**Algorithms-Binary Search**

**Objective :** To find an element in an sorted array

**Input:** A sorted array, arrA[] and an key

**Output :** Return true if element is found, else false.

**Approach:** The idea is to compare the middle element of array with the key, if key equal to the middle element , that’s it you have find your element, return true. If key is greater than the middle element, chuck out the first half of the array, you wont find your key in the first half and do the recursive search on the right half of the array and vice versa.

If(mid\_element==key)

return true;

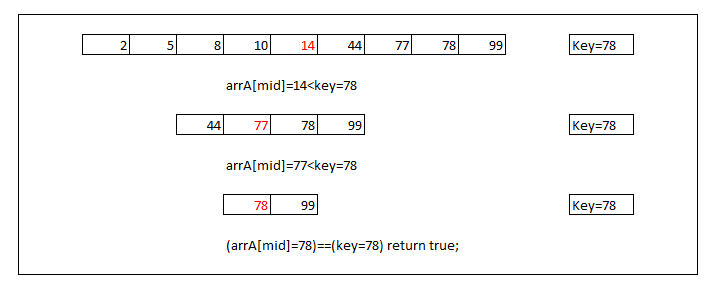
else if (mid<key)

do recursive search on the right half of the array.

else

do recursive search on the left half of the array.

**Time Complexity**: O(logN) –since we are eliminating half of the array with every comparison.



**Complete Code:**

**package** interviewQuestion;

**public** **class** BinarySearch {

**private** **int** [] arrA;

**private** **int** number;

**public** BinarySearch(**int** [] arrA){

**this**.arrA = arrA;

}

**public** Boolean Search(**int** low,**int** high, **int** number){

**if**(low>high){

**return** **false**;

}

**int** mid = (high+low)/2;

**if**(arrA[mid]==number)**return** **true**;

**else** **if** (arrA[mid]>number) **return** Search(low,mid-1,number);

**else** **return** Search(mid+1,high,number);

}

**public** **static** **void** main(String args[]){

**int** [] a = {2,5,8,10,14,44,77,78,99};

**int** number = 99;

BinarySearch b = **new** BinarySearch(a);

System.*out*.println("The "+ number + " present in array a ??? :" + b.Search(0, a.length-1, number));

number = 76;

System.*out*.println("The "+ number + " present in array a ??? :" + b.Search(0, a.length-1, number));

}

}

Output:

The 99 present in array a ??? :true

The 76 present in array a ??? :false

Download link:

**Algorithms – Merge Sort**

**Objective :** To sort elements in an array

**Input:** A insorted array, arrA[].

**Output :** A sorted array.

**Approach:**

**Divide and Conquer:** In this approach we divide the main problems into smaller problems, solve them and merge the results to get the final result.

**How Divide and conquer works in Merge Sort:**

We divide the elements into two half’s by middle of the array. We solve the left half and right half recursively and merge the results.

**Merging:**

Once the sorting is done individually on both the half’s, our next task will be merge them. To merge we start with both the arrays at the beginning, pick the smaller one put into array and then compare the next elements and so on.

4

9

10

5

3

6

1

2

6

5

4

3

2

1

10

9

5

4

6

3

2

1

10

9

9

4

6

3

2

1

10

5

4

9

10

5

3

6

1

2

4

9

10

5

3

6

1

2

4

9

10

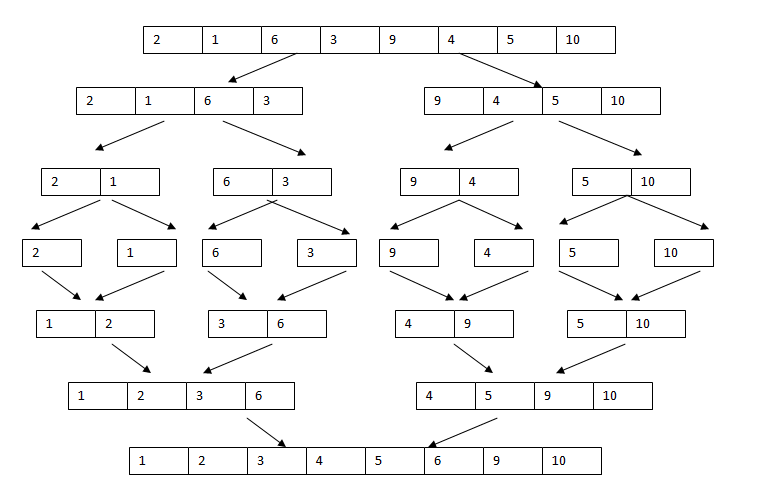
5

3

2

6

1



**Time Complexity : O(nlogn) { O(logn) for dividing and O(n) for merging.**

**Note: we can make merging more efficient by implementing these approaches**

**Using Auxiliary Array with copying data** – In this approach you wont create new array everytime for merging instead you create Auxiliary array. This will save memory for you.

**Alternate Merging Between Primary and Auxiliary Array:** This is the best approach for merging. You don’t copy the entire array to auxiliary array for merging instead you do alternate merging between main array and auxiliary array.

Below is the running time comparison between all three approaches

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Size** | **Dynamic Memo Allocation Algo** | **Using Auxillary Array with copying data** | **Alternate Merging Between Primary and Auxillary Array** |
| 1 Million | 600-630 mili sec | 450-470 mili sec | 400-425 mili sec |
| 10 million | 6 secs | 4.2 secs | 2.3 secs |
| 100 million | 56 secs | 46 sec | 18 sec |

**You can find the implementation of all these approaches here –**

<https://github.com/SumitJainUTD/DataStructuresAlgos/tree/master/3%20Different%20Impl%20of%20Merge%20Sort>

**Complete Code:**

**package** interviewQuestion;

**public** **class** MergeSort {

**private** **int** arrSize;

**private** **int** [] arrAux;

**private** **int** [] arrInput;

**public** MergeSort(**int** [] arrInput){

**this**.arrInput = arrInput;

arrSize = arrInput.length;

arrAux = **new** **int** [arrSize];

}

**public** **int**[] mergeSorting(){

sort(0,arrSize-1);

**return** arrInput;

}

**public** **void** sort(**int** low, **int** high){

**if**(low<high){

**int** mid = low+((high-low))/2;

sort(low,mid);

sort(mid+1,high);

merge(low, mid, high);

}

}

**public** **void** merge(**int** low, **int** mid, **int** high){

//copy the entire array in the Auxilary array

**for**(**int** i=low;i<=high;i++){

arrAux[i] = arrInput[i];

}

**int** i = low;

**int** j = mid+1;

**int** k = low;

**while**(i<=mid && j<=high){

**if**(arrAux[i]<=arrAux[j]){

arrInput[k]=arrAux[i];

i++;

}

**else**{

arrInput[k]=arrAux[j];

j++;

}

k++;

}

**while**(i<=mid){

arrInput[k]=arrAux[i];

i++;

k++;

}

}

**public** **void** displayArray(**int** [] b){

**for**(**int** i=0;i<b.length;i++){

System.*out*.print(" " + b[i]);

}

}

**public** **static** **void** main(String[] args){

**int** [] a = {2,1,6,3,9,4,5,10};

MergeSort m = **new** MergeSort(a);

**int** [] b = m.mergeSorting();

m.displayArray(b);

}

}

Output :

1 2 3 4 5 6 9 10

**Algorithms - Hash Table Implementation**

**Objective :** To implement a Hash Table

**Input:** A set of pairs of keys and values

**Approach:**

* **Create a Hash Table**
  + Hashtable<Integer, String> ht = new Hashtable<Integer, String>();
* **Insert values in hash table using put(key,value)**
  + ht.put(key, value);
* **Get values from hash table using get(key)** 
  + ht.get(key);

|  |  |
| --- | --- |
| Key | Value |
| 1 | Sumit |
| 2 | Raghav |
| 3 | Rishi |

hashTable Object  
Advantage : The search time for any element is O(1) since it uses key to find an element so it takes constant time. But drawback is that it takes extra space.

**Complete Code:**

**package** interviewQuestion;

**import** java.util.Hashtable;

**public** **class** SimpleHashTable {

**int** [] a = **new** **int**[5];

String [] arrNames = **new** String[]{"Sumit","Jain","Raghav","Garg","Gaurav","Rishi"};

Hashtable<Integer, String> ht = **new** Hashtable<Integer, String>();

**public** **void** insertValues(){

**for**(**int** i=0;i<arrNames.length;i++ ){

ht.put(i+1,arrNames[i]);

}

}

**public** String getValue(**int** key){

**return** ht.get(key);

}

**public** **static** **void** main (String [] args){

SimpleHashTable sht = **new** SimpleHashTable();

sht.insertValues();

System.*out*.println("All values inserted");

System.*out*.println("Employee with ID 1 is "+ sht.getValue(1));

System.*out*.println("Employee with ID 3 is "+ sht.getValue(6));

}

}

Output:

All values inserted

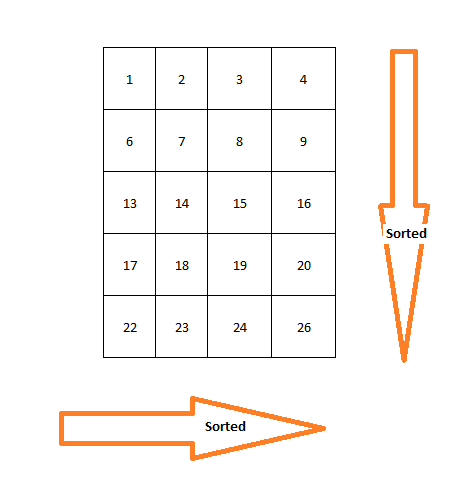
Employee with ID 1 is Sumit

Employee with ID 3 is Rishi

**Algorithms – Find an Element in 2 dimensional sorted array**

**Objective :** To **Find an Element in 2 dimensional array where rows and columns are sorted respectively.**

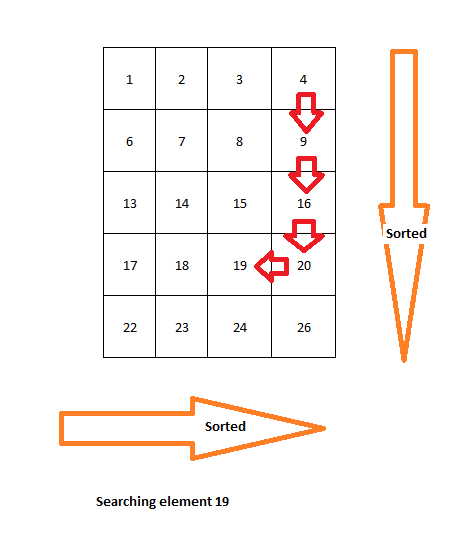
**Input:** A two dimensianl sorted array, arrA[][].



**Output :** True or false based on whether element exists and its location

**Approach:**

* Start from the left top corner, say ele;
* If ele>number -> move left
* If ele<number -> move right
* If you cant move further to find the number , return false



Complete Code:

**package** interviewQuestion;

//Here objective is find an element in two dimensional array

//all rows and columns of an array are sorted in ascending order respectively

**public** **class** FindElementInSorted2DArray {

**public** Boolean findElement(**int** [][] arrA, **int** number){

//start from the left top corner, say ele;

//if ele>number -> move left

//if ele<number -> move right

//if you cant move further to find the number , return false

**int** row = 0;

**int** col = arrA[1].length-1;

**boolean** numberFound = **false**;

System.*out*.print("The Movement : " );

**while**(numberFound==**false**){

**int** ele = arrA[row][col];

System.*out*.print(ele + "->" );

**if**(ele==number){

**return** **true**;

}

**else** **if**(ele>number)col--;

**else** **if** (ele<number)row++;

**if**(row>arrA[0].length-1||col<0)**return** **false**;

}

**return** **false**;

}

**public** **static** **void** main(String args[]){

**int** [][] a = {{1,2,3,4},{6,7,8,9},{13,14,15,16},{17,18,19,20},{22,23,24,26}};

**int** number1 = 15;

**int** number2 = 5;

**int** number3 = 19;

**int** number4 = 25;

FindElementInSorted2DArray f = **new** FindElementInSorted2DArray();

System.*out*.println("The "+ number1 + " present in 2D array a ??? :" + f.findElement(a, number1));

System.*out*.println("The "+ number2 + " present in 2D array a ??? :" + f.findElement(a, number2));

System.*out*.println("The "+ number3 + " present in 2D array a ??? :" + f.findElement(a, number3));

System.*out*.println("The "+ number4 + " present in 2D array a ??? :" + f.findElement(a, number4));

}

}

Output:

The Movement : 4->9->16->15->The 15 present in 2D array a ??? :true

The Movement : 4->9->8->7->6->The 5 present in 2D array a ??? :false

The Movement : 4->9->16->20->19->The 19 present in 2D array a ??? :true

The Movement : 4->9->16->20->The 25 present in 2D array a ??? :false

**Algorithms – Find a Missing Number From a Sequence of Consecutive Numbers**

**Objective :** Find a Missing Number From a Sequence of Consecutive Numbers

**Input:** Array, arrA[] with a missing number and Range

**Output :** missing number

**Approach:**

* Approach is very simple, Add all the given numbers say S
* Calculate sum of N numbers by formula n(n+1)/2 , say N
* Find missing number m = N-S

Example : suppose array given is {1,2,3,4,5,6,8,9,10} and range is 10.

So N will be sum of 1 to 10 = 10(10+1)/2 = 55

S will be sum of all the array elements which is = 48

So missing number will be = 55-48 = 7

**Complete Code:**

**package** interviewQuestion;

//find the missing number from the sequence of consecutive number

//Approach is very simple, Add all the given numbers say S

//Calculate sum of N numbers by formula n(n+1)/2 , say N

//Find missing number m = N-S

**public** **class** FindMissingNumber {

**int** Sum;

**int** Sum\_N;

**public** **int** missingNumber(**int** [] arrA, **int** size){

Sum\_N = size\*(size+1)/2;

**for**(**int** i=0;i<arrA.length;i++){

Sum +=arrA[i];

}

**return** Sum\_N-Sum;

}

**public** **static** **void** main(String args[]){

**int** [] arrA = {1,2,3,4,5,7,8,9,10};

System.*out*.println("Missing number is :" + (**new** FindMissingNumber()).missingNumber(arrA,10));

}

}

Output :

Missing number is :6

**Algorithms – Find two Missing Numbers in a Sequence of Consecutive Numbers**

**Objective :** Find two Missing Numbers in a Sequence of Consecutive Numbers

**Input:** Array, arrA[] with two missing numbers and Range

**Output :** Two missing numbers

**Approach:**

* Approach is very simple, Add all the given numbers say S
* Calculate sum of N numbers by formula n(n+1)/2 , say N
* Find sum of two missing numbers a+b = N-S
* Now take the product of all given numbers say P
* Now take the product of N numbers , say Np;
* Find the product of two missing numbers ab = Np-P
* Now we have a+b and ab , we can easily calculate a and b

**Example :**

Given array : {10,2,3,5,7,8,9,1}; Range : 10

N (Sum of 1 to 10 ) = 55

S (Sum of given elements ) = 45

a+b = 10------------------------------------(1)

Np(Product of 1 to 10) = 3628800

P(Product of given elements) = 151200

So a\*b = 24---------------------------------(2)

Now we have two equations and two variables, if we solve we will get values 6 and 4.

