Using the List-Interface: using import.java.util.\*

* List<Obj> list = new ArrayList<Obj>();
* List a = new ArrayList();
* List b = new LinkedList();
* List c = new Vector();
* List d = new Stack ();

Concurrent: if it can support two or more actions in progress at the same time. Multi-threading.

Cloud–platform: The hardware and operating environment of a server in an Internet-based datacenter. The system software components (OS, virtual machine monitor, etc.) are called the "cloud stack."

Dynamic Programming: simplifying a complicated problem by breaking it down into simpler sub-problems in a [recursive](https://en.wikipedia.org/wiki/Recursion) manner.

Difference in Interface and Abstract class.

Difference in Implement and extend

Interface: A class with methods that have no implementations.

You implement the Interface with the keyword “Implement” in a class.

The key with the interface is that when you implement the interface, you must include all functions inside the class that implements the interface.

Abstract: It can be a normal class, but must be named abstract, and if abstract then it must contain AT LEAST 1 abstract method with no implementation inside that method. You use the abstract class with the keyword “extend”. And now you can use any function from the abstract class without having to create the object of that class.

Difference between Array and ArrayList.

* Array:
  + Simple fixed sized arrays that we create in Java.
  + Is a fixed size data structure.
  + Can contain both primitive and data types.
* ArrayList<>:
  + Dynamic sized arrays that implement list interface in Java.
  + Part of a collection framework in Java.
  + Has a set of methods to access elements and modify them.
  + Not a fixed sized data structure.
  + Only supports Object entry’s, not primitive types.

Define enum.

* Enumerations serve the purpose of representing a group of named constants in a programming language.
* Enums are used when we know all possible values at compile time.
* It is not necessary that the set of constants in an enum type stay fixed for all time.

Data Structures:

**Performance:**

* **Time:** Number of processes, operations.
* **Space:** Memory needed by code to store info, disk space needed for persistent storage.
* **Network:** Bandwidth code uses to pass info to clients or machines.
* **Complexity:** 
  + A measure of how resource requirements change as the input size gets larger.
  + Higher complexity effects performance.
  + Measured by worst case.
* **Arithmetic Operations:**
  + Read, assign, write, test.

**Big-O Notation:**

* **O(1) :** When complexity doesn’t change no matter what input size.
* **O(N):** If time taken when the Algorithm increases linearly when N increases.
* **O(N^2):** If time taken in the algorithm increases quadratically when N increases.
* **O(logN):** If it takes half the time to complete.

**LinkedList:**

* **Complexity:**
  + Insert-end-of-List: O(N) Traverse through entire list to insert element.
  + Insert-beginning-of-list: O(1) Insert right at the head.
  + Find element: O(N) Traverse each node and compare for element in search.
  + Delete 1st element: O(1) Delete head.
  + Delete random element O(N): Traverse through elements to find the one to delete.
* Using import java.util.\*
* Elements are linked to, or references to the next element, they chained together.
* LinkedList<T> Standard Java library.
  + Starts: ->Head, Ends: Tail -> null.
  + Each node contains data, nextNode
  + Node class. getNext(), setNext(), getData()
  + Pop(): Returns the first (head) value in the list.
* **Use LinkedList for:**
  + When having a large number of insert/delete operations.
  + When you have no idea how large the list might be.
* **Use Arrays for:**
  + When read operations need to be extremely fast.
  + When needing random access to elements.

**Stacks:**

* **Complexity:**
  + Push() : O(1)
  + Pop(): O(1)
  + isEmpty(): O(1) (With the helper of getSize() else is O(N))
  + isFull(): O(1)
* Best data structure for a stack is a LinkedList.
* **Stack:**
  + Last element added is the first one accessed. (LIFO)
  + Top(): Focused on one end of the stack .
  + Push(): Adds new element to the top of stack.
  + Pop():Remove element from top of stack.
  + Peek():View top element of stack.
  + isEmpty():Helper method that checks if its empty.
  + isFull():Helper method that checks if its full.
  + getSize(): The number of elements present in stack.
* **Use Stacks for:**
  + Undo, redo operations.
  + Back button in browser.
  + Holds memory for recursive calls.
  + Translating infix notation for expressions to postfix.
* **Ex:**
* **Find min value in a stack.** 
  + You need 2 stacks. minStack, regStack.

