Chapter 1 Sorting

Algorithm	Type	Recursive	Stable	In-Place	Time Complexity		
					Random	Sorted	
						Ascending	Descending
Bubble Sort	Comparison		√	√	$O(N^2)$	$O(N)$	$O(N^2)$
Selection Sort	Comparison			√	$O(N^2)$	$O(N^2)$	$O(N^2)$
Insertion Sort	Comparison		√	√	$O(N^2)$	$O(N)$	$O(N^2)$
Merge Sort	Comparison	√	√		$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
Quick Sort	Comparison	√		√	$O(N \log N)$	$O(N^2)$	$O(N^2)$
Random Quick Sort	Comparison	√			$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
Counting Sort	Non-Comparison		√		$O(N)$	$O(N)$	$O(N)$
Radix Sort	Non-Comparison		√		$O(N)$	$O(N)$	$O(N)$

Additional Notes:

1. All comparison-based sorting algorithms have a lower-bound time complexity of $\Omega(N \log_2 N)$.
2. Non-comparison-based sorting algorithms achieved linear time complexity with the presence of certain assumptions.

Bubble Sort

Algorithm Outline:

1. Compare a pair of adjacent items (a, b).
2. Swap that pair if the items are out of order (eg. when $a > b$).
3. Repeat steps 1 and 2 until the end of array (the last pair is the $(N-2)$-th and $(N-1)$-th items due to 0-based indexing).
4. By now, the largest item will be at the last position. Reduce N by 1 and repeat Step 1 until $N = 1$.
5. Optimize by implementing a check where the sorting will be terminated early if no swapping is done.

```
void BubbleSort(int arr[], int size) {
    int temp;
    bool swapEncountered;

    for (int limit = size-2; limit >= 0; limit--) {
        swapEncountered = false;
        for (int i=0; i<=limit; i++) {
            if (arr[i] > arr[i+1]) {
                temp = arr[i];
                arr[i] = arr[i+1];
                arr[i+1] = temp;
                swapEncountered = true;
            }
        }
        if(!swapEncountered){
            break;
        }
    }
}
```

Insertion Sort

Algorithm Outline:

1. Start with item X, in the range of $[L...N-1]$.
2. Check for the insertion point of X (eg. between N and M where $N < X$ and $M > X$) and insert X.
3. Increase the lower bound L by 1 and repeat steps 1 and 2 until $L = N - 1$.

```
void InsertionSort(int arr[], int size) {
    int temp;

    for (int i = 1; i < size; i++) { // O(N)
        temp = arr[i]; // temp is the item to be inserted
        for (int j = i-1; j >= 0 && arr[j] > temp; j--) // can be fast or slow
            arr[j+1] = arr[j]; // make a place for temp
        arr[j+1] = temp; // this is the insertion point
    }
}
```

Selection Sort

Algorithm Outline:

1. Find the position of the smallest item X, in the range of $[L...N-1]$.
2. Swap X with the L-th item.
3. Increase the lower-bound L by 1 and repeat step 1 and 2 until $L = N - 2$.

```
void SelectionSort(int arr[], int size) {
    int min_index, temp;

    for (int start = 0; start < size-1; start++) {
        min_index = start;
        for (int i = start+1; i < size; i++){
            if (arr[i] < arr[min_index]) {
                min_index = i;
            }
        }
        temp = arr[start];
        arr[start] = arr[min_index];
        arr[min_index] = temp;
    }
}
```

Merge Sort

Algorithm Outline:

1. Merge each pair of individual element (which is by default, sorted) into sorted arrays of 2 elements.
2. Merge each pair of sorted arrays of 2 elements into sorted arrays of 4 elements.
3. Repeat the process.
4. Merge 2 sorted arrays of $\frac{N}{2}$ elements to obtain a fully sorted array of N elements.

```
void merge(int a[], int low, int mid, int high) {

    // subarray1 = a[low..mid], subarray2 = a[mid+1..high], both sorted
    int N = high-low+1;
    int b[N];
    int left = low, right = mid+1, bIdx = 0;

    while (left <= mid && right <= high) // the merging
        b[bIdx++] = (a[left] <= a[right]) ? a[left++] : a[right++];
    while (left <= mid)
        b[bIdx++] = a[left++]; // leftover, if any
    while (right <= high)
        b[bIdx++] = a[right++]; // leftover, if any
    for (int k = 0; k < N; k++)
        a[low+k] = b[k]; // copy back
}

void MergeSort(int a[], int low, int high) {
    // the array to be sorted is a[low..high]
    if (low < high) { // base case: low >= high (0 or 1 item)
        int mid = (low+high) / 2;
        MergeSort(a, low  , mid ); // divide into two halves
        MergeSort(a, mid+1, high); // then recursively sort them
        merge(a, low, mid, high); // conquer: the merge routine
    }
}
```

Additional Notes:

1. The actual execution of `MergeSort()` does not split to two subarrays level by level, but it recursively sorts the left subarray first before dealing with the right subarray.
2. The number of times `MergeSort()` is called is $\log N$ times and in each call of `MergeSort()`, there are N comparisons made. Hence, the time complexity is $O(N \log N)$.
3. Merge Sort guarantees its $O(N \log N)$ performance regardless of the original ordering of the input.
4. Merge Sort however has a space complexity of $O(N)$ i.e. not memory efficient.

Quick Sort

Algorithm Outline:

1. Choose an item p (known as the pivot).
2. Then partition the items of $a[i..j]$ into three parts: $a[i..m-1]$, $a[m]$, and $a[m+1..j]$:
   a. $a[i..m-1]$ (possibly empty) contains items that are smaller than p.
   b. $a[m]$ is the pivot p, i.e. index m is the correct position for p in the sorted order of array a.
   c. $a[m+1..j]$ (possibly empty) contains items that are greater than or equal to p.
3. Recursively sort the two parts.

```
int partition(int arr[], int i, int j) {
    int p = arr[i]; // p is the pivot
    int m = i; // S1 and S2 are initially empty
    for (int k = i+1; k <= j; k++) { // explore the unknown region
        if (arr[k] < p) {
            m++;
            swap(arr[k], arr[m]); // C++ STL algorithm std::swap
        } // notice that we do nothing in case 1: a[k] >= p
    }
    swap(arr[i], arr[m]); // final step, swap pivot with arr[m]
    return m; // return the index of pivot, to be used by Quick Sort
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivotIdx = partition(arr, low, high); // O(N)
        // a[low..high] ~> a[low..pivotIdx-1], pivot, a[pivotIdx+1..high]
        quickSort(arr, low, pivotIdx-1); // recursively sort left subarray
        // a[pivotIdx] = pivot is already sorted after partition
        quickSort(arr, pivotIdx+1, high); // then sort right subarray
    }
}
```

Additional Notes:

1. Partitioning function has a time complexity of $O(N)$.

Counting Sort:

Assumption: Items to be sorted are integers with small range.

Algorithm Outline:

1. Count the frequency of occurrence of each integer (in that small range).
2. Loop through that small range to output the items in sorted order.

Additional Notes:

1. The time complexity is $O(N)$ to count the frequencies and $O(N+k)$ to print out the output in sorted order where k is the range of the input integers.
2. Counting Sort is not feasible when k is relatively big due to memory limitation, since there is the need to store frequencies of those k integers.

Radix Sort:

Assumption: Items to be sorted are integers with large range but of few digits.

Algorithm Outline:

1. Treat each item to be sorted as a string of w digits (pad integers that have less than w digits with leading zeroes if necessary).
2. From the least significant (rightmost) digit to the most significant digit (leftmost), pass through the N items and put them according to the active digit into 10 Queues (one for each digit $[0..9]$).
3. Re-concatenate the groups again for subsequent iteration.

Additional Notes:

1. Radix sort is a modified counting sort to achieve the linear time complexity and preserve stability.
2. Time complexity is $O(w(N+k))$ where w is the number of digits.

Additional Notes:

1. Algorithm in the form of $O\left(1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{N}\right)$ follows the harmonic series and thus have a time complexity of $O(\log N)$.
2. Algorithm in the form of $O(N+\frac{N}{2}+\frac{N}{4}+\cdots+\frac{N}{N})$ follows the convergent geometric series and thus have a time complexity of $O(2N) = O(N)$.

Chapter 2 Linked List

Basic Operators

Operators	Functions
`get(i)`	Return element a_i at index i
`search(v)`	Check if item/data v exists and returns its index if it exists
`insert(i, v)`	Insert item/data v at index i
`remove(i)`	Remove element a_i at index i

Array Implementation

Properties:

1. Contiguous in memory i.e. each element is connected to one another in actual memory allocation.
2. Must be a compact array (array with no gap) i.e. if there are N items in the array (that has size M, where $M \geq N$), then only index $[0..N-1]$ are occupied and other indices $[N..M-1]$ should remain empty.

Advantages	Disadvantages
Simple construct to handle a collection of items	Size of compact array in static-size array types is not infinite, but finite, and maximum size may not be known in advanced, resulting in unused or extra space
Some operations are faster as opposed to linked list ADT	Dynamic size array types still pose issue of space wastage and copying/shifting items overhead

Operators	Functions	Time Complexity	
		Best Case	Worst Case
`get(i)`	Returns `A[i]`. *This simple operation will be unnecessarily complicated if the array is not compact.*	$O(1)$	
`search(v)`	Check each index $i \in [0..N-1]$ if `A[i] == v`. *This is because v (if it exists) can be anywhere in index [0..N-1].*	$O(1)$ v is found at the first position	$O(N)$ v is not found in the list
`insert(i, v)`	Shift items $\in [i..N-1]$ *to* $[i+1..N]$ (from backwards) and set `A[i] = v`. *This is so that v is inserted correctly at index i and maintain compactness.*	$O(1)$ insert at $i = N$, there is no shifting of element	$O(N)$ insert at $i = 0$, we shift all N elements, $O(N)$
`remove(i)`	Shift items $\in [i+1..N-1]$ to $[i..N-2]$, overwriting the old `A[i]`. *This is to maintain compactness.*	$O(1)$ remove at $i = N-1$, there is no shifting of element	$O(N)$ remove at $i = 0$, shift all N elements

Linked List

Properties:

1. Non-contiguous memory i.e. each element is not connected to one another in actual memory allocation.
2. Elements are ordered from index 0 to index $N-1$ by associating item i with its neighbour item $i+1$ through a pointer.
3. Structure of a single vertex of Singly Linked List (SLL) resembles the following:

```
struct Vertex { // we can use either C struct or C++/Java class
    int item; // the data is stored here, an integer in this example
    Vertex* next; // this pointer tells us where is the next vertex
};
```

4. A head and a tail pointer pointing to a_0 element and a_{N-1} element respectively.
5. Structure of a single vertex of Doubly Linked List (DLL) resembles the following:

```
struct Vertex { // we can use either C struct or C++/Java class
    int item; // the data is stored here, an integer in this example
    Vertex* next; // this pointer tells us where is the next vertex
    Vertex* prev; // this pointer tells us where is the previous vertex
};
```

6. `prev` pointer in DLL makes it possible to move/iterate backwards at the expense of two-times memory usage requirement since each vertex records one additional pointer.

