List of Data Structures C++ 11 standard

1. Array
2. Vector
3. List
4. Queue
5. Priority_queue
6. Stack
7. Map
8. Unordered_map
9. Set
10. Unordered_set
11. Linked List
12. Binary tree

Overview of Complexity

Data Structures

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Indexing	Search	Insertion	Deletion	Indexing	Search	Insertion	Deletion	
Basic Array	O(1)	O(n)	-	-	O(1)	O(n)	-	-	O(n)
Dynamic Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartresian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

ARRAY: #include **<vector>**

push_back( ) pop_back( ) O(1) operator [ ]	begin( ) end( ) insert( ) } O(n) erase( )	size() empty()

LIST: #include **<list>**

push_front( ) push_back( ) pop_back( ) O(1) pop_front( )	begin( ) end( ) insert( ) } O(n) erase( )	size() empty()

STACK: #include **<stack>**

push( ) pop( ) O(1) top( )		size() empty()

QUEUE: #include **<queue>**

push( ) pop( ) front( ) O(1) back( )		size() empty()

MAX BINARY HEAP: #include **<priority_queue>**

push( ) O(log n) pop( )	top() O(1)	size() empty()

BINARY SEARCH TREE: #include **<set>** or **<map>**

insert( ) erase( ) O(log n) operator [ ] (map only)	begin( ) end( ) find( ) O(log n) *return iterator or set::end* count( ) *return 0 or 1*	size() empty()

Nota: if you want to insert several times the same value prefer to use **<multiset>** or **<multimap>** but operator [] will be not available anymore.

HASH TABLE: #include **<unordered_set>** or **<unordered_map>**

insert( ) erase( ) O(1) operator [ ] (unordered_map only)	begin( ) end( ) find( ) O(1) *return iterator or set::end* count( ) *return 0 or 1* reserve( ) *set the number of buckets*	size() empty()

Nota: hash_table is a very efficient data structure but elements can not be ordered.

Data Structure Details

Vector

Pros:

- Good for adding but not deleting
- Quick access

Cons:

- Slow insertion/deletion in the middle of the array
- Slow when dynamically changing storage

Constructors:

default (1)	`explicit vector (const allocator_type& alloc = allocator_type());`

fill (2)	`explicit vector (size_type n);`
	`vector (size_type n, const value_type& val,`
	`const allocator_type& alloc = allocator_type());`

range (3)	`template <class InputIterator>`
	`vector (InputIterator first, InputIterator last,`
	`const allocator_type& alloc = allocator_type());`

copy (4)	`vector (const vector& x);` `vector (const vector& x, const allocator_type& alloc);`
move (5)	`vector (vector&& x);` `vector (vector&& x, const allocator_type& alloc);`
initializer list (6)	`vector (initializer_list<value_type> il,` `const allocator_type& alloc = allocator_type());`

Constructor Examples:

```
// constructing vectors
#include <iostream>
#include <vector>

int main ()
{
  // constructors used in the same order as described above:
  std::vector<int> first;                                // empty vector of ints
  std::vector<int> second (4,100);                       // four ints with value 100
  std::vector<int> third (second.begin(),second.end());  // iterating through second
  std::vector<int> fourth (third);                       // a copy of third

  // the iterator constructor can also be used to construct from arrays:
  int myints[] = {16,2,77,29};
  std::vector<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

  std::cout << "The contents of fifth are:";
  for (std::vector<int>::iterator it = fifth.begin(); it != fifth.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
```

Iterators:

std::vector::begin()	
std::vector::end()	

Capacitor:

size()	size_type size() const
capacity()	size_type capacity() const
empty()	bool empty() const

Element Access:

operator[]	O(1)	
front()		reference front()
back()		reference back()

Modifiers:

push_back()	O(1)	void push_back(const value_type& val)
pop_back()	O(1)	void pop_back();
insert()	O(n)	iterator insert (iterator position, const value_type &val)
		void insert (iterator position, size_type n, const value_type *val)

Queue

Pros:

