STLs

Agenda

1. Introduction
    a. What is an Algorithm?
    b. What is a Data Structure (Container)?
    c. Components of Data Structure
    d. Components of C++ STL?
2. Sequential containers
    a. Array
    b. Vector
    c. Deque
3. Container Adapters
    a. Stack
    b. Queue
    c. Priority Queue
4. Comparison between some Containers
5. Associative containers
    a. Prerequisite: Pair
    b. Set
    c. Multiset
    d. Map
    e. Multimap
6. Important methods
7. Unordered containers

8. Reasons to use C++ STL

1. Introduction

STLs (Standard Template Libraries) are a set of tools that assist with implementing a wide range of data structures and algorithms in C++ programming. Before we dive into the details of STL and its components, it's important to understand the most significant benefits of using STL which is the availability of data structures and algorithms.

a. What is an Algorithm?

- An algorithm is a step-by-step procedure that describes how to solve a problem or perform a task. Think of it as a recipe for solving a problem. Just as a recipe tells you how to prepare a meal, an algorithm tells you how to solve a problem.
- To create an algorithm, you need to first identify the problem you want to solve. You then need to break the problem down into smaller, more manageable steps. Each step should be clear and unambiguous so that anyone following the algorithm can understand what they need to do. Once you have identified the steps, you can start to implement the algorithm.
- It's important to test the algorithm to make sure it works as expected and to refine it as necessary. One of the most important things to keep in mind when creating an algorithm is to make it as efficient as possible. This means finding the fastest and most effective way to solve the problem.

b. What is a data structure (Container)?

- When we solve a problem or develop a program, we usually need to collect data from the user as input. To manipulate this data effectively, we first need to store it in memory. However, if the user provides millions of integers, we cannot create millions of variables to store the input. This is where data structures come into play. They define a set of rules for organizing and storing the data, ensuring that it is readily available for processing at a later stage.
- Without a well-defined data structure, accessing or modifying the input data effectively would be difficult. Additionally, data structures enable us to optimize memory usage by allocating only the required amount of memory to store the data. Ultimately, using data structures allows us to process input data more efficiently, leading to faster and more effective program execution.

c. Components of Data Structures

Data structures typically consist of four components:

1. **Data values:** The representation of the actual information being stored.
2. **Relations:** The links between the data values.
3. **Operations:** The allowable manipulations on the data.
4. **Constraints:** The valid limitation (Size of the data that could be stored) of the data structure.

- One example of data structures is Arrays, they are an important data structure in programming and are widely used and provided in many programming languages. However, there are many other data structures that support a variety of algorithms and applications.
- While these data structures are significant, their implementation can be challenging. To simplify this process, the developers of C++ have included common data structures as part of the C++ Standard Template Libraries (STL). With the STL, we can access a set of Black Box (pre-defined) data structures and algorithms, making it easier for us to solve complex problems efficiently.
- Overall, Data structures serve as tools to store and manipulate the data (Containers) that are utilized in specific algorithms to address problem requirements. They are not the solution to the problem in its entirety, but rather a part of it.
- The choice of data structure will rely heavily on the problem being solved and the operations that must be carried out on the data. Choosing the appropriate data structure would help us to optimize the performance, resulting in a more efficient and effective solution.
- However, no data structure is perfect, each comes with its own set of pros and cons. Selecting the wrong one can cause our program to run too slowly or use too much memory, but fortunately, we will learn together how to pick the ideal data structure that satisfies the problem requirements.

d. Components of C++ STL

There are three main components of the C++ STL:

1. **Containers**: The data structures.
2. **Algorithms:** Common algorithms to be used with the data structures (Sort function, find function, etc.).
3. **Iterators:** Provide a means for accessing and traversing data stored in the data structure.

- We can say that STL is a set of C++ template classes to provide common programming data structures and functions.
- We now have a solid foundation that enables us to dive deeper into the topic. Our next step is to examine specific containers and their corresponding iterators to gain a more comprehensive understanding.
- C++ provides a variety of containers that can be used to store and manipulate data. These containers are implemented as classes and templates that provide a variety of operations for manipulating the elements they contain. Here are the main containers available in C++:
  - ❖ **Sequential Containers.**
  - ❖ **Container Adapters.**
  - ❖ **Associative Containers.**
  - ❖ **Unordered Containers.**

2. Sequential containers

Sequence containers are a type of container that stores elements in a linear sequence. This means that elements are stored in the order they were inserted, and they can be accessed by their position in the container.

