Data Structures

A data structure is a way to store data in the memory of a computer. It is important to choose an appropriate data structure for a problem,because each data structure has it own advantages and disadvantages. This chapter introduces the most important data structures in the C++ standard library. It is a good idea to use the standard library whenever possible because it will save a lot time.

Dynamic Arrays

A dynamic array is an array whose size can be changed during the execution of the program. The most popular dynamic array in C++ is the vector structure, which can be used almost like an ordinary array.

```
vector<int>v;
v.push_back(3); //[3]
v.push_back(2); //[3,2]
v.push_back(5); //[3,2,5]
```

After this, the elements can be accessed like in an ordinary array:

```
cout<<v[0]<<"\n"; //3
cout<<v[1]<<"\n"; //2
cout<<v[2]<<"\n"; //5
```

Traversal

```
for(int i=0;i<v.size();i++){
    cout<<v[i]<<"\n";
}
```

```
for(auto x : v){
    cout<<x<<"\n";
}
```

To get Last Element

```
cout<<v.back()<<"\n";
```

To remove Last Element

```
v.pop_back();
```

Different Ways to Create Vector

```
vector<int> v(10); // Vector of size 10,initial value 0
```

```
vector<int> v={2,4,5,1}; //Vector with 5 elements
```

```
vector<int>v(10,5); //size = 10 ,initial value =5
```

> The internal implementation of vector uses an ordinary array. If the size of the vector increases and array becomes full or too small , then a new array is allocated and all the elements are moved to the new array. However this doesn't happen often so the average time complexity of push_back is O(1).

Set

A set is a data structure that maintains a collection of elements. The C++ standard library contains two set implementations.

- set
- unordered_set

	SET or ORDERED SET	**UNORDERED SET**
Ordering	Increasing Order	No Ordering
Implementation	Self Balancing BST like Red Black Tree	Hash Table
Search Time	Log(n)	O(1) - -> Average , O(n) - -> Worst Case
Insertion Time	Log(n) + Rebalance	Same as search
Deletion Time	Log(n) + Rebalance	Same as search

Rebalance has a average of O(1) & worst case O(n) time complexity.

Set or Ordered Set is better because BSTs, all operations are guaranteed to work in O(Logn) time. But with Hashing(Unordered Set), Θ(1) is average time and some particular operations may be costly, especially when table resizing happens.

```
set<int>s;
unordered_set<int>S;
s.insert(1);
S.insert(1);
cout<<s.count(1)<<"\n"; // 1
cout<<S.count(1)<<"\n"; // 1
cout<<s.count(2)<<"\n"; // 0
cout<<S.count(2)<<"\n"; // 0
//Remove element which has given values
s.erase(1);
S.erase(1);
s.insert(2);
S.insert(2);
//To Traverse
for (itr = s.begin();itr != s.end(); ++itr){
    cout << *itr << ",";
}

//For removal of all element less than x in ordered set
```

```
s2.erase(s2.begin(), s2.find(x));
```

A set can be used mostly like vector, but it is not possible to access the element using [] notation. The following code creates the set, prints the number of elements in it and then iterates through the element.

```
set<int>s={2,3,4,5};
cout<<s.size()<<"\n";
for(auto x:s){
    cout<<x<<"\n";
}
```

An important property of set is that all their element are distinct. Thus the function count always return either 0 or 1.

```
set<int>s;
s.insert(1);
s.insert(1);
s.insert(1);
cout<<s.count(1)<<"\n"; //1
```

Multi-Set

C++ also contains the structure multiset & unordered_multiset thar work similar to set with an additional feature that it can contain duplicate element as well.

```
multiset<int>s;
s.insert(1);
s.insert(1);
s.insert(1);
cout<<s.count(1)<<"\n"; //3
```

The function erase removes all instances of an element from a multiset:

```
s.erase(1);
cout<<s.count(1)<<"\n"; //0
```

