**Data Structures:**

* **Arrays**
* **Stacks –** *Push, Pop, Peek***.**
* **Queue –** *Enqueue, Dequeue, Peek.*
* **Linked List –** *Use to implement stack and queue.*
* **Tree –** *BST can be implemented using linked list*
* **Tries**
* **Graph**
* **Hash tables –** *Can be implemented using dictionary.*

**Algorithms:**

* **Sorting** – *Bubble Sort Selection Sort Insertion Sort Shell Sort Merge Sort Quick Sort.*
* **Dynamic Programming**
* **BFS DFS Searching – Linear and Binary searching.**
* **Recursion**

**Time Complexity:**

**Time Complexity:**

**Time Complexity: Amount of time taken by algorithm to execute**

**Space complexity – Amount of space required by algorithm.**

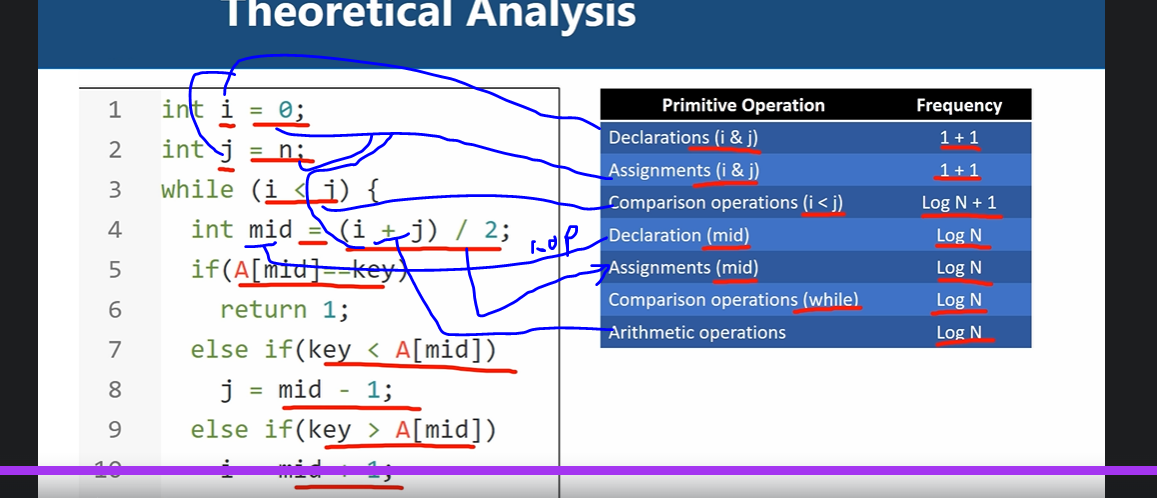
**Running time of algorithms is proportional which increase with size of input .**

**Time complexity measured using**

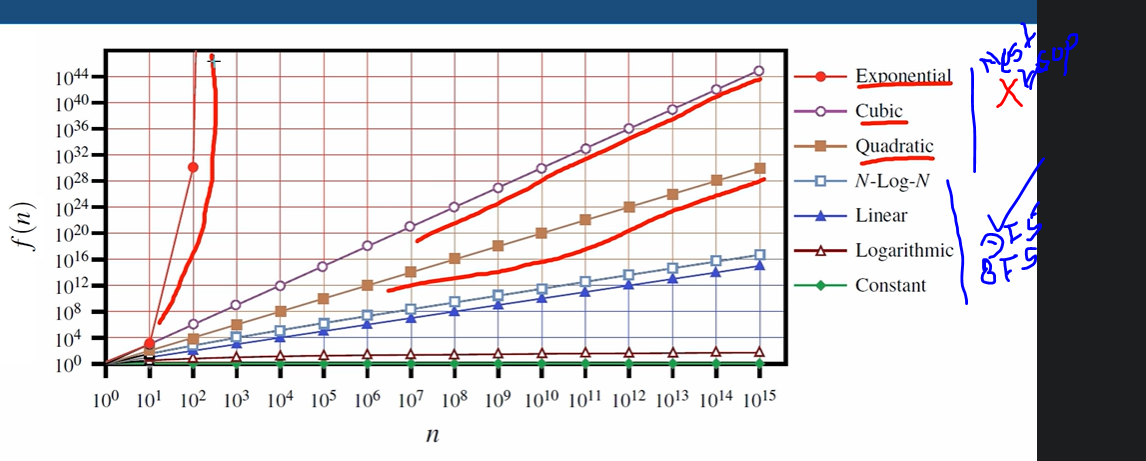
**Experimental analysis – using time function – dependent on hardware OS and all. Limited input**

**Theoretical analysis – Mathematical model where analysis is perform based on description of algorithm. Independent of hardware os . All possible input possible.**

|  |
| --- |
| **Constant execution time** |
| **Declarations** |
| **Assignment** |
| **Arithmetic operation** |
| **Comparison statement** |
| **Accessing element** |
| **Calling function** |
| **Returning function** |



|  |  |  |
| --- | --- | --- |
| **Constant** | **1** | **Single line statement** |
| **Logarithm** | **Log(n)** | **Uncertain loop based on condition** |
| **Linear** | **n** | **Single loop** |
| **n-log-n** | **Nlog(n)** | **Binary tree** |
| **Quadratic** | **N^2** | **Loop inside loop** |
| **Cubic** | **N^3** | **3 loops inside** |
| **Exponential** | **N^n** | **N loop inside** |



Asymptotic Analysis

**Approaching a value**

**F(n) = n^2 + 6n + 5 and n =1000**

**N^2 = 1000000**

**6n – 6000 is insignificant can be ignore**

**Hence f(n) is said to be asymptotically equivalent to n^2,**

**Example**

**Array: [84, 21, 47, 96, 15]**

**0 1 2 3 4**

**Bases on search key**

1. **84 – Best case – Lower bound – Big-Omega - f(n) > = cg(n)**
2. **15 – worst case – Upper bound - Big-Oh O() - f(n) <= cg(n) for n>= n0**
3. **47 – average case – between lower and upper bound – big-Theta cg(n) <= f(n) <= cg(n) for n>= n0**

**Space Complexity**

|  |  |
| --- | --- |
| **Primitive data types** | **Bytes** |
| **Boolean** | **1** |
| **Byte** | **1** |
| **Char** | **2** |
| **Int** | **4** |
| **Float** | **4** |
| **Long** | **8** |
| **Double** | **8** |
| **Array int of size n** | **4\*n** |

**Recursion**

**Function make call itself called reclusive function, which has condition to stop recursion.**

**Time complexity of recursive function.**

**Its linear O(n)**

**Function calculate(n){ -- T(n)**

**If(n>0){ -- 1**

**K = n \* n; --1**

**Print(k); --1**

**Calculate(n-1) --n-1**

**}**

**}**

**Calculate time complexity of above function**

**T(n) = T(n - 1) + 1+1+1= T(n - 1) + 3**

**T(n) = T(n -1 ) + 1**

**T(n-1) = T(n-1-1) +1**

**T(n-2)= T(n-2-1)+1**

**So on**

**T(n) = T(n-1)+1**

**T(n) = T(n-2)+1+1= T(n-2)+2**

**T(n)= T(n-3)+1+2= T(n-3)+3**

**So on**

**T(n) = T(n - K) + k**

**Then n-k = 0 => k=n**

**T(n) = T(n-n) + n**

**= T(0) +n**

**= 1 + n**

**Hence O(n) which is linear.**

**Types of recursion**

* **Tail recursion – execute login first then call recursive**
* **Head recursion – First goes on calling all recursion and at last start executing login on reverse**
* **Tree recursion- recursive function call more then once called tree recursion**
* **Indirect recursion – More then one functions calling each other**

|  |  |
| --- | --- |
| **Function calculate(n){ -- T(n)**  **If(n>0){ -- 1**  **K = n \* n; --1**  **Print(k); --1**  **Calculate(n-1) --n-1**  **}**  **}** | **Tail recursion** |
| **Function calculate(n){ -- T(n)**  **If(n>0){ -- 1**  **Calculate(n-1) --n-1**  **K = n \* n; --1**  **Print(k); --1**  **}**  **}** | **Head recursion** |
| **Function calculate(n){ -- T(n)**  **If(n>0){ -- 1**  **Calculate(n-1) --n-1**  **K = n \* n; --1**  **Print(k); --1**  **Calculate(n-1) --n-1**  **}**  **}** | **Tree recursion O(2^n)**  **2^0 + 2^1 + 2^2 + 2^3 ------2^n = O(2^n)**  **=2^n+1 - 1 = O(2^n+1 - 1) = O(2^n)** |
| **Function calculateA(n){ -- T(n)**  **If(n>0){ -- 1**  **CalculateB(n-1) --n-1**  **}**  **}**  **Function calculateB(n){ -- T(n)**  **If(n>0){ -- 1**  **CalculateA(n-1) --n-1**  **}**  **}** | **Indirect recursion**  **There could be menu function but at last last function call first function.** |

**Recursive sum of n natural numbers n(n+1) / 2 == sum O(1)**

**Function Sum(n){ if(n == 1){ return 1;} return Sum(n – 1);}} O(n)**

**Factorial of number 5! = 5 \* 4 \* 3 \* 2 \* 1 = ? O(n)**

**public int factorialrecursive(int n)**

**{**

**if (n == 0)**

**return 1;**

**return factorialrecursive(n - 1) \* n;**

**}**

**Searching Algorithms:**

**Linear search:** *Sequentially search each element in array to find value***.**

**public int linearsearch(int[] A, int n, int key)**

**{**

**int index = 0;**

**while (index < n)**

**{**

**if (A[index] == key)**

**return index;**

**index = index + 1;**

**}**

**return -1;**

**}**

**C# linear search example Complexity O(n)**

var studentList = new List<Student>() {

new Student(){ Age=21 ,Name="Ganesh" },

new Student(){ Age=22 ,Name="Ramesh" },

new Student(){ Age=32 ,Name="Amir" },

new Student(){ Age=43 ,Name="Sohan" },

new Student(){ Age=9 ,Name="Rima" },

};

var intList = new List<int>() { 3,4,2,5,7,2,8,6};

bool contains = intList.Contains(3);

bool containsObject = studentList.Contains(new Student() { Age = 9, Name = "Rima" });

bool exists = studentList.Exists(x => x.Age == 21);

int min = intList.Min();

int max = intList.Max();

int youngStudentAge = studentList.Min(x => x.Age);

Student ganesh = studentList.Find(x => x.Name =="Ganesh");

Student lastGanesh = studentList.FindLast(x => x.Name == "Ganesh");

Student lastGanesh2 = studentList.Last(x => x.Name == "Ganesh");

List<Student> students = studentList.FindAll(x => x.Age > 23);

IEnumerable<Student> studentsWhereAge = studentList.Where(x => x.Age > 23);

int index1 = studentList.FindIndex(x => x.Age == 21);

int Lastindex1 = studentList.FindLastIndex(x => x.Age > 21);

int indexOf = intList.IndexOf(2);

int lastIndexOf = intList.LastIndexOf(2);

bool isTrueForAll = studentList.TrueForAll(x => x.Age > 30);

bool isTrueForAll2 = studentList.TrueForAll(x => x.Age > 7);

bool all = studentList.All(x => x.Age == 44);

bool all2 = studentList.All(x => x.Age >4);

bool areThereAny = studentList.Any(x => x.Age == 21);

int count = studentList.Count(x => x.Age == 21);

Student firstGanesh = studentList.First(x => x.Name == "Ganesh");

Student singleGanesh = studentList.Single(x => x.Name == "Ganesh");//Exception when more then one elements

**Binary Search: Search sorted array by finding mid element and breaking array recursively until we find element. Much better than linear search no need to traverse each element.**

**O(logn)**

**public int binarysearch(int[] A, int n, int key)**

**{**

**int l = 0;**

**int r = n - 1;**

**while (l <= r)**

**{**

**int mid = (l + r) / 2;**

**if (key == A[mid])**

**return mid;**

**else if (key < A[mid])**

**r = mid - 1;**

**else if (key > A[mid])**

**l = mid + 1;**

**}**

**return -1;**

**}**

**Binary Search recursive**

**public int binarysearch(int[] A, int key, int l, int r)**

**{**

**if (l > r)**

**return -1;**

**else**

**{**

**int mid = (l + r) / 2;**

**if (key == A[mid])**

**return mid;**

**else if (key < A[mid])**

**return binarysearch(A, key, l, mid - 1);**

**else if (key > A[mid])**

**return binarysearch(A, key, mid + 1, r);**

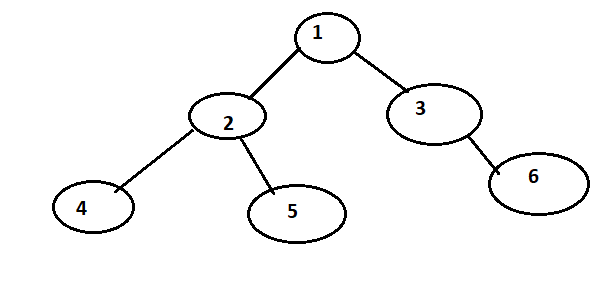
**}**

**return -1;**

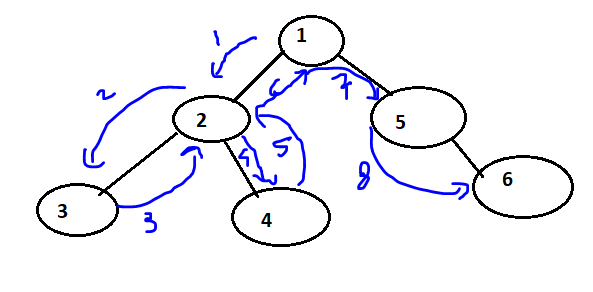
**}**

**Graph and Tree Traversal O(n) - > Traversal meaning visiting each element in tree or graph for this we have BFS and DFS. Breadth first search and depth first search.**

**BFS – visit Root then left – right – children in same order.**



**DFS : -** Root – Left – root -right



**Example**

**9**

1. **10**

**2 3 4 5**

**BFS order – [9, 1, 10, 2, 3, 4, 5]**

**DFS order – [9, 1, 2, 3, 10, 4, 5]**

**BFS:**

public static List<int> BeardthFirstSearch(Node root)

{

//Root - > left - > Right - and repeaset untill leaf nodes

List<int> list = new List<int>();

Node tempNode = root;

Queue<Node> queue = new Queue<Node>();

queue.Enqueue(tempNode);

while (queue.Count > 0)

{

tempNode = queue.Dequeue();

list.Add(tempNode.Value);

if (tempNode.Left != null)

{

queue.Enqueue(tempNode.Left);

}

if (tempNode.Right != null)

{

queue.Enqueue(tempNode.Right);

}

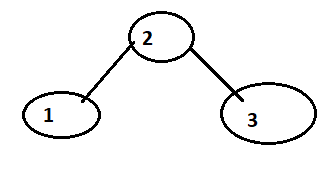
}

return list;

}

**DFS:**

**Depth First Search can be implemented in 3 different ways**



* **In order [ 1, 2 ,3]**
* **Preorder: [2, 1, 3]**
* **Post order [1, 3, 2]**

**Example**

**9**

1. **10**

**2 3 4 5**

**In order [2, 1, 3,9,4,10,5 ]**

**Preorder [9,1,2,3,10,4,5]**

**Post order [2,3,1,4,5,10,9]**

**Sorting**

**Types of sorting**

* **Selection Sort**
* **Insertion sort**
* **Bubble sort**
* **Merge Sort**
* **Quick Sort**
* **Shell Sort**
* **Heap Sort**
* **Count Sort**
* **Bucket Sort**
* **Radix Sort**

**Stable and Unstable sorting algorithms:**

**When we try to sort duplicate element original index order of duplicate element is preserve call stable and if duplicate element order changes in sort called unstable**

**Stable unstable dose not matter for int array but object sorting matters two same value object are diff in memory .**

**Selection Sort: Select min element and place right position. This happen by swapping min element to left side O(n^2)**

public void selectionsort(int[] A, int n)

{

for (int i = 0; i < n - 1; i++)

{

int position = i;

for (int j = i + 1; j < n; j++)

if (A[j] < A[position])

position = j;

int temp = A[i];

A[i] = A[position];

A[position] = temp;

}

}

**Insertion Sort: Get sorting number find the position at which we can insert it and repeat relative order of duplicate element is preserve hence it is stable algorithm**

**O(n^2)**

public static void InsertionSorts(int[] array,int n)

{

for(int i = 1; i < n; i++)

{

int position = i;

int insertValue = array[position];

while(position > 0 && array[position-1] > insertValue)

{

array[position] = array[position - 1];

position = position - 1;

}

array[position] = insertValue;

}

}

**Bubble Sort: Compare consecutive element if element to the left is greeter then right swap them repeat; find out largest element and keep swapping until it move last and bubble out by reducing index.**

**O(n^2)**

**Logic is keep compare and swapping adjacent element and bubble out largest at end**

public static void BubbleSotrs(int[] arra)

{

int length = arra.Length-1;

for(int i = length; i >=0; i--)

{

for (int j = 0; j < length; j++)

{

if(arra[j] > arra[j+1])

{

int temp = arra[j+1];

arra[j+1] = arra[j];

arra[j] = temp;

}

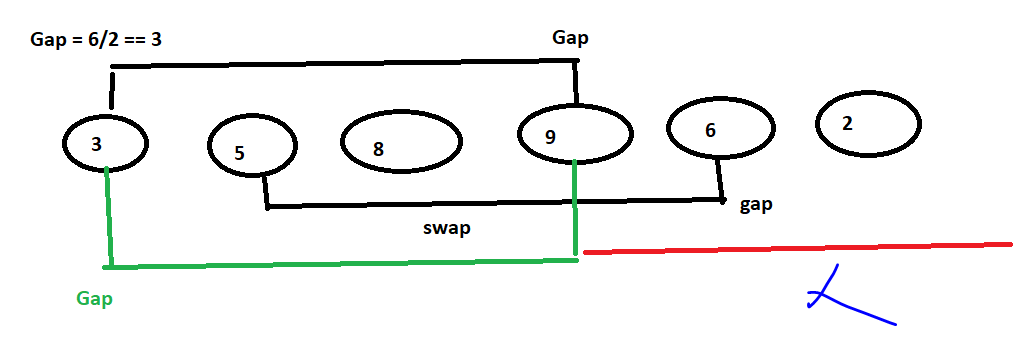
}

length--;

}

}

**Shell Sort: Select and insert element at its proper position using gap gap = n/2 then select i=0 to gap elements**



**Sort until last again Gap/2 and so on till gap ==1;**

**O(nlogn)**

public static void ShellSort(int[] array, int n)

{

int gap = n / 2;

while (gap > 0)

{

int i = gap;

while(i < n)

{

int temp = array[i];

int j = i - gap;

while(j>=0 && array[j] > temp)

{

array[j + gap] = array[j];

j = j - gap;

}

array[j + gap] = temp;

i = i + 1;

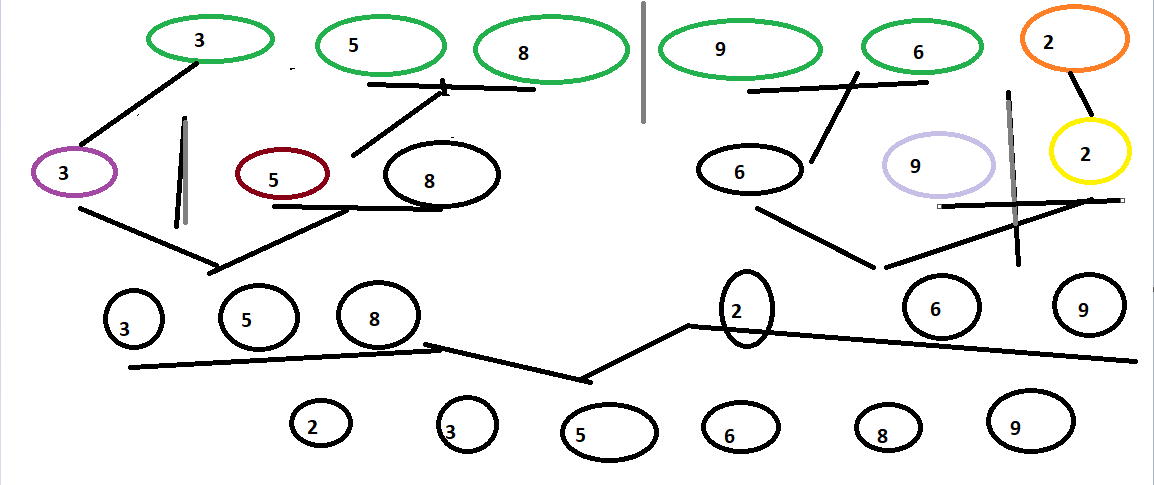
}

gap = gap / 2;

}

}

**Merge Sort: Divide collection into smaller sub sets and recursively sort sub sets then combine the sorted subarray. divide and conquer- O(nlogn)**



public static void MergeSort(int[] A, int left, int right)

{

if (left < right)

{

int mid = (left + right) / 2;

MergeSort(A, left, mid);

MergeSort(A, mid + 1, right);

Merge(A, left, mid, right);

}

}

public static void Merge(int[] A, int left, int mid, int right)

{

int i = left;

int j = mid + 1;

int k = left;

int[] B = new int[right + 1];

while (i <= mid && j <= right)

{

if (A[i] < A[j])

{

B[k] = A[i];

i = i + 1;

}

else

{

B[k] = A[j];

j = j + 1;

}

k = k + 1;

}

while (i <= mid)

{

B[k] = A[i];

i = i + 1;

k = k + 1;

}

while (j <= right)

{

B[k] = A[j];

j = j + 1;

k = k + 1;

}

for (int x = left; x < right + 1; x++)

A[x] = B[x];

}

**Quick Sort:**

**Divide the collection in sub arrays. Partition based on pivot sort partitions**

**Pivot element selected such that all left element < pivot > right elements**

**Always start i=0 and j = count and pivot = 0**

**Compare I and j with pivot and swap i and j increment I decrement j and so on**

**Then pivot element at last will be at mid and all small element on left and all larger element on right of pivot then again recursively perform Quick sort on left and right sub arrays**

**O(n^2)**

public static void QuickSort(int[] A, int low, int high)

{

if (low < high)

{

int pi = Partition(A, low, high);

QuickSort(A, low, pi - 1);

QuickSort(A, pi + 1, high);

}

}

public static int Partition(int[] A, int low, int high)

{

int pivot = A[low];

int i = low + 1;

int j = high;

do

{

while (i <= j && A[i] <= pivot)

i = i + 1;

while (i <= j && A[j] > pivot)

j = j - 1;

if (i <= j)

Swap(A, i, j);

} while (i < j);

Swap(A, low, j);

return j;

}

public static void Swap(int[] A, int i, int j)

{

int temp = A[i];

A[i] = A[j];

A[j] = temp;

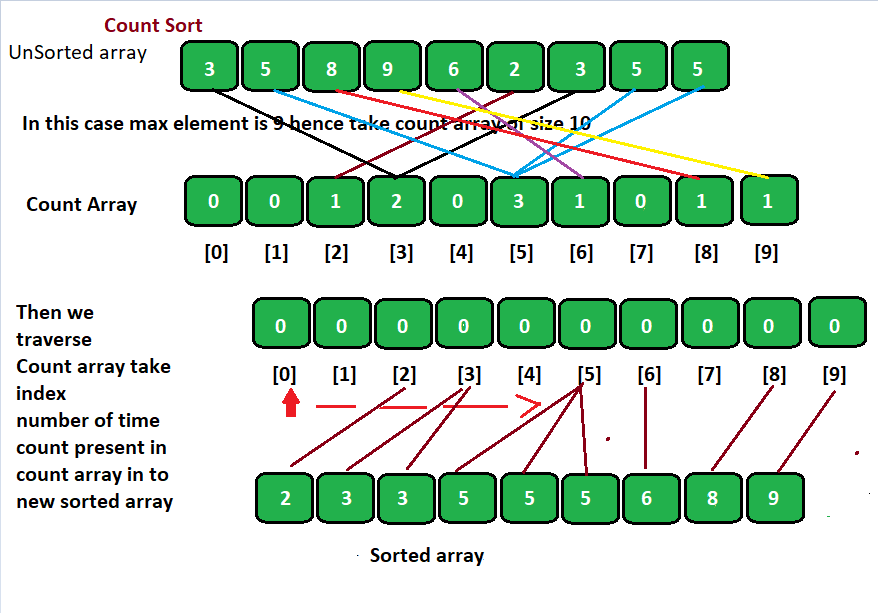
}

**Count Sort - How does it Work**

**Count sort is I index based sorting technic in which we will take a count of element and store in index array at same index number as element**

**Then while sorting we traverse index array and find out count if count is greater then 1 add number of count element in sorted array at number of count times and decrease index array count to 0;**

**Count or index array size is equal to largest element in unsorted array**



**Count Sort - Algorithm and Analysis**

**countSort(A,n)**

**maxSize= Max(A);**

**countArray = [maxSize + 1]**

**for i =0 i< n;i ++**

**countArray[A[i]] = countArray[A[i]] + 1;**

**I,j=0;**

**While(i < maxSize + 1)**

**If countArray[i] > 0 then**

**A[j++] = I;**

**countArray[i] = countArray[i] -1**

**else**

**i = I + 1;**

**Time complexity is O(n)**

**Count sort code**

public static int[] CountSort(int[] array, int n)

{

int size = array.Max();

int[] Counts = new int[size + 1];

for (int i = 0; i < n; i++)

{

Counts[array[i]] = Counts[array[i]] + 1;

}

int m = 0, j = 0;

while(m < size + 1)

{

if(Counts[m] > 0)

{

array[j++] = m;

Counts[m] = Counts[m] - 1;

}

else

{

m = m + 1;

}

}

return array;