Sequence containers are divided into:

- **Array**
- **Vector**
- **Deque**
- ~~**List**~~

a. Array

```
int arr[3];
arr[0] = 1;
arr[1] = 2;
arr[2] = 3;
```

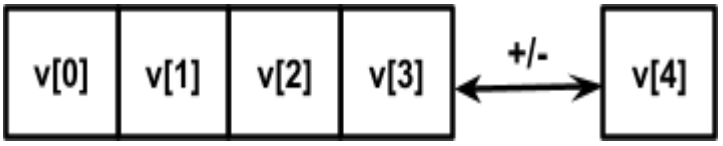
- Is an array considered a data structure? Yes, it is considered a data structure because an array is a collection of elements of the same data type stored in contiguous memory locations. It provides random access to its elements using an index or a subscript, making it a very basic and fundamental data structure. Arrays are used extensively in programming for storing and manipulating data efficiently and are a fundamental building block for more complex data structures.
- Unfortunately, the number of operations applicable to arrays is limited due to their fixed (static) size. Assuming we have an array of size 100 and we want to add a new element, we would need to initialize a new array with a size of 101, copy the old 100 elements, and add the new element at index 100 (0-based). Erasing follows the same criteria. When searching for a specific element in the array, since arrays do not order the elements, we need to iterate through each element one by one to find the target, resulting in a time complexity of O(n).
- Although other data structures offer better time complexity for these operations, they come with their own disadvantages, such as the lack of random access. Therefore, we can see that random access is the most important feature of arrays.

Array Properties:

Property	Array
Size	**Fixed**
Random Access	**Yes**
Search	**O(N)**
Insert	**NO/ new array O(N)**
Erase	**NO/ new array O(N)**

b. Vector

```
#include <vector>
vector <type> name;
vector <int> v;
```



- A vector is similar to an array, but it offers some additional advantages, such as the ability to remove or insert an element from the end with a complexity of O(1). To be more accurate, adding an element to a vector takes amortized O(1) time. Actually, sometimes when we add an element, it takes O(1) time, while at other times it takes O(n) time. However, the number of times it takes O(1) time is much greater than the number of times it takes O(n) time. Therefore, if we calculate the total number of operations performed on the vector and divide it by the number of operations, it will be approximately 2-3 operations, which is considered a constant time complexity of O(1).

- Now, let's discuss the advantages and disadvantages of vectors. The first advantage is that vectors are dynamic, unlike arrays, which are fixed in size. Additionally, vectors maintain the random-access property, meaning that search time is the same as an array. However, when it comes to insertion and deletion, the time complexity is O(1) when performed from the end, but O(n) when performed from other positions.

i. Vector Properties:

Property	Vector
Size	**Dynamic**
Random Access	**Yes**
Search	**O(N)**
Insert: • **Front** • **Back** • **Middle**	 **O(N)** **O(1)** **O(N)**
Erase: • **Front** • **Back** • **Middle**	 **O(N)** **O(1)** **O(N)**

ii. Vector Constructors:

- `vector<double> v; // Empty vector`

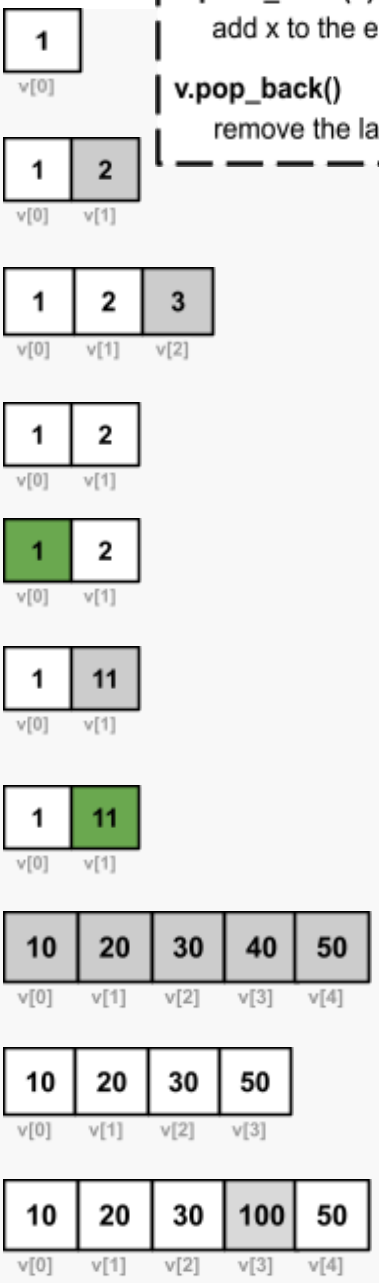
- `vector<long long> v(5); // Vector of size 5, initialized with zeros`
- `vector<int> v(5,3); // Vector of size 5, initialized with threes`

iii. Vector Common Operations:

v.push_back(x) O(1)
add x to the end of the vector v.

