1. Time Complexity

Algorithm	bubble sort	selection sort	insertion sort	merge sort	quick sort	random quick sort	counting sort	radix sort	stl:sort
Average	$O(N^2)$	$O(N^2)$	$O(N^2)$	O(N log N)	O(N log N)	O(N log N)	O(N)	O(N)	O(N log N)
Sorted Ascending	O(N)	$O(N^2)$	O(N)	O(N log N)	$O(N^2)$	O(N log N)	O(N)	O(N)	O(N log N)
Sorted Descending	$O(N^2)$	$O(N^2)$	$O(N^2)$	O(N log N)	$O(N^2)$	O(N log N)	O(N)	O(N)	O(N log N)
Worst Ascending	O(N)	$O(N^2)$	O(N)	O(N log N)	$O(N^2)$	O(N log N)	O(N)	O(N)	O(N log N)
Worst Descending	$O(N^2)$	$O(N^2)$	$O(N^2)$	O(N log N)	$O(N^2)$	O(N log N)	O(N)	O(N)	O(N log N)
Stability[1]	yes	no	yes	yes	no	no	yes	yes	no
In place[2]	yes	yes	yes	no	yes	yes	no	no	no
Recursive	no	no	no	yes	yes	yes	no	no	yes
Comparisons based	yes	yes	yes						
Divide & Conquer				yes	yes	yes			yes

Data Structure	Vector	Singly Linked List	Stack	Queue	Double Linked List	Deque
access i[th] element	O(1)	O(N)	O(N)	O(N)	O(N)	O(1)
search(num)	O(N)	O(N)	Not allowed	Not allowed	O(N)	Not allowed
peek-front()/top()	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
peek-back()	O(1)	O(1)	Not allowed	O(1)	O(1)	O(1)
insert(0, num)	O(N)	O(1)	O(1)	Not allowed	O(1)	O(1)
insert(n, num)	O(1)	O(1)	Not allowed	O(1)	O(1)	O(1)
insert(i, num)	O(i) / worst O(N)	O(N)	Not allowed	Not allowed	O(N)	Not allowed
remove head	O(N)	O(1)	O(1)	O(1)	O(1)	O(1)
remove tail	O(1)	O(N)	Not allowed	Not allowed	O(1)	O(1)
remove(i)	O(i)/O(N)	O(N)	Not allowed	Not allowed	O(N)	Not allowed

[1] Stable: the relative order of elements with the same key value is preserved after sorting.

[2] In-place: it requires only a constant amount (e.g O(1)) of extra space during the sorting process. Not in-place sorting requires additional temporary array of size N.

2. Sorting Algorithm

A. Bubble sort

1. Pair-wise comparison of adjacent elements(a, b),
2. Swap the pair if (a>b),
3. Repeat step 1 & 2 until the end of the array,
4. Then reduce N by 1 and repeat step 1 until N = 1.

```
void bubble_sort(){
    for (j=0; j<n-1; j++)
        for(i=0; i<n-j-1; i++)
            if(A[i]>A[i+1])
                swap(A[i], A[i+1]);
}
```

B. Insertion sort

1. Start with one element,
2. Pick the next element and insert it into its proper sorted order,
3. Repeat step 2 for all the elements.

```
void insertion_sort(){
    for(i=1; i<n; i++){
        e = A[i]; j=i;
        while(j>0){
            if(A[j-1]>e) A[j]=A[j-1];
            else break;
        }
    }
}
```

C. Selection sort

1. Find the position of the smallest element X in the range of [lowerbound L ~ N-1],
2. Swap X with the Lth element,
3. Increase the lowerbound L by 1 and repeat step 1 until L = N-2.

```
void selection_sort(){
    int i, j, min_idx;

    for(i=0; i<n-1; i++){
        min_idx = i;
        for(j=i+1; j<n; j++){
            if(arr[j]<arr[min_idx]) min_idx=j;
        }
        swap(&arr[min_idx], &arr[i]);
    }
}
```

D. Merge sort

1. Merge each pair of individual element into sorted arrays of 2 elements,
2. Merge each pair of sorted arrays of 2 elements into sorted arrays of 4 elements..
3. Merge 2 sorted arrays of N/2 elements to obtain a fully sorted array of N.
4. Not memory efficient.

```
void merge(){
    int N = high-low+1;
    int b[n];
    int left=low, right=mid+1, bIdx=0;

    while(left<=mid && right<=high) //merging
        b[bIdx++] = (a[left]<=a[right]) ? a[left++]:a[right++];
    while(left<=mid) b[bIdx++] = a[left++];
    while(right<=high) b[bIdx++] = a[right++];

    for(int k=0; k<N; k++) a[low+k]=b[k];
}

void merge_sort(){
    if(low<high) {
        int mid=(low+high)/2;
        merge_sort(a, low, mid);
        merge_sort(a, mid+1, high);
        merge_sort(a, low, mid, high);

    }
}
```

E. Counting sort

1. Used for small range of inputs with duplicates,
2. O(N) is to count the frequencies of the elements and O(N+k) is to print out the output in sorted order where k is the range of the input integers.
3. If k is relatively big, counting sort is not feasible due to memory limitation.