**Complete Code:**

**package** interviewQuestion;

//find the two missing numbers from the sequence of consecutive number

//Approach is very simple, Add all the given numbers say S

//Calculate sum of N numbers by formula n(n+1)/2 , say N

//Find sum of two missing numbers a+b = N-S

//Now take the product of all given numbers say P

//now take the product of N numbers , say Np;

//find the product of two missing numbers ab = Np-P

//now we have a+b and ab , we can easily calculate a and b

**public** **class** FindTwoMissingNumbers {

**int** Sum;

**int** SumN;

**int** P=1;

**int** Np=1;

**int** a,b;

**public** **int** [] missingNumbers(**int** [] arrA, **int** range){

SumN = range\*(range+1)/2;

**for**(**int** i=0;i<arrA.length;i++){

Sum +=arrA[i];

}

**int** s= SumN-Sum;

**for**(**int** i=0;i<arrA.length;i++){

P \*=arrA[i];

}

**for**(**int** i=1;i<=range;i++){

Np \*=i;

}

**int** product = Np/P;

// System.out.println(product);

**int** diffSqr = (**int**)Math.*sqrt*(s\*s-4\*product); // (a-b)^2 = (a+b)^2-4ab

a = (s+diffSqr)/2;

b= s-a;

**int** [] result = {a,b};

**return** result;

}

**public** **static** **void** main(String args[]){

**int** [] arrA = {10,2,3,5,7,8,9,1};

FindTwoMissingNumbers f = **new** FindTwoMissingNumbers();

**int** [] results = f.missingNumbers(arrA, 10);

System.*out*.println("Missing numbers are :" + results[0] + " and " + results[1]);

}

}

**Output:**

Missing numbers are :6 and 4

**Algorithms – Find a peak element in a Given Array**

**Objective :** Find a peak element in a Given Array, where peak element is the one.

**Peak Element:** peak element is the element which is greater than or equal to both of its neighbors.

**Input:** Array, arrA[] .

**Output:** A peak element and its index

**Approach:**

A simple approach is to do a linear scan to a array and using few variables we can find a peak element. But the Time Complexity will be O(n) but real question is, Can we do better?

The Answer is yes, by using Binary Search techniques.

* If middle element is the peak element, return it
* If middle element is smaller than its left element , we will get our peak element on the left half
* If middle element is the smaller than its right element, we will our peak element on the right.

**Notes:**

1. If array has all the same elements, every element is a peak element.
2. Every array has a peak element.
3. Array might have has many peak elements but we are finding only one.
4. If array is in ascending or descending order then last element or the first element of the array will be the peak element respectively.

**Complete Code:**

**package** interviewQuestion;

//we will use binary search techniques

//if middle element is the peak element, return it

//if middle element is smaller than its left element , we will get our peak element on the left half

//if middle element is the smaller than its right element, we will our peak element on the right.

**public** **class** PeakElement {

**public** **int** peak(**int** [] arrA,**int** low, **int** high, **int** size){

**int** mid = (low+high)/2;

**if**((mid==0||arrA[mid]>=arrA[mid-1]) && (arrA[mid]>=arrA[mid+1]||mid==size-1)){

**return** mid;

}

**else** **if**(mid>0 && arrA[mid]<arrA[mid-1]) **return** peak(arrA,low,mid-1,size);

**else** **return** peak(arrA,mid+1,high,size);

}

**public** **static** **void** main(String args[]){

PeakElement pe = **new** PeakElement();

**int** arrA[] = { 1,2,3,4,0,1,5,4,3,2,1};

**int** peakEle = pe.peak(arrA, 0, arrA.length-1, arrA.length);

System.*out*.println("Peak Element is found at index [" + peakEle +"] = "+ arrA[peakEle]);

}

}

**Output:**

Peak Element is found at index [6] = 5

**Algorithms – Find Whether Given String is palindrome or Not.**

**Objective :** Find Whether Given String is palindrome or Not.

**Input:** A String,

**Output:** true or false on whether string is palindrome or not

**Approach:**

* Use recursive approach
* Compare first and last characters if they are not same- return false
* If they are same make, remove the first and last characters and make a recursive call.

**Example:**

Jain niaJ => compare ‘J’ with ‘J’ =>returns true

ain nia => compare ‘a’ with ‘a’ =>returns true

in ni => compare ‘i’ with ‘i’ =>returns true

n n => compare ‘n’ with ‘n’ =>returns true

string length <2 => returns true

**Complete Code:**

**package** interviewQuestion;

//Use recursive approach

//Compare first and last characters if they are not same- return false

//If they are same make, remove the first and last characters and make a recursive call.

**public** **class** Palindrome {

**public** Boolean isPalindrome(String strX){

**if**(strX.length()<2) **return** **true**;

**if**(strX.charAt(0)==strX.charAt(strX.length()-1)){

isPalindrome(strX.substring(0, strX.length()-2));

}

**else** **return** **false**;

**return** **true**;

}

**public** **static** **void** main(String args[]){

String S1 = "Sumit";

String S2 = "SumuS";

String S3 = "ABCDEFGHGFEDCBA";

String S4 = "Jain niaJ";

Palindrome p = **new** Palindrome();

System.*out*.println("Is "+ S1 + " Palindrome ??? :" + p.isPalindrome(S1));

System.*out*.println("Is "+ S2 + " Palindrome ??? :" + p.isPalindrome(S2));

System.*out*.println("Is "+ S3 + " Palindrome ??? :" + p.isPalindrome(S3));

System.*out*.println("Is "+ S4 + " Palindrome ??? :" + p.isPalindrome(S4));

}

}

**Output:**

Is Sumit Palindrome ??? :false

Is SumuS Palindrome ??? :true

Is ABCDEFGHGFEDCBA Palindrome ??? :true

Is Jain niaJ Palindrome ??? :true

**Algorithms – Find Whether Given the Sequence of parentheses are well formed.**

**Objective:** Find Whether Given the Sequence of parentheses are well formed.

**Input:** A String contains a sequence of parentheses

**Output:** true or false on whether parentheses are well formed or not

**Approach:**

* Idea is to have two counters, one for open parentheses '{' and one for close '}'
* Read one character at a time and increment one of the counters
* If any given point of time count of close parentheses is greater than the open one, return false
* If at the end both counters are equal, return true

**Example: { { } { } } – Well formed**

**{ { } { = Not well formed**

**Complete Code:**

**package** interviewQuestion;

**public** **class** WellFormedParentheses {

**public** Boolean isWellFormed(String strParentheses){

**if**(strParentheses==**null**){

**return** **false**;

}

//Idea is to have two counters, one for open parentheses '{' and one for close '}'

//Read one character at a time and increment one of the counters

//If any given point of time count of close parentheses is greater than the open one, return false

//If at the end both counters are equal, return true

**int** openParenCounter=0;

**int** closeParenCounter=0;

**for**(**int** i =0; i<strParentheses.length();i++){

**char** x = strParentheses.charAt(i);

**if**(x=='{') openParenCounter++;

**else** **if**(x=='}') closeParenCounter++;

**if**(closeParenCounter>openParenCounter){

**return** **false**;

}

}

**if**(openParenCounter==closeParenCounter)**return** **true**;

**else** **return** **false**;

}

**public** **static** **void** main(String args[]){

String x1 = "{{{{}}}}{}{}{}{}{}{}{}{}{}{}{{{}}}";

String x2 = "{{{{}}}}{}{}{}{{}{}{}{}{}{}{}{{{}}}";

String x3 = "{}{";

String x4 = "}{";

String x5 = "{{{{{{{{}}}}}}}}";

WellFormedParentheses w = **new** WellFormedParentheses();

System.*out*.println("Is "+ x1 + " well formed ??? :" + w.isWellFormed(x1));

System.*out*.println("Is "+ x2 + " well formed ??? :" + w.isWellFormed(x2));

System.*out*.println("Is "+ x3 + " well formed ??? :" + w.isWellFormed(x3));

System.*out*.println("Is "+ x4 + " well formed ??? :" + w.isWellFormed(x4));

System.*out*.println("Is "+ x5 + " well formed ??? :" + w.isWellFormed(x5));

}

}

**Output**

Is {{{{}}}}{}{}{}{}{}{}{}{}{}{}{{{}}} well formed ??? :true

Is {{{{}}}}{}{}{}{{}{}{}{}{}{}{}{{{}}} well formed ??? :false

Is {}{ well formed ??? :false

Is }{ well formed ??? :false

Is {{{{{{{{}}}}}}}} well formed ??? :true

**Algorithms – Rearrange Positive and Negative Numbers of Array On Each Side in O(nlogn)**

**Objective:** Rearrange Positive and Negative Numbers of an Array so that one side you have positive numbers and other side with negative Integers without changing their respective order.

Example : Input : 1 -2 3 -4 5 -6 7 -8 9 -10

ReArranged Output : -2 -4 -6 -8 -10 1 3 5 7 9

**Input:** An array with positive and negative numbers

**Output:** Modified array with positive numbers and negative numbers are on each side of the array.

**Approach:**

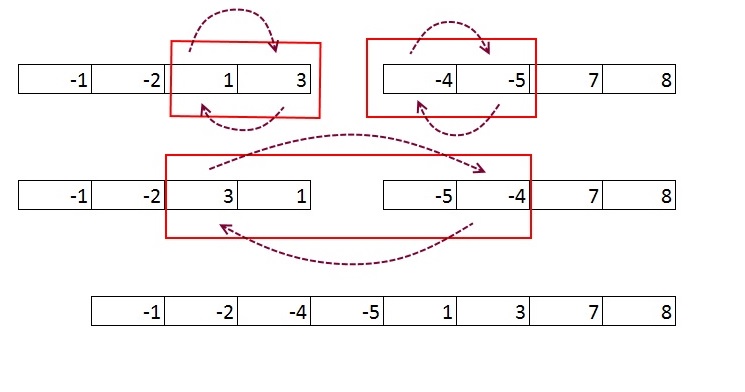
**Method 1.** One naive approach is to have another array of same size and navigate the input array and one scan place all the negative numbers to the new array and in second scan place all the positive numbers to new array. Here the Space complexity will be O(n). We have a better solution which can solve this in O(1) space.

**Method 2: Divide and Conquer**

* This approach is similar to merge sort.
* Divide the array into two half, Solve them individually and merge them.
* Tricky part is in merging.

Merging: (Negative on left side and positives on the right side)

* Navigate the left half of array till you won't find a positive integer and reverse the array after that point.(Say that part is called 'A')
* Navigate the right half of array till you won't find a negative integer and reverse the array after that point. (Say that part is called 'B')
* Now reverse the those parts from both the array (A and B), See example for better explanations



Complete Code:

**package** interviewQuestion;

**public** **class** RearrageArrayPositiveNegative {

**int** [] arrA;

**public** RearrageArrayPositiveNegative(**int** [] arrA){

**this**.arrA = arrA;

}

**public** **void** divideGroups(**int** low, **int** high){

**if**(low>=high) **return**;

**int** mid = (low+high)/2;

divideGroups(low, mid);

divideGroups(mid+1, high);

merge(low,mid,high);

}

**public** **void** merge(**int** low, **int** mid, **int** high){

**int** l = low;

**int** k = mid+1;

**while**(l<=mid && arrA[l]<=0)l++;

**while**(k<=high && arrA[k]<=0)k++;

reverse(l,mid);

reverse(mid+1,k-1);

reverse(l,k-1);

}

**public** **void** reverse(**int** x, **int** y){

**while**(y>x){

**int** temp = arrA[x];

arrA[x]=arrA[y];

arrA[y]=temp;

x++;

y--;

}

}

**public** **void** display(){

**for**(**int** i=0;i<arrA.length;i++){

System.*out*.print(" "+ arrA[i]);

}

}

**public** **static** **void** main(String args[]){

**int** [] a = {1,-2,3,-4,5,-6,7,-8,9,-10};

RearrageArrayPositiveNegative r = **new** RearrageArrayPositiveNegative(a);

System.*out*.print("Input : ");r.display();

r.divideGroups(0, a.length-1);

System.*out*.println("");

System.*out*.print("ReArranged Output : ");r.display();

}

}

Output:

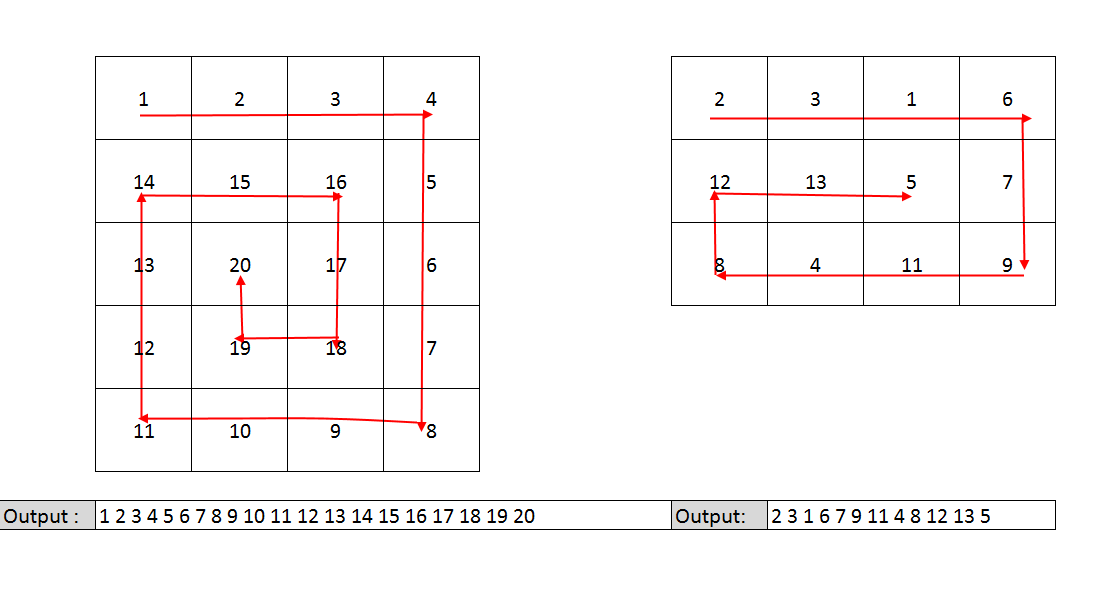
Input : 1 -2 -3 -4 5 -6 7 -8 9 -10 -11 -12 20

ReArranged Output : -2 -3 -4 -6 -8 -10 -11 -12 1 5 7 9 20

**Algorithms – Print All Elements of Two Dimensional Array in Spiral**

**Objective:** Print all the elements of two dimensional array in spiral.

Example :



**Input:** Two dimensional array

**Output:** All array elements printed in spiral.

**Approach:**

* Start printing from first row.
* Print row and columns, forward and backward alternatively
* With every iteration of (either row or column), reduce the size of an row or column by 1
* Call recursively

Complete Code:

**package** interviewQuestion;

**public** **class** Print2DArrayInSpiral {

**public** **int** arrA[][] = { { 1, 2, 3, 4, 5 }, { 18, 19, 20, 21, 6 },

{ 17, 28, 29, 22, 7 }, { 16, 27, 30, 23, 8 },

{ 15, 26, 25, 24, 9 }, { 14, 13, 12, 11, 10 } };

**public** **int** printSpiral(**int** row\_S, **int** row\_E, **int** col\_S, **int** col\_E,**boolean** reverse, **boolean** rowPrint) {

**if** (row\_S > row\_E && col\_S>col\_E) {

**return** 1;

}

**if** (rowPrint == **false**) {

**if** (reverse == **false**) {

**for** (**int** i = col\_S; i <= col\_E; i++) {

System.*out*.print(" " + arrA[row\_S][i]);

}

}

row\_S++;

rowPrint = **true**;

reverse = **false**;

}

**if** (rowPrint == **true**) {

**if** (reverse == **false**) {

**for** (**int** i = row\_S; i <= row\_E; i++) {

System.*out*.print(" " + arrA[i][col\_E]);

}

}

col\_E--;

rowPrint = **false**;

reverse = **true**;

}

**if** (rowPrint == **false**) {

**if** (reverse == **true**) {

**for** (**int** i = col\_E; i >= col\_S; i--) {

System.*out*.print(" " + arrA[row\_E][i]);

}

}

row\_E--;

rowPrint = **true**;

reverse = **true**;

}

**if** (rowPrint == **true**) {

**if** (reverse == **true**) {

**for** (**int** i = row\_E; i >= row\_S; i--) {

System.*out*.print(" " + arrA[i][col\_S]);

}

}

col\_S++;

rowPrint = **false**;

reverse = **false**;

}

printSpiral(row\_S, row\_E, col\_S, col\_E, reverse, rowPrint);

**return** 0;

}

**public** **static** **void** main(String args[]) {

Print2DArrayInSpiral p = **new** Print2DArrayInSpiral();

p.printSpiral(0, 5, 0, 4, **false**, **false**);

}

}

Output:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

**Algorithms – Find Out Whether String Contains All The Unique Characters**

**Objective:** find out whether in a given string contains all the unique characters

**Input:** A String

**Output:** True or false based upon whether string contains all the unique characters or not

**Approach:**

**Method 1.**

**When characters are not ASCII but could be anything alphabets or special characters**

* Create a boolean array of size 256, and put false at every index.
* Navigate the input string one character at a time, say 'char a'
* Check array position of array[a], if it is false, make it true.
* If it is false, update it as true.

**Method 2:**

Sort the array and do the linear scan to find out whether string contains unique elements or not.

**Complete Code:**

**package** interviewQuestion;

//This Program is to find out whether String contains all the unique characters

//With out using any additional data structures

**public** **class** UniqueCharString {

**private** String ip;

**public** UniqueCharString(String ip){

**this**.ip = ip;

}

//method 1 : When characters are not ASCII but could be anything alphabets or special characters

//Time Complexity : O(n)

//Space Complexity : O(1)

//

**public** Boolean UniChars(){

Boolean [] bln = **new** Boolean[256];

**for**(**int** i=0;i<256;i++){

bln[i]=**false**;

}

**for**(**int** i = 0;i<ip.length();i++){

**char** a = ip.charAt(i);

**if**(bln[a]==**true**){

**return** **false**;

}

**else**{

bln[a]=**true**;

}

}

**return** **true**;

}

//method 2: Sort the array and do the linear scan to find out whether string

//contains unique elements or not

//Time Complexity : O(nLogn)

//Space Complexity : O(n)

**public** Boolean UniqueCharSorting(){

**char** [] a = ip.toCharArray();

java.util.Arrays.*sort*(a);

String tmp = **new** String(a);

**for**(**int** i=1;i<tmp.length();i++){

**char** t = tmp.charAt(i-1);

**if**(t==tmp.charAt(i)){

**return** **false**;

}

}

**return** **true**;

}

**public** **static** **void** main(String args[]){

String a = "Sumit\_Jain";

UniqueCharString u = **new** UniqueCharString(a);

System.*out*.println("Method 1 : Does String ' " + a +" ' has all unique characters :" + u.UniChars());

a = "Sumit";

u = **new** UniqueCharString(a);

System.*out*.println("Method 1 : Does String ' " + a +" ' has all unique characters :" + u.UniChars());

a = "Sumit\_Jain";

u = **new** UniqueCharString(a);

System.*out*.println("Method 2 : Does String ' " + a +" ' has all unique characters :" + u.UniqueCharSorting());

a = "Sumit";

u = **new** UniqueCharString(a);

System.*out*.println("Method 2 : Does String ' " + a +" ' has all unique characters :" + u.UniqueCharSorting());

}

}

Output:

Method 1 : Does String ' Sumit\_Jain ' has all unique characters :false

Method 1 : Does String ' Sumit ' has all unique characters :true

Method 2 : Does String ' Sumit\_Jain ' has all unique characters :false

Method 2 : Does String ' Sumit ' has all unique characters :true

**Algorithms – Quick Sort Implementation**

**Objective:** To sort an array in increasing or decreasing order using Quick Sort.

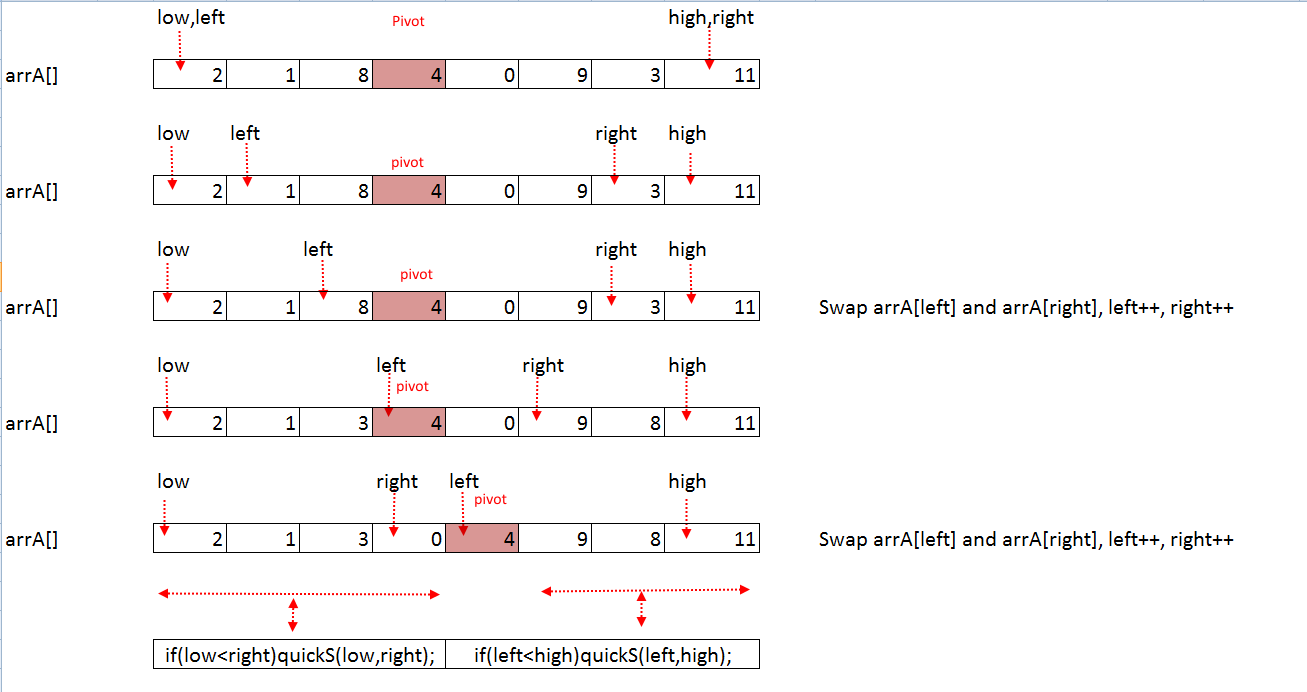
**Input:** An Array arrA[]

**Output:** A sorted array.

**Approach:**

* Choose any element from the array and call it as pivot element, Example here we have selected middle element as pivot
* Place all the elements smaller than pivot in the left side of pivot.
* Place all the elements greater than pivot in the right side of pivot.
* Sort left side and right side recursively.

Example:



**Complete Code:**

**package** interviewQuestion;

**public** **class** QuickSort

{

**private** **int** [] arrA;

**public** QuickSort(**int** [] arrA){

**this**.arrA = arrA;

}

**public** **void** quickS(**int** low, **int** high){

**int** mid = (low+high)/2;

**int** left = low;

**int** right = high;

**int** pivot = arrA[mid]; //select middle element as pivot

**while**(left<=right){

**while**(arrA[left]<pivot) left++;//find element which is greater than pivot

**while**(arrA[right]>pivot)right--;////find element which is smaller than pivot

//System.out.println(arrA[left] + " " + pivot + " " + arrA[right] );

//if we found the element on the left side which is greater than pivot

//and element on the right side which is smaller than pivot

//Swap them, and increase the left and right

**if**(left<=right){

**int** temp = arrA[left];

arrA[left] = arrA[right];

arrA[right]= temp;

left++;

right--;

}

}

//Recursion on left and right of the pivot

**if**(low<right)quickS(low,right);

**if**(left<high)quickS(left,high);

}

**public** **void** display(){

**for**(**int** i =0;i<arrA.length;i++){

System.*out*.print(" " + arrA[i] );

}

}

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

// your code goes here

**int** a [] = { 2,1,8,4,0,9,3,11};

QuickSort i = **new** QuickSort(a);

System.*out*.print("UnSorted : ");

i.display();

i.quickS(0,a.length-1);

System.*out*.println("");

System.*out*.print("Quick Sorted : ");

i.display();

}

}

Output:

UnSorted : 2 1 8 4 0 9 3 11

Quick Sorted : 0 1 2 3 4 8 9 11

Time Complexity : O(n^2) worst case and O(nlogn) average case analysis

Space Complexity : O(1)

**Algorithms – Given an array and a number k, check for pair in array with sum as k in O(nlgn)**

**Keywords :** **Given an array A[] and a number x, check for pair in A[] with sum as x, Given an array and a number k, check for pair in array with sum as k, Checking if there are 2 elements in an array that sum to X in O(n lg n),** **Find two numbers in an array whose sum is x,** **Find a pair of elements from an array whose sum equals a given number**

**Objective:** To find out whether in a given array there exists or not two numbers whose sum is exactly equals to a given number

**Input:** An array arrA[], A number k

**Output:** True or false based upon we have found any two numbers in array arrA[] whose sum is equal to k

**Approach:**

Method 1: Using Binary Search

* First sort the array using Merge Sort(To know about Merge Sort and its implementation Click Here)
* Now do the linear scan to the from the left array , say starting index i=0
* Calculate Rem\_Sum = number - arrA[i]
* If Rem\_Sum<0, move to the next element
* If Rem\_Sum>0, Perform Binary Search for Rem\_Sum on the remaining elements on the right side.

Time Complexity - O(nlogn)

**Method 2: Using Both Ends**

* First sort the array using Merge Sort(To know about Merge Sort and its implementation Click Here)
* Start from the both ends of the array
* Add first (say 'a') and last element(say 'b') of the array say S
* If S > number , S = S-(last\_element) + (element before that)
* If S < number , S = S - (first element) + (next element)
* If if S=number, return true
* Repeat step
* If iteration gets over and retrun false.

**Complete Code:**

**package** interviewQuestion;

**public** **class** TwoNumbersInArray {

**private** **int** [] arrA;

**private** **int** number;

**public** TwoNumbersInArray(**int** [] arrA,**int** number){

**this**.arrA = arrA;

**this**.number = number;

}

**public** Boolean usingBinarySearch(){

//1. First sort the array

MergeSort m = **new** MergeSort(arrA);

**int** [] arrSorted = m.mergeSorting();

BinarySearch bs = **new** BinarySearch(arrSorted);

//2. now do the linear scan to the from the left array , say starting index i=0

//3. Calculate Rem\_Sum = number - a[i]

//4. if Rem\_Sum<0, move to the next element

//5. if Rem\_Sum>0, Perform Binary Search on the remaining elements on the right side.

**for**(**int** i =0;i<arrA.length-1;i++){

**int** RemS = number - arrA[i];

**if**(RemS>0){

**if**(bs.Search(i+1, arrA.length-1, RemS)) **return** **true**;

}

}

**return** **false**;

}

**public** Boolean usingBothEnds(){

//1. First sort the array

MergeSort m = **new** MergeSort(arrA);

**int** [] arrSorted = m.mergeSorting();

//2. Start from the both ends of the array

//3. add first (say 'a') and last element(say 'b') of the array say S

//4. if S > number , S = S-(last\_element) + (element before that)

//5. if S < number , S = S - (first element) + (next element)

//6. if S=number, return true

//7. Repeat step

//8. If iteration gets over and retrun false.

**int** i =0;

**int** j = arrSorted.length-1;

**int** Sum = 0;

**while** (i!=j){

Sum = arrSorted[i]+ arrSorted[j];

**if**(Sum==number)**return** **true**;

**else** **if** (Sum<number)i++;

**else** **if** (Sum>number)j--;

}

**return** **false**;

}

**public** **static** **void** main(String[] args){

**int** a [] = { 1,2,3,4,5,16,17,18,19,249};

**int** number = 269;

**int** number1 = 8;

TwoNumbersInArray tn = **new** TwoNumbersInArray(a, number);

System.*out*.println("USING Both Ends -Sum of two numbers in array A is "+ number + " ??? :" + tn.usingBothEnds());

System.*out*.println("USING Binary Search -Sum of two numbers in array A is "+ number + " ??? :" + tn.usingBinarySearch());

TwoNumbersInArray tn1 = **new** TwoNumbersInArray(a, number1);

System.*out*.println("USING Both Ends -Sum of two numbers in array A is "+ number1 + " ??? :" + tn1.usingBothEnds());

System.*out*.println("USING Binary Search -Sum of two numbers in array A is "+ number1 + " ??? :" + tn1.usingBinarySearch());

}

}

Output:

USING Both Ends -Sum of two numbers in array A is 269 ??? :false

USING Binary Search -Sum of two numbers in array A is 269 ??? :false

USING Both Ends -Sum of two numbers in array A is 8 ??? :true

USING Binary Search -Sum of two numbers in array A is 8 ??? :true

**Algorithms - Find The Longest Sequence Of Prefix Shared By All The Words In A String**

**Objective:** Find The Longest Sequence Of Prefix Shared By All The Words In A String

**Input:** A String

**Output:** The longest sequence of prefix common in all the words in a string

**Example:**

"Bedroom BedClock BedTable BedWall" => "Bed"

**Approach:**

* Split the input by blank space and store it in arrA[].
* Create int resultLen and store the first index string length in it (int resultLen = arrA[0].length())
* Create another interger variable, int curr
* Now run a loop in rest of the array.
* Check if curr < resultLen and curr<length of current string in a loop
* If so check if character at curr position matched in first index string and with the current string a loop, if so, increase curr by 1
* Change resultLen = curr
* Return substring of resultLen length

**Complete Code:**

**package** interviewQuestion;

**public** **class** LongestPrefixSequence {

**private** String [] arrA;

**public** LongestPrefixSequence(String [] arrA){

**this**.arrA = arrA;

}

**public** String findPrefix(){

**int** resultLen = arrA[0].length();

**int** curr;

**for**(**int** i=1;i<arrA.length;i++){

curr=0;

**while**(curr<resultLen && curr<arrA[i].length() && arrA[0].charAt(curr)==arrA[1].charAt(curr)){

curr++;

}

resultLen = curr;

}

**return** arrA[0].substring(0,resultLen);

}

**public** **static** **void** main(String args[]){

String x = "Sumit Summation Summit Sum";

String [] arrA = x.split(" ");

LongestPrefixSequence lp = **new** LongestPrefixSequence(arrA);

System.*out*.println("Original String : " + x);

System.*out*.println("Common Prefix is : " + lp.findPrefix());

}

}

**Output:**

Original String : Sumit Summation Summit Sum

Common Prefix is : Sum

**Keyword:** Find The Longest Sequence Of Prefix Shared By All The Words In A String, The longest sequence of prefix common in all the words in a string, The longest substring of prefix common in all the words in a string**, Google interview, Common prefix in all strings, Common prefix in multiple strings**

**Algorithms - Singly Linked List Implementation**

Linked List- As the name suggests it's a list which is linked.