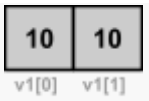
v.pop_back() O(1)
remove the last element.

```
vector<int> v(1, 1);
v.push_back(2);
v.push_back(3);
v.pop_back();
cout << v[0]; // prints  1
v[1] = 11;
cout << v[1]; // prints 11
v = {10, 20, 30, 40, 50};
v.erase(v.begin() + 3);
v.insert(v.begin()+3,100);
```

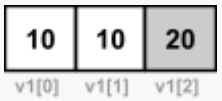


iv. Vector Common Operations (Shared on all STL containers)

```
vector<int> v1(2, 10);
```




```
v1.push_back(20);
```




```
cout << v1.size(); // prints 3
v1.clear(); // clears the vector
```

v.size()	number of elements
v.clear()	clear all elements
v.empty()	empty or not empty
v1.swap(v2)	swap v1 with v2

```
v1.push_back(10);
```


```
vector<int> v2(2, 20);
```


```
v1.swap(v2);
```


```
cout << v1[0]; // prints 20
```


```
cout << v2[0]; // prints 10
```

VI: Don't use the swap(x, y) method

```
v1.swap(v2);                    O(1)

swap(v1, v2); // implementation goes like this:
vector<int> temp = v1;          O(n)
v1 = v2;                        O(n)
v2 = temp;                      O(n)
v1.swap(v2); /* swaps the addresses (i.e. the containers
exchange references to their data) of two vectors rather
than swapping each element one by one which is done in
constant time O(1). */
```

v. Vector Traversing:

```
vector<int> v = {10, 20, 30, 40, 50};
```

Iterating through the elements of a vector is a common operation, and there are several methods to accomplish this task. In the following code examples, we will explore these methods and highlight the most frequently used ones among them.

Common Ways:

```
// array method
for (int i = 0; i < v.size(); i++)
      cout << v[i] << " ";

// range-based method
for(int elem: v)
      cout << elem << " ";
```

Note: Iterators are like pointers so to access the value inside them we need to de-reference using an asterisk (*it).

```
// forward iterator method
vector<int>::iterator it = v.begin();

while(it != v.end()){
      cout << *it << " ";
      it++;
}
// prints: 10 20 30 40 50
// reverse iterator method
```


```
vector<int>::reverse_iterator it = v.rbegin();
while(it != v.rend()){
      cout << *it << " ";
      it--;
}
// prints: 50 40 30 20 10
```

c. Deque

```
#include <deque>

deque <type> name;

deque <int> d;
```

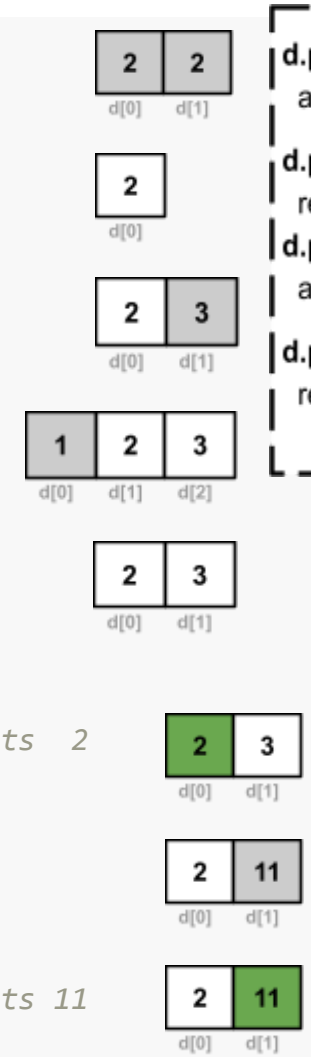
A Deque is similar to a vector, but it offers some additional advantages, which is the ability to remove or insert an element from the beginning with a complexity of O(1). To be more accurate, adding an element to a deque's beginning or end takes amortized O(1) time (Just the same way vector works).

i. Deque Properties:

Property	Deque
Size	Dynamic
Random Access	Yes
Search	O(N)
Insert: • Front • Back • Middle	 O(1) O(1) O(N)
Erase: • Front • Back • Middle	 O(1) O(1) O(N)

ii. Deque Common Operations:

```
deque<int> d(2, 2);
d.pop_back();
d.push_back(3);
d.push_front(1);
d.pop_front();

cout << d[0]; // prints  2
d[1] = 11;
cout << d[1]; // prints 11
```



d.push_back(x) O(1)
add x to the end of the deque d.

d.pop_back() O(1)
remove the last element.

d.push_front(x) O(1)
add x to the beginning of the deque d.

d.pop_front() O(1)
remove the first element.

As we have mentioned before, deque is similar to vector but with two extra functionalities which are push_front() and pop_front(). Therefore, there is no need to repeat the discussion as the same concepts apply to both data structures.

3. Container Adapters

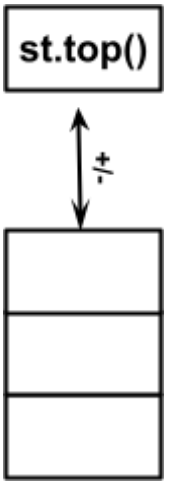
Container adaptors are designed to make it easy to understand the concept of the algorithm without having to clarify or comment on the algorithm steps. They can be used to change the way data is stored and accessed, to add or remove elements in a specific way.

Container Adapters are divided into:

- **Stack**
- **Queue**
- **Priority Queue**

a. Stack

```
#include <stack>

stack <type> name;

stack <int> st;
```



- A stack is a container adapter that allows elements to be added and removed from one end, known as the top. The stack follows the Last-In-First-Out (LIFO) principle, where the last element added to the stack will be the first to be removed.

