**Greedy Algorithm:-**

-Prim's, Kruskal

-Kruskal Algorithm O(E log V)

- Dijkstra Single Source Shortest Path Algorithm O(V2)

-Fractional Knapsack problem

-Huffman encoding Alg

**Divide and conquer**

**-**Merge Sort, Quick Sort

**Dynamic Programming**

-Floyd Warshall Algorithm All pair shortest path (Time Complexity: O(V^3) )

-Bellman-Ford algorithm Single Source Shortest path O(V + VE + E) = O(VE).

**TIME COMPLEXITY**

**MERGE SORT :-**

- time complexity is O(nlogn);

- recursive function is T(n) = 2T(n/2) + ɵ(n)

**SHELL SORT :-**

Time complexity of above implementation of shellsort is O(n^2).

**QUICK SORT :-**

- Worst case time complexity is ɵ(n^2)

- Best case time complexity is ɵ(n log n)

**INSERTION SORT :-**

- Worst case time complexity is ɵ(n^2)

- Best case time complexity is ɵ(n)

**HASH TABLE AND SELF ORGANIZATION LIST:-**

- Best case insertion is O(1)

- Worst case insertion is O(n)

**STRASSEN'S MATRIX MULTIPLICATION :-**

- Time complexity is 0(n^log(7))

**SELECTION SORT:-**

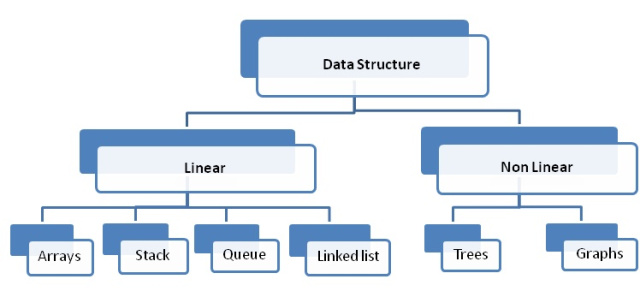
Worst Case Time Complexity : O(n2)

Best Case Time Complexity : O(n2)

Average Time Complexity : O(n2)

Space Complexity : O(1)

**LINEAR & NON LINEAR DATA STRUCTURES:-**

****

**INORDER TRAVERSAL** :-

LEFT, ROOT, RIGHT

**PREORDER TRAVERSAL:-**

ROOT, LEFT, RIGHT

**POST ORDER TRAVERSAL:-**

LEFT, RIGHT, ROOT

| **Infix Expression** | **Prefix Expression** | **Postfix Expression** |
| --- | --- | --- |
| A + B \* C + D | + + A \* B C D | A B C \* + D + |
| (A + B) \* (C + D) | \* + A B + C D | A B + C D + \* |
| A \* B + C \* D | + \* A B \* C D | A B \* C D \* + |
| A + B + C + D | + + + A B C D | A B + C + D + |
| A + B | + A B | A B + |
| A + B \* C | + A \* B C | A B C \* + |

**POLISH NOTATION:-**

// WRITE HERE IN FUTURE

**REVERSE POLISH NOTATION:-**

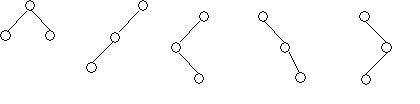
Reverse Polish notation, also known as postfix notation

**POSSIBLE DIFFERENT TREE**:-

If there are n nodes, there exist 2n - n different trees

For example, consider a tree with 3 nodes(n=3),

it will have the maximum combination of 5 different (ie, 8 – 3 = 5) trees



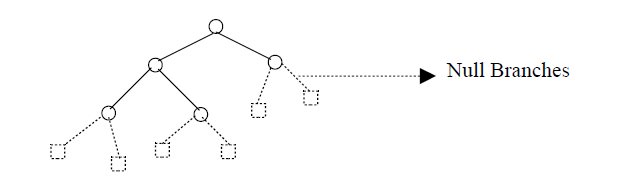
**BINARY TREE NULL BRANCHES** :-

It will have only 6 (ie,5+1) null branches.

A binary tree with n nodes has exactly n+1 null nodes.

Total Number of Binary trees with n nodes is





**COMPLETE BINARY TREE:-**

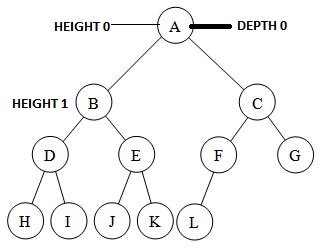
-A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

-Total Number of nodes in a perfect binary tree of height h = 2h+1– 1

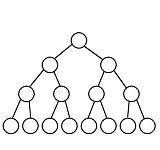
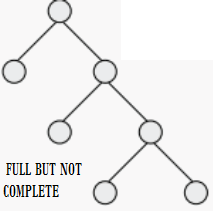
-The minimum number of nodes in a binary tree of height h = h + 1

-All nodes except leaf nodes are internal nodes (i.e., with at least 1 child node).