**Queues:**

* **Complexity:** 
  + Enqueue: **O(1)**
  + Dequeue: **O(1)**
  + isEmpty: **O(1)**
  + isFull: **O(1)**
* Elements added to end of queue, removed from beginning of queue. (FIFO)
* Enqueue: adds the element to the queue.
* Dequeue: removes element from beginning of the queue.
* You can peek() for the first element in the queue.
* Offer(): Offers element to queue if there’s space.

* **Circular Queue:** 
  + Can be used as an array, last element wraps around to the first element.
  + Contains special value: head = -1, to denote emptyList.
  + Only one element when head and tail are in the same index.
* **Can be used in:**
  + Customer service hotline.
  + Jobs to be printed.
  + In order processing systems. (transactions in atm, deposit first then you can withdraw.)
* **Ex.**
* **Implement queue using 2 stacks.** 
  + **Complexity:**
    - O(M): number of operations performed in the queue.
  + Create **Stack<T> forwardSta**ck to push enqueues.
  + Create **Stack<T> reverseStack**. Holds elements in reverse from f **Stack<T> forwardStack.**
  + Pop all elements from **Stack<T> forwardStack** then push into reverseStack, then dequeue.
  + We operate the full enqueue process via the **Stack<T> forwardStack**.

**Sorting Algorithms:**

* **SelectionSort:**
  + **Complexity:** 
    - Comparisons: **O(N^2)**
    - Swaps: **O(N)**
  + Selects one element for every iteration, compare with every element in list, find the smallest, then swap with first in the list, continue.
  + Move to the next index, repeat.
  + Once first element is in place, we don’t go back to compare it again.
  + Not stabled, if 2 equal values are in list, the order can get rearranged.
  + Not adaptive, can’t break out of loop even if the list is sorted already.
* **BubbleSort:**
  + **Complexity:**
    - **O(N^2)**: Swaps
    - **O(N^2)**: Comparisons
  + Each iteration, every element is compared to its neighbor.
  + Stable, doesn’t require iteration if the list is sorted already, break out of the loop if no swaps have been done.
  + Horrible, processes additional passes over all elements.
* **InsertionSort:**
  + **Complexity:**
    - **O(N^2):** Worst case if in descending and we want ascending.
  + Start with sorted sub-list, size 1.
  + Insert new element, compare to its neighbor, swap until it’s in place.
  + Stable sort, if no swap, you can break out of inner loop.
* **ShellSort:**
  + **Complexity:**
    - **O(N) or O(N^2):** Depends on increment value.
    - **O(N^3/2):** For increments (2^k) -1, for k = 1, 2, 3…
  + Takes in InsertionSort and processes it differently.
  + Partitions original list into sublists, sublists made of elements separated by an increment.
  + Each sublist is then sorted using insertionSort.
  + Increment slowly reduced till its 1.
  + **Ex.**
  + If increment = 3.
    - Take every element in indexes multiple of 3 ; 0, 3, 6, 9 … n, and forms the first sublist.
    - Use insertionSort to sort the sublist.
    - Next sublist with increment 3 starts at index 1 ; 1, 4, 7, 10 … n.
    - Function would take (List[], startIndex, increment)
* **MergeSort:**
  + **Complexity:**
    - **O(NlogN):**
  + Algorithm: Divide and Concur – recursion based.
  + Sub-divides the list, until each list is 1 element.
  + Helper methods:
    - Split() : breaks down list into smaller and smaller list.
      * Takes in original list, firstHalfList, secondHalfList
      * Copies over elements from original list to subList.
    - Merge(): Takes in 2 sorted list, firstHalf, secondHalf, and combines them into 1 sortedList.
  + The idea:
    - Split the input in half.
    - Sort each half by recursively using the same process.
    - Merge the sorted halves back together.
  + Not adaptive. Its in a recursive call and can’t break out.
  + Stable: 2 equal elements will maintain the order when sort is complete.
* **QuickSort:**
  + **Complexity:**
    - **O(N):**
  + Not based on the length, its based on a pivot.
  + Pivot splits the list into 2, values < pivot on left, values > pivot on right.
  + Contains 2 main methods.
    - Partition(): Find pivot, rearrange, values < pivot on left, values > pivot on right.
    - quicksort(): Uses partition method to find pivot.
  + Not stable sort.