- The stack supports three main methods: push, which adds an element to the top of the stack; pop, which removes the top element from the stack; and top, which returns the top element of the stack without removing it.

- Stacks are used in various computer science applications, such as expression evaluation, recursion, and undo/redo functionality. For instance, they are used in expression evaluation to store operands and operators and evaluate expressions in postfix notation. Also, they are used in recursion to keep track of function calls and return addresses and in undo/redo functionality to keep a history of changes. Overall, stacks are an essential concept in computer science and play a significant role in many applications.

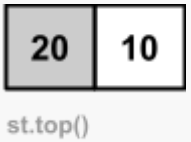
i. Stack Properties:

Property	Stack
Size	Dynamic
Random Access	No
Search	No
Insert (Front Only)	O(1)
Erase (Front Only)	O(1)

ii. Stack Common Operations:

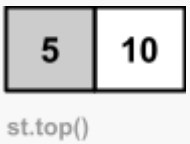
```
stack<int> st;

st.push(10);
```


```
st.push(20);
```




```
st.pop();
```


```
st.push(5);
```




```
s.top(); // returns 5
s.empty(); // returns false
s.size(); // returns 2
```

st.push(x) O(1)
add x to the top of the stack st.

st.pop() O(1)
remove the top element.

st.top() O(1)
get the value of the top element.

No Random Access
No Iterators

iii. Stack Traversing:

- Unfortunately, the stack is not traversable in the traditional sense. The only way to access the elements within a stack is by removing its top element. This is because stacks operate on the principle of (LIFO).
- If you need to access all the elements in a stack, you would have to repeatedly remove the top element until the entire stack has been emptied.

```
// The only way to traverse a stack
while(!st.empty()){
    cout << st.top() << " ";
    st.pop();
}
// prints 5, 10
```

iv. Stack Use Cases:

- Expression parsing
- Undo-redo operations

Example: Parenthesis matching

- A common use case of stacks is in checking if a given expression has matching parentheses. The algorithm works by iterating over the characters of the expression and pushing each opening parenthesis onto the stack.
- When a closing parenthesis is encountered, the stack is popped, and the parentheses are checked for a match.
- If a match is found, the algorithm continues to the next character.
- If the stack is empty or a mismatch is found, the algorithm returns false, indicating that the expression is invalid.

Here's an example implementation of the algorithm:

```
bool checkParenthesis(string expr) {
    stack<char> s;
    for (char c: expr) {
        if (c == '(')
            s.push(c);
        else if (c == ')') {
            if (s.empty() or s.top() != '(')
                return false;
            else
                s.pop();
        }
    }
    return s.empty();
}
```

Overall we can say that stack is an abstract data type that follows the Last-In-First-Out (LIFO) principle. It supports two basic operations: push (add element to top) and pop (remove element from top), commonly used to enhance algorithm clarity.

b. Queue

```
#include <queue>

queue <type> name;

queue <int> q;
```

- A queue is a container adapter that operates on a First-In-First-Out (FIFO) basis, meaning that the first element added to the queue is the first one to be removed. Elements can be inserted at one end of the queue (the back) and removed from the other end (the front).

- The queue data structure provides three main methods: push, which adds an element to the back of the queue; pop, which removes the front element from the queue, and front, which returns the front element without removing it.
- Queues are widely used in computer science applications such as operating systems, scheduling, and simulation. In operating systems, queues are used to store processes waiting to be executed, with the first process added to the queue being the first one to be executed. Similarly, in networking, packets of data are stored in a queue until they can be processed. Overall, queues are a fundamental concept in computer science, much like stacks, and play a crucial role in various applications.

i. Queue Properties:

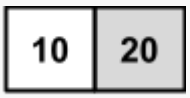
Property	Queue
Size	**Dynamic**
Random Access	**No**
Search	**No**
Insert (Back Only)	**O(1)**
Erase (Front Only)	**O(1)**

ii. Queue Common Operations:

```
queue<int> q;

q.push(10);
```

q.front()

```
q.push(20);
```

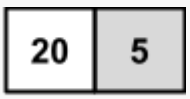


q.front()

```
q.pop();
```

q.front()

```
q.push(5);
```



q.front()

q.push(x) O(1)
add x to the beginning of the queue q.

q.pop() O(1)
remove the front element.

q.front() O(1)
get the value of the front element.