F. Radix sort

1. Used for a big range of inputs with **w** digits
2. For the least significant digit to the most significant digit, N items will be passed through and put according to the active digit into 10 queues ([0..9]).
3. Propotional to number of digits -> O(dN)
4. Can combine with counting sort for integers with large range but of few digits.

G. Quick sort	H. Random quick sort
Divide step: 1. in arr[i..j], choose arr[i] as the pivot **p** 2. Partition the items of arr[i..j] into three parts: arr[i..m-1], arr[m], and arr[m+1...j]. 3. arr[i..m-1] contains items smaller than **p**. 4. arr[m] is the pivot (index m is the position of **p**). 5. arr[m+1..j] contains items that a greater or equal to **p**. 6. Recurively sort the two parts. **Conquer step:** do nothing :o 1. O(N) for partition(a, i, j) since there is only a single for-loop. 2. Time complexity depends on the number of times partition() is called. 3. Best case: quick sort splits the array into two equal halves -> gives the depth of recursion O(log N). 4. At each level of O(N) comparisons, the time comlexity is O(NlogN).	1. Randomly select the pivot between arr[i..j]. 2. This combination of randomness yiels partitions of lucky (half-pivot-half), somewhat lucky, somewhat unlucky and extremely unlucky (empty, pivot, the rest), which gives an average time complexity of O(N log N).

```
int partition(int arr[], int i, int j) {
    int p=arr[i];
    int m=i;
    for(int k=i+1; k<=j; k++){
        if(arr[k]<p){
            m++;
            swap(a[k], a[m]);
        }
    }
    swap(a[i], a[m]);

    return m;
}

void quick_sort(int arr[], int low, int high){
    if(low<high){
        int pivotIdx = partition(a, low, high); //O(N)
        quick_sort(a, low, pivotIdx-1); //recursively sort left subarr
        quick_sort(a, pivotIdx+1, high);//then sort right subarr
    }
}
```

```
void quick_sort(int arr[], int low, int high){
    if(low<high){
        int pivotIdx = partition(a, low, high); //O(N)
        quick_sort(a, low, pivotIdx-1); //recursively sort left subarr
        quick_sort(a, pivotIdx+1, high);//then sort right subarr
    }
}

int random_pivoting(int arr[], int i, int j) {
    srand(time(NULL));
    int random = i + rand()%(j-i);

    swap(a[random], a[j]);

    return partition(arr, i, j);
}

void quick_sort(int arr[], int low, int high){
    if(low<high){
        int pivotIdx = random_pivoting(a, low, high); //O(N)
        quick_sort(a, low, pivotIdx-1); //recursively sort left subarr
        quick_sort(a, pivotIdx+1, high);//then sort right subarr
    }
}
```

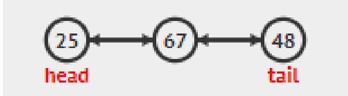
3. List + Binary Heaps

A. Singly Linked List

- forward_list<datatype> variable (SLL)
- linear data structure

before_begin()	Return iterator to before beginning
end()	Return iterator to end
empty()	Return true if list is empty
size()	Return size
front()	Return the head value
erase_after(it)	Erase at pos of it
Insert_after(it, num)	Insert element at position of iterator
push_front(num)	Insert element at beginning
pop_front()	Delete the first element(head)
reverse()	Reverse list
x.swap(y)	Swap the content in two lists
x.merge(y)	Merge list y into list x
x.splice_after(x.before_begin(), y)	Transfer y to x at x's beginning

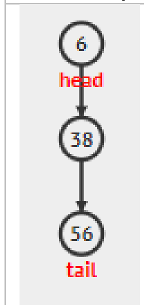
B. Doubly Linked List



- list<datatype> variable (DLL)
- linear data structure

empty()	Return true if list is empty
size()	Return size
front()	Return the head value
back()	Return the tail value
insert(i, num)	Push element at index i
push_front(num)	Add element at the head
push_back(num)	Add element at the tail
pop_front()	Pop the first element(head)
pop_back()	Pop tail
reverse()	Reverse list
x.swap(y)	Swap the content in two lists
x.merge(y)	Merge list y into list x
x.splice(pos, y)	Transfer list y to x at a position pos

C. Stack (LIFO)

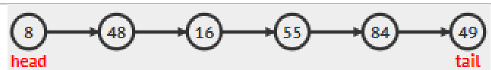


- stack<datatype> variable
- linear data structure

empty()	Return true if queue is empty
size()	Return size
front()	Return the head value
back()	Return the tail value
push(num)	Add element at the tail / Enqueue
pop()	Pop the first element(head) / Dequeue
x.swap(y)	Swap the content in two queues

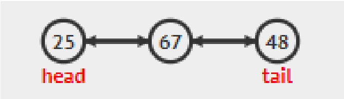
C. Queue (FIFO)