-Internal nodes in a perfect binary tree of height **h = 2h – 1**



**FULL BINARY TREE :-**

**The Radix Sort Algorithm :-**

Original, unsorted list: 170, 45, 75, 90, 802, 24, 2, 66

**Step 1 :-**Sorting by least significant digit (1s place) gives:

170, 90, 802, 2, 24, 45, 75, 66

**Step 2 :-**Sorting by next digit (10s place) gives:

802, 2, 24, 45, 66, 170, 75, 90

**Step 3:-**Sorting by most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802

**Time complexity:-**

Let there be d digits in input integers. b is the base for representing numbers, for example, for decimal system, b is 10.the set array {1, 2, ..., n}

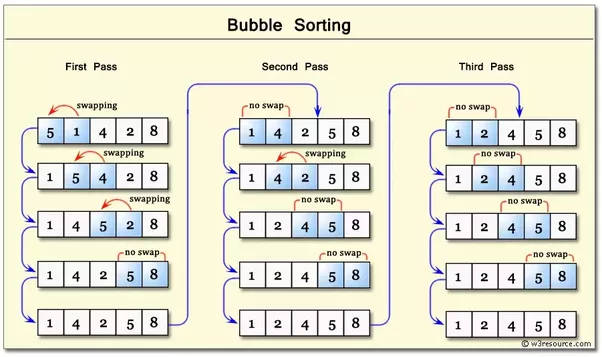
Radix Sort takes O(d\*(n+b)) time

**INSERTION SORT:-**

You have to sort a list L, consisting of a sorted list followed by a few ‘random’ elements. In this case this sorting method is the right method.



**BUBBLE SORT:-**

****

**Time Complexity:-**

Total number of comparisons in bubble sort is (n - 1) + (n - 2) + (n-3) +(n-4) +(n-5) ….....(2) + (1) = O(n(n - 1)/2) i.e, O(n2).

**TOPOLOGICAL SORTING:-**

1. Time complexity is O(V + E)

2.Compute the indegrees of all vertices

3.Find a vertex U with indegree 0 and print it (store it in the ordering)

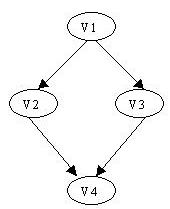
4.If there is no such vertex then there is a cycle

4.and the vertices cannot be ordered. Stop.

5.Remove U and all its edges (U,V) from the graph.

6.Update the indegrees of the remaining vertices.

7.Repeat steps 2 through 4 while there are vertices to be processed.



V1, V2, V3, V4 and V1, V3, V2, V4 are legal orderings

Degree of a vertex U: the number of edges (U,V) - outgoing edges

Indegree of a vertex U: the number of edges (V,U) - incoming edges

**Algorithm** :-

The fractional knap sack problem can be solved greedy algorithm

The 0-1 knap sack problem can be solved with a dynamic programming approach

Time complexity: ClearlyO(nW)

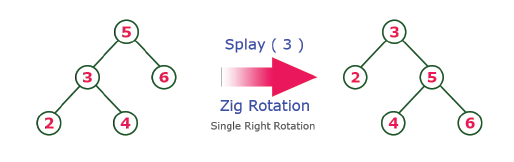
**Splay Tree O(log n) :-**

Splay trees are self-adjusting binary search trees that reduces the number of operations required to access recently accessed nodes. It achieves this property by bringing recently accessed nodes closer to the root of the tree.

All operations are in amortized O(log n) time.

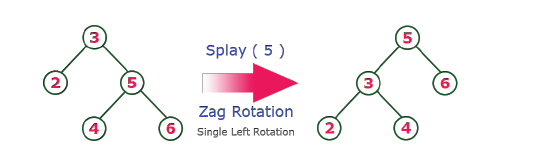
**Zig Rotation:-**

It is similar to the single right rotation in AVL Tree rotations. In zig rotation every node moves one position to the right from its current position.



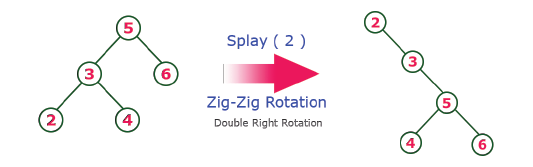
**Zag Rotation:-**

It is similar to the single left rotation in AVL Tree rotations. In zag rotation every node moves one position to the left from its current position.



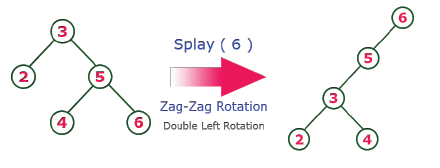
**Zig-Zig Rotation:-**

It is a double zig rotation. In zig-zig rotation every node moves two position to the right from its current position.



