ChatGPT

Comprehensive C++ STL and Algorithms Guide

This guide covers the core C++ Standard Template Library (STL) containers and algorithms essential for competitive programming and coding interviews. It includes container usage, common algorithms, graph/tree representations, heap usage, CRUD examples, tips, and tricky cases. Code examples are well-commented in English and Hindi for clarity.

1. Core STL Containers

1.1 `std::vector`

A **vector** is a dynamic array that provides random access to elements and amortized constant-time insertion at the end [1]. Key points:

- **Create:** `vector<int> v;` or `vector<int> v = {1,2,3};`.
- **Insert:** `v.push_back(x);` (amortized O(1) time) [1].
- **Access:** `v[i]` or `v.at(i)` (O(1) time) [2].
- **Update:** Assign via index, e.g. `v[0] = 5;`.
- **Delete:** `v.pop_back();` removes last element in O(1) time [3], or `v.erase(it);` to remove at an iterator (O(n) if not at end).
- **Traverse:** Use range-for or iterators: `for(int x : v) cout << x;`.

Example: Inserting, updating, deleting in a vector.

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    vector<int> v;
    v.reserve(5);            // Reserve capacity to avoid reallocations [4]
    v.push_back(3);          // Add 3 at end (वेक्टर के अंत में 3 डालता है)
    v.push_back(1);
    v.push_back(4);
    cout << "Elements: ";
    for(int x: v) cout << x << " ";  // Traverse and print (Traversal O(n) [1])
    cout << endl;
    // Update
    v[1] = 2;               // Update second element to 2 (अद्यतन करता है)
    cout << "After update: ";
    for(int x: v) cout << x << " ";
    cout << endl;
    // Delete
    v.pop_back();           // Remove last element (Last element हटाता है)
```

```
    cout << "After pop_back: ";
    for(int x: v) cout << x << " ";
    cout << endl;
    return 0;
}
```

Explanation: We created a vector `v`, inserted elements with `push_back`, updated an element via index, and deleted the last element with `pop_back`. Random access and end operations are efficient [1].

1.2 `std::string`

`std::string` is the standard C++ class for text. It stores a sequence of characters and supports many operations [5]. Key operations:

- **Length:** `s.length()` or `s.size()` returns the number of characters.
- **Concatenate:** `s += "abc";` or `s.append("xyz");`.
- **Search:** `s.find("sub")` returns index or `string::npos` if not found.
- **Substring:** `s.substr(pos, len)` extracts part.
- **Delete:** `s.erase(pos, len)` removes characters.

Example: Basic string usage.

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    string s = "Hello";
    s += " World";                      // Append (जोड़ना)
    cout << s << endl;                  // Outputs: Hello World
    cout << "Length: " << s.size() << endl;   // Length (लंबाई)
    // Find substring
    size_t idx = s.find("World");
    if(idx != string::npos) {
        cout << "\"World\" found at index " << idx << endl;
    }
    // Replace
    s.replace(6, 5, "C++");             // Replace "World" with "C++" (स्थानापन्न)
    cout << "After replace: " << s << endl;
    return 0;
}
```

Explanation: We show concatenation, finding a substring, and replacing. The `std::string` class provides these built-ins [5].

1.3 `std::set` and `std::unordered_set`

A **set** stores **unique** elements in sorted order (ascending by default) [6]. Insertion, deletion, and search are logarithmic time. An **unordered_set** stores unique elements without order, using a hash table for average O(1) time operations [7] [8].

- **Create:** `set<int> s;` or `unordered_set<int> us;`.
- **Insert:** `s.insert(x); us.insert(x);` (set: O(log n), unordered_set: average O(1) [8] [9]).
- **Search:** `if(s.count(x))` or `us.find(x)!=us.end()`.
- **Delete:** `s.erase(x); us.erase(x);`.
- **Traverse:** Set iterates in sorted order: `for(int x : s) cout << x;`. Unordered set order is arbitrary.

Example: Using set and unordered_set with CRUD.