**Binary Search:**

* **Complexity:**
  + **O(logN)**
* List must be sorted to work.
* We have a key value we are in search for, compare to the midpoint value of the list.
* If key > midpointVal, then we knock out the right half of the list.

**Binary Trees:**

* Non-Linear Data structure made up of nodes.
* Normally used to represent hierarchical information.
* Each node can have 0, 1, 2 children.
* Contain a Root, edges, siblings(same level) and leafs.
* Node class in a tree:
  + Node<T> rightChild
  + Node<T> leftChild
  + Node(T data)
* Helper methods:
  + getData()
  + getLeftChild()
  + setLeftChild()
  + getRightChild()
  + setRightChild()

**Breadth First SearchTraversal (BFS):**

* All siblings must be traversed before moving to the next level the of tree.
* Starts from Root, move to next level, start from left, once cleared, move to next level, start from left.
* Iterative algorithm.
* **Implementation:**
  + Start from root, add it to the queue.
  + Set up a loop, to dequeue and process the node,
  + Add its children to the queue, add the left child first, then right child.

**Depth First Traversal (DFT):**

* Goes deep, leaf node of the tree, before moving up.
* Uses recursion.
* **Pre-order:** Process root, then left sub tree, then right subtree, stay on left side.
  + print(root); postOrder(root.getLeftChild()); postOrder(root.getRightChild());
* **In-order:** From leaf of leftSubtree, to root, to leaf of right subtree, then its parent.
  + postOrder(root.getLeftChild()); print(root); postOrder(root.getRightChild());
* **Post-order:** Process left sub-tree, right sub-tree then root.
  + postOrder(root.getLeftChild()); postOrder(root.getRightChild()); print(root);

A

/ \

B C

/ \

D E

/ \ \

F H G

* Pre-order: A->B->C->D->F->H->E->G
* In-order: B->A->F->D->H->C->E->G
* Post-order: B->F->H->D->G->E->C->A

**Binary Search Tree (BST):**

* **Complexity:**
  + Insertion: **O(logN);** if Skewed shaped trees: **O(N);**
  + Lookup: **O(logN);** if Skewed shaped trees: **O(logN);**
* Used for fast insertion, fast lookup
* Insertion value is compared with root, then decides if to go in left/right sub tree.
* **Ex.**
* **Find min value in binary search tree.**
  + Traverse left sub tree of every node till you hit the leaf.
  + If no left child, then that’s the min value.
  + Recursive after base case: return minimumValue(head.getLeftChild());
* **Ex.**
* **Find max depth in binary search tree.**
  + Furthest distance away from the root.
  + Create some Objects to design the tree.
    - Node<Character> a = new Node<>('A');
    - Node<Character> b = new Node<>('B');
    - Node<Character> c = new Node<>('C');
  + Set children to parent nodes.
    - a.setLeftChild(b);
    - a.setRightChild(c);
    - c.setLeftChild(d);
    - c.setRightChild(e);
  + **BASE CASE**
    - if (root.getLeftChild() == null&& root.getRightChild() == null) { return 0;}
  + **Find max depth on left/right and add 1 for current depth**
    - int leftMaxDepth = 1 + maxDepth(root.getLeftChild());
    - int rightMaxDepth = 1 + maxDepth(root.getRightChild());
    - return Math.max(leftMaxDepth, rightMaxDepth);
* **Ex.**
* **Print nodes within range binary search tree.**
  + Pass in (Node>Integer> root MIN, MAX) indicating the range.
  + BASE CASE, nothing to do for a null root.
  + If the range low value is less than the current node, run the operation on left subtree.
  + Check the node value to see if its within the range, if yes, print.
  + If the range high value is greater than the current node, run the operation on the right subtree.
* **Ex.**
* **Check if a BST is a BST.**
  + Check if constraints are satisfied.
  + if (root == null) { return true; }
  + if (root.getData() <= min || root.getData() > max) { return false; }
  + return isBinarySearchTree(root.getLeftChild(), min, root.getData())
  + && isBinarySearchTree(root.getRightChild(), root.getData(), max);
  + Pass in (Node>Integer> root MIN, MAX) indicating the range for sub tree.
  + BASE CASE: root == null, return true.
  + If node lies outside the range then BST constraint has been violated and we return false.
  + Check the left and right subtrees to see if they’re valid search trees. Note how the range for the checks change.
  + For left subtree the current nodes value should be max.
  + For right subtree the current nodes value should be min.