- queue<datatype> variable
- linear data structure



empty()	Return true if queue is empty
size()	Return size
front()	Return the head value
back()	Return the tail value
push(num)	Add element at the tail / Enqueue
pop()	Pop the first element(head) / Dequeue
x.swap(y)	Swap the content in two queues

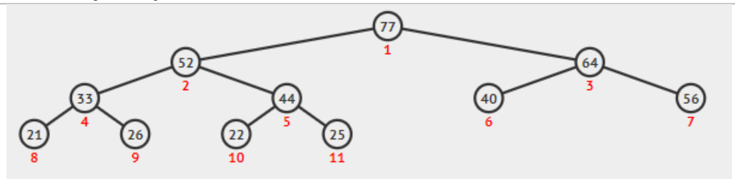
E. Deque



- deque<datatype> variable
- linear data structure

begin()	Return iterator to beginning
end()	Return iterator to end
empty()	Return true if deque is empty
size()	Return size
at(index)	Access the element at the index
front()	Return the head value
back()	Return the tail value
insert(i, num)	Push element at index i
push_front(num)	Add element at the head
push_back(num)	Add element at the tail
pop_front()	Pop the first element(head)
pop_back()	Pop tail
clear()	Erase the content in deque

F. Binary Heaps



- priority_queue<datatype, vector<datatype>, greater<datatype>> min_heap;
- priority_queue<datatype, vector<datatype>, less<datatype>> max_heap;

empty()	Return true if container is empty
size()	Return size
top()	Return the head value / top / max / min
push(num)	Insert elements.
pop()	Pop the top element and calls heap sort functions in pop()
x.swap(y)	Swap the content in two queues
empty()	Return true if container is empty

Algorithm	**Average**	**Worst**
build heap / create	O(N)	O(N log N)
search	O(N)	O(N)
Insert – bubble sort upwards in O(log N) heighet	O(1)	O(log N)
top – peek max/min element	O(1)	O(1)
extractmax / pop – bubble sort downwards in O(log N) height	O(log N)	O(log N)
heap_sort – call extracmax N times	O(N log N)	O(N log N)

4. Hash Table

A. General Information

Properties	Definitions
1) Map some non-integer keys to integers keys. Map large integers to smaller integers. 2) The key must be mapped to non-negative integer values. The range of keys must be small. Otherwise the memory usage will be very large. 3) The keys must be dense (i.e. not many gaps in the kay values). 4) Fast to compute, uses as minimum slots as possible, scatter the keys into different base addresses as uniformly as possible together with minimum collisions.	1) M = HT.length = the current hash table size. 2) base = (key%HT.length). 3) step = the current probing step starting from 1. 4) secondary = smaller_prime – key%smaller_prime, which is computed by another hash function.

B. Collision Resolution

Open Addressing			Closed addressing
- Always find an empty slot if it exists. Thus the storing depends on other objects in the hash table. - More cache friendly. - Give different probes sequences / alternative addresses when 2 different keys collide - Require deleted markers and inefficient if there are many deletions and insertions.			- Storing of the object is completely dependent on the hash function. - Easy to obtain elements with same hash.
Linear Probing	**Quadratic Probing**	**Double Hashing**	**Separate Chaining**
Index = (base+step*1)%M	Index = (base+step*step)%M	Index = (base+step*secondary)%M	Looks like adjacency list
- Scanning forward one index at a time for the next empty slot. - search(v) function stops when its encountered an empty cell if there's no lazy deletion. - primary cluster -> covers the base address of a key which increase the running time beyond O(1). - Long sequences of filled slots increase search and insert time.	- Scanning forward by quadratic factor for the next empty slot - Secondary cluster -> clusters formed along the path of probing instead of around the base address. - not all cells are examined when looking for an insertion index. - may stuck in an infinite loop searching unless **a**<0.5 and **M** is prime number. - increase time taken if collision occurs. - may result in a waste of memory.	- scanning forward by the second hash function for the next empty slot. - reduce primary and secondary clustering.	- If two keys **x** and **y** both have the same hash value I, both will be appended to the front/back of a doubly linked list i. - Thus remove(v) requires search(v) whose time complexity depends on the load factor **a**[3] - search(v) and remove(v) are worst O(1+**a**), best O(1). - insert(v) will be O(1) - Unable to fully utilize unused slots

[3] Load factor **a** = N/M where M is the number of copies of DLL

C. STL Maps

map

- Associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order.
- Slow access to individual elements by its key.
- Allows for direct iteration of the subsets based on their order.
- Implemented as a binary search tree

at(index) / operator[index]	Access element
rbegin / rend	Reverse beginning / end
crbegin / crend	Const reverse beginning / end
lower_bound (X)	Return the iterator to X
upper_bound (X)	Return the iterator to 1 element > X
lower_bound (v.begin, v.end(), X)	Return the iterator to pos X via BST
upper_bound (v.begin, v.end(), X)	Return the iterator to pos X via BST
key_comp	Define the order of the keys
value_comp	Define the order of the values
equal_range(X)	Return the pair of iterators to the lower bound & upper bound+1 of that element
equal_range(v.begin(), v.end())	Return the pair of iterators to the lower bound & upper bound+1 of that range
find(X)	Return the iterator point to X
erase(X) / erase(it)	Erase all occurrence of X / delete *it
Insert(first, second)	O(log N)

unodered_map

- Associative containers that store elements formed by the combination of a key value and a mapped value.
- Allows for fast retrieval of individual elements based on their keys
- The elements are not sorted in any order.
- Less efficient for range iteration through a subset of their elements.
- Uses separate chaining.