* Linked List consist of Nodes
* Nodes are nothing but objects of a class and each node has data and a link to the next node.

**class** Node {

**public** **int** data;

**public** Node next;

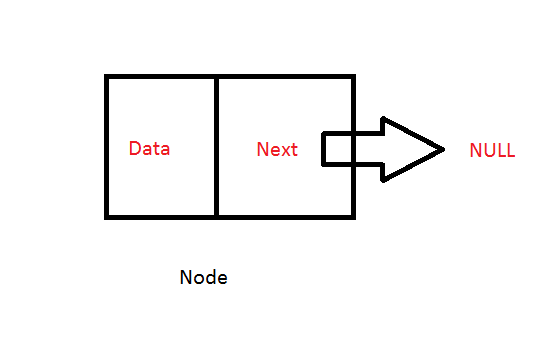
**public** Node(**int** data) {

**this**.data = data;

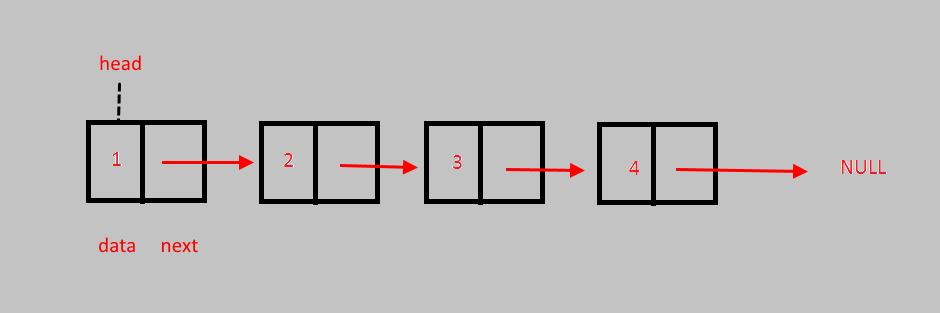
**this**.next = **null**;

}

}



* The last node in the list points to NULL , so when you reach there you will know that the list ends here.



**Operations:**

**Add at the Start :** Add a node the beginning of the linked list. Its O(1).

**Add at the End :** Add a node at the end of the linked list. its O(n) since to add a node at the end you need to go till the end of the array.

**Delete at the Start :** Delete a node from beginning of the linked list. Its O(1).

**Delete at the End :** Delete a node from the end of the linked list. its O(n) since to delete a node at the end you need to go till the end of the array.

**Get Size:** returns the size of the linked list.

**Get Element at Index :** Return the element at specific index, if index is greater than the size then return -1. its O(n) in worst case.

**Add Element at Specific Index :** Add element at specific index. If index is greater than size then print "INVALID POSITION". Worst case its O(n)

Display(): Prints the entire linked list. O(n).

**Complete Code:**

**package** interviewQuestion;

**public** **class** LinkListImplementation {

**public** **static** **void** main(String[] args) **throws** java.lang.Exception {

LinkedListT a = **new** LinkedListT();

a.addAtBegin(5);

a.addAtBegin(15);

a.addAtEnd(20);

a.addAtEnd(21);

a.deleteAtBegin();

a.deleteAtEnd();

a.addAtIndex(10, 2);

a.addAtEnd(15);

a.display();

System.*out*.println("\n Size of the list is: " + a.size);

System.*out*.println(" Element at 2nd position : " + a.elementAt(2));

System.*out*.println(" Searching element 20, location : " + a.search(15));

}

}

**class** Node {

**public** **int** data;

**public** Node next;

**public** Node(**int** data) {

**this**.data = data;

**this**.next = **null**;

}

}

**class** LinkedListT {

**public** Node head;

**public** **int** size;

**public** LinkedListT() {

head = **null**;

}

**public** **void** addAtBegin(**int** data) {

Node n = **new** Node(data);

n.next = head;

head = n;

size++;

}

**public** **int** deleteAtBegin() {

**int** tmp = head.data;

head = head.next;

size--;

**return** tmp;

}

**public** **void** deleteAtEnd() {

Node currNode = head;

**if** (head.next == **null**) {

head = **null**;

} **else** {

**while** (currNode.next.next != **null**) {

currNode = currNode.next;

}

**int** temp = currNode.next.data;

currNode.next = **null**;

size--;

}

}

**public** **void** addAtEnd(**int** data) {

**if** (head == **null**) {

addAtBegin(data);

} **else** {

Node n = **new** Node(data);

Node currNode = head;

**while** (currNode.next != **null**) {

currNode = currNode.next;

}

currNode.next = n;

size++;

}

}

**public** **int** elementAt(**int** index){

**if**(index>size){

**return** -1;

}

Node n = head;

**while**(index-1!=0){

n=n.next;

index--;

}

**return** n.data;

}

**public** **int** getSize(){

**return** size;

}

**public** **int** search(**int** data){

Node n = head;

**int** count = 1;

**while**(n!=**null**){

**if**(n.data==data){

**return** count;

}**else**{

n = n.next;

count++;

}

}

**return** -1;

}

**public** **void** addAtIndex(**int** data, **int** position){

**if**(position == 1){

addAtBegin(data);

}

**int** len = size;

**if** (position>len+1 || position <1){

System.*out*.println("\nINVALID POSITION");

}

**if**(position==len+1){

addAtEnd(data);

}

**if**(position<=len && position >1){

Node n = **new** Node(data);

Node currNode = head; //so index is already 1

**while**((position-2)>0){

System.*out*.println(currNode.data);

currNode=currNode.next;

position--;

}

n.next = currNode.next;

currNode.next = n;

size++;

}

}

**public** **void** display() {

System.*out*.println("");

Node currNode = head;

**while** (currNode != **null**) {

System.*out*.print("->" + currNode.data);

currNode = currNode.next;

}

}

}

**Output:**

->5->10->20->15

Size of the list is: 4

Element at 2nd position : 10

Searching element 20, location : 4

**Algorithms - Merge or Combine Two Sorted Linked Lists**

**Objective:** Given two sorted linked lists, objective is to merge both the lists in sorted order.

**Input:** Two sorted linked list List a, List b.

**Example:**

**List a :** ->2->6->18

**List b:** ->1->3->17->19

**Merged List: ->1->2->3->6->17->18->19**

**Approach:**

**Without Recursion:**

* Create a new node say result
* Navigate through both the linked lists at the same time, starting from head
* Compare the first node values of both the linked lists
* Whichever is smaller, add it to the result node
* Move the head pointer of the linked list whose value was smaller
* Again compare the node values
* Keep doing until one list gets over
* Copy the rest of the nodes of unfinished list to the result

**With Recursion:**

* Base Case :

If List A gets finished , return List B.

If List B gets finished, return List A.

* Create a result node and initialize it as NULL
* Check which node (List A or List B) has a smaller value.
* Whichever is smaller, add it to the result node.
* Make recursive call and add the return node as result.next

result.next = recurrsionMerge(nA.next, nB)

**Complete Code:**

**package** interviewQuestion;

//WithOut Recursion

//create a new node say result

//navigate through both the linked lists at the same time, starting from head

//compare the first node values of both the linked lists

//which ever is smaller, add it to the result node

//move the head pointer of the linked list whose value was smaller

//again compare the node values

//keep doing until one list gets over

//copy the rest of the nodes of unfinished list to the result

**public** **class** MergeTwoLinkList {

**private** LinkedListT a;

**private** LinkedListT b;

**public** MergeTwoLinkList(LinkedListT a, LinkedListT b){

**this**.a=a;

**this**.b=b;

}

**public** LinkedListT mergeWithOutRecur(){

LinkedListT result = **new** LinkedListT();

**while**(a.head!=**null** && b.head!=**null**){

// System.out.println(a.head.data + " " + b.head.data);

**if**(a.head.data<b.head.data){

result.addAtEnd(a.head.data);

a.head = a.head.next;

}

**else**{

result.addAtEnd(b.head.data);

b.head = b.head.next;

}

}

**while**(a.head!=**null**){

result.addAtEnd(a.head.data);

a.head = a.head.next;

}

**while**(b.head!=**null**){

result.addAtEnd(b.head.data);

b.head = b.head.next;

}

**return** result;

}

**public** Node recurrsionMerge(Node nA, Node nB){

//base case

Node result = **null**;

**if**(nA==**null**) **return** nB;

**else** **if**(nB==**null**) **return** nA;

**if**(nA.data<nB.data){//Check which node has a smaller value

result = nA; //add it to the result node

result.next = recurrsionMerge(nA.next, nB);//Recursive call and add the return node as result.next

}

**else**{

result = nB; //add it to the result node

result.next = recurrsionMerge(nA, nB.next); //Recursive call and add the return node as result.next

}

**return** result;

}

**public** **void** display(Node head) {

System.*out*.println("");

Node currNode = head;

**while** (currNode != **null**) {

System.*out*.print("->" + currNode.data);

currNode = currNode.next;

}

System.*out*.println("");

}

**public** **static** **void** main(String [] args){

System.*out*.println("Method : with Recursion");

LinkedListT a = **new** LinkedListT();

a.addAtBegin(8);a.addAtBegin(6);a.addAtBegin(5);

LinkedListT b = **new** LinkedListT();

b.addAtBegin(9);b.addAtBegin(7);b.addAtBegin(3);b.addAtBegin(1);

MergeTwoLinkList m = **new** MergeTwoLinkList(a, b);

m.display(a.head);m.display(b.head);

Node result;

result = m.recurrsionMerge(a.head, b.head);

m.display(result);

//method 2

System.*out*.println("Method : without Recursion");

LinkedListT a1 = **new** LinkedListT();

a1.addAtBegin(18);a1.addAtBegin(6);a1.addAtBegin(2);

LinkedListT b1 = **new** LinkedListT();

b1.addAtBegin(19);b1.addAtBegin(17);b1.addAtBegin(3);b1.addAtBegin(1);

MergeTwoLinkList m1 = **new** MergeTwoLinkList(a1, b1);

m1.display(a1.head);m1.display(b1.head);

LinkedListT res = m1.mergeWithOutRecur();

m1.display(res.head);

}

}

**Output:**

Method : with Recursion

->5->6->8

->1->3->7->9

->1->3->5->6->7->8->9

Method : without Recursion

->2->6->18

->1->3->17->19

->1->2->3->6->17->18->19

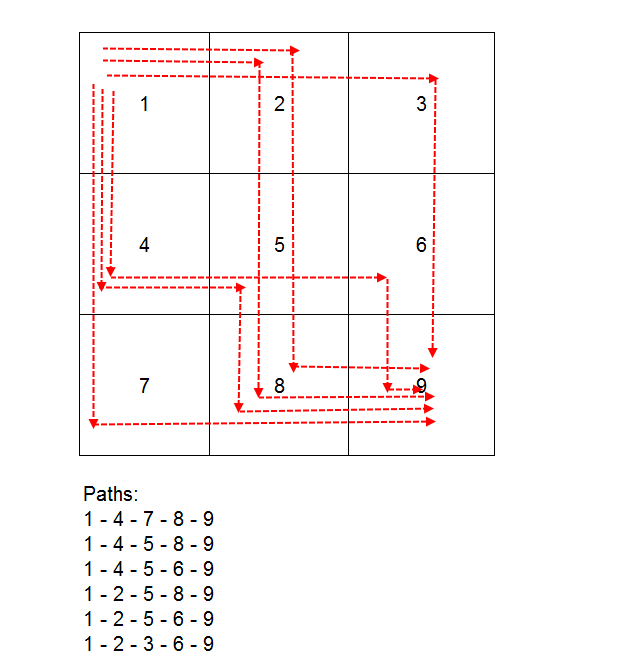
**Algorithms - Print All Paths from Top left to bottom right in Two Dimensional Array**

**Objective:** Print all the paths from left top corner to right bottom corner in two dimensional array.

**Input:** Two Dimensional array

**Output:** Print all the paths.

**Example:**

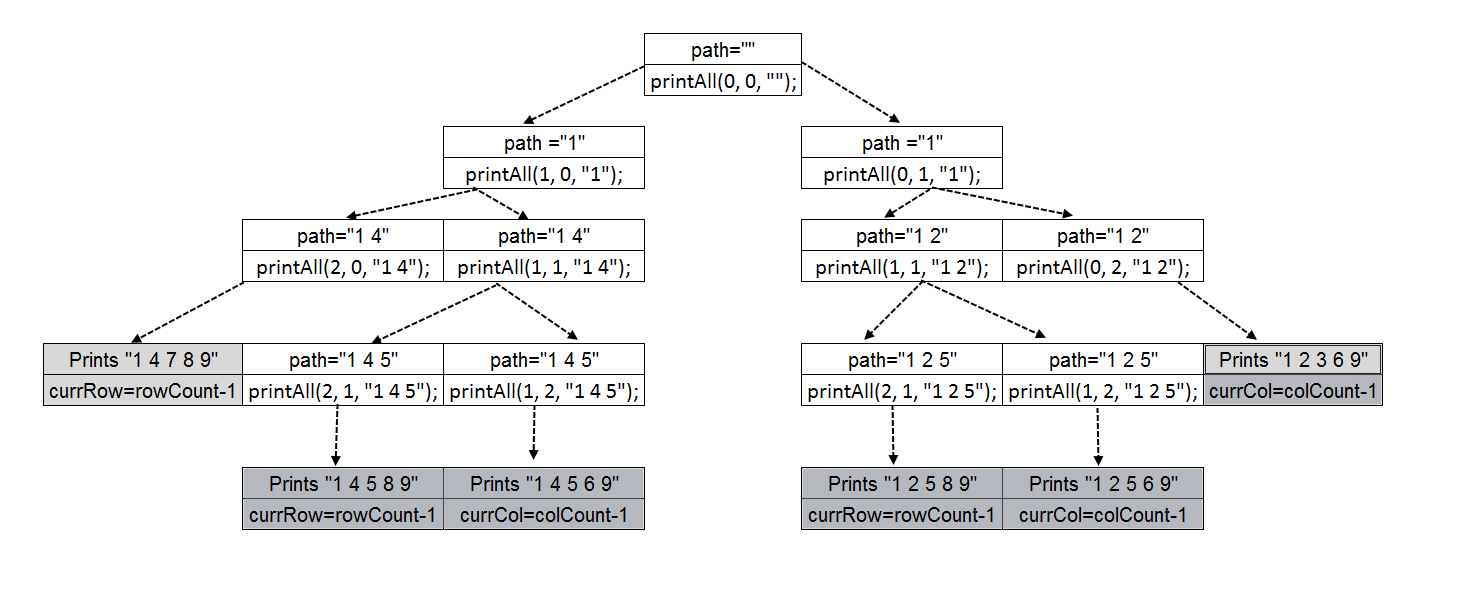


**Approach:**

**As we need to explore all the paths from top left corner to bottom right corner, we will either travel down OR travel right. so every time either we increase the row or column.**

