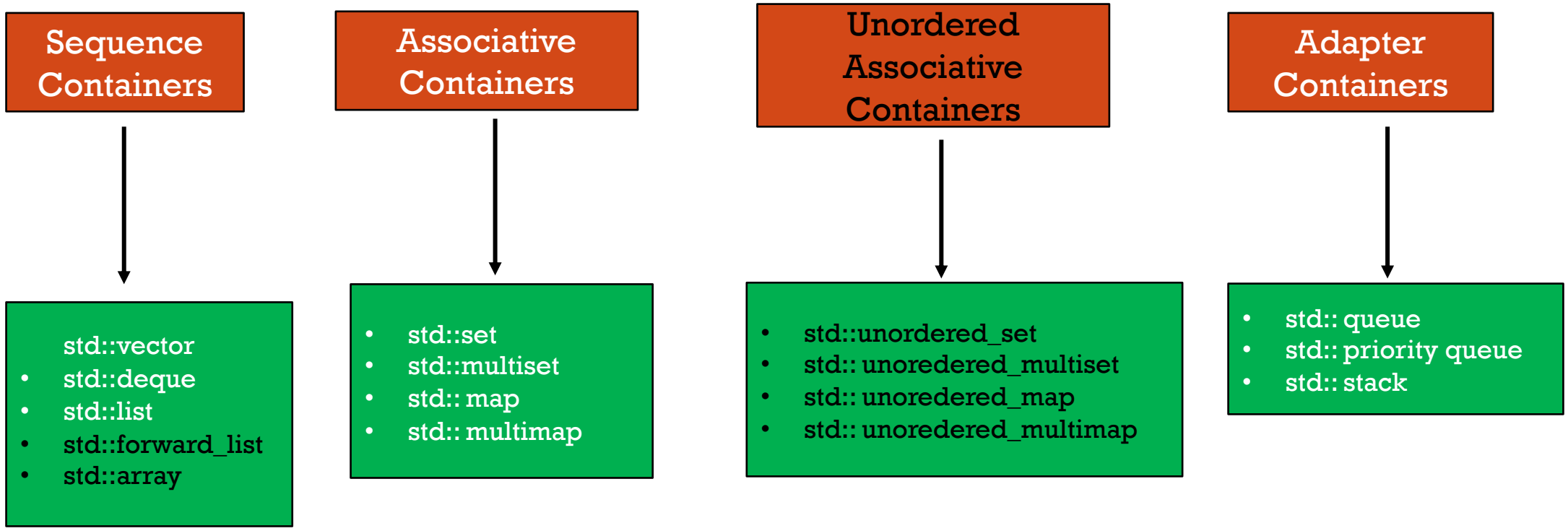
STL CONTAINERS

Sequence Containers

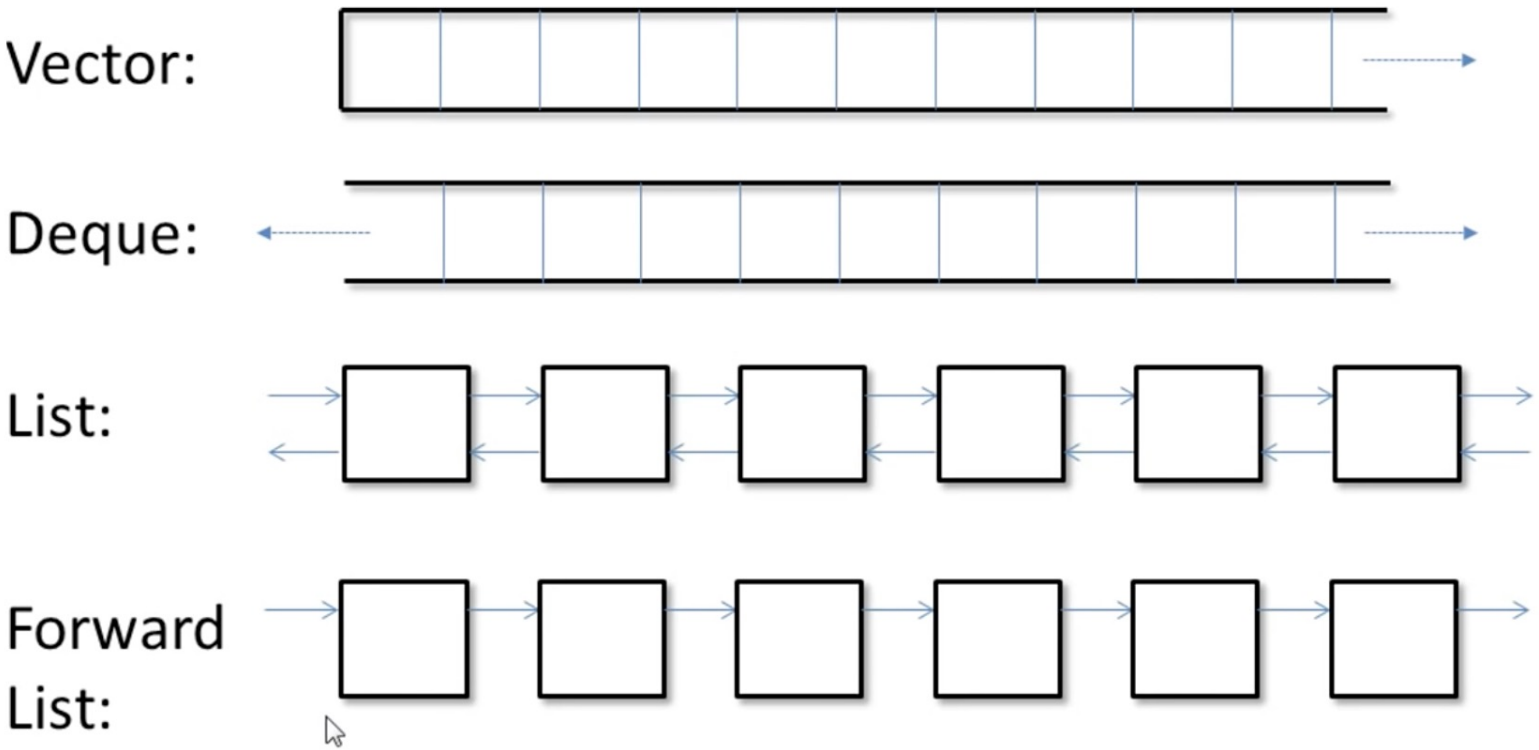
Ordered/Unordered Associative Container

Container Adaptors

STL CONTAINERS



SEQUENCE CONTAINERS



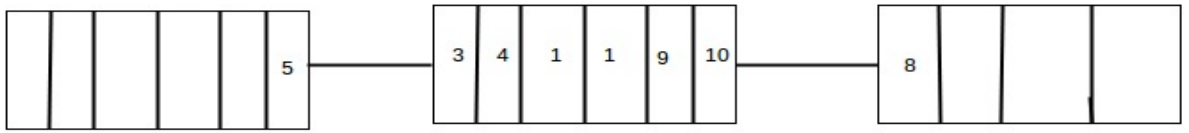
Vector:

Dynamic array which grows in one direction.

Properties:

- **O(1)** : push_back(), myVec.erase(myVec.end() -1);
- **O(n)** : slow insert/remove at the beginning or in the middle :
- **O(n)** : slow search

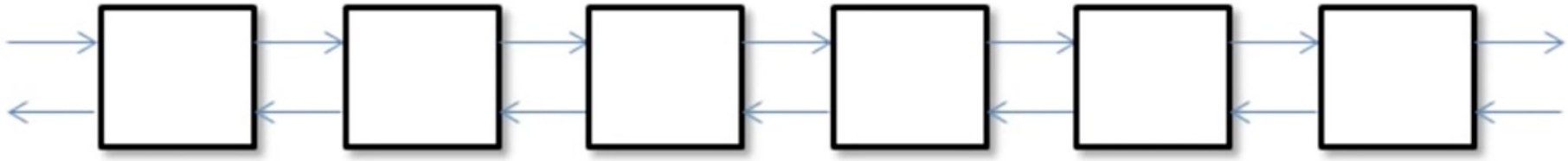
Internally implemented as a collection of memory blocks (linked list of vectors)



deque memory blocks

- **Properties:**
  - **O(1)**: push_back(), push_front(), pop_back(), pop_front()
  - **O(1)**: random access via operator[i]
  - **O(n/2)**: worst case insertion
- **Drawbacks:**
  - std::deque implementations use a chain of chunks to implement deque but use relatively small chunk sizes.
  - Multiple new allocations.
  - Iterating the container requires jumping through a lot of disjoint memory blocks.
- **Usage:**
  - a data structure suitable for a ***stack*** or a ***queue*****,** with fast push/pop and random access operations
  - a vector with half the shifts during an insertion/deletion, and no doubling of memory or reallocations during push_back().