**Heaps: Highest Priority.**

* **Priority Queue:**
  + **Common operations:**
    - Insert.
    - Access.
    - Remove.
  + **An Array or List:**
    - **Insert:**
      * UnOrdered: O(1) : Anywhere in the unsorted list or array.
      * Ordered: O(N) : Requires finding the right position for element based on priority.
    - **Accessing:**
      * UnOrdered: **O(N):** Requires going through all elements.
      * Ordered**: O(1):** Highest priority is on the root.
    - **Remove:**
      * Unordered: **O(N):** Requires going through all elements.
      * Ordered: **O(1):** Highest priority is on the root.
  + **RECALL: Balanced Binary Search tree can’t have a difference of height 1.**
    - Insertion: **O(logN)**
    - Access: **O(logN)**
    - Remove: **O(logN)**
  + **BinaryHeap:**
    - Insertion: **O(logN)**
    - Access: **O(1)**
    - Remove: **O(logN)**

**Binary Heap:**

* **Heap property:** Is a tree with special properties or constraints on the Nodes (Min&Max).
* **Min Heap:** Highest priority being the smallest value.
  + Every Node value should be >=Min value to all its children.
* **Max Heap:** Highest priority being the largest value.
  + Every Node value should be <=Max value to all its children.
* Leaf Nodes are only found in H or H-1 (When satisfying the shape property of the heap).
* All nodes at level H-1 Must be filled before starting the next level.
* **Implementation:**
  + Each node contains pointer to left and right child.
  + A Node needs 2 child pointers, 1 parent pointer.
  + Needs to **extend Comparabl**e to check for highest priority.
  + Some variables: MAX\_SIZE, array[], count.
  + Heaps can be presented in Arrays for efficiency.
    - getLeftChild(): 2n + 1
    - getRightChild(): 2n + 2
    - getParent(): (n -1) / 2
  + **Operations:**
    - Traverse downwards towards leaf nodes from root.
    - Traverse upwards from leaf nodes towards root.
    - getLeftChild(); getRightChild(); getParent();
* **Min Heap Ex.**
* In Array form {5, 8, 6, 9, 12, 11, 7, 15, 10}

5

/ \

8 6

/ \ / \

9 12 11 7

/ \

15 10

* **Max Heap Ex.**
* In Array form: {47, 32, 28, 9, 12, 11, 7, 5, 3}

47

/ \

32 28

/ \ / \

9 12 11 7

/ \

5 3

**Heapify: Highest priority.**

* **Complexity:**
  + Insertion: **O(logN)**
  + Access: **O(N)**
  + Remove: **O(logN)**: Because we have to swap the nodes to their right position after removal.
* Finds the correct position where the element should be when inserting.
* **SiftDown:**
  + Element must traverse downwards to its right position.
  + Compare left&Right child, find the mostMinimum element,
* **SiftUp:**
  + Element must traverse upwards to its right position.
  + Only compares with its parent to determine if to shift.
  + Calls itself recursively.
* **Ex. MinHeap: 13 is out of place. SiftDown.**

**13**

/ \

8 **6**

/ \ / \

9 12 11 7

/ \

15 10

* Compare left&Right child, find the mostMinimum element, 6, so swap.

**6**

/ \

8 **13**

/ \ / \

9 12 11 **7**

/ \

15 10

* Compare left&Right child, find the mostMinimum element, 7, so swap.
* MinHeap Property SATISFIED!