}

**Radix Sort - How does it Work**

**O(n)**

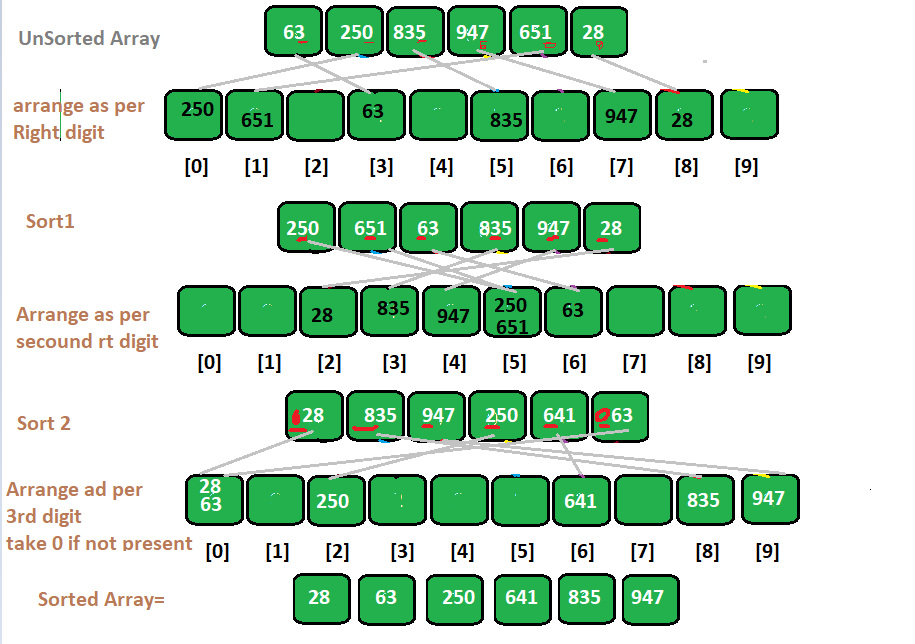
**Radix sort is based on number of digits**

**First take right most least significant element and as per element put it at same index in other array**

**Repeat this for all numbers if duplicate number add in same index as an array**

**Arrange array is same order we placed in second index array**

**Then select second element from right and repeat until all element and all digits covers**



**Radix Sort - Algorithm and Analysis**

**(A,n)**

**maxelement = max(A);**

**digits = findDigits(maxElement);**

**bins[] = size 10**

**for(i=0;I digits;i++)**

**for(j =0;j< n;j++)**

**e = (A[j] / pow(10,i)) % 10**

**bins[e].append(A[j])**

**k=0;**

**for x =0 ; x< 10;x++**

**A[k] = bins[x].remove()**

**K = k+1;**

**Radix Sort Code**

namespace Codding

{

public class RadixSorts

{

public static int[] radixSort(int[] A, int n)

{

int maxElement = A.Max();

int digits = maxElement.ToString().Length;

List<List<int>> temp = new List<List<int>>() {

new List<int>(){ },new List<int>(){ },new List<int>(){ },new List<int>(){ },new List<int>(){ },

new List<int>(){ },new List<int>(){ },new List<int>(){ },new List<int>(){ },new List<int>(){ },

};

for(int i=0;i< digits; i++)

{

for (int j = 0; j < n; j++)

{

int element = (A[j] / (int)Math.Pow(10, i)) % 10;

List<int> temp1 = temp[element];

temp1.Add(A[j]);

temp1.Sort();

temp[element] = temp1;

}

int k = 0;

for(int x = 0; x < 10; x++)

{

List<int> ls= temp[x];

temp[x] = new List<int>();

foreach (int e in ls)

{

A[k] = e;

k++;

}

}

}

return A;

}

}

**Bucket Sort - How does it Work**

**In bucket sort we maintain bucket array of length 10**

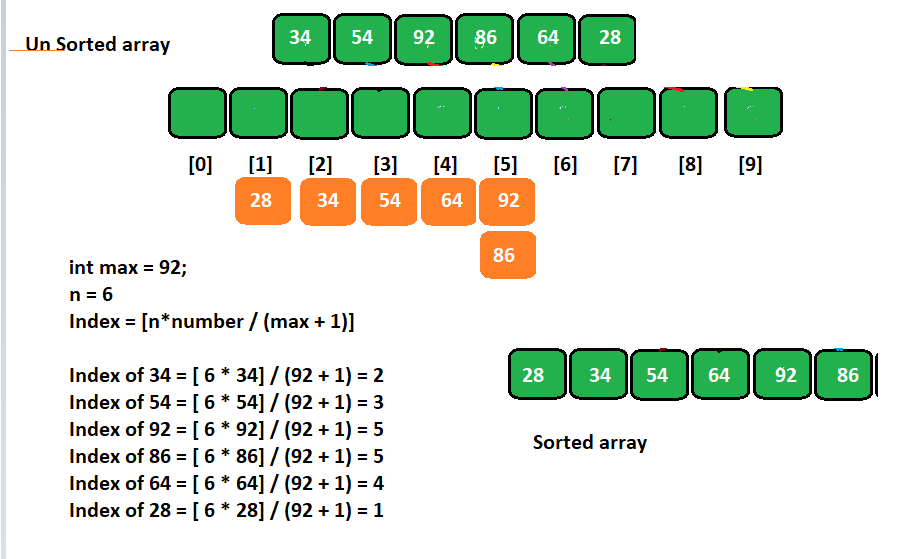
**We calculate index of each unsorted array using function index = (n\*element)/max +1**

**Then at the index in bucket array we store that element in case of duplicate index we store list in bucket**

**It Uses an array of buckets to perform sorting**

**Insert elements in the buckets according to index computed**

**Sort non – empty bucket using insertion sort**

**Traverse buckets in sequence and store elements back into array** 

**Bucket Sort - Algorithm and Analysis**

**O(n^2)**

**Bucketsort(A,n)**

**Max = max(A)**

**Buckets = [];**

**For i=0 i< n i++**

**J = n \* A[i] / Max +1**

**Buckets[j] = A[i]**

**For i=0; i < 10 ;i++**

**insertionSort(buckets[i])**

**k=0;**

**for i=0;i<10;i++**

**A[k] = bucktes[i].remove()**

**K = k +1;**

**Bucket Sort Code sample**

public static List<int> BucketSort(int[] A, int n)

{

List<int> sortedArray = new List<int>();

int numOfBuckets = 10;

//Create buckets

List<int>[] buckets = new List<int>[numOfBuckets];

for (int i = 0; i < numOfBuckets; i++)

{

buckets[i] = new List<int>();

}

//Iterate through the passed array

//and add each integer to the appropriate bucket

for (int i = 0; i < A.Length; i++)

{

int bucket = (A[i] / numOfBuckets);

buckets[bucket].Add(A[i]);

}

//Sort each bucket and add it to the result List

for (int i = 0; i < numOfBuckets; i++)

{

List<int> temp = InsertionSort(buckets[i]);

sortedArray.AddRange(temp);

}

return sortedArray;

}

//Insertion Sort

public static List<int> InsertionSort(List<int> input)

{

for (int i = 1; i < input.Count; i++)

{

//2. Store the current value in a variable

int currentValue = input[i];

int pointer = i - 1;

//3. As long as we are pointing to a valid value in the array...

while (pointer >= 0)

{

//4. If the current value is less

// than the value we are pointing at...

if (currentValue < input[pointer])

{

//5. Move the pointed-at value up one space,

// and insert the current value

// at the pointed-at position.

input[pointer + 1] = input[pointer];

input[pointer] = currentValue;

}

else break;

}

}

return input;

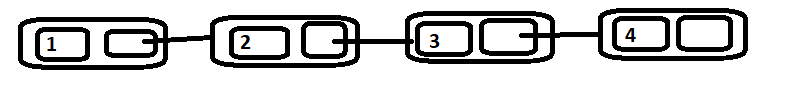
}

**Summary of Complexities - Index Based Sorting Algorithms**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithm** | **Best case** | **Average case** | **Worst case** | **Space** | **Stable** |
| **Count Sort** | **O(n)** | **O(n)** | **O(n)** | **O(n)** | **Yes** |
| **Bucket Sort** | **O(n)** | **O(n^2)** | **O(n^2)** | **O(n)** | **Yes** |
| **Radix Sort** | **O(n)** | **O(n)** | **O(n)** | **O(n)** | **Yes** |

**Linked List**

**Array sequentially store element in memory hence size need to declare. whereas linked list element placed at diff location connected by pointer**



**To create linked list we need Node we can define node using node class**

public class Node

{

public int Value { get; set; }

public Node Next { get; set; }

public Node(int \_value)

{

Value = \_value;

Next = null;

}

}

First node known as Head and Last node known as tail of linked list

Head is also called root node.

using System;

using System.Collections.Generic;

using System.Text;

namespace Codding

{

public class Node

{

public int element;

public Node next;

public Node(int e, Node n)

{

element = e;

next = n;

}

}

public class LinkedList

{

private Node head;

private Node tail;

private int size;

public LinkedList()

{

head = null;

tail = null;

size = 0;

}

public int length()

{

return size;

}

public bool isEmpty()

{

return size == 0;

}

public void addAny(int e, int position)

{

if (position <= 0 || position >= size)

{

Console.WriteLine("Invalid Position");

return;

}

Node newest = new Node(e, null);

Node p = head;

int i = 1;

while (i < position - 1)

{

p = p.next;

i = i + 1;

}

newest.next = p.next;

p.next = newest;

size = size + 1;

}

public int removeFirst()

{

if (isEmpty())

{

Console.WriteLine("List is Empty");

return -1;

}

else

{

int e = head.element;

head = head.next;

size = size - 1;

if (isEmpty())

tail = null;

return e;

}

}

public int removeLast()

{

if (isEmpty())

{

Console.WriteLine("List is Empty");

return -1;

}

if(size == 1)

{

var temp = head;

head = null;

tail = null;

size = size - 1;

return temp.element;

}

Node p = head;

int i = 1;

while (i < size - 1)

{

p = p.next;

i = i + 1;

}

tail = p;

p = p.next;

int e = p.element;

tail.next = null;

size = size - 1;

return e;

}

public int removeAny(int position)

{

if (position <= 0 || position >= size - 1)

{

Console.WriteLine("Invalid Position");

return -1;

}

Node p = head;

int i = 1;

while (i < position - 1)

{

p = p.next;

i = i + 1;

}

int e = p.next.element;

p.next = p.next.next;

size = size - 1;

return e;

}

public int search(int key)

{

Node p = head;

int index = 0;

while (p != null)

{

if (p.element == key)

return index;

p = p.next;

index = index + 1;

}

return -1;

}

public void addFirst(int e)

{

Node newest = new Node(e, null);

if (isEmpty())

{

head = newest;

tail = newest;

}

else

{

var temp = head;

head = newest;

newest.next = temp;

}

size = size + 1;

}

public void addLast(int e)

{

Node newest = new Node(e, null);

if (isEmpty())

head = newest;

else

tail.next = newest;

tail = newest;

size = size + 1;

}

public void insertsorted(int e)

{

Node newest = new Node(e, null);

if (isEmpty())

head = newest;

else

{

Node p = head;

Node q = head;

while (p != null && p.element < e)

{

q = p;

p = p.next;

}

if (p == head)

{

newest.next = head;

head = newest;

}

else

{

newest.next = q.next;

q.next = newest;

}

}

size = size + 1;

}

public void display()

{

Node p = head;

while (p != null)

{

Console.Write(p.element + " --> ");

p = p.next;

}

Console.WriteLine();

}

}

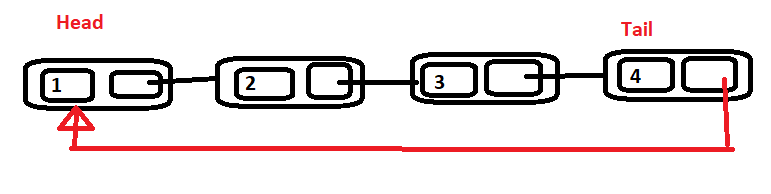
}

**Circular Linked List?**

**Next reference of last node will point to first node**

**Tail.Next = Head;**

**Circular linked list dose not have start and end but still we maintain head and tail.**



public class Node

{

public int element;

public Node next;

public Node(int e, Node n)

{

element = e;

next = n;

}

}

class CircularLinkedList

{

private Node head;

private Node tail;

private int size;

public CircularLinkedList()

{

head = null;

tail = null;

size = 0;

}

public int length()

{

return size;

}

public bool isEmpty()

{

return size == 0;

}

public void addLast(int e)

{

Node newest = new Node(e, null);

if (isEmpty())

{

newest.next = newest;

head = newest;

}

else

{

newest.next = tail.next;

tail.next = newest;

}

tail = newest;

size = size + 1;

}

public void addFirst(int e)

{

Node newest = new Node(e, null);

if (isEmpty())

{

newest.next = newest;

head = newest;

tail = newest;

}

else

{

tail.next = newest;

newest.next = head;

head = newest;

}

size = size + 1;

}

public void addAny(int e, int position)

{

if (position <= 0 || position >= size)

{

Console.WriteLine("Invalid Position");

return;

}

Node newest = new Node(e, null);

Node p = head;

int i = 1;

while (i < position - 1)

{

p = p.next;

i = i + 1;

}

newest.next = p.next;

p.next = newest;

size = size + 1;

}

public int removeFirst()

{

if (isEmpty())

{

Console.WriteLine("Circular List is Empty");

return -1;

}

int e = head.element;

tail.next = head.next;

head = head.next;

size = size - 1;

if (isEmpty())

{

head = null;

tail = null;

}

return e;

}

public int removeLast()

{

if (isEmpty())

{

Console.WriteLine("Circular List is Empty");

return -1;

}

Node p = head;

int i = 1;

while (i < length() - 1)

{

p = p.next;

i = i + 1;

}

tail = p;

p = p.next;

tail.next = head;

int e = p.element;

size = size - 1;

return e;

}

public int removeAny(int position)

{

if (position <= 0 || position >= size - 1)

{

Console.WriteLine("Invalid Position");

return -1;

}

Node p = head;

int i = 1;

while (i < position - 1)

{

p = p.next;

i = i + 1;

}

int e = p.next.element;

p.next = p.next.next;

size = size - 1;

return e;

}

public void display()

{

Node p = head;

int i = 0;

while (i < length())

{

Console.Write(p.element + "-->");

p = p.next;

i = i + 1;

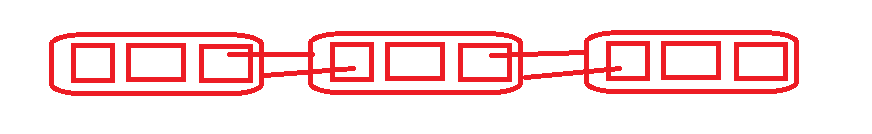
}

Console.WriteLine();

}

**Doubly Linked List?**

**Linked list in which node point to next and prev node.**



**We can create doubly linked list using below Node**

**Public class Node{**

**Public int Value {get;set;}**

**Public Node Next {get;set;}**

**Public Node Prev {get;set;}**

**Public Node(int val){**

**Value=val;**

**Next=null;**

**Prev= null;**

**}**

**}**

namespace LearnDSAlgorithms

{

public class Node

{

public int element;

public Node next;

public Node prev;

public Node(int e, Node n, Node p)

{

element = e;

next = n;

prev = p;

}

}

class DoublyLinkedList

{

private Node head;

private Node tail;

private int size;

public DoublyLinkedList()

{

head = null;

tail = null;

size = 0;

}

public int length()

{

return size;

}

public bool isEmpty()

{

return size == 0;

}

public void addLast(int e)

{

Node newest = new Node(e, null, null);

if (isEmpty())

{

head = newest;

tail = newest;

}

else

{

tail.next = newest;

newest.prev = tail;

tail = newest;

}

size = size + 1;

}

public void addFirst(int e)

{

Node newest = new Node(e, null, null);

if (isEmpty())

{

head = newest;

tail = newest;

}

else

{

newest.next = head;

head.prev = newest;

head = newest;

}

size = size + 1;

}

public void addAny(int e, int position)

{

if (position <= 0 || position >= size)

{

Console.WriteLine("Invalid Position");

return;

}

Node newest = new Node(e, null, null);

Node p = head;

int i = 1;

while (i < position - 1)

{

p = p.next;

i = i + 1;

}

newest.next = p.next;

p.next.prev = newest;

p.next = newest;

newest.prev = p;

size = size + 1;

}

public int removeFirst()

{

if (isEmpty())

{

Console.WriteLine("Doubly List is Empty");

return -1;

}

int e = head.element;

head = head.next;

size = size - 1;

if (isEmpty())

tail = null;

else

head.prev = null;

return e;

}

public int removeLast()

{

if (isEmpty())

{

Console.WriteLine("Doubly List is Empty");

return -1;

}

int e = tail.element;

tail = tail.prev;

tail.next = null;

size = size - 1;

return e;

}

public int removeAny(int position)

{

if (position <= 0 || position >= size - 1)

{

Console.WriteLine("Invalid Position");

return -1;

}

Node p = head;

int i = 1;

while (i < position - 1)

{

p = p.next;

i = i + 1;

}

int e = p.next.element;

p.next = p.next.next;

p.next.prev = p;

size = size - 1;

return e;

}

public void display()

{

Node p = head;

while (p != null)

{

Console.Write(p.element + "-->");

p = p.next;

}

Console.WriteLine();

}

**Stack**

**Stack is collection worked based on LIFO basis.**

**Push Pop and Peek are operations of statck**

**Application of stack**

* **Used to store history of web pages visited by user**
* **We can do undo operation in editing**
* **Html tag mapping {{}}{}}**
* **Evaluate arithmetic operation**
* **Parenthesis matching**
* **Infix to post fix conversion in math**

**Stack Abstract data type:**

* **Push: Add element at top of stack**
* **Pop: Remove and return element at top of stack. Which is inserted at last.**
* **Peek: Return top element in stack. Which is inserted at last.**
* **Len: Return number of elements in stack**
* **isEmpty: check if stack is empty**

**We can implement stack using Array and linked list.**

**Implement stack using arrays:**

public class StacksArray

{

int[] data;

int top;

public StacksArray(int n)

{

data = new int[n];

top = 0;

}

public int length()

{

return top;

}

public bool isEmpty()

{

return top == 0;

}

public bool isFull()

{

return top == data.Length;

}

public void push(int e)

{

if (isFull())

{

Console.WriteLine("Stack if Full/Overflow");

return;

}

else

{

data[top] = e;

top = top + 1;

}

}

public int pop()

{

if (isEmpty())

{

Console.WriteLine("Stack is Empty/UnderFlow");

return -1;

}

int e = data[top - 1];

top = top - 1;

return e;

}

public int peek()

{

if (isEmpty())

{

Console.WriteLine("Stack is Empty");

return -1;

}

return data[top - 1];

}

public void display()

{

for (int i = 0; i < top; i++)

Console.Write(data[i] + "--");

Console.WriteLine();

}

}

**Implement stack using Linked List:**

**We perform add delete at any one end head or tail.**

**But removing element at tail will take O(n)**

**And removing element at Head will take O(1)**

**Hence we will implement stack using head.**

public class Node

{

public int element;

public Node next;

public Node(int e, Node n)

{

element = e;

next = n;

}

}

class StacksLinked

{

Node top;

int size;

public StacksLinked()

{

top = null;

size = 0;

}

public int length()

{

return size;

}

public bool isEmpty()

{

return size == 0;

}

public void push(int e)

{

Node newest = new Node(e, null);

if (isEmpty())

{

top = newest;

}

else

{

newest.next = top;

top = newest;

}

size = size + 1;

}

public int pop()

{

if (isEmpty())

{

Console.WriteLine("Stack is empty");

return -1;

}

int e = top.element;

top = top.next;

size = size - 1;

return e;

}

public int peek()

{

if (isEmpty())

{

Console.WriteLine("Stack is empty");

return -1;

}

return top.element;

}

public void display()

{

Node p = top;

while (p != null)

{

Console.Write(p.element + "-->");

p = p.next;

}

Console.WriteLine();

}

**Queue:**

**Queue is collection of object**

**FIFO: First in First Out**

**Fundamental operation of Queue are**

* **Enqueue**
* **Dequeue**
* **Peek**

**Applications of queue**

**Used in printers**

**Access to memory or shared memory**

**Web server responding to page request manage by queue**

**Some computer application and algorithms**

**Queue abstract data type**

* **Enqueue: Insert element**
* **Dequeue: Remove and return first in element**
* **Peek: Return element inserted at first**
* **Len: Return size of queue**
* **isEmpty: check is queue is empty**

**Queue can be implanted using array and linked list:**

**Queue using arrays:**

public class QueuesArray

{

int[] data;

int front;

int rear;

int size;

public QueuesArray(int n)

{

data = new int[n];

front = 0;

rear = 0;

size = 0;

}

public int length()

{

return size;

}

public bool isEmpty()

{

return size == 0;

}

public bool isFull()

{

return size == data.Length;

}

public void enqueue(int e)

{

if (isFull())

{

Console.WriteLine("Queue is Full");

return;

}

else

{

data[rear] = e;

rear = rear + 1;

size = size + 1;

}

}

public int dequeue()

{

if (isEmpty())

{

Console.WriteLine("Queue is Empty");

return -1;

}

else

{

int e = data[front];

front = front + 1;

size = size - 1;

return e;

}

}

public void display()

{

for (int i = front; i < rear; i++)

Console.Write(data[i] + "--");

Console.WriteLine();

}

}

**Queue using linked list:**

**We can perform insert delete operation only at head or tail**

**Removing element at Tail need O(n)**

**Removing element at head need O(1)**

**Inserting element at head and tail need O(1)**

**Hence, we will choose remove element at head and insert a tail to implement queue.**

public class Node

{

public int element;

public Node next;

public Node(int e, Node n)

{

element = e;

next = n;

}

}

class QueuesLinked

{

Node front;

Node rear;

int size;

public QueuesLinked()

{

front = null;

rear = null;

size = 0;

}

public int length()

{

return size;

}

public bool isEmpty()

{

return size == 0;

}

public void enqueue(int e)

{

Node newest = new Node(e, null);

if (isEmpty())

front = newest;

else

rear.next = newest;

rear = newest;

size = size + 1;

}

public int dequeue()

{

if (isEmpty())

{

Console.WriteLine("Queue is Empty");

return -1;

}

int e = front.element;

front = front.next;

size = size - 1;

if (isEmpty())

rear = null;

return e;

}

public void display()

{

Node p = front;

while (p != null)

{

Console.Write(p.element + "-->");

p = p.next;

}

Console.WriteLine();

}

}

**What are Double Ended Queues DEQue?**

**Double ended queue are collection of object**

**It supports insertion and deletion both front and rear end of queue.**

namespace CoddingDEQ

{

public class Node

{

public int element;

public Node next;

public Node(int e, Node n)

{

element = e;

next = n;

}

}

public class DeQueLinked

{

private Node front;

private Node rear;

private int size;

public DeQueLinked()

{

front = null;

rear = null;

size = 0;

}

public int length()

{

return size;

}

public bool isEmpty()

{

return size == 0;

}

public void addlast(int e)

{

Node newest = new Node(e, null);

if (isEmpty())

front = newest;

else

rear.next = newest;

rear = newest;

size = size + 1;

}

public void addFirst(int e)

{

Node newest = new Node(e, null);

if (isEmpty())

{

front = newest;

rear = newest;

}

else

{

newest.next = front;

front = newest;

}

size = size + 1;

}

public int removeFirst()

{

if (isEmpty())

{

Console.WriteLine("List is Empty");

return -1;

}

int e = front.element;

front = front.next;

size = size - 1;

if (isEmpty())

rear = null;

return e;

}

public int removeLast()

{

if (isEmpty())

{

Console.WriteLine("List is Empty");

return -1;

}

Node p = front;

int i = 1;

while (i < size - 1)

{

p = p.next;

i = i + 1;

}

rear = p;

p = p.next;

int e = p.element;

rear.next = null;

size = size - 1;

return e;

}

public int search(int key)

{

Node p = front;

int index = 0;

while (p != null)

{

if (p.element == key)

return index;

p = p.next;

index = index + 1;

}

return -1;

}

public void display()

{

Node p = front;

while (p != null)

{

Console.Write(p.element + " --> ");

p = p.next;

}

Console.WriteLine();

}

public int first()

{

if (isEmpty())

{

Console.WriteLine("DeQueue is Empty");

return -1;

}

return front.element;

}

public int last()

{

if (isEmpty())

{

Console.WriteLine("DeQueue is Empty");

return -1;

}

return rear.element;

}

}

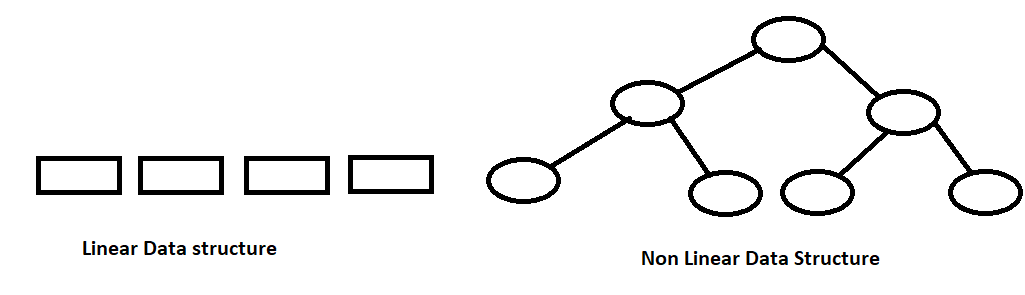
}

**Binary Trees**

**Tree is a non-linear data structure.**

**Linear work sequentially before and after relations**

**Where nonlinear relations are tree based or graph based**



**Non linear data structure relations are in hierarchical order .**

**Arrays or linked list implementation become easy using Tree as compare to linear Data structure.**

**Tree Data structure**

**Tree consists of Node**

**Node has Parent child relationship which has edges to denote relationship**

**In we have a n number of node then we have a n – 1 number of edges .**

**Tree can be empty a tree which does not having any node.**

**Tree Terminology:**

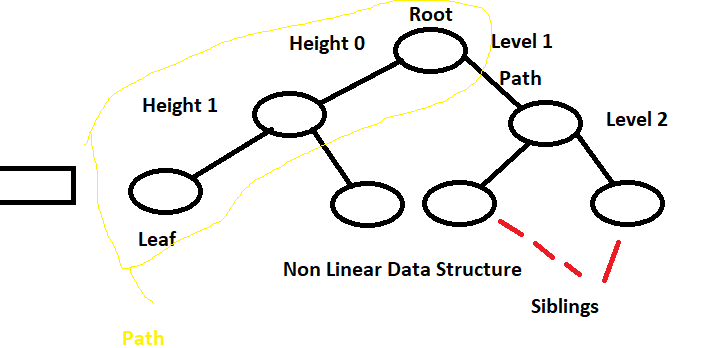
1. **Root - Tree has Root node which is at top not having parent else all other node have parent**
2. **Siblings – Node which are children of same parent known as siblings.**
3. **Leaf/External node: Which has no children. are last nodes.**
4. **Non Leaf/Internal node – It should have one or more children’s.**
5. **Edge – Edge represent relation between parent and child by line.**
6. **Path – sequence of node from parent to leaf node and any two node edge**
7. **Subtree – Any node with its children consider as sub tree**
8. **Forest – is a collections of a trees.**