List:

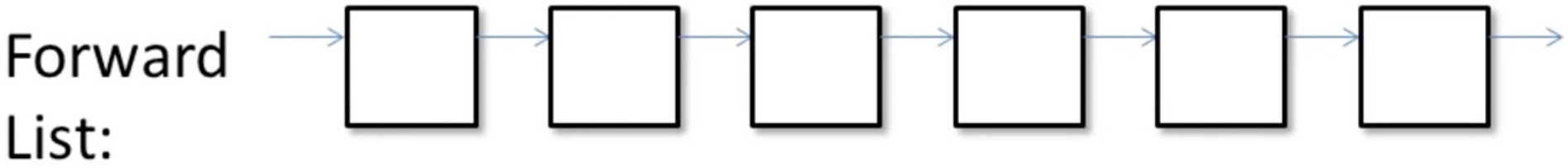


```
std::list<int> mylist =  {5, 2, 9};
mylist.push_back(6);    // mylist : {5, 2, 9, 6};
mylist.push_front(4);   // mylist : {4, 5, 2, 9, 6};

std::list<int>::iterator itr = std::find(mylist.begin(), mylist.end(), 2); // itr->2, O(n)
mylist.insert(itr, 8); // mylist: {5, 8, 2, 9, 6},  O(1)
++itr;                 // itr->9
mylist.erase(itr);     // mylist: {4, 8, 5, 2, 6}, O(1)

std::list<int> mylist2 = { 10, 20, 30, 40, 50 };

// Transfers elements from one list to another. O(1)
mylist.splice(mylist.begin(), mylist2); // list1:  10 20 30 40 50 4 5 8 2 6

std::cout << "list: " << mylist << "\n";
```

Properties:

- O(1) fast insert/remove at any place
- no random access operator – operator[]
- O(n) slow search

STD::FORWARD_LIST (SINCE C++11)



- **Similar to std::list**, std::forward_list is a container that supports fast insertion and removal of elements from anywhere in the container.
- **Compared to std::list**, this container **provides more space efficient storage** when bidirectional iteration is not needed.

STD::ARRAY (SINCE C++11)

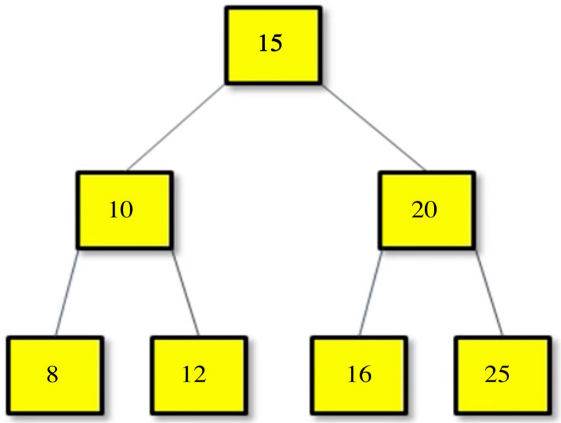
- Acts as a thin layer of C-style arrays : std::array is a container that encapsulates fixed size arrays.
- It does not loose the information of its length when decayed to a pointer.

```
void printArray(const std::array<int,3> &n)
{
    std::cout << "length: " << n.size() << std::endl;

    for (int j = 0; j < n.size(); j++ )
    {
        std::cout << "n[" << j << "] = " << n[j] << std::endl;
    }
}

int main(int argc, const char * argv[])
{
    int a1[3] = {3, 4, 5};
    std::array<int, 3> a2 = {3,4,5};
    std::array<int, 3> a3;

    a2.begin();
    a2.end();
    a2.size();
    a2.swap(a2);

    std::array<int, 4> b = {3,4,5};

    printArray(b);      No matching function for call to 'printArray'
    printArray(a2);
}
```