at(index) / operator[index]	Access element
equal_range(X)	Return the pair of iterators to the lower bound & upper bound+1 of that element
equal_range(v.begin(), v.end())	Return the pair of iterators to the lower bound & upper bound+1 of that range
key_eq	?
hash_function	?
bucket	?
bucket_count	?
bucket_size	?
rehash	?
load_factor	?
insert(first, second)	O(1)

multimap

- Allow multiple elements to have equivalent keys

unordered_multimap

- Allow multiple elements to have equivalent keys.
- Elements with equivalent keys are grouped together in the same bucket and requires an iterator (equal_range) to iterate through.

D. STL Sets

set

- Container that stores unique elements following a specific order.
- The value is itself the key and each value is unique.
- The elements cannot be modified once in the container but can be inserted or remove from the container.
- Slower than unordered_set to access individual elements by their key
- Allow for direct iteration on subsets based on their order
- Traversal using iterators
- Implemented as binary search trees

at(index) / operator[index]	Access element
rbegin / rend	Reverse beginning / end
crbegin / crend	Const reverse beginning / end
lower_bound (X)	Return the iterator to X
upper_bound (X)	Return the iterator to 1 element > X
lower_bound (v.begin, v.end(), X)	Return the iterator to pos X via BST
upper_bound (v.begin, v.end(), X)	Return the iterator to pos X via BST
key_comp	Define the order of the keys
value_comp	Define the order of the values
equal_range(X)	Return the pair of iterators to the lower bound & upper bound+1 of that element
equal_range(v.begin(), v.end())	Return the pair of iterators to the lower bound & upper bound+1 of that range
find(X)	Return the iterator point to X
erase(X) / erase(it)	Erase all occurrence of X / delete *it
Insert(first, second)	O(log N)

unordered_set

- Container that stores unique elements in no particular order, and which allow for fast retrieval of individual elements based on their value.
- The value of an element is also its key.
- Keys are immutable and the elements cannot be modified once in the container but can be inserted and removed.
- They are organized into buckets to allow for fast access to individual elements directly by their value but less efficient for range iteration.
- Hash table used but only able to hash *int, float, string.*

at(index) / operator[index]	Access element
equal_range(X)	Return the pair of iterators to the lower bound & upper bound+1 of that element
equal_range(v.begin(), v.end())	Return the pair of iterators to the lower bound & upper bound+1 of that range
key_eq	?
hash_function	?
bucket	?
bucket_count	?
bucket_size	?
rehash	?
load_factor	?
insert(first, second)	O(1)

multiset

- Container that stores elements following a specific order and where multiple elements can have equivalent values (duplicates).
- Allow for direct iteration on subsets based on their order.

unordered_multiset

- Containers that store elements in no particular order, allowing fast retrieval of individual elements based on their value.
- Allow different elements to have equivalent values (duplicates).

5. Binary Search Tree

A. General Properties

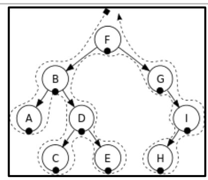
Query		Update		Others	
Search(v)	O(h) ~ O(N)	Insert(v)	O(h) ~ O(N)	Rank(v)	rank(Min())=1, rank(Max())=N – log(N)
Predecessor(v) / Successor(v)	O(h)	Remove(v)	O(h) ~ O(N)	Select(k)	Get the element at rank k – log(N)
Inorder traversal	O(logN) / O(N)	Create BST		BST Height	Floor($\log_2$N) ~ N-1
Min / Max	O(h) ~ O(N)			AVL Height	O($\log_2$N) or h < 2$\log_2$N

B. Definition

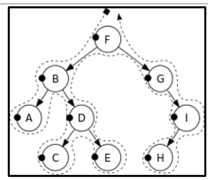
Internal vertices	Leaf
- Nodes that are not leaf	- Nodes that have no other children
Successor	**Predecessor**
- If V has right subtree -> get min in the right subtree - If V has no right subtree -> traverse the ancestor until a right turn	- If V has left subtree -> get max in the left subtree - If V has no left subtree -> traverse the ancestor until a left turn

C. Tree Traversal

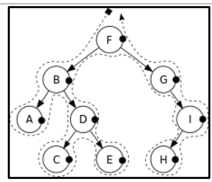
Inorder:
1. Traverse the left subtree.
2. Visit the root.
3. Traverse the right subtree
4. Order: A B C D E F G H I



Preorder:
1. Visit the root.
2. Traverse the left subtree
3. Traverse the right subtree
4. Order: F B A D C E G I H



Postorder:
1. Traverse the left subtree
2. Traverse the right subtree
3. Visit the root
4. Order: A C E D B H I G F