**Height and Levels of Trees**

**Level of tree start from root and labeled as 1**

**Height also start from root and labeled as 0**

**Children of root node are at level 2 and height is 1**



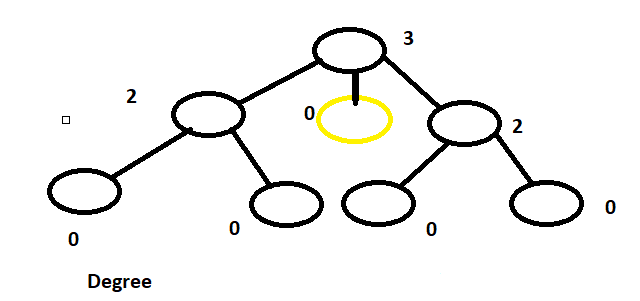
**Number of edges to root node calculated from level and height**

**Any node which is at Height 1 and level 2 has one edge till root node**

**Degree of Node and Tree**

**Degree of node is number of children node has. Hence degree of leaf node is 0 as it has no children’s**

**Degree of a tree is a maximum of degree of a node**



**Binary Trees and Its Properties:**

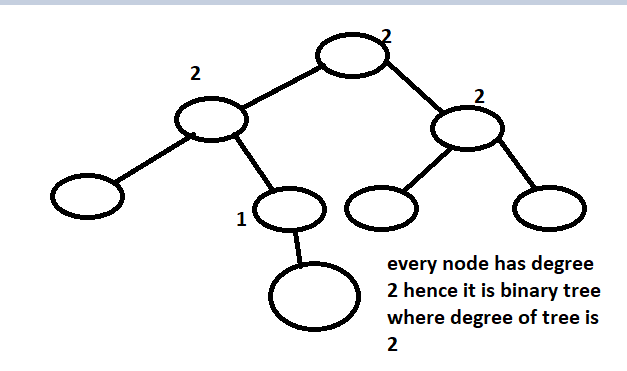
**Every node in binary tree has most two children’s.**

**Every child is labeled as left and right child**

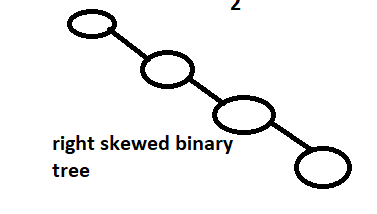
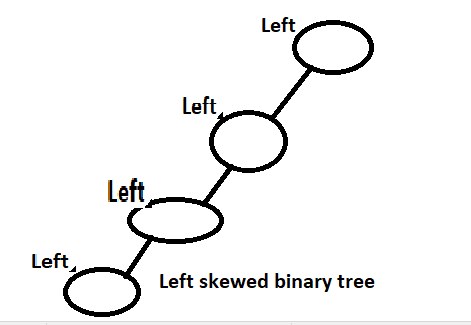
**Left child precedes the right child in order of node**

**Which degree of tree is two it means it is a binary tree – having two children’s**

**In Binary tree node can have less then two but not more than two child’s**



**Node in binary tree can have a degree 0 , 1, 2.**



**How we cab calculate number of nodes from height**

**Height 0 has 1 node**

**Height 1 has 3 nodes**

**Height 2 has 7 nodes**

**And so on…..**

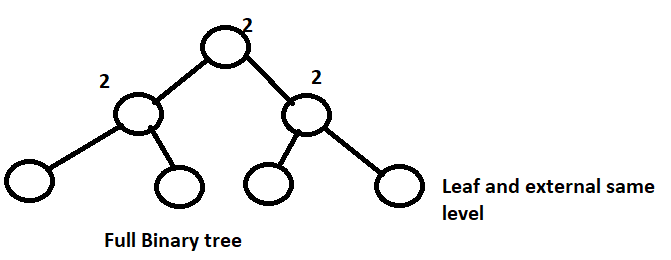
**We can calculate node using 2^(h+1) - 1 = nodes**

**Proper Binary Tree:**

**Binary tree is called proper binary tree if every node has 0 or 2 children. Where degree of node is 0 or 2 but NOT 1**

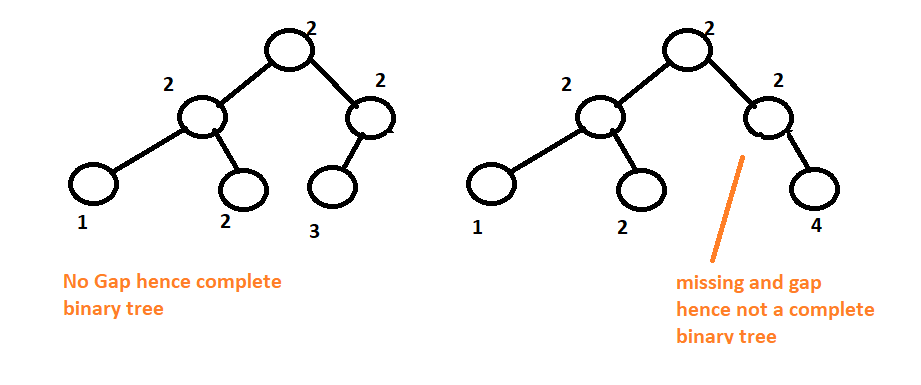
**Full Binary Tree:**

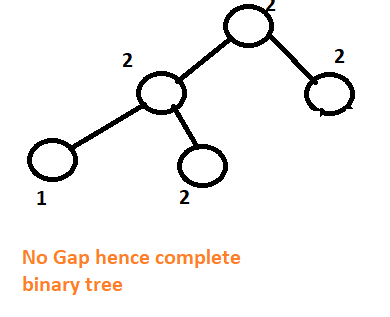
**Binary tree is called full binary tree if all node has exactly 2 children’s and external and leaf nodes are at same level**



**Complete Binary Tree:**

**Number from left to right are numbered without any gap. Gap mean missing node**





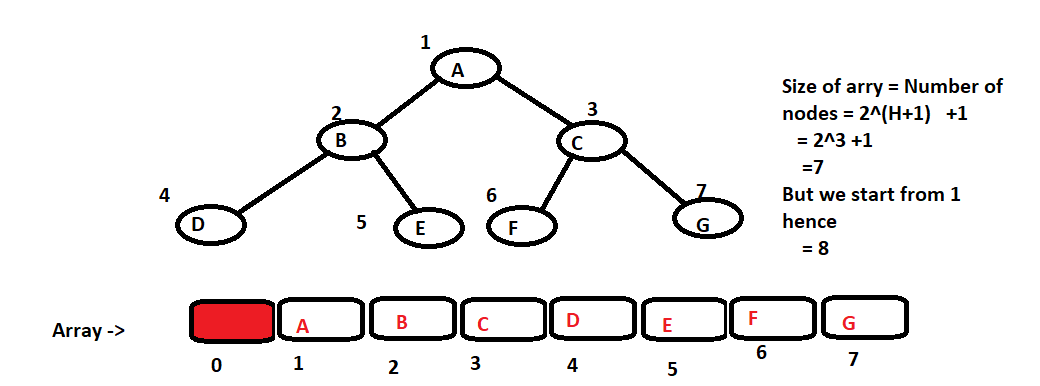
**Complete binary tree of height H is full binary tree up to height H -1.**

Full Vs Complete Vs Proper Binary Tree

**Binary Tree Representation - Array Based**

**Array are fix sized then how we will set size of array using formula to calculate numbers of nodes = 2^H+1 +1.**

**Store node in array always start from 1 index of root to left right and level down**



**We can find the parent of any one example**

|  |  |  |  |
| --- | --- | --- | --- |
| **Element** | **Index** | **Left Child** | **Right Child Index** |
| **A** | **1** | **2** | **3** |
| **B** | **2** | **4** | **5** |
| **C** | **3** | **6** | **7** |
|  |  |  |  |
|  | **i** | **i\*2** | **i\*2 +1** |

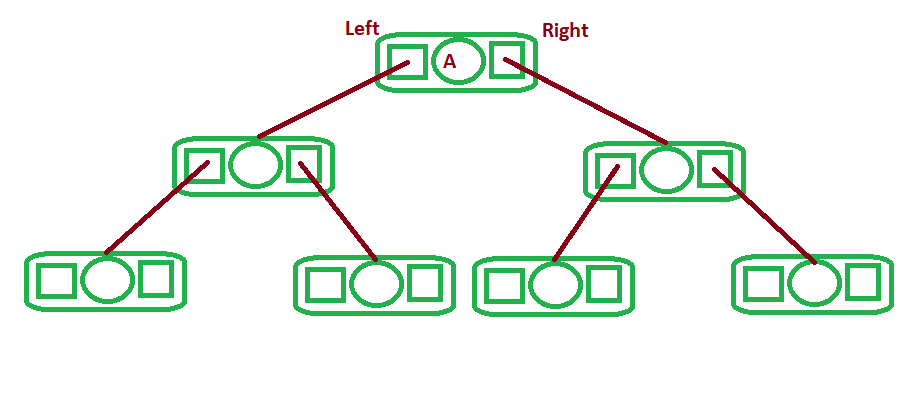
**Lest find parent of D:**

**D is at index 4 = i\*2**

**I = 4/2 = 2 and 2 in index of B hence B is parent of D.**

**Binary Tree Representation - Linked Based**

**We can create binary tree using node node can have a value and left right child**



**Traversing Binary Trees**

**Is a technic of visiting all element in a binary tree. Visit node in some pattern**

**In linear data structure traversing is easy we can start and go sequentially.**

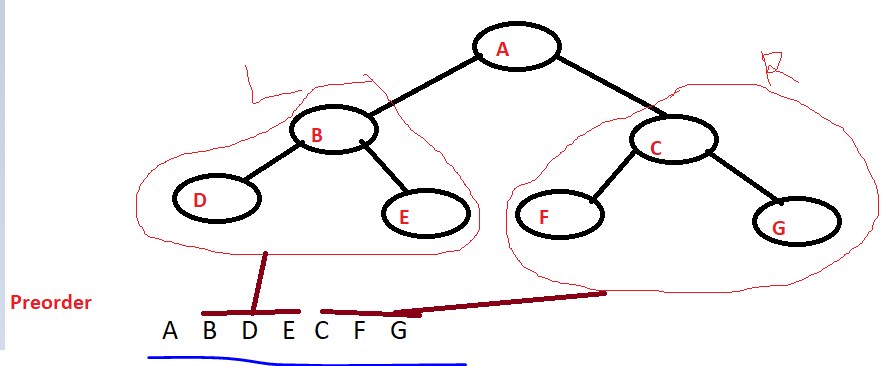
**Binary tree is nonlinear We have 4 different technic to traverse binary tree.**

1. **In order traversal**
2. **Pre order traversal**
3. **Post order traversal**
4. **Level order traversal**

**Binary Trees Traversal –**

**Preorder**

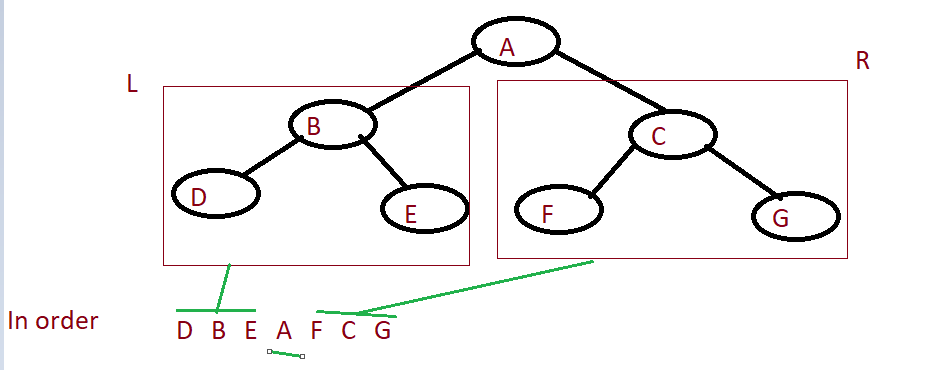
1. **Root visited first**
2. **Then left subtree is visited recursively in preorder**
3. **Then right subtree is visited recursively in preorder**



Binary Trees Traversal –

**Inorder**

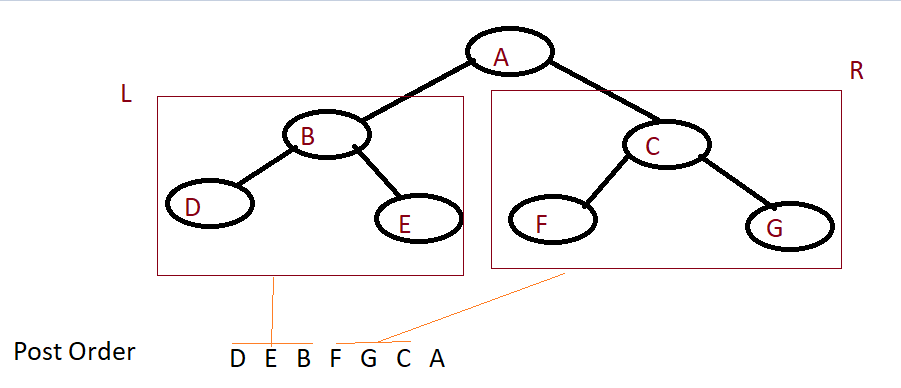
1. **Visit Left sub tree recursively in order**
2. **Visit root**
3. **Then Visit Right sub tree recursively in order.**



Binary Trees Traversal –

**Postorder**

1. **Visit left sub tree recursively in post order**
2. **Visit right sub tree recursively in post order**
3. **Visit root**

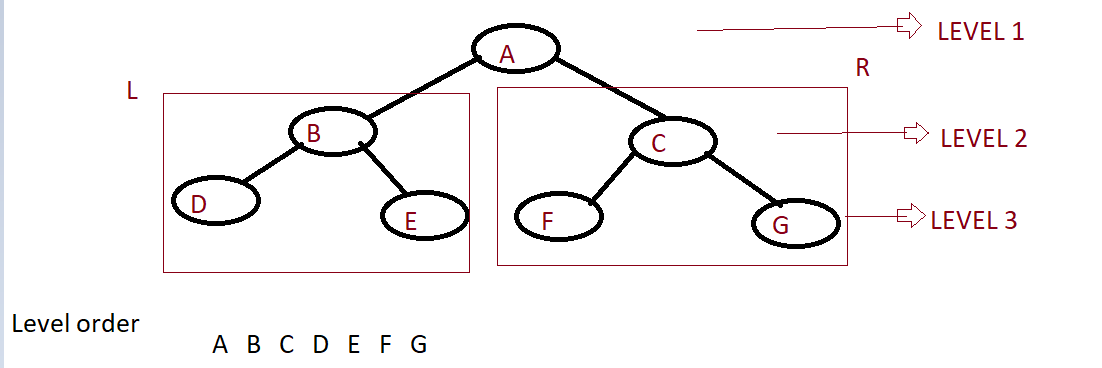


Binary Trees Traversal –

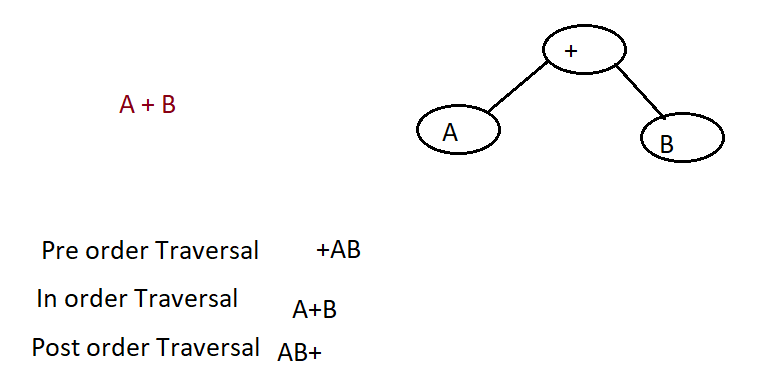
**Level Order**

**Visit nodes level by level from top to bottom**

**Within level visit nodes from left to right**

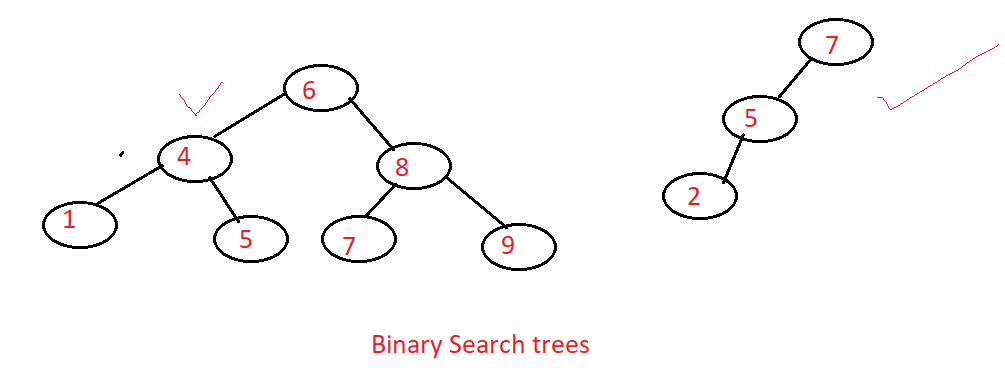


**Easy way of remembering Binary Trees Traversals**



**Binary Search Trees**

**It’s a binary tree in which left nodes are smaller than right nodes**



* **Every node has a key**
* **Keys of left sub tree are smaller than key in node**
* **Keys of right sub tree are larger than key in node**
* **Left and right subtree are also binary search tree.**

**Binary search key will not having duplicates elements**

**Inorder traversal will have element in sorted order**

**Binary search tree can be represented using**

**Arrays based = memory required more**

**Linked based = We can efficiently utilize memory.**

**Binary Search Trees - Searching (Concept)**

**We start from root if match we return true if element is smaller than root we will search in left sub tree if element is larger than root we will search in right sub tree and so on till we reach leaf node.**

**Binary Search Trees - Searching (Concept)**

**Algorithm**

Public bool Search(int key){

Var temp = Root;

While(Temp!=null){

If(Key == temp.Value){

Return true;

}else if(Key < temp.Value){

Temp=temp.Left;

} else if(Key > temp.Value){

Temp=temp.Right;

}

}

Return false;

}

**Time complexity O(Height)**

**Binary Search Trees - Recursive Search Function**

**Public bool rSearch(Node root,int key){**

**If(root !=null){**

**If(key == root.Value){**

**Return true;**

**}else if(key < root.Value){**

**rSearch(root.left,key);**

**} else if(key > root.Value){**

**rSearch(root.Right,key);**

**}**

**}else{**

**Return false;**

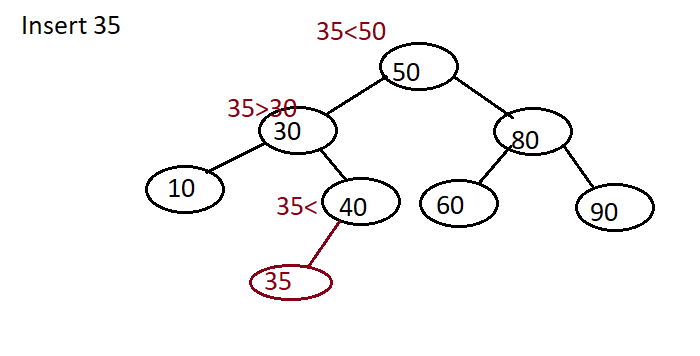
**}**

**}**

**Time complexity O(Height + 1)**

**Binary Search Trees - Insertion (Concept)**

**Keep comparing element with starting from root . remember Binary search tree not having duplicates element Insert 35 shown in Fig.**



**Binary Search Trees - Iterative Insert Function**

**For insertion first traverse binary tree to find node at which we need to insert then perform insert operation**

**Algo**

**Public void insert(Node Root,int key){**

**//Lets find out node at which we can insert**

**Var temp=null;**

**Var tRoot=Root;**

**While(tRoot != null){**

**Temp=tRoot;**

**If(key == tRoot.Value){  
 return; //Invalid**

**}else if(key < tRoot.Value){**

**tRoot= tRoot.Left;**

**} else if(key > tRoot.Value){**

**tRoot= tRoot.Right;**

**}**

**}**

**//Now we have a temp node at which we can insert**

**Node insertNode = new Node(Key);**

**If(!isEmpty){**

**If(key < temp.Value){**

**Temp.Left = insertNode;**

**}else if(key > temp.Value){**

**Temp.Right = insertNode;**

**}**

**}**

**Else{**

**Root = insertNode;**

**}**

**}**

**Iterative code sample**

**Binary Search Trees - Recursive Insert Function**

**O(h+1)**

**Public Node rInsert(Node tRoot,int key){**

**If(tRoot != null){**

**If(key == tRoot.Value){**

**Return null;//Invalid**

**}else if(key < tRoot.value){**

**tRoot.Left = rInsert(tRoot.left,key);**

**} else if(key > tRoot.value){**

**tRoot.Right = rInsert(tRoot.Right,key);**

**}**

**}**

**Else{**

**Node newNode = new Node(key);**

**tRoot = newNode;**

**}**

**Return troot;**

**}**

**Traversing Binary Search Tree**

**Same as binary tree**

* **InOrder: A+B**
* **PreOrder : +AB**
* **PostOrder AB+**
* **Level order**

**Function for Inorder Traversal**

**Mean Left Root Left – and repeat**

**Public void inOrder(Node root){**

**If(root != null){**

**Inorder(root.Left);// *Go in depth of left sub tree top to bottom left sub tree***

**Console.WriteLine(root.Value);**

**Inorder(root.Right);// Go Up words of right sub tree. Bottom to top right sub trees**

**}**

**}**

**Insertion in Binary Search Tree - Implementation**

**Iterative and recursive insert**

namespace CoddingBST

{

public class Node

{

public int element;

public Node left;

public Node right;

public Node(int e, Node l, Node r)

{

element = e;

left = l;

right = r;

}

}

class BinarySearchTree

{

Node root;

public BinarySearchTree()

{

root = null;

}

public void insertIterative(Node temproot, int e)

{

Node temp = null;

while (temproot != null)

{

temp = temproot;

if (e == temproot.element)

return;

else if (e < temproot.element)

temproot = temproot.left;

else if (e > temproot.element)

temproot = temproot.right;

}

Node n = new Node(e, null, null);

if (root != null)

{

if (e < temp.element)

temp.left = n;

else

temp.right = n;

}

else

root = n;

}

public Node insertRecirsive(Node temproot, int e)

{

if (temproot != null)

{

if (e < temproot.element)

temproot.left = insertRecirsive(temproot.left, e);

else if (e > temproot.element)

temproot.right = insertRecirsive(temproot.right, e);

}

else

{

Node n = new Node(e, null, null);

temproot = n;

}

return temproot;

}

public void inorder(Node temproot)

{

if (temproot != null)

{

inorder(temproot.left);

Console.Write(temproot.element + " ");

inorder(temproot.right);

}

}

}

}

**Function for Preorder Traversal**

**First we print root node then we go left to depth then start right depth to top**

**Fucntion PreOrder(troor){**

**If(tRoot !=null){**

**Console.WriteLine(troot.Value);**

**PreOrder(troor.left);**

**PreOrder(troor.Right);**

**}**

**}**

public void preorder(Node temproot)

{

if (temproot != null)

{

Console.Write(temproot.element + " ");

preorder(temproot.left);

preorder(temproot.right);

}

}

**Function for Postorder Traversal**

**First go Left sub tree in depth**

**Then come right sub tree in up**

**Then visit root at end**

**Fucntion PostOrder(troor){**

**If(tRoot !=null){**

**PreOrder(troor.left);**

**PreOrder(troor.Right);**

**Console.WriteLine(troot.Value);**

**}**

**}**

public void PostOrder(Node temproot)

{

if (temproot != null)

{

preorder(temproot.left);

preorder(temproot.right);

Console.Write(temproot.element + " ");

}

}

**Function for Level Order Traversal**

**We can implement this by using queue**

**We can add root in queue and chechk its right and left and add right and left in queue and gose on print till queue is empty**

**Right and left condition swap enqueue opearation on both side of loop and print level wise**

**Algo:**

**Function LevelOrderTraversal(Node tRoot){**

**Queue queue = new Queue()**

**Queue.Enqueue(tRoot);**

**Print(tRoot.Element);**

**While(! Queue.IsEmpty){**

**Node n = queue.Dequeue();**

**If(n.left != Null){**

**Print(n. Left .Element);**

**Queue.Enqueue(n.Left);**

**}**

**If(n.right !=null){**

**Print(n. Right .Element);**

**Queue.Enqueue(n.Right);**

**}**

**}**

**}**

**Code sample**

using System;

using System.Collections.Generic;

using System.Text;

namespace CoddingBST

{

public class Node

{

public int element;

public Node left;

public Node right;

public Node(int e, Node l, Node r)

{

element = e;

left = l;

right = r;

}

}

class BinarySearchTree

{

Node root;

public BinarySearchTree()

{

root = null;

}

public void insertIterative(Node temproot, int e)

{

Node temp = null;

while (temproot != null)

{

temp = temproot;

if (e == temproot.element)

return;

else if (e < temproot.element)

temproot = temproot.left;

else if (e > temproot.element)

temproot = temproot.right;

}

Node n = new Node(e, null, null);

if (root != null)

{

if (e < temp.element)

temp.left = n;

else

temp.right = n;

}

else

root = n;

}

public Node insertRecirsive(Node temproot, int e)

{

if (temproot != null)

{

if (e < temproot.element)

temproot.left = insertRecirsive(temproot.left, e);

else if (e > temproot.element)

temproot.right = insertRecirsive(temproot.right, e);

}

else

{

Node n = new Node(e, null, null);

temproot = n;

}

return temproot;

}

public void inorder(Node temproot)

{

if (temproot != null)

{

inorder(temproot.left);

Console.Write(temproot.element + " ");

inorder(temproot.right);

}

}

public void PostOrder(Node temproot)

{

if (temproot != null)

{

PostOrder(temproot.left);

PostOrder(temproot.right);

Console.Write(temproot.element + " ");

}

}

public void preorder(Node temproot)

{

if (temproot != null)