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    set<int> s = {3, 1, 4};
    unordered_set<int> us = {3, 1, 4};

    // Inserting elements
    s.insert(2);    // Inserts 2 (sorted order), duplicates ignored
    us.insert(2);   // Inserts 2 in hash set
    // Attempt duplicate
    s.insert(3);    // 3 already present; no effect
    us.insert(1);   // 1 already present; no effect

    // Traverse and print
    cout << "set elements: ";
    for(int x : s) cout << x << " ";       // Sorted: 1 2 3 4 (increasing)
    cout << "\nunordered_set elements: ";
    for(int x : us) cout << x << " ";      // Unordered order
    cout << endl;

    // Search
    if(us.count(4))
        cout << "4 is in us\n";

    // Delete
    s.erase(1);      // Remove 1 from set
    us.erase(3);     // Remove 3 from unordered_set

    cout << "After erase, set: ";
    for(int x: s) cout << x << " ";       // 2 3 4 (since 1 removed)
    cout << endl;
```

```
    return 0;
}
```

Explanation: We demonstrate `set` (sorted, O(log n) ops) and `unordered_set` (hash-based, average O(1) ops) [7] [8]. Inserting duplicates has no effect, and `erase` removes elements.

1.4 `std::map` and `std::unordered_map`

A **map** stores key–value pairs sorted by key, usually implemented as a red-black tree. All operations (insert/find/erase) are O(log n) [10] [11]. An **unordered_map** uses hashing for key/value pairs, offering average O(1) operations [12] [11].

- **Create:** `map<int,string> mp;` or `unordered_map<int,string> ump;`.
- **Insert/Update:** `mp[key] = val;` or `ump[key] = val;`. (Note: `[]` creates if absent, otherwise updates existing key.)
- **Find:** `auto it = mp.find(key);`
- **Delete:** `mp.erase(key); ump.erase(key);`.
- **Traverse:** `for(auto &p : mp) cout << p.first << ":" << p.second;`.

Example: Map and unordered_map usage.

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    map<int,string> mp;
    unordered_map<int,string> ump;

    // Insert/Update
    mp[2] = "two";     // Insert key 2
    mp[5] = "five";
    mp[2] = "dos";     // Update key 2 to "dos"
    ump[10] = "ten";   // Insert key 10
    ump[3] = "three";

    // Access and print
    cout << "map:\n";
    for(auto &p : mp) {
        cout << p.first << " -> " << p.second << "\n";
    }
    cout << "unordered_map:\n";
    for(auto &p : ump) {
        cout << p.first << " -> " << p.second << "\n";
    }

    // Search
    if(mp.count(5)) cout << "Key 5 found with value " << mp[5] << endl;
```

```
    // Delete
    mp.erase(2);    // Remove key 2
    ump.erase(3);   // Remove key 3

    return 0;
}
```

Explanation: We use `mp[key]` and `ump[key]` to insert or update. After insertion, we traverse and print all pairs. The map is sorted by key, the unordered_map is not sorted [12] [11]. Erasing keys removes them.

1.5 `std::stack`

A **stack** is a LIFO (last-in-first-out) container adapter. It allows push/pop at one end. Operations `push`, `pop`, `top` are O(1) [13]. Internally it can use a vector or deque, but only top element is accessible [14].

- **Create:** `stack<int> st;`.
- **Push:** `st.push(x);`.
- **Top:** `st.top()` gives the top element.
- **Pop:** `st.pop();`.
- **Empty/Size:** `st.empty()`, `st.size()`.

Example: Basic stack operations.

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    stack<int> st;
    st.push(10);   // Stack: [10]
    st.push(20);   // Stack: [10,20]
    st.push(30);   // Stack: [10,20,30]
    cout << "Top element: " << st.top() << endl; // 30
    st.pop();      // Remove 30
    cout << "After pop, new top: " << st.top() << endl; // 20
    // Empty the stack
    while(!st.empty()) {
        cout << "Popping " << st.top() << endl;
        st.pop();
    }
    return 0;
}
```

Explanation: Elements are added and removed only at the top (LIFO). We show `push`, `top`, and `pop`. All are constant-time operations [13].

1.6 `std::queue`

A **queue** is a FIFO (first-in-first-out) container adapter. It allows insertions at one end (back) and deletions at the other (front) in O(1) time [15] [16].

- **Create:** `queue<int> q;`.
- **Enqueue:** `q.push(x);`.
- **Front/Back:** `q.front()`, `q.back()`.
- **Dequeue:** `q.pop();`.
- **Empty/Size:** `q.empty()`, `q.size()`.

Example: Basic queue operations.

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    queue<int> q;
    // Enqueue elements
    q.push(1); // Queue: [1]
    q.push(2); // [1,2]
    q.push(3); // [1,2,3]
    cout << "Front: " << q.front() << ", Back: " << q.back() << endl; // 1,3
    q.pop(); // removes 1, queue is [2,3]
    cout << "After pop, Front: " << q.front() << endl; // 2
    // Empty the queue
    while(!q.empty()) {
        cout << "Dequeuing " << q.front() << endl;
        q.pop();
    }
    return 0;
}
```