Operators	**Implementation in C++**		**Time Complexity (Worst Case)**

get(i)	Vertex* get(int i) { // returns the vertex
	    Vertex* ptr = head; // we have to start from head
	    for (int k = 0; k < i; k++) // advance forward i time(s)
	        ptr = ptr->next; // the pointers are pointing to the higher index
	    return ptr;
	}
			$O(N)$
			
			i can be as big as index $N-2$

insert(i,v)	Empty List
	Head
		Vertex* vtx = new Vertex(); // create new vertex vtx from item v
		vtx->item = v;
		vtx->next = head; // link this new vertex to the (old) head vertex
		head = vtx; // the new vertex becomes the new head
			$O(1)$

	In Between	Vertex* pre = get(i-1); // traverse to (i-1)-th vertex, O(N)
		aft = pre.next // aft cannot be null, think about it
		Vertex vtx = new Vertex(); // create new vertex
		vtx->item = v;
		vtx->next = aft; // link this
		pre->next = vtx; // and this
			$O(N)$
			
			Need to traverse the list

	Tail	Vertex* vtx = new Vertex();
		vtx->item = v; // create new vertex vtx from item v
		tail->next = vtx; // just link this, as tail is the i = (N-1)-th item
		tail = vtx; // now update the tail pointer
			$O(1)$

remove(i)	Head	if (head == NULL) return; // avoid crashing when SLL is empty
		Vertex* temp = head; // so we can delete it later
		head = head->next; // book keeping, update the head pointer
		delete temp; // which is the old head
			$O(1)$

	In Between	Vertex* pre = get(i-1); // traverse to (i-1)-th vertex, O(N)
		Vertex* del = pre->next, aft = del->next;
		pre->next = aft; // bypass del
		delete del;
			$O(N)$
			
			Need to traverse the list

	Tail
	(SLL)
		Vertex* pre = head;
		temp = head->next;
		while (temp->next != null) // while my neighbour is not the tail
		pre = pre->next, temp = temp->next;
		pre->next = null; // alternatively: pre = get(N-2), temp = get(N-1)
		delete temp; // temp = (old) tail
		tail = pre; // update tail pointer
			$O(N)$
			
			Need to update the tail pointer from item $N-1$ backwards by one unit to item $N-2$ so that future insertion after tail remains correct

	Tail (DLL)	`Vertex temp = tail // remember tail item`
		`tail = tail.prev // the key step :O`
		`tail.next = null // remove this dangling reference`
		`delete temp // remove the old tail`
			$O(1)$

Linked List Variants

Properties		Stack	Queue
Design Type		Protected Singly Linked List	Protected Singly Linked List
Data Structure		Last-In-First-Out	First-In-First-Out
Accessibility	**Peek**	Head	Head
	Push	Head	Tail
	Pop	Head	Head
Time Complexity	**Peek**	$O(1)$	$O(1)$
	Push	$O(1)$	$O(1)$
	Pop	$O(1)$	$O(N)$ for array implementation $O(1)$ for linked list implementation
Application		1. Bracket Matching 2. Postfix Calculator 3. Depth-First-Search graph traversal (implicit)	1. Breadth-First-Search graph traversal

Double-Ended Queue (Deque)

Properties:

1. Protected Doubly Linked List which elements can be searched, added to or removed only from either the head or tail.

Operators	Time Complexity
`GetHead()`	$O(1)$
`GetTail()`	
`InsertHead()`	
`InsertTail()`	
`RemoveHead()`	
`RemoveTail()`	

Time Complexity of ADT Operations

Input Type Algorithm	Singly Linked List	Stack	Queue	Doubly Linked List	Deque
`search(v)`	$O(N)$	*Not allowed*	*Not allowed*	$O(N)$	*Not allowed*
`peek-front()`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`peek-back()`	$O(1)$	*Not allowed*	$O(1)$	$O(1)$	$O(1)$
`insert(0, v)`	$O(1)$	$O(1)$	*Not allowed*	$O(1)$	$O(1)$
`insert(N, v)`	$O(1)$	*Not allowed*	$O(1)$	$O(1)$	$O(1)$
`insert(i, v)`, i ∈[1..N-1]	$O(N)$	*Not allowed*	*Not allowed*	$O(N)$	*Not allowed*
`remove(0)`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`remove(N−1)`	$O(N)$	*Not allowed*	*Not allowed*	$O(1)$	$O(1)$
`remove(i)`, i ∈[1..N-1]	$O(N)$	*Not allowed*	*Not allowed*	$O(N)$	*Not allowed*

Additional Notes:

1. The above time complexities of algorithms for ADTs (except for DLL) assumes the implementation of using a Singly Linked List.
2. Actual time complexity for certain algorithms for each ADT may differ in C++ STL implementation as the container type used for the ADTs are unlikely to be Singly Linked List (i.e. may implement the ADTs using container type `std::vector` or `std::deque` by default).

C++ ADT STL Implementation

ADT	Type	Header File	Remarks
`std::list`	Container	`<list>`	Already a DLL
`std::deque`	Container	`<deque>`	Not using DLL implementation
`std::stack`	Container Adaptor	`<stack>`	Uses `std::deque` container type by default Can be defined with `std::list` and `std::vector` container type
`std::queue`	Container Adaptor	`<queue>`	Uses `std::deque` container type by default Can be defined with `std::list` container type

Chapter 3 Binary Heap

Properties:

1. Binary (Max) Heap: A complete binary tree that maintains the Max Heap property.
2. Complete Binary Trees: Every level in the binary tree, except possibly the last/lowest level, is completely filled, and all vertices in the last level are as far left as possible.
3. Binary Max Heap Property: The parent of each vertex - except the root - contains value greater than the value of that vertex.
4. For a Binary Heap of N elements, the height of its complete binary tree will not be taller than $O(\log N)$.
5. Binary Heap is a possible data structure to model an efficient Priority Queue ADT.

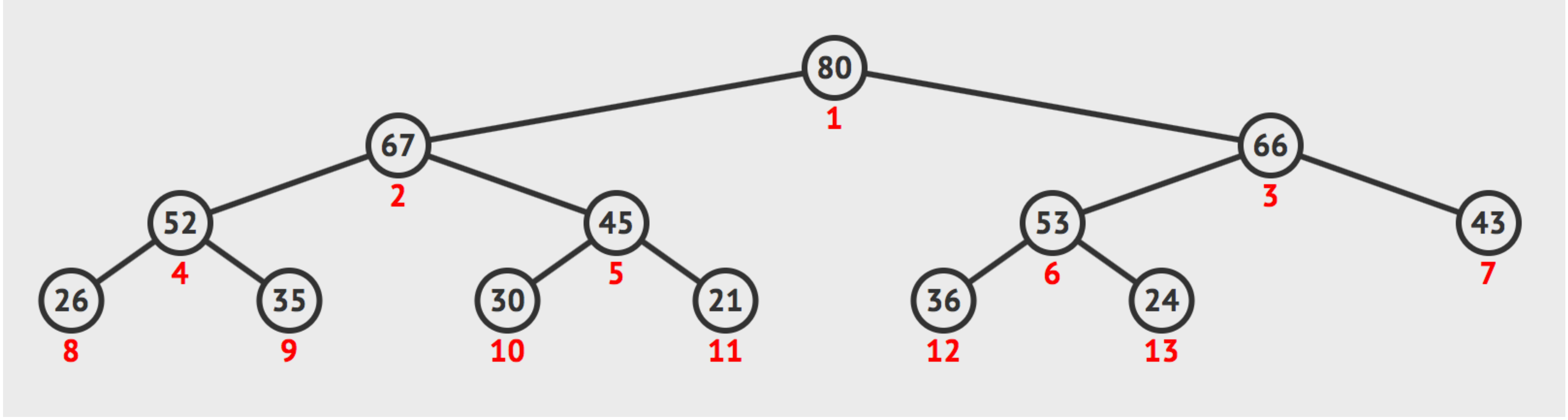
Priority Queue

Properties:

1. Each element has a "priority" and an element with higher priority is served before an element with lower priority.
2. Ties are broken with standard First-In First-Out rule as with normal Queue.
3. Array and Linked List implementations will lead to slow enqueue( ) and dequeue( ) i.e. $O(N)$ – constantly searching for the max which is $O(N)$

Operators	Functions
enqueue(x)	Put a new element (key) x into the PQ (in some order)
dequeue()	Return an existing element that has the highest priority (key) in the PQ and if ties, return the one that is inserted first

Compact-Based Array



A complete binary tree can be stored efficiently as a compact array A as there is no gap between vertices of a complete binary tree/elements of a compact array. To simplify the navigation operations below, 1-based array is used. The above diagram displays the index of each vertex as a red label below each vertex. As such, basic binary tree traversal operations with simple index manipulations with help of bit shift manipulation.

Operator	**Functions**
`i>>1`	Index i divided by 2. Obtains parent node.
`i<<1`	Index i multiplied by 2. Obtains left node.
`(i<<1)+1`	Index i multiplied by 2 and added by 1 Obtains right node.

