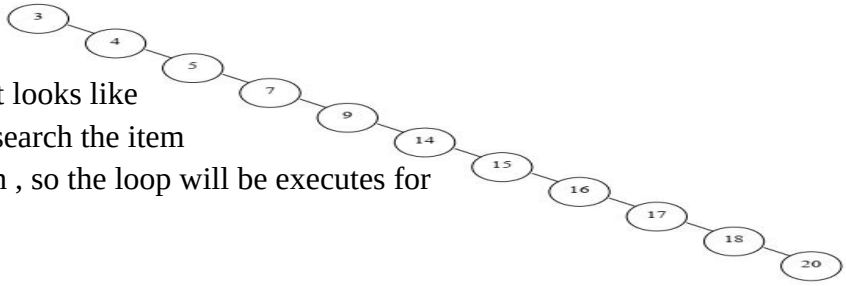
DAWOOD UNIVERSITY OF ENGINEERING & TECHNOLOGY KARACHI
BACHELOR OF SCIENCE IN COPUTER SCIENCE
DATA STRUCTURES AND ALGORITHMS

AVL Tree

Why AVL?

Let insert the following elements in the given order,

BST for 3, 4, 5, 7, 9, 14, 15, 16, 17, 18, 20



The BST is skewed tree, and it looks like the link list , so if we want to search the item 20, have to go upto last, or at n , so the loop will be executes for 'n' times

Remember , the depth Binary Tree,

$$d = \log_2 (n + 1) - 1$$

if nodes, n = 100,000

$$\text{so, } d = \log_2 (100000 + 1) - 1 = \log_2 (100001) - 1 = 20$$

20 levels in tree and maximum 20 comparisons be made therefore, one comparison at each level. It is benefit of tree as compared to the linked list, search is fast.

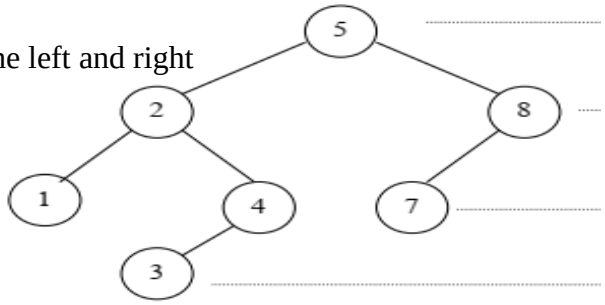
In the above case, skewed tree (sorted order), the benefit of BST is not applicable.

AVL is balanced BST, with following condition

- Height of the left and right subtrees may differ by at most 1.
- Height of an empty tree is defined to be (–1).

The condition in the AVL tree is that at any node the height of left subtree can be one more or one less than the height of right subtree.

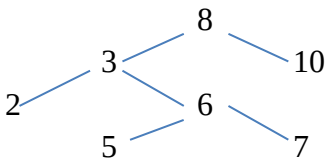
In the given figure, lets find the difference at each node of the left and right Sub trees.

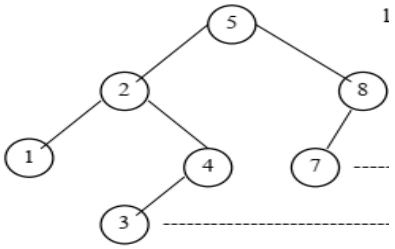


At node 2: height of Left subtree (LST) is 1 and RST is 2
So the difference is 1, condition valid
At node 8: the height of LST is 1 and RST is zero, so the difference is 1, condition valid
and at leaves, height is zero, so the above tree is in AVL

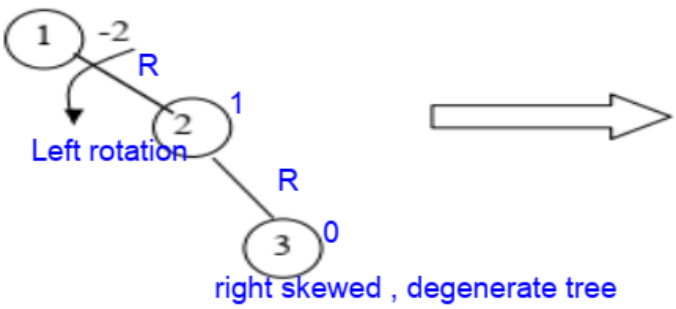
"The *height* of a binary tree is the maximum level of its leaves (also called the depth). The height of a tree is the longest path from the root to the leaf".