Explanation: We push to the back and pop from the front. This follows FIFO order [16]. All basic operations (`push`, `pop`) are O(1) [15].

1.7 `std::deque`

A **deque** (double-ended queue) is a sequence container that allows insertion/removal at both front and back in O(1) amortized time [17] [18]. It also supports random access in O(1).

- **Create:** `deque<int> dq;`.
- **Push/Pop:** `dq.push_back(x); dq.push_front(y); dq.pop_back(); dq.pop_front();`.
- **Access:** `dq[i]` or `dq.at(i)`.
- **Traverse:** As with vector, but can push/pop at both ends.

Example: Using `deque` as a double-ended queue.

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    deque<int> dq = {3, 1, 4};
    dq.push_front(2);  // Deque: [2,3,1,4]
    dq.push_back(5);   // [2,3,1,4,5]
    cout << "Deque front: " << dq.front() << ", back: " << dq.back() << endl;
    // Insert in middle
    auto it = dq.begin() + 2;
    dq.insert(it, 7);  // Insert 7 at position 2
    cout << "After insert: ";
    for(int x: dq) cout << x << " ";
    cout << endl;
    dq.pop_front();    // Remove front (2)
    dq.pop_back();     // Remove back (5)
    cout << "After pops: ";
    for(int x: dq) cout << x << " ";
    cout << endl;
    return 0;
}
```

Explanation: We demonstrate `push_front`, `push_back`, `pop_front`, `pop_back`. Inserting at arbitrary position is O(n), but end operations are constant amortized [17] [18]. Deque maintains fast operations at both ends.

1.8 `std::priority_queue`

A **priority_queue** is a heap-based container adapter. By default it is a max-heap (top is the largest element) [19]. It supports `push`, `pop`, `top` in O(log n) time. To use as a min-heap, provide a comparator like `greater<int>`.

- **Create:** `priority_queue<int> pq;` (max-heap). For min-heap: `priority_queue<int, vector<int>, greater<int>> minpq;`.
- **Insert:** `pq.push(x);`.
- **Access:** `pq.top();` (largest for max-heap).
- **Remove:** `pq.pop();`.
- **Traverse:** Not directly; to view all elements, pop them off.

Example: Max-heap and min-heap usage.

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    // Max-heap (default)
    priority_queue<int> pq;
```

```
    pq.push(10);
    pq.push(5);
    pq.push(20);
    cout << "Max-heap top: " << pq.top() << endl; // 20
    pq.pop();
    cout << "After pop, top: " << pq.top() << endl; // 10

    // Min-heap using greater<int>
    priority_queue<int, vector<int>, greater<int>> minpq;
    minpq.push(10);
    minpq.push(5);
    minpq.push(20);
    cout << "Min-heap top: " << minpq.top() << endl; // 5
    minpq.pop();
    cout << "After pop, top: " << minpq.top() << endl; // 10

    return 0;
}
```

Explanation: The first queue is a max-heap (largest element first) [19]. The second uses `greater<int>` to make it a min-heap. All operations (`push`/`pop`) run in O(log n) time due to the underlying binary heap.

