Data Structure

Udemy Data Structure

* 1 D Array:
* 2D Array:
* Singly Linked List
* Doubly Linked List
* Memory Efficient Linked List –
* XOR Linked List **–** The reference pointer contains two values, One XOR value of Previous and the next pointer value.
* Hash table
* Hash Function
* Collisions

**-** Open Addressing -Occupying up the next available memory slot in the index table.

- Separate Chaining –Adding up a different data structure on the collided node.

* Stack - (LIFO)
* Implementation through Linked List
* Implementation through Array
* Queue – (FIFO)
* Implementation through Linked List
* Implementation through Array

Priority Queue **-** After every insertion in priority queue the queue sorts the element in the sorted order of ascending. And other operation are similar to a normal queue.

* Tree –

Tree can be treated as a kind of graph.

**Binary Tree** – Child node should have utmost 2 child.

* Strict BT – All the Node should have at least and utmost 2 child nodes
* Full BT – All the node should have at least and utmost 2 child nodes and all the leaf node should be at same level.
* Completed BT – At least all the left nodes till the leaves should contain data.

**Binary Search Tree** – Child node should have utmost 2 child and should follow below relation.

**Relation**: node.left.keyValue < node.root.keyValue < node.right.keyValue

* Search Implementation
* Insert Implementation
* Delete Implementation
* **Tree traversal**

**Breadth First Search (BFS) –** This involved identifying the depth levels and visiting all the levels sequentially.

**Depth First Search (DFS) –** Based on selected technique, this involves traversing the left unit first and switching over to another unit.

* Pre Order – N , L , R
* In Order – L , N , R
* Post Order – L , R , N
* **Graph**

Graph is also a nonlinear data structure which doesn’t haven’t parent child relationship, but it has various nodes known as **vertices**. The nodes are connected with a relationship called as **edges**.

**Types of Graph**

* Directed / Undirected Graph
* Weighted / Un weighted Graph

**Uses of Graph**

* Mutual Friends Identification on social media
* Shortest route identification based in graph edges weight.

**Graph representation**

* Adjacency Matrix

This type of relationship representation is depicted using 2 D Array- where the value is assigned 0 or 1 based on the node connectivity and on directed and undirected routes.

Note: For an undirected graph, the matrix will always be symmetric along the principal diagonal.

* Adjacency List

This type of relationship representation is depicted using a List. Every value in the list corresponds to a next data structure linked List, which contains all the nodes as part of connected with edge relationship based on directed or undirected routes.

**Comparison between types of graph representation**

**Compare Value Adjacency Matrix Adjacency List**

1. **Access Better (as O(1)) Need to traverse (O(n))**
2. **Access Child Not Possible Accessing linked list**
3. **Memory Xtra 0’s wastage Efficient**
4. **Iteration Need to iterate on NR values Need to iterate on R values**

**Graph Traversal**

* Breadth First Search - Involves visiting of all the nodes in same levels first
* Depth First Search – Involves visiting of left node level or the first node/vertex in the path to its complete depth prior to visiting the second node/vertex depth
* Refer this below website to simulate the flow for Graph traversal

<https://visualgo.net/en>

**\*\* Code Talks for graph representation implementation using Adjacency List \*\***

1. Constructor is responsible to initialize the graph based on the No of vertex values passed through main. Also array should be initialized in the constructor to contain linked list for required no of nodes in linked list array.
2. Size of linked list array will be equal to no of vertex.
3. addEdge() method should be used just to add the linked list node based on source and destination from main call based on requirement of creation of directed or undirected graph.
4. printRelation () method can be used to print the linked list nodes.

**\*\* Code Talks for graph traversal using Adjacency List \*\***

1. Using Adjacency list, we need to make used Boolean visited [] to keep track of visit.
2. Iteration here is done through queue, while loop should be used to iterate over queue, and internally again another while loop to iterate over the linkedArrayList from Adjacency List to identify the child nodes.
3. Always the call to DFS or BFS Algo should be done through root node.
4. Key is to make root node visited and add it in the queue and traverse over the queue. This holds good for both DFS and BFS.