“The balance of a node in a binary tree is defined as the height of its left subtree minus height of its right subtree.”
AVL saves the BST from its degenerate form





Building AVL (rotation of elements)



To balance a tree

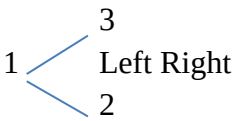
We do , left rotation or right rotation

Right Skewed tree ------→ left rotation

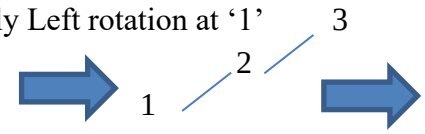
Left skewed tree ----------→ right rotation

Combination of Rotations

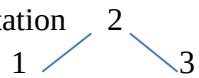
Right – Right,



Left – Left, apply Left rotation at ‘1’



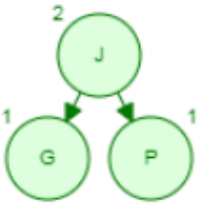
Right – Left, apply right rotation at ‘3’

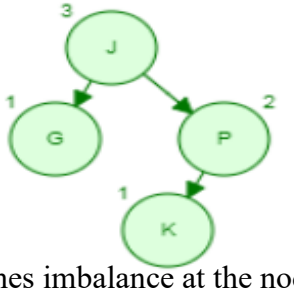


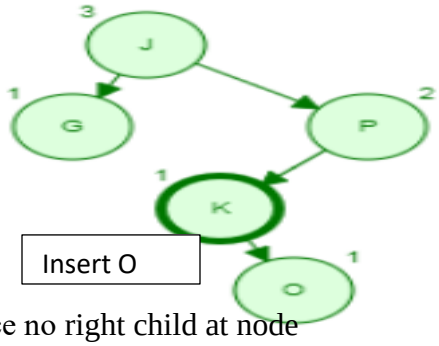
Left – Right

Example: AVL Insertion

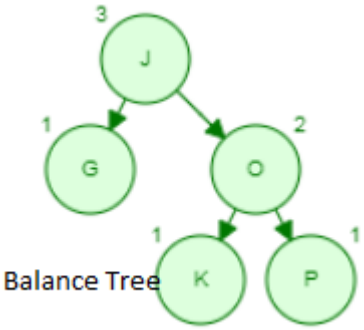
Insert Following Elements in AVL-------→ J, P, G, K, O, I, H, Q, R

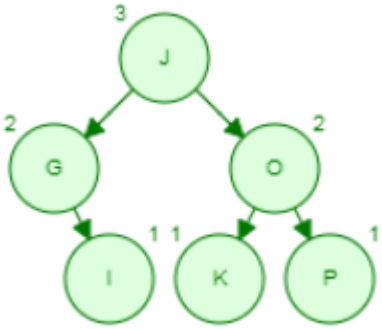


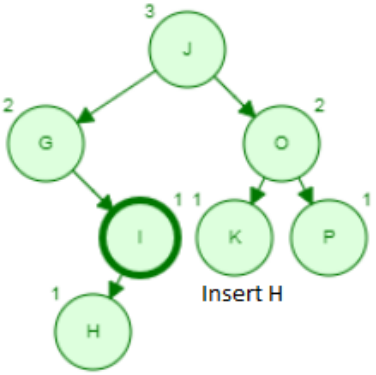




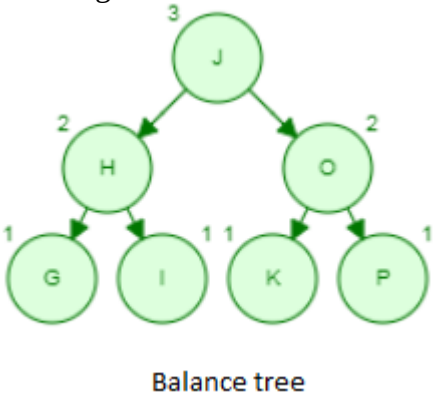
After the insertion' O', the tree becomes imbalance at the node 'P' since no right child at node 'P'. It has two rotations Left – Right Rotations, now apply balancing the tree becomes,