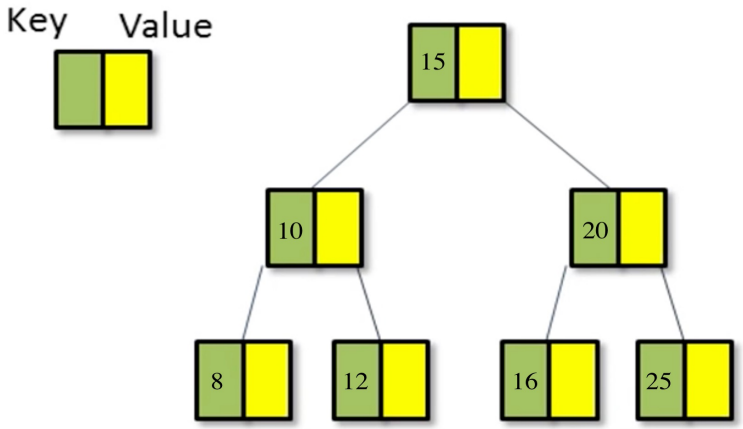
ASSOCIATIVE CONTAINERS (SET, MULTISET, MAP, MULTIMAP)

- Typically implemented as a **self-balancing binary search tree.**
- **Always sorted**, default creteria is **<**
- No push_back(), push_front()
- **Fast searching, guaranteed O(logn)**
- **Slow traversing** (compared to vector & deque).

Set or Multiset:

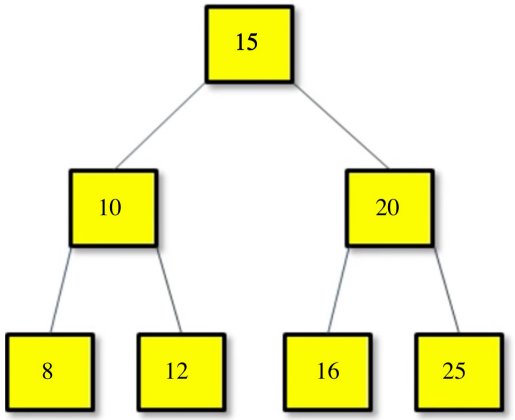


Map or Multimap:



STD::SET & STD::MULTISET

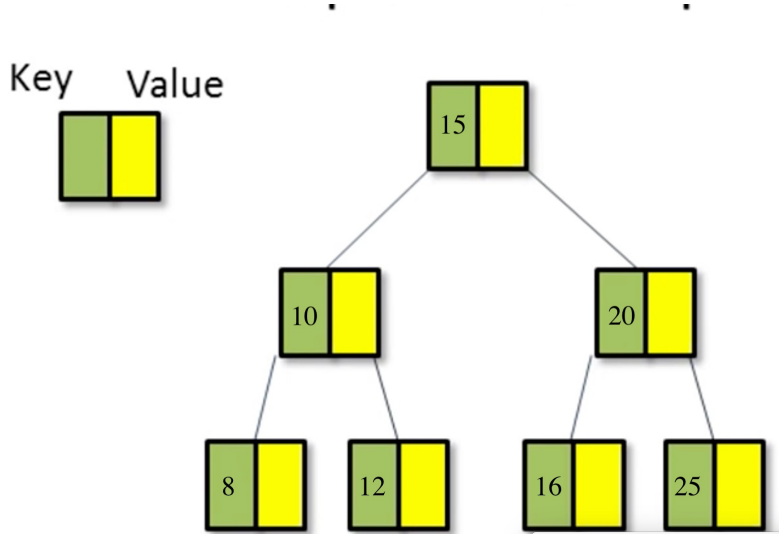
```cpp
std::set<int> myset;

myset.insert(3); // myset: {3}
myset.insert(1); // myset: {1, 3}
myset.insert(7); // myset: {1, 3, 7}, O(log(n))

std::set<int>::iterator it;
it = myset.find(7);  // O(log(n)), it points to 7
// Sequence containers don't even have a find() member function


std::pair<std::set<int>::iterator, bool> ret;
ret = myset.insert(3);  // no new element inserted
if(ret.second == false)
    it=ret.first;       // "it" now points to element 3

myset.insert(it, 9);    // myset: {1, 3, 7, 9}, O(log(n)) => O(1)

myset.erase(it);        // myset: {1, 7, 9}, O(1)

myset.erase(7)_         // myset: {1, 9}, O(log(n))
```



- The value of the elements cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.
- Sorting is done using the key comparison function Compare.
- **Search, insertion, and removal** operations have **O(log(n)).**
- No random access, no [] operator.