* Recursion is the key here.
* Take the rows count and column counts say rowCount and colCount respectively
* Take currentRow =0 and currentColumn =0 and path =""
* Call printAll(currentRow, currentcolumn,path)
* Add array[0][0] to the path.
* Call recursively printAll(currentRow+1, currentcolumn,path)
* Call recursively printAll(currentRow, currentcolumn+1,path)
* **Base Case 1:** when currentRow = rowCount-1(because array index starts with 0) , print the rest of the columns, return
* **Base Case 2:** when currentcolumn = colCount-1(because array index starts with 0) , print the rest of the rows, return

**Like always you need to trust Recursion to get you the correct result. :)**

****

**Complete Code:**

**package** interviewQuestion;

**public** **class** PrintAllPathIn2DArray {

**int** rowCount;

**int** colCount;

**int** [][] arrA;

**public** PrintAllPathIn2DArray(**int** arrA[][]){

**this**.arrA = arrA;

rowCount=arrA[0].length;

colCount = arrA[1].length;

}

**public** **void** printAll(**int** currentRow, **int** currentColumn, String path){

**if**(currentRow==rowCount-1){

**for**(**int** i=currentColumn;i<colCount;i++){

path += "-" + arrA[currentRow][i];

}

System.*out*.println(path);

**return**;

}

**if**(currentColumn==colCount-1){

**for**(**int** i=currentRow;i<=rowCount-1;i++){

path += "-" + arrA[i][currentColumn];

}

System.*out*.println(path);

**return**;

}

path = path +"-"+ arrA[currentRow][currentColumn];

printAll(currentRow+1,currentColumn,path);

printAll(currentRow,currentColumn+1,path);

}

**public** **static** **void** main(String args[]){

**int** [][]a = {{1,2,3},{4,5,6},{7,8,9}};

PrintAllPathIn2DArray p = **new** PrintAllPathIn2DArray(a);

p.printAll(0, 0, "");

}

}

**Output:**

-1-4-7-8-9

-1-4-5-8-9

-1-4-5-6-9

-1-2-5-8-9

-1-2-5-6-9

-1-2-3-6-9

**Algorithms - Count All Paths from Top left to bottom right in Two Dimensional Array including Diagonal Paths**

**Objective:** Count all the paths from left top corner to right bottom corner in two dimensional array.

**Input:** Two Dimensional array

**Output:** No of paths.

**Approach :**

**1. Recursive**

Recursive solution to this problem is similar to "link to print all paths"

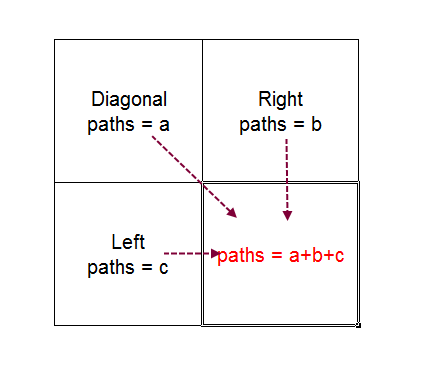
But the Time complexity will be exponential because there will be many sub problems which will be solved again and again to get the final solution. read this : "Dynamic programming vs Recursion

**2. Dynamic Programming (Better Solution)**

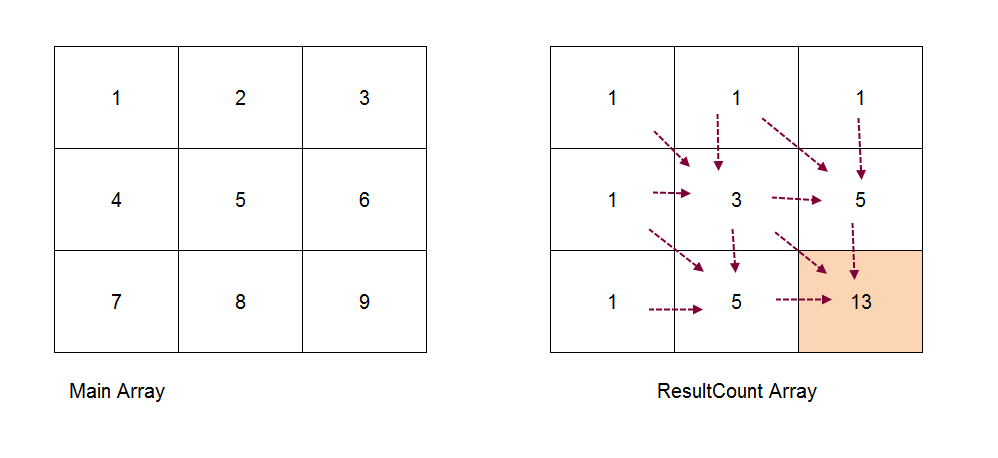
**Create two dimensional resultCount array to store the number of paths from top left corner.**

**Base Case:** To reach to any cell in either first row or column from first cell(top left at 0,0) will be 1.

You can reach to any cell from 3 different ways, from left, from top, from diagonal. So total no of paths to reach to that cell will be sum of all the paths to reach to left, top and diagonal cell. see picture



**Example:**



**Complete Code:**

**package** newInterviewQuestions;

**public** **class** CountAllPaths {

**int** rowCount;

**int** colCount;

**int** [][] arrA;

**public** CountAllPaths(**int** arrA[][]){

**this**.arrA = arrA;

rowCount=arrA.length;

colCount = arrA[0].length;

}

**public** **int** countAllResursion(**int** currentRow, **int** currentColumn){

**if**(currentRow==rowCount-1){

**return** 1;

}

**if**(currentColumn==colCount-1){

**return** 1;

}

**return** countAllResursion(currentRow+1,currentColumn) + countAllResursion(currentRow,currentColumn+1)+ countAllResursion(currentRow+1,currentColumn+1);

}

**public** **int** printAllDynamic(**int** [][] arrA){

**int** [][]resultCount = **new** **int** [arrA.length][arrA[0].length];

**for**(**int** i=0;i<arrA.length;i++){

resultCount[i][0]=1;

}

**for**(**int** i=0;i<arrA[1].length;i++){

resultCount[0][i]=1;

}

**for**(**int** i=1;i<arrA.length;i++){

**for**(**int** j=1;j<arrA[1].length;j++){

resultCount[i][j]= resultCount[i][j-1] + resultCount[i-1][j] + resultCount[i-1][j-1];

}

}

**return** resultCount[arrA.length-1][arrA[0].length-1];

}

**public** **static** **void** main(String args[]){

**int** [][]a = {{1,2,3},{4,5,6},{7,8,9}};

CountAllPaths p = **new** CountAllPaths(a);

System.*out*.println("No of Paths By Recursion: " +p.countAllResursion(0, 0));

System.*out*.println("No of paths By Dynamic Programming: " +p.printAllDynamic(a));

}

}

**Output:**

No of Paths By Recursion: 13

No of paths By Dynamic Programming: 13

**Algorithms - Reverse a Linked List**

**Objective:** Reverse the given linked list.

**Input:** A Linked List

**Output:** Reversed Linked List

**Example:**

Input : ->30->25->20->15->10->5

Reversed : ->5->10->15->20->25->30

**Approach:**

**Iterative:**

* Create 3 nodes, currNode, PrevNode and nextNode.
* Initialize them as currNode = head;,nextNode = **null**;prevNode = **null**;
* Now keep reversing the pointers one by one till currNode!=null.

**while**(currNode!=**null**){

nextNode = currNode.next;

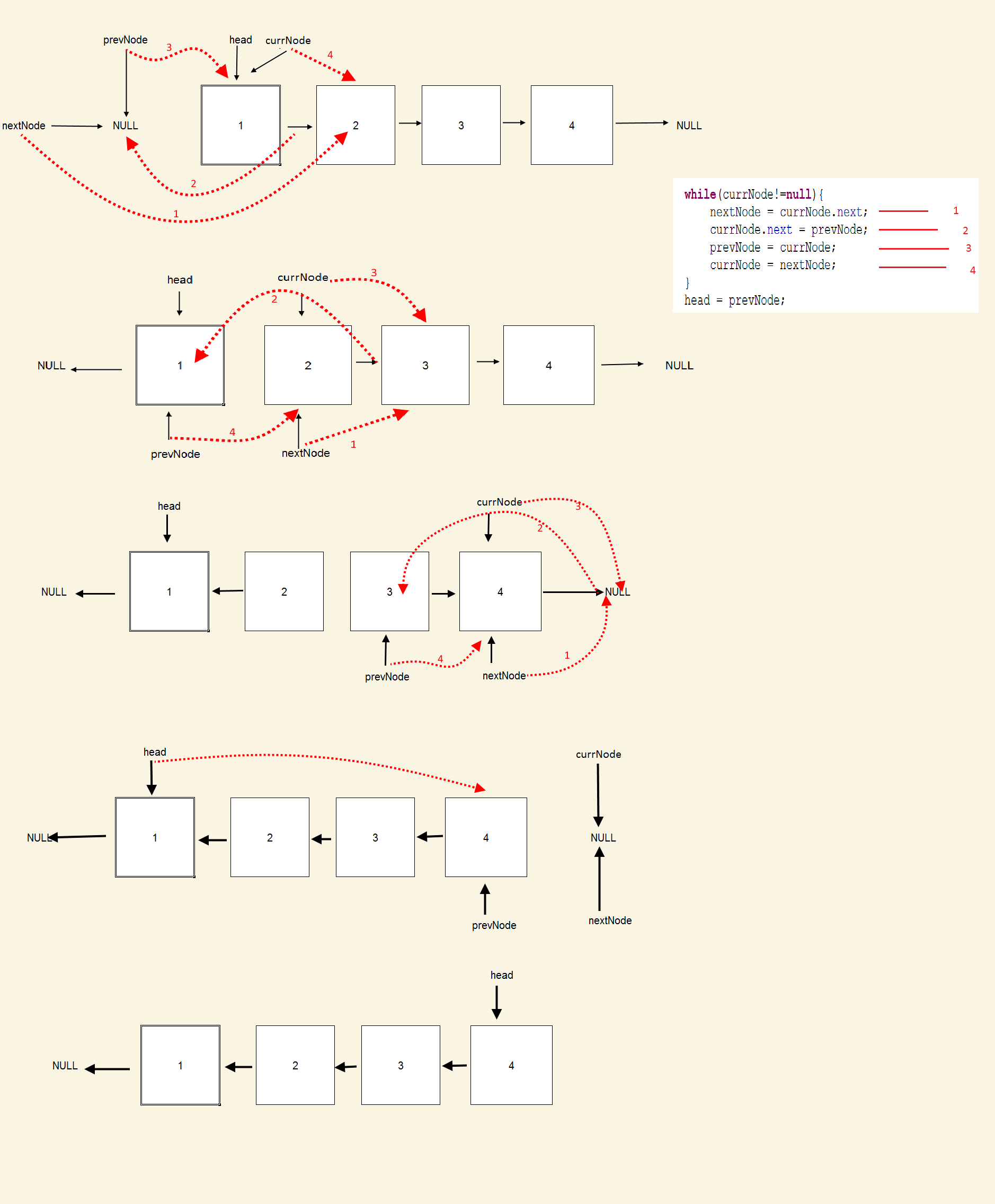
currNode.next = prevNode;

prevNode = currNode;

currNode = nextNode;

}

* At the end set head = prevNode;
* See Example:



**Recursive Approach:**

* Take 3 nodes as Node ptrOne,Node ptrTwo, Node prevNode
* Initialize them as ptrOne = head; ptrTwo=head.next, prevNode = null.
* Call reverseRecursion(head,head.next,null)
* Reverse the ptrOne and ptrTwo
* Make a recursive call for reverseRecursion(ptrOne.next,ptrTwo.next,null)

Complete Code:

**package** reverseALinkedList;

**public** **class** ReverseLinkedList {

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

LinkedListT a = **new** LinkedListT();

a.addAtBegin(5);

a.addAtBegin(10);

a.addAtBegin(15);

a.addAtBegin(20);

a.addAtBegin(25);

a.addAtBegin(30);

// System.out.print("Original Link List 1 : ");

a.display(a.head);

a.reverseIterative(a.head);

LinkedListT b = **new** LinkedListT();

b.addAtBegin(31);

b.addAtBegin(32);

b.addAtBegin(33);

b.addAtBegin(34);

b.addAtBegin(35);

b.addAtBegin(36);

System.*out*.println("");

System.*out*.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

System.*out*.print("Original Link List 2 : ");

b.display(b.head);

b.reverseRecursion(b.head,b.head.next,**null**);

}

}

**class** Node{

**public** **int** data;

**public** Node next;

**public** Node(**int** data){

**this**.data = data;

**this**.next = **null**;

}

}

**class** LinkedListT{

**public** Node head;

**public** LinkedListT(){

head=**null**;

}

**public** **void** addAtBegin(**int** data){

Node n = **new** Node(data);

n.next = head;

head = n;

}

**public** **void** reverseIterative(Node head){

Node currNode = head;

Node nextNode = **null**;

Node prevNode = **null**;

**while**(currNode!=**null**){

nextNode = currNode.next;

currNode.next = prevNode;

prevNode = currNode;

currNode = nextNode;

}

head = prevNode;

System.*out*.println("\n Reverse Through Iteration");

display(head);

}

**public** **void** reverseRecursion(Node ptrOne,Node ptrTwo, Node prevNode){

**if**(ptrTwo!=**null**){

**if**(ptrTwo.next!=**null**){

Node t1 = ptrTwo;

Node t2 = ptrTwo.next;

ptrOne.next = prevNode;

prevNode = ptrOne;

reverseRecursion(t1,t2, prevNode);

}

**else**{

ptrTwo.next = ptrOne;

ptrOne.next = prevNode;

System.*out*.println("\n Reverse Through Recursion");

display(ptrTwo);

}

}

**else** **if**(ptrOne!=**null**){

System.*out*.println("\n Reverse Through Recursion");

display(ptrOne);

}

}

**public** **void** display(Node head){

//

Node currNode = head;

**while**(currNode!=**null**){

System.*out*.print("->" + currNode.data);

currNode=currNode.next;

}

}

}

Output:

->30->25->20->15->10->5

Reverse Through Iteration

->5->10->15->20->25->30

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Original Link List 2 : ->36->35->34->33->32->31

Reverse Through Recursion

->31->32->33->34->35->36

**Algorithms - Find the Loop in a Linked list, find its length and Break the Loop**

**Objective:** In a given linked list, check whether it contains the loop in it, if yes then find the Loop length and break the loop.