{

Console.Write(temproot.element + " ");

preorder(temproot.left);

preorder(temproot.right);

}

}

public void levelorder(Node temproot)

{

QueuesLinked q = new QueuesLinked();

Node t = root;

Console.Write(t.element + " ");

q.enqueue(t);

while (!q.isEmpty())

{

t = (Node)q.dequeue();

if (t.left != null)

{

Console.Write(t.left.element + " ");

q.enqueue(t.left);

}

if (t.right != null)

{

Console.Write(t.right.element + " ");

q.enqueue(t.right);

}

}

}

}

#region

/// <summary>

/// Below code is just to implement queue using linked List

/// </summary>

public class QNode

{

public Object element;

public QNode next;

public QNode(Object e)

{

element = e;

next = null;

}

}

class QueuesLinked

{

QNode front;

QNode rear;

int size;

public QueuesLinked()

{

front = null;

rear = null;

size = 0;

}

public int length()

{

return size;

}

public bool isEmpty()

{

return size == 0;

}

public void enqueue(Object e)

{

QNode newest = new QNode(e);

if (isEmpty())

front = newest;

else

rear.next = newest;

rear = newest;

size = size + 1;

}

public Object dequeue()

{

if (isEmpty())

{

Console.WriteLine("Queue is Empty");

return -1;

}

Object e = front.element;

front = front.next;

size = size - 1;

if (isEmpty())

rear = null;

return e;

}

public void display()

{

QNode p = front;

while (p != null)

{

Console.Write(p.element + " --> ");

p = p.next;

}

Console.WriteLine();

}

}

#endregion

**Iterative Searching in Binary Search Tree -**

public bool search(int key)

{

Node temproot = root;

while (temproot != null)

{

if (key == temproot.element)

return true;

else if (key < temproot.element)

temproot = temproot.left;

else if (key > temproot.element)

temproot = temproot.right;

}

return false;

}

**Recursive Searching in Binary Search Tree -**

public bool search(Node temproot, int key)

{

if (temproot != null)

{

if (key == temproot.element)

return true;

else if (key < temproot.element)

return search(temproot.left, key);

else if (key > temproot.element)

return search(temproot.right, key);

}

return false;

}

**Binary Search Tree – Deletion**

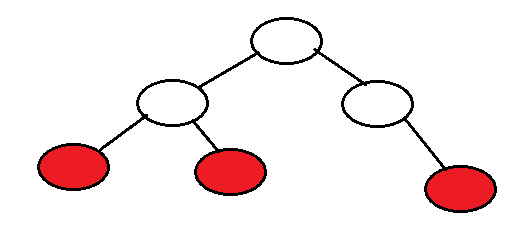
**We need to perform search operation before deleting node in binary tree**

**Possibilities of deleting**

* **Node can be Leaf node or internal node**
* **Deleting node can have a one sub tree Left or right**
* **Deleting node can have a two sun trees Left and right.**

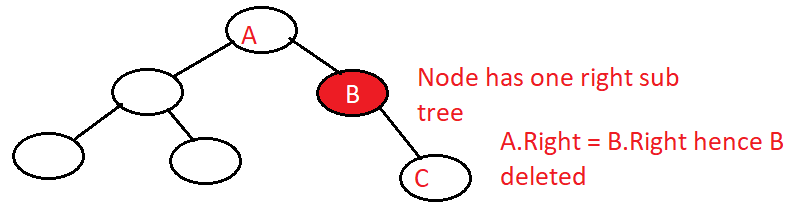
**Binary Search Tree - Deletion Case-Leaf Node**

**Node which we are deleting is a leaf node.**



**Binary Search Tree - Deletion Case-Node with One Subtree**

**Deleting node can have a one sub tree left or right**



**Binary Search Tree - Deletion Case-Node with Both Subtrees**

**Deleting node which has both left and right sub trees.**

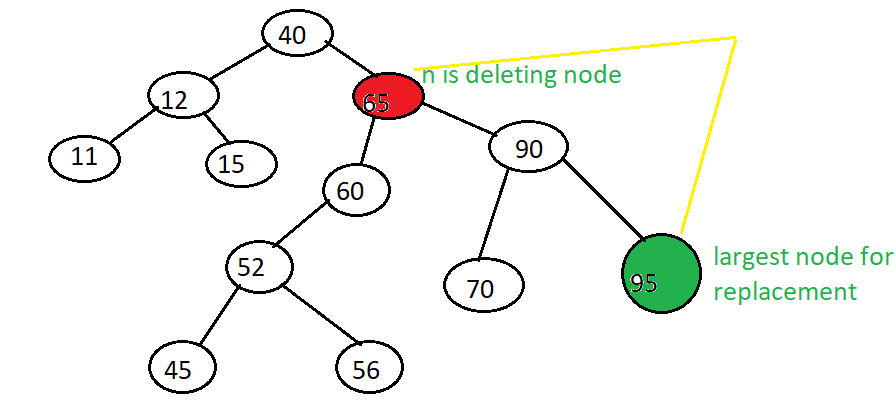
**In this case we need to find largest element in left subtree or smallest element in right sub tree and replace that with deleting node. Here we are not actually replacing node reference but we just swap values and delete other switching node.**

**T= n.Right**

**While t.Left != null**

**T = t.Left**

**n.element = t.element**



public bool delete(int e)

{

Node p = root;

Node pp = null;

while (p != null && p.element != e)

{

pp = p;

if (e < p.element)

p = p.left;

else

p = p.right;

}

if (p == null)

return false;

if (p.left != null && p.right != null)

{

Node s = p.left;

Node ps = p;

while (s.right != null)

{

ps = s;

s = s.right;

}

p.element = s.element;

p = s;

pp = ps;

}

Node c = null;

if (p.left != null)

c = p.left;

else

c = p.right;

if (p == root)

root = null;

else

{

if (p == pp.left)

pp.left = c;

else

pp.right = c;

}

return true;

}

**Count Number of Nodes in Binary Tree**

**Function Count(Node root){**

**If(root != null){**

**X = Count(root.Left);**

**y = Count(root.Right);**

**return x + y +1;**

**}**

**Return 0;**

**}**

public int count(Node temproot)

{

if (temproot != null)

{

int x = count(temproot.left);

int y = count(temproot.right);

return x + y + 1;

}

return 0;

}

**Find Height of Binary Tree**

**Height of tree is highest hight amoung left and right subtree hence find height of left and right sub tree which is greater is the height of tree**

**Function height(Node root){**

**If(root != null){**

**X = height(root.left);  
 Y = height(root.right);**

**If(x>y)**

**Return x +1;**

**Else**

**Return y +1;**

**}**

**Return 0;**

**}**

public int height(Node temproot)

{

if (temproot != null)

{

int x = height(temproot.left);

int y = height(temproot.right);

if (x > y)

return x + 1;

else

return y + 1;

}

return 0;

}

**Performance and Problem of Binary Search Trees**

**Binary search tree of height H and N nodes**

**Space complexity : O(n)**

**Time Complexity:**

**Searching: O(h)**

**Inserting: O(h)**

**Deleting: O(h)**

**Balanced Tree**

* **Reduced the height of Binary search tree**
* **Rotations and restructuring**
* **Modify the relationship between parent – child**
* **Example: AVL tree, Red-Black tree, Splay Tree.**

**AVL Tree**

**It is a balance search tree**

**Better performance**

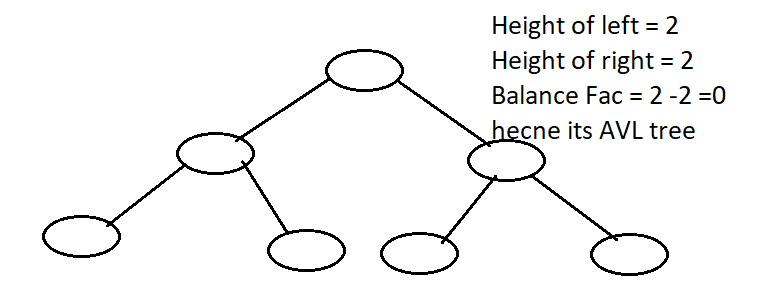
**Binary search tree- Reshape and reduced height**

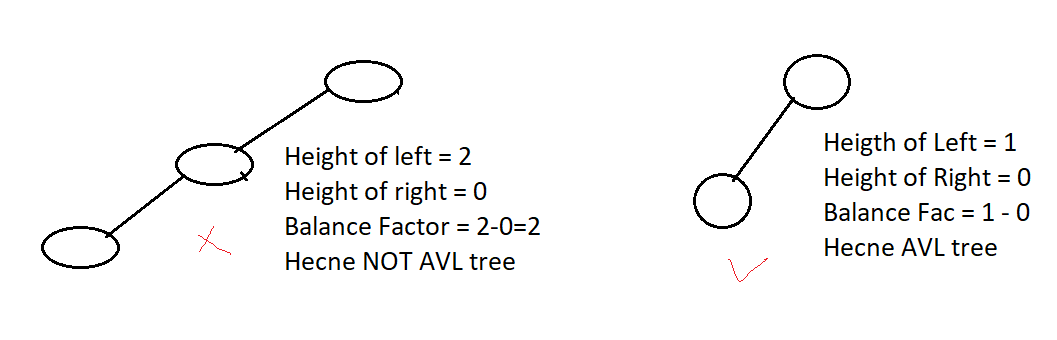
**Height- Balance property pr balance Factor**

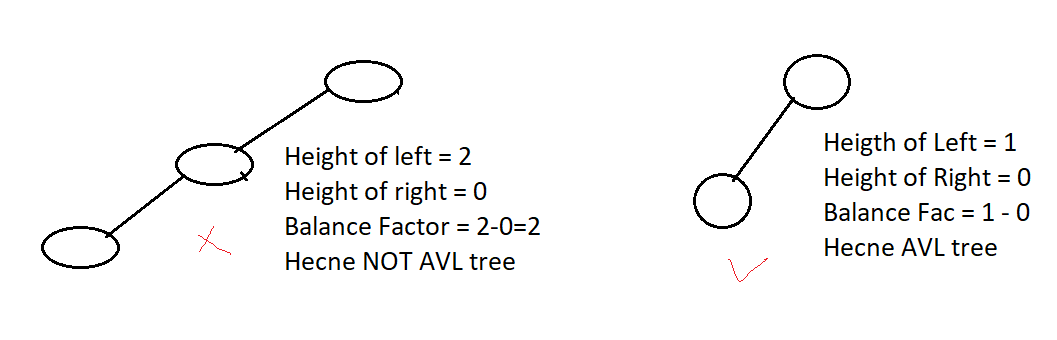
**Height – Balance property – for every node height of** children **differ by are most** 1

**AVl Tree- Any binary search tree that satisfies height – balance property**

**Balance Factor – Height of left sub tree – height of right sub tree == 1, 0, -1**







**Insertion and deleting is same as binary search tree but reposition is done after every insertion and deletion to maintain its AVL tree**

**AVL Tree Rotations for Insertion**

**Once we insert delete node we need to rotate tree to maintain its an AVL tree.**

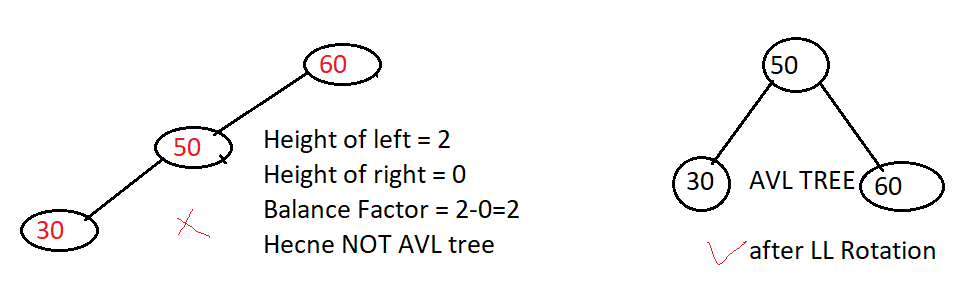
* **LL Rotation**
* **RR Rotation**
* **LR Rotation**
* **RL Rotation**

**LL Rotation:**

**Suppose we insert two node at left side hence balance will disturb the we do LL rotation**

**It dosent matter how big is binary search tree we always perform 3 rotation**

**Lets insert 50 and 30 new nodes in below tree**

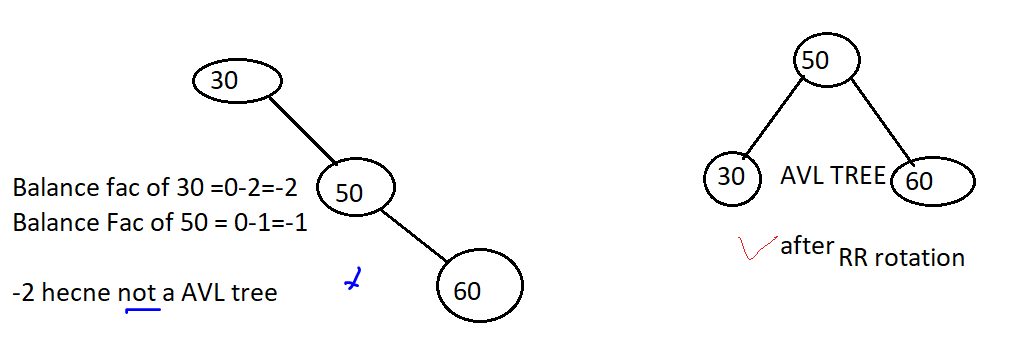


**RR Rotation:**

**Suppose we insert two nodes at right side hence balance will disturb the we do RR rotation to maintain AVL tree**

**It doesn’t matter how big is binary search tree we always perform 3 rotation**

**Lets insert 50 and 60 new node in below tree.**

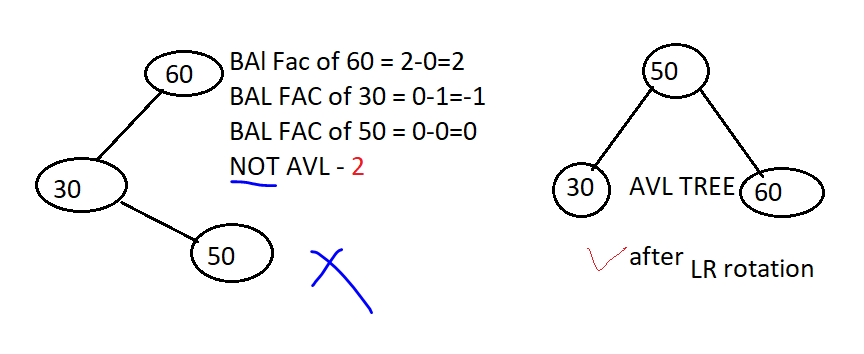


**LR Rotation:**

**I this node inserted in such a way that first insert to left then insert to right of new inserted node then we need to perform LR rotation to maintain LR Rotation**

**Lets insert 30 and 50 new node in below tree**

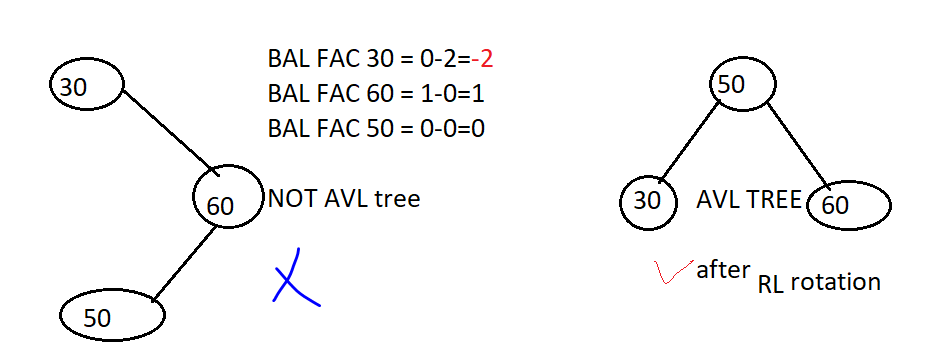
**Here rotation take two steps.**



**RL Rotation:**

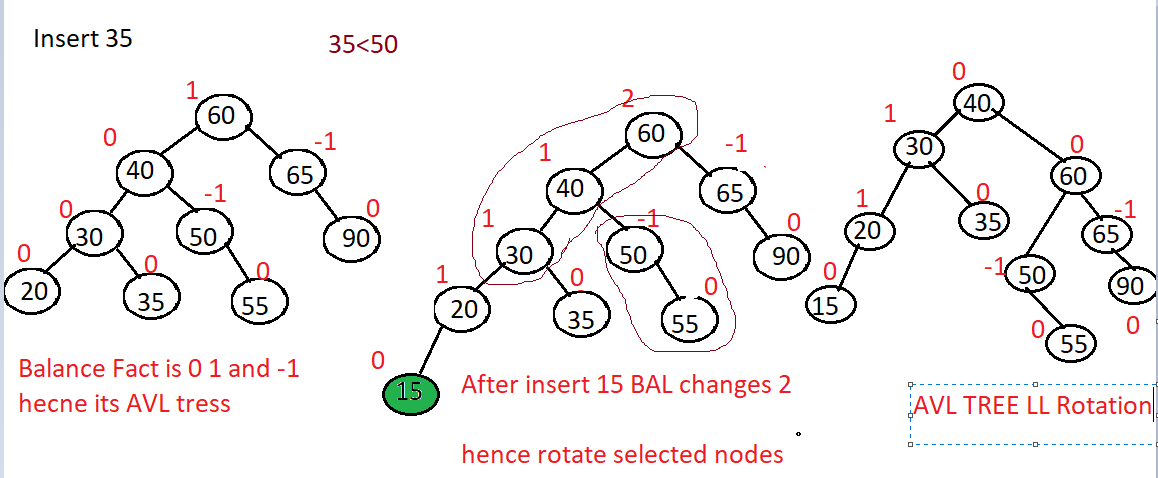
**I this node inserted in such a way that first insert to right then insert to left of new inserted node then we need to perform RL rotation to maintain LR Rotation**

**Lets insert 60 and 50 new node in below tree**

We need two rotation

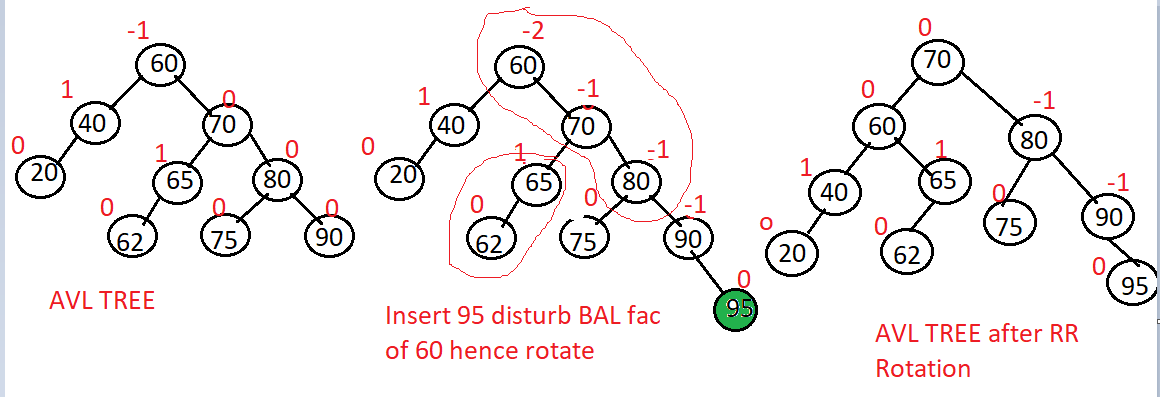
**AVL Tree - LL Rotation**

**Lets consider below example where balance factor is calculated whos balance factor will be greater than 1 need to rotation and we always rotate only 3 nodes**



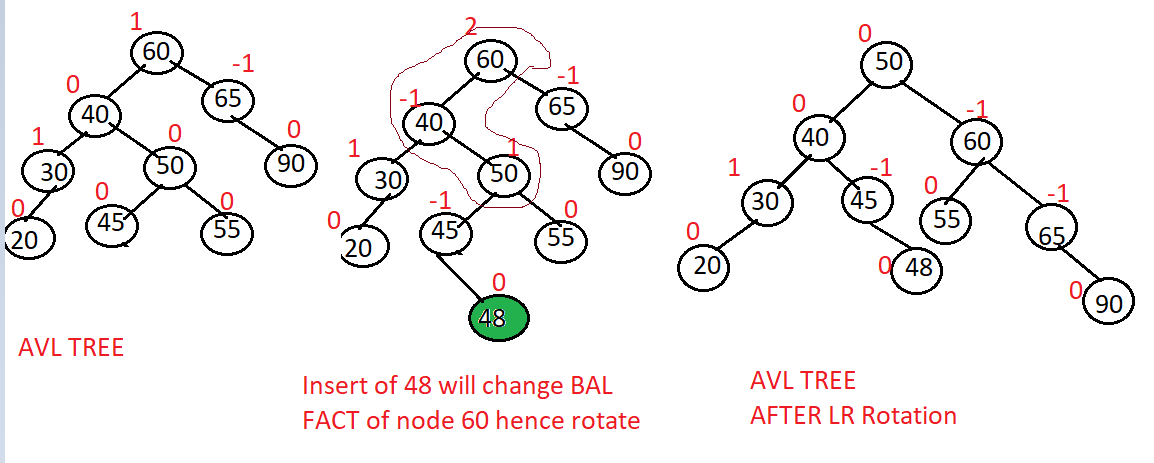
**AVL Tree - RR Rotation**

**Lets consider below example where balance factor is calculated whos balance factor will be greater than 1 need to rotation and we always rotate only 3 nodes**

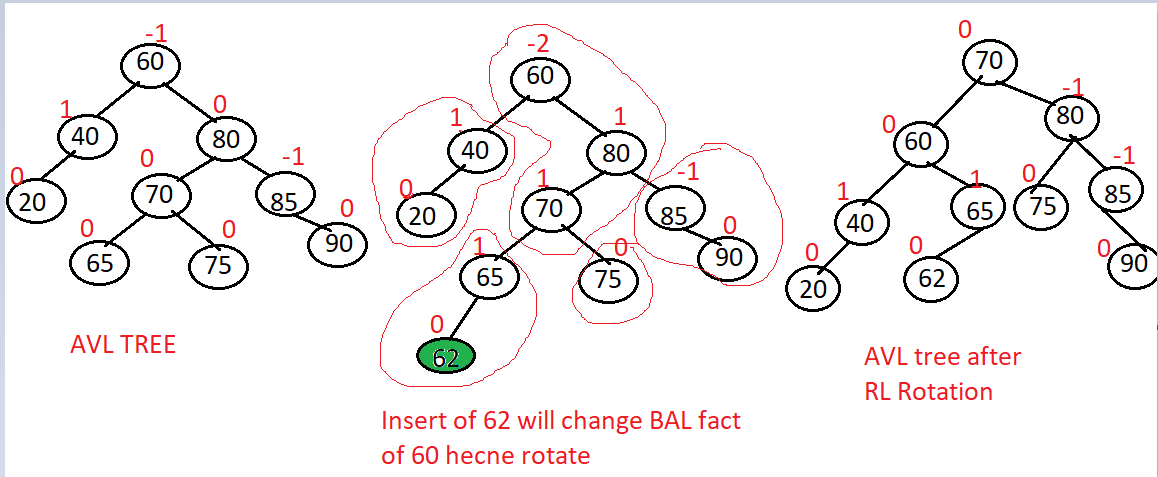


**AVL Tree - LR Rotation**

**Lest insert 40 and rotate**



**AVL Tree - RL Rotation**



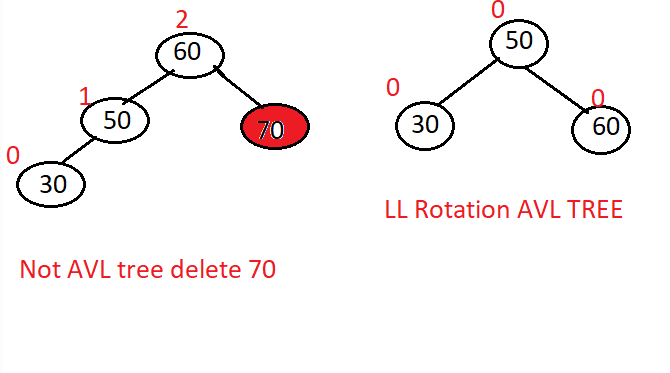
**AVL Tree Rotations after Deletion**

**For deletion as well we have a 4 rotation**

1. **LL Rotation**
2. **RR Rotation**
3. **LR Rotation**
4. **RL Rotation**

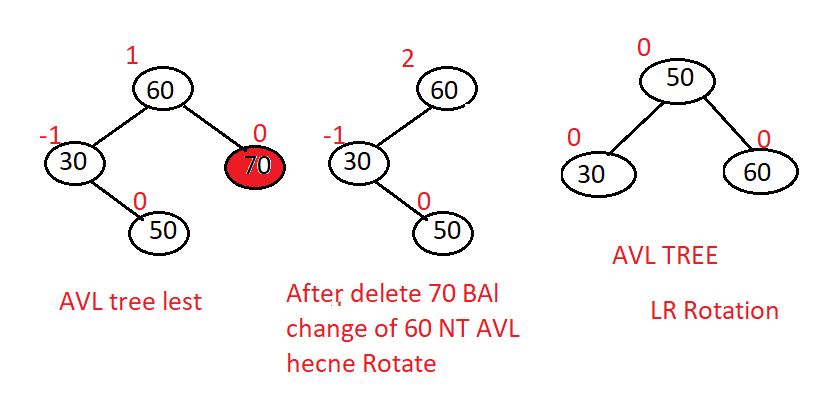
**LL Rotation**

**Lest delete 70 in below example**



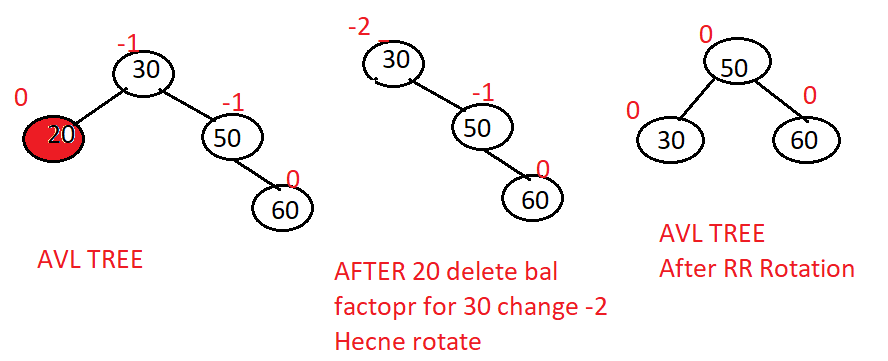
**LR Rotation**

**Delete 70 in below example**



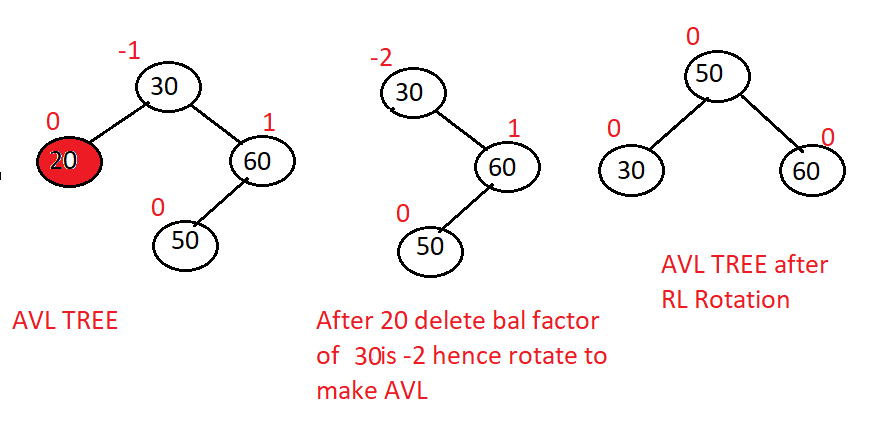
**RR Rotation**

**Delete 20**



**RL Rotation**

**Delete 20 in below example**



**Performance Analysis of AVL Trees**

**Time complexity of AVL tree rotations**

* **LL Rotation: O(1)**
* **RR Rotation: O(1)**
* **LR Rotation: O(1)**
* **RL Rotation: O(1)**