Often, only one instance should be removed, which can be done as follows:

```
s.erase(s.find(1));
cout<<s.count(1)<<"\n"; //2
```

Map

A map is a data structure that has key-value pairs. The keys in an ordinary array are always the consecutive integers 0,1...n-1 where n is the size of array but in case of map it is not as such . We can have any of the primary data types or derived data types as key or value data types in a map. The C++ standard library contains two map implementation:

- map : It is based on a balanced binary tree and accessing elements takes O(logn) time.
- unordered_map : It uses hashing and accessing elements takes O(1) time on average.

```
map<string,int>m1; //here key is string and value is int
map<string,string>m2; //here key is string and value is string
map<int,int>m3; //here key is int and value is int
```

The following code creates a map where the key are strings and the values are integers:

```
map<string,int>m;
m["abc"]=4;
m["def"]=5;
m["ghi"]=6;
cout<<m["abc"]<<"\n"; //4
```

If the value of a key is requested but the map doesn't contain it, the key is automatically added to the map with default value 0.

```
map<string,int>m;
cout<<m["Saurav"]<<"\n"; //0
```

The function count checks if a key exist in map :

```
if(m.count("abc")){
    // key exist
}
```

The following code prints all the keys and values in a map:

```
for(auto x : m){
    cout << x.first << " " << x.second <<"\n";
}
```

Bitset

A bitset is an array whose each value is either 0 or 1.

```
bitset<10> s;
s[1] = 1;
s[2] = 1;
s[3] = 1;
cout<<s[3]<<"\n"; //1
cout<<s[5]<<"\n"; //0
cout<<s.count()<<endl; //Returns count of 1's
```

The benefit of using bitsets is that they require less memory than ordinary arrays, because each element in a bitset only uses one bit of memory. For example if there are n bits are stored in an int array, 32n bits of memory will be used, but a corresponding bitset only require n bits of memory. In addition the values of bitset can be efficiently manipulated using bit operators, which makes it possible to optimize the algorithms using bit sets.

```
bitset<10> s(string("0010011010")); //From right to left
bitset<10> s1(string("1000001011"));
cout<<s[4]<<"\n";//1
cout<<s[5]<<"\n";//0
cout<<s.count()<<"\n";//4
cout<<(s&s1)<<"\n";//0000001010
```

Iterators and Ranges

An iterator is a variable that points to an element in a data structure. The often used iterators begin and end define a range that contains all elements in a data structure. The iterators begin points to the first element in the data structure, and the iterator end points to the position after the last element.

The situation looks as follows :

```
{ 3 ,  4,  6,  8,  12,  13,  14,  17 }
  |                                 |
s.begin()                        s.end()
```

Working with ranges

Iterators are used in C++ standard library functions that are given a range of element in a data structure. Usually, we want to process all elements in a data structure, so the iterators begin and end are given for the function.

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

These functions can also be used with an ordinary array. In that case, the functions are given pointers of the array instead of iterators :

```
sort(a,a+n);
reverse(a,a+n);
random_shuffle(a,a+n);
```

Set Iterators

Iterators are often used to access elements of a set. The following code creates an iterator it that points to the smallest element in a set:

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

A shorter way to write the code is as follows:

```
auto it = s.begin();
```

The element to which an iterator points can be accessed using the * symbol. For example, the following code prints the first element in the set:

```
auto it = s.begin();
cout<< *it<<"\n";
```

Iterators can be moved using the operators ++ (forward) and -- (backward), meaning that the iterator moves to the next or previous element in the set.

```
for(auto it=s.begin(); it!=s.end(); it++){
    cout<< *it<< "\n";
}
```

The following code prints the largest element in the set.

```
auto it = s.end(); it--;
cout<< *it<<"\n";
```

The function find(x) returns an iterator that points to an element whose values is x. However, if the set doesn't contain x, the iterator will be an end.

```
auto it = s.find(x);
if(it==s.end()){
    //x is not found
}
```