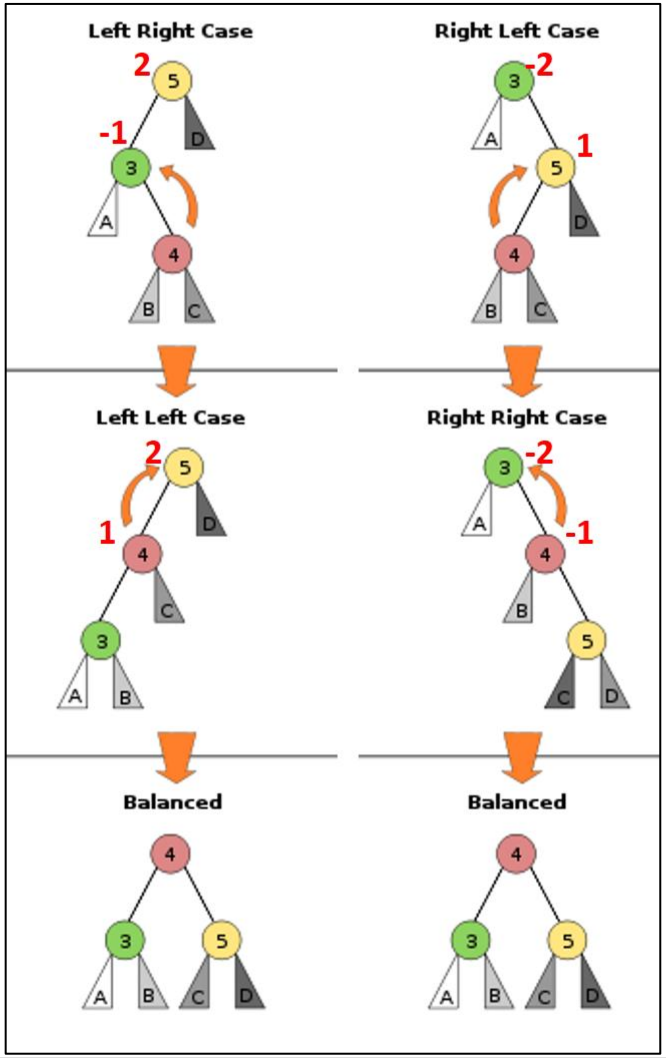
D. Adelson-Velskii & Landis Tree

- Height-balanced if balance factor = |v.left.height – v.right.height| <= 1
- Compare the difference between left and right subtrees
- height is the number of edges on the path from the a vertex V to its deepest leaf thus get_height(V) is O(1)

1) bf(x) == +2; bf(x.left) == 1; right_rotate(x);
2) bf(x) == -2; bf(x.right) == -1; left_rotate(x);
3) bf(x) == +2; bf(x.left) == -1; left_rotate(x.left); right_rotate(x);
4) bf(x) == -2; bf(x.right) == -1; right_rotate(x.right); left_rotate(x)

Algorithm	Average	Worst
Space	O(N)	O(N)
Search	O(logN)	O(logN)
Insert	O(logN)	O(logN)
Delete	O(logN)	O(logN)
Rank	O(logN)	O(logN)
Select	O(logN)	O(logN)

- Insert(v): update the height and balance factor of the traversed vertices and use one of the 4 rotations to balance it at most once.
- Remove(v): update the height and balance factor of the traversed vertices and use one of the 4 rotations to balance it up to logN times.

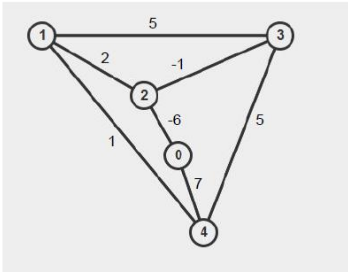


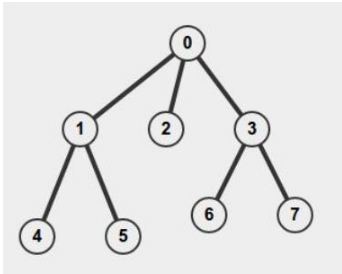
6. STL Containers

Container	**Vector**		**Stack Queue**		**Deque**		**List**		**Priority Queue**		**Set**		**Map**	
Insert	push_back	**O(1)**	push	**O(1)**	push_back push_front	**O(1)**	push_back push_front	**O(1)**	push	**O(logN)**	insert	**O(logN)**	[] operator	**O(logN)**
Delete	pop_back	**O(1)**	pop	**O(1)**	pop_front pop_back	**O(1)**	pop_front pop_back	**O(1)**	pop	**O(logN)**	erase	**O(logN)**	erase	**O(logN)**
Random Access	[] operator	**O(1)**	NIL		[] operator	**O(1)**	Loop Through	**O(N)**	NIL		find	**O(logN)**	[] operator	**O(logN)**
Access	front back	**O(1)**	s.top q.front	**O(1)**	front back	**O(1)**	front back	**O(1)**	top	**O(1)**	NIL (Use iterators)		NIL (Use iterators)	
Sorted	No (Use STL sort)		No		No (Use STL sort)		No (Use List.sort)		**Yes**		**Yes**		**Yes**	
Binary Search	lower_bound upper_bound	**O(logN)**	NIL		lower_bound upper_bound	**O(logN)**	NIL		NIL		lower_bound upper_bound	**O(logN)**	lower_bound upper_bound	**O(logN)**
Unique	No		No		No		No		No		Yes (Use Multiset for non-unique keys)		Unique **keys** Non-unique values (Use Multimap for non-unique keys)	
Iterators	**Yes**		No		**Yes**		**Yes**		No		**Yes**		**Yes**	