**Run time performance on AVL insertion deletion**

**Height: O(log n)**

**Searching: O(log n)**

**Insertion: O(log n)**

**Deletion: O(log n)**

**Red-Black Trees**

**Is a balanced binary search tree**

**AVL tree required additional restructuring operations-Insert delete**

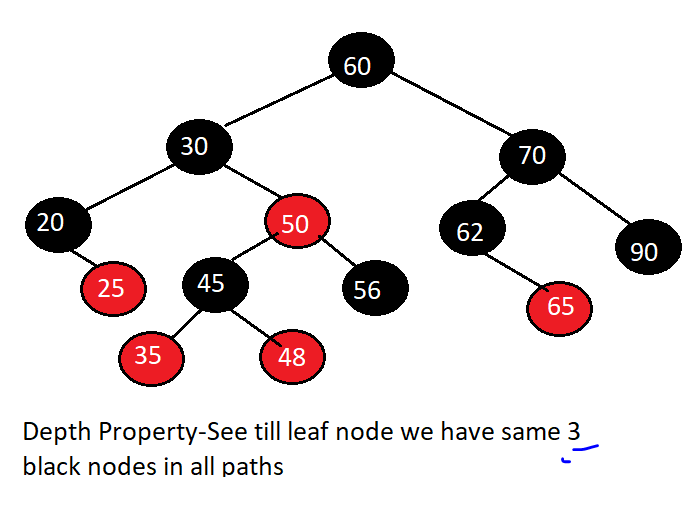
**RED black do not have this hence better perform then AVL**

**In Red-Black tree node colored red and black**

**Red black tree property:**

* **Root is always black**
* **Children of red node are black**
* **All node has same number of black nodes**

**Exmaple**



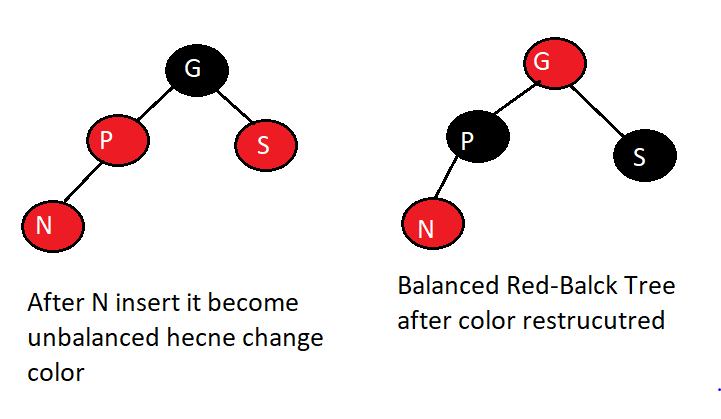
**Red-Black Trees - Restructuring**

**When we insert element in red black tree insertion is perform in same way as binary tree, But insertion deletion may make red black tree un balanced hence to make it balanced we restructured it**

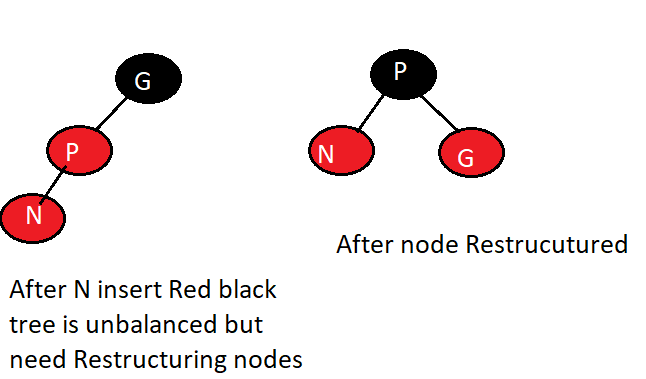
**Restructuring happened after insertion or deletion**

**It Involves coloring and may be rotations to balance the red black tree**

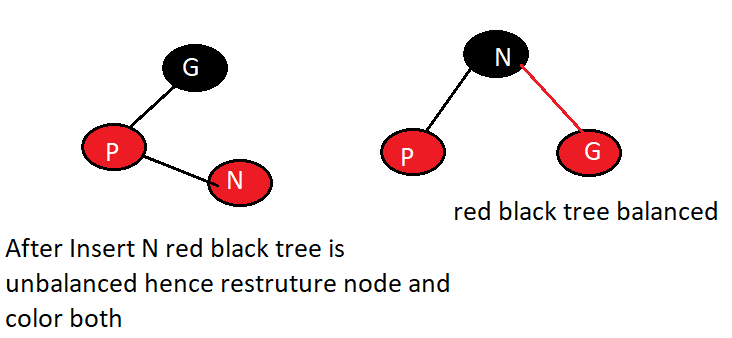
**Scenario:1**



**Scenario2:**



**Scenario 3:**



**Red-Black Trees - Insertion**

**Elements: 20 28 35 52 45 68**

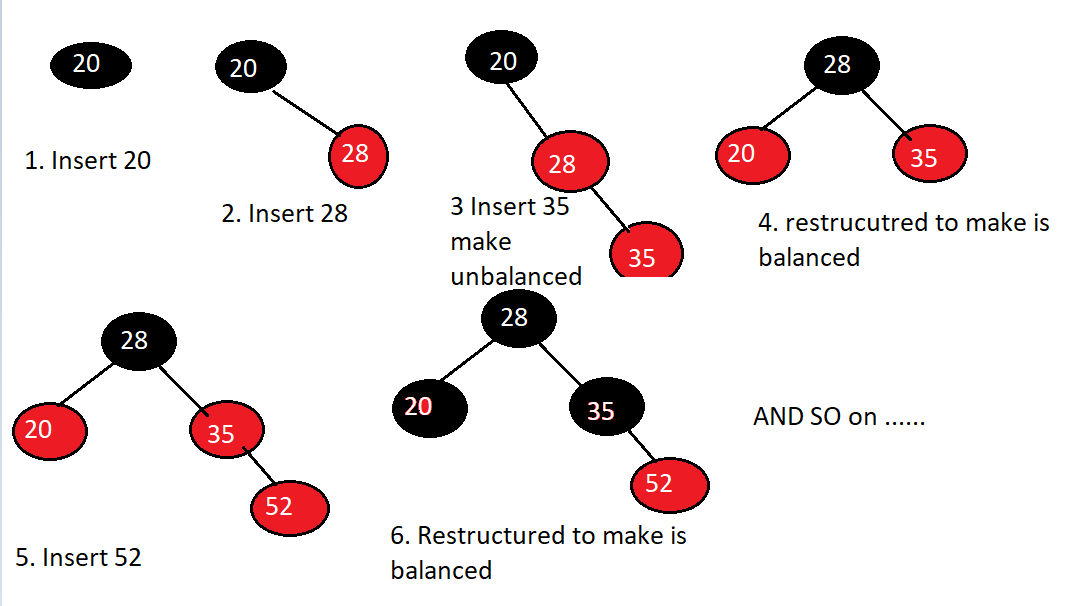
**While inserting we need to tale care of all 3 properties of red black tree and arrange color and node according**

**Lets insert 20 as root node as we know root is black make it black**

**Lets insert 28 \_ to right or root and male It red**

**Lets insert 35 right of 28 but need to restructured color and node**

**And so on till node inserted**



**Red-Black Trees - Deletion**

**Deleting node is same a binary search tree**

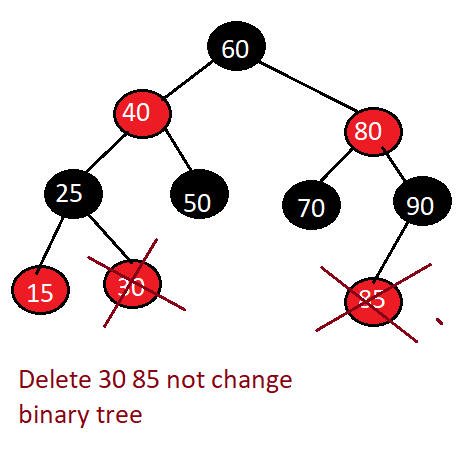
**Deleting scenarios**

* **Scenario-1: Removed node is leaf node and is red**
* **Scenario-2: Removed node is leaf node and black**
* **Scenario-3: Removed node is RED non – leaf node with one subtree**
* **Scenario-4: Removed node is Black non-leaf node with one subtree**
* **Scenario-5: Removed node is RED non-leaf node with two sub trees**
* **Scenario-6: Removed node is Black non-leaf node with two subtrees.**

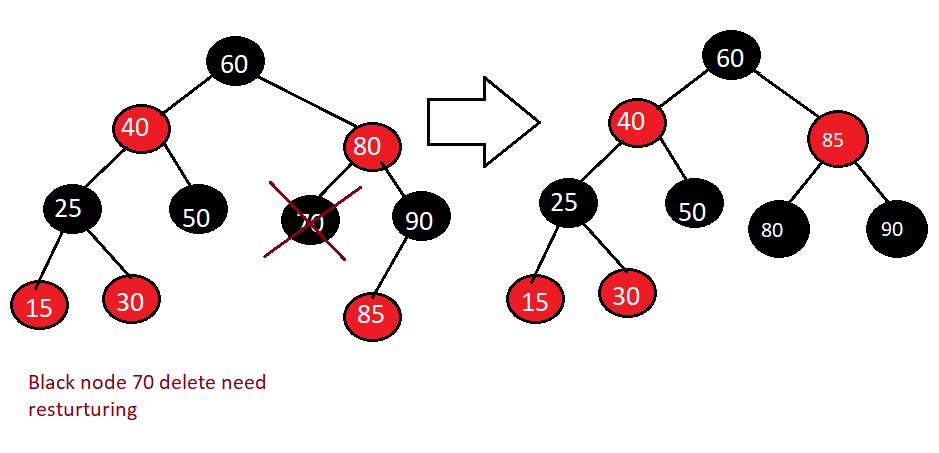
**Scenario-1: Removed node is leaf node and is red**

**Deleting node in leaf dose not change binary tree**

**Hence no restructuring**



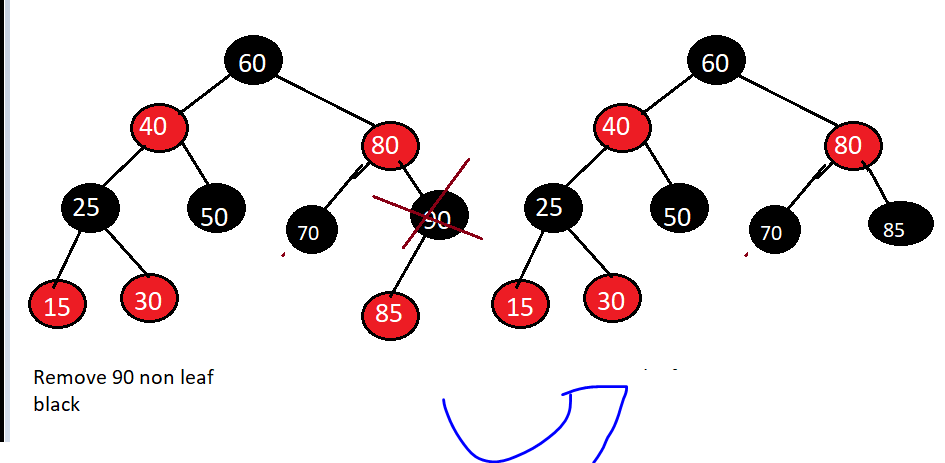
**Scenario-2: Removed node is leaf node and black**



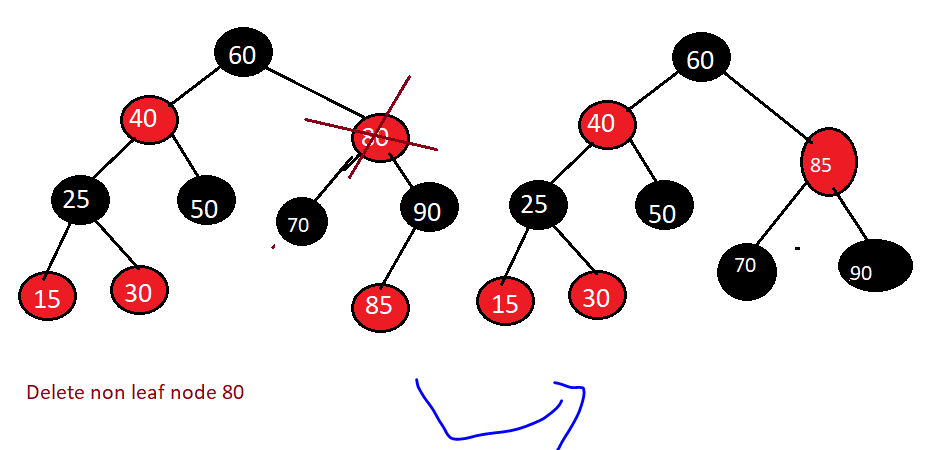
**Scenario-3: Removed node is RED non – leaf node with one subtree,**

**Practically not possible**

**Scenario-4: Removed node is Black non-leaf node with one subtree**

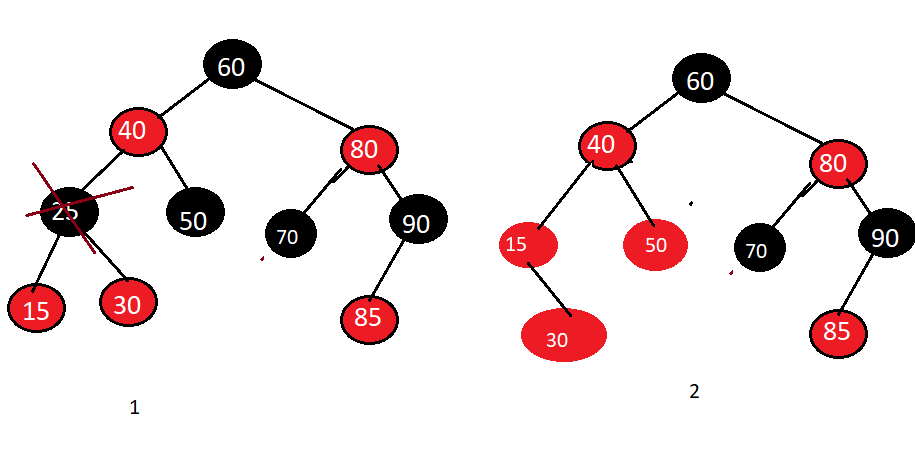


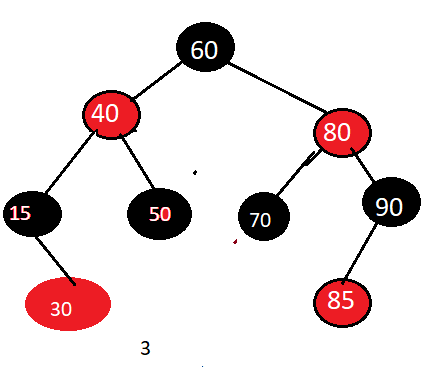
**Scenario-5: Removed node is RED non-leaf node with two sub trees**



**Scenario-6: Removed node is Black non-leaf node with two subtrees**

**Delete 25 n**





**Performance Analysis of Red-Black Trees**

* **Searching: O(log n)**
* **Inserting: : O(log n)**
* **Deleting: : O(log n)**

**Splay Trees**

* **Splay trees are balanced search tress**
* **Dose not enforce bound on the height- not use height or balanced factor associated with tree**
* **Performance achieved through splaying – Splaying operation is performed when node in inserted or searched**
* **Frequently accessed element is near the root**

**Splaying operation is performed by moving the node to the root of tree through sequence of restructuring operations**

**Specific restructuring operation we perform depends on the node also depends on parent or grant parent if any**

**Rule of performing Splaying:**

**Splaying is perform when node is**

**Searching: Splaying perform the position where element is found if not found then playing perform from leaf node where search end .**

**Inserting: Newly inserted node is splay from the position where node is inserted**

**Deleting: Splaying is perform from parent of node which is deleted**

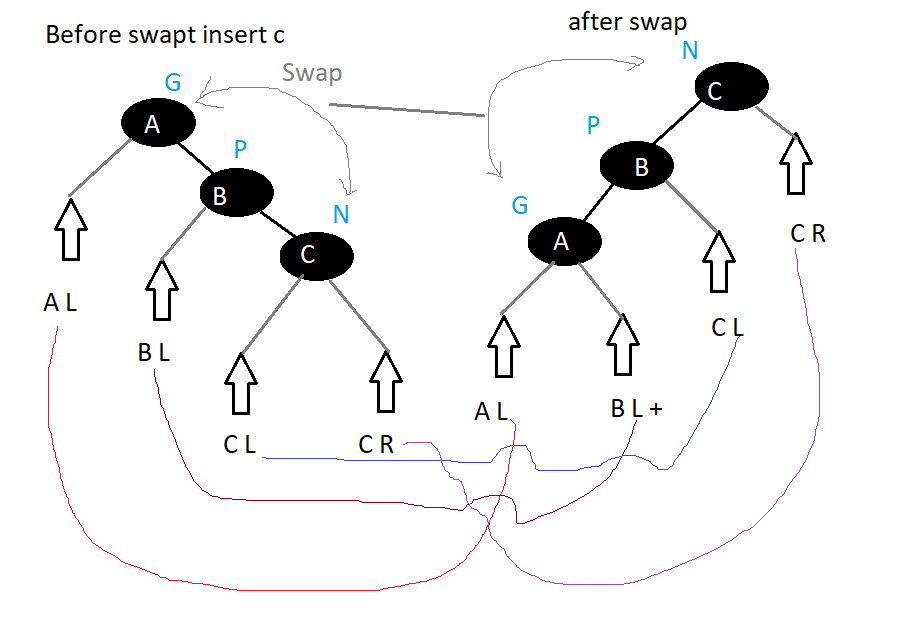
**3 cases of restructuring**

* **Zig-Zig: Node and parent both left and right children**
* **Zig-Zag: Node as left and parent as right child and vice cerse**
* **Zig: Node dose not have grandparent**

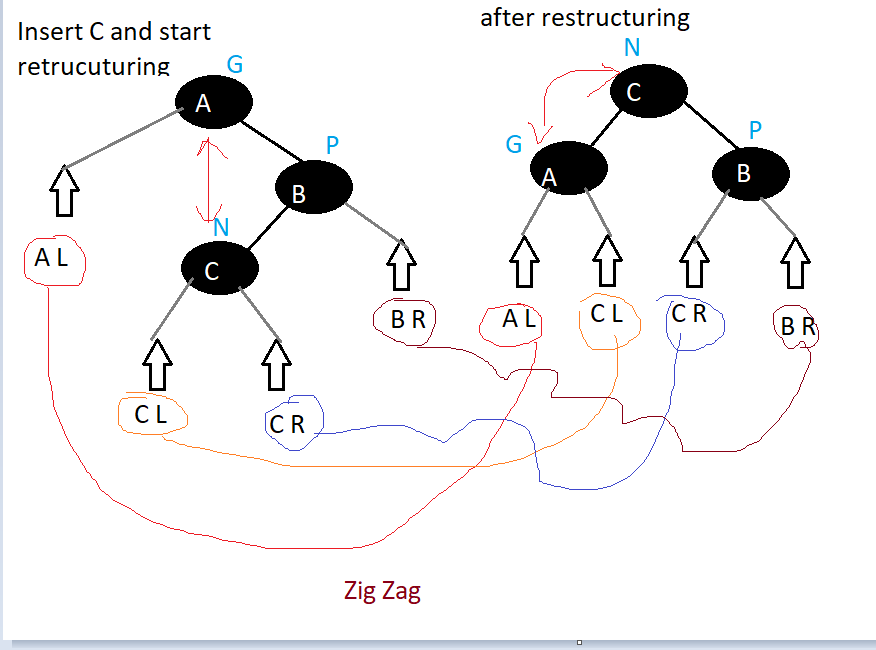
**Splay Trees - Zig-Zig Restructuring**

**New inserted node will be make as root node then we restructured tree to make is as binary tree.**

**Left and right subtrees swap happened.**

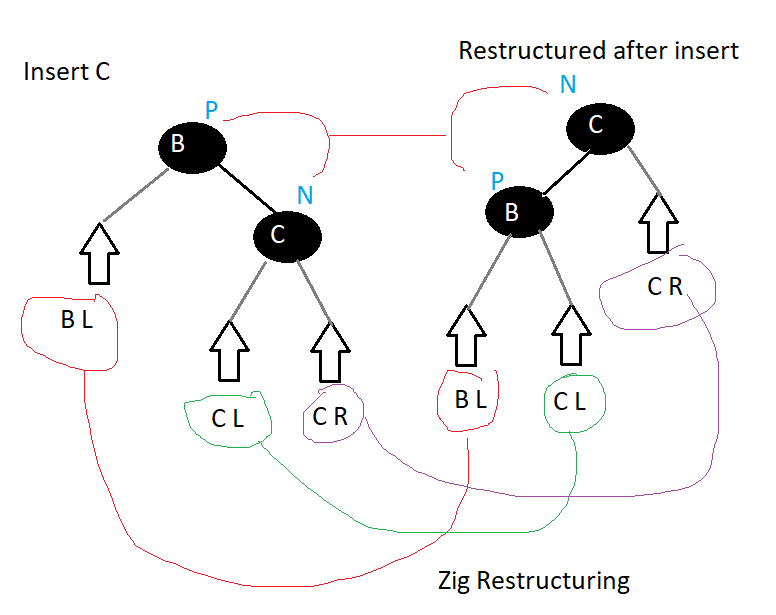


**Splay Trees - Zig-Zag Restructuring**



**Splay Trees - Zig Restructuring**

**Here node dose not have grant parent new node inserted right of its parent. After insert validate binary search tree property not matching than restructured tree**



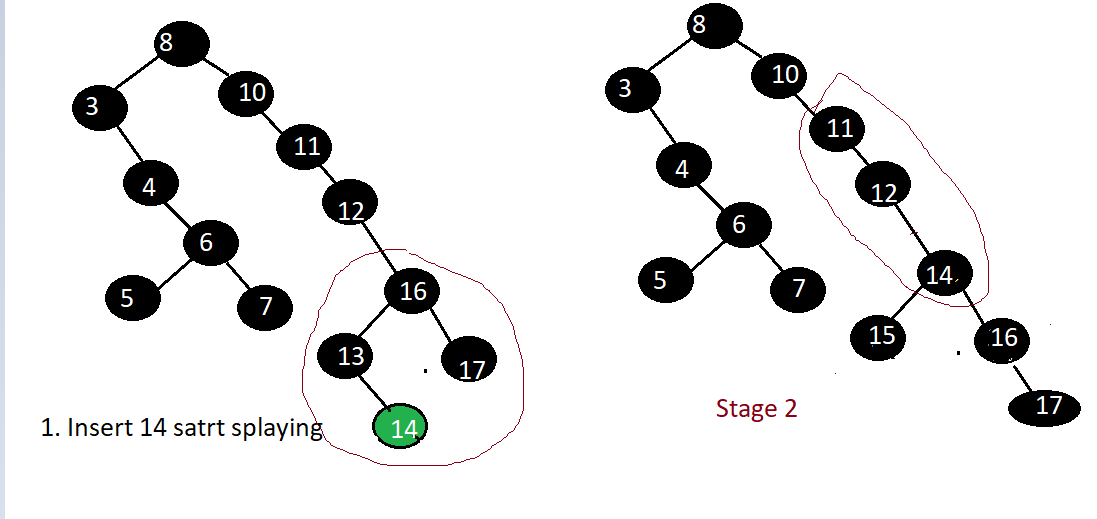
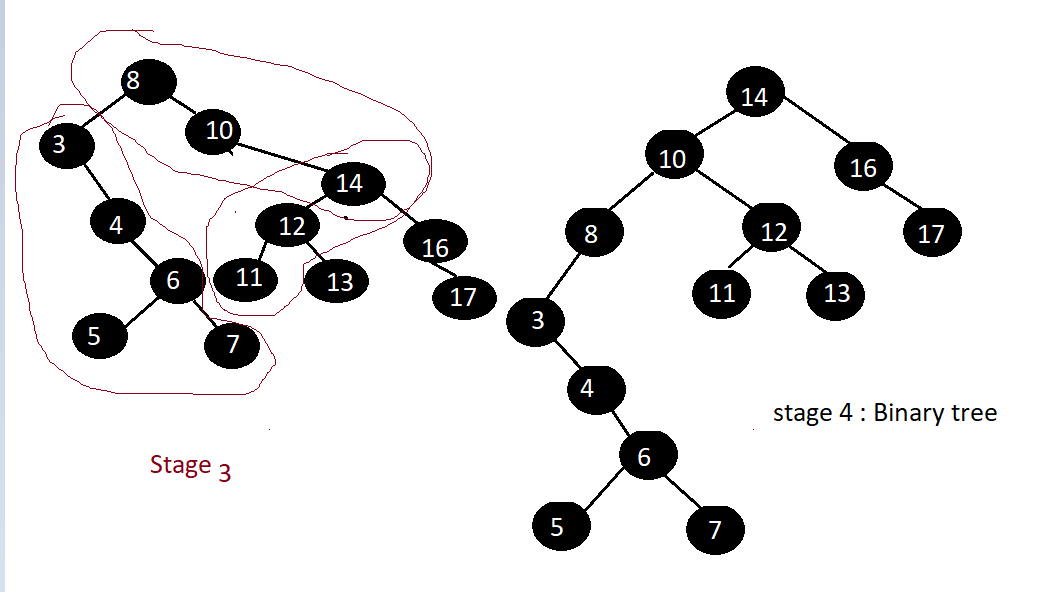
**Splay Trees - Splaying**