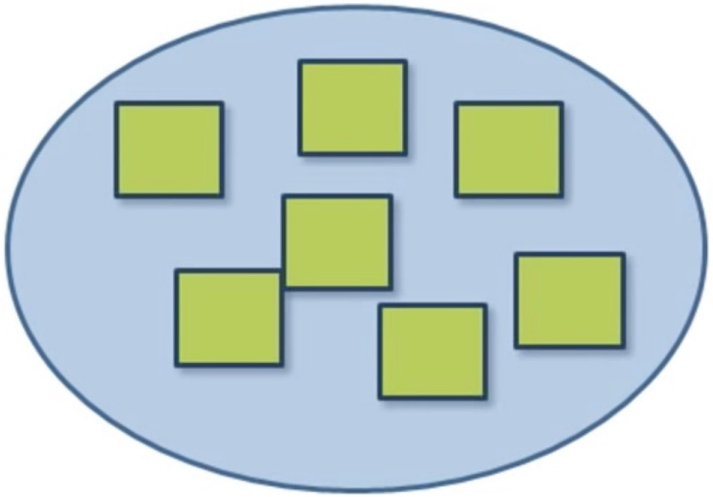
STD::MAP & STD::MULTIMAP

```cpp
std::map<char, int> mymap;
mymap.insert(std::pair<char, int>('a', 100));
mymap.insert(std::make_pair('z', 200));

std::map<char, int>::iterator it = mymap.begin();
mymap.insert(it, std::make_pair('c', 3)); // it is a hit O(logn) => O(1)

it = mymap.find('z');  // O(log(n))

std::multimap<char, int> mymultimap;

// map/multimap
// - keys cannot be modified
// type of *it: pair<const char, int>
(*it).first = 'd'; // ERROR
```

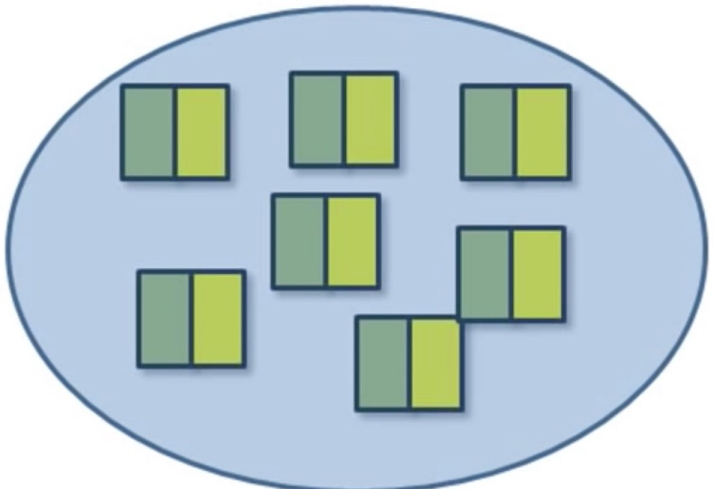


UNORDERED CONTAINERS. (SINCE C++11)

Unordered
Set or Multiset:

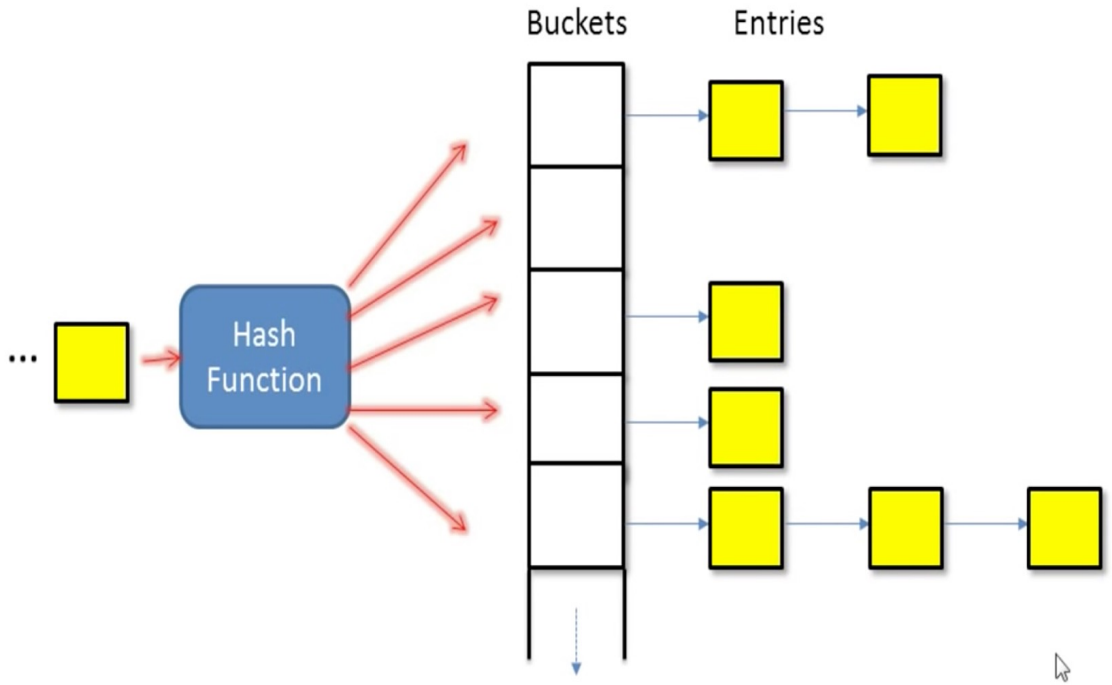


Unordered
Map or Multimap:

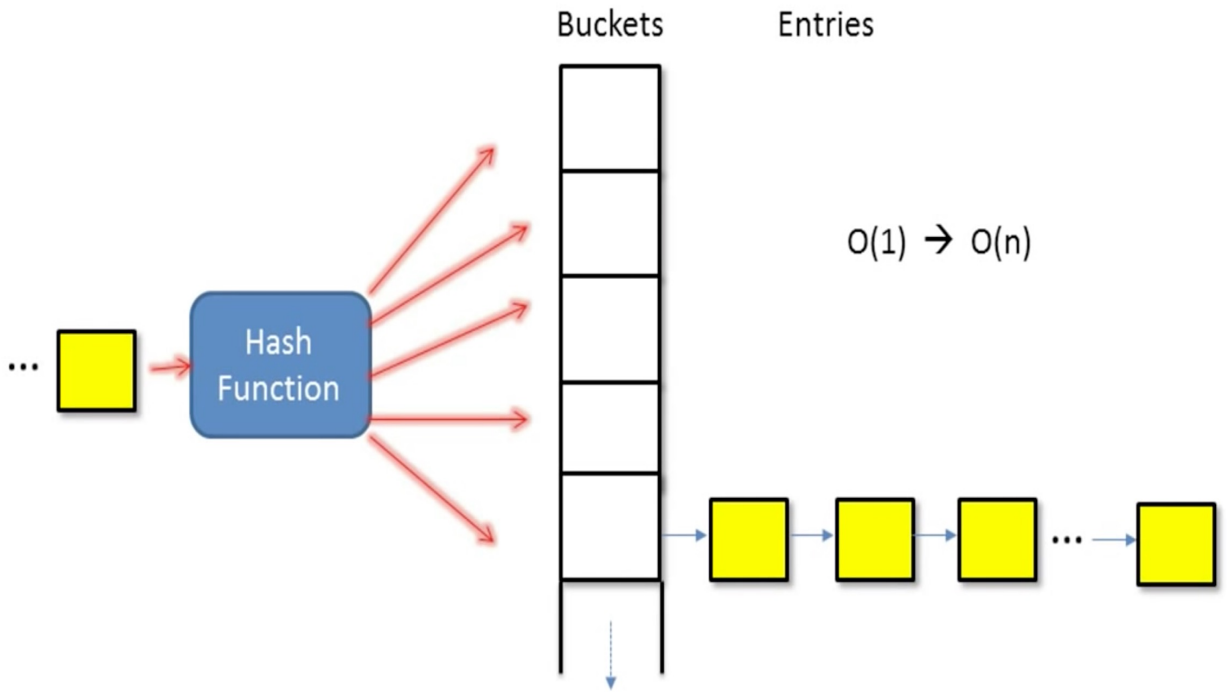


IMPLEMENTATION OF UNORDERED CONTAINERS

```
/*
 * unordered set
 */
  unordered_set<string> myset = { "red","green","blue" };
  unordered_set<string>::const_iterator itr = myset.find ("green"); // O(1)
  if (itr != myset.end())    // Important check
     cout << *itr << endl;
  myset.insert("yellow");  // O(1)

  vector<string> vec = {"purple", "pink"};
  myset.insert(vec.begin(), vec.end());

// Hash table specific APIs:
  cout << "load_factor = " << myset.load_factor() << endl;
  string x = "red";
  cout << x << " is in bucket #" << myset.bucket(x) << endl;
  cout << "Total bucket #" << myset.bucket_count() << endl;
```



DEGRADED UNORDERED CONTAINERS



UNORDERED CONTAINERS

Properties:

1. **Fastest search/insert** at any place **O(1)**

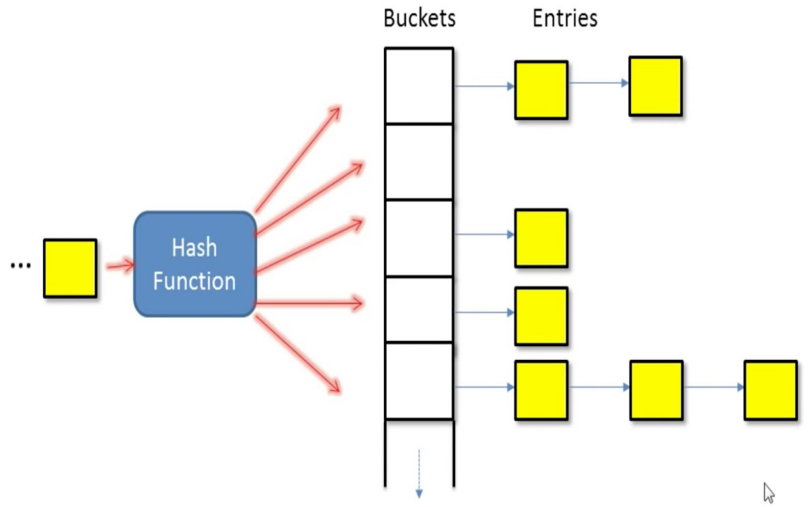
   Associative Cointainer takes O(log(n))

   Vector, deque takes O(n)

   List takes O(1) to insert, O(n) to search

2. Unordered set/multiset: element value is **read only**

Unordered set/multiset: element value is **read only**



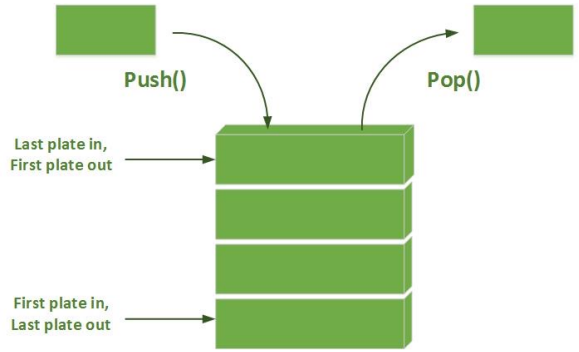
CONTAINER ADAPTORS (SINCE C++11)

Each of these "container adaptors" is constructed by taking one of the sequential containers (**vector, deque, list**) and "adapting" (**restricting/modifying**) its interface to provide the desired behavior.

- **Stack :** Adapts a sequence container to provide strict last-in, first-out (LIFO) behavior.
- **Queue:** Adapts a sequence container to provide strict first-in, first-out (FIFO) behavior.
- **Priority queue:** Adapts the vector container to maintain items in a sorted order.