7. General Properties of Graphs

A. Graph Representation





Edge List

Vertex **u**, Vertex **v**, Edge/weight **w**
(0, 2, -6)
(0, 4, 7)
(1, 2, 2)
(1, 3, 5)
(1, 4, 1)
(2, 3, -1)
(3, 4, 5)

- vector<tuple<int, int, int>>
- Space Complexity O(E)

Adjacency List

Vertex	List (neighbour, weight)
0	(2, -6), (4, 7)
1	(2, 2), (3, 5), (4, 1)
2	(0, -6), (1, 2), (3, -1)
3	(1, 5), (2, -1), (4, 5)
4	(0, 7), (1, 1), (3, 5)

- vector<vector<pair<int, int>>>
- Space Complexity O(V+E)

Adjacency Matrix

-	**0**	**1**	**2**	**3**	**4**
0	-	-	-6	-	7
1	-	-	2	5	1
2	-6	2	-	-1	-
3	-	5	-1	-	5
4	7	1	-	5	-

- Symmetric if bidirectional
- Space Complexity $O(V^2)$
- 2D array
- O(1) checking of edges

Tree Representation

Vertex	Parent Node
0	-
1	0
2	0
3	0
4	1
5	1
6	3
7	3

- V-1 edges

B. Directed Acyclic Graph

Definition: A graph whose edges are not bi-directional and has no cycles or self-loops.

Properties:
1) Maximum number of directed edges of a graph with **V** number of vertices - **V(V-1)/2.**
2) Adding more than **V(V-1)/2** edges will form a bi-directional edge.

C. Complete Graph

Definition: A simple graph that is the densest.

Properties:
1) There is an edge between any pair of vertices.
2) A graph of **V** vertices and **E = V(v-1)/2** edges (**$E=O(V^2)$**)
3) No multi-edges, self-loops, unweighted, undirected.
4) Space complexity $O(V^2)$

D. Bipartite Graph

Definition: Undirected graph with **V** vertices that can be partitioned into 2 disjoint set of vertices **m, n** where **V = m + n**. There is no edge between members of the same set.

Properties:
2) Free from odd-length cycle
3) Complete graph if there's an edge from any **m** to all **n**.

8. Graph Traversal

<table>
<tr><th>A. Depth First Search (DFS)</th><th>B. Breadth First Search (BFS)</th></tr>
<tr><td>

Definition: Starts from a source vertex s which has X neighbours. It then recursively (or with stack) explores all reachable vertices from the first neighbour u and backtrack to vertex u, then do the same for other neighbours until It finishes exploring the last neighbour and its reachable vertices.

Properties:

1) Time complexity O(V+E) -> adjacency list
2) The sequence of the traversal forms a DFS spanning tree.
3) Find connected components - one DFS per CC. (still O(V+E))
4) Detecting cycle - status[u] to update {unvisited, explored, visited}.
5) Bipartite checker (alternate colouring)
6) Find strongly connected components

</td><td>

Definition: Starts from a source vertex s and uses a queue to order the visitation sequence as breadth as possible before going deeper. Every vertex of the same level order is enqueued and then dequeued to explore its neighbours which are then enqueued from the queue. Then dequeue the next one from the queue.

Properties:

1) Time complexity O(V+E) -> adjacency list
2) Forms BFS spanning tree (in 2040C) – a connected set of directed edges that form a rooted tree covering all vertices.
3) Topological sort
4) Find connected components
5) Bipartite checker
6) Applications: BitTorrent, social network, crawlers in search engines, GPS.

</td></tr>
<tr><td>

```
void dfs(int vertex_id) {
    if (visited[vertex_id]) return;
    visited[vertex_id] = true;

    for(int i=0; i<adjList[Vertex_id].size(); i++){
        dfs(adjList[vertex_id][i]);
    }
}
```

</td><td>

```
void bfs(int source_v){
    queue<int> q;
    visited[source_v] = 1;

    q.push(source_v);
    while(!q.empty()){
        int v = q.front(); q.pop();
        for(auto& it : adjList[v]){
            if(visited[it]) continue;
            q.push(it);
            visited[it] = 1;
        }
    }
}
```

