination and get either p marks if the answer is right, or q marks if the answer is wrong, or leave the question unattended and get 0 marks. The task is to find the count of all the different possible marks that one can score in the examination. [Medium]

Input - 1:

n = 4, p = 2, q = -1

Output - 1:

12

Input - 2:

n = 2, p = 1, q = -1

Output - 2:

5

Explanation:

The different possible marks are: -2, -1, 0, 1, 2

Solution Code: https://pastebin.com/1KcqJy7D

Code Explanation:

- Iterate through all the possible number of correctly solved and unsolved problems or all possible pairs of (p, q).
- Store the scores in a set containing distinct elements keeping in mind that there is a positive number of incorrectly solved problems.
- Print the size of set in the end.

Time complexity: O(N2), since we are using nested loops.

Space complexity: O(N) linear space complexity, since we are using a set here.

Output:

```
n = 4
p = 2
q = -1

Different possible marks that one can score = 12
```

```
n = 2
p = 1
q = -1

Different possible marks that one can score = 5
```

2. Given n integers (can be duplicates), print the second smallest integer. If it does not exist, print -1. (Medium)

Input-1:

n = 4

1 2 2 -4

Output-1:

1

Input-2:

n = 5

1 2 3 1 1

Output-2:

2

Solution Code: https://pastebin.com/vVDu4VUR

Code Explanation: A set stores unique elements in ascending order by default. So, if we insert all the elements of the vector into the set, then at the second position in the set, we will be having our second smallest number. Thus, we erase the first element and after that print the first element which is the second smallest number.

Time complexity: O(N), linear time complexity as we are traversing all the elements of the vector at least once in order to insert them into the set.

Space complexity: O(N), linear space, because we are using a set to store values.

Code Output:

```
Enter the number of elements = 4
Enter the elements = 1 2 2 -4

Second smallest number = 1
```

```
Enter the number of elements = 5
Enter the elements = 1 2 3 1 1

Second smallest number = 2
```

Upcoming Class Teasers:

- Backtracking.