STD::STACK (LIFO)

The class template **acts as a wrapper** to the underlying container - only a specific set of functions is provided.



std::stack

Defined in header <stack>

```
template<
    class T,
    class Container = std::deque<T>
> class stack;
```

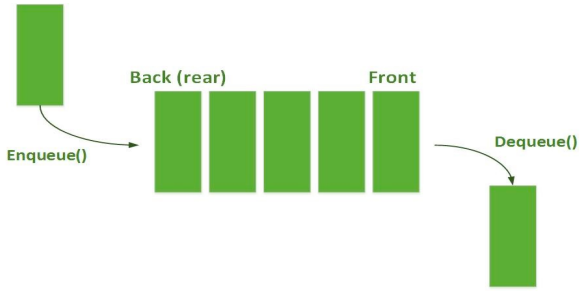
Container - The type of the underlying container to use to store the elements. The container must satisfy the requirements of *SequenceContainer*. Additionally, it must provide the following functions with the usual semantics:

- back()
- push_back()
- pop_back()

The standard containers std::vector, std::deque and std::list satisfy these requirements. By default, if no container class is specified for a particular stack class instantiation, the standard container std::deque is used.

STD::QUEUE. (FIFO)

- The class template **acts as a wrapper** to the underlying container - only a specific set of functions is provided.



std::queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::deque<T>
> class queue;
```

Template parameters

T - The type of the stored elements. The behavior is undefined if T is not the same type as `Container::value_type`. (since C++17)

Container - The type of the underlying container to use to store the elements. The container must satisfy the requirements of *SequenceContainer*. Additionally, it must provide the following functions with the usual semantics:

- `back()`
- `front()`
- `push_back()`
- `pop_front()`

The standard containers `std::deque` and `std::list` satisfy these requirements.

STD::PRIORITY_QUEUE

- A priority queue is a **container adaptor** that **provides constant time lookup of the largest (by default) element,** at the expense of logarithmic insertion and extraction.

std::priority_queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
> class priority_queue;
```

Container - The type of the underlying container to use to store the elements. The container must satisfy the requirements of *SequenceContainer*, and its iterators must satisfy the requirements of *RandomAccessIterator*. Additionally, it must provide the following functions with the usual semantics:

- `front()`
- `push_back()`
- `pop_back()`

The standard containers `std::vector` and `std::deque` satisfy these requirements.

Properties:

O(1) : getMinHeap, getMaxHeap()

O(log n) insert(), delete();

STD::PRIORITY_QUEUE AS BINARY HEAP

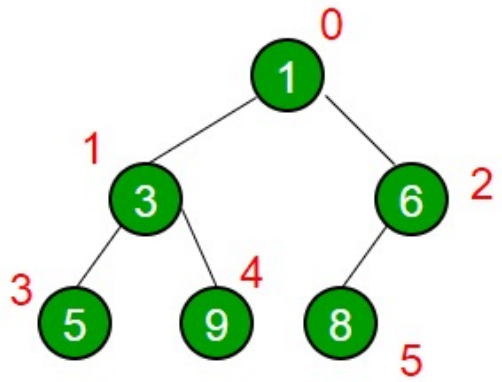
A Binary Heap is a Binary Tree with following properties.
1) It's a **complete tree** (makes them suitable to be stored in an array.)
2) A Binary Heap is either **Min Heap or Max Heap**. The same property must be **recursively true** for all nodes in Binary Tree.

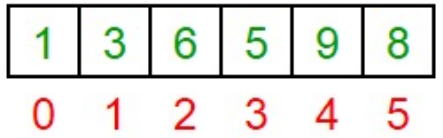
How is Binary Heap represented?

A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as an array.

- The root element will be at Arr[0].
- Below table shows indexes of other nodes for the i^{th} node, i.e., Arr[i]:

Arr[(i-1)/2]	Returns the parent node
Arr[(2*i)+1]	Returns the left child node
Arr[(2*i)+2]	Returns the right child node





Array representation is **Level Order**

STL CONTAINERS UNDERLING IMPLEMENTATIONS

Sequence Containers

- std::vector
- std::deque
- std::list
- std::forward_list
- std::array

Associative Containers

- std::set
- std::multiset
- std:: map
- std:: multimap

Unordered Associative Containers

- std::unordered_set
- std:: unoredered_multiset
- std:: unoredered_map
- std:: unoredered_multimap

Adapter Containers

- std:: stack
- std::queue
- std:: priority queue