- First-In First-Out operations
- BFS

Constructors:

```
initialize (1)              explicit queue (const container_type& ctnr);
move-initialize (2)         explicit queue (container_type&& ctnr = container_type());
allocator (3)               template <class Alloc> explicit queue (const Alloc& alloc);
init + allocator (4)        template <class Alloc> queue (const container_type& ctnr, const Alloc& alloc);
move-init + allocator (5)   template <class Alloc> queue (container_type&& ctnr, const Alloc& alloc);
copy + allocator (6)        template <class Alloc> queue (const queue& x, const Alloc& alloc);
move + allocator (7)        template <class Alloc> queue (queue&& x, const Alloc& alloc);
```

Constructor Examples:

```
// constructing queues
#include <iostream>       // std::cout
#include <deque>          // std::deque
#include <list>           // std::list
#include <queue>          // std::queue

int main ()
{
  std::deque<int> mydeck (3,100);        // deque with 3 elements
  std::list<int> mylist (2,200);         // list with 2 elements

  std::queue<int> first;                 // empty queue
  std::queue<int> second (mydeck);       // queue initialized to copy of deque

  std::queue<int,std::list<int> > third; // empty queue with list as underlying container
  std::queue<int,std::list<int> > fourth (mylist);

  std::cout << "size of first: " << first.size() << '\n';
  std::cout << "size of second: " << second.size() << '\n';
  std::cout << "size of third: " << third.size() << '\n';
  std::cout << "size of fourth: " << fourth.size() << '\n';

  return 0;
}
```

Member Functions:

empty()		bool empty() const
size()		size_type size() const
front()	O(1)	reference &front()
back()	O(1)	reference &back()
push()	O(1)	void push (const value_type &val)
pop()	O(1)	void pop();

Priority Queue

Note:
When overloading operator, the left variable in the top element of the priority queue

Description:

Priority queues are a type of container adaptors, specifically designed such that its first element is always the greatest of the elements it contains, according to some strict weak ordering criterion.

This context is similar to a heap, where elements can be inserted at any moment, and only the max heap element can be retrieved (the one at the top in the priority queue).

Priority queues are implemented as container adaptors, which are classes that use an encapsulated object of a specific container class as its underlying container, providing a specific set of member functions to access its elements. Elements are popped from the "back" of the specific container, which is known as the top of the priority queue.

The underlying container may be any of the standard container class templates or some other specifically designed container class. The container shall be accessible through random access iterators and support the following operations:

empty()
size()
front()
push_back()
pop_back()

The standard container classes vector and deque fulfill these requirements. By default, if no container class is specified for a particular priority_queue class instantiation, the standard container vector is used.

Constructor:

```
initialize (1)  priority_queue (const Compare& comp, const Container& ctnr);
                template <class InputIterator>
range (2)         priority_queue (InputIterator first, InputIterator last,
                                  const Compare& comp, const Container& ctnr);
```

Constructor Example:

```
// constructing priority queues
#include <iostream>       // std::cout
#include <queue>          // std::priority_queue
#include <vector>         // std::vector
#include <functional>     // std::greater

class mycomparison
{
  bool reverse;
public:
  mycomparison(const bool& revparam=false)
    {reverse=revparam;}
  bool operator() (const int& lhs, const int&rhs) const
  {
    if (reverse) return (lhs>rhs);
    else return (lhs<rhs);
  }
};

int main ()
{
  int myints[]= {10,60,50,20};

  std::priority_queue<int> first;
  std::priority_queue<int> second (myints,myints+4);
  std::priority_queue<int, std::vector<int>, std::greater<int> >
                           third (myints,myints+4);
  // using mycomparison:
  typedef std::priority_queue<int,std::vector<int>,mycomparison> mypq_type;

  mypq_type fourth;                         // less-than comparison
  mypq_type fifth (mycomparison(true));   // greater-than comparison

  return 0;
}
```

The example does not produce any output, but it constructs different priority_queue objects:
- First is empty.
- Second contains the four ints defined for `myints`, with `60` (the highest) at its top.
- Third has the same four ints, but because it uses greater instead of the default (which is less), it has 10 as its top element.
- Fourth and fifth are very similar to first: they are both empty, except that these use `mycomparison` for comparisons, which is a special stateful comparison function that behaves differently depending on a flag set on construction.