**Shortest Path Algorithm**

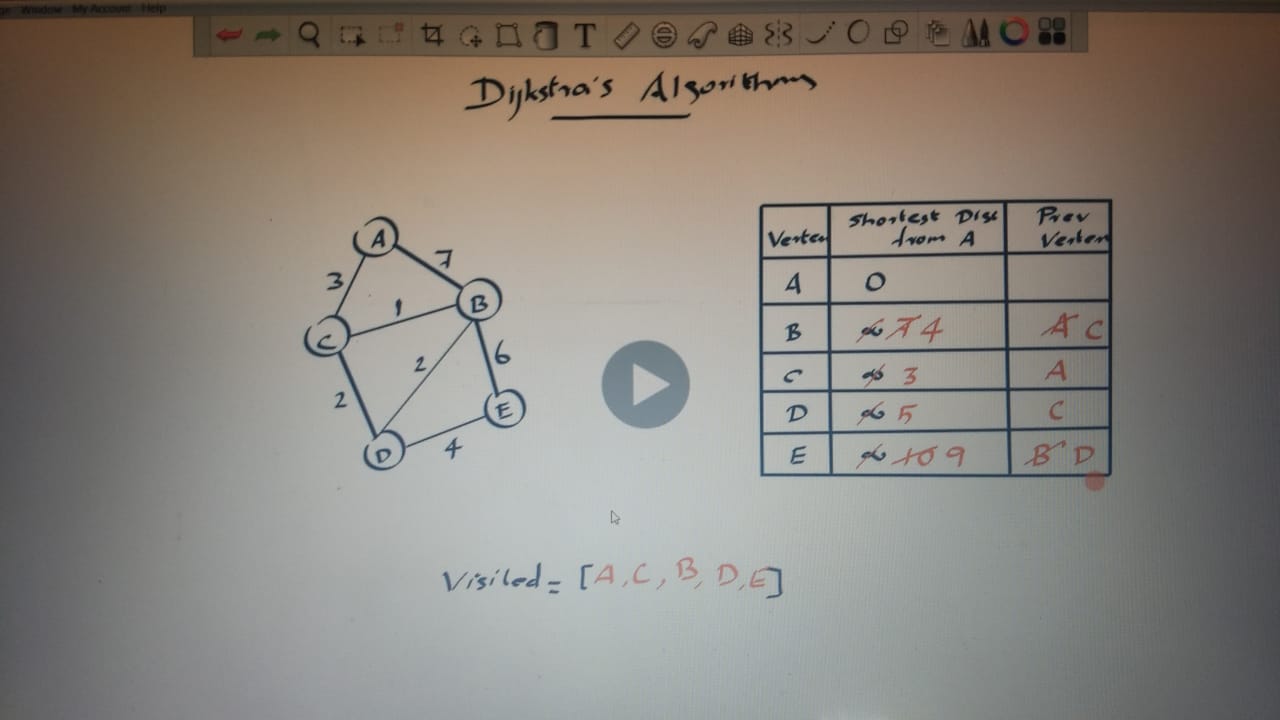
**Dijkstra’s Algorithm**

1. We have to choose vertex which has small distance/weight.
2. We have to check the neighboring vertices.
3. Calculate the distance for neighboring vertices by summing the cost of the vertices
4. If distance to the vertices is lesser than known distance, then update shortest distance.

Steps for shortest path analysis through stack

We need to start from last value in the table and start inserting that into the stack which will be followed by their parent and so on until we arrive at source node, which will essentially have no parent.

As now we have stack ready, we need to start popping from stack to get the shortest distance.



**A\* Algorithm for shortest path**

1. Need to start from source vertex, and travel across the enabled directed nodes.
2. While traveling the nodes value is calculated with a function involving heuristic intervention.