Binary Heaps Operators

Operator	Function	Time Complexity
`insert(v)`	Insert new item v into the Binary Max Heap at last index $N+1$ to maintain a compact array Fixes the max heap property from insertion point upwards(an operation known as Shift Up/Bubble Up/Increase Key)	$O(\log N)$
`ExtractMax()`	Removes the max item (root) from the Binary Max Heap Last element will replace the root to maintain a compact array Fixes the max heap property from root downwards (an operation known as Shift Down/Bubble Down/Heapify)	$O(\log N)$
`create()`	Insert N integers by calling `insert(v)` N times	$O(N \log N)$
	Fixes the max heap property from last internal vertex back to the root	$O(N)$ Derivation: $Height\ of\ Binary\ Tree, H = \log_2 N$ $Number\ of\ Nodes\ at\ level\ h, V = \lceil \frac{N}{2^{h+1}} \rceil$ $Time\ Complexity\ of\ Shift\ Down\ of\ a\ node\ at\ level\ h = O(h)$ $\therefore Time\ Complexity\ of\ Shift\ Down\ at\ level\ h = V \times O(h)$ $\therefore Total\ Time\ Complexity$ $= Sum\ of\ Time\ Complexity\ of\ Shift\ Down\ Operation\ for\ all\ h$ $= \sum_{h=0}^{H} V \times O(h) = O(2N) = O(N)$
`HeapSort()`	Sort the heap by calling `extractMax()` N times	$O\ (N \log N)$

Additional Notes:

1. Maximum number of swaps between heap element = **max(left, right) + 1**
2. Minimum number of swaps between heap element = **0**
3. Maximum number of comparisons between heap element = **max(left, right) + number of direct children**
4. Minimum number of comparisons between heap element = **N – 1**

Chapter 4 Hash Tables

Basic Operators

Operators	Functions
`search(v)`	Determine if v exists in the ADT
`insert(v)`	Insert v into the ADT
`remove(v)`	Remove v from the ADT

Direct Addressing Table

Properties:

1. For integer keys that are small $[0\ to\ M-1]$.
2. All Table ADT Operations are $O(1)$.
3. An Boolean array of size M can be implemented as follow:

Operators	Implementations
`search(v)`	Check if `A[v]` is `true` (filled) or `false` (empty)
`insert(v)`	Set `A[v]` to be `true`
`remove(v)`	Set `A[v]` to be `false`

4. It is possible to use satellite data (such as Strings) instead of Boolean variables to record existence of keys.

Limitations:

1. Keys must be easily mapped to non-negative integer values.
2. Range of keys must be small or memory usage will be large.
3. Keys must be dense (i.e. not many gaps in the key values) or DAT will contain too many empty cells.

Hashing

Properties:

1. Able to map non-integer keys to integer keys.
2. Able to map large integers to smaller integers.
3. Influence the density or load factor $\alpha = \frac{N}{M}$ of hash table where N is the number of keys and M is the size of Hash Table.
4. A hash function of `h(v)=v%M` where M is the size of the hash table can be implemented to hash operators as follow:

Operators	Implementations
`search(v)`	Check if `A[h(v)] != -1` (where −1 represents empty cell)
`insert(v)`	Set `A[h(v)] = v`
`remove(v)`	Set `A[v] = -1`

5. It is also possible to use satellite data, which has the following implementation for hash operators:

Operators	Implementations
`search(v)`	Return `A[h(v)]` which is a `pair<v, satellite-data>`, though possibly empty
`insert(v)`	Set `A[h(v)] = pair<v, satellite-data>`
`remove(v)`	Set `A[v] = empty pair`

Hash Function

Properties:

1. A good hash function has the following desirable properties:
    a. Fast to compute i.e. $O(1)$.
    b. Uses as minimum Hash Table of size M as possible.
    c. Scatter the keys into different base addresses as possible.
    d. Experience minimum collisions as possible.
2. Perfect hash function has the following properties:
    a. One-to-one mapping between keys and hash values i.e. no collisions at all.
    b. Table size is the same as the same as the number of keywords supplied.

3. Best implementation for hashing of integer is `h(v)=v%M` where v is the item to be mapped and `M` is the size of Hash Table (number of slots). `M` should also be a prime number to reduce the number of collisions.
4. Best implementation for hashing of string is as follows:

```
int hash_function(string v) { // assumption 1: v uses ['A'..'Z'] only
    int sum = 0;                // assumption 2: v is a short string
    for (auto &c : v) // for each character c in v
        sum = ((sum*26)%M + (c-'A'+1))%M; // M is table size
    return sum;
}
```

Additional Notes:

1. Hashing of floating-point variable types should be avoided due to the precision of floating-point variables, which may result in different keys being generated for the same value.
2. Size of Hash Table, M, should be a prime number to reduce the number of common factors and common multiples of M, thereby reducing collisions.
3. In C++11, it is only possible to hash integer, floating-point and string type variables. Declarations like `unordered_set<pair<int, int>>` are undefined.

Collision Resolution

Open Addressing

Outline: All hashed keys are located in a single array. The hash code of a key gives its base address. Collision is resolved by checking/probing multiple alternative addresses (hence the name open) in the table based on a certain rule.

Desirable Open Addressing Properties:

1. Always find an empty slot if it exists.
2. Minimize clustering.
3. Give different probe sequences when 2 different keys collide.
4. Fast time-complexity of $O(1)$.

Techniques:

Techniques	Linear Probing	Quadratic Probing	Double Hashing
Probing Sequence/Index	$(base + step \times 1)\%M$	$(base + step \times step)\%M$	$(base + step \times secondary)\%M$

$$M = HT.length = current\ hash\ table\ size$$
$$base = (key\%HT.length)$$
$$step = current\ probing\ step$$
$$secondary = smaller_{prime} - key\%smaller_{prime}$$

Operation	Implementation
`insert(v)`	1. Start from primary hash value key, $k = h(v)$. 2. If k is empty, map v to k. 3. Otherwise, move forward i steps where i is the probing index until an empty cell/deleted is encountered and map v to $k + i$.
`search(v)`	1. Start from primary hash value key, $k = h(v)$. 2. If k is v, then v has been found. 3. Otherwise, move forward i steps where i is the probing index and check if $k + i$ is v. 4. If an empty cell/deleted cell is encountered, then v is not in the Hash Table.
`remove(v)`	1. Start from primary hash value key $k = h(v)$. 2. If k is v, set k to *DEL*. 3. Otherwise, move forward i steps where i is the probing index and set $k + i$ to *DEL* if $k + i$ is v. Note: DEL is a special symbol to indicate that the cell can be by-passed if necessary by future `search(v)` operation, but can be overwritten by future `insert(V)` operation. This deletion strategy is known as **Lazy Deletion**.

Linear Probing Technique:

1. Linear Probing collision resolution technique involves scanning forwards one index at a time for the next empty/deleted slot (wrapping around when necessary) whenever there is a collision.
2. Linear Probing can create large primary clusters (a cluster is a collection of consecutive occupied slots; a cluster that covers the base address of a key is called the primary cluster of the key) that will increase the running time of `search(v)/insert(v)/remove(v)` operations beyond O(1).

Quadratic Probing Technique:

1. Quadratic Probing collision resolution technique involves scanning forward index that increases quadratically for the next empty/deleted slot (wrapping around if necessary) whenever there is a collision.
2. Quadratic Probing can create secondary clusters (clusters that are formed along the path of probing as a result of using the same pattern in probing by all keys). If two distinct keys have the same base address, their Quadratic Probing sequences are going to be the same indexes
3. Probing might also be in an endless cycle unless $\alpha < 0.5$ and M is a prime number:

Proof by Contradiction:
    Let x and y be 2 quadratic probing steps where x ≠ y and x and y yield the same address modulo M
    h(v) + x * x = h(v) + y * y mod M;
    x * x = y * y mod M // strike out h(v) from both sides)
    x * x – y * y = 0 mod M // move y * y to LHS
    (x – y) * (x+y) = 0 mod M // rearrange the formula
    Now, either (x-y) or (x+y) has to equal to 0;
    Since x != y, then (x-y) cannot be 0.
    As 0 <= x < y <= (M/2) and M is a prime > 3 (an odd integer), then (x+y) also cannot be 0 mod M.
Contradiction! Hence for the first M/2 quadratic probing steps cannot yield he same address modulo M (if we set M to be a prime number greater than 3).

4. However, secondary clustering in Quadratic Probing is not as bad as primary clustering in Linear Probing as a good hash function should theoretically disperse the keys into different base addresses ∈ $[0..M-1]$ in the first place.

Double Hashing Technique:

1. Double Hashing collision resolution technique involves scanning forward index that base on a second hash function, $h2(v)$ for the next empty/deleted slot (wrapping around if necessary) whenever there is a collision.
2. Usually (for Integer keys), $h(2v) = M' - v\%M'$ where M' is a smaller prime than M. This makes $h2(v) \in [1..M']$, which is diverse enough to avoid secondary clustering.

Closed Addressing

Outline: Hash Table looks like an Adjacency List (a graph data structure). The hash code of a key gives its fixed/closed base address. Collision is resolved by appending the collided keys inside a (Doubly) Linked List identified by the base address.

Separate Chaining Technique:

1. Using M copies of auxiliary data structures (usually Doubly Linked Lists) if two keys a and b both have the same hash value i, both will be appended to the (front/back) of Doubly Linked List i.
2. The load factor $\alpha = \frac{N}{M}$ describes the average length of the M lists and it will determine the performance of `search(v)` due to the need to explore α elements on average i.e. `search(v)` has a time-complexity of $O(1 + \alpha)$.
3. `remove(v)` will also have a time-complexity of $O(1 + \alpha)$ since it requires `search(v)` .
4. `insert(v)` has a time complexity of $O(1)$.
5. If bound α to a small constant, all `search(v)/insert(v)/remove(v)` operations using Separate Chaining will be $O(1)$.

Additional Notes:

1. Open Addressing collision resolution has poor performance when there are a lot of deletions involved as compared to Closed Addressing collision resolution due to the need to travel through the DEL spaces as well.

Rehashing

Function: The performance of Hash Table degrades when the load factor α gets higher. For (standard) Quadratic Probing collision resolution technique, insertions might fail when the Hash Table $\alpha > 0.5$. Rehashing resolves such performance issue of the Hash table.

Algorithm Outline:

1. Build another Hash Table about twice as big with a new hash function.
2. Go through all keys in the original Hash Table, recompute the new hash values, and re-insert the keys (with their satellite-data) into the new Hash Table.
3. Delete the original Hash Table.
4. A rule of thumb is to rehash when $\alpha \geq 0.5$ if using Open Addressing and when $\alpha > small\ constant\ close\ to\ 1.0$ if using Separate Chaining.