Loop in a linked list means the last node does not point to the null, instead it points to some node in the list.

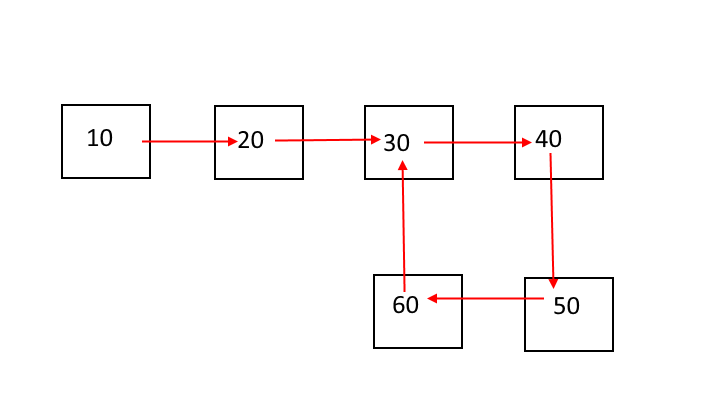
**Input:** A Linked List

**Output:** Linked list contains loop or not, if yes its length and linked list after breaking the loop.

**Example:**

Input : ->10->20->30->40->50->60->30->40->50->60->30->40->50->60->30

here you can see that 30->40->50->60 are repeating ,that means it has a loop



**Approach:**

* **Find the Loop**
* **Take two pointers, both starts from head**
* Move one pointer with normal speed and another with double speed
* If both pointers meets at some point, we have found the loop
* Now find the loop length
* At the point where both pointers have met, stop one pointer and keep moving the nother one
* When another pointer meets the first pointer, stop.
* Keep counting number of hops, that will your loop length
* Now To break the loop
* Move one pointer by the loop length
* Now move both pointers with normal speed.
* When secondpointer.next = first pointer, set secondpinter.next=null.

**Complete Code:**

**package** findLoopInLinkedList;

//Find the Loop

//Take two pointers, both starts from head

//move one pointer with normal speed and another with double speed

//if both pointers meets at some point, we have found the loop

//Now find the loop length

//at the point where both pointers have met, stop one pointer and keep moving the nother one

//when another pointer meets the first pointer, stop.

//keep counting number of hops, that will your loop length

//Now To break the loop

//move one pointer by the loop length

//now move both pointers with normal speed.

//when secondpointer.next = first pointer, set secondpinter.next=null.

**public** **class** FindLoopInLinkedList

{ **public** LinkedListLoop a;

**public** **int** loopLength;

**public** **void** createLoop(){

a = **new** LinkedListLoop();

a.addAtEnd(10);

a.addAtEnd(20);

a.addAtEnd(30);

a.addAtEnd(40);

a.addAtEnd(50);

a.addAtEnd(60);

a.insertLoop(2);

a.displayLoop();

}

**public** **void** findLoop(){

Node ptrOne =a.head;

Node ptrTwo =a.head.next.next;

**while**(ptrOne!=**null**){

**if**(ptrOne!=ptrTwo){

ptrOne = ptrOne.next;

ptrTwo = ptrTwo.next.next;

}

**else**{

System.*out*.println("");

System.*out*.println("Loop Found--starts at " + ptrOne.data);

findLoopLength(ptrOne, ptrTwo);

breakLoop(ptrOne, ptrTwo);

**break**;

}

}

}

**public** **void** findLoopLength(Node one, Node two){

one = one.next;

loopLength = 1;

**while**(one!=two){

one = one.next;

loopLength++;

}

System.*out*.println("Loop length is " + loopLength);

}

**public** **void** breakLoop(Node one, Node two){

one = one.next;

**while**(one.next!=two){

one = one.next;

}

one.next = **null**;

System.*out*.println("Loop breaks");

a.display();

}

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

FindLoopInLinkedList i = **new** FindLoopInLinkedList();

i.createLoop();

i.findLoop();

}

}

**class** Node{

**public** **int** data;

**public** Node next;

**public** Node(**int** data){

**this**.data = data;

**this**.next = **null**;

}

}

**class** LinkedListLoop{

**public** Node head;

**public** LinkedListLoop(){

head=**null**;

}

**public** **void** addAtEnd(**int** data){

Node n = **new** Node(data);

**if** (head==**null**){

n.next = head;

head = n;

}

**else**{

Node currNode = head;

**while**(currNode.next!=**null**){

//System.out.print("---->" + currNode.data);

currNode = currNode.next;

}

currNode.next = n;

}

}

**public** **void** insertLoop(**int** index){

Node endNode = head;

**while**(endNode.next!=**null**){

//System.out.print("---->" + currNode.data);

endNode = endNode.next;

}

Node indexNode = head;

**while**(index!=0){

indexNode = indexNode.next;

index--;

}

endNode.next = indexNode;

}

**public** **void** displayLoop(){

System.*out*.println("");

Node currNode = head;

**int** cnt = 15;

**while**(cnt!=0){

System.*out*.print("->" + currNode.data);

currNode=currNode.next;

cnt--;

}

}

**public** **void** display(){

System.*out*.println("");

Node currNode = head;

**while**(currNode!=**null**){

System.*out*.print("->" + currNode.data);

currNode=currNode.next;

}

}

}

Output:

->10->20->30->40->50->60->30->40->50->60->30->40->50->60->30

Loop Found--starts at 30

Loop length is 4

Loop breaks

->10->20->30->40->50->60

**Algorithms - Find Intersection Point in Two Linked List**

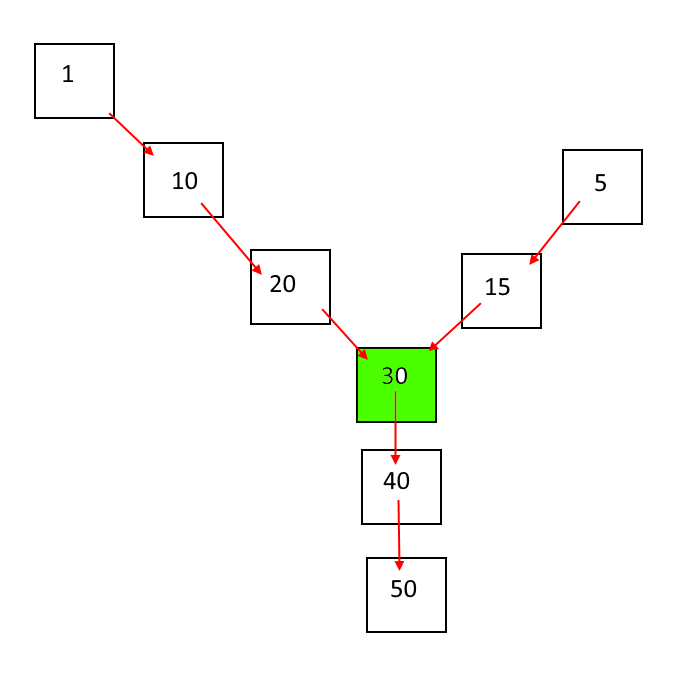
**Objective:** Given Two linked list, check whether both list intersect each other, if yes then find the starting node of the intersection.

Intersection point means end of one linked list is linked with some node in another linked list and it forms a Y shape.

**Input:** Two Linked List

**Output: Intersection Node or point**

**Example:**

****

**Approach:**

* Find the length of both the linked lists say : a\_len and b\_len
* Find the lenDiff = (a\_len ~ b\_len)
* Traverse the longer linked list by lenDiff
* Now traverse both the lists at the same time
* Check whether nodes are same, if yes then we have found the intersection point
* If we reach the end of the link lists then there is no intersection point.

**Trick Solution:**

* Take one linked list and join it both ends.
* Nor for the second Linked List, the problem is reduced to "Find a loop in a linked list and find the starting point of the linked list. So see inplementation, Click here.
* Starting point will be our intersection point.

**Complete Code:**

**package** findIntersectionOfLinkedLists;

//find the length of both the linked lists say : a\_len and b\_len

//find the lenDiff = (a\_len ~ b\_len)

//traverse the longer linked list by lenDiff

//Now traverse both the lists at the same time

//check whether nodes are same, if yes then we have found the intersection point

//if we reach the end of the link lists then there is no intersection point.

**public** **class** FindIntersectionOfLinkedLists {

**public** LinkedListIntersection a;

**public** LinkedListIntersection b;

**public** **void** createLists(){

a = **new** LinkedListIntersection();

a.addAtEnd(1);

a.addAtEnd(10);

a.addAtEnd(20);

Node tmp = a.addAtEnd(30);

a.addAtEnd(40);

a.addAtEnd(50);

a.addAtEnd(60);

System.*out*.print("List A : ");

a.display();

b = **new** LinkedListIntersection();

b.addAtEnd(5);

b.addAtEnd(15);

b.createIntersection(a,tmp);

System.*out*.print("List B : ");

b.display();

}

**public** **void** findIntersectionByLength(){

**int** a\_len=0;

**int** b\_len=0;

**int** lenDiff=0;

**boolean** intsctFound = **false**;

Node an = a.head;

Node bn = b.head;

**while**(an!=**null**){

an=an.next;

a\_len++;

}

**while**(bn!=**null**){

bn=bn.next;

b\_len++;

}

an = a.head;

bn = b.head;

**if**(a\_len>b\_len){

lenDiff = a\_len-b\_len;

// System.out.print("length diff " +lenDiff );

**while**(lenDiff!=0){

an = an.next;

lenDiff--;

}

}**else**{

lenDiff = b\_len-a\_len;

**while**(lenDiff!=0){

bn = bn.next;

lenDiff--;

}

}

**while**(an!=**null** && bn!=**null**){

//System.out.print(an.data + " " + bn.data);

**if**(an==bn) {

System.*out*.print("Intersection found at " + an.data);

intsctFound = **true**;

**break**;

}

**else**{

an = an.next;

bn = bn.next;

}

}

**if**(intsctFound!=**true**){

System.*out*.print("Intersection Not Found");

}

}

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

FindIntersectionOfLinkedLists i = **new** FindIntersectionOfLinkedLists();

i.createLists();

i.findIntersectionByLength();

}

}

**class** Node{

**public** **int** data;

**public** Node next;

**public** Node(**int** data){

**this**.data = data;

**this**.next = **null**;

}

}

**class** LinkedListIntersection{

**public** Node head;

**public** LinkedListIntersection(){

head=**null**;

}

**public** Node addAtEnd(**int** data){

Node n = **new** Node(data);

**if** (head==**null**){

n.next = head;

head = n;

}

**else**{

Node currNode = head;

**while**(currNode.next!=**null**){

//System.out.print("---->" + currNode.data);

currNode = currNode.next;

}

currNode.next = n;

}

**return** n;

}

**public** **void** createIntersection(LinkedListIntersection a, Node nd){

Node hd = a.head; // this is the list to whcih another list will intersect, in our example its list a

**while**(hd!=nd){

hd = hd.next;

}

Node currNode = head;// this is for the list which will connect, in our example its list b

**while**(currNode.next!=**null**){

currNode = currNode.next;

}

currNode.next = hd; ;

}

**public** **void** display(){

System.*out*.println("");

Node currNode = head;

**while**(currNode!=**null**){

System.*out*.print("->" + currNode.data);

currNode=currNode.next;

}

System.*out*.println("");

}

}

**Output:**

List A :

->1->10->20->30->40->50->60

List B :

->5->15->30->40->50->60

Intersection found at 30

**Algorithms - Swap Every Kth Node in a LinkedList**

**Objective:** Given a linked list, swap every kth node in that. If at the end of the list remaining nodes are less than k, leave them untouched.

**Input:** A linked list, A number k.

**Example:**

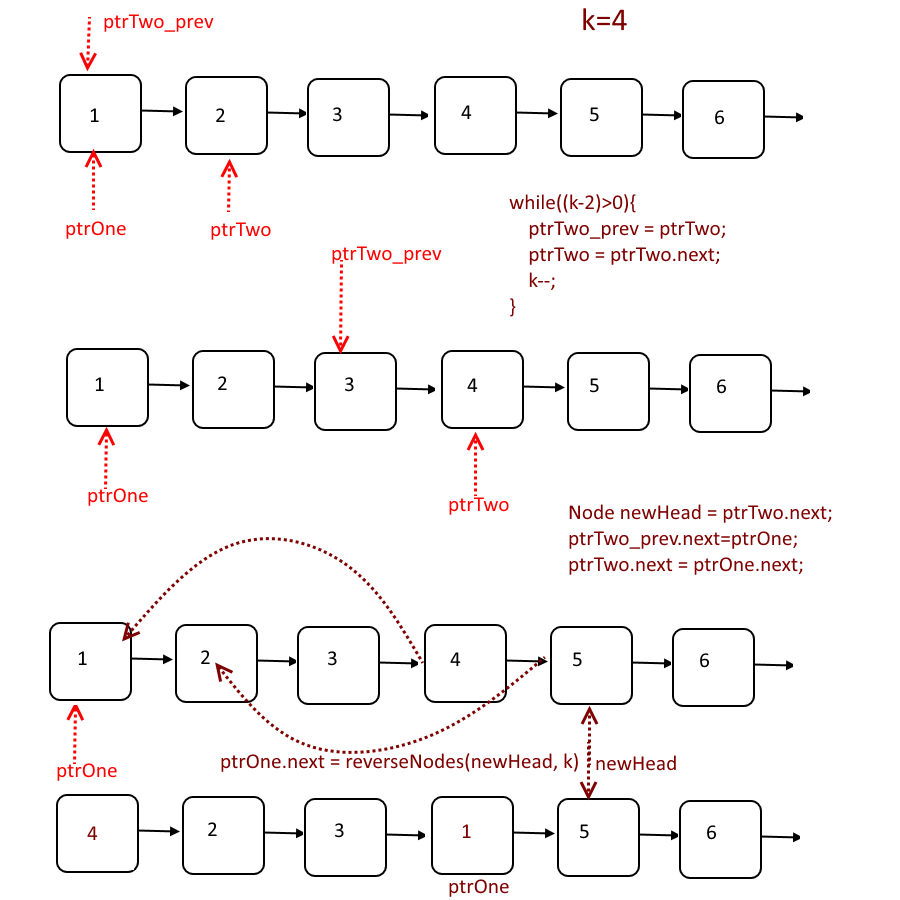
Input : ->1->2->3->4->5->6->7->8->9->10 , K = 4

Output: ->4->2->3->1->8->6->7->5->9->10

**Approach**:

* Take 3 Pointers, ptrOne, ptrTwo and ptrTwo\_prev.
* ptrOne and ptrTwo\_prev points at head node.
* ptrTwo points at next node of ptrTwo\_prev.
* Move the ptrTwo and ptrTwo\_prev k-2 times, since we need one pointer each at both ends for swapping so move pointers only k-2 times.
* Create another pointer , NewHead and point it to ptrTwo.next.
* Now we have ptrOne at head and ptrTwo at kth position, swap them with the help of ptrTwo\_prev.
* This function will returns the head.
* Now make a recursive call with newHead.