Member Functions:

empty()
size()
top()
push()
pop()

Stack

Description

Stacks are a type of container adaptor, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from one end of the container.

stacks are implemented as *containers adaptors*, which are classes that use an encapsulated object of a specific container class as its *underlying container*, providing a specific set of member functions to access its elements. Elements are *pushed*/*popped* from the *"back"* of the specific container, which is known as the *top* of the stack.

The underlying container may be any of the standard container class templates or some other specifically designed container class. The container shall support the following operations:

- `empty`
- `size`
- `back`
- `push_back`
- `pop_back`

The standard container classes `vector`, `deque` and `list` fulfill these requirements. By default, if no container class is specified for a particular `stack` class instantiation, the standard container `deque` is used.

Constructor

initialize (1)	`explicit stack (const container_type& ctnr);`
move-initialize (2)	`explicit stack (container_type&& ctnr = container_type());`
allocator (3)	`template <class Alloc> explicit stack (const Alloc& alloc);`
init + allocator (4)	`template <class Alloc> stack (const container_type& ctnr, const Alloc& alloc);`
move-init + allocator (5)	`template <class Alloc> stack (container_type&& ctnr, const Alloc& alloc);`
copy + allocator (6)	`template <class Alloc> stack (const stack& x, const Alloc& alloc);`
move + allocator (7)	`template <class Alloc> stack (stack&& x, const Alloc& alloc);`

Constructor Examples:

```
// constructing stacks
#include <iostream>       // std::cout
#include <stack>          // std::stack
#include <vector>         // std::vector
#include <deque>          // std::deque

int main ()
{
  std::deque<int> mydeque (3,100);          // deque with 3 elements
  std::vector<int> myvector (2,200);        // vector with 2 elements

  std::stack<int> first;                    // empty stack
  std::stack<int> second (mydeque);         // stack initialized to copy of deque

  std::stack<int,std::vector<int> > third;  // empty stack using vector
  std::stack<int,std::vector<int> > fourth (myvector);

  std::cout << "size of first: " << first.size() << '\n';
  std::cout << "size of second: " << second.size() << '\n';
  std::cout << "size of third: " << third.size() << '\n';
  std::cout << "size of fourth: " << fourth.size() << '\n';

  return 0;
}
```

Member Functions:

empty()
size()
top()
push()
pop()

---

map

Description:

Maps are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order.

In a map, the key values are generally used to sort and uniquely identify the elements, while the mapped values store the content associated to this key. The types of key and mapped value may differ, and are grouped together in member type value_type, which is a pair type combining both:

```
typedef pair<const Key, T> value_type;
```

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

map containers are generally slower than unordered_map containers to access individual elements by their key, but they allow the direct iteration on subsets based on their order.

The mapped values in a map can be accessed directly by their corresponding key using the bracket operator ((operator[]).

Maps are typically implemented as binary search trees.

Constructor:

empty (1)	`explicit map (const key_compare& comp = key_compare(),`
	`  const allocator_type& alloc = allocator_type());`
	`explicit map (const allocator_type& alloc);`

range (2)	`template <class InputIterator>`
	`  map (InputIterator first, InputIterator last,`
	`  const key_compare& comp = key_compare(),`
	`  const allocator_type& = allocator_type());`

copy (3)	`map (const map& x);` `map (const map& x, const allocator_type& alloc);`
move (4)	`map (map&& x);` `map (map&& x, const allocator_type& alloc);`

initializer list (5)	`map (initializer_list<value_type> il,`
	`  const key_compare& comp = key_compare(),`
	`  const allocator_type& alloc = allocator_type());`

Constructor Examples:

```
// constructing maps
#include <iostream>
#include <map>

bool fncomp (char lhs, char rhs) {return lhs<rhs;}

struct classcomp {
  bool operator() (const char& lhs, const char& rhs) const
  {return lhs<rhs;}
};

int main ()
{
  std::map<char,int> first;

  first['a']=10;
  first['b']=30;
  first['c']=50;
  first['d']=70;

  std::map<char,int> second (first.begin(),first.end());

  std::map<char,int> third (second);

  std::map<char,int,classcomp> fourth;                 // class as Compare

  bool(*fn_pt)(char,char) = fncomp;
  std::map<char,int,bool(*)(char,char)> fifth (fn_pt); // function pointer as
Compare

  return 0;
}
```

