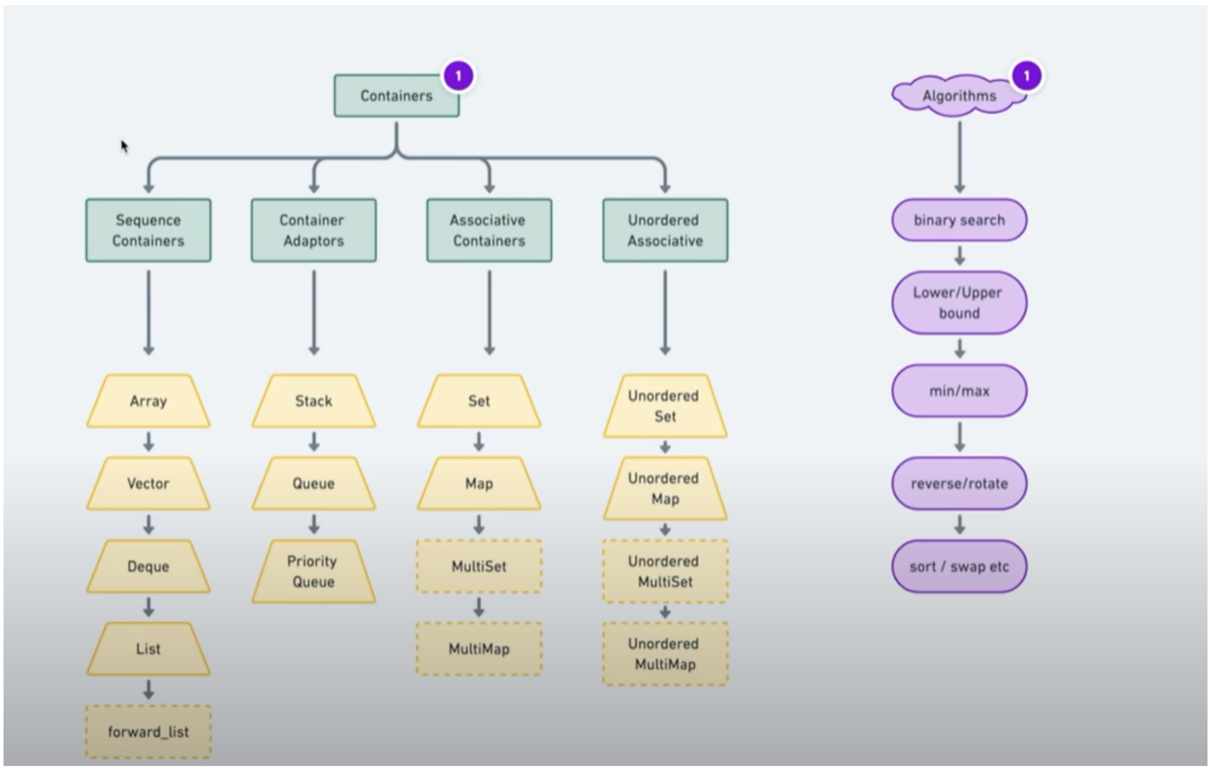
STL C++

Author	
Property	

STL can be divided into two parts: Containers and Algorithms



Containers

Arrays

Data is stored in contagious memory allocation. That is, it is static.

It's implementation is done by the basic static array only

```
array<int, 4> a = {1, 2, 3, 4};
```

```
a.size();
```

All the operations done here is of O(1) time complexity.

```
a.at(2);
```

```
a.empty();
```

```
a.front(); // returns the first element
a.back(); // return the last element of the array
```

Vectors

Vectors are dynamic, i.e., the size can be increased or decreased.

- It tries to double its size if we try to add any element.
- It creates another vector with double the size and copies all the elements existing in the older one.
- And it dumps the older vector

```
vector<int> v;
vector<int> a(5, 1); // Size: 5; All initialized with 1
```

Size tells the current number of elements and ***capacity*** tells the total memory allocated.

```
v.capacity();
v.size();
```

Function *.erase(),* time complexity: O(n)

```
v.push_back(3); // to enter an element in the vector array
v.at(2); // Element at index = 2
v.front(); // 1st Element
v.back(); // Last Element
v.pop_back(); // rempves the last element
```

Print the vector as: -

```
for(int i:v) {
  cout << i << " ";
}
```

Clearing all the elements:

```
v.clear();
```

```
vector<int> last(a);
```

Deque

- We can perform insertion, deletion etc from both front and back.
- Here, we have multiple static arrays, unlike arrays and vectors, which have contiguous memory allocations

```
deque<int> d;
d.push_back(1);
d.push_back(3);
for (int i:d) {
  cout << i << " ";
}
d.at(); // tells the element at ith index
```

Pop Back and Front Functions

```
d.pop_back();
d.pop_front();
```

```
d.front(); // returns the 1st element
d.back(); // returns the last element
d.empty(); // returns boolean, 0: if not empty, 1: if empty
```