**ptrOne.next = reverseNodes(newHead, k);**

****

**Complete Code:**

**package** swapEverykthNodes;

**public** **class** SwapEveryKthNode {

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

LinkedListT a = **new** LinkedListT();

a.addAtBegin(10);

a.addAtBegin(9);

a.addAtBegin(8);

a.addAtBegin(7);

a.addAtBegin(6);

a.addAtBegin(5);

a.addAtBegin(4);

a.addAtBegin(3);

a.addAtBegin(2);

a.addAtBegin(1);

System.*out*.print("Original Link List 1 : ");

a.display(a.head);

**int** k = 4;

Node n = a.reverseNodes(a.head, k);

System.*out*.println("\n Swap Every " + k + "th Node : ");

a.display(n);

}

}

**class** Node{

**public** **int** data;

**public** Node next;

**public** Node(**int** data){

**this**.data = data;

**this**.next = **null**;

}

}

**class** LinkedListT{

**public** Node head;

**public** LinkedListT(){

head=**null**;

}

**public** Node reverseNodes(Node head, **int** k){

**int** x =k;

Node ptrOne = head;

Node ptrTwo\_prev = head;

Node ptrTwo = **null**;

**if**(k<2)**return** head;

**if**(ptrOne!=**null**){

ptrTwo = head.next;

}**else** **return** **null**;

**while**((x-2)>0){

**if**(ptrTwo!=**null**){

ptrTwo\_prev = ptrTwo;

ptrTwo = ptrTwo.next;

x--;

}**else**{

**return** head;

}

}

Node newHead = ptrTwo.next;

ptrTwo\_prev.next=ptrOne;

ptrTwo.next = ptrOne.next;

ptrOne.next = reverseNodes(newHead, k);

**return** ptrTwo;

}

**public** **void** addAtBegin(**int** data){

Node n = **new** Node(data);

n.next = head;

head = n;

}

**public** **void** display(Node head){

Node currNode = head;

**while**(currNode!=**null**){

System.*out*.print("->" + currNode.data);

currNode=currNode.next;

}

}

}

**Output:**

Original Link List 1 : ->1->2->3->4->5->6->7->8->9->10

Swap Every 4th Node :

->4->2->3->1->8->6->7->5->9->10

**Algorithms - Find Whether Two Strings are Permutation of each other**

**Objective:** Given Two Strings, check whether one string is permutation of other

**Input:** Two Linked list.

**Output:** True or false based on whether strings are permutation of other or not.

**Example:**

**"sumit" and "tiums" are permutations of each other.**

**"abcd" and bdea" are not permutations of each other.**

**Approach:**

**Method 1: Time Complexity - O(nlgn)**

Sort both the strings and compare it.

**Method 2 : Using Hash Table - Time Complexity - O(n)**

* Check if both Strings are having the same length, if not , return false.
* Create a Hash Table, make character as key and its count as value
* Navigate the string one taking each character at a time
* check if that character already existing in hash table, if yes then increase its count by 1 and if it doesn't exist insert into hash table with the count as 1.
* Now navigate the second string taking each character at a time
* check if that character existing in hash table, if yes then decrease its count by 1 and if it doesn't exist then return false.
* At the end navigate through hash table and check if all the keys has 0 count against it if yes then return true else return false.

Complete Code:

**package** newInterviewQuestions;

**import** java.util.Hashtable;

**import** java.util.Iterator;

**import** java.util.Set;

**public** **class** PermutationStrings {

**public** **boolean** isPermutation(String s1, String s2){

**if**(s1.length()!=s2.length()){

**return** **false**;

}

Hashtable<Character , Integer> ht = **new** Hashtable<>();

**for**(**int** i=0;i<s1.length();i++){

**char** c = s1.charAt(i);

**if**(ht.containsKey(c)){

**int** val = ht.get(c) +1;

ht.put(c, val);

}**else**{

ht.put(c, 1);

}

}

**for**(**int** i=0;i<s2.length();i++){

**char** c = s2.charAt(i);

**if**(ht.containsKey(c)){

**int** val = ht.get(c);

**if**(val==0){

**return** **false**;

}

val--;

ht.put(c, val);

}**else**{

**return** **false**;

}

}

Set<Character> keys = ht.keySet();

**for**(Character k :keys ){

**if**(ht.get(k)!=0){

**return** **false**;

}

}

**return** **true**;

}

**public** **static** **void** main(String args[]){

String s1 = "sumit";

String s2 = "mtisu";

PermutationStrings p = **new** PermutationStrings();

System.*out*.println(s1 +" and "+ s2 + " are permutation of each other? " + p.isPermutation(s1, s2));

s1 = "xyzab";

s2 = "bayzxx";

System.*out*.println(s1 +" and "+ s2 + " are permutation of each other? " + p.isPermutation(s1, s2));

}

}

Output:

sumit and mtisu are permutation of each other? true

xyzab and bayzxx are permutation of each other? false

**Algorithms - Replace all spaces in a String with '%20'**

**Objective:** In a given string , **replace all spaces in a String with '%20'. You can consider that string has enough space at the end of the string to hold the extra characters.**

**Input:** A String and true length of a string

**Output:** Updated string in which each space is replaced by the '%20'

**Example:**

Input String : I am Sumit Jain

Output String : I%20am%20Sumit%20Jain

**Approach:**

* Count the total spaces in a string in one iteration, say the count is spaceCount
* Calculate the new length of a string by newLength = length + 2\*spaceCount; (we need two more places for each space since %20 has 3 characters, one character will occupy the blank space and for rest two we need extra space)
* Do another iteration in reverse direction
* If you encounter the space, for next 3 spaces put %,2,0.
* If you encounter the character, copy it

Complete Code:

**package** newInterviewQuestions;

**public** **class** ReplaceAllSpaces {

**public** **void** replace(String s1, **int** length){

**char** [] chars = s1.toCharArray();

**int** spaceCount =0;

**for**(**int** i=0;i<length;i++){

**if**(chars[i]==' '){

spaceCount++;

}

}

**int** newLength = length + 2\*spaceCount;

**for**(**int** i=length-1;i>=0;i--){

**if**(chars[i]==' '){

chars[newLength-1]='0';

chars[newLength-2]='2';

chars[newLength-3]='%';

newLength = newLength-3;

}**else**{

chars[newLength-1]=chars[i];

newLength = newLength-1;

}

}

s1 = String.*valueOf*(chars);

System.*out*.println("Output String : " + s1);

}

**public** **static** **void** main(String args[]){

String s1 = "I am Sumit Jain ";

**int** trueLength = 15;

System.*out*.println("Input String : " + s1);

ReplaceAllSpaces r = **new** ReplaceAllSpaces();

r.replace(s1, trueLength);

}

}

Output:

Input String : I am Sumit Jain

Output String : I%20am%20Sumit%20Jain

**Algorithms - String Compression using count of repeated characters.**

**Objective:** Compress the given string by using the count of repeated characters and if new compressed string length is not smaller than the original string then return the original string.

**Example:**

Input String : ssssuuuummmmmmiiiittttttttttttt

Compressed String : s4u4m6i4t13

Input String : Jaain

Compressed String : Jaain (Since compressed string is length is greater than original string)

**Input:** A String

**Output: either** A compressed string or original string whichever us smaller.

**Approach:**

* Create a StringBuffer sb, int count
* Navigate the string taking each character at a time.
* If you find the same characters increase the count.
* if not then append the character and its count to the string buffer sb.
* reset the count value.
* Compare the length of compressed String and original and whichever is smaller return that string.

Complete Code:

**package** newInterviewQuestions;

**public** **class** StringCompression {

**public** String compression(String s1){

StringBuffer sb = **new** StringBuffer();

**int** count =1;

**char** prev = s1.charAt(0);

**for**(**int** i=1;i<s1.length();i++){

**char** curr =s1.charAt(i);

**if**(prev==curr){

count++;

}**else**{

sb.append(prev);

sb.append(count);

prev = curr;

count=1;

}

}

sb.append(prev);

sb.append(count);

**if**(s1.length()<sb.length()){

**return** s1;

}**else**{

**return** sb.toString();

}

}

**public** **static** **void** main(String args[]){

String s1 = "ssssuuuummmmmmiiiittttttttttttt";

StringCompression sc = **new** StringCompression();

System.*out*.println("Compression of " + s1 + " is : " +sc.compression(s1));

s1 = "Jaain";

System.*out*.println("Compression of " + s1 + " is : " +sc.compression(s1));

}

}

Output:

Compression of ssssuuuummmmmmiiiittttttttttttt is : s4u4m6i4t13

Compression of Jaain is : Jaain

**Algorithms - Check if one string is Rotation of another string**

**Objective:** **Check if one string is Rotation of another string**.

**Example:**

Input Strings : 'sumitjain' and 'tjainsumi'

Output : true

Input String : 'Jaain' and 'ainJ'

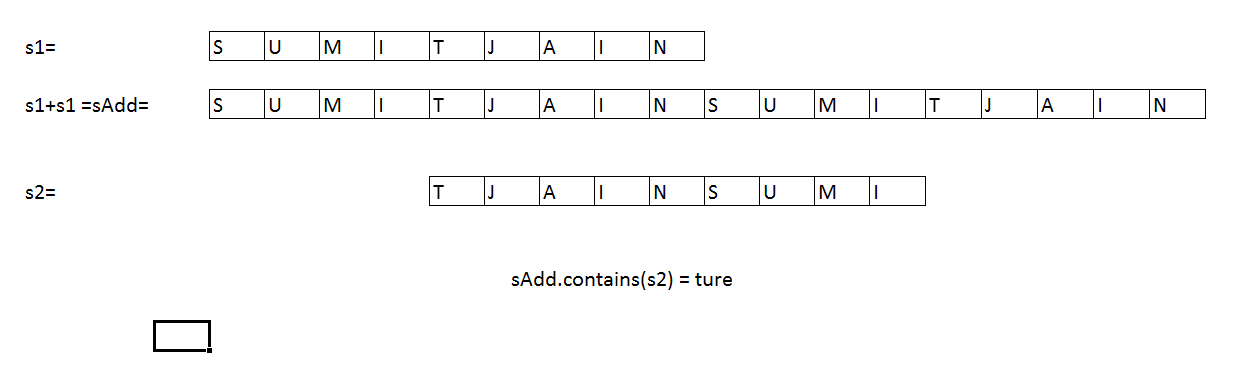
Output: false

**Input:** Two Strings

**Output: True or false based on whether strings are rotation of each other.**

**Approach:**

* Make a new String by appending the appending the first string with itself
* Check if second string is sub string of new String



Complete Code:

**package** newInterviewQuestions;

**public** **class** RotatedArray {

**public** **boolean** isRotated(String s1, String s2){

**if**(s1.length()!=s2.length()){

**return** **false**;

}

String sAdd = s1 + s1;

**if**(sAdd.contains(s2)){

**return** **true**;

}**else**{

**return** **false**;

}

}

**public** **static** **void** main(String arg[]){

String s1 = "sumitjain";

String s2 = "tjainsumi";

RotatedArray r = **new** RotatedArray();

System.*out*.println("Is '" + s1 + "' and '" + s2 + "' are rotated?? : " + r.isRotated(s1, s2));

}

}

Output:

Is 'sumitjain' and 'tjainsumi' are rotated?? : true

**Algorithms - Remove Duplicates from an unsorted Linked list.**

**Objective:** **Write a program to remove the duplicates from an unsorted linked list**

**Example:**

Input Linked List : 1->2->2->4->3->3->2

Output : 1->2->4->3->2

**Input:** An unsorted linked list

**Output: Linked list with no duplicates.**

**Approach:**

* Create a Hash Table
* Take two pointers, prevNode and CurrNode.
* PrevNode will point to the head of the linked list and currNode will point to the head.next.
* Now navigate through the linked list.
* Check every node data is present in the HashTable.
* if yes then delete that node using prevNode and currNode.
* If No, then insert that node data into the linked list
* Return the head of the list

Time Complexity : O(n)

Space Complexity : O(n)

**Follow Up**: If suppose addition buffer is not allowed then we have option but to check every node data against every other node data and if find duplicates, delete that node.

Time Complexity : O(n^2)

Complete Code for the Hash Table method:

**package** RemoveDuplicatesFromLL;

**import** java.util.Hashtable;

**public** **class** RemoveDuplicates {

**public** Node removeDup(Node head){

Hashtable<Integer, Integer> ht = **new** Hashtable<>();

**if**(head==**null**){

**return** **null**;

}

Node currNode = head.next;

Node prevNode = head;

**int** count =0;

**while**(currNode!=**null**){

**int** data = currNode.data;

**if**(ht.contains(data)){

prevNode.next = currNode.next;

currNode = currNode.next;

}**else**{

ht.put(count, data);

count++;

prevNode = currNode;

currNode = currNode.next;

}

} **return** head;

}

**public** **void** display(Node head){

Node n=head;

**while**(n!=**null**){

System.*out*.print("->" + n.data);

n=n.next;

}

}

**public** **static** **void** main(String args[]){

Node n = **new** Node(1);

n.next = **new** Node(2);

n.next.next = **new** Node(2);

n.next.next.next = **new** Node(3);

n.next.next.next.next = **new** Node(4);

n.next.next.next.next.next = **new** Node(4);

n.next.next.next.next.next.next = **new** Node(2);

System.*out*.print("Original List : ");

RemoveDuplicates rm = **new** RemoveDuplicates();

rm.display(n);

System.*out*.print("\n Updated List: ");

Node x =rm.removeDup(n);

rm.display(x);

}

}

**class** Node{

**int** data;

Node next;

**public** Node(**int** data){

**this**.data = data;

next = **null**;

}

}

Output:

Original List : ->1->2->2->3->4->4->2

Updated List: ->1->2->3->4

**Algorithms - To find the kth to Last Element of a Singly Linked List**

**Objective:** **Write a program to find the kth to Last Element of a Singly Linked List**

**Example:**

Original List : ->1->2->8->3->7->0->4

Output : 3rd Element from the end is : 7

**Input:** An unsorted linked list and integer k

**Output: The kth to Last Element of a Singly Linked List**

**Approach:**

**Recursive Approach:**

* Recurse through the Linked list
* When we reach to the end of the list, base case will return 0
* Now with each passing back call through stack, add 1 and return.
* When count hits k, print the value.

**Iterative Approach:**

* Take two pointers approach
* Move first pointer by k
* now move both the pointers and when the first pointer reaches the end of the list the second pointer will be at the kth node from the end.
* Return the kth node data.''