**6**

/ \

8 **7**

/ \ / \

9 12 11 **13**

/ \

15 10

* **Ex. MinHeap: 4 is out of place. SiftUp.**

6

/ \

8 7

/ \ / \

9 12 11 13

/ \

15 **4**

* Compare 4 with its parent, 9, swap 4 and 9.

6

/ \

8 **7**

/ \ / \

**4** 12 11 **13**

/ \

15 **9**

* Compare 4 with its parent, 8, swap 4 and 8.

6

/ \

**4** **7**

/ \ / \

**8** 12 11 **13**

/ \

15 **9**

* Compare 4 with its parent, 6, swap 4 and 6.
* MinHeap Property SATISFIED!

**4**

/ \

**6** **7**

/ \ / \

**8** 12 11 **13**

/ \

15 **9**

* **Insert into Heap**
  + Insert element to the last leaf: Last index in an array.
  + Process SiftUp.
* **Remove Highest priority in a heap.**
  + Removes the highest priority in the MinHeap, 2
  + Copy the last element in the array to index 0 then SiftDown.

( )

/ \

4 7

/ \ / \

6 8 11 17

/

**15**

**15**

/ \

4 7

/ \ / \

6 8 11 17

/

* + 15 must be siftedDown to its correct position.

4

/ \

**15** 7

/ \ / \

**6** 8 11 17

4

/ \

**6** 7

/ \ / \

**15** 8 11 17

**Hash Map:**

* Like a dictionary which contains a value and a key.
* **Hashing:**
  + The transformation of a string of characters into a usually shorter fixed-length value or key that represents the original string.
  + Hashing is a process of converting an object into integer form by using the method hashCode().
* **HashMap:**
  + It’s a java collection with the implementation of the map interface.
  + Contains an array of Node and a node is represented as a class which contains 4 fields:
    - int hash, K k, V v, Node <K, V>
  + Default hashMap comes table size 16, n = 16.
  + Index = hash & (n-1)
* **Objections of hashFunctions:**
  + Minimize collisions.
  + Uniform distributions of hash values.
  + Easy to calculate.
  + Used to index large amount of data.
  + Address of each key calculated using the key itself.
  + Collisions resolved with open or closed addressing.
  + Used in database indexing, compilers, caching, password authentication, and more.
  + Insertion, deletion from hashTable, constant Time, unless ther are collisions.
* **Collision Resolution:**
  + **Open addressing**
    - **Linear Probing:**
      * Continuous searching for the empty spot in the array and then places it. (but causes clustering).
      * You can also have a +3 to check for empty spots every 3 spaces from current index.
    - **Quadratic Probing:**
      * Squares the number of fail attempts, when deciding how far along from the point of original collision to next, the distance from original point of collision grows rapidly.
    - **Double hashing:** 
      * Applies second hashfunction to the key when collision occurs, Results the number of positions along from original collision to try next.
  + **Closed addressing.** 
    - Involve Chaining items that have collided into a linked list.

**Heap Sort:**

* Convert the heap into an Array.
* Recursively sift up or down.
* Not adaptive.
* Not stable.
* Start with the last element in the unsorted Array.
  + Find its parent. **(n – 1) / 2.**
  + Compare the left and right child with the parent.
  + Swap based on the condition given.
  + Move to the previous parent, check, compare their children.
* **Ex: Find Max element in the min heap.** 
  + Start from the leaf nodes with the first leaf node.
  + The last parent in the tree takes you to the first leaf node.
* **Ex: Find K largest element in a stream.**
  + Use min heap to store K elements as they come in.
  + Top if the heap will have the smallest of the K largest elements.
  + If the new element in the heap is larger than the min heap, add it to the heap.