</td></tr>
</table>

9. Graph Traversal Application

A. Detecting cycles

DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back-edge present in the graph. Back-edge always point from a vertex to one of its ancestors. We need to track 3 states in the DFS – status[u] = {unvisited, exploring/explored (>= 1 neighbour not explored), visited (all neighbours explored}.

```
void dfs(string place){
    status[place] = 1; //exploring
    //cout << "mom at a newly explored city " << place <<endl; //test

    for(auto &neighbours: cities[place]){
        //cout << "mom is exploring this neighbour " << neighbours <<endl;

        if(status[neighbours]==0) dfs(neighbours); //to visit the neighbours
        else if(status[neighbours]==1) { //explored
            found = true;
            //cout << "mom found a cycle at " <<neighbours <<endl;
        }
    }

    status[place] = 2; //fully visited
    //cout << "mom finish visitng this city " <<place <<endl;
}
```

B. Finding connected components

Enumerate all vertices v that are reachable from an **unvisited** vertex s in an undirected graph and enumerate all vertex v with status updated to **visited**.

```
void findCC() {
    int CC=0;

    for(auto&u : map) status[u]=0;
    for(auto&u : map){
        if(status[u]==0) {
            CC++;
            dfs(u);//floodfill all the v in that CC
    }
}
```

C. Topological Sort / Kahn's Algorithm

Linear ordering in directed acyclic graph in which each vertex comes before all vertices to which it has outbound edges.

1) Min number of topological ordering: 1
2) Max number of topological ordering: V! (for V disjoint vertices which gives O(V!))

```
void toposort(){
    queue<int> Q;
    /*Finding the verticies v with no incoming edges*/
    /*Adding v with no incoming edge to queue Q*/
    while (!Q.empty()){
        int u = Q.front(); Q.pop();
        for(auto& x : AdjList[u]){
            /*deleting edge u -> x*/
            if (/*x has no incoming edge*/) Q.push(x);
        }
    }
}
```

D. Bipartite Checker

Check whether the set of vertices can be divided into two sets such that there is an edge between set 1 and set 2 but no edges belonging to the same set.

- Use BFS to 'colour' the vertices. Adjacent vertices should have different colours.

```
bool bipartite_checker(vectir<vector<int>> AL){
    /*initialising all colours to -1*/
    colour[0] = 1;
    queue<int> q;
    q.push(0);

    while(!q.empty()){
        int tempV = q.front();
        q.pop();

        for(auto& it : AL[tempV]){
            if(colour[it]==-1){
                colour[it] = 1 - colour[temp];
                q.push(it);
            }else if(colour[it]==colour[temp]) return false;
        }
    }

    return true;
```

10. Single-Source Shortest Paths

Algorithm	Bellman Ford	Dijkstra	Modified Dijkstra	DFS	BFS	Dynamic Programming
Time Complexity	O(VE) - very very slow	O((V+E)*logV)	O((V+E)*logV)	O(V+E)	O(V+E)	O(V+E)
Negative weight	Yes	No	Yes	No	No	Yes
Negative cycle	No -INF	No	No	No	No	No
Faster Algorithm for SSSP	- Detect negative cycle	- Graph without negative weight	- Graph without negative weight cycle	-Tree	- Unweighted graphs - Tree	- Directed Acyclic Graphs
All Pairs Shortest Path (APSP)	lol	-O(V(V+E)logV)	- O(V(V+E)logV)	- O(V(V+E))	- O(V(V+E))	- Floyd Warshall - O(V^3)

A. Bellman Ford's Algorithm

Algorithm

```
void bellman_ford(){
    dist[s]=0;

    for(int i=1; i<V-1; i++){
        for(auto&x : egdeList){
            //if tuple is used
            u = get<0>(x);
            v = get<1>(x);
            w = get<2>(x);

            //relax
            dist[v] = min(dist[v], dist[u]+w);
        }
    }
}
```

Application – finding negative cycles (run once will do)

```
void find_negative_circle(){
    bellman_ford();
    bool negative_circle=false;

    for(int i=1; i<V; i++){
        //if tuple is used
            u = get<0>(x);
            v = get<1>(x);
            w = get<2>(x);

            //relax
            if(dist[u]!=INF && d[v] > d[u]+w) {
                negative_circle = true;
            }
    }
}
```

Bellman Ford Killer (VisuAlgo processes edges in decreasing order)

B. Dijkstra

Time Complexity Explanation	Real Life Application
The implementation of min heap extractMin() runs in O(logV). Since V vertices will be extracted, it will be O(VlogV). Also, all the edges will be relaxed and updated in the min heap with the previous value deleted, giving O(ElogV). Thus O((V+E)logV).	- Electric car loses (positive) energy on uphill and gains (negative) energy on downhill. - Money exchanger (where negative cycle possibly exists)