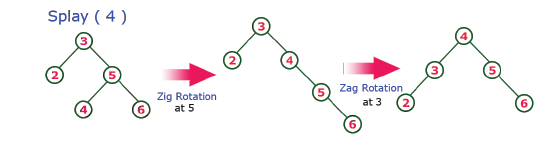
**Zag-Zag Rotation:-**

It is a double zag rotation. In zag-zag rotation every node moves two position to the left from its current position



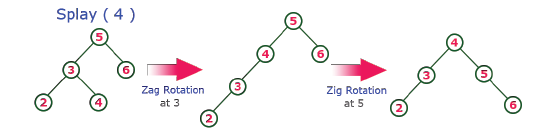
**Zig-Zag Rotation:-**

It is a sequence of zig rotation followed by zag rotation. In zig-zag rotation every node moves one position to the right followed by one position to the left from its current position.



**Zag-Zig Rotation:-**

It is a sequence of zag rotation followed by zig rotation. In zag-zig rotation every node moves one position to the left followed by one position to the right from its current position.



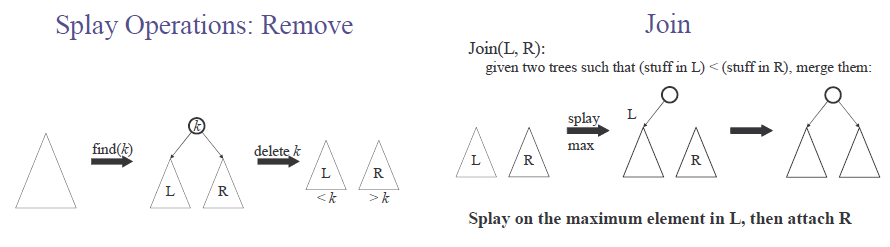
**Insertion:-**

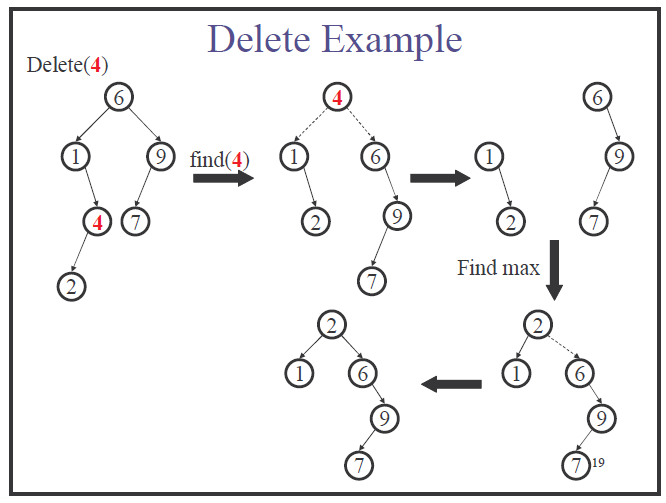
\* insert the new Node as a leaf node using Binary Search tree insertion logic.

\* After insertion, Splay the newNode

**Deletion:-**

It is similar to deletion operation in Binary Search Tree. But before deleting the element first we need to splay that node then delete it from the root position then join the remaining tree.





**Mathematical Properties of Spanning Tree**

Spanning tree has n-1 edges, where n is the number of nodes (vertices).

From a complete graph, by removing maximum e - n + 1 edges, we can construct a spanning tree.

A complete graph can have maximum nn-2 number of spanning trees.

**FLOOR & CEIL function:**

Floor function: the largest integer < X , Symbol ⎣ X ⎦

Ceiling function: the smallest integer > X, Symbol ⎣2.7 ⎦

**B TREE**

B tree of order M is a tree with the following properties

1. The root is either a leaf or has between 2 and M children

2. All non leaf nodes(except the root) have between M/2 and M children

3. All the leaves are at the same depth

The formula for calculating the maximum number of nodes in a B-tree of order order n of depth h is m pow(h+1) -1 = m ^ ( h + 1 ) - 1

The upper bound and lower bound for the number of leaves in a B-tree of degree K with height h is given by

C:\Users\antony\Desktop\Untitled.png

There are lower and upper bounds on the number of keys a node can contain. These bounds can be expressed in terms of a fixed integer t >= 2 called the minimum degree of the B-tree:

Every node other than the root must have at least t-1 keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

Every node can contain at most 2t-1 keys. Therefore, an internal node can have at most 2t children. We say that a node is full if it contains exactly 2t-1 keys.