2. Common Algorithms (STL `<algorithm>`)

2.1 Sorting (`std::sort`)

`std::sort` sorts a range in ascending order by default. It typically runs in O(n log n) time. Example:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    vector<int> v = {4, 2, 5, 1, 3};
    sort(v.begin(), v.end()); // Sort in-place (ascending)
    for(int x: v) cout << x << " "; // 1 2 3 4 5
    cout << endl;
    // Custom comparator: descending
    sort(v.begin(), v.end(), greater<int>());
    for(int x: v) cout << x << " "; // 5 4 3 2 1
    cout << endl;
    return 0;
}
```

Explanation: Sorting is often required before binary search or for ordering results. Use `greater<>` for descending. Complexity is O(n log n) on average.

2.2 Binary Search (`std::binary_search`)

`std::binary_search` checks if a value exists in a sorted range (needs sorted data, else undefined) [20]. It returns a bool in O(log n). Example:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    vector<int> v = {1,3,6,8,9};
    int key = 8;
    if(binary_search(v.begin(), v.end(), key))
        cout << key << " is present\n";
    else
        cout << key << " is NOT present\n";
    // Note: v must be sorted for binary_search [20].
    return 0;
}
```

Explanation: `binary_search` requires the range be sorted; otherwise behavior is undefined [20]. It simply tells presence of the element.

2.3 `lower_bound` and `upper_bound`

These functions find boundaries in a sorted range (O(log n) time). `lower_bound` returns iterator to the first element ≥ **val**, while `upper_bound` gives first **> val** [21]. Example:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    vector<int> v = {11, 34, 56, 67, 89}; // sorted
    auto itlow = lower_bound(v.begin(), v.end(), 56);
    auto itup  = upper_bound(v.begin(), v.end(), 56);
    cout << "lower_bound of 56: " << *itlow << endl; // 56
    cout << "upper_bound of 56: " << *itup << endl;  // 67

    // If not sorted, sort first (undefined behavior otherwise)
    vector<int> u = {89,11,56,34,67};
    sort(u.begin(), u.end());
    // Now find bounds in a subrange
    cout << *lower_bound(u.begin(), u.begin()+3, 34) << endl; // first 3 sorted
    return 0;
}
```

Explanation: Both functions run in O(log n) time [22]. They require sorted input; if the data isn't sorted, call `sort` first.

2.4 Other Algorithms: `reverse`, `rotate`

- `std::reverse(begin, end)`: Reverses elements in the range (O(n)).
- `std::rotate(begin, middle, end)`: Rotates the range so that *middle* becomes the first element (O(n)).

Example:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    vector<int> v = {1,2,3,4,5};
    reverse(v.begin(), v.end());    // v = {5,4,3,2,1}
    rotate(v.begin(), v.begin()+2, v.end()); // v = {3,2,1,5,4}
    for(int x: v) cout << x << " ";
    cout << endl;
    return 0;
}
```

Explanation: These algorithms modify containers in linear time. They are useful for shifting and inverting sequences.

3. Graph and Tree Data Structures using STL

3.1 Adjacency List (using vectors or maps)

Graphs are often represented by adjacency lists. In C++, we can use **vectors of vectors** or **maps of lists/vectors**.

- **Using** `vector<vector<int>>`: For *n* vertices numbered 0..n-1, `vector<vector<int>> adj(n);`. Add edge by `adj[u].push_back(v)`, and for undirected graphs also `adj[v].push_back(u)`.
- **Using** `map<int, list<int>>` **or** `unordered_map<int, vector<int>>`: If vertices aren't 0..n-1 or are dynamic, use a map from vertex to list of neighbors [23].

Example: Building and printing an undirected graph with adjacency list.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int V = 4;
    vector<vector<int>> adj(V);
    // Add undirected edges
    auto addEdge = [&](int u,int v){
```

```
        adj[u].push_back(v);
        adj[v].push_back(u);
    };
    addEdge(0,1);
    addEdge(0,2);
    addEdge(1,2);
    addEdge(2,3);