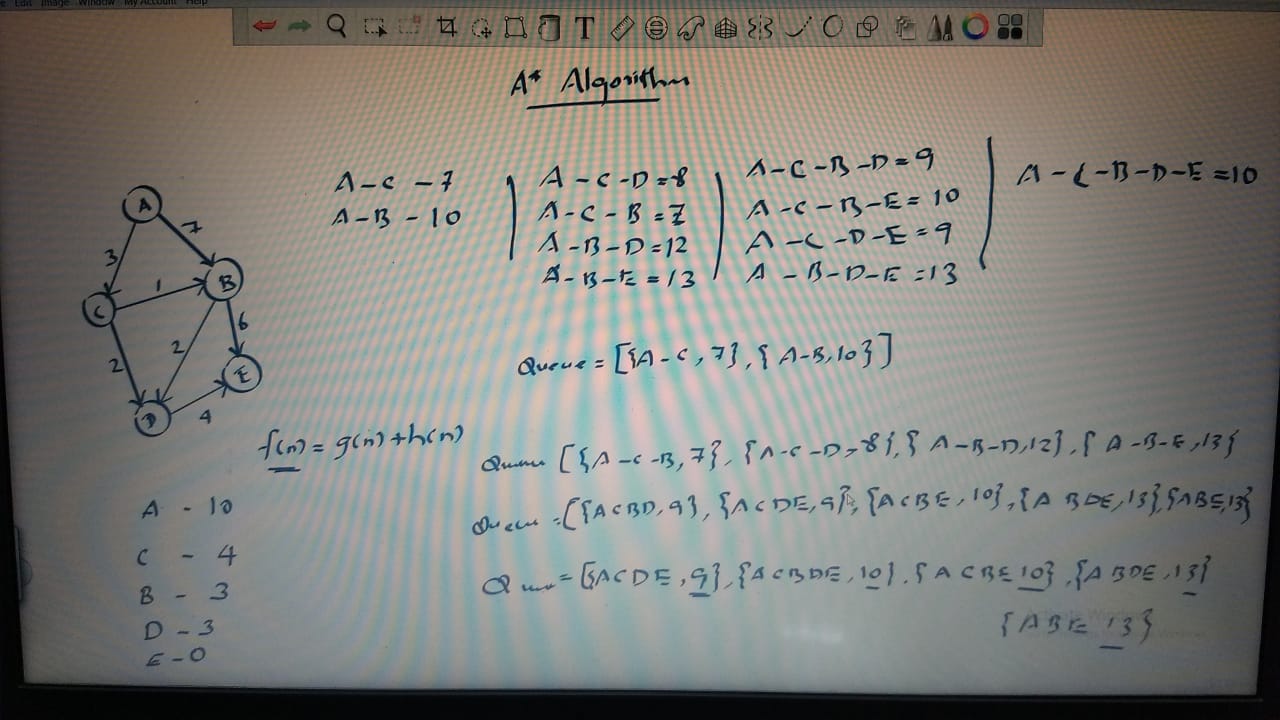
f (n) = g (n) + h (n)

* where n is the node value
* where g(n) is the edge summation for the pathway considered
* where h(n) is some heuristic value assumed for each node

1. Each value has to be iterated and till the point where it reaches the destination node.
2. The output value of f(n) has to be stored in priority queue .
3. Eventually after all the translations, the value at the start of the priority queue gives the shortest distance and shortest path.

\*\* Benefit than Dijkstra’s\*\*

* Here we get all possible paths with the coordinate and in the increasing distance order. Could be specifically used in map.



**Search Algorithm**

* **Linear search Algorithm -** complexity – O(n)
* **Binary Search Algorithm -** complexity – O(nlogn)

E.g.: Search over array is performed with low high values and recursive calls on the basis of mid value calculated.

For mid value = search Value: return value

For mid value > search value: recursive search on arr (arr, low, mid -1, search Value)

For mid value < search value: recursive search on arr (arr, mid+1, high, search Value)

**Sorting Algorithm**

* **Bubble sort – O(n^2)**
* Main aim is to shift the larger element to the extreme right in every iteration
* After every iteration , the last item gets abandoned for next iteration
* Continuously compares the element in the right and shift if its greater than the no in its right
* O(n^2) is the time complexity

**Github link -** <https://github.com/faizandroid01/Java_Workspace/blob/master/Intellij_workspace/DS/sorting/BubbleSort.java>

* **Selection sort – O(n^2)**
* Main aim is to shift the smaller element to the extreme left in every iteration
* After every iteration , the first item gets abandoned for next iteration
* Continuously compares the element in the arr to find the minimum one, and shift it to first of the eligible enabled arr.
* O(n^2) is the time complexity

**Github link -** <https://github.com/faizandroid01/Java_Workspace/blob/master/Intellij_workspace/DS/sorting/SelectionSort.java>

* **Insertion sort – O(n^2) –** Playing Cards Sorting Technique
* Main aim is to fill the sorted subset, from unsorted subset.
* After each iteration, the sorted subset increases by 1 and unsorted subset decreases by 1
* Continuously compares the element in the unsorted set with one in sorted subset starting from the end index of the sorted subset to place it appropriately in the arr.
* After every comparison of the sorted set member with the temp value, if the member is larger than temp, then it will be shifted to next index and temp will be placed at emptied index, basically swapping of temp is done with the larger index val.
* O(n^2) is the time complexity
* **Merge sort – O(Time complexity - O(n logn), Space complexity – O(n) )**
* Main aim is to shift the larger element to the right in every iteration
* After every iteration , the last item gets abandoned for next iteration
* Continuously compares the element in the right and shift if its greater than the no in its right
* O(n^2) is the time complexity
* **Quick sort – O (Best case – O(n logn) – Worse Case – O(n^2) - Space complexity – O(1))**
* Main aim is to shift the larger element to the right in every iteration
* After every iteration , the last item gets abandoned for next iteration
* Continuously compares the element in the right and shift if its greater than the no in its right
* O(n^2) is the time complexity
* **Heap sort – O(Time complexity - O(n logn), Space complexity – O(1) )**