**Graph:**

* **Graph:** A set of vertices and edges.
* **Vertex:** Represents the entity.
* **Edges:** Represents the relationship.
* **DegreeOf:** Number of edges coming from the vertex.
* **Forest:** A disjoint set of trees.
* Graphs can be directional / undirectoinal.
* Connected graph with no cycle is a tree, and these trees can have more than 2 children.
* **3 ways to represent a graph.** (Giving a node who are its neighbors?)
  + **Adjacency List:**
  + **Adjacency Matrix:** to represent a graph. What other vertices is a vertex that’s adjacency to or directly connects with.
  + **Adjacency Set:**
* **Implementation:**
  + Create an interface Graph.
    - Enum { Directed, Undirected }
    - Void addEdge(int v1, int v2);
    - List<Integer>GetAdjacentVerticies(int v);
  + Have a class implement the Graph interface.
  + Create a Vertex Node class.
    - Set<Integer> adjacencySet = new Hashset<>;
    - Node(int vertexNumber)
    - getVertexNumber();
    - addEdge(int vertexNumber)
    - List<Integer>getAdjacenctVerticies(){ List<Integer>sortedList = new ArrayList<>(adjacencySet); Collections.sort(sortedList); return sortedList;
* **Adjacency Matrix:**
  + **Complexity:**
    - Space complexity: **O(V^2) –** rows and columns.
    - lookUp: **O(1) –** Find if edge is present from V1 – V2.
    - Iterate: **O(V) –** to iterate through the entire graph.

* + Works best when well connected. Ex. 1 vertex with many edges.
  + Set up a 2d array[][], mark 0 or 1 if the vertices connect with a vertex.
  + Directional graphs will only mark as 1.
  + Un-directional graphs mark 1 for the 2 vertices.
* **Adjacency List:**
  + **Complexity:**
    - Space: **O(E+V)** – Ea. vertex is a node + ea. edge is presented by vertex.
    - lookUpEdge: **O(Degree of V).**
    - IterateoverEdges: **O(Degree of V).**
  + Every node is represented as a LinkedList.
  + And the LinkedList contains the paths that node can take.
  + Major Downside:
    - This graph can have many different representations.
    - Operations become tricky, deleting a node involves looking through all the adjacency lists to remove that node. (Use MATRIX INSTEAD).

* **Adjacency Set:**
  + **Complexity:** 
    - Space: **O(E+V)** - # of edges, # of vertices.
    - checkIfEdgeisPresent**: O(logDegree of V).**
    - IterateOverEdgesOnVertex: **O(Degree Of V).**
  + Uses a set instead of a LinkedList to main the adjacent vertices.

**Graph Traversal - Depth First and Breadth:**

* **Traversal:**
  + Trees only have 1 path though, no cycle.
  + Graphs can have multiple paths, and possible cycle.
* **Depth first breath:**
  + Keep track of nodes that have been visited in a Boolean array list.
  + Uses PostOrder Traversal.
  + **If node visited, return**.
  + Recursive.
* **Breadth first Traversal:**
  + Start at Node 1, go to next node 2 from 1, and then the following node that also connects to 1, 3. Then 5 because it connects with 2, and then 4.
  + Use a queue to add the children in BFT.
  + Keep track of visited nodes in a Boolean list.
  + **If node, visited, continue,** for all adjacent vertices, add it to the list.
  + Non-recursive.

**Topological Sort in a Graph:**

* **Complexity:**
* **Topological:** Ordering of vertices in a directed acyclic graph in which each node comes before all nodes to which it has out going edges.
  + **Ex.** **Classes a, b, c, come before class d.**
* Graphs can have multiple topological sort.
* **The Idea:**
  + Start with vertex that has 0 inDegree. (No edges going in)
  + If the graph does not have Nodes with 0 inDegree, its cyclic.
  + When removing a Node, all verticies connected to the Node removed must decrement the indegree.
* **Implementation:**
  + Pass in the indegree value in the function
  + Check if the vertex is valid. 0 < numVertices – 1.
  + Iterate through all vertices.