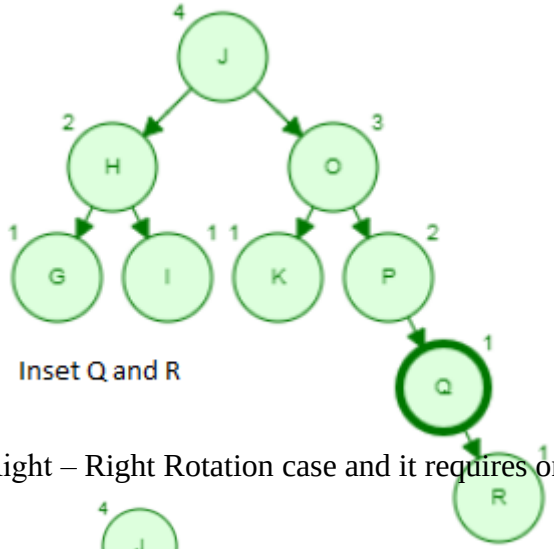




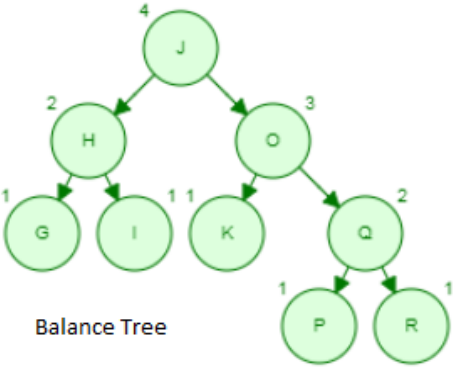


After inserting 'H', the balance factor is violate at node 'G', since right child of node 'G' and at left subtree has height of 2 (G to I to H), now it becomes Left – Right Rotation case, after balancing the node the tree becomes.





At 'P' the balance factor is not valid, so Right – Right Rotation case and it requires only single left rotation to balancing the tree



Example: AVL Deletion

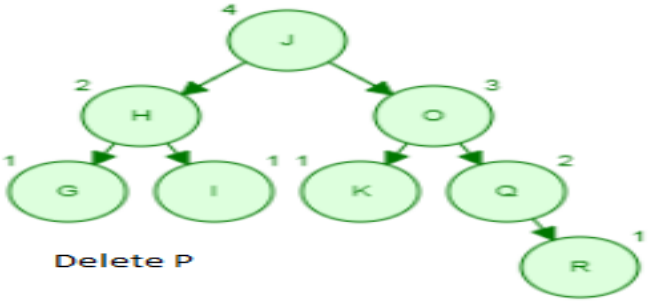
To delete the node from AVL, two ways are available,

i) In-order Predecessor (if, node has left subtree (LST), shift the largest element of left Sub Tree at position of the node to be deleted)

ii) In-order Successor (if, node has right subtree (RST), shift the smallest element of Right Sub Tree at position of the node to be deleted)

Delete 'P'

'P' is leaf node, so deletion of 'P' not violate the balance factor



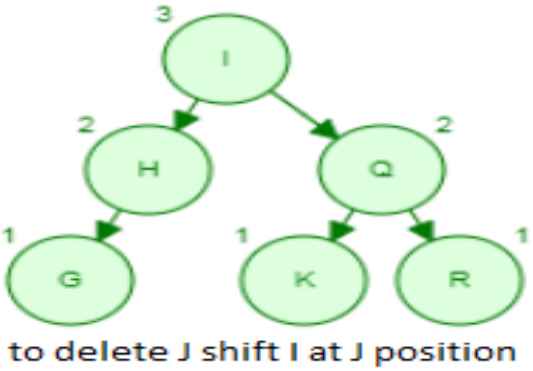
Delete 'O'

Node 'O' has left and right subtree, in this example consider 'In-order Predecessor', so the largest element at LST is 'K', now 'K' is shifted at position of 'O' (Node to be Deleted) and then check the balance factor.

At ' K', node is imbalance, so the tree will be

Delete 'J'

TO Delete 'J', (considering In-order Predecessor), the largest element is 'I', so replace 'I' with to be deleted node 'J', and then check balance factor, the tree is balanced.