**Splaying is moving node from its position to the root**

**Depend on the position of the node**

**Also depends on the its parent and grand parent**

**Example lets consider we have inserted node 14 perform Zig Zag ordering**

**Performance Analysis of Splay Trees**

**Zig-Zig : O(d)**

**Zig-Zag : : O(d)**

**Zig: : O(d)**

**Searching : O(h)**

**Inserting : O{h)**

**Deleting: O(h)**

**Heaps**

**What are Priority Queues?**

**Queue is a data structure that is**

* **Collections of objects**
* **FIFO**

**Example Tickets booking queue**

**Air traffic control.**

**Sometime FIFO policy does not works hence priority queue come into picture**

**Example in ticket booking Senior citizen given high priority.**

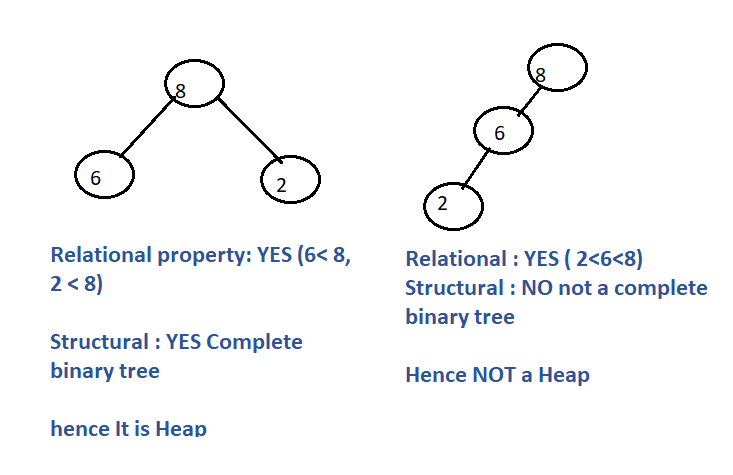
**Priority Queue**

* **Collection of prioritized objects**
* **Insertion: According to first come basis**
* **Removal: Based on priority of objects**
* **Key is associated when element in inserted in priority queue**
* **Element with minimum key will be next element to be removed**

**Heaps Data Structure**

**Use to implement priority queue. We can perform insertion and deletion in much better ways.**

* **Heaps are priority queue which store collection**
* **Collections are stored using binary tree. Hence heap are also known as binary heap.**
* **Relational property: Key in each node is greater than or equal to its children**
* **Structural property: Binary tree should be complete binary tree.**
* **Max Heap and Min Heap:** 
  + **If key in each node greater than or equal its children called max heap**
  + **If key in each node is smaller than or equal it children called min heap.**



**Heap Abstract data types:**

**Members**

* **Max size**
* **Current Size**

**Operations:**

1. **Insert: : Insert an element according to property of heap.**
2. **DeleteMax(): Delete and return maximum element from the heap.**
3. **Max (): Return maximum element from heap.**

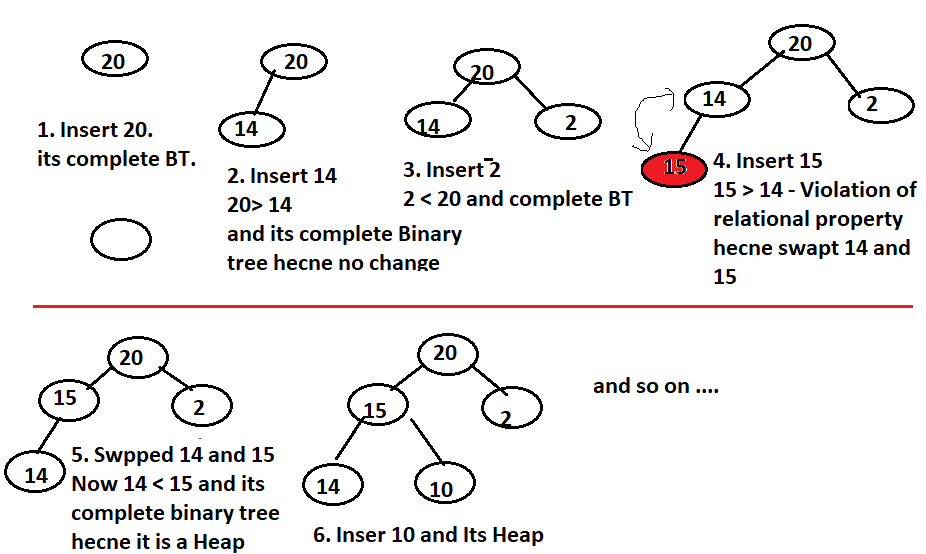
**Heaps - Insertion**

**Element is inserted in heap it should satisfy below two properties**

**Structural property: Node is inserted after the last node always. If its not complete binary tree (Having continues leaf node) rearrange nodes**

**Relational Property: Element should be greater than children if not perform swapping element value. Perform up heap bubbling**

**O(log n)**



**Heaps - Insert Function**

**Index of parent can be found by dividing index of child by 2**

**We will create heap using array**

**While inserting element always compare element with its parent and we will find Index of parent = Index Of child / 2 this comparison loop will swap and continue till root node**

public void insert(int e)

{

if (csize == maxsize)

{

Console.WriteLine("No Space in Heap");

return;

}

csize = csize + 1;

int hi = csize;

while (hi > 1 && e > data[hi / 2])

{

data[hi] = data[hi / 2];

hi = hi / 2;

}

data[hi] = e;

}

**IMP**

**e > data[hi / 2])** this check if inserting element is grater than parent

and **hi > 1** repeat loop till we reach swap to root.

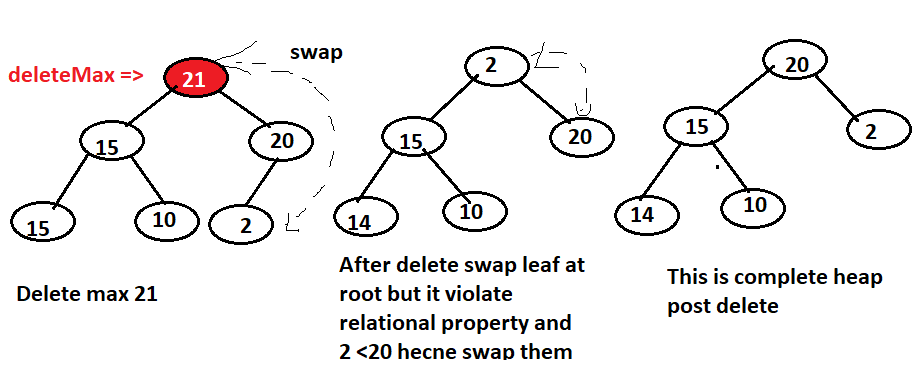
**Heaps - Deletion**

**Heap is priority queue hence we cannot delete any arbitrary element.**

**Element is removed from the root of heap.**

**Structural property: Root is replaced by last node.**

**Relational Property: Perform down heap bubbling. Compare and swap from root to leaf**



**Heaps - Delete Function**

**We know we always delete element from root of if and root has max element**

1. **Get max root value in element which we will return at end**
2. **Replace last node with root node data[1] = data[cSize]**
3. **Then delete last node data[cSize] = null**
4. **Decrement Size of heap size— -**
5. **Here structural property is satisfy but relational < than property violate**
6. **Hence to down heap bubbling goes on comparing root with next element and swap**
7. **So declare i=1; j = I \*2 . where i is index of parent and j is index of child**
8. **Now perform down heap bubbling**
9. **While j <= csize then**
10. **Compare left right child if(data[j] < data[j+1] where I is root j is left child j+1 id right child**
11. **Then j=j+1; //Where j is the element use to swap with its parent Then**
12. **Then check if this swap element id greater then root if data[i] < data[j] then**
13. **Perform swap temp = data[i]; data[i] = data[j]; data[j’ = temp; I = j;**
14. **J = j\*2;**

**Lab: Deletion in Heaps - Implementation**

public int deletemax()

{

if (isEmpty())

{

Console.WriteLine("Heap is Empty");

return -1;

}

int e = data[1];

data[1] = data[csize]; // replace last element with root

data[csize] = -1; // delete last element

csize = csize - 1; .. Decrement current size

int i = 1;

int j = i \* 2;

while (j <= csize) //Down heap bubbling to make compete relational property

{

if (data[j] < data[j + 1]) // right child > left child

j = j + 1;

if (data[i] < data[j]) // element in parent is less than element in child

{

int temp = data[i];

data[i] = data[j];

data[j] = temp;

i = j;

j = i \* 2;

}

else

break;

}

return e;

}

**Heap Sort - How does it Work**

**First to create Heap from array then delete max and stored is sorted array.**

**Hence, we need Insert and delete both operation**

**Complete Heap program:**

class Heap

{

int[] data;

int maxsize;

int csize;

public Heap()

{

maxsize = 10;

data = new int[maxsize];

csize = 0;

for (int i = 0; i < data.Length; i++)

data[i] = -1;

}

public int length()

{

return csize;

}

public bool isEmpty()

{

return csize == 0;

}

public void insert(int e)

{

if (csize == maxsize)

{

Console.WriteLine("No Space in Heap");

return;

}

csize = csize + 1;

int hi = csize;

while (hi > 1 && e > data[hi / 2])

{

data[hi] = data[hi / 2];

hi = hi / 2;

}

data[hi] = e;

}

public int deletemax()

{

if (isEmpty())

{

Console.WriteLine("Heap is Empty");

return -1;

}

int e = data[1];

data[1] = data[csize];

data[csize] = -1;

csize = csize - 1;

int i = 1;

int j = i \* 2;

while (j <= csize)

{

if (data[j] < data[j + 1])

j = j + 1;

if (data[i] < data[j])

{

int temp = data[i];

data[i] = data[j];

data[j] = temp;

i = j;

j = i \* 2;

}

else

break;

}

return e;

}

public void heapsort(int[] A, int n)

{

Heap h = new Heap();

for (int i = 0; i < n; i++)

h.insert(A[i]);

int k = n - 1;

for (int i = 0; i < n; i++)

{

A[k] = h.deletemax();

k = k - 1;

}

}

public int max()

{

if (isEmpty())

{

Console.WriteLine("Heap is Empty");

return -1;

}

return data[1];

}

public void display()

{

for (int i = 0; i < data.Length; i++)

Console.Write(data[i] + " ");

Console.WriteLine();

}

}

**Hashing**

**What is Hashing ?**

**Hashing is a technique used for searching inserting deleting elements from a collection.**

**How fast is searching?**

**Linear search: O(n)**

**Binary Search: O (log n) : but elements are in sorted order**

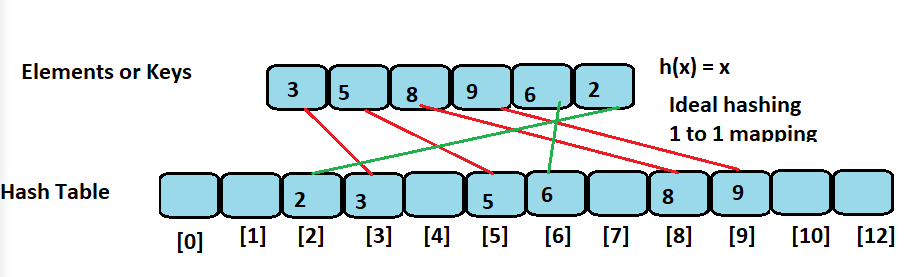
**Binary Search tree: O(h) – O (log n)**

**Hashing: O (1) – required more memory**

**I Hashing we use hash table to store data**

**Hash function map elements to corresponding indices.**

**Lest take elements and add element at index(element ) in hash table example 3 will be added at index 3 in hash table**



**In above case has function is h(x) = x;**

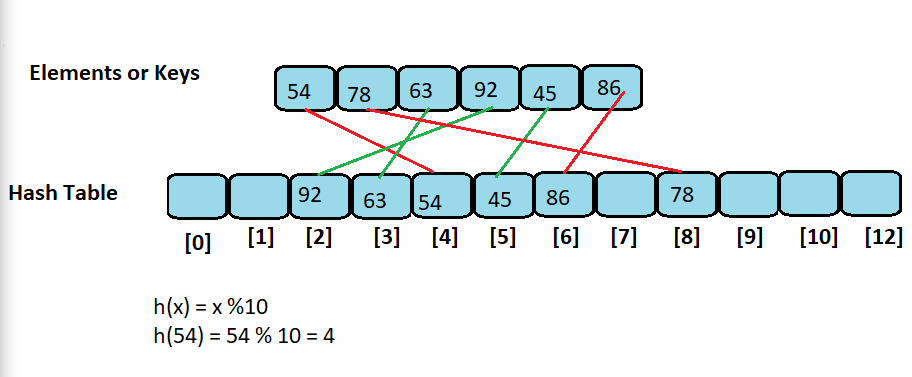
**But what is we have a lots of gap in elements or key in this case we lost memory in hash hence in this case hash index will be calculated using hash function?**

**H(x) = x % 10 .**

**In this search only index is not sufficient we need to match element as well**

**Example when we try to search 35**

**H(35) = 35 % 10 == 5 but element present at 5 is 45.**



**Drawback of hash function h(x) = x %10**

**Collision when we try to insert 28 and 78 both will give same index as 8**

**H(28) = 28 % 10 == 8 and h(78) = 78 % 10 == 8 this is called collision.**

**Hence, we need collision handling scheme**

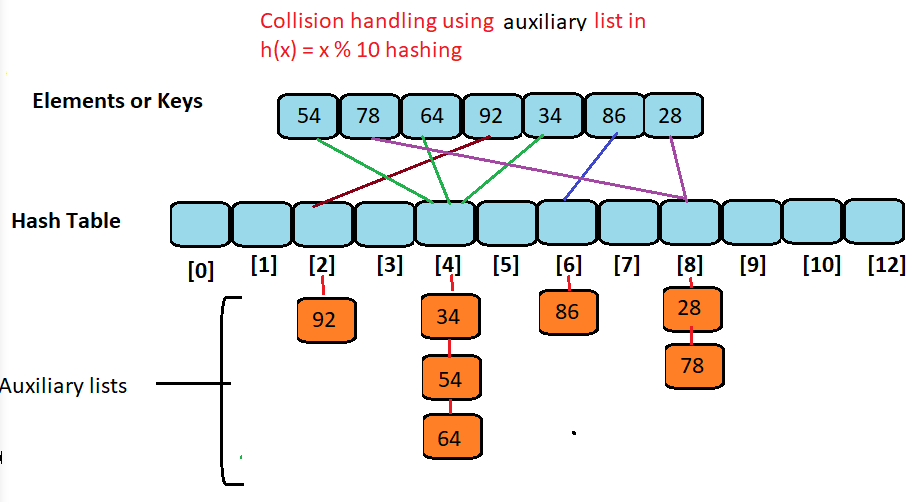
**Collision: when more then one elements or key maps to the same index in hash table**

1. **Chaining: Use auxiliary list to store colliding elements. Store list of elements at index**
2. **Open Addressing**
   1. **Linear Probing:**
   2. **Quadratic Probing**
   3. **Double Hashing**

**Chaining - Collision Detection Scheme**

**In this type we use auxiliary list to store common elements at same index in h(x) = x % 10**

**Its simplest way of handling collision.**



**Auxiliary list is sorted and search element from low to high**

**Lest we have a n elements and size of hash table is s**

**Loading factor = n/2**

**Time complexity O(n/s)**

**Implement HashTable using linked list example :**

namespace CoddingHC

{

public class HashChain

{

int hashtablesize;

LinkedList[] hashtable;

public HashChain()

{

hashtablesize = 10;

hashtable = new LinkedList[hashtablesize]; // Create hash table

for (int i = 0; i < hashtablesize; i++)

hashtable[i] = new LinkedList(); //Each hash table entry will hold list of element computed by hashCode fucntion.

}

public int hashcode(int key)

{

return key % hashtablesize; // This fucntion use to find index of hash table where we can puch element

}

public void insert(int element)

{

int i = hashcode(element);

hashtable[i].insertsorted(element);

}

public bool search(int key)

{

int i = hashcode(key);

return hashtable[i].search(key) != -1;

}

public void display()

{

for (int i = 0; i < hashtablesize; i++)

{

Console.Write("[" + i + "]");

hashtable[i].display();

}

Console.WriteLine();

}

}

public class Node

{

public int element;

public Node next;

public Node(int e, Node n)

{

element = e;

next = n;

}

}

class LinkedList

{

private Node head;

private Node tail;

private int size;

public LinkedList()

{

head = null;

tail = null;

size = 0;

}

public int length()

{

return size;

}

public bool isEmpty()

{

return size == 0;

}

public void addlast(int e)

{

Node newest = new Node(e, null);

if (isEmpty())

head = newest;

else

tail.next = newest;

tail = newest;

size = size + 1;

}

public void addFirst(int e)

{

Node newest = new Node(e, null);

if (isEmpty())

{

head = newest;

tail = newest;

}

else

{

newest.next = head;

head = newest;

}

size = size + 1;

}

public void addAny(int e, int position)

{

if (position <= 0 || position >= size)

{

Console.WriteLine("Invalid Position");

return;

}

Node newest = new Node(e, null);

Node p = head;

int i = 1;

while (i < position - 1)

{

p = p.next;

i = i + 1;

}

newest.next = p.next;

p.next = newest;

size = size + 1;

}

public void insertsorted(int e)

{

Node newest = new Node(e, null);

if (isEmpty())

head = newest;

else

{

Node p = head;

Node q = head;

while (p != null && p.element < e)

{

q = p;

p = p.next;

}

if (p == head)

{

newest.next = head;

head = newest;

}

else

{

newest.next = q.next;

q.next = newest;

}

}

size = size + 1;

}

public int removeFirst()

{

if (isEmpty())

{

Console.WriteLine("List is Empty");

return -1;

}

int e = head.element;

head = head.next;

size = size - 1;

if (isEmpty())

tail = null;

return e;

}

public int removeLast()

{

if (isEmpty())

{

Console.WriteLine("List is Empty");

return -1;

}

Node p = head;

int i = 1;

while (i < size - 1)

{

p = p.next;

i = i + 1;

}

tail = p;

p = p.next;

int e = p.element;

tail.next = null;

size = size - 1;

return e;

}

public int removeAny(int position)

{

if (position <= 0 || position >= size - 1)

{

Console.WriteLine("Invalid Position");

return -1;

}

Node p = head;

int i = 1;

while (i < position - 1)

{

p = p.next;

i = i + 1;

}

int e = p.next.element;

p.next = p.next.next;

size = size - 1;

return e;

}

public int search(int key)

{

Node p = head;

int index = 0;

while (p != null)

{

if (p.element == key)

return index;

p = p.next;

index = index + 1;

}

return -1;

}

public void display()

{

Node p = head;

while (p != null)

{

Console.Write(p.element + " --> ");

p = p.next;

}

Console.WriteLine();

}

public int sum()

{

Node p = head;

int total = 0;

while (p != null)

{

total = total + p.element;

p = p.next;

}

return total;

}

public int maxelement()

{

Node p = head;

int max = -1;

while (p != null)

{

if (p.element > max)

max = p.element;

p = p.next;

}

return max;

}

}

}

**Linear Probing:**

**Open addressing scheme to handle collision**

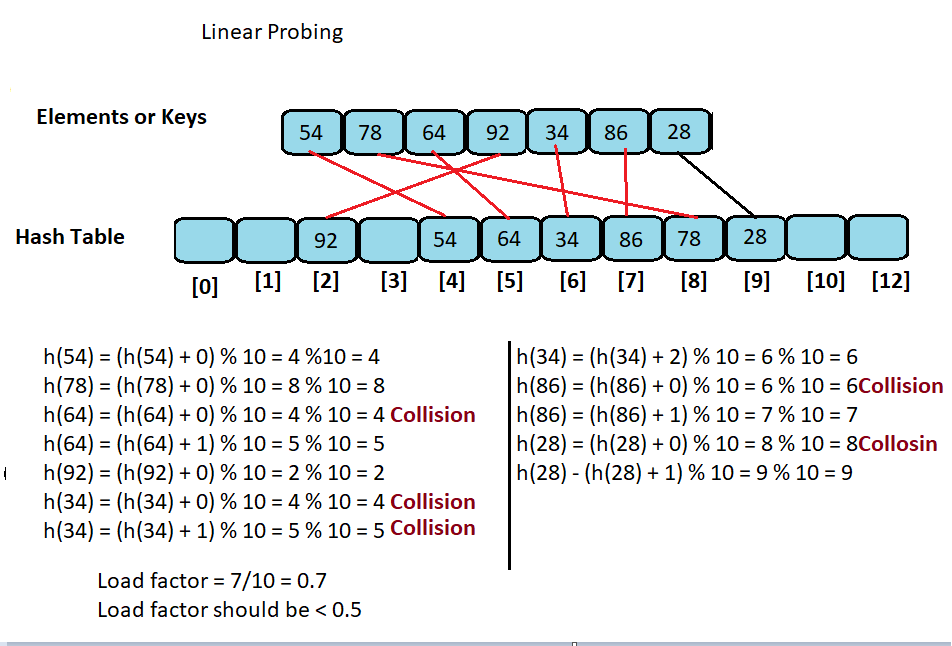
**Insert element in the next available index if cell is already occupied**

**In this technic if element found at index find next empty index to insert element**

**Hash**

**Hence has function created as follows**

* **First find index – h(x) = x % 10**
* **If index occupied, then find next empty slot using – h(x) = (h(x)) % 10**
* **If index occupied, then find next empty slot using – h(x) = (h(x) +1) % 10**
* **If index occupied, then find next empty slot using – h(x) = (h(x) + 2) % 10**



**Lab: Hash Linear Probing - Implementation**

**Code**

namespace Codding

{

public class HashLinearProbe

{

int hashtablesize;

int[] hashtable;

public HashLinearProbe()

{

hashtablesize = 10;

hashtable = new int[hashtablesize];

}

public int hashcode(int key)

{

return key % hashtablesize;

}

public int lprobe(int element)

{

int i = hashcode(element);

int j = 0;

while (hashtable[(i + j) % hashtablesize] != 0)

j = j + 1;

return (i + j) % hashtablesize;

}

public void insert(int element)

{

int i = hashcode(element);

if (hashtable[i] == 0)

hashtable[i] = element;

else

{

i = lprobe(element);

hashtable[i] = element;

}

}

public bool search(int key)

{

int i = hashcode(key);

int j = 0;

while (hashtable[(i + j) % hashtablesize] != key)

{

if (hashtable[(i + j) % hashtablesize] == 0)

return false;

j = j + 1;

}

return true;

}

public void display()

{

for (int i = 0; i < hashtablesize; i++)

Console.Write(hashtable[i] + " ");

Console.WriteLine();

}

}

}

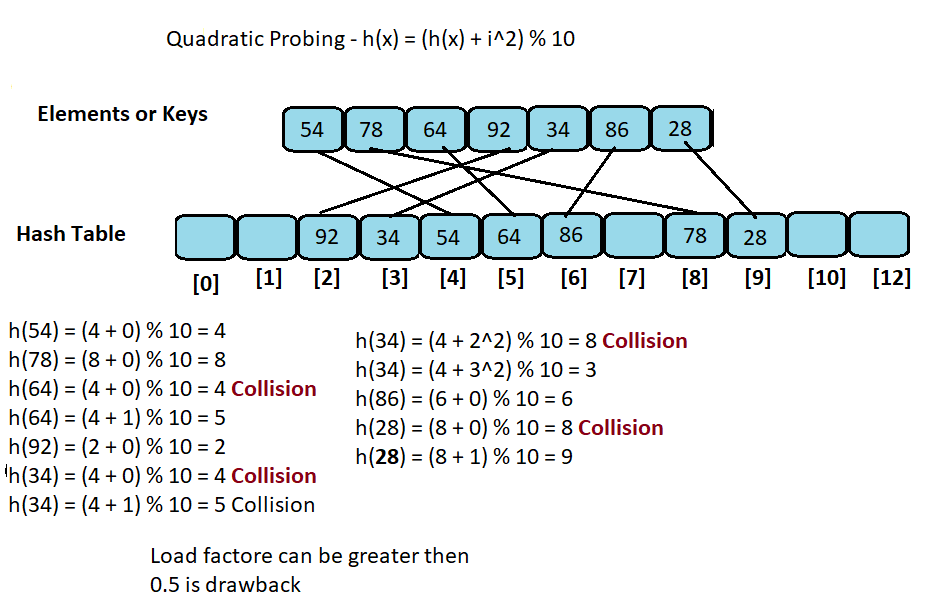
**Quadratic Probing**

**Is a open addressing scheme to handle collision?**

**If we insert element which is available in hash table, we will insert element next slot which is calculated using quadratic formula**