Chapter 5: Binary Search Tree

Properties:

1. Binary tree where each vertex has only up to 2 children.
2. All vertices in the left subtree of a vertex must hold values smaller than its own while all vertices in the right subtree of a vertex must hold values larger than its own.
3. There can only be one root vertex in a BST.
4. Leaf vertex does not have any child and there can be more than one leaf vertex.
5. Vertices that are neither leaf nor root are called internal vertices.
6. For N elements in a BST, the lower bound height $h > \log N$ if N elements can be inserted in perfect order so that the BST is perfectly balanced.

$$\therefore Minimum\ Height = \lfloor \log_2 N \rfloor$$

$N \leqslant 1 + 2 + 4 + ... + 2h$
$N \leqslant 20 + 21 + 22 + \cdots + 2h$
$N < 2h+1$ (sum of geometric progression)
$\log_2 N < \log_2 2h+1$
$\log_2 N < (h+1) * \log_2 2$ ($\log_2 2$ is 1)
$h > (\log_2 N)-1$ (algebraic manipulation)
$h > \log_2 N$

7. For N elements in a BST, the upper bound height $h < N$ if N elements are inserted in ascending/descending order to get skewed right/left BST.

$$\therefore Maximum\ Height = N - 1$$

Comparisons with Vector/Arrays

	Unsorted Arrays/Vectors	Sorted Arrays/Vectors	Unbalanced BST	Balanced BST
`search(v)`	$O(N)$ There might be the need to explore all N elements if v does not actually exist.	$O(\log N)$ Implemented using Binary Search method.	$O(N)$ Height can be $N-1$	$O(\log N)$
`insert(v)`	$O(1)$ Implemented by putting v at the back of the array.	$O(N)$ Implement using insertion-sort like strategy to ensure that the array remains sorted.	$O(N)$ Height can be $N-1$	$O(\log N)$
`remove(v)`	$O(N)$ Implemented by first searching for v (which is an $O(N)$ operation) and later closing the resulting gap after deletion (which is also an $O(N)$ operation).	$O(N)$ Closing the resulting gap after deletion is an $O(N)$ operation.	$O(N)$ Height can be $N-1$	$O(\log N)$

Data Structure Types

Static Data Structure are data structures that are only efficient if there is rarely or no update operations (such as insert and remove operations).

Dynamic Data Structure are data structures that are efficient even if there are many update operations. BST (especially Balanced BST) are in this category.

Query Operations

These operations do not alter the BST structure.

Operation	Implementation	Time Complexity
`search(v)`	1. Set the current vertex to be the root 2. Check if the current vertex is smaller/equal/larger than integer v 3. Traverse to the right subtree if current vertex is smaller than v 4. Traverse to the left subtree if current vertex is smaller than v 5. Repeat steps 1 to 4 until v is found	$O(H)$ $where\ H\ is\ the\ height\ of\ the\ tree$
`FindMin()` `FindMax()`	1. Set the current vertex to be the root 2. Keep traversing to the left subtree to obtain `FindMin()` 3. Keep traversing to the right subtree to obtain `FindMax()`	
`successor(v)`	1. If v has a right subtree, the minimum integer in the right subtree of v must be the successor of v 2. If v does not have a right subtree, traverse the ancestors until the first vertex w that is greater than v (Note: v should be the maximum element in the left subtree of w) 3. If v is the maximum integer in the BST, v does not have any successor	
`predecessor(v)`	1. If v has a left subtree, the maximum integer in the left subtree of v must be the predecessor of v 2. If v does not have a left subtree, traverse the ancestors until the first vertex w that is smaller than v (Note: v should be the minimum element in the right subtree of w) 3. If v is the minimum integer in the BST, v does not have any predecessor	
In-order Traversal Order: A B C D E F G H I	1. Check if the current node is empty/null 2. Traverse the left subtree by recursively calling the in-order function 3. Display the data part of the root (or current node) 4. Traverse the right subtree by recursively calling the in-order function TLDR: left – root – right; this operation obtains a list of sorted integers inside the BST.	$O(N)$
Pre-order Traversal Order: F B A D C E G I H	1. Check if the current node is empty/null 2. Display the data part of the root (or current node) 3. Traverse the left subtree by recursively calling the pre-order function 4. Traverse the right subtree by recursively calling the pre-order function TLDR: root – left - right	

Post-order Traversal Order: A C E D B H I G F	1. Check if the current node is empty/null. 2. Traverse the left subtree by recursively calling the post-order function. 3. Traverse the right subtree by recursively calling the post-order function. 4. Display the data part of the root (or current node) TLDR: left – right – root	

Update Operations

These operations are likely to alter the BST structure.

Operation	Implementation	Time Complexity
`insert(v)`	1. Set the current vertex to be the root 2. Check if the current vertex is smaller/equal/larger than integer v 3. Traverse to the right subtree if current vertex is smaller than v 4. Traverse to the left subtree if current vertex is smaller than v 5. Create new vertex in insertion point and place v there	$O(H)$ $where\ H\ is\ the\ height\ of\ the\ tree$
`remove(v)`	1. Set the current vertex to be the root 2. Check if the current vertex is smaller/equal/larger than integer v 3. Traverse to the right subtree if current vertex is smaller than v 4. Traverse to the left subtree if current vertex is smaller than v 5. If v found is a leaf vertex, just simply remove the vertex ($O(H + 1)$) 6. If v found is an internal vertex with exactly one child, remove the vertex and connect the child with the vertex's parent ($O(H + 1)$) 7. If v found is an internal vertex with exactly two children, replace the vertex with its successor and delete the duplicated successor in its right subtree ($O(H + H)$)	

Additional Notes:

1. Number of different BST = $\frac{1}{n}\binom{2n}{n}$

AVL Tree

Properties:

1. AVL Tree is an implementation of Balanced BST which height is consistently less than $2 \times \log N$
2. `Height(v)` of each vertex (where `height(v)` is the number of edges on the path from vertex v down to its deepest leaf) is saved in each vertex so that each vertex's height can be accessed in O(1).
3. `Height(v)` can be computed by $height(v) = \max(v.left.height, v.right.height) + 1$ where $v.left.height$ and $v.right.height$ are the height of left and right children respectively.
4. `Height(v)` is only required to be computed after insert and remove operation as only the `height(v)` of vertices along insertion and removal path may be affected. This means that only $h = 2 \times \log N$ vertices may change its height attribute:

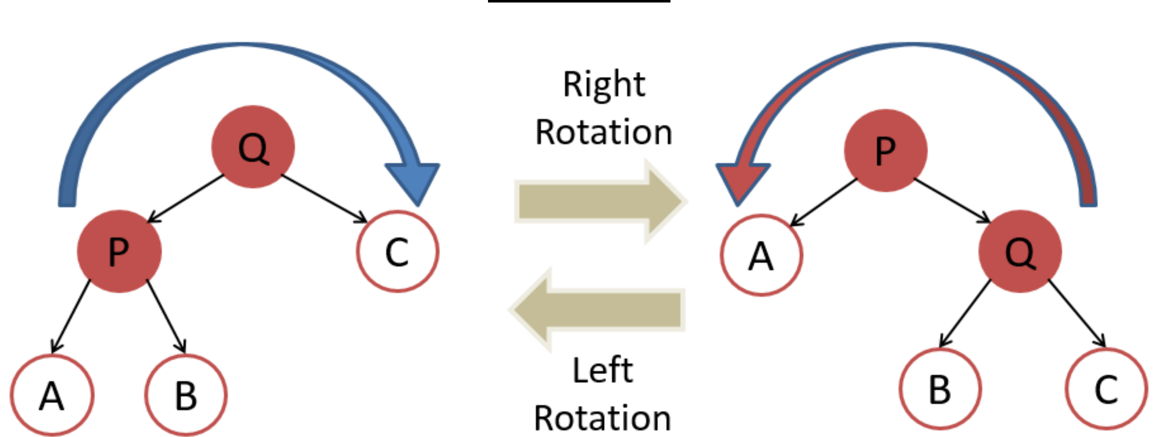
   Let N_h be the minimum number of vertices in a height-balanced AVL Tree of height h.
   For any AVL Tree of N vertices, $N \geqslant N_h$ – (1)

   $N_h = 1 + N_{h-1} + N_{h-2}$ (formula for minimum-size AVL tree of height h)
   $N_h > 1 + 2*N_{h-2}$ (as $N_{h-1} > N_{h-2}$)
   $N_h > 2*N_{h-2}$ (obviously)
   $N_h > 4*N_{h-4}$ (recursive)
   $N_h > 8*N_{h-6}$ (another recursive step)
   ... (we can only do this h/2 times, assuming initial h is even)
   $N_h > 2^{h/2}*N_0$ (we reach base case)
   $N_h > 2^{h/2}$ (as $N_0 = 1$) – (2)

   From (1) and (2):
   $N \geqslant N_h > 2^{h/2}$
   $N > 2^{h/2}$
   $\log_2(N) > \log_2(2^{h/2})$ ($\log_2$ on both sides)
   $\log_2(N) > h/2$ (formula simplification)
   $2 * \log_2(N) > h$ or $h < 2 * \log_2(N)$
   $h = O(\log(N))$ (the final conclusion)

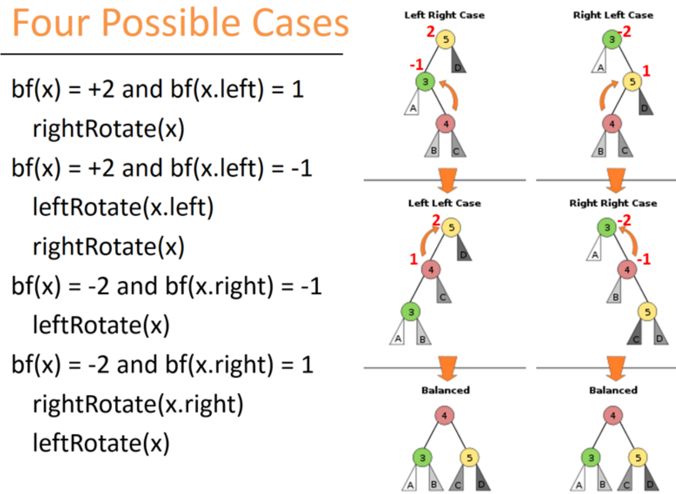
5. AVL Tree Invariant states that vertex *v* is said to be height balance if $|v.left.height - v.right.height| \leq 1$.
6. To quickly detect if a vertex *v* is height balanced or not, a balance factor $bf(v) = v.left.height - v.right.height$ can be used.