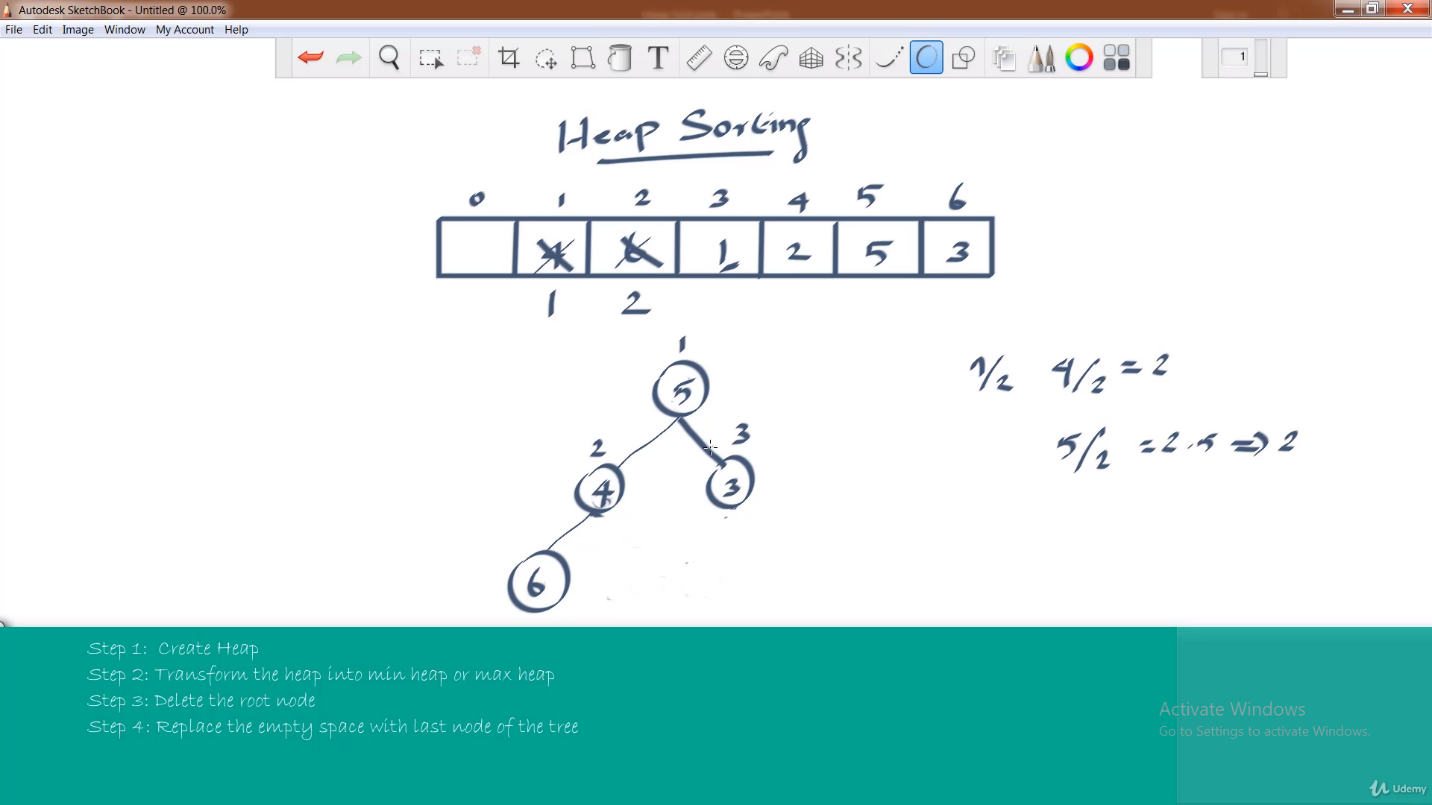
Note: Minimum Heap Tree -> Ascending Order Sorting -> Each parent node < Child node

Maximum Heap Tree -> Descending Order Sorting -> Each parent node > Child node

Steps to perform heap sort:

1. Create Heap.
2. Transform the heap into minimum heap or maximum heap.
3. Delete the root node and place it in the array position starting from start index.
4. Replace the empty space with last node of the tree. (Last node is basically the right most value available from right sub or left sub in the last level.)

* Main aim is to convert the heap in to either the min heap or max heap.
* After every successful transformation, the root node deleted will give the next sorted value for the sorted array.
* The comparison for heap conversion starts from n/2 of the remaining array pos.
* O(n logn) is the time complexity

****