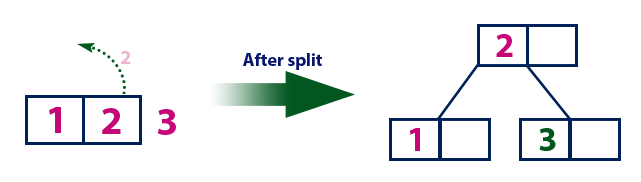
**Construct B-tree of order 3 by inserting 1 to 10**

(https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html)

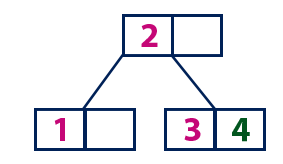
Insertion 1 & 2



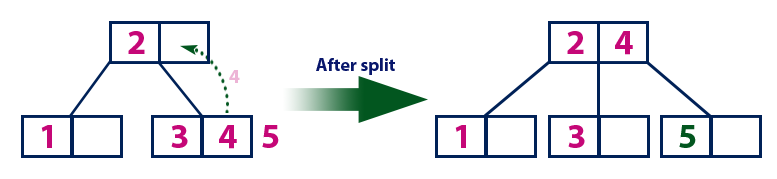
Insertion of 3



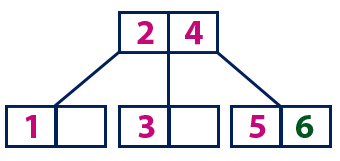
Insertion of 4



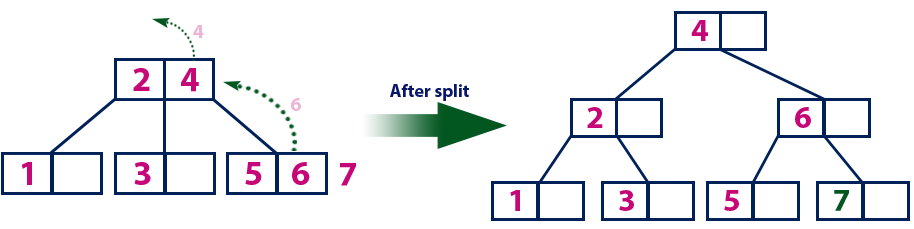
Insertion of 5



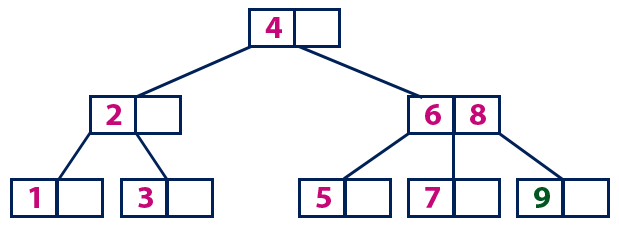
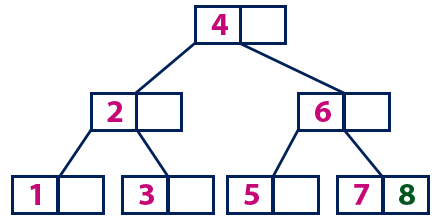
Insertion of 6



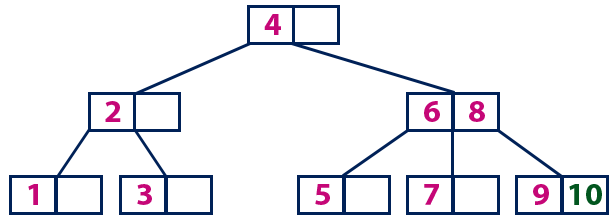
Insertion of 7



Insertion of 8& 9



Insertion of 10



**AVL TREE :-**

AVL tree is a type of binary search tree in which at any given node, absolute difference between heights of left sub-tree and right sub-tree cannot be greater than 1. This property of the AVL tree helps to keep the tree height balanced.

- Worst-case complexity of find: O(log n)

- Worst-case complexity of insert: O(log n)

– A rotation is O(1) and there’s an O(log n) path to root

– (Same complexity even without one-rotation-is-enough fact)

- Worst-case complexity of buildTree: O(n log n)

**RED BLACK TREE:-**

A red-black tree is a binary search tree in which each node is colored red or black such that

-The root is black

-The children of a red node are black

-Every path from the root to a 0-node or a 1-node has the same number of black nodes.

**Advantages RED BLACK TREE**

-Red-black trees are self-balancing so insert, delete, get operations are guaranteed to be O(log(n));

-A simple binary search tree, on the other hand, could potentially become unbalanced, degrading to O(n) performance for Insert, Delete, and Get.

-Particularly useful when inserts and/or deletes are relatively frequent.