**Hash table index calculate using quadratic function**

**H(x) = (h(x) + i^2) % 10**



**Double Hashing**

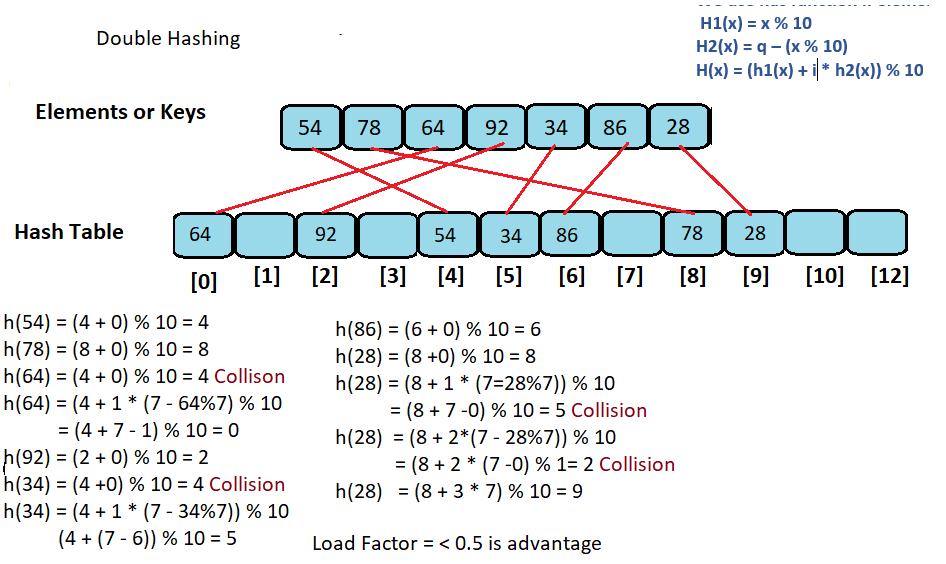
**Is one open addressing scheme collision handling**

**We use has function if element is already occupied, we use other hash function**

**H1(x) = x % 10**

**H2(x) = q – (x % 10)**

**H(x) = (h1(x) + i \* h2(x)) % 10**



**Graphs**

**Graphs - Introduction**

**Graph is a relationship that exist between objects**

**It is a collection of objects along with pairwise connection between objects**

**The objects are called vertices**

**Pairwise relations between objects are called edges**

**Hence graph is a collection of objects called as vertices and together with relationship between them called as edges**

**Graph(G) = {V, E}**

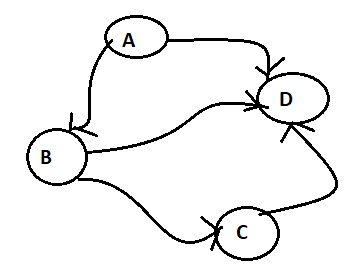
**Vertices also called as nodes**

**And edges as arcs**

**Example**

**Vertices (V) = {A, B, C, D}**

**Edges (E) = {A -> B, A -> D, B -> C, B -> D, C- > D}**

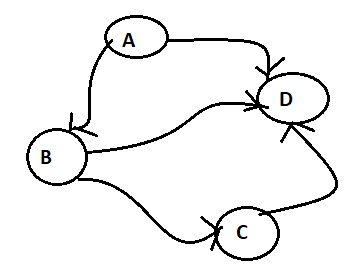


**Graph edges Types**

1. **Directed edge: An edge (u, v) is a directed if pair (u, v) is ordered with u preceding v.**

**Edge is oriented or direction**

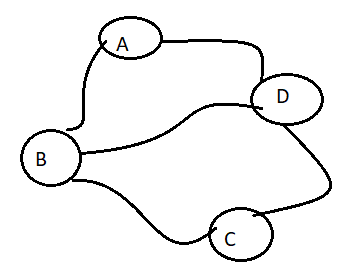
**If all edge has direction graph called Directed graph or digraph**



1. **Undirected edge: An edge (u,v) is undirected of pair (u,v) is not ordered**

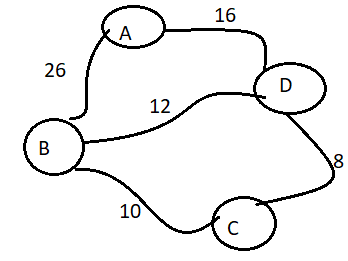
**Edge has no orientation**

**If all edge are undirected then graph is called undirected graph**

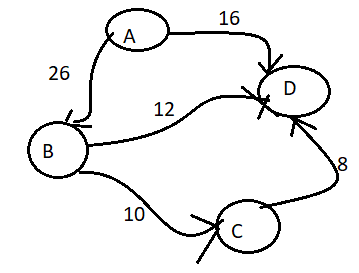


1. **Weighted edge: Cost or weight is assigned to each edge**

**Weighted undirected graph**



**Weighted Directed graph**



**End vertices: Two vertices joined by edge**

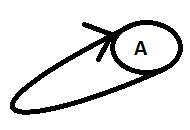
**Adjacent Vertices: Two vertices are adjacent if there is an edge between them**

**Incident edge: If vertex is one of the end points**

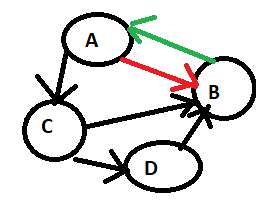
**Outgoing Edge: Origin is the vertex**

**Incoming Edge: Destination is the vertex**

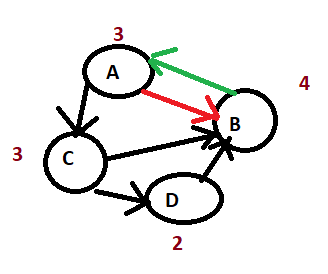
**Self-loop: if two end points are same**



**Parallel edges: Edge from U to V as well as an edge from v to U**



**Degree of a Vertex: Is the number of edges connecting to vertex**



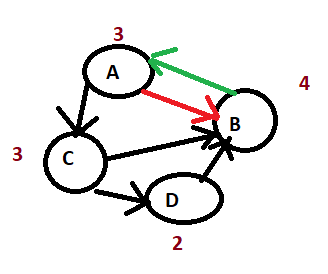
**In Degree: Number for incoming edges: A = 1**

**Out Degree: Number of outgoing edges: A = 2**

**Path and Cycle**

**Path: is a sequence od edges starting at one vertex and ending at another vertex.**

**In directed graph we need to follow direction of edge to construct a path.**

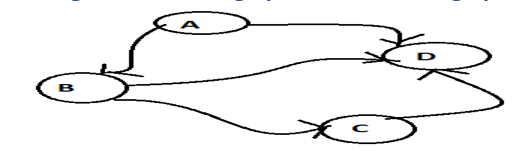


**Path A-C-B, A-C-D-B**

**Cycle: Is a path that start from one vertex and end at same vertex**

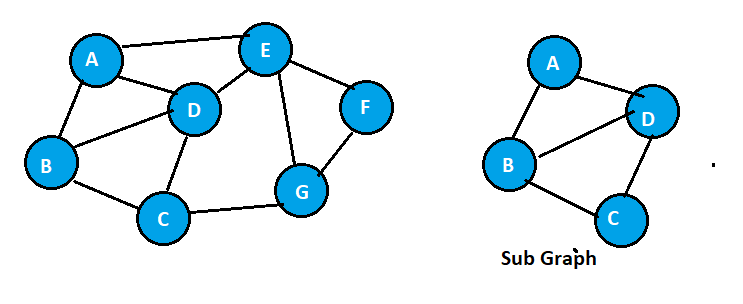
**A-B-C-D-A**

**Directed Acyclic Graph: when there is no cycle in directed graph**

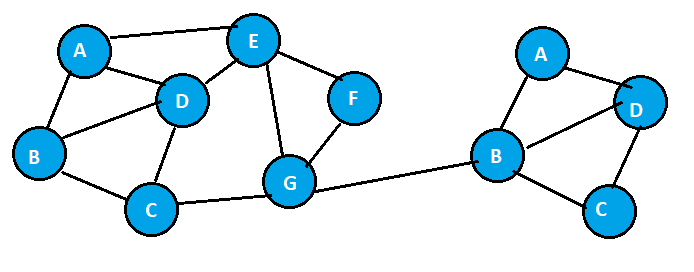


**Subgraphs and Connected Components**

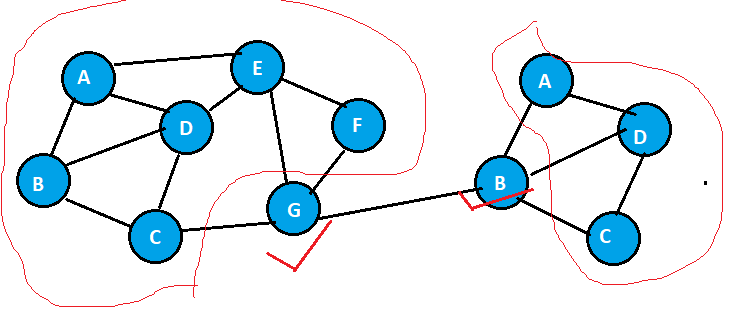
**Subgraph: is a graph whose vertices and edges are subsets of vertices and edges of another graph**



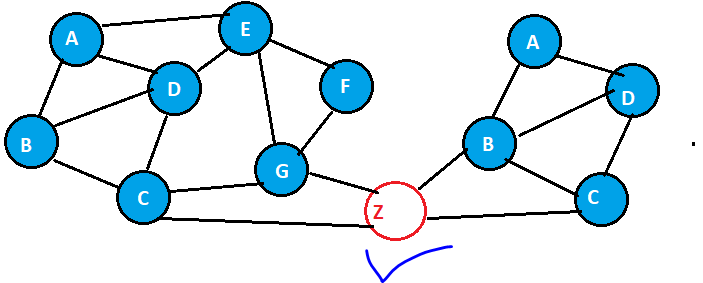
**Connected Components: Connected subgraphs are known as connected components.**



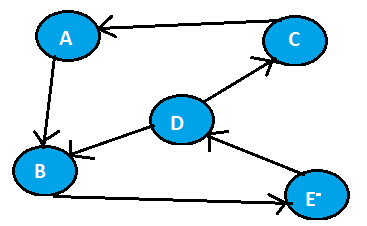
**Articulation Point: Removal of vertex split graph in to two sub graphs**



**Bi-Connected Components: Component connected by two edges. And removal of any one edge will not split graph in to two sub graphs**



**Strongly Connected Graph: All the vertices are reachable from any vertex**



**Graphs Abstract Data Type**

**A Graph is a collection of vertices and edges**

* **Create(n): Create graph with n vertices but no edges**
* **Insert\_edge(u,v,w=1): Create edge from u and v, storing weight w**
* **Remove\_edge(u,v): delete edge from u to v.**
* **Exist\_edge(u,v): return edge if exist between u and v else false**
* **Vertex\_count(): return the number of vertices in graph**
* **Edge\_count(): return number of edges in graph**
* **Vertices(): return all vertices of the graph**
* **Edges(): return all edges of the graph**
* **Degree(u): Return the degree of the vertex U**
* **Indegree(u): return indegree of vertex u**
* **Outdegree(u): return outdegree of vertex u.**

**Graphs Representation**

**A Graph can be represented using different data structure.**

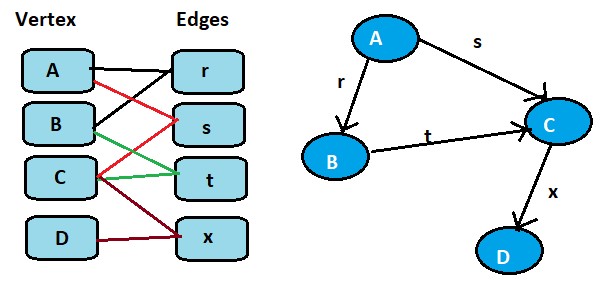
* **Edge List: Maintain list of all edges**
* **Adjacency List: For each vertex separate list of edges is maintained**
* **Adjacency matrix: Maintain matrix of vertices, where each cell store the reference to the edge**

**Edge List Representation**

**Maintain list of all edges**

**Two linked list are use to store vertex and edges**

**Then edges list will have a pointer to vertex**

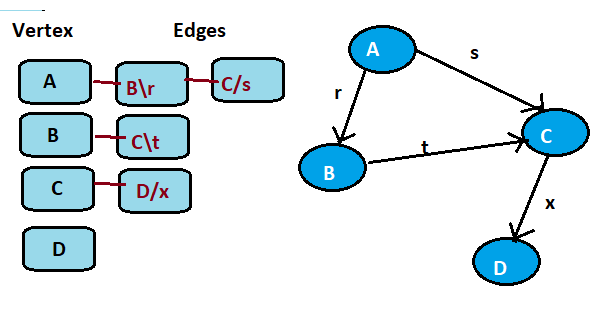


**Time Complexity: O(n+m) where n is vertex and m are edges**

**Adjacency List Representation**

**In this technic for each vertex list if edges is maintain**

**Example**

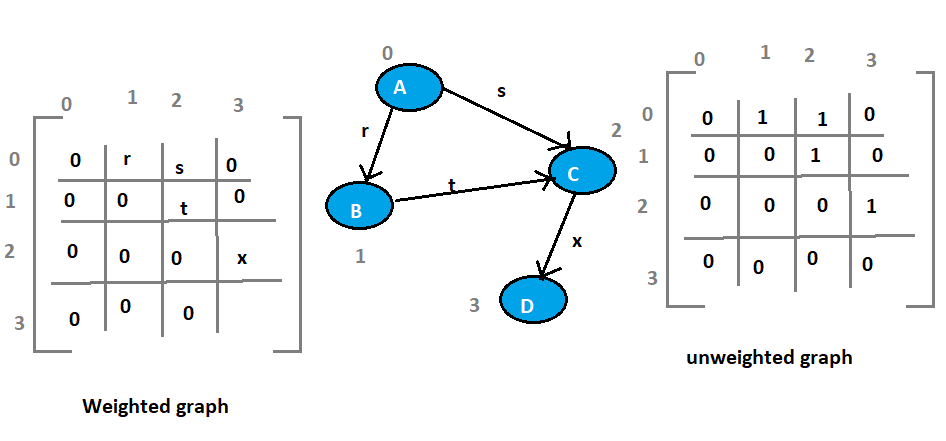


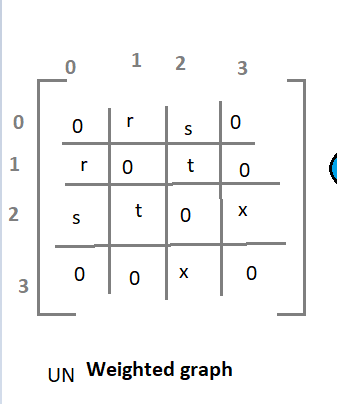
**O(n+m)**

**Adjacency Matrix Representation**

**O(n^2)**

**Maintain the matrix of vertices where each cell stores the references to the edges**





**Graphs Representation - Summary of Performance**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Edge List** | **Adjacency List** | **Adjacency Matrix** |
| **Space Complexity** | **O(n+m)** | **O(n+m)** | **O(n^2)** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Operations** | **Edge List** | **Adjacency List** | **Adjacency Matrix** |
| **Insert\_edge(u,v,w=1)** | **O(1)** | **O(1)** | **O(1)** |
| **Remove\_edge(u,v)** | **O(1)** | **O(1)** | **O(1)** |
| **Exist\_edge(u,v)** | **O(m)** | **O(min(d u, d v))))** | **O(1)** |
| **Vertex\_count()** | **O(1)** | **O(1)** | **O(1)** |
| **Edge\_Count()** | **O(1)** | **O(1)** | **O(1)** |
| **Vertices()** | **O(n)** | **O(n)** | **O(n)** |
| **Edges()** | **O(m)** | **O(m)** | **O(m)** |
| **Degree()** | **O(m)** | **O(1)** | **O(n)** |

**Lab: Graphs ADT - Implementation**

class Graphs

{

int vertices;

int[,] adjMat;

public Graphs(int n)

{

vertices = n;

adjMat = new int[n, n];

}

public void insertEdge(int u, int v, int x)

{

adjMat[u, v] = x;

}

public void removeEdge(int u, int v)

{

adjMat[u, v] = 0;

}

public bool existEdge(int u, int v)

{

return adjMat[u, v] != 0;

}

public int vertexCount()

{

return vertices;

}

public int edgeCount()

{

int count = 0;

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

if (adjMat[i, j] != 0)

count = count + 1;

}

return count;

}

public void edges()

{

Console.WriteLine("Edges:");

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

if (adjMat[i, j] != 0)

Console.WriteLine(i + "--" + j);

}

}

public int outdegree(int v)

{

int count = 0;

for (int j = 0; j < vertices; j++)

if (adjMat[v, j] != 0)

count = count + 1;

return count;

}

public int indegree(int v)

{

int count = 0;

for (int i = 0; i < vertices; i++)

if (adjMat[i, v] != 0)

count = count + 1;

return count;

}

public void display()

{

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

Console.Write(adjMat[i, j] + "\t");

Console.WriteLine();

}

}

**Lab: Undirected Graph - Implementation**

int vertices;

int[,] adjMat;

public Graphs(int n)

{

vertices = n;

adjMat = new int[n, n];

}

public void insertEdge(int u, int v, int x)

{

adjMat[u, v] = x;

}

public void removeEdge(int u, int v)

{

adjMat[u, v] = 0;

}

public bool existEdge(int u, int v)

{

return adjMat[u, v] != 0;

}

public int vertexCount()

{

return vertices;

}

public int edgeCount()

{

int count = 0;

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

if (adjMat[i, j] != 0)

count = count + 1;

}

return count;

}

public void edges()

{

Console.WriteLine("Edges:");

for(int i=0;i<vertices;i++)

for(int j=0;j<vertices;j++)

if(adjMat[i,j]!=0)

Console.WriteLine(i+"--"+j);

}

public int outdegree(int v)

{

int count = 0;

for (int j = 0; j < vertices; j++)

if (adjMat[v, j] != 0)

count = count + 1;

return count;

}

public int indegree(int v)

{

int count = 0;

for (int i = 0; i < vertices; i++)

if (adjMat[i, v] != 0)

count = count + 1;

return count;

}

public void display()

{

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

Console.Write(adjMat[i, j] + "\t");

Console.WriteLine();

}

}

static void Main(string[] args)

{

Graphs g = new Graphs(4);

Console.WriteLine("Graphs Adjacency Matrix:");

g.display();

Console.WriteLine("Vertices: " + g.vertexCount());

Console.WriteLine("Edges Count: " + g.edgeCount());

g.insertEdge(0, 1, 1);

g.insertEdge(0, 2, 1);

g.insertEdge(1, 0, 1);

g.insertEdge(1, 2, 1);

g.insertEdge(2, 0, 1);

g.insertEdge(2, 1, 1);

g.insertEdge(2, 3, 1);

g.insertEdge(3, 2, 1);

Console.WriteLine("Graphs Adjacency Matrix:");

g.display();

Console.WriteLine("Vertices: " + g.vertexCount());

Console.WriteLine("Edges Count: " + g.edgeCount());

g.edges();

Console.WriteLine("Edge between 1--3: " + g.existEdge(1, 3));

Console.WriteLine("Edge between 1--2: " + g.existEdge(1, 2));

Console.WriteLine("Degree of Vertex 2: " + g.indegree(2));

Console.WriteLine("Graphs Adjacency Matrix:");

g.display();

g.removeEdge(1, 2);

Console.WriteLine("Graphs Adjacency Matrix:");

g.display();

Console.WriteLine("Edge between 1--2: " + g.existEdge(1, 2));

Console.ReadKey();

}

**Lab: Weighted Undirected Graph - Implementation**

class GraphsWrightedUndirected

{

int vertices;

int[,] adjMat;

public GraphsWrightedUndirected(int n)

{

vertices = n;

adjMat = new int[n, n];

}

public void insertEdge(int u, int v, int x)

{

adjMat[u, v] = x;

}

public void removeEdge(int u, int v)

{

adjMat[u, v] = 0;

}

public bool existEdge(int u, int v)

{

return adjMat[u, v] != 0;

}

public int vertexCount()

{

return vertices;

}

public int edgeCount()

{

int count = 0;

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

if (adjMat[i, j] != 0)

count = count + 1;

}

return count;

}

public void edges()

{

Console.WriteLine("Edges:");

for (int i = 0; i < vertices; i++)

for (int j = 0; j < vertices; j++)

if (adjMat[i, j] != 0)

Console.WriteLine(i + "--" + j);

}

public int outdegree(int v)

{

int count = 0;

for (int j = 0; j < vertices; j++)

if (adjMat[v, j] != 0)

count = count + 1;

return count;

}

public int indegree(int v)

{

int count = 0;

for (int i = 0; i < vertices; i++)

if (adjMat[i, v] != 0)

count = count + 1;

return count;

}

public void display()

{

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

Console.Write(adjMat[i, j] + "\t");

Console.WriteLine();

}

}

static void Main(string[] args)

{

GraphsWrightedUndirected g = new GraphsWrightedUndirected(4);

Console.WriteLine("Graphs Adjacency Matrix:");

g.display();

Console.WriteLine("Vertices: " + g.vertexCount());

Console.WriteLine("Edges Count: " + g.edgeCount());

g.insertEdge(0, 1, 26);

g.insertEdge(0, 2, 16);

g.insertEdge(1, 0, 26);

g.insertEdge(1, 2, 12);

g.insertEdge(2, 0, 16);

g.insertEdge(2, 1, 12);

g.insertEdge(2, 3, 8);

g.insertEdge(3, 2, 8);

Console.WriteLine("Graphs Adjacency Matrix:");

g.display();

Console.WriteLine("Vertices: " + g.vertexCount());

Console.WriteLine("Edges Count: " + g.edgeCount());

g.edges();

Console.WriteLine("Edge between 1--3: " + g.existEdge(1, 3));

Console.WriteLine("Edge between 1--2: " + g.existEdge(1, 2));

Console.WriteLine("Degree of Vertex 2: " + g.indegree(2));

Console.WriteLine("Graphs Adjacency Matrix:");

g.display();

g.removeEdge(1, 2);

Console.WriteLine("Graphs Adjacency Matrix:");

g.display();

Console.WriteLine("Edge between 1--2: " + g.existEdge(1, 2));

Console.ReadKey();

}

}

**Lab: Directed Graph - Implementation**

class GraphsDirected

{

int vertices;

int[,] adjMat;

public GraphsDirected(int n)

{

vertices = n;

adjMat = new int[n, n];

}

public void insertEdge(int u, int v, int x)

{

adjMat[u, v] = x;

}

public void removeEdge(int u, int v)

{

adjMat[u, v] = 0;

}

public bool existEdge(int u, int v)

{

return adjMat[u, v] != 0;

}

public int vertexCount()

{

return vertices;

}

public int edgeCount()

{

int count = 0;

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

if (adjMat[i, j] != 0)

count = count + 1;

}

return count;

}

public void edges()

{

Console.WriteLine("Edges:");

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

if (adjMat[i, j] != 0)

Console.WriteLine(i + "--" + j);

}

}

public int outdegree(int v)

{

int count = 0;

for (int j = 0; j < vertices; j++)

if (adjMat[v, j] != 0)

count = count + 1;

return count;

}

public int indegree(int v)

{

int count = 0;

for (int i = 0; i < vertices; i++)

if (adjMat[i, v] != 0)

count = count + 1;

return count;

}

public void display()

{

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

Console.Write(adjMat[i, j] + "\t");

Console.WriteLine();

}

}

static void Main(string[] args)

{

GraphsDirected g = new GraphsDirected(4);

Console.WriteLine("Graphs Adjacency Matrix:");

g.display();

Console.WriteLine("Vertices: " + g.vertexCount());

Console.WriteLine("Edges Count: " + g.edgeCount());

g.insertEdge(0, 1, 1);

g.insertEdge(0, 2, 1);

g.insertEdge(1, 2, 1);

g.insertEdge(2, 3, 1);

Console.WriteLine("Graphs Adjacency Matrix:");

g.display();

Console.WriteLine("Vertices: " + g.vertexCount());

Console.WriteLine("Edges Count: " + g.edgeCount());

g.edges();

Console.WriteLine("Edge between 1--3: " + g.existEdge(1, 3));

Console.WriteLine("Edge between 1--2: " + g.existEdge(1, 2));

Console.WriteLine("Edge between 2--1: " + g.existEdge(2, 1));

Console.WriteLine("Degree of Vertex 2: " + (g.indegree(2) + g.outdegree(2)));

Console.WriteLine("In-Degree of Vertex 2: " + g.indegree(2));

Console.WriteLine("Out-Degree of Vertex 2: " + g.outdegree(2));

Console.WriteLine("Graphs Adjacency Matrix:");

g.display();

g.removeEdge(1, 2);

Console.WriteLine("Graphs Adjacency Matrix:");

g.display();

Console.WriteLine("Edge between 1--2: " + g.existEdge(1, 2));

Console.ReadKey();

}

}

**Lab: Weighted Directed Graph - Implementation**

class GraphsWeightedDirected

{

int vertices;

int[,] adjMat;

public GraphsWeightedDirected(int n)

{

vertices = n;

adjMat = new int[n, n];

}

public void insertEdge(int u, int v, int x)

{

adjMat[u, v] = x;

}

public void removeEdge(int u, int v)

{

adjMat[u, v] = 0;

}

public bool existEdge(int u, int v)

{

return adjMat[u, v] != 0;

}

public int vertexCount()

{

return vertices;

}

public int edgeCount()

{

int count = 0;

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

if (adjMat[i, j] != 0)

count = count + 1;

}

return count;

}

public void edges()

{

Console.WriteLine("Edges:");

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

if (adjMat[i, j] != 0)

Console.WriteLine(i + "--" + j);

}

}

public int outdegree(int v)

{

int count = 0;

for (int j = 0; j < vertices; j++)

if (adjMat[v, j] != 0)

count = count + 1;

return count;

}

public int indegree(int v)

{

int count = 0;

for (int i = 0; i < vertices; i++)

if (adjMat[i, v] != 0)

count = count + 1;

return count;

}

public void display()

{

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

Console.Write(adjMat[i, j] + "\t");

Console.WriteLine();

}

}

static void Main(string[] args)

{

GraphsWeightedDirected g = new GraphsWeightedDirected(4);

Console.WriteLine("Graphs Adjacency Matrix:");

g.display();

Console.WriteLine("Vertices: " + g.vertexCount());

Console.WriteLine("Edges Count: " + g.edgeCount());

g.insertEdge(0, 1, 26);

g.insertEdge(0, 2, 16);

g.insertEdge(1, 2, 12);

g.insertEdge(2, 3, 8);

Console.WriteLine("Graphs Adjacency Matrix:");

g.display();

Console.WriteLine("Vertices: " + g.vertexCount());

Console.WriteLine("Edges Count: " + g.edgeCount());

g.edges();

Console.WriteLine("Edge between 1--3: " + g.existEdge(1, 3));

Console.WriteLine("Edge between 1--2: " + g.existEdge(1, 2));

Console.WriteLine("Edge between 2--1: " + g.existEdge(2, 1));

Console.WriteLine("Degree of Vertex 2: " + (g.indegree(2) + g.outdegree(2)));