Tree Rotation



Properties:

1. `RotateRight(T)` can only be called if T has a left child, while `RotateLeft(T)` can only be called if T has a right child.
2. Note that Tree Rotation preserves the BST property i.e. Before rotation, $P \leq B \leq Q$. After rotation, notice that subtree rooted at B (if it exists) changes parent, but $P \leq B \leq Q$ does not change.
3. There are only 4 possible imbalance cases as shown below:



AVL Tree Operations

Operation	Implementation	Complexity
`insert(v)`	1. Insert v as in normal BST 2. Traverse up the AVL Tree from the insertion point back to the root 3. At every step, update the height and balance factor of the affected vertices 4. Stop at the first vertex that is out of balance 5. Use one of the four tree rotation cases to rebalance it	$O(\log N)$
`remove(v)`	1. Remove v as in normal BST 2. Traverse up the AVL Tree from the deletion point back to the root 3. At every step, update the height and balance factor of the affected vertices 4. For every vertex that is out-of-balance, use one of the four tree rotation cases rebalance (Note: This triggers not more than $O(\log N)$ times)	

Additional Notes:

1. `rank(V)` and `select(K)` operations can be implemented by storing the total number of vertices in each subtree, then recursively process the each subtrees until the conditions (where vertex < V in `rank(V)` and total vertices < K in `rank(K)`) are no longer satisfied. Pseudocodes for the two implementations are as follow:

```
int rank(node, V) {
    if (node.key == V)      return node.left.size + 1;
    else if (node.key > V)  return rank(node.left, V)
    else                    return node.left.size + 1 + rank(node.right, v);
}

int select(node, K) {
    int q = node.left.size;

    if (q + 1 == K)     return node.key;
    else if (q + 1 > K) return select(node.left, k);
    else                return select(node.right, k − q − 1);
}
```

Chapter 6: Graphs

<table>
<tr><th>Terminology</th><th>Definition</th></tr>
<tr><td>Graph</td><td>A graph is made up of vertices/nodes and edges/lines that connect those vertices.</td></tr>
<tr><td>Undirected/Birectional Graph</td><td>A graph that has no distinction between the two vertices associated with each edge or its edges may be directed from one vertex to another but not necessarily in the other direction</td></tr>
<tr><td>Weight Graph</td><td>A graph that has been assigned a weight to each edge, which represent numerical values associated with that connection.</td></tr>
<tr><td>Unweighted Graph</td><td>A graph where all edges have unit weight 1 or all edges have the same constant weight.</td></tr>
<tr><td>Simple Graph</td><td>A graph that has no self-loop edges (an edge that connects a vertex with itself) and no multiple or parallel edges (edges between the same pair of vertices) i.e. a graph that has up to only one edge between a pair of distinct vertices.</td></tr>
<tr><td>Incident</td><td>Edge (u, v) is said to be incident with its two end-point vertices u and v if it u and v are adjacent i.e. u and v are connected by edge (u, v).</td></tr>
<tr><td>Adjacent</td><td>Undirected Graph: Two vertices are called adjacent (or neighbour) if they are incident with a common edge i.e. a single edge connects the 2 vertices.

Directed Graph: Vertex u is adjacent to vertex v if and only if they are connected by an edge and the direction of the edge is from u to v i.e. v is not adjacent to u.

Two edges are called adjacent if they are incident with a common vertex i.e. both edges share a common vertex.</td></tr>
<tr><td>Degree</td><td>Undirected Graph: Degree of a vertex in an undirected graph is the number of edges incident with the vertex.

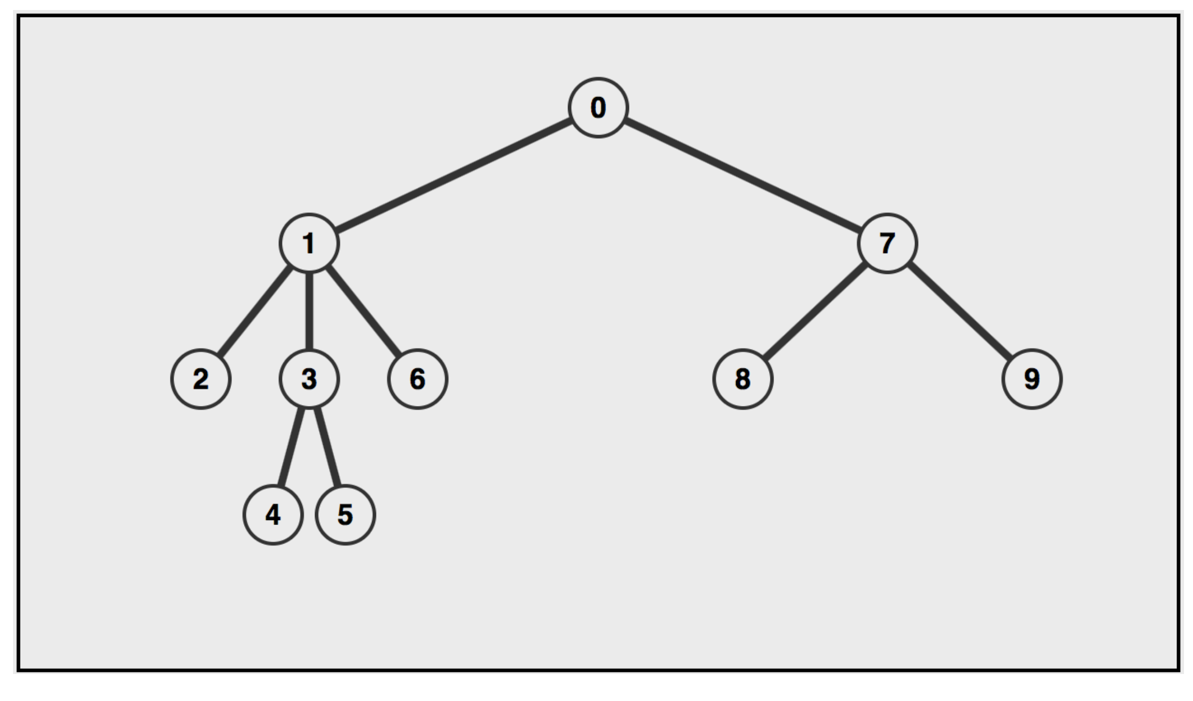
Directed Graph: Further classified into in-degree and out-degree. In-degree refers to the number edges coming into a vertex while out-degree refers to the number of edges going out of a vertex.</td></tr>
<tr><td>Isolated vertex</td><td>A vertex with a degree of 0 i.e. a vertex with no edges.</td></tr>
<tr><td>Subgraph</td><td>A subgraph of a graph is a smaller graph that contains subset of vertices and edges of the main graph.</td></tr>
<tr><td>Path</td><td>In an undirected graph, a path is a sequence of vertices $\{v_0, v_1, \ldots, v_{n-1}, v_n\}$ such that there's an edge between v_i and $v_{i+1} \forall i \in [0..n-1]$ along the path.</td></tr>
<tr><td>Simple Path</td><td>A path that contains no repeated vertex.</td></tr>
<tr><td>Cycle</td><td>A cycle is a path that starts and ends with the same vertex.</td></tr>
<tr><td>Acyclic Graph</td><td>An acyclic graph is a graph that contains no cycle.</td></tr>
<tr><td>Connected</td><td>An undirected graph is called connected if there is a path between every pair of distinct vertices.</td></tr>
<tr><td>Connected Component</td><td>An undirected graph C is called a connected component of the undirected graph G if:
1. C is a subgraph of G
2. C is connected
3. No connected subgraph of G has C as a subgraph and contains vertices or edges that are not in C (i.e. C is the maximal subgraph that satisfies the other two criteria)</td></tr>
</table>

Trees

Properties:

1. Tree is a connected graph with V vertices and $E = V - 1$ edges, acyclic, and has one unique path between any pair of vertices.
2. Usually a Tree is defined on undirected graph.
3. An undirected Tree contains trivial cycles but it does not contain non-trivial cycle.
4. A directed Tree is acyclic.
5. As a Tree only have $V - 1$ edges, it is usually considered a sparse graph.
6. Tree with one of its vertex designated as root vertex is called a rooted Tree.

Terms	Vertices	Results
Parents	0	None
	1	0
	2	1
	4	3
Children	0	{1, 7}
	1	{2, 3, 6}
	7	{8, 9}
Ancestors	4	{3, 1, 0}
	8	{7, 0}
Descendants	1	{2, 3, 4, 5, 6}
	7	{8, 9}
Subtree	1	{2, 3, 4, 5, 6}
Level	Level 0	0
	Level 1	{1, 7}
	Level 2	{2, 3, 6, 8, 9}
	Level 3	{4, 5}
Max Level	-	3



Other Special Graphs

Graph	Properties	Visualization
Complete	1. A graph where there is an edge between any pair of vertices. 2. Maximum number of edges is $\frac{V(V-1)}{2}$ or C_2^V i.e. $E = O(V^2)$ 3. Can be denoted with K_V. 4. Most dense simple graph.	
Bipartite	1. An undirected graph with V vertices that can be partitioned into two disjoint set of vertices of size m and n where $V = m + n$. 2. No edge between members of the same set. 3. Free from odd-length cycle. 4. Complete Bipartite Graph, i.e. all m vertices from one disjoint set are connected to all n vertices from the other disjoint set. 5. When $m = n = \frac{V}{2}$, such Complete Bipartite Graphs have $E = O(V^2)$. $\therefore Max\ Edge = \frac{V^2}{4} if\ V\ is\ even$ $Max\ Edge = \frac{V^2 - 1}{4} if\ V\ is\ odd$	

Directed Acyclic	1. A directed graph that has no cycle. 2. The maximum number of edges a Directed Acyclic Graph can have is $\frac{v(v-1)}{2}$. Adding any more edges will form bidirectional edges.	

Additional Notes:

1. For a general undirected connected simple graph of V vertices, the minimum number of edges is $V - 1$ (a tree) and the maximum number of edges is $\frac{V(V-1)}{2}$ or C_2^V (a complete graph).
2. For a general directed graph, the maximum number of edges is $2C_2^V$ or P_2^V.