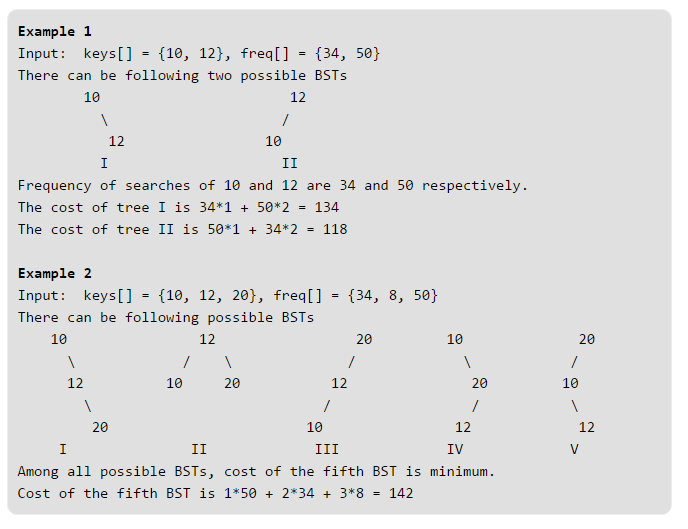
-Relatively low constants in a wide variety of scenarios.

-All the advantages of binary search trees.

**Optimal binary search trees(Dynamic Programming) :-**

- It is called weight balanced binary tree

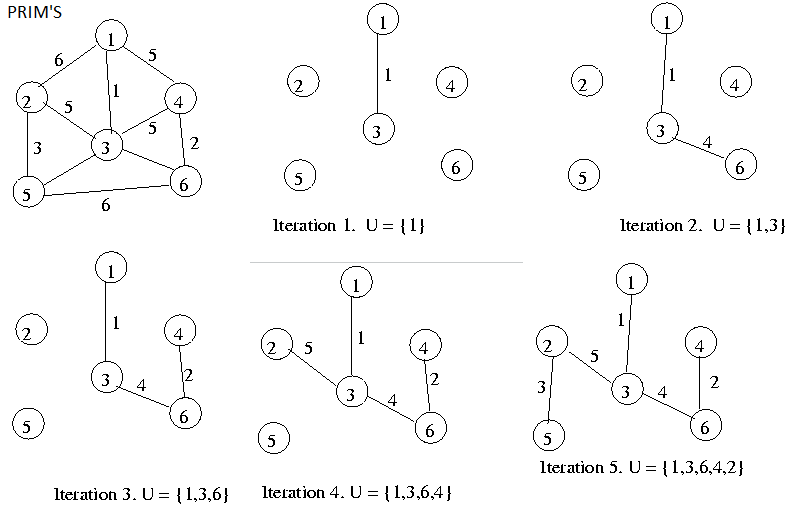
- Given a sorted array keys[0.. n-1] of search keys and an array freq[0.. n-1] of frequency counts, where freq[i] is the number of searches to keys[i]. Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.



* + Sorted set of keys k1,k2,...,kn k1,k2,...,kn
  + Key probabilities: p1,p2,...,pnp1,p2,...,pn
  + What tree structure has lowest expected cost?
  + Cost of searching for node ii: cost(ki)=depth(ki)+1

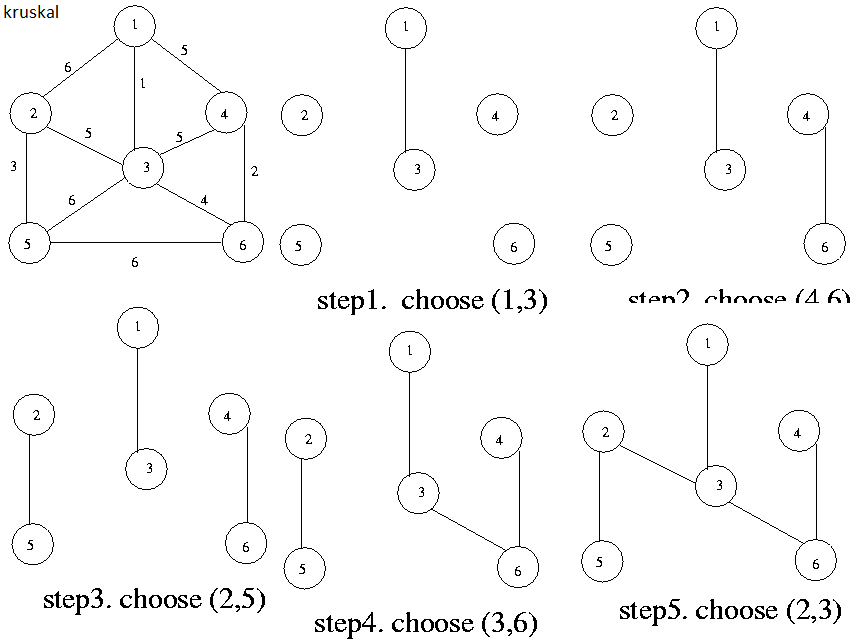
**PRIM’S ALGORITHM(Greedy Alg) :-**

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

****

**KRUSKAL ALGORITHM(Greedy Alg) :-**

Time complexity is O(E log V) time, where E is the number of edges in the graph and V is the number of vertices



**DIJKSTRA ALG (O(V2)):-**

Finding the shortest paths to all the nodes in a graph from a single designated source.