Erase Function

```
d.erase(d.begin(), d.begin()+1); // erases the 1st element
```

List

- It uses the implementation of doubly linked lists.
- Front and back pointers
- We need to traverse till the ith element if we want the ith element

```
list<int> l;
l.push_back(1);
l.push_front(2);
for(int i:l){
    cout << i << " ";
}
list<int> n(l); // copies the list l
```

Erase Function:

- Time Complexity for Erase function: O(n)

```
l.erase(l.begin));
```

Stacks

- With the concept of last-in-first-out

```
stack<string> s;
s.push("Shara");
s.push("Ari");
s.push("Arak");
```

```cpp
cout << "Top Element: " << s.top() << endl;
s.pop(); // removes "Arak"
s.size(); // size of stack
s.empty(); // returns boolean if empty or not
```

Queue

- Forming a line
- First-in-first-out

```cpp
queue<string> q;
q.push("Shara");
q.push("Ari");
q.push("Arak");
```

```cpp
q.front(); // returns the first element
q.pop(); // removes the last element
q.size();
```

Priority Queue

- Its first element will always be the ***greatest***
- Like Max Heap
- If we make a max heap, it always returns the maximum element while extracting
- If we make the min heap, it always returns the minimum element while extracting

```cpp
// For Max Heap
priority_queue<int> maxh;
// For Min Heap
priority_queue<int, vector<int>, greater<int> > minh;
```

```
maxh.push(1);
maxh.push(3);
maxh.push(0);
maxh.push(5);
```

```
cout << "Size: " << maxh.size() << endl;

int n = maxh.size();
for(int i = 0; i < n; i++) {
   cout << maxh.top() << " ";
   maxh.pop();
}
cout << endl;
// returns values in descending order
```

```
minh.push(5);
minh.push(1);
minh.push(0);
minh.push(4);
minh.push(3);
```

```
int m = minh.size();
for(int i = 0; i < m; i++) {
    cout << minh.top() << " ";
    minh.pop();
}
// will return values in ascending order
cout << "Empty or not: " << minh.empty() << endl;
```

Sets

- Every element should be unique
- Its implementation is done on the basis of BST
- No modification can be done
- Only insertion and deletion
- Returns elements in sorted order
- Set is a bit slower than unordered set
- Element doesn’t come in a sorted order when in unordered

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

Time complexity of *insert()*: O(log(n))

```
s.insert(5);
s.insert(5);
s.insert(0);
s.insert(6);
s.insert(1);

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

```
set<int>::iterator it = s.begin();
it++;

s.erase(it); // will delete the second element
for(auto i:s) {
    cout << i << endl;
}

cout << "5 is present or not: " << s.count(5) << endl;

set<int>::iterator itr = s.find(5);

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

Maps

- Stores data as key values
- All keys are unique and all points to the same value
- When printed, it returns the values in a sorted way
- Time complexity for all the operations: *O(log(n))*

```
map<int, string> m; // key: int; value: string
m[1] = "shara";
m[2] = "arak";
m[13] = "ari";
```

```
m.insert( {5, "aram"} );

for (auto i:m) {
    cout << i.first << " " << i.second << endl;
}
// i.first: for returning keys; i.second: for returning values
```

Erase

```
m.erase(13);
// erases key = 13 from the map
```

- For unordered map: Time Complexity = *O(n)*
- find() : returns the iterator of the element

```
auto it = m.find(5);

for (auto i = it; i != m.end(); i++) {
    cout << (*i).first << endl;
}
```

Algorithms

The functions used in here are optimized from the one used

- Conditions for Binary Search: Elements should be initialized in a sorted order only

```
vector<int> v;
v.push_back(1);
v.push_back(3);
v.push_back(6);
v.push_back(7);
```

```
cout << "Finding 6: " << binary_search(v.begin(), v.end(), 6) << endl;
    // returns boolean
```

```
cout << "Lower Bound: " << lower_bound(v.begin(), v.end(), 6)-v.begin() << endl;
cout << "Upper Bound: " << upper_bound(v.begin(), v.end(), 6)-v.begin() << endl;
```

Max and Min

```
int a = 3;
int b = 5;

cout << "Max: " << max(a, b) << "\n";
cout << "Min: " << min(a, b) << "\n";
```

Swap and Reverse

```
swap(a, b);
cout << "After swap, a = " << a << endl;
string s = "abcd";
reverse(s.begin(), s.end());
cout << "After reverse, string s: " << s << endl;
```

Rotate

```
rotate(v.begin(), v.begin()+1, v.end());
cout << "After rotate: " << endl;
for(int i:v) {
    cout << i << " ";
}
cout << endl;
```

Sort

- Based on intro sort: which is combination of three algos; heap sort, quick sort and insertion sort

```
sort(v.begin(), v.end());

cout << "After sort: " << endl;
for(int i:v) {
    cout << i << " ";
}
cout << endl;
```