1 Binary Heap

Defined in header <queue>
Implementation of a **max** heap
Default container: vector

To insert an element: `q.push(x)` / `q.emplace(x)`
To get the largest element: `q.top()`
To remove the largest element: `q.pop()`
Time complexity: *O(1)* for `top()` and *O(log n)* for `push()` / `pop()`

C++ Implementation

```
priority_queue<int> pq; // Create an integer max heap
pq.push(1); // Insertion
pq.push(2);
pq.push(3);
cout << pq.size() << endl; // Size, output 3
cout << pq.top() << endl; // Max, output 3
pq.pop(); // Delete max
cout << pq.size() << endl; // output 2

priority_queue<int, vector<int>, greater<int>> pq2; // Create an integer min heap

struct my { int val; int rank; };
struct mycmp
{
    bool operator() (my const& A, my const& B) { return A.val > B.val; }
    // if A.val > B.val, then A will be “at the back of” B
};
priority_queue<my, vector<my>, mycmp> pq3;
```

2 Binary Search Tree

To declare an empty int set: `set<int> s`
To insert an element *x*: `s.insert(x)` / `s.emplace(x)`
To remove **elements** that are equal to *x*: `s.erase(x)`
To remove the element at *it*: `s.erase(it)`
To find *x*: `s.find(x)`
To get the lower bound of *x*: `s.lower_bound(x)`
(`lower_bound(s.begin(), s.end(), x)` compiles but is $O(n)$)
To get the upper bound of *x*: `s.upper_bound(x)`
(`upper_bound(s.begin(), s.end(), x)` compiles but is $O(n)$)

C++ Implementation

```
set<int> s; // Implement a set
s.insert(4); // Insertion
s.insert(9);
s.insert(6);
cout << s.size() << endl; // Size, output: 3
for (auto str : s)
    cout << str << endl; // Iteration in ascending order, output: 4 6 9
if (s.find(4) != s.end()) // Check if 4 is in the BST
    cout << "4 is in the BST" << endl;
s.erase(6); // Erase elements 6
s.erase(s.begin()); // Erase the minimum elements
cout << *s.begin() << endl; // Minimum element
cout << *s.rbegin() << endl; // Maximum element
auto it = s.lower_bound(2); // Binary search
if (it != s.end()) cout << *it << endl; // Output the content after binary search
```

Defined in header <map>
Associative containers
map contains **key-value pairs** with **unique** keys
multimap contains a sorted list of **key-value pairs**
The value can be accessed by `operator[]` in map
Time complexity: *O(log n)* for each operation

C++ Implementation

```
map<string> mp; // Implement a set
mp["Hi"] = 2;
mp["BSTC"] = 24212580;
auto it = mp.find("Hi");
if (it != mp.end()) cout << it->first << ' ' << it->second << endl;
// Iterators ~= pointers
else cout << "None" << endl;
for (auto [key, val]: mp) cout << key << ' ' << val << endl;
// Output all pair of keys and values
```

3 Disjoint sets union-find

C++ Implementation

```
void init() { for (int i = 1; i <= n; ++i) p[i] = i; } // Initialization

int find(int u) // find(u) with path compression
{
    return p[u] == u ? u : p[u] = find(p[u]);
}

void union(int u, int v) // union(u, v)
{
    int pu = find(u);
    int pv = find(v);
    p[pu] = pv;
}
```