**K-WAY MERGE SORT:-**

-m sorted lists having n elements in total

-A sorted list containing all elements of the m lists

-The problem can be solved in O(n log m) time by using a min heap or a min priority queue

Given k sorted arrays of size n each, merge them and print the sorted output.

Input: k = 3, n = 4

arr[][] = { {1, 3, 5, 7},

{2, 4, 6, 8},

{0, 9, 10, 11}} ;

Output: 0 1 2 3 4 5 6 7 8 9 10 11

A simple solution is to create an output array of size n\*k and one by one copy all arrays to it. Finally, sort the output array using any O(nLogn) sorting algorithm. This approach takes O(nkLognk) time.

We can merge arrays in O(nk\*Logk) time using Min Heap.

**Hashing;-**

Hashing can be used to build, search, or delete from a table.

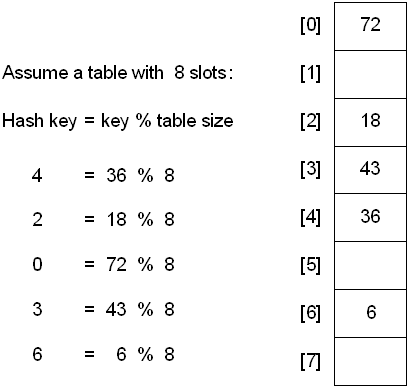
hash key = key % number of slots in the table

When collison occurs, there are two simple solutions:

1.chaining 2. linear probe (aka linear open addressing)

3.Quadratic Probe 4.Double Hashing

Average time to search for an element is O(1), while worst-case time is O(n).



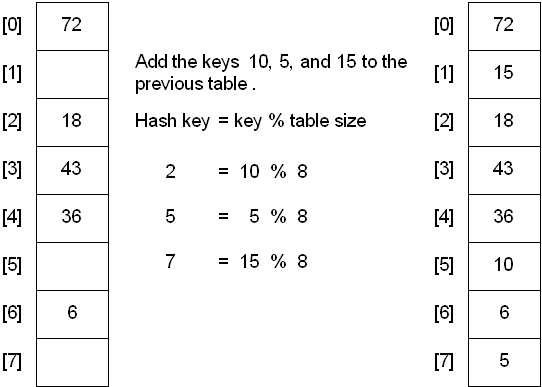
**Hashing with Chains**

When a collision occurs, elements with the same hash key will be chained together. A chain is simply a linked list of all the elements with the same hash key.



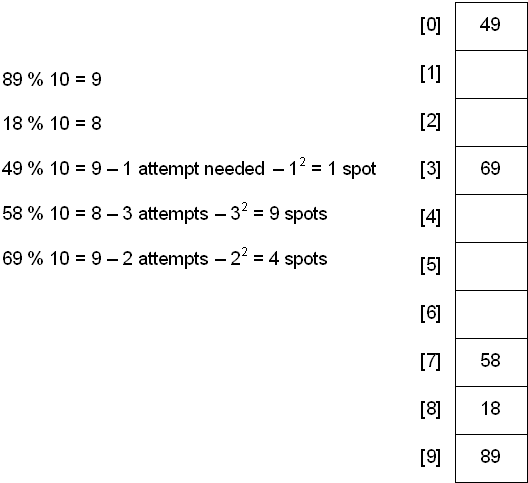
**Hashing with Linear Probe**

When using a linear probe, the item will be stored in the next available slot in the table, assuming that the table is not already full.

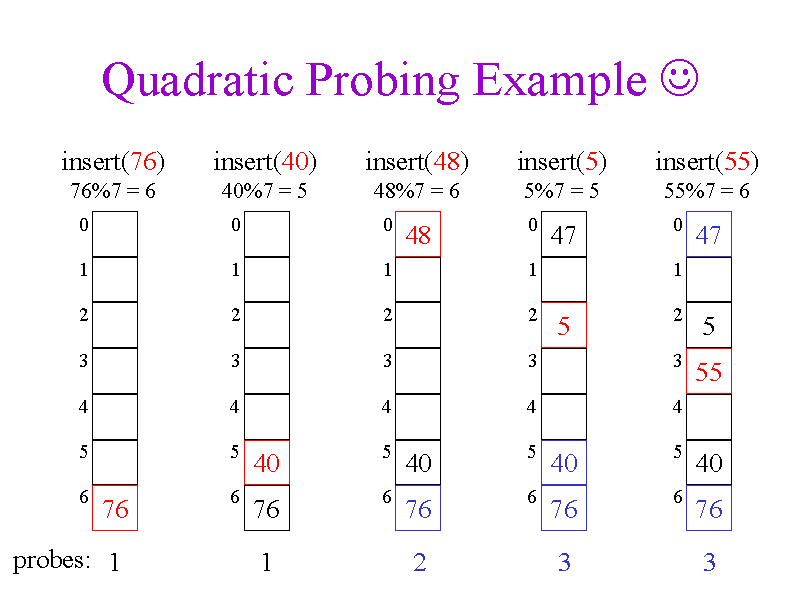


**Hashing with Quadratic Probe**

Quadratic Probing is similar to Linear probing. The difference is that if you were to try to insert into a space that is filled you would first check 1^2 = 1 *element away* then 2^2 = 4 elements away, then 3^2 =9 elements away then 4^2=16 elements away and so on.