Console.WriteLine("In-Degree of Vertex 2: " + g.indegree(2));

Console.WriteLine("Out-Degree of Vertex 2: " + g.outdegree(2));

Console.WriteLine("Graphs Adjacency Matrix:");

g.display();

g.removeEdge(1, 2);

Console.WriteLine("Graphs Adjacency Matrix:");

g.display();

Console.WriteLine("Edge between 1--2: " + g.existEdge(1, 2));

Console.ReadKey();

}

}

**Graph Traversals**

**Graph is a nonlinear data stricture hence we cannot visit node using linear technics or we cannot visit them sequentially.**

**Traversal is a systematic procedure exploring a graph.**

**Exploring: Examining all the vertices and edges of the graph**

**Efficient traversal: Visit to all vertices and edges in linear time**

**Graph traversal algorithm are used to analyze how to travel from one vertices to other following oaths in the graph**

**Graphs Traversal – Undirected Graph:**

* **Computing a path from one vertex to another vertex**
* **Compute path to reach all other vertices**
* **Find whether a graph is connected**
* **Computing connected components of the graph**
* **Computing cycle in a graph**
* **Computing spanning tree of the graph**

**Graphs Traversal – Directed Graph:**

* **Computing direct path from one vertex to other vertex**
* **Finding all the vertices that can be reachable from a given vertex**
* **Determine whether a graph is strongly connected**
* **Determine whether a graph is cyclic**

**Graph Traversal algorithms:**

* **Breadth First Search**
* **Depth First Search**

**Breadth First Search**

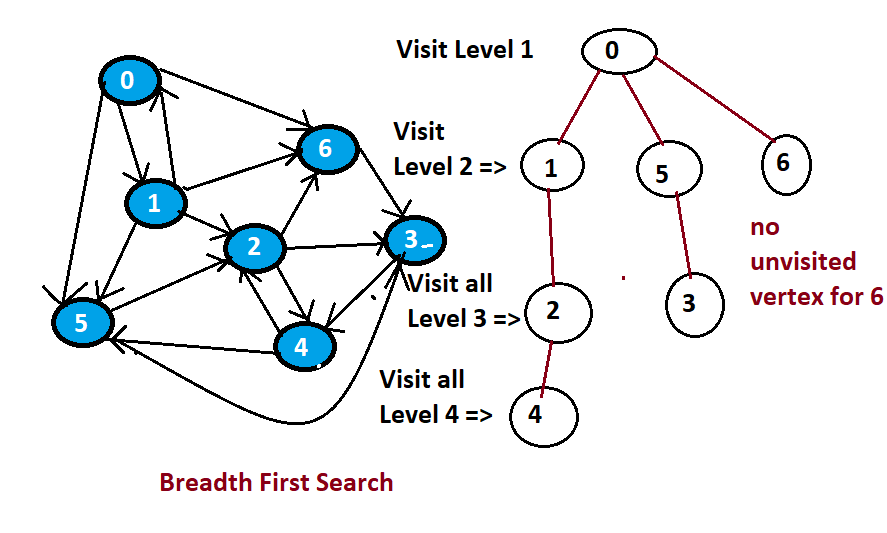
**Breadth first search subdivides the vertices into levels and proceeds in rounds**

**Start at a vertex which is considered at level 0**

**Identifies all the vertices reachable from starts vertex at level 1, marks them visited**

**In next round identifies new vertices reachable from level 1 vertices which are not yet visited marks them visited**

**This process continue until no vertices are found**



**0 -> 1 -> 5 -> 6 -> 2 -> 3 -> 4**

**Breadth First Search Algorithm**

**Keep a vertex in a queue and traverse queue and remove vertex from queue and put in visited array**

**BFS(s)**

**i=s; , q = Queue(), visited[] = [0,0,0,….n-1]**

**print(i) // Vertex visited**

**visites[i] = 1 // Mark visited first root**

**q.enqueue(i) // add visited in queee**

**while ! q.isEmpty then**

**I = q. dequeue // pull out element**

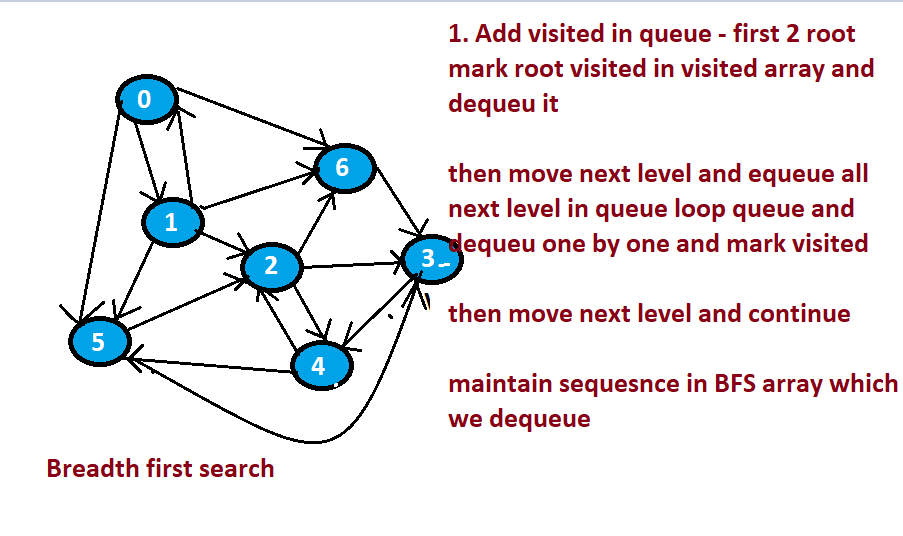
**For j=0;j<n;j++**

**If(adjMat[i][j] == 1 && visited[j] == 0) // find if not visited**

**Print(j)**

**Visited[j] = 1**

**q.enqueue{j}**



**Lab: Breadth First Search - Implementation**

using System;

using System.Collections.Generic;

using System.Text;

namespace CoddingBFS

{

class Graphs

{

int vertices;

int[,] adjMat;

public Graphs(int n)

{

vertices = n;

adjMat = new int[n, n];

}

public void insertEdge(int u, int v, int x)

{

adjMat[u, v] = x;

}

public void removeEdge(int u, int v)

{

adjMat[u, v] = 0;

}

public bool existEdge(int u, int v)

{

return adjMat[u, v] != 0;

}

public int vertexCount()

{

return vertices;

}

public int edgeCount()

{

int count = 0;

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

if (adjMat[i, j] != 0)

count = count + 1;

}

return count;

}

public void edges()

{

Console.WriteLine("Edges:");

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

if (adjMat[i, j] != 0)

Console.WriteLine(i + "--" + j);

}

}

public int outdegree(int v)

{

int count = 0;

for (int j = 0; j < vertices; j++)

if (adjMat[v, j] != 0)

count = count + 1;

return count;

}

public int indegree(int v)

{

int count = 0;

for (int i = 0; i < vertices; i++)

if (adjMat[i, v] != 0)

count = count + 1;

return count;

}

public void display()

{

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

Console.Write(adjMat[i, j] + "\t");

Console.WriteLine();

}

}

public void BFS(int s)

{

int i = s;

QueuesLinked q = new QueuesLinked();

int[] visited = new int[vertices];

Console.Write(i + " ");

visited[i] = 1;

q.enqueue(i);

while (!q.isEmpty())

{

i = q.dequeue();

for (int j = 0; j < vertices; j++)

{

if (adjMat[i, j] == 1 && visited[j] == 0)

{

Console.Write(j + " ");

visited[j] = 1;

q.enqueue(j);

}

}

}

}

static void Main(string[] args)

{

Graphs g = new Graphs(7);

g.insertEdge(0, 1, 1);

g.insertEdge(0, 5, 1);

g.insertEdge(0, 6, 1);

g.insertEdge(1, 0, 1);

g.insertEdge(1, 2, 1);

g.insertEdge(1, 5, 1);

g.insertEdge(1, 6, 1);

g.insertEdge(2, 3, 1);

g.insertEdge(2, 4, 1);

g.insertEdge(2, 6, 1);

g.insertEdge(3, 4, 1);

g.insertEdge(4, 2, 1);

g.insertEdge(4, 5, 1);

g.insertEdge(5, 2, 1);

g.insertEdge(5, 3, 1);

g.insertEdge(6, 3, 1);

Console.WriteLine("Breadth First Search:");

g.BFS(0);

Console.ReadKey();

}

}

public class Node

{

public int element;

public Node next;

public Node(int e, Node n)

{

element = e;

next = n;

}

}

class QueuesLinked

{

Node front;

Node rear;

int size;

public QueuesLinked()

{

front = null;

rear = null;

size = 0;

}

public int length()

{

return size;

}

public bool isEmpty()

{

return size == 0;

}

public void enqueue(int e)

{

Node newest = new Node(e, null);

if (isEmpty())

front = newest;

else

rear.next = newest;

rear = newest;

size = size + 1;

}

public int dequeue()

{

if (isEmpty())

{

Console.WriteLine("Queue is Empty");

return -1;

}

int e = front.element;

front = front.next;

size = size - 1;

if (isEmpty())

rear = null;

return e;

}

public void display()

{

Node p = front;

while (p != null)

{

Console.Write(p.element + " --> ");

p = p.next;

}

Console.WriteLine();

}

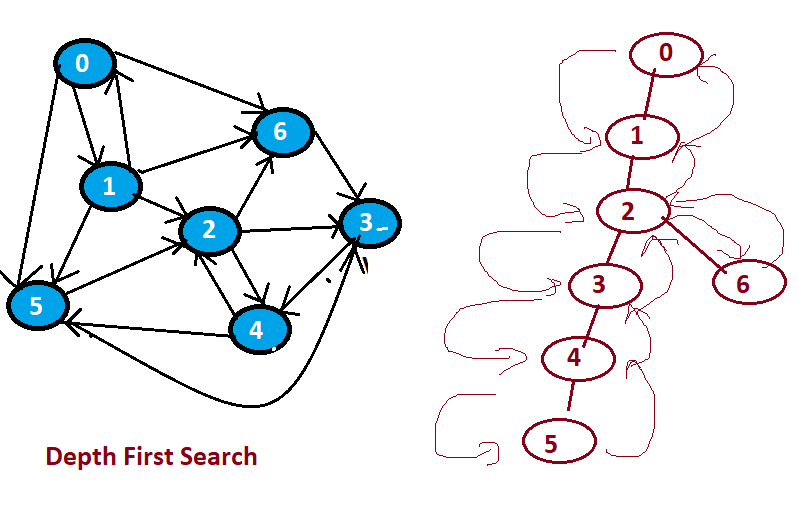
}

}

**Depth First Search**

**Gose on visiting ajcenet vertex until last then start from last to up to find unvisited vertex**

* **Depth first search start at a vertex**
* **Selects the adjacent vertex from the start vertex**
* **Visit the adjacent vertex mark as visited**
* **Continue the above procedure until there are no more unexplored edges then terminate**



**Depth First Search Algorithm**

**DFS(s)**

**If(visited[s] == 0 then**

**Print(s)**

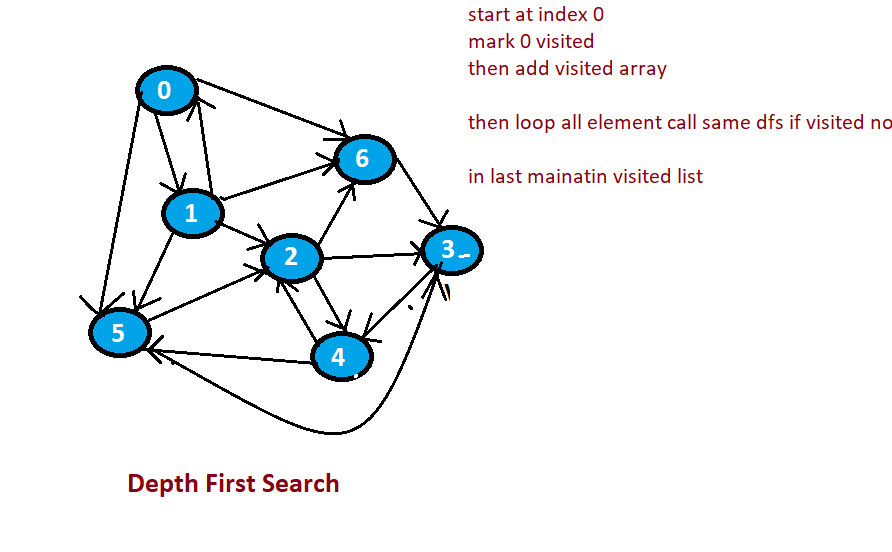
**Visites[s] = 1;**

**For j =0; j< n; j ++**

**If(ajmat[s][j] == 1 && visited[j] == 0) then**

**Dfs(j)**

**In this search first select root element mark it visited then loop all elements check is visited not then called recursively for that element when loop dose not found op dose not found visited element will exit loop and recursion will not called**



**Lab: Depth First Search - Implementation**

using System;

using System.Collections.Generic;

using System.Text;

namespace Codding

{

class GraphDepthFirstSearch

{

int vertices;

int[,] adjMat;

int[] visited;

public GraphDepthFirstSearch(int n)

{

vertices = n;

adjMat = new int[n, n];

visited = new int[vertices];

}

public void insertEdge(int u, int v, int x)

{

adjMat[u, v] = x;

}

public void removeEdge(int u, int v)

{

adjMat[u, v] = 0;

}

public bool existEdge(int u, int v)

{

return adjMat[u, v] != 0;

}

public int vertexCount()

{

return vertices;

}

public int edgeCount()

{

int count = 0;

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

if (adjMat[i, j] != 0)

count = count + 1;

}

return count;

}

public void edges()

{

Console.WriteLine("Edges:");

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

if (adjMat[i, j] != 0)

Console.WriteLine(i + "--" + j);

}

}

public int outdegree(int v)

{

int count = 0;

for (int j = 0; j < vertices; j++)

if (adjMat[v, j] != 0)

count = count + 1;

return count;

}

public int indegree(int v)

{

int count = 0;

for (int i = 0; i < vertices; i++)

if (adjMat[i, v] != 0)

count = count + 1;

return count;

}

public void display()

{

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

Console.Write(adjMat[i, j] + "\t");

Console.WriteLine();

}

}

public void DFS(int s)

{

if (visited[s] == 0)

{

Console.Write(s + " ");

visited[s] = 1;

for (int j = 0; j < vertices; j++)

if (adjMat[s, j] == 1 && visited[j] == 0)

DFS(j);

}

}

static void Main(string[] args)

{

GraphDepthFirstSearch g = new GraphDepthFirstSearch(7);

g.insertEdge(0, 1, 1);

g.insertEdge(0, 5, 1);

g.insertEdge(0, 6, 1);

g.insertEdge(1, 0, 1);

g.insertEdge(1, 2, 1);

g.insertEdge(1, 5, 1);

g.insertEdge(1, 6, 1);

g.insertEdge(2, 3, 1);

g.insertEdge(2, 4, 1);

g.insertEdge(2, 6, 1);

g.insertEdge(3, 4, 1);

g.insertEdge(4, 2, 1);

g.insertEdge(4, 5, 1);

g.insertEdge(5, 2, 1);

g.insertEdge(5, 3, 1);

g.insertEdge(6, 3, 1);

Console.WriteLine("Depth First Search:");

g.DFS(0);

Console.ReadKey();

}

}

}

**Top Qu**estions

Want some extra practice? Here are a list of some of the top interview questions focusing on data structures and algorithms:#344 [Reverse String](https://leetcode.com/problems/reverse-string)

#412 [Fizz Buzz](https://leetcode.com/problems/fizz-buzz)

#136 [Single Number](https://leetcode.com/problems/single-number)

#104 [Maximum Depth of Binary Tree](https://leetcode.com/problems/maximum-depth-of-binary-tree)

#283 [Move Zeroes](https://leetcode.com/problems/move-zeroes)

#371 [Sum of Two Integers](https://leetcode.com/problems/sum-of-two-integers)

#206 [Reverse Linked List](https://leetcode.com/problems/reverse-linked-list)

#171 [Excel Sheet Column Number](https://leetcode.com/problems/excel-sheet-column-number)

#169 [Majority Element](https://leetcode.com/problems/majority-element)

#13 [Roman to Integer](https://leetcode.com/problems/roman-to-integer)

#237 [Delete Node in a Linked List](https://leetcode.com/problems/delete-node-in-a-linked-list)

#122 [Best Time to Buy and Sell Stock II](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii)

#242 [Valid Anagram](https://leetcode.com/problems/valid-anagram)

#217 [Contains Duplicate](https://leetcode.com/problems/contains-duplicate)

#387 [First Unique Character in a String](https://leetcode.com/problems/first-unique-character-in-a-string)

#108 [Convert Sorted Array to Binary Search Tree](https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree)

#268 [Missing Number](https://leetcode.com/problems/missing-number)

#350 [Intersection of Two Arrays II](https://leetcode.com/problems/intersection-of-two-arrays-ii)

#121 [Best Time to Buy and Sell Stock](https://leetcode.com/problems/best-time-to-buy-and-sell-stock)

#21 [Merge Two Sorted Lists](https://leetcode.com/problems/merge-two-sorted-lists)

#202 [Happy Number](https://leetcode.com/problems/happy-number)

#118 [Pascal's Triangle](https://leetcode.com/problems/pascals-triangle)

#70 [Climbing Stairs](https://leetcode.com/problems/climbing-stairs)

#101 [Symmetric Tree](https://leetcode.com/problems/symmetric-tree)

#53 [Maximum Subarray](https://leetcode.com/problems/maximum-subarray)

#326 [Power of Three](https://leetcode.com/problems/power-of-three)

#191 [Number of 1 Bits](https://leetcode.com/problems/number-of-1-bits)

#198 [House Robber](https://leetcode.com/problems/house-robber)

#66 [Plus One](https://leetcode.com/problems/plus-one)

#1 [Two Sum](https://leetcode.com/problems/two-sum)

#38 [Count and Say](https://leetcode.com/problems/count-and-say)

#26 [Remove Duplicates from Sorted Array](https://leetcode.com/problems/remove-duplicates-from-sorted-array)

#172 [Factorial Trailing Zeroes](https://leetcode.com/problems/factorial-trailing-zeroes)

#20 [Valid Parentheses](https://leetcode.com/problems/valid-parentheses)

#141 [Linked List Cycle](https://leetcode.com/problems/linked-list-cycle)

#234 [Palindrome Linked List](https://leetcode.com/problems/palindrome-linked-list)

#88 [Merge Sorted Array](https://leetcode.com/problems/merge-sorted-array)

#155 [Min Stack](https://leetcode.com/problems/min-stack)

#14 [Longest Common Prefix](https://leetcode.com/problems/longest-common-prefix)

#160 [Intersection of Two Linked Lists](https://leetcode.com/problems/intersection-of-two-linked-lists)

#28 [Implement strStr()](https://leetcode.com/problems/implement-strstr)

#69 [Sqrt(x)](https://leetcode.com/problems/sqrtx)

#190 [Reverse Bits](https://leetcode.com/problems/reverse-bits)

#125 [Valid Palindrome](https://leetcode.com/problems/valid-palindrome)

#189 [Rotate Array](https://leetcode.com/problems/rotate-array)

#204 [Count Primes](https://leetcode.com/problems/count-primes)

#7 [Reverse Integer](https://leetcode.com/problems/reverse-integer)

#94 [Binary Tree Inorder Traversal](https://leetcode.com/problems/binary-tree-inorder-traversal)

**Amazon Interview Questions**

If you would like to practice real life interview questions asked by Amazon, then here they are below. Keep in mind that it will be hard to just get everything right from the beginning. With enough practice you will become better and better, but there is an entire community of us learning, so I recommend you tackle these questions together with our online Discord community (see lesson #3 in this course for the link) and join the conversation and tackle problems in the #interview-questions channel:  
  
**1.** [Past Interview Questions](https://www.glassdoor.ca/Interview/Amazon-Software-Development-Engineer-Interview-Questions-EI_IE6036.0,6_KO7,36.htm)

**2.** *From Leetcode*:  
  
#1 [Two Sum](https://leetcode.com/problems/two-sum)

#2 [Add Two Numbers](https://leetcode.com/problems/add-two-numbers)

#3 [Longest Substring Without Repeating Characters](https://leetcode.com/problems/longest-substring-without-repeating-characters)

#200 [Number of Islands](https://leetcode.com/problems/number-of-islands)

#20 [Valid Parentheses](https://leetcode.com/problems/valid-parentheses)

#5 [Longest Palindromic Substring](https://leetcode.com/problems/longest-palindromic-substring)

#138 [Copy List with Random Pointer](https://leetcode.com/problems/copy-list-with-random-pointer)

#121 [Best Time to Buy and Sell Stock](https://leetcode.com/problems/best-time-to-buy-and-sell-stock)

#21 [Merge Two Sorted Lists](https://leetcode.com/problems/merge-two-sorted-lists)

3. Bonus Question asked by Amazon:

From: https://www.dailycodingproblem.com/

There's a staircase with N steps, and you can climb 1 or 2 steps at a time. Given N, write a function that returns the number of unique ways you can climb the staircase. The order of the steps matters.

For example, if N is 4, then there are 5 unique ways:

* 1, 1, 1, 1
* 2, 1, 1
* 1, 2, 1
* 1, 1, 2
* 2, 2

What if, instead of being able to climb 1 or 2 steps at a time, you could climb any number from a set of positive integers X? For example, if X = {1, 3, 5}, you could climb 1, 3, or 5 steps at a time. Generalize your function to take in X.

Solution

It's always good to start off with some test cases. Let's start with small cases and see if we can find some sort of pattern.

* N = 1: [1]
* N = 2: [1, 1], [2]
* N = 3: [1, 2], [1, 1, 1], [2, 1]
* N = 4: [1, 1, 2], [2, 2], [1, 2, 1], [1, 1, 1, 1], [2, 1, 1]

What's the relationship?

The only ways to get to N = 3, is to first get to N = 1, and then go up by 2 steps, or get to N = 2 and go up by 1 step. So f(3) = f(2) + f(1).

Does this hold for N = 4? Yes, it does. Since we can only get to the 4th step by getting to the 3rd step and going up by one, or by getting to the 2nd step and going up by two. So f(4) = f(3) + f(2).

To generalize, f(n) = f(n - 1) + f(n - 2). That's just the [Fibonacci sequence](https://en.wikipedia.org/wiki/Fibonacci_number)!

1. def staircase(n):
2. if n <= 1:
3. return 1
4. return staircase(n - 1) + staircase(n - 2)

Of course, this is really slow (O(2N)) — we are doing a lot of repeated computations! We can do it a lot faster by just computing iteratively:

1. def staircase(n):
2. a, b = 1, 2
3. for \_ in range(n - 1):
4. a, b = b, a + b
5. return a

Now, let's try to generalize what we've learned so that it works if you can take a number of steps from the set X. Similar reasoning tells us that if X = {1, 3, 5}, then our algorithm should be f(n) = f(n - 1) + f(n - 3) + f(n - 5). If n < 0, then we should return 0 since we can't start from a negative number of steps.

1. def staircase(n, X):
2. if n < 0:
3. return 0
4. elif n == 0:
5. return 1
6. else:
7. return sum(staircase(n - x, X) for x in X)

This is again, very slow (O(|X|N)) since we are repeating computations again. We can use dynamic programming to speed it up.

Each entry cache[i] will contain the number of ways we can get to step i with the set X. Then, we'll build up the array from zero using the same recurrence as before:

1. def staircase(n, X):
2. cache = [0 for \_ in range(n + 1)]
3. cache[0] = 1
4. for i in range(1, n + 1):
5. cache[i] += sum(cache[i - x] for x in X if i - x >= 0)
6. return cache[n]

This now takes O(N \* |X|) time and O(N) space.

Facebook Interview Questions

If you would like to practice real life interview questions asked by Facebook, then here they are below. Keep in mind that it will be hard to just get everything right from the beginning. With enough practice you will become better and better, but there is an entire community of us learning, so I recommend you tackle these questions together with our online Discord community (see lesson #3 in this course for the link) and join the conversation and tackle problems in the #interview-questions channel:  
  
**1.** [Past Facebook Interview Questions](https://www.glassdoor.ca/Interview/Facebook-Software-Engineer-Interview-Questions-EI_IE40772.0,8_KO9,26.htm)  
  
**2.** *From Leetcode:*

#1 [Two Sum](https://leetcode.com/problems/two-sum)

#200 [Number of Islands](https://leetcode.com/problems/number-of-islands)

#20 [Valid Parentheses](https://leetcode.com/problems/valid-parentheses)

#121 [Best Time to Buy and Sell Stock](https://leetcode.com/problems/best-time-to-buy-and-sell-stock)

#56 [Merge Intervals](https://leetcode.com/problems/merge-intervals)

#206 [Reverse Linked List](https://leetcode.com/problems/reverse-linked-list)

3. Bonus Interview Question asked by Facebook:

Determine the 10 most frequent words given a terabyte of strings. ([solution](https://stackoverflow.com/questions/12525455/most-frequent-words-in-a-terabyte-of-data))

**Google Interview Questions**

If you would like to practice real life interview questions asked by Google, then here they are below. Keep in mind that it will be hard to just get everything right from the beginning. With enough practice you will become better and better, but there is an entire community of us learning, so I recommend you tackle these questions together with our online Discord community (see lesson #3 in this course for the link) and join the conversation and tackle problems in the #interview-questions channel:  
  
**1.** [Past Google Interview Questions](https://www.careercup.com/page?pid=google-interview-questions)  
  
**2.** *From Leetcode:*

#155 [Min Stack](https://leetcode.com/problems/min-stack)

#200 [Number of Islands](https://leetcode.com/problems/number-of-islands)

#20 [Valid Parentheses](https://leetcode.com/problems/valid-parentheses)

#42 [Trapping Rain Water](https://leetcode.com/problems/trapping-rain-water)

#56 [Merge Intervals](https://leetcode.com/problems/merge-intervals)

#681 [Next Closest Time](https://leetcode.com/problems/next-closest-time)

#139 [Word Break](https://leetcode.com/problems/word-break)

#31 [Next Permutation](https://leetcode.com/problems/next-permutation)

References

<https://adports.udemy.com/course/data-structures-and-algorithms-in-depth-using-c-sharp/learn/lecture/22619922#overview>

https://adports.udemy.com/course/data-structures-and-algorithms-in-depth-using-c-sharp/learn/lecture/22674743#overview