3rd Week

Heap Structures:

- Used in priority queue
- Implementation of priority queue with an array is not efficient in terms of time
- Priority queue, itself a major data structure, with the help of heap is a major application.
- Priority queue data structure is used in network devices especially in routers. Heap data structure is also employed in sorting algorithms.
- **Heap is a complete binary tree that conforms to the heap order.**
- Heap order is a property that states, that in a (min) heap for every node X, the key in the parent is smaller than (or equal to) the key in X. In other words, the parent node has key smaller than or equal to both of its children nodes.

Thing to be noticed, in a BST the value of a node is greater than the values in its left subtree and less than the values in its right subtree. In heap, the value of a node is less than the values of its left and right children. Consider the following diagram, a example of minimum Heap.

Fig A, is Complete Binary Tree but not in BST, tree is a Heap order.

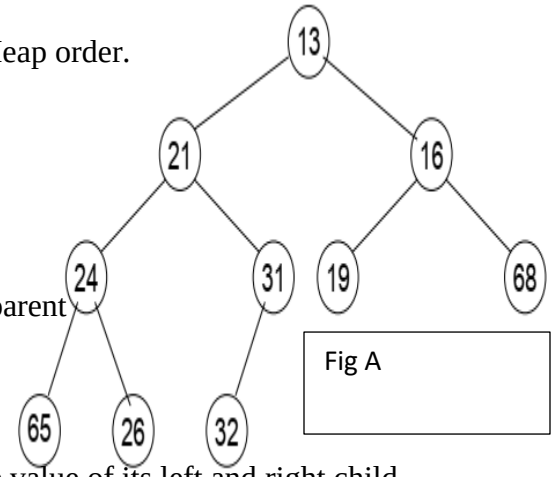
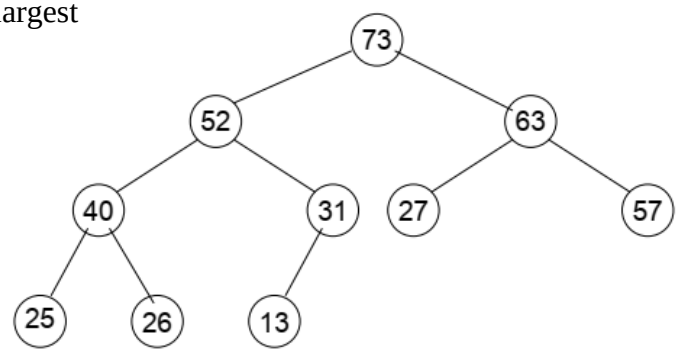


Fig A

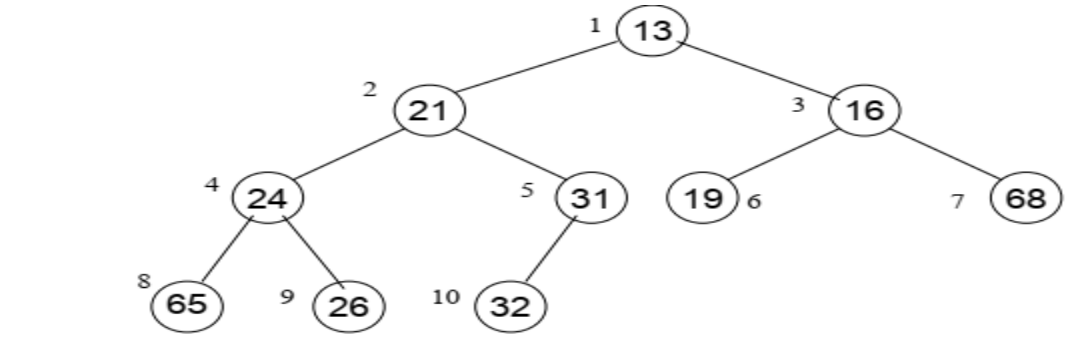
Tree in fig A fulfills the three properties.