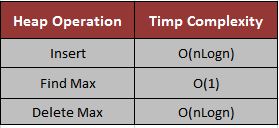


Another Example For Quadratic Probe:-



**HEAP (MIN & MAX) TREE :-**

* The structure of heap should be a complete binary tree

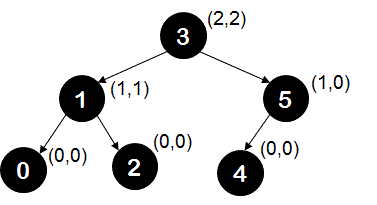
****

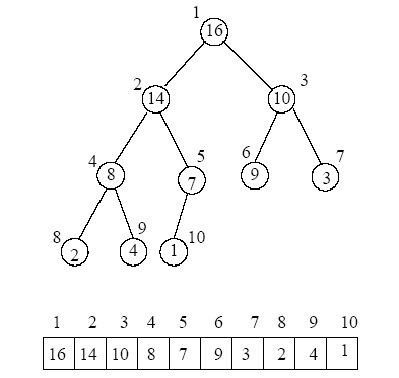
**MIN HEAP :-**



**MAX HEAP :-**





****

Start storing from index 1, not 0.

For any given node at posi­tion i:

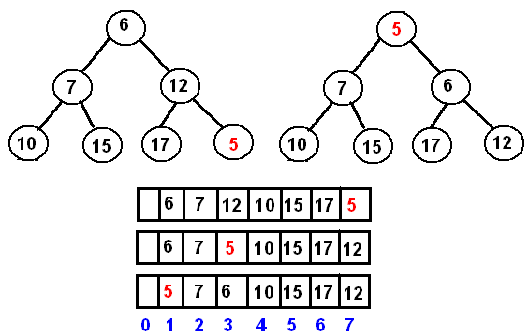
Its Left Child is at [2\*i] if available.

Its Right Child is at [2\*i+1] if available.

Its Parent Node is at [i/2]if available.

**Insertion:-**

The new element is initially appended to the end of the heap (as the last element of the array). The heap property is repaired by comparing the added element with its parent and moving the added element up a level (swapping positions with the parent). This process is called "percolation up". The comparison is repeated until the parent is larger than or equal to the percolating element.



**ACTIVITY SELECTION PROBLEM(Greedy Alg) :-**

You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

Greedy Method it takes O(n logn )

Dynamic Programming takes O(n3)

**Example 1** : Consider the following 3 activities sorted by finish time.

start[] = {10, 12, 20};

finish[] = {20, 25, 30};

A person can perform at most two activities. The maximum set of activities that can be executed is {0, 2} [ These are indexes in start[] and finish[] ]

**Example 2** : Consider the following 6 activities sorted by finish time.

start[] = {1, 3, 0, 5, 8, 5};

finish[] = {2, 4, 6, 7, 9, 9};

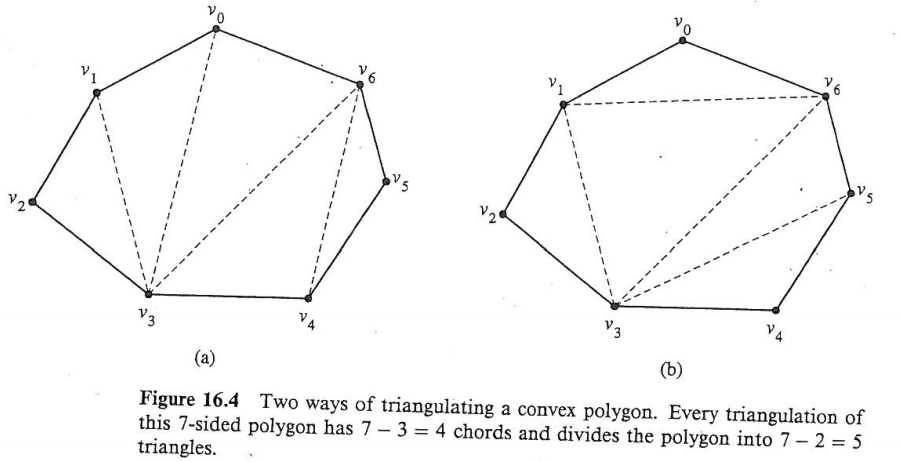
A person can perform at most four activities. The maximum set of activities that can be executed is {0, 1, 3, 4} [ These are indexes in start[] and finish[] ]