No Random Access
No Iterators

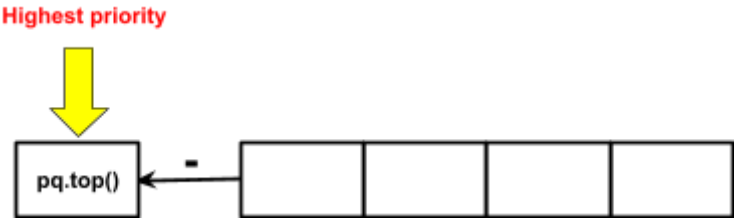
iii. Queue Traversing:

We can see that Queue is similar to Stack but with a different order.

```
// The only way to traverse a queue
while (!q.empty()){
   cout << q.front() << " "; // prints 10, 5
   q.pop();
}
```

c. Priority Queue

```
#include <queue>

priority_queue <type> name;

priority_queue <int> pq;
```



- A priority queue is a container adapter in C++ that allows elements to be added and removed based on their priority.
- Elements are stored in a way such that the highest priority element is always at the top, and elements with lower priority come after it.
- This property makes it suitable for scenarios where processing elements based on their priority is essential, such as in scheduling or event processing.
- The priority queue supports two primary operations: push and pop.
    - The **push** operation adds an element to the priority queue based on its priority,
    - while the **pop** operation removes the highest priority element from the priority queue.
    - Additionally, the **top** operation retrieves the highest priority element from the priority queue without removing it.

a. Priority queue Properties:

Property	Priority queue
Size	Dynamic
Random Access	No
Search	No
Insert	O(log(n))
Erase (Highest priority)	O(log(n))

b. Priority queue Common Operations:

For example, suppose we have a priority queue that stores integers in decreasing order of their value. We can use the push operation to add elements to the priority queue as follows:

pq.push(x) O(log(n))
add x to the queue q.

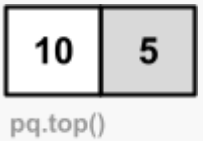
pq.pop() O(log(n))
remove the highest priority element.

pq.top() O(1)
get the value of the highest priority element

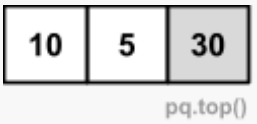
No Random Access
No Iterators

```
priority_queue<int> pq;

pq.push(10);
```

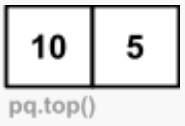

```
pq.push(5);
```



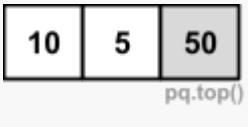

```
pq.push(30);
```




```
pq.pop();
```




```
pq.push(50);
```

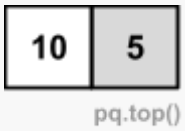


Another example, let's say we have a priority queue of integers in increasing order of their value. We can use the push operation to add elements to the priority queue as follows:

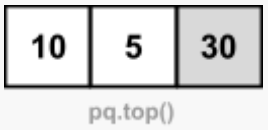
```
priority_queue<int,vector<int>,greater<>> pq;
```

```
pq.push(10);
```

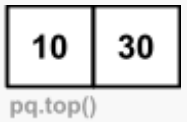

```
pq.push(5);
```



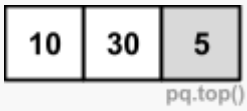

```
pq.push(30);
```




```
pq.pop();
```




```
pq.push(5);
```



c. Priority queue Traversing:

Traversing a priority queue is typically done using a loop that iteratively removes the highest priority element and processes it. We can use the top method to retrieve the highest priority element, and the pop method to remove it.

```
// The only way to traverse a priority queue
while(!pq.empty()){
   cout << pq.top() << " ";
   pq.pop();
}
// prints 5, 10, 30
```

Overall, priority queues are an essential concept in computer science and play a significant role in many applications where prioritization is crucial. The priority_queue container adapter in C++ provides an efficient way of implementing priority queues in our programs, and understanding its properties is essential for effectively working with it.

4. Comparison between some Containers

Property	Array	Vector	Deque	Stack	Queue	Priority Queue
Size	Static	Dynamic	Dynamic	Dynamic	Dynamic	Dynamic
Random access	Yes	Yes	Yes	No	No	No
Insert	NO/new array O(n)	Back: O(1) Other: O(n)	Front/Back: O(1) Other: O(n)	Front: O(1)	Back: O(1)	O(log(n))
Erase	NO/new array O(n)	Back: O(1) Other: O(n)	Front/Back: O(1) Other: O(n)	Front: O(1)	Front: O(1)	O(log(n))
Ordering	Ordered	Ordered	Ordered	Not applicable	Not applicable	Ordered
Usage	Used when storing data with static size is required	Used when frequent insertion/deletion in the middle or random access is needed	Used when frequent insertion/deletion is required in front/back or middle	Used when LIFO structure is required	Used when FIFO structure is required	Used when priority-based insertion/deletion is required

Note: The "ordering" column indicates whether the container preserves the order in which elements were inserted. For stack and queue, this is not applicable as they do not preserve the order. For the priority queue, the ordering is based on the priority level of the elements.