The function *lower_bound(x)* returns an iterator to the smallest element in the set whose values is at least x, and the function *upper_bound(x)* returns an iterator to the smallest element in the set whose values is larger than x. **These function aren't supported by unordered_set structure as it doesn't maintain the order of element.**

Map Iterators

```
map<string,int>m;
map<string,int>::iterator it=m.begin();
```

```
auto it=m.begin();
```

```
for(auto it=s.begin(); it!=s.end(); it++){
    cout<<it->first<<"\t"<<it->second<< "\n";
}
```

```
for(auto x:m){
        cout<<x.first<<"\t"<<x.second<<endl;
    }
```

Map also supports the above given functions.

Deque

A deque is a double ended queue where insertion and deletion occurs at both end of the array. Similar to a vector it provides the functions push_back() & pop_back() , but also provide the facility to insert push_front() & delete at front pop_front() which aren't available in vector. Unlike vectors, contiguous storage allocation may not be guaranteed.

```
deque<int>d;
d.push_back(5); //[5]
d.push_back(6); //[5,6]
d.push_front(2); //[2,5,6]
d.push_front(1); //[1,2,5,6]
d.pop_back(); //[1,2,5]
d.pop_front(); //[2,5]
```

The internal implementation of a deque is more complex than that of vector, and for this reason, a deque is slower than a vector. Still, both adding and removing element take **O(1)** time on average at both ends.

Stack

A stack is a data structure that provides two O(1) time operations:

- Adding an element at the top.
- Removing an element from the top.

  It is only possible to access the top element of a stack.

The following code shows how a stack can be used:

```
stack<int>s;
s.push(3);
s.push(2);
s.push(1);
cout<<s.top(); // 1
s.pop();
cout<<s.top(); // 2
```

Queue

A queue is a data structure that provides two O(1) time operations:

- Adding an element to the end
- Removing the first element

  It is only possible to access the first and last element of the queue.

```
queue<int>q;
q.push(1); //[1]
q.push(2); //[1,2]
q.push(3); //[1,2,3]
cout<<q.front(); // 1
q.pop();
cout<<q.front(); // 2
cout<<q.back(); //3
```

Priority Queue

A priority queue maintains a set of elements. The supported operations are insertion , retrieval and removal of either the minimum or maximum element. Insertion and removal takes O(logn) time and retrieval takes O(1) time.

> While an ordered set efficiently supports all the operations of priority queue, the benefit of priority queue is that it has smaller constant factors. A priority queue is usually implemented using heap but a ordered set uses balanced binary tree.

By default, the elements in C++ priority queue are sorted in decreasing order.

```
priority_queue<int>q;
q.push(3); //[3]
q.push(4); //[4,3]
q.push(5); //[5,4,3]
cout<<q.top()<<"\n"; // 5
q.pop();
cout<<q.top()<<"\n"; // 4
q.pop();
cout<<q.top()<<"\n"; // 3
```

If we want to create a priority queue the sorts the element in increasing order.

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

List

Normally, when we say a List, we talk about doubly linked list. For implementing a singly linked list, we use forward list. Lists are sequence containers that allow non-contiguous memory allocation. As compared to vector, list has slow traversal, but once a position has been found, insertion and deletion are quick.

```
//To declare single linked list
forward_list<int> singlyLinkedList;
// To declare doubly linked list
list<int> doublyLinkedList;
//To Insert At Last
doublyLinkedList.push_back(x);
//To Insert At Start
doublyLinkedList.push_front(x);
//To Delete At Last
doublyLinkedList.pop_back(x);
//To Delete At Start
doublyLinkedList.pop_front(x);
//To Reverse a Linked List
doublyLinkedList.reverse();
//To Sort a Linked List
doublyLinkedList.sort()

//To iterate list
for(it = doublyLinkedList.begin(); it != doublyLinkedList.end(); ++it)
        cout << '\t' << *it;
```