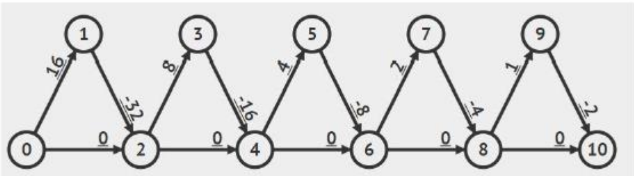
Dijkstra Algorithm

```
void dijkstra(int source_v){
    priority_queue<pi, vector<pi>, greater<pi>> pq;

    dist[source_v]=0;
    pq.push({0, source_v});

    while(!pq.empty()){
        auto x = pq.top(); pq.pop();
        int curr_v = x.second, curr_w = x.first;

        if (visited[curr_v]) continue; //lazy deletion method
        visited[curr_v] = 1;

        for(auto& i: adjList[curr_v]){
            if(dist[i.first]==INF || dist[i.first]>curr_w+i.second){
                dist[i.first] = curr_w+i.second;
                pq.push({dist[i.first], i.first});
            }
        }
    }
}
```

Modified Dijkstra Algorithm (Lazy Update)

```
void modified_dijkstra(int source_v){
    priority_queue<pi, vector<pi>, greater<pi>> pq;

    dist[source_v]=0;
    pq.push({0, source_v});

    while(!pq.empty()){
        auto x = pq.top(); pq.pop();
        int curr_v = x.second, curr_w = x.first;

        if(curr_w>dist[curr_v]) continue; //bellmanford if uncommented

        for(auto& i: adjList[curr_v]){
            if(dist[i.first]==INF || dist[i.first]>curr_w+i.second){
                dist[i.first] = curr_w+i.second;
                pq.push({dist[i.first], i.first});
            }
        }
    }
}
```

Dijkstra / Modified dijkstra killer (wrong answer / TLE)



C. Floyd Warshall

Algorithm

Find the length of all the shortest paths between all pairs of vertices.

```
void Floyd_Warshall(){
    for(int k=0; k<V; k++)
        for(int i=0; i<V; i++)
            for(int j=0; j<V; j++)
                AM[i][j] = min(AM[i][j], AM[i][k]+AM[k][j]);
```

11. Sample Question

1. In DFS/BFS, compare stack/queue implementation with recursion?

- Stack/Queue consumes memory and is cache-friendly, recursive calls might take up a huge portion of RAM

- Stack uses only one traversal at a time

2. Find the second shortest path with at least a differing edge

- Solvable by Floyd Warshall / modified dijkstra

- Select one edge from U -> V; Find the shortest path from Source->U and V-> Destination by enumerating all edges.

3. Set intersection/union between two sorted arrays A and B.

- If binary search is used: intersection -> O(N log M); union -> O(N log M + M log N) for arrA[N] and arrB[N].

- Use 2 pointers method -> if *pointer A > *pointer B, pointer B increases; if *pointer B > *pointer A, pointer A increases; if they are equal, intersection is found.

- This results in a time complexity of O(N+M) .

4. Finding a target pair x and y such that x+y equals to a target z in the same array.

- If binary search is used: O(N log N).

- Use 2 pointers method -> set pointer X = arr.begin() and pointer Y = arr.end() – 1; if *pointer X + *pointer Y < Z, pointer X++; if *pointer X + *pointer Y > Z, pointer Y--; if the sum equals to Z, keep it and move the two pointers towards the centre.

5. Array v.s. vector

- Array: specific number of elements of a data type + may result in unused extra space.

- Vector: dynamic memory allocation + no declaration of array size.

6. Array v.s. linked list implementation

Array	**Linked List**
- Collection of elements having same data type.	- An ordered collection of elements which are linked by pointers.

- Array provides fast and random access in O(1) operation - Elements are stored in consecutive manner in memory - Insertion and deletion takes O(N) time - Can be single dimension or multidimensional - No requirement of extra space to store pointer	- Elements can only be accessed sequentially in O(N) operation - Dynamic allocation of nodes. - Fast insertion and deletion. - Singly, doubly or circular linked list - Extra memory is needed for the pointers in each node.

7. Reverse a SLL?

- Loop through L, store pointers to every element in an array. Then loop through the array in reverse order, construct the reversed linked list.

- This results in O(N). We only can do this in O(N) time since there are N elements in total.

8. Reverse a queue?

- Push the front of the queue into stack and then de-queue. After every element is transferred to the stack, push/enqueue the top of the stack into the queue and then pop the top of the stack.

- This results in O(N) operation. We only can do this in O(N) time since there are N elements in total.

9. Why is quick sort recommended for array but merge sort for linked list?

- Quick sort is an in-place sort which does not require any extra storage whereas merge sort requires O(N) extra storage. The N denoting the array size can be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Unlike arrays, linked list nodes may not be adjacent in memory so we cannot do random access in the linked list. Random quick sort requires a lot of this random access while merge sort accesses the data sequentially. Inserting items in the middle in the linked list requires O(1) extra space and O(1) time.

10. Simulate a calculator

- Use stack A to store brackets and numbers and stack B to store the operators. Whenever the closing bracket is read, operate the numbers with the operator from the top of stack B until an opening bracket is read in stack A.

11. Total number of simple paths in a DAG

- Get the topological order from vertex s to vertex t. Count the number of paths/outgoing edges at that vertex to in the reverse topo order to reach the destination vertex, which is the sum of paths from its outgoing edges to reach dest. The total number of simple paths is the sum of the reachable roads to dest vertex from vertex s.