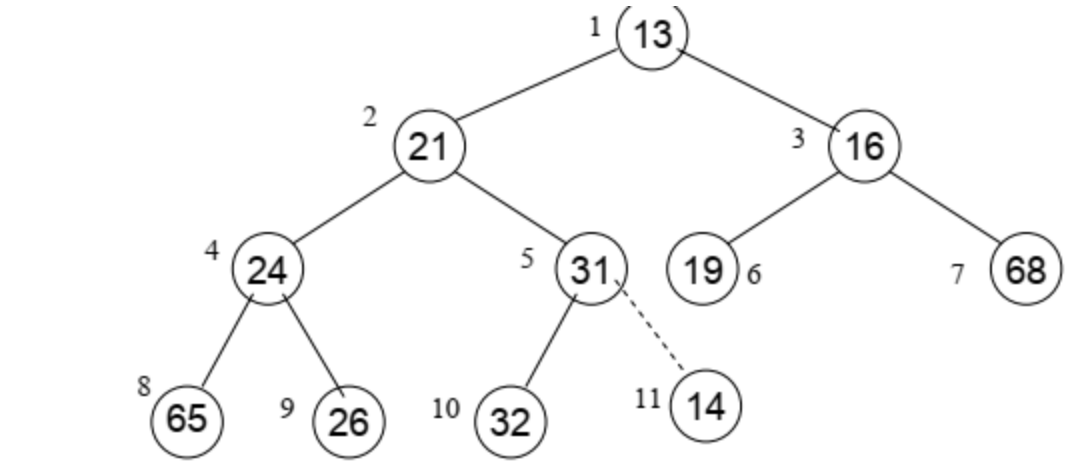
i. It is a binary tree.
ii. It is a complete binary tree. T
iii. It fulfills the heap order property (the value of the parent node is less than that of its left and right child nodes.
This property is valid for every node of the tree.

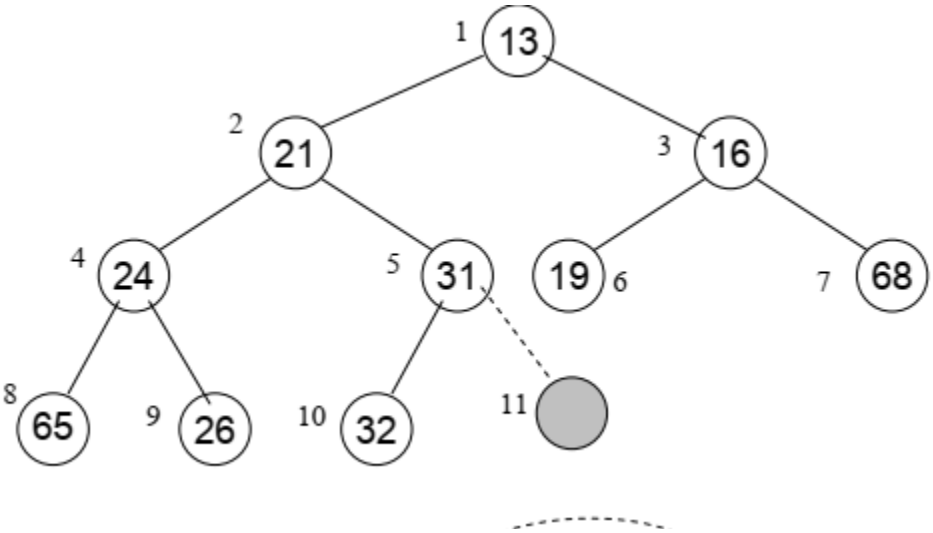
- In max heap, each node has a value greater than the value of its left and right child nodes. The value of the root node will be largest