5. Associative containers

- Associative containers are special containers that keep their elements sorted at all times.
- When we add a new element, the container automatically finds the best place to put it in to maintain the sorted order. Unlike sequence containers, we cannot specify exactly where the new element should go, as the container has its own way of organizing the elements. Instead, we wait for the container to do its job and return an iterator pointing to the new element's position.
- One big advantage of associative containers is their fast search complexity, which is **O($log(n)$)**. This means that even with a large number of elements, searching for a specific element is much faster than with sequence containers, which have a search complexity of **O(n)**.
- Additionally, insertions and deletions are also efficient, with a complexity of **O($log(n)$)**, while still maintaining dynamic properties. However, we lose the ability to access elements randomly based on their original order.

a. Associative Properties

Property	Associative
Size	Dynamic
Random Access	No
Search	O(log n)
Insert	O(log n)
Erase	O(log n)

Before we dive into associative containers, let's first discuss a simple container called pair. As its name suggests, it holds a pair of values. When we create a pair, we specify two types for it - one for the first value and one for the second value. Let's look at an example to understand it better.

b. Pair

```
pair<type of first, type of second> p;

pair<string, int> p;

p.first = "ahmed";

p.second = 20;

p = {"ahmed", 20}; // C++ 11 or more

p = make_pair("ahmed", 20);
```

Associative Containers are divided into:

- **Set**
- **Multiset**
- **Map**
- **Multimap**

c. Set

```
#include <set>

set <type> name;

set <int> s;
```

s.insert(x)	O(log(n))
add x to the set s.	
s.erase(x)	O(log(n))
remove element x if exists in the set	
s.find(x)	O(log(n))
returns the iterator pointing to x and **s.end()** if x not exist	
No Random Access	

- Set is a container that organizes its elements in a particular order.
- It never contains any duplicates. This means that every element inside a set is unique.
- Set automatically sorts its elements as we insert them, ensuring that they remain in a specific order.

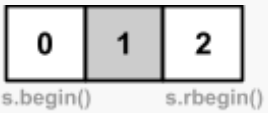
Let's take a closer look to understand how it works.

i. Set insert

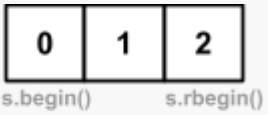
```
set<int> s;
/* insert function returns a pair, where the first is the
value to be inserted and the second is boolean indicating
whether the value is added or was already exist
*/
s.insert(2);
```


```
s.insert(0);
```

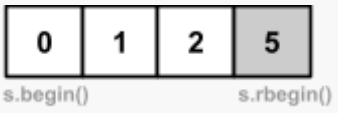

```
s.insert(1);
```




```
s.insert(0);
```

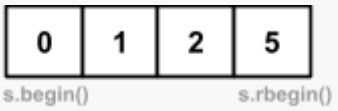



```
auto ins = s.insert(5);
```




```
cout << *(ins.first) << " " << ins.second; // 5 1
```

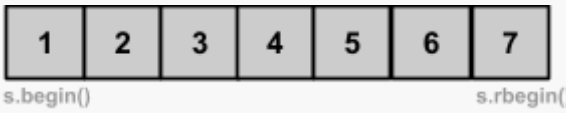
```
ins = s.insert(2);
```



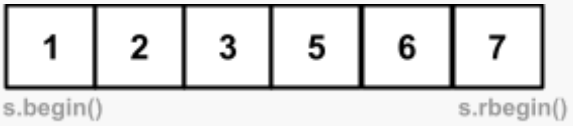

```
cout << *(ins.first) << " " << ins.second; // 2 0
```

ii. Set erase

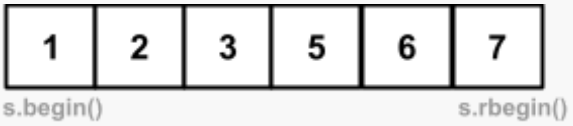
```
set<int> s = {1, 2, 3, 4, 5, 6, 7};
```



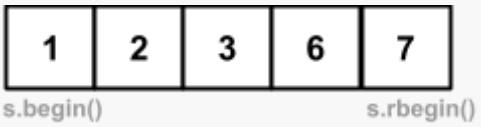

```
// erase function returns an integer which is the number of
items erased
s.erase(4);
```



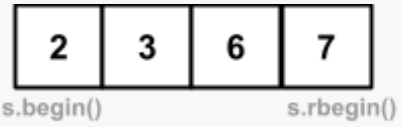

```
int cnt = s.erase(9);
// cnt = 0
```



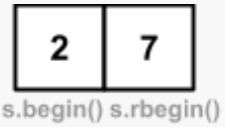

```
cnt = s.erase(5); // cnt = 1
```




```
// erase can work with iterators too
s.erase(s.begin());
```




```
auto it1 = s.begin(), it2 =
s.end();
it1++, it2--;
s.erase(it1, it2); /* O(log(n)+ NO. of
erased elements)
also erase works with ranges but the
range must be iterators range
[start,end[ (start included, end excluded)*/
```



iii. Set find

```
auto it = s.find(10);
if(it!= s.end())
   cout << "Found" << endl;
else
   cout << "Not Found" << endl;

if(s.count(10))
   cout << "Found" << endl;
else
   cout << "Not Found" << endl;
```

iv. Set Traversing

```
// forward iterator method
for(auto it = s.begin(); it != s.end(); it++)
   cout << *it << " "; // asterisk to get the value

// range-based method
for(auto it: s)
   cout << it << " ";
```

v. Set Use Cases

- **Finding distinct elements:** You may be given a list of numbers or strings and be asked to find the number of distinct elements in the list. This can be easily achieved by inserting each element into a set, as Sets only store unique elements. Here's an example:

```
int countDistinct(vector<element> v) {
   set<element> st;
   for (auto elem : v)
       st.insert(elem);
   return st.size();
}
```