**Shortest path unweighted algorithm (Graph):**

* **Complexity:**
  + Using Adjacency List: **O(V+E)**
  + Using Adjacency Matrix: **O(V^2)**
* Uses a distance table. With 3 columns.
  + Row, Distance, Last vertex.
* **Vertex:** All nodes in the graph.
* **Distance:** From source node to the vertex.
* **Last Vertex:** Last vertex encountered form the path, from source node to that vertex.
* You start by setting all values to -1 other than the source vertex because its 0.
* Use breadth first search to traverse the graph.
* **Concept. A connects to both A and C. update distance table.**
  + Add B and C to the queue.
  + Then remove 1 node from the queue that we want to check for its neighbors. **C**
    - C’s neighbors is +1 distance from A. distance [E] = distance [C] + 1
  + Enqueue E, process it, explore its neighbors, repeat.
  + Update the table and Continue same process.
  + Use Stack data structure to back track from destination to source node.
  + D is our destination, start back tracking.
  + Place D into the stack. Backtrack, how we got to D? from B
  + Place B into the stack. BackTrack, how we got to B? from E.
  + Continue till we get back to our source.
  + Then pop elements from stack to get shortest path.
* **Ex. Assuming our path is A, C, E, B, D**
* **Vertex Distance LastVertex**
* A 0 A : A🡪A : A prior to A
* B 3 E : A🡪C🡪E🡪B : E prior to B
* C 1 A : A🡪C : A prior to C
* D 4 B : A🡪C🡪E🡪B🡪D : B prior to D
* E 2 C : A🡪C🡪E : C prior to E
* Distance from source: unweighted graph.
  + **distance [vertex] = distance [lastVertex] + 1**
* **DistanceInfo table data structure.** 
  + **Class distanceInfo()**
  + Private int distance, lastVertex;
  + DistanceInfo(){ distance= -1, lastVertex= -1 }
  + getDistance();
  + getLastVertex();
  + setDistance();
  + setLastVertex()
* **Create distance Table idea.**
  + Map<Integer, DistanceInfo> distanceTable = new HashMap<>();
  + for (int j = 0; j < graph.getNumVertices(); j++) {
  + distanceTable.put(j, new DistanceInfo());
  + }

**Shortest path weighted algorithm (Graph):**

* Weight can be positive or negative**.** -VE +VE
* **3 differences from unweighted graphs:**
  + Distance from node has to account for weight of edges traversed.
    - **dist [neighbor] = dist [vertex] + weightOfEdge[vertex, neighbor].**
    - Init the table with inifinite instead of -1 till short path is found.
  + Each vertex has neighbors, but we only visit the neighbor with the lowest weight.
    - Using priority queue.
    - To get to next vertex, pop the element with lowest weight
  + Possible to visit vertex more than once.
    - Check if new distance via alternative route is smaller than old distance.
    - **newDist [neighbor]= dist [vertex]+weightOfEdge[vertex, neighbor].**

**Dijkstras algorithm:**

* **Dijkstras Algorithm:**
  + **Complexity:**
    - **add/remove** : **O(ElogV) –** Using binary heap.
    - **add/remove: O( E + V^2) –** Using array for priority queue.
  + Find shortest path in unweighted graph.
  + Like a **Greedy Algorithm:**
    - Builds up solution step by step.
    - Take lowest weight from current vertex.
    - Often fails to find the best solution.
    - Mainly used for optimizations, great for approximate solutions for very hard problems.
  + **Relaxation:**
    - Overwrites the distance and path to the distance.

**Bellmand-Ford algorithm:**

* Complexity:
  + Using Adjacency list: **O(E\*V).**
  + Using Adjacency Matrix: **O(V^3) –** V for checking and V^2 for matrix.
* Combination of Dijkstras algorithm.
  + However no greedy algorithm used.
  + No priority queue to traverse nodes.
  + Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths.

**Minimal Spanning Tree:**

* **Spanning Tree:**
  + Sub-graph that contains all vertices and is a tree.
  + All vertices connect but not all edges are connected.
  + Multiple spanning trees are possible from a single graph.
* **Prims Algorithm:**
  + **Complexity:**
    - **add/remove** : **O(ElogV) –** Using binary heap.
    - Using Array: **O(E+V^2) –** As priority queue.
  + Generate distance tableDijkstras algorithm using any vertex as the source.
  + Use the distance table to get paths to all other vertices from a random selected source.
  + Minimize total distance and not only the distance from one specific vertex.
    - Note: Dijkstras alg. Specifies one source vertex.
  + Minimize the cost of connecting all vertices.
  + Only care about the weight of the edge not cumalitive distance from source.
* **Use minimal spanning tree:**
  + Use shortPath alg. when wanting the shortest route from source 🡪 destination.
  + Use minimal spanning tree to find the cheapest way to interconnect cities.