<u>Graph Data Structures</u>

Data Structure	Implementation	Space Complexity
Adjacency Matrix	Adjacency Matrix (AM) is a square matrix where the entry `AM[i][j]` shows the edge's weight from vertex `i` to vertex `j`. For unweighted graphs, set a $unit\ weight\ = 1$ for all edge weights. An 'x' means that that vertex does not exist (deleted).	$O(V^2)$
Adjacency List	Adjacency List (AL) is an array of V lists, one for each vertex (usually in increasing vertex number) where for each vertex `i`, `AL[i]` stores the list of `i`'s neighbours. For weighted graphs, store pairs of (neighbour vertex number, weight of this edge) instead using Vector of Vector pairs: `vector<vector<pair<int neighborVertex,int edge>>> AL;`	$O(V + E)$

	Reason: 1. Pair is used due to the need to store pairs of information for each edge: (neighbor vertex number, edge weight) where weight can be set to 0 or unused for unweighted graph. 2. Vector of pairs is used due to vector's auto-resize feature. If there are k neighbours of a vertex, just add k times to an initially empty Vector of airs of this vertex (this Vector can be replaced with Linked List). 3. Vector of Vectors of Pairs is used for Vector's indexing feature, i.e. to enumerate neighbours of vertex u, use AL`[u]` to access the correct Vector of Pairs.	
Edge List	Edge List (EL) is a collection of edges with both connecting vertices and their weights. Usually, these edges are sorted by increasing weight. Can be implemented with a Vector of triples: `vector<tuple<int vertexU, int vertexV, int weight>> EL;`	$O(E)$

Operations

Operation	Adjacency Matrix	Adjacency List	Edge List
`CountV()`	Number of rows	Number of rows	Loop through the list and sum the number of unique vertices
`CountE()`	Loop through all the matrix and report AM`[u][j]` that is not zero	Sum the length of all V lists and divide the final answer by 2	Number of rows
`EnumNeighbours()`	Loop through all columns of AM`[u][j]` $\forall j \in [0..V-1]$ and report AM`[u][j]` that is not zero	Scan through AL`[u]`	Loop through the list and scan through the entry that contains u
`CheckEdge(u,v)`	Check if AM`[u][v]` is non zero	Check whether AL`[u]` contains vertex V or not	Check if EL contains (u, v)

Depth-First-Search

Algorithm Outline:

1. DFS starts from a distinguished source vertex s and uses recursion (an implicit stack) to order the visitation sequence as deep as possible before backtracking.
2. If DFS is at a vertex u and it has X neighbours, it will pick the first neighbour v_1 (usually the vertex with the lowest vertex number), recursively explore all reachable vertices from vertex v_1, and eventually backtrack to vertex u. DFS will then do the same for the other neighbours until it finishes exploring the last neighbour v_X and its reachable vertices.
3. DFS uses another array `p[u]` of size V vertices to remember the parent/predecessor/previous of each vertex u along the DFS traversal path. The predecessor of the source vertex, i.e. `p[s]` is set to -1 to say that the source vertex has no predecessor (as the lowest vertex number is vertex 0).
4. The sequence of vertices from a vertex u that is reachable from the source vertex s back to s forms the DFS spanning tree.
5. If the graph is cyclic, the previous 'try-all' strategy may lead DFS to run in cycle, so the basic form of DFS uses a Boolean array `status[u]` of size V vertices to distinguish between two states: visited and unvisited vertices. Only if vertex u is still unvisited, then DFS can visit vertex u. When DFS runs out of option, it backtrack to previous vertex (`p[u]`) as the recursion unwinds.

Properties:

1. Time complexity is $O(V+E)$ because:
   a. Each vertex is only visited once due to the fact that DFS will only recursively explore a vertex u if `status[u] = unvisited` i.e. an $O(V)$ operation.
   b. Each time a vertex is visited, all its k neighbours are explored and therefore after all vertices are visited and all E edges are examined i.e. an $O(E)$ operation as the total number of neighbours of each vertex equals to E.
2. To obtain an $O(V+E)$ time complexity, an adjacency list graph data structure should be used.

Breadth-First-Search

Algorithm Outline:

1. BFS starts from a source vertex s but it uses a queue to order the visitation sequence as breadth as possible before going deeper.
2. If BFS is at a vertex u and it has X neighbours, it will explore all the X neighbours. BFS will then do the same for each of the X neighbours until it finishes exploring the last reachable vertex.
3. BFS uses another array `p[u]` of size V vertices to remember the parent/predecessor/previous of each vertex u along the BFS traversal path. The predecessor of the source vertex, i.e. `p[s]` is set to -1 to say that the source vertex has no predecessor (as the lowest vertex number is vertex 0).
4. The sequence of vertices from a vertex u that is reachable from the source vertex s back to s forms the BFS spanning tree, which is equal to its SSSP spanning tree.
5. If the graph is cyclic, the previous 'try-all' strategy may lead BFS to run in cycle, so the basic form of BFS uses a Boolean array `status[u]` of size V vertices to distinguish between two states: visited and unvisited vertices. Only if vertex u is still unvisited, then BFS can visit vertex u.

Properties:

1. Time complexity is $O(V + E)$ because:
   a. Each vertex is only visited once as it can only enter the queue once i.e. an $O(V)$ operation.
   b. Each time a vertex is dequeued from the queue, all its k neighbours are explored and therefore all vertices are visited and all E edges are examined i.e. an $O(E)$ operation as the total number of neighbours of each vertex equals to E.
2. To obtain an $O(V + E)$ time complexity, an adjacency list graph data structure should be used.

DFS/BFS Applications

Application	**Logic**	**Implementation**
Detecting Cycle	The presence of at least one back edge shows that the traversed graph (component) is cyclic while its absence shows that at least the component connected to the source vertex of the traversed graph is acyclic.	Modify array `status[u]` to record three different states: 1. Unvisited: DFS has not reach vertex u before 2. Explored: DFS has visited vertex u, but at least one neighbour of vertex u has not been visited yet (DFS will go depth-first to that neighbour first) 3. Visited: all neighbours of vertex u have also been visited and DFS is about to backtrack from vertex u to vertex `p[u]`.

Print Traversal Path	Call DFS or BFS from source vertex to store the parent/predecessor/previous vertex of each vertex in array p and then call a function backtrack which takes in the array p and vertex of interest to print the traversal path from the source vertex to the vertex of interest.	Backtrack function can take the recursive form as follow:
		`void backtrack(vector<int> &p, int t) {`
		`    if (p[t] == -1) // if p[t] is source vertex`
		`        return 0;   // break recursion`
		`    else {`
		`        backtrack(p, p[t]);    // run backtrack on parent node`
		`        cout << t << " ";    // print current node`
		`    }`
		`}`
		
		Do note that a recursive implementation will result in a path from source vertex to vertex of interest. Eg. for a graph with path $1 \rightarrow 2 \rightarrow 3$, calling `DFS(1)` and `backtrack(3)` will print out `1 2 3`.

Reachability Test	Call DFS or BFS from source vertex to store the parent/predecessor/previous vertex of each vertex in array p and then check status of vertex of interest in the `status` array. If `status` of vertex of interest is visited, vertex of interest can be reached from source vertex.	

<table>
<tr><th>Application</th><th>Logic</th><th>Implementation</th></tr>
<tr><td>Connected Components</td><td>Enumerate all vertices that are reachable from a source vertex in an undirected graph by calling DFS or BFS and enumerate all vertices that have visited status. This will form the first group of connected component.
Repeat the process for vertices with unvested status to find the number of connected components in the graph.

Note: This is an $O(V+E)$ and not $O(V^2+VE)$ complexity as the V and E is used loosely i.e. V and E are for each individual connected component.</td><td>

```
void connected(vector<int> &status, vector<int> &vertex) {
    int CC = 0;

    for (auto &i : vertex)
        if (status[i] == 0) {     // Using 0 as unvisited
            CC++;
            DFS(i);      // Call DFS for unvisited vertex
        }
    cout << CC << endl;
}
```

</td></tr>
<tr><td>Topological Sort</td><td>Topological sort of a DAG is a linear ordering of the DAG's vertices in which each vertex comes before all vertices to which it has outbound edges.

Every DAG has at least one but possibly more topological sorts/ordering.</td><td>The DFS version requires just one additional line compared to the normal DFS and is basically the post-order traversal of the graph.

The BFS version is based on the idea of vertices without incoming edge and is also called as Kahn's algorithm.</td></tr>
</table>

Other Applications

<table>
<tr><th>Application</th><th>Logic</th><th>Implementation</th></tr>
<tr><td>Connected Components (Flood Fill)</td><td>Given a 2D array where cells containing the same element are considered connected graph, a recursive flood fill algorithm can be used to determine the number of connected component.

The recursive flood fill algorithm check for each cell’s horizontal and vertical neighbouring cells.