Iterators:

std::vector::begin()	
std::vector::end()	

Capacity:

size()	size_type size() const
empty()	bool empty() const

Element Access:

operator[]	

Modifiers:

insert()	pair<iterator, bool> insert (const value_type &val)
erase()	iterator erase (const_iterator position)
	size_type erase (const key_type &k)
	iterator erase (const_iterator first, const_iterator last)

Operations:

find	iterator find (const key_type &k)

Unordered_map

Description:

Unordered maps are associative containers that store elements formed by the combination of a key value and a mapped value, and which allows for fast retrieval of individual elements based on their keys.

In an unordered_map, the key value is generally used to uniquely identify the element, while the mapped value is an object with the content associated to this key. Types of key and mapped value may differ.

Internally, the elements in the unordered_map are not sorted in any particular order with respect to either their key or mapped values, but organized into buckets depending on their hash values to allow for fast access to individual elements directly by their key values (with a constant average time complexity on average).

unordered_map containers are faster than map containers to access individual elements by their key, although they are generally less efficient for range iteration through a subset of their elements.

Unordered maps implement the direct access operator (operator[]) which allows for direct access of the mapped value using its key value as argument.

Iterators in the container are at least forward iterators.

Constructors:

empty (1)	`explicit unordered_map ( size_type n = /* see below */,`
	`const hasher& hf = hasher(),`
	`const key_equal& eql = key_equal(),`
	`const allocator_type& alloc = allocator_type() );`
	`explicit unordered_map ( const allocator_type& alloc );`

range (2)	`template <class InputIterator>`
	`unordered_map ( InputIterator first, InputIterator last,`
	`size_type n = /* see below */,`
	`const hasher& hf = hasher(),`
	`const key_equal& eql = key_equal(),`
	`const allocator_type& alloc = allocator_type() );`

copy (3)	`unordered_map ( const unordered_map& ump );` `unordered_map ( const unordered_map& ump, const allocator_type& alloc );`
move (4)	`unordered_map ( unordered_map&& ump );` `unordered_map ( unordered_map&& ump, const allocator_type& alloc );`

initializer list (5)	`unordered_map ( initializer_list<value_type> il,`
	`size_type n = /* see below */,`
	`const hasher& hf = hasher(),`
	`const key_equal& eql = key_equal(),`
	`const allocator_type& alloc = allocator_type() );`

Constructor Examples:

```
// constructing unordered_maps
#include <iostream>
#include <string>
#include <unordered_map>

typedef std::unordered_map<std::string,std::string> stringmap;

stringmap merge (stringmap a,stringmap b) {
  stringmap temp(a); temp.insert(b.begin(),b.end()); return temp;
}

int main ()
{
  stringmap first;                                  // empty
  stringmap second ( {{"apple","red"},{"lemon","yellow"}} );        // init list
  stringmap third ( {{"orange","orange"},{"strawberry","red"}} );  // init list
  stringmap fourth (second);                        // copy
  stringmap fifth (merge(third,fourth));            // move
  stringmap sixth (fifth.begin(),fifth.end());  // range

  std::cout << "sixth contains:";
  for (auto& x: sixth) std::cout << " " << x.first << ":" << x.second;
  std::cout << std::endl;

  return 0;
}
```

Iterators:

std::vector::begin()
std::vector::end()

Capacity:

size()
capacity()
empty()

Element Access:

operator[]	

Modifiers

insert()
erase()

Set

Description:

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).

set containers are generally slower than 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.