    // Print adjacency list
    cout << "Adjacency list:" << endl;
    for(int u = 0; u < V; ++u) {
        cout << u << ":";
        for(int w: adj[u]) {
            cout << " " << w;
        }
        cout << endl;
    }
    return 0;
}
```

Explanation: We used `vector<vector<int>>` for a graph of 4 vertices. The space is O(V+E) and traversal is O(V+E) [24]. Alternatively, one could use `map<int,list<int>> adj;` for a map-based adjacency list [25].

3.2 Trees

Trees can be represented similarly (e.g. parent–child lists) or using maps/sets for ordered data. For example, a **BST** can be implemented via `std::set` (which is a red-black tree internally). In interviews, you might also store parent pointers or use adjacency lists for rooted trees. STL itself has no dedicated "tree" structure beyond `map`/`set`.

4. Heaps (Priority Queue)

Covered above in container section. Recall **max-heap** vs **min-heap** by comparator [19]. Use `priority_queue<int>` for max-heap, or `priority_queue<int, vector<int>, greater<int>>` for min-heap.

5. CRUD Examples Summary

Below is a concise summary of Create, Read, Update, Delete for key structures:

- **Vector:**
- *Create:* `vector<int> v;`
- *Read:* iterate with index or iterator (`v[i]` in loop).
- *Update:* `v[i]=x;` or modify via iterator.

• *Delete:* `v.pop_back()`, or `v.erase(pos)`. [3]

• **String:**

• *Create:* `string s = "Hello";`
• *Read:* iterate characters or use `s[i]`.
• *Update:* `s[1] = 'a';` or `s.replace()`.

• *Delete:* `s.erase(pos, len);`.

• **Set/Unordered_Set:**

• *Create:* `set<int> s; unordered_set<int> us;`
• *Read:* `for(int x: s)`, or use iterators.
• *Update:* `s.insert(x);` or (not applicable since unique).

• *Delete:* `s.erase(x);`.

• **Map/Unordered_Map:**

• *Create:* `map<int,string> mp;`
• *Read:* `for(auto &p: mp) cout<<p.first;`
• *Update:* `mp[key] = value;` (inserts or overwrites).

• *Delete:* `mp.erase(key);`.

• **Stack:**

• *Create:* `stack<int> st;`
• *Read:* peek via `st.top()`.
• *Update:* `st.push(x);`.

• *Delete:* `st.pop();`.

• **Queue:**

• *Create:* `queue<int> q;`
• *Read:* peek via `q.front()`.
• *Update:* `q.push(x);`.

• *Delete:* `q.pop();`.

• **Deque:**

• *Create:* `deque<int> dq;`
• *Read:* via indices or iterators.
• *Update:* `dq.push_back(x); dq.push_front(y);`.

- *Delete:* `dq.pop_back(); dq.pop_front();`.

- **Priority Queue:**

- *Create:* `priority_queue<int> pq;`
- *Read:* `pq.top()`.
- *Update:* `pq.push(x);`.
- *Delete:* `pq.pop();`.

6. Example Interview Problems

1. **Sliding Window Maximum:** Use a deque to solve the "max of all subarrays of size k" problem in O(n) time. Example snippet:

```
vector<int> maxSlidingWindow(vector<int>& a, int k) {
    deque<int> dq;
    vector<int> ans;
    for(int i = 0; i < a.size(); i++){
        // Remove out-of-window elements
        if(!dq.empty() && dq.front() == i-k) dq.pop_front();
        // Maintain decreasing order in deque
        while(!dq.empty() && a[dq.back()] < a[i]) dq.pop_back();
        dq.push_back(i);
        if(i >= k-1) ans.push_back(a[dq.front()]);
    }
    return ans;
}
```

Tip: `deque` allows fast push/pop at both ends [18], ideal for sliding window.

2. **BFS/DFS on Graph:** Use `queue` for BFS and `vector<bool>` for visited. Example (BFS):