- **Removing duplicates:** Sets are very useful when it comes to removing duplicates from an array or vector. Suppose you have an array of integers and you want to remove duplicates from it. You

can simply insert all the elements of the array into a set, which will automatically remove duplicates. Here's an example:

```
void removeDuplicates(vector<element> v) {
   set<element> st;
   for (auto elem: v)
       st.insert(elem);
   for(auto elem : st)
       cout << elem << " ";
}
```

Overall, Set is an important data structure in computer science, providing efficient access, insertion, and deletion of elements, while maintaining the ordering and uniqueness of the elements.

d. Multiset

```
#include <set>

multiset <type> name;

multiset <int> ms;
```

Multisets are just a version of Sets that allow duplicate values and almost have the same operations as Sets.

ms.insert(x) O(log(n))
add x to ms.

ms.erase(x) O(log(n))
remove all occurrences of x (must be exist)

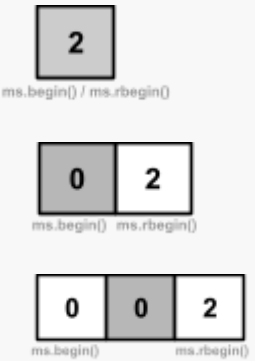
ms.find(x) O(log(n))
returns the iterator pointing to x and **ms.end()** if x not exist

ms.count(x) O(log(n) + freq[x])
count number of occurrences of x

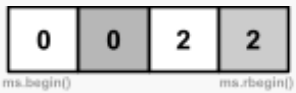
No Random Access

i. Multiset Common Operations

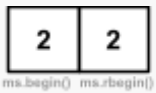
```
multiset<int> ms;

ms.insert(2);

ms.insert(0);

ms.insert(0);
```



```
ms.insert(2);
```




```
// the erase removes all
occurrences of a value
ms.erase(0);
```




```
// to remove exactly one occurrence we would need an
iterator pointing to the value which is returned from the
.find() function

ms.erase(ms.find(2));
```

Do not use the count function if u just need to check whether the element exist or not.

ii. Multiset Use Cases

- **Finding kth smallest/largest element:** We can use a multiset to store the elements and then use the nth_element algorithm to find the kth smallest/largest element efficiently. For example, given an array of integers and an integer k, we can find the kth smallest element using the following code:

```
int kthSmallest(vector<int> v, int k) {
   multiset<int> ms;
   for (int i = 0; i < (int) v.size(); i++)
       ms.insert(v[i]);
   auto it = ms.begin();
   advance(it, k - 1);
   return *it;
}
```

- **Sliding window problems:** In some problems, we need to maintain a sliding window of elements and perform operations on that window, such as finding the minimum or maximum element in the window. We can use a multiset to store the elements in the window and perform these operations efficiently. For example,

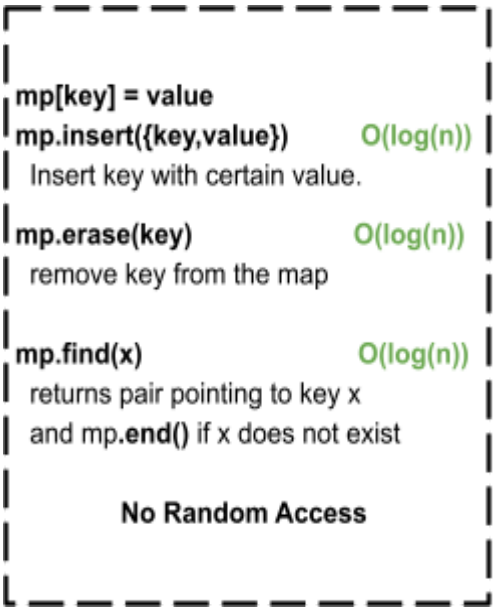
given an array of integers and a window size k, we can find the maximum element in each window using the following code:

```
vector<int> maxInWindow(vector<int> v, int k) {
    vector<int> window;
    multiset<int> ms;
    for (auto it : v)
        ms.insert(it);
    for (int i = k - 1; i < v.size(); i++) {
        window.push_back(*ms.rbegin());
        ms.erase(ms.find(v[i - k]));
        ms.insert(v[i]);
    }
    return window;
}
```

Overall, multisets can be a powerful tool in competitive programming, allowing us to efficiently store and manipulate a collection of elements, including duplicates.

e. Map

```
#include <map>

map <key_type,value_type> name;

map <string, int> mp;
```



- Maps enable us to store pairs of values that are associated with each other.
- We can access the second value in each pair by using the first value as a **key**.
- Maps are ordered (sorted) and do not contain duplicates (According to keys).