Constructor:

empty (1)	`explicit set (const key_compare& comp = key_compare(),`
	`  const allocator_type& alloc = allocator_type());`
	`explicit set (const allocator_type& alloc);`

range (2)	`template <class InputIterator>`
	`  set (InputIterator first, InputIterator last,`
	`  const key_compare& comp = key_compare(),`
	`  const allocator_type& = allocator_type());`

copy (3)	`set (const set& x);` `set (const set& x, const allocator_type& alloc);`
move (4)	`set (set&& x);` `set (set&& x, const allocator_type& alloc);`

initializer list (5)	`set (initializer_list<value_type> il,`
	`  const key_compare& comp = key_compare(),`
	`  const allocator_type& alloc = allocator_type());`

Constructor Examples:

```
// constructing sets
#include <iostream>
#include <set>

bool fncomp (int lhs, int rhs) {return lhs<rhs;}

struct classcomp {
  bool operator() (const int& lhs, const int& rhs) const
  {return lhs<rhs;}
};

int main ()
{
  std::set<int> first;                           // empty set of ints

  int myints[]= {10,20,30,40,50};
  std::set<int> second (myints,myints+5);        // range

  std::set<int> third (second);                  // a copy of second

  std::set<int> fourth (second.begin(), second.end());  // iterator ctor.

  std::set<int,classcomp> fifth;                 // class as Compare

  bool(*fn_pt)(int,int) = fncomp;
  std::set<int,bool(*)(int,int)> sixth (fn_pt);  // function pointer as Compare

  return 0;
}
```

Iterators:

std::vector::begin()
std::vector::end()

Capacity:

size()	size_type size() const
empty()	bool empty() const

Modifiers:

insert()	pair<iterator,bool> insert (const value_type& val);
erase()	iterator erase (const_iterator position)
	size_type erase (const value_type &val)
	iterator erase(const_iterator first, const_iterator last)

Operations:

find()	iterator find (const value_type& val);

Unordered_set

Description:

Unordered sets are containers that store unique elements in no particular order, and which allow for fast retrieval of individual elements based on their value.

In an unordered_set, the value of an element is at the same time its key, that identifies it uniquely. Keys are immutable, therefore, the elements in an unordered_set cannot be modified once in the container - they can be inserted and removed, though.

Internally, the elements in the unordered_set are not sorted in any particular order, but organized into buckets depending on their hash values to allow for 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, although they are generally less efficient for range iteration through a subset of their elements.

Iterators in the container are at least forward iterators.

Constructor:

empty (1)	`explicit unordered_set ( size_type n = /* see below */,`
	`const hasher& hf = hasher(),`
	`const key_equal& eql = key_equal(),`
	`const allocator_type& alloc = allocator_type() );`
	`explicit unordered_set ( const allocator_type& alloc );`

range (2)	`template <class InputIterator>`
	`unordered_set ( InputIterator first, InputIterator last,`
	`size_type n = /* see below */,`
	`const hasher& hf = hasher(),`
	`const key_equal& eql = key_equal(),`
	`const allocator_type& alloc = allocator_type() );`

copy (3)	`unordered_set ( const unordered_set& ust );` `unordered_set ( const unordered_set& ust, const allocator_type& alloc );`
move (4)	`unordered_set ( unordered_set&& ust );` `unordered_set ( unordered_set&& ust, const allocator_type& alloc );`

initializer list (5)	`unordered_set ( initializer_list<value_type> il,`
	`size_type n = /* see below */,`
	`const hasher& hf = hasher(),`
	`const key_equal& eql = key_equal(),`
	`const allocator_type& alloc = allocator_type() );`

Constructor Example:

```
// constructing unordered_sets
#include <iostream>
#include <string>
#include <unordered_set>

template<class T>
T cmerge (T a, T b) { T t(a); t.insert(b.begin(),b.end()); return t; }

int main ()
{
  std::unordered_set<std::string> first;                                // empty
  std::unordered_set<std::string> second ( {"red","green","blue"} );    // init list
  std::unordered_set<std::string> third ( {"orange","pink","yellow"} ); // init list
  std::unordered_set<std::string> fourth ( second );                    // copy
  std::unordered_set<std::string> fifth ( cmerge(third,fourth) );       // move
  std::unordered_set<std::string> sixth ( fifth.begin(), fifth.end() ); // range

  std::cout << "sixth contains:";
  for (const std::string& x: sixth) std::cout << " " << x;
  std::cout << std::endl;

  return 0;
}
```

Iterators:

std::vector::begin()
std::vector::end()

Capacity:

size()
empty()

Modifiers:

insert()
erase()

Operations:

find()

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