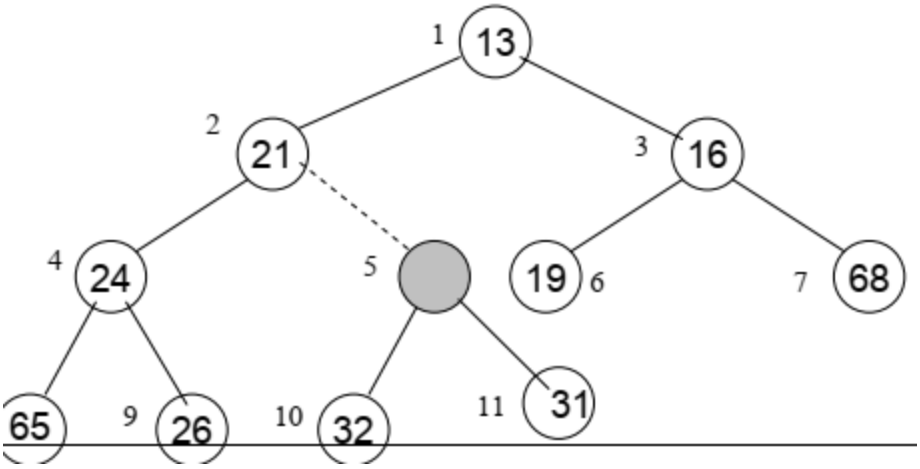


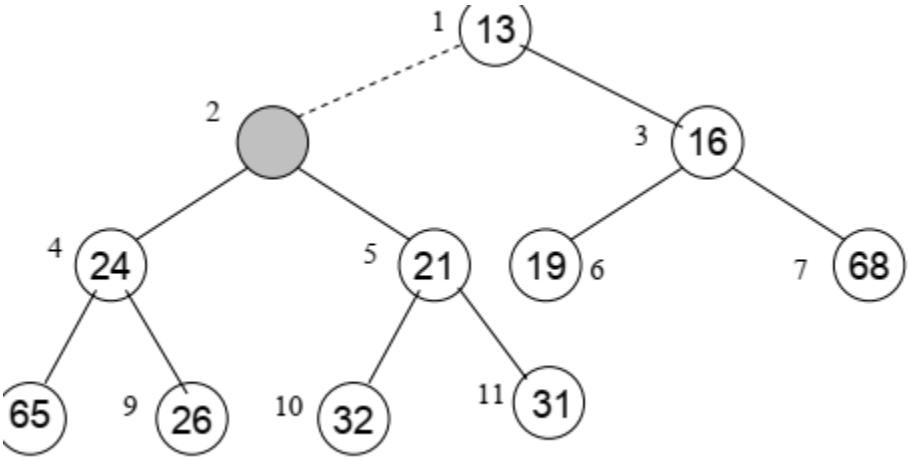
Example of insertion in Min Heap:

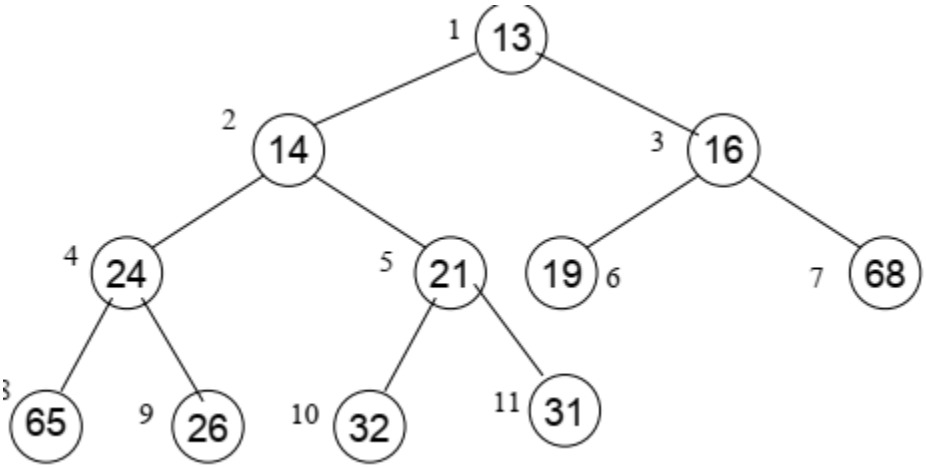












Insertion complexity
O(1) to o(log n)
O (log n)
n(log n)

Heap Sorting

20, 30, 10, 50, 15, 5
Insertion of elements in Heap, n x(log n)
Deletion from heap n x (log n)
Total complexity of heap sort, = 2n log n
Bog O notation O log n

Hashing

The hashing is an algorithmic procedure and a methodology.

A way to use existing Data Structures

Hashing is use in case of having large data or data in the form of tables.

The amount of time required to look up an element in the array is either O(log n) or O(n) based on whether the array is sorted or not. If the array is sorted then a technique such as binary search can be used to search the array. Otherwise, the array must be searched linearly.

Either case may not be desirable if need to process a very large data set, a new technique called hashing that allows to update and retrieve any entry in constant time O(1). The constant time or O(1) performance means, the amount of time to perform the operation does **not depend on data size n**.