It terminates if the cell index exceeds the size of the array, or if the cell contains a different element. Otherwise, it fills the current cell with a different character before running the algorithm on its neighbours.</td><td>

```
int r[] = {0, 0, 1, -1};
int c[] = {1, -1, 0, 0};
void floodFill(int row, int col) {
    // Cell exceeds the range of 2D array
    if (row < 0 || row >= R || col < 0 || col >= C)return;
    // Cell does not contain the same character
    else if (array[row][col] == '#')return;
    else {
        // Change current cell to different character
        array[row][col] = '#';
        // Check for cells vertically and horizontally
        for (int k = 0; k < 4; k++)
            floodFill(row + r[k], col + c[k]);
    }
}
```

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

Chapter 7: Single Source Shortest Path

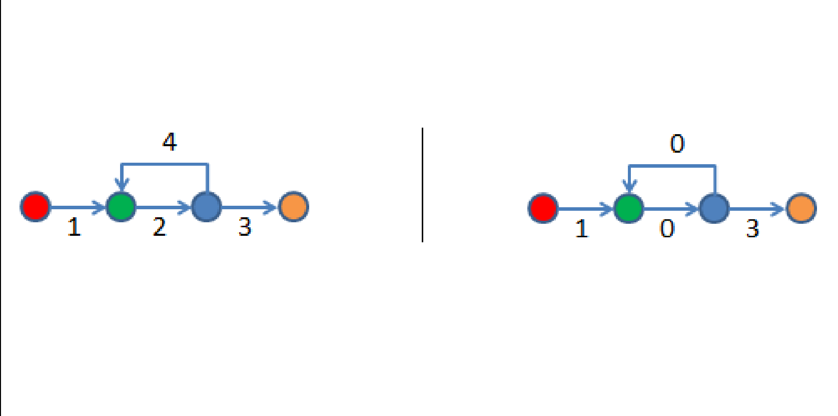
Algorithm Outline: To find the shortest paths weights from a particular single-source vertex to all other vertices in a directed weighted graph (if such paths exist) denoted by $\delta(s,u)$ where s is the source vertex and u is the destination vertex:

1. Input consists of:
    a. A directed graph $G(V,E)$ where V is the vertex and E is the edge
    b. A source vertex $s \in V$
2. Output consists of:
    a. An array/vector D of size V to store the distance:
        i. Initially D[u]$=$ 0 if $u = s$; otherwise D[u]$=\infty$ where ∞ can be an extremely large value (eg. INT_MAX) or a placeholder value (eg. −1, if no negative weights).
        ii. D[u] decreases as shorter paths are found
        iii. D[u] $\geq \delta(s,u)$ throughout the execution of SSSP algorithm
        iv. D[u] $= \delta(s,u)$ at the end of SSSP algorithm
    b. An array/vector p of size V to store the parent/predecessor/previous vertex
        i. p[u] = the predecessor on best path from source s to u
        ii. p[u] = $NULL$
3. The main operation for all SSP algorithm is the `relax(u, v, weight(u, v))` operation, which essentially traverses down an edge and updating the tentative shortest distance, as shown in pseudo code below:

```
relax(u, v, weight (u, v)
      if D[v] > D[u] + weight(u, v) // if shortest path can be shortened
           D[v] = D[u] + weight(u, v) // 'relax' this edge; update shortest path
           p[v] = u // update the predecessor
```

Properties:

1. Shortest Paths are Simple Paths if $G(V, E)$ has no negative weight cycle:

<table>
<tr>
<td>
Proof by Contradiction:
1. Suppose the shortest path p is not a simple path
2. Then p must contains one (or more) cycle(s) (by definition of non-simple path)
3. Suppose there is a cycle c in p with positive weight (e.g. green → blue → green on the left image)
4. If c is removed from p, then there will be a shorter 'shortest path' than the shortest path p.
5. This contradicts 1, thus p must be a simple path.
6. Even if c is actually a cycle with zero (0) total weight — it is possible according to the assumption: no negative weight cycle (see the same green → blue → green), c still can be removed from p without increasing the shortest path weight of p.
7. In conclusion, p is a simple path (from point 5) or can always be made into a simple path (from point 6)
</td>
<td>


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

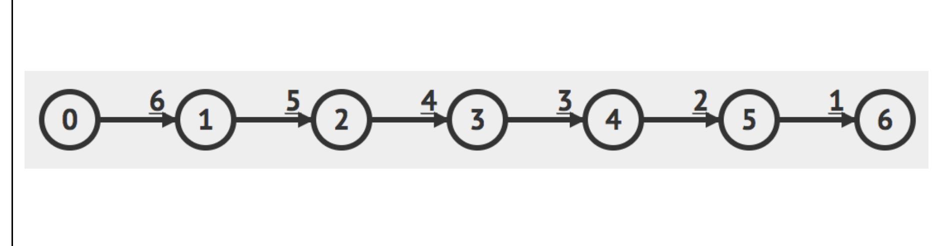
Bellman Ford's Algorithm

Algorithm Outline:

1. Bellman Ford's Algorithm has the following pseudo code:

```
for i = 1 to |V|-1 // O(V) here, so O(V×E×1) = O(V×E)
    for each edge(u, v) ∈ E // O(E) here, e.g. by using an Edge List
        relax(u, v, w(u, v)) // O(1) here
```

2. From the pseudo code, the algorithm runs in a time complexity of $O(VE)$.
3. The optimized form of Bellman Ford's Algorithm runs in $O(KE)$ where K is the number of iterations before no further updates are done to the path distance of each vertices from the source vertex.
4. Bellman Ford's Algorithm ensures shortest path from source vertex to all the vertices in the graph after it terminates:

Proof by Induction: 1. Initially, $D[v_0] = \delta(s, v_0) = 0$, as v_0 is just the source vertex s 2. After 1 pass through E, we have $D[v_1] = \delta(s, v_1)$ 3. After 2 pass through E, we have $D[v_2] = \delta(s, v_2)$ 4. After k pass through E, we have $D[v_k] = \delta(s, v_k)$ 5. When there is no negative weight cycle, the shortest path p is a simple path (see Theorem 1), thus the last iteration should be iteration $\|V\| - 1$ 6. After $\|V\| - 1$ pass through E, we have $D[v_{\|V\|-1}] = \delta(s, v_{\|V\|-1})$, regardless the ordering of edges in E	

5. Bellman Ford's Algorithm can be used to detect if the input graph contains at least one negative weight cycle reachable from the source vertex s by using the corollary of theorem in point 4: If at least one value `D[v]` fails to converge after $|V| - 1$ passes, then there exists a negative-weight cycle reachable from the source vertex s.

Dijkstra's Algorithm

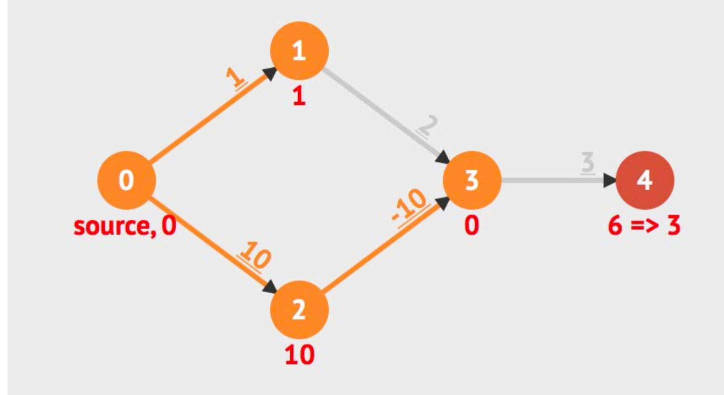
Algorithm Outline:

1. The algorithm maintains a set S of solved vertices whose final shortest path weights have been determined. Initially $S = \{s\}$, the source vertex s only.
2. It repeatedly selects vertex u in $\{V \backslash S\}$ with the minimum shortest path estimate, adds u to S, and relaxes all outgoing edges of u
3. This entails the use of a Priority Queue as the shortest path estimates keep changing as more edges are processed.
4. The choice of relaxing edges emanating from vertex with the minimum shortest path estimate first is greedy, i.e. use the "best so far.

Properties:

1. Dijkstra's algorithm runs with a time complexity of $O((V + E) \log V)$:
   a. Each vertex will only be extracted from the Priority Queue (PQ) once. As there are V vertices, this will be done maximum $O(V)$ times. `ExtractMin()` operation runs in $O(\log V)$ whether the PQ is implemented using a Binary Min Heap or using a balanced BST. Therefore this part is $O(V \log V)$.
   b. Every time a vertex is processed, its neighbours are relaxed. In total, E edges are processed. By relaxing $edge(u, v)$, as there is a need to decrease `D[v]`, call the $O(\log V)$ `DecreaseKey()` operation in Binary Min Heap (harder to implement as C++ STL `priority_queue` does not support this operation efficiently yet) or simply delete the old entry and then re-insert a new entry in balanced BST (which also runs in $O(\log V)$, but this is much easier to implement). Therefore, this part is $O(E \log V)$.
   c. Overall, Dijkstra's algorithm runs in $O(V \log V + E \log V) = O((V + E) \log V)$ time.

2. When input graph has at least one negative weight edge, Dijkstra's Algorithm can produce wrong answer. This is because even though the shortest path for the vertices that have negative edges will be updated as per normal, the neighbour vertices of the affected vertices will not have their paths updated as the affected vertices are no longer processed.



Modified Dijkstra's Algorithm

Algorithm Outline:

1. Key modification done is the use of Lazy Update as `DecreaseKey()` operation in priority queue.
   a. Lazy Update leave the outdated/weaker/bigger-valued information in the minimum priority queue instead deleting it straight-away.
   b. Since the items are ordered from smaller value to bigger values in the minimum priority queue, it is guaranteed that the most up-to-date (smallest) item will be encountered first.

Properties:

1. On non-negative weighted graphs, the behaviour is exactly the same as the original Dijkstra's and has a time complexity of $O((V+E)\log V)$.
2. On graph with at least one negative weight edge but no negative weight cycle, the time complexity can be greater than $O((V+E)\log V)$.

Single Source Shortest Path Algorithms

Graph Type	Algorithm	Time Complexity	Remarks
Unweighted Graph	Breadth-First-Search	$O(V+E)$	Simple modifications need to be made in order for BFS to solve the unweighted version of the SSSP problem: 1. First, change Boolean array visited to integer array D. 2. At the start of BFS, instead of setting `visited[u] = false`, set `D[u] = 1e9` (a large number to symbolise $+\infty$ or even -1 to symbolise 'unvisited' state, but we cannot use 0 as `D[0] = 0`) $\forall u \in V\backslash\{s\}$; Then we set $D[s] = 0$ 3. We change the BFS main loop to `if (D[v] = 1e9) { D[v] = D[u]+1 ... } // v is 1 step away from u`
Graph without Negative Edges	Dijkstra's Algorithm	$O((V+E)\log V)$	
Graph without Negative Weight Cycle	Modified Dijkstra's Algorithm	$\geq O((V+E)\log V)$	
Trees (Weighted/Unweighted)	Depth-First-Search Breadth-First-Search	$O(V)$	In a Tree, there is only one unique and acyclic path that connects two distinct vertices. Thus the unique path that connects the source vertex s to any another vertex $u \in V$ is actually also the shortest path. This also gives rise to the $O(V)$ time complexity.
Directed Acyclic Graph	Dynamic Programming	$O(V+E)$	Compute the topological order of the graph and relaxes the outgoing edge of vertices listed in the topological order. This can be achieved with a time complexity of $O(V+E)$. This is also known as one-pass Bellman Ford's algorithm as it is essentially Bellman Ford's algorithm being executed only once with the topological order instead of $V-1$ times.