```
void bfs(int start, const vector<vector<int>>& adj) {
    vector<bool> vis(adj.size(), false);
    queue<int> q;
    q.push(start); vis[start] = true;
    while(!q.empty()) {
        int u = q.front(); q.pop();
        cout << u << " ";
        for(int v: adj[u]) {
            if(!vis[v]) { vis[v] = true; q.push(v); }
        }
    }
}
```

Tip: BFS uses a `queue` FIFO structure [16].

3. **Two-Sum with `unordered_map`:** Store seen numbers in an `unordered_map` for O(n) solution. For each `x`, check if `target-x` exists in map.

These realistic examples combine multiple STL components (vectors, deque, queue, unordered_map).

7. Edge Cases and STL Tips

- **Sorted Data Requirements:** Algorithms like `binary_search`, `lower_bound`, `upper_bound` **require sorted ranges**; otherwise behavior is undefined [20]. Always `sort()` first if needed.
- **Reserve Capacity:** For `vector`, if you know the maximum size `N`, call `v.reserve(N);` to avoid repeated reallocations [4].
- **Unordered Containers:** After many insertions, `unordered_map` / `unordered_set` may become slow if buckets are too few. Use `rehash()` or `reserve()` to reduce rehashes.
- **Iterator Invalidation:** In `vector`, inserting or erasing can invalidate iterators beyond the point. In `set/map`, erase returns a valid iterator to next element. Always be careful when erasing in loops.
- **Access vs. Find:** For `map` / `unordered_map`, using `mp[key]` will insert a key if it doesn't exist. Use `mp.find(key)` or `mp.count(key)` to check existence without insertion.
- **Stack/Queue:** Do not iterate directly. To "traverse" a stack/queue, copy it (like `stack<int> tmp = st;`) and pop elements in a loop.
- **Memory:** `std::string::reserve()` and container `max_size()` can help understand limits, but typically not needed in contest code.
- **Complexities Recap:**
- **Vector:** random access O(1), push_back amortized O(1), insert/erase middle O(n) [1].
- **Set/Map:** insert/search/erase O(log n) [6] [10].
- **Unordered:** insert/search/erase average O(1) [7] [12] (worst O(n)).
- **Stack/Queue:** all basic ops O(1) [13] [15].
- **Deque:** push/pop ends O(1) amortized [18].
- **Priority Queue:** push/pop O(log n), top O(1).
- **Algorithms:** sort O(n log n), binary_search and bounds O(log n) [22].

By leveraging these STL containers and algorithms, you can write concise, efficient code for many common problems in programming contests and interviews. Always consider corner cases, such as empty containers or large data sizes, and use appropriate container methods for clarity and speed.

Sources: Authoritative references on C++ STL (GeeksforGeeks) for containers and algorithms have been cited throughout [1] [6] [12] [13] [15] [17] [19] [22] [20], ensuring accurate details and complexities.

---

[1] [2] [3] [4] Vector in C++ STL - GeeksforGeeks
https://www.geeksforgeeks.org/cpp/vector-in-cpp-stl/

[5] String Functions in C++ - GeeksforGeeks
https://www.geeksforgeeks.org/cpp/cpp-string-functions/

6 Set in C++ STL - GeeksforGeeks
https://www.geeksforgeeks.org/cpp/set-in-cpp-stl/

7 9 Unordered Sets in C++ STL - GeeksforGeeks
https://www.geeksforgeeks.org/cpp/unordered_set-in-cpp-stl/

8 set vs unordered_set in C++ STL - GeeksforGeeks
https://www.geeksforgeeks.org/set-vs-unordered_set-c-stl/

10 Map in C++ STL - GeeksforGeeks
https://www.geeksforgeeks.org/cpp/map-associative-containers-the-c-standard-template-library-stl/

11 map vs unordered_map in C++ - GeeksforGeeks
https://www.geeksforgeeks.org/cpp/map-vs-unordered_map-c/

12 Unordered Map in C++ STL - GeeksforGeeks
https://www.geeksforgeeks.org/cpp/unordered_map-in-cpp-stl/

13 14 Stack in C++ STL - GeeksforGeeks
https://www.geeksforgeeks.org/cpp/stack-in-cpp-stl/

15 16 Queue in C++ STL - GeeksforGeeks
https://www.geeksforgeeks.org/cpp/queue-cpp-stl/

17 18 Deque in C++ STL - GeeksforGeeks
https://www.geeksforgeeks.org/cpp/deque-cpp-stl/

19 Priority Queue in C++ STL - GeeksforGeeks
https://www.geeksforgeeks.org/cpp/priority-queue-in-cpp-stl/

20 std::binary_search() in C++ STL - GeeksforGeeks
https://www.geeksforgeeks.org/cpp/binary-search-algorithms-the-c-standard-template-library-stl/

21 22 std::upper_bound and std::lower_bound for Vector in C++ STL - GeeksforGeeks
https://www.geeksforgeeks.org/cpp/upper_bound-and-lower_bound-for-vector-in-cpp-stl/

23 24 25 Implementation of Graph in C++ - GeeksforGeeks
https://www.geeksforgeeks.org/cpp/implementation-of-graph-in-cpp/