**Triangulation of a Convex Polygon(Dynamic Programming) :-**

A triangulation of a polygon can be thought of as a set of chords that divide the polygon into triangles such that no two chords intersect (except possibly at a vertex). This is a triangulation of the same polygon:

A chord is a line segment connecting any two vertices. A chord splits the polygon into two smaller polygons. Note that a chord always divides a convex polygon into two convex polygons

Every triangulation of n-vertex convex polygon has n-3 chords and divides the polygon into n-2 triangles.

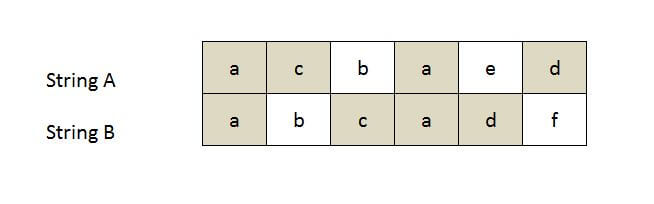


**Longest Common Subsequence(Dynamic Programming): -**

A longest subsequence is a sequence that appears in the same relative order, but not necessarily contiguous(not sub­string) in both the string.

Example1:

String A = "acbaed"; String B = "abcadf";



Longest Common Subsequence(LCS): acad, Length: 4

2. LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.

3. LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

So, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. etc are subsequences of “abcdefg”. So a string of length n has 2^n different possible subsequences.

**Subset Sum Problem (Backtracking):-**

**Objective:** Given a set of positive integers, and a value sum S, find out if there exist a sub­set in array whose sum is equal to given sum S.

**Example1:**int[] A = { 3, 2, 7, 1}, S = 6

Output: True, subset is (3, 2, 1}

**Example2:**Input :arr[] = {2, 3, 5, 6, 8, 10} sum = 10

Output : { 5, 2, 3} {2, 8} {10}

**Example2:** Input :arr[] = {1, 2, 3, 4, 5}sum = 10

Output : { 4, 3, 2, 1 } {5, 3, 2 } { 5, 4, 1}

**Matrix Chain Multiplication(Dynamic Programming):-**

matrix multiplication is associative. For example, if we had four matrices A, B, C, and D, we would have:

(ABC)D = (AB)(CD) = A(BCD) = ....

the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency.

For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations

A(BC) = (10×30×60) + (30×5×60) = 9000 + 18000 = 27000 operations.

Clearly the first parenthesization requires less number of operations.

Input: p[] = {40, 20, 30, 10, 30}

Output: 26000

There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30.

Let the input 4 matrices be A, B, C and D. The minimum number of

multiplications are obtained by putting parenthesis in following way

(A(BC))D --> 20\*30\*10 + 40\*20\*10 + 40\*10\*30

Input: p[] = {10, 20, 30, 40, 30}

Output: 30000

There are 4 matrices of dimensions 10x20, 20x30, 30x40 and 40x30.

Let the input 4 matrices be A, B, C and D. The minimum number of

multiplications are obtained by putting parenthesis in following way

((AB)C)D --> 10\*20\*30 + 10\*30\*40 + 10\*40\*30

Input: p[] = {10, 20, 30}

Output: 6000

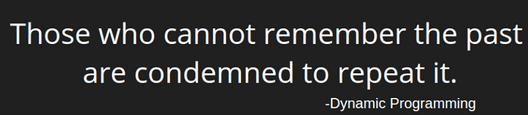
There are only two matrices of dimensions 10x20 and 20x30. So there

is only one way to multiply the matrices, cost of which is 10\*20\*30

**DYNAMIC PROGRAMMING**

(https://www.hackerearth.com/practice/algorithms/dynamic-programming/introduction-to-dynamic-programming-1/tutorial/)

The image below says a lot about Dynamic Programming. So, is repeating the things for which you already have the answer



Some famous Dynamic Programming algorithms are:

1.Unix diff for comparing two files

2.Bellman-Ford for shortest path routing in networks

3.TeX the ancestor of LaTeX

4.WASP - Winning and Score Predictor

The core idea of Dynamic Programming is to avoid repeated work by remembering partial results.

Dynamic programming is basically, recursion plus using common sense. What it means is that recursion allows you to express the value of a function in terms of other values of that function.

Every Dynamic Programming problem has a schema to be followed:

1.Show that the problem can be broken down into optimal sub-problems.

2.Recursively define the value of the solution by expressing it in terms of optimal solutions for smaller sub-problems.

3.Compute the value of the optimal solution in bottom-up fashion.

4.Construct an optimal solution from the computed information.