All Pair Shortest Path Algorithms

Graph Type	Algorithm	Time Complexity	Remarks
Any	Dijkstra's Algorithm	$O(V(V+E)\log V)$	Execute Dijkstra's algorithm V times to find the shortest path between all vertices.

	Floyd-Warshall Algorithm	$O(V^3)$	Using an Adjacency Matrix where `AM[i][j]` represents the distance between vertex i and vertex j, the shortest path `AM[i][j] = min(AM[i][j], AM[i][k] + AM[k][j])` where k = [0, 1, 2, 3 … N) as shown below:
			`for (int k = 0; k < V; k++)`
			`    for (int i = 0; i < V; i++)`
			`        for (int j = 0; j < V; j++)`
			`            if (AM[i][j] < AM[i][k] + AM[k][j])`
			`                AM[i][j] = AM[i][k] + AM[k][j];`
			Effectively, the three for-loop algorithm calculates the distance between vertex i and vertex j by checking if vertex i and vertex j has a common neighbour (eg. vertex k) and calculate the distance between vertex i to vertex k and vertex k to vertex j. If the new distance is smaller than the existing distance in `AM[i][j]`, `AM[i][j]` gets updated with the new distance.

Additional Notes:

1. Both Floyd-Warshall and Dijkstra's algorithm can also be used to determine kth-shortest path from i to j. This can be achieved by:
    a. Choosing an $edge(s,t)$. Time complexity for this step is $O(E)$.
    b. If using Floyd-Warshall algorithm with time complexity of $O(V^3)$:
        i. Obtain the shortest path from i to s and from t to j using the Adjacency Matrix from the resulting Floyd-Warshall algorithm.
    c. If using Dijkstra's algorithm with time complexity of $O((V+E)\log V)$.
        i. Determine the shortest path from i to s.
        ii. Determine the shortest path from t to j (this requires graph with reverse edges).
    d. The total distance can then be calculated by $distance(i,s) + weight(s,t) + distance\ (t,j)$.
    e. Ensure that distance calculated is less than distance of $k-1$ shortest path.

Appendices

DFS and BFS Implementation

```
void dfs(int &vertex, vector<bool> &visited, vector<vector<int>> AdjList) {
    visited[vertex] = true;     // Set status of current vertex to visited
    cout << vertex << endl;     // Prints current vertex

    // For all neighbours of current vertex
    for (auto &i : AdjList[vertex])
        // Check if neighbour vertex has been visited
        if (!visited[i])
            // If not visited, run dfs on neighbour vertex
            dfs(i, visited);
}

void bfs(int &source, vector<bool> &visited, vector<vector<int>> AdjList) {
    queue<int> q;

    visited[source] = true;     // Set status of source vertex to visited
    q.push(source);             // Push source vertex into queue

    // While there are still vertices in the queue to be processed
    while (!q.empty()) {
        int buffer = q.front();
        cout << buffer << endl; // Print current vertex to be processed
        q.pop();                // Remove current vertex from the queue

        // For all neighbours of current vertex
        for(auto &i : AdjList[buffer])
            // Check if neighbour vertex has been visited
            if(!visited[i]) {
                // If not visited, set status neighbour to be visited and push neighbour into the queue
                visited[i] = true;
                q.push(i);
            }
    }
}
```

Topological Sort Implementation (DFS Method)

```
class Graph {
    int V;    // No. of vertices'

    // Pointer to an array containing adjacency listsList
    list<int> *adj;

    // A recursive function used by topologicalSort
    void topologicalSortUtil(int v, bool visited[], stack<int> &Stack) {
        // Mark the current node as visited.
        visited[v] = true;

        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); ++i)
            if (!visited[*i])
                topologicalSortUtil(*i, visited, Stack);

        // Push current vertex to stack which stores result
        Stack.push(v);
    }
public:
    Graph(int V) {   // Constructor
        this->V = V;
        adj = new list<int>[V];
    }

    // function to add an edge to graph
    void addEdge(int v, int w) {
        adj[v].push_back(w); // Add w to v’s list.
    }
```

```
    // // The function to do Topological Sort. It uses recursive topologicalSortUtil()
    void topologicalSort() {
        stack<int> Stack;

        // Mark all the vertices as not visited
        bool *visited = new bool[V];
        for (int i = 0; i < V; i++)
            visited[i] = false;

        // Call the recursive helper function to store Topological
        // Sort starting from all vertices one by one
        for (int i = 0; i < V; i++)
            if (visited[i] == false)
                topologicalSortUtil(i, visited, Stack);

        // Print contents of stack
        while (Stack.empty() == false) {
            cout << Stack.top() << " ";
            Stack.pop();
        }
    }
};
```

Topological Sort Implementation (BFS Method/Kahn's Algorithm)

```
class Graph {
    int V;    // No. of vertices'

    // Pointer to an array containing adjacency listsList
    list<int> *adj;

public:
    Graph(int V) {   // Constructor
        this->V = V;
        adj = new list<int>[V];
    }

    // function to add an edge to graph
    void addEdge(int u, int v) {
        adj[u].push_back(v);
    }

    // prints a Topological Sort of the complete graph
    void topologicalSort() {
        // Create a vector to store indegrees of all vertices. Initialize all indegrees as 0.
        vector<int> in_degree(V, 0);

        // Traverse adjacency lists to fill indegrees of vertices. This step takes O(V+E) time
        for (int u=0; u<V; u++) {
            list<int>::iterator itr;
            for (itr = adj[u].begin(); itr != adj[u].end(); itr++)
                in_degree[*itr]++;
        }

        // Create an queue and enqueue all vertices with indegree 0
        queue<int> q;
        for (int i = 0; i < V; i++)
            if (in_degree[i] == 0)
                q.push(i);

        // Initialize count of visited vertices
        int cnt = 0;
```

```
        // Create a vector to store result (A topological ordering of the vertices)
        vector <int> top_order;

        // One by one dequeue vertices from queue and enqueue adjacents if indegree of adjacent becomes 0
        while (!q.empty()) {
            // Extract front of queue (or perform dequeue) and add it to topological order
            int u = q.front();
            q.pop();
            top_order.push_back(u);

            // Iterate through all its neighbouring nodes of dequeued node u and decrease their in-degree by 1
            list<int>::iterator itr;
            for (itr = adj[u].begin(); itr != adj[u].end(); itr++)

                // If in-degree becomes zero, add it to queue
                if (--in_degree[*itr] == 0)
                    q.push(*itr);

            cnt++;
        }

        // Check if there was a cycle
        if (cnt != V)
        {
            cout << "There exists a cycle in the graph\n";
            return;
        }

        // Print topological order
        for (int i=0; i<top_order.size(); i++)
            cout << top_order[i] << " ";
        cout << endl;
    }
};
```

Bellman Ford's Algorithm Implementation

```
void BellmanFord() {
    int V, E, s, a, b, w;
    vector<vector<pair<int, int>>> AdjList;

    cin >> V >> E >> s;

    AdjList.assign(V, vector<pair<int, int>>()); // assign blank vectors of pair<int, int>s to AdjList
    for (int i = 0; i < E; i++) {
        cin >> a >> b >> w;
        AdjList[a].push_back(pair<int, int> (b, w));
    }

    // Bellman Ford routine
    vector<int> dist(V, INF);
    dist[s] = 0;
    for (int i = 0; i < V - 1; i++)  // relax all E edges V-1 times, overall O(VE)
        for (int u = 0; u < V; u++)                        // these two loops = O(E)
            for (int j = 0; j < (int)AdjList[u].size(); j++) {
                pair<int, int> v = AdjList[u][j];        // we can record SP spanning here if needed
                dist[v.first] = min(dist[v.first], dist[u] + v.second);         // relax
            }

    bool hasNegativeCycle = false;

    for (int u = 0; u < V; u++)                          // one more pass to check
        for (int j = 0; j < (int)AdjList[u].size(); j++) {
            pair<int, int> v = AdjList[u][j];
            if (dist[v.first] > dist[u] + v.second)                  // should be false
                hasNegativeCycle = true;     // but if true, then negative cycle exists!
        }
    printf("Negative Cycle Exist? %s\n", hasNegativeCycle ? "Yes" : "No");

    if (!hasNegativeCycle)
        for (int i = 0; i < V; i++)
            printf("SSSP(%d, %d) = %d\n", s, i, dist[i]);

}
```

Dijkstra's Algorithm Implementation

```
void Dijkstra() {

    int n; // Number of vertices
    int s; // Source vertex

    vector<int> dist(n, INF);
    dist[s] = 0;

    set<pair<int, int>> pq;
    pq.insert({0, s});
    for (u = 0; u < n; u++) // O(V log V) already
        if (u != s)
            pq.insert({INF, u});

    while (!pq.empty()) {
        pair<int, int> front = *pq.begin(); // this is the min, O(log V)
        pq.erase(pq.begin()); // erase / extract min, O(log V), twice

        d = front.first; // this value is actually the same as dist[u]
        u = front.second;

        for (auto &v_weight : AL[u]) { // for all neighbors of vertex u, O(k)
            v = v_weight.first;
            w = v_weight.second;
            if (dist[u]+w < dist[v]) { // if can relax this edge, O(1)
                // update/decrease PQ value :O
                pq.erase(pq.find({dist[v], v})); // delete the old one, O(log V) twice (reason negative edge is not allowed)
                dist[v] = dist[u]+w; // O(1)
                pq.insert({dist[v], v}); // delete the old one, O(log V)
            }
        }
    }
}
```

Modified Dijkstra's Algorithm Implementation

```
void ModifiedDijkstra() {

    int n; // Number of vertices
    int s; // Source vertex

    // compute the shortest path values, using Dijkstra's
    vector<int> dist(n, INF);
    dist[s] = 0;

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push({0, s});

    while (!pq.empty()) {
        pair<int, int> min = pq.top();
        pq.pop(); // O(1) + O(log V)

        d = min.first;
        u = min.second;

        // This prevents the degeneration of Modified Dijkstra's into a kind of Bellman Ford's
        if (dist[u] < d)
            continue;

        for (auto &v_weight : AL[u]) {
            v = v_weight.first; w = v_weight.second;
            if (dist[u]+w < dist[v]) { // if can relax this edge
                dist[v] = dist[u]+w;
                pq.push({dist[v], v}); // O(log V), insert the new pair, with lower dist[v]
            }
        }
    }
}
```