The methods- find, insert and remove of table will get of constant time

Unsorted Sequential Array Table Operations

Insert	O(1) as you will add to the end of the array
Find	O(n) as you have to sequentially search the array, in the worst case through the entire array
Remove	O(n) as you have to sequentially search the array, delete the element and copy all elements one place up

Sorted Sequential Array Table Operations

Insert	O(n) as you will add to the end of the array
Find	Olog(n) as you can perform a binary search operation.
Remove	O(n) as you have to perform a binary search and shuffle elements one place up

Linked List Table Operations

Insert	O(n) for a sorted list O(1) for an unsorted list, insert at the beginning
Find	O(n) as you have to traverse the entire list in the worst case

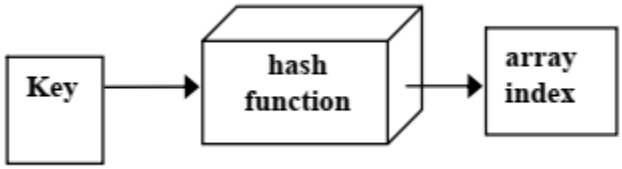
Remove	O(n) as you have to traverse the list to find the node, removal is carried out using pointer operations

Binary Tree Table Operations

Insert	O(logn)) tree is an ordered binary tree
Find	O(log(n)) as the tree is an ordered binary tree
Remove	O(log(n)) as finding takes O(log(n)) and removal takes constant time as it is carried out using pointer operations

O(log(N)) is good but as the size of the table becomes larger, even this value becomes significant, now need strategy for finding of O(1)(constant).

In Hashing, internally use array but not store data in consecutive locations, place of storage is calculated using the key and a hash function.



Key that may be a name, or roll number (any object as the key)

The key for a given object can be calculated using a function called a hash function.

Pass a key to a mathematical function, called Hash Function, which will return an array index, but this index in some range but not in sequence.

Keys and entries are scattered throughout the array. Suppose want to insert the data of an employee. The key is the name of the employee. We will pass this key to the hash function which will return an integer, using this number as array index. We will insert the data of the employee at that index.

Hash tables are very good for stable table (less insertion and deletion operations, mostly only search operation), and if insertion and deletion are frequently take place so there is choice to implement with AVLs, but there are no hard and fast statistics to go for hash table and then to AVL tree. (Hashing is very slow for any operations which require the entries to be sorted, Find the minimum key)

Properties of Hash Table:

- **hash table** is an array-like data structure for storing and retrieving data.
- Constant-time data access and search.
    - And the constant should be close to one.
- Other array-like properties may be sacrificed for the constant-time operations.
    - Ordered sequential access, for example.
- Even so, constant-time operations may be elusive and expensive to achieve.

Hash Functions

Modular Arithmetic

Example: let your Roll # is the key or any integer value, i.e ; 25 MOD 4 = 1

Truncating

Let the key is the last three digits of your cell/ mobile number i.e; 0300 9211 420, so value or index will be 420 (what is the range of the table)

Folding

Divide the key into parts and then combine it to from index

Let your name, JAMAL, is Key so 'J' is the 10^{th} letter and 'L' is 12^{th} letter, the value will be 10+12=22

JAMEEL-→10+12=22 (may be generate same index value)

Our Class example is:

If Roll # is 22 is key so the index value will be 2 + 2 = 4

The array index is calculated on the basis of addition of digits of the student roll # , and two students A and B have the roll # 25 and 52 so what will be the index value?

A roll # is 25, 2 + 5 = 7 is the index value

B roll # is 52, 5 + 2 = 7 is the index value

Both keys generate the same index value, this problem is known as Collision

Problem with Hashing (Collision)

In some cases, the hash function returns the same value for the two or more Keys, means the resultant of Hash function points the same index, leads to Collision.

Needs enough table size to avoid collision, Explain?

Hash Values

Suppose, a hash table of size N. A hash function is used to compute a hash value, computed from the key with the use of a hash function to get a number in the range 0 to N−1

It is desirable that, the addresses are different and spread evenly over the range

A Hash Function is considered good if,

- Fast to compute, O(1)
- Scatter keys evenly throughout the hash table
- Less collisions

Perfect hash function is a one-to-one mapping between keys and hash values. So no collision occurs, (will be it possible? Only when all keys are known, Any application/ examples?)