i. Map Common Operations

```
// key is string and value is int
map<string, int> fruits;
fruits["Apple"] = 10;
fruits["Orange"] = 2;
// another way to add value to map
fruits.insert({"Banana", 6});

cout << fruits["Apple"]; //prints 10

// if we tried to access a key which does not exist the map
will create a pair with that key and a default value for int

cout << fruits["Avocado"]; //prints 0
fruits.erase("orange");
fruits["Apple"] = 9;

auto it = fruits.find("Apple");
if (it != fruits.end())
   cout << (*it).first << " " << it->second << endl;
      //prints Ahmed 9
```

Apple	10
Banana	6
Orange	2

Apple	9
Avocado	0

ii. Map Traversing

```
// forward iterator method
for(auto it = mp.begin(); it != s.end(); it++)
   cout << (*it).first << " " << it->second << endl;

// range-based method
for(auto it : mp)
   cout << it.first << " " << it.second << endl;
```

iii. Map Use Cases

- **Counting the frequency of elements:** A map can be used to count the frequency of elements in an array or sequence. We can iterate over the sequence, insert each element as a key in the map, and increment its value each time we encounter it. Here's an example:

```
void countFrequency(vector<element> v) {
   map<element, int> freq;
   for (int i = 0; i < (int)v.size(); i++)
       freq[v[i]]++;
   for (auto elem: freq)
       cout << elem.first << " occurs " << elem.second
<< " times" << endl;
}
```

- **Finding nearest elements:** A map can be used to find the nearest elements to a given value. For example, we can use a map to store the positions of elements in an array, where the keys are the elements themselves and the values are their positions. Then, given a value x, we can find the nearest elements to x by finding the lower and upper bounds of x in the map. Here's an example:

```
int nearest(vector<int> v, int target) {
   map<int, int> pos;
   for (int i = 0; i < (int) v.size(); i++)
       pos[v[i]] = i;
   auto it = pos.lower_bound(target);
   if (it == pos.begin())
       cout << v[it->second] << endl;
   else if (it == pos.end()) {
       it--;
       cout << v[it->second] << endl;
   } else {
       auto it2 = it--;
       cout << ((target - it->first < it2->first -
target) ? v[it->second] : v[it2->second]) << endl;
   }
}
```

Overall, maps are very versatile and can be used in many different types of problems in competitive programming.

f. Multimap

```
#include <map>

multimap<key_type,value_type> name;

multimap <string, int> mp;
```

mp.insert({key,value}) O(log(n))
Insert key with a certain value.

mp.erase(key) O(log(n) + freq[x])
remove all occurrences of key from the map

mp.find(x) O(log(n))
returns pair pointing to the first {key,value}, key = x, and mp.**end()** if x does not exist

mp.count(x) O(log(n) + freq[x])
count number of occurrences of key x

No Random Access

Multimaps are just a version of Maps that allow duplicate values and almost have the same operations as Maps.

i. Multimap Common Operations

```
// key is string and value is int
multimap<string, int> fruits;
// the only way to insert into map
fruits.insert({"Apple", 2});
fruits.insert({"Apple", 4});
fruits.insert({"Apple", 6});
fruits.insert({"Banana", 10});


// this method is not available in
multimap
cout << fruits["Apple"];

cout << fruits.count("Apple"); // prints 3

fruits.erase("Apple");
// removes all occurrences of key
```

Apple	2
Apple	4
Apple	6
Banana	10

Banana	10

 Do not use the count function if u just need to check whether the element exist or not.

6. Important methods

Vector	Deque	Stack	Queue	Priority Queue	Set	Map
size	size	size	size	size	size	size
swap	swap	swap	swap	swap	swap	swap
push_back	push_back	push	push	push	insert	insert
pop_back	pop_back	pop	pop	pop	erase	erase
insert	push_front	top	front	top	find	find
erase	pop_front				count (multi)	count (multi)
clear	clear				clear	clear
	insert					
	erase					

7. Unordered containers

Unordered containers are associative containers that were introduced in C++11. They use a concept called hashing to optimize the average complexity of search, insertion, and deletion operations to O(1). However, in the worst case, the complexity can still be O(n). While we are not always interested in using unordered containers for competitive programming, they can be useful in real-life applications and in some cases in competitions when we are confident that our algorithm will not encounter worst-case scenarios.

Unordered Containers are divided into:

a. **unordered_set**
b. **unordered_multiset**
c. **unordered_map**
d. **unordered_multimap**

8. Reasons to use C++ STL:

- There is no need to develop data structures or algorithms from scratch as C++ developers have already provided the Standard Template Library (STL), which offers numerous benefits.
- The STL is a comprehensive library of reusable containers, iterators, and algorithms that provide efficient, bug-free, simplified code, standardized, and well-tested solutions to common programming tasks.
- By using the STL, developers can save time and effort, reduce errors, improve code quality, and increase productivity. Therefore, it is highly recommended to utilize the STL and its components whenever possible.