Hashing Performance

There are three factors the influence the performance of hashing:

Hash function
- should distribute the keys and entries evenly throughout the entire table
- should minimize collisions

Collision resolution strategy
- Open Addressing: store the key/entry in a different position
- Separate Chaining: chain several keys/entries in the same position

Table size
- Too large a table, will cause a wastage of memory
- Too small a table will cause increased collisions and eventually force *rehashing*

Handling Collisions

i. Linear probing---→ if collision, the next available space is given
ii. Quadratic probing---→ if collision occurs, then do n^2 for next location (n is no of collision)
iii. Double hashing-→ if collision occurs, a new/ next hash function is used to generate index value (means, double hash functions are used)
iv. Separate chaining

[NB: the above three techniques are called Open Addressing/ Close Hashing, and the fourth one Close Addressing and Open Hashing]

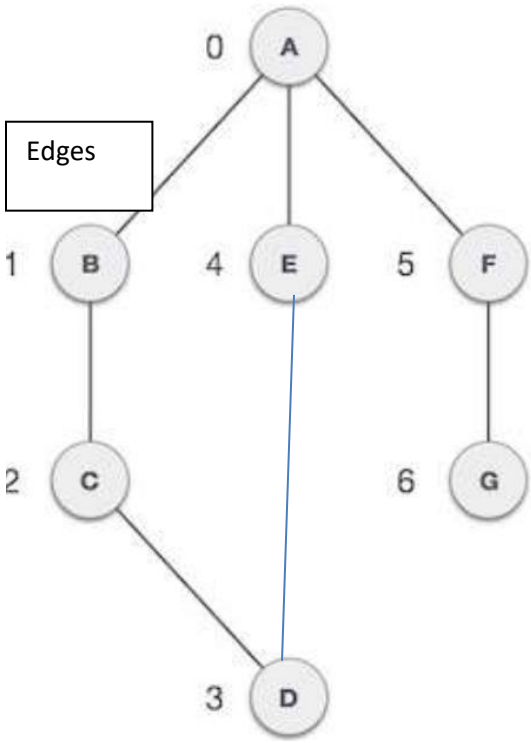
Key Points – Hashing

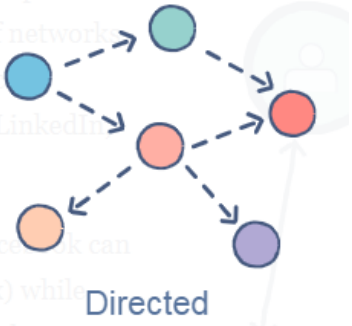
- Hash tables are very good if there is a need for many searches in a reasonably stable table.
- Hash tables are not so good if there are many insertions and deletions, or if table traversals are needed — in this case, AVL trees are better.
- Also, hashing is very slow for any operations which require the entries to be sorted

Load factor = num of item/ num of slots

Graph Data Structure:

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. a graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices.
To represent a graph in computers, an array of vertices and a two-dimensional array of edges.
Vertex − Each node of the graph is represented as a vertex, represent them using an array
Edge − Edge represents a path between two vertices or a line between two vertices, use a two-dimensional array to represent edges.
Adjacency − Two node or vertices are adjacent if they are connected to each other through an edge.
Path − Path represents a sequence of edges between the two vertices.





Primary operation on graph-→ Add vertex, add edge, display vertex

A cyclic graph→ graph does not have any cycle, it may be directed or undirected

e.g: Tree→ acyclic graph

(BFS and DFS of graph from simulation site)

a->b (directed graph), (a,b) is not equal to (b,a), is in one way form

a---- b (undirected graph), (a,b) = (b,a)-→ undirected graph is two way

in directed graph # of edges→ 0 <=n(n-1)

in undirected graph # of edges → 0 <={n(n-1)}/2 (if no self loop)

Dense Graph→ more # of edges-→ Adjacency Matrix

Sparse Graph→ too few edges→ adjacency List

BFS

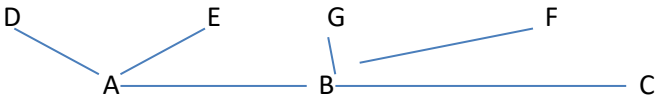
For BFS in graphs Queue Data Structure is used

BFS finds the shortest path, so BFS has the minimal steps to solution, it is simple

DFS

For DFS in graphs, Stack Data Structure is used

BFS and DFS starts from any vertex in graph, therefore many solutions exist



DFS:

Start from vertex A

1: A, B

2: A,B,C

3: A,B,C,F

4: A,B C, F, G

5: A,B,C,F,G,D

6: A,B,C,F,G,D,E