**Complete Code:**

**package** KthToLastElementofLL;

**public** **class** KthToLastElementofLL {

**public** **int** kthByRecursion(Node head, **int** k){

**if**(head==**null**){

**return** 0;

}

**int** i = kthByRecursion(head.next, k)+1;

**if**(i==k){

System.*out*.println(head.data);

}

**return** i;

}

**public** **int** kthByIteration(Node head, **int** k){

**if**(head==**null**){

**return** 0;

}

Node curr = head;

**while**(k>0){

curr=curr.next;

k--;

}

Node sec = head;

**while**(curr!=**null**){

curr = curr.next;

sec = sec.next;

}

**int** i = sec.data;

**return** i;

}

**public** **void** display(Node head){

Node n=head;

**while**(n!=**null**){

System.*out*.print("->" + n.data);

n=n.next;

}

}

**public** **static** **void** main(String args[]){

Node n = **new** Node(1);

n.next = **new** Node(2);

n.next.next = **new** Node(8);

n.next.next.next = **new** Node(3);

n.next.next.next.next = **new** Node(7);

n.next.next.next.next.next = **new** Node(0);

n.next.next.next.next.next.next = **new** Node(4);

System.*out*.print("Original List : ");

KthToLastElementofLL rm = **new** KthToLastElementofLL();

rm.display(n);

System.*out*.print("\n Recursion::3rd Element from the end is : ");

rm.kthByRecursion(n, 3);

System.*out*.print("\n Iteration::5th Element from the end is : " + rm.kthByIteration(n, 5));

}

}

**class** Node{

**int** data;

Node next;

**public** Node(**int** data){

**this**.data = data;

next = **null**;

}

}

Original List : ->1->2->8->3->7->0->4

Recursion::3rd Element from the end is : 7

Iteration::5th Element from the end is : 8

**Algorithms - Delete a Node in the Middle of a linked list, Given only access to that Node**

**Objective:** **Write a program to Delete a Node in the Middle of a linked list, Given only access to that Node**

**Example:**

Original List : ->1->2->8->3->7->0->4

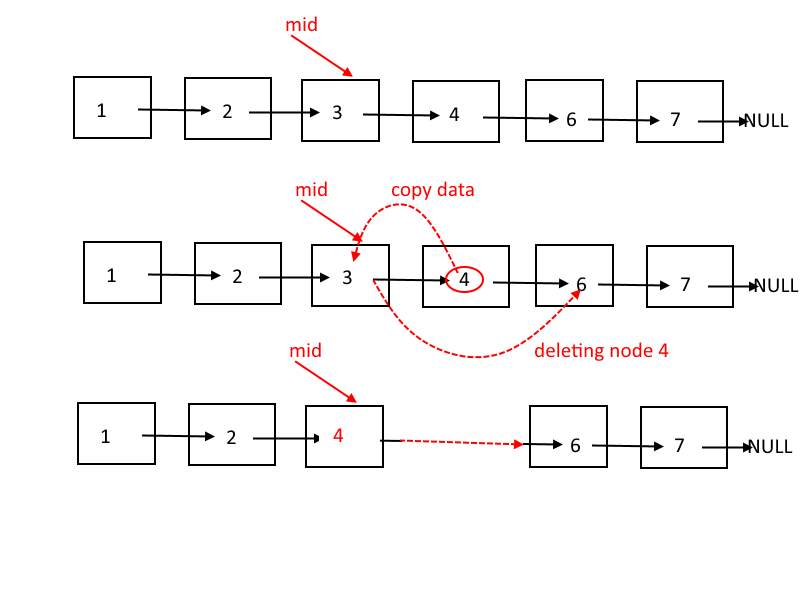
After Deleting the mid node (say 7) : ->1->2->8->3->0->4

**Input:** A Linked List and access to the node which needs to be deleted

**Output: Linked list with deleted node**

**Approach:**

* Approach is tricky and simple
* Copy the value of next node to the node which you want to delete
* Delete the next node



**Complete Code:**

**package** deleteMiddleNodeInLL;

**public** **class** deleteMiddleNodeInLL {

**public** **void** deleteMiddle(Node mid){

**if**(mid.next==**null**){

**return**; // we cant delete the node if it is the last node in the linked list

}

Node curr = mid;

curr.data = curr.next.data;

curr.next = curr.next.next;

}

**public** **void** display(Node head){

Node n=head;

**while**(n!=**null**){

System.*out*.print("->" + n.data);

n=n.next;

}

}

**public** **static** **void** main(String args[]){

Node n = **new** Node(1);

n.next = **new** Node(2);

n.next.next = **new** Node(8);

n.next.next.next = **new** Node(3);

Node mid = **new** Node(7);

n.next.next.next.next = mid;

n.next.next.next.next.next = **new** Node(0);

n.next.next.next.next.next.next = **new** Node(4);

System.*out*.print("Original List : ");

deleteMiddleNodeInLL rm = **new** deleteMiddleNodeInLL();

rm.display(n);

System.*out*.print("\n Aftter Deleting the mid node : ");

rm.deleteMiddle(mid);

rm.display(n);

}

}

**class** Node{

**int** data;

Node next;

**public** Node(**int** data){

**this**.data = data;

next = **null**;

}

}

Original List : ->1->2->8->3->7->0->4

Aftter Deleting the mid node (say 7) : ->1->2->8->3->0->4

**Algorithms - Reverse the binary representation of a number.**

**Objective:** **Write a program to Reverse the binary representation of a number**

**Example:**

Input : 30

Output : 15

**Explanation**:

binary representation of 30 is : 11110

reverse of binary representation : 01111

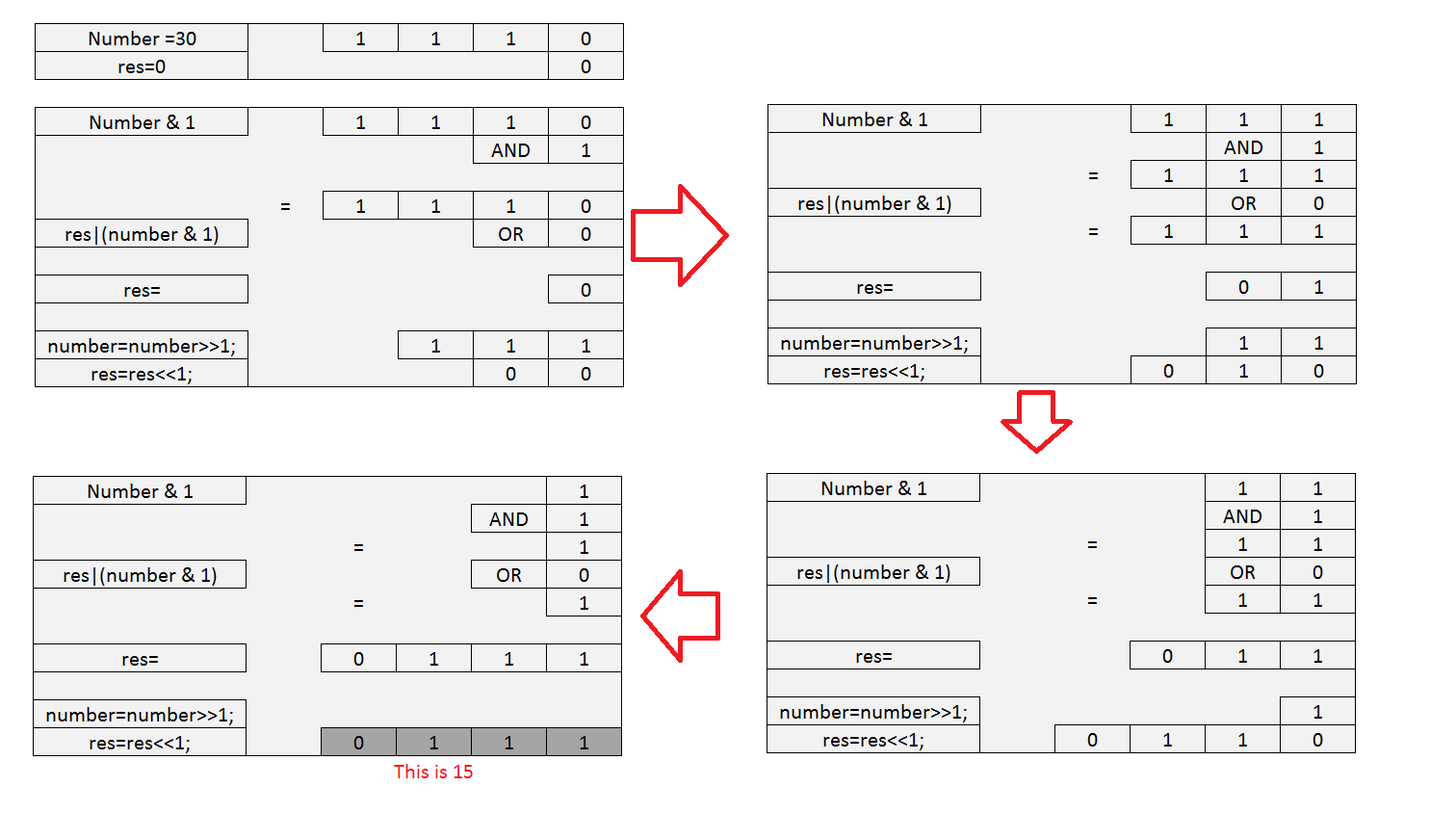
decimal of reversed binary representation is : 15

**Input:** A Number

**Output: Decimal of reversed binary representation of a number.**

**Approach:**

* Initialize int res =0
* Now from a number , take one bit at a time
* take AND of that bit with 1 and then OR with res and store it in res
* make right shift in number by 1
* make left shift in res by 1



**Complete Code:**

**package** newInterviewQuestions;

**public** **class** BinaryRotate {

**public** **int** rotateBinary(**int** number){

**int** res = 0;

**while**(number>0){

res=res<<1;

res = res|(number & 1);

number=number>>1;

}

**return** res;

}

**public** **static** **void** main(String args[]){

**int** x =30;

BinaryRotate b = **new** BinaryRotate();

System.*out*.println("Binary rotation of "+ x + " is : " + b.rotateBinary(x));

}

}

**Output:**

Binary rotation of 30 is : 15

**Algorithms - Reverse a Linked List - Part 2**

**This post is the extension of our earlier post Reverse a linked list. Here We have provided the better recursive solution in which we start reversing the list from the end.**

**Objective:** Reverse the given linked list.

**Input:** A Linked List

**Output:** Reversed Linked List

**Example:**

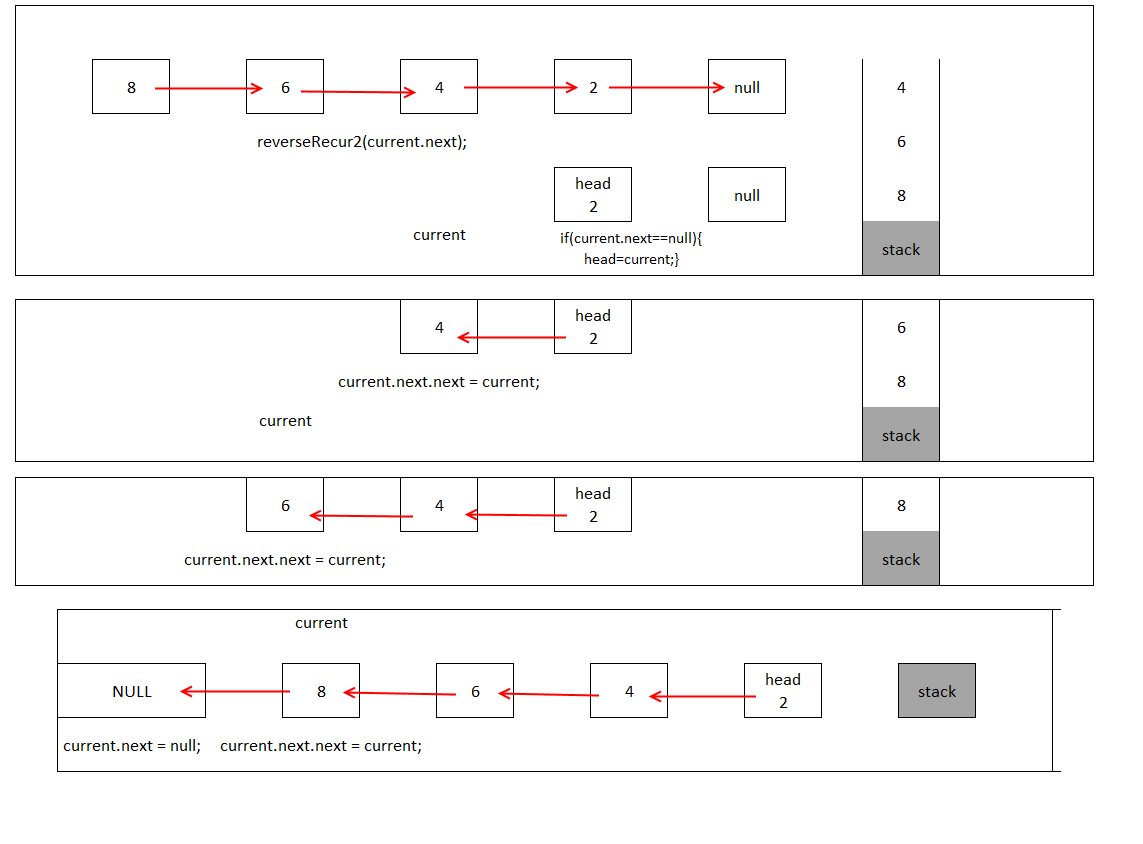
Original List :->10->8->6->4->2

Reversed List :->2->4->6->8->10

**Approach:**

**Recursion:**

* Traverse till the end of list through recursion.
* Make the last node as head.
* Now you are at the end of the list and rest of the nodes are stores in a stack
* Now while coming back, each node will pop out from the stack in reverse order
* take these nodes and start pointing it to next node coming out of stack.

****

**Complete Code:**

**package** reverseLinkedList2;

**public** **class** reverseLinkedList2 {

**public** **static** Node *head*=**null**;

**public** Node reverseRecur2(Node current){

**if**(current==**null**){

**return** **null**;

}

**if**(current.next==**null**){

*head* = current;

**return** **null**;

}

reverseRecur2(current.next);

current.next.next = current;

current.next = **null**;

**return** *head*;

}

**public** **void** display(Node head){

//

Node currNode = head;

**while**(currNode!=**null**){

System.*out*.print("->" + currNode.data);

currNode=currNode.next;

}

}

**public** **static** **void** main(String args[]){

*head* = **new** Node(10);

*head*.next = **new** Node(8);

*head*.next.next = **new** Node(6);

*head*.next.next.next = **new** Node(4);

*head*.next.next.next.next = **new** Node(2);

System.*out*.println("Original List :");

reverseLinkedList2 r= **new** reverseLinkedList2();

r.display(*head*);

System.*out*.println("\nReversed List :");

Node x = r.reverseRecur2(*head*);

r.display(x);

}

}

**class** Node{

**public** **int** data;

**public** Node next;

**public** Node(**int** data){

**this**.data = data;

**this**.next = **null**;

}

}

Original List :

->10->8->6->4->2

Reversed List :

->2->4->6->8->10

**Algorithms - Add two numbers represented by a linked list, Number Stored in REVERSE order**

**We have similar post - Two numbers represented by a linked list, Number Stored in FORWARD order**

**Objective:** Two numbers represented by a linked listwhere each node contains single digit. The digits are stored in REVERSE order, means head is pointing to the first digit of the number.

**Input:** Two numbers represented by Linked Lists

**Output:** Addition of two numbers represented by a Linked List.

**Example:**

First Number in REVERSE order: 5957

Second Number in REVERSE order : 59

Addition in REVERSE order : 0967

Actual Result in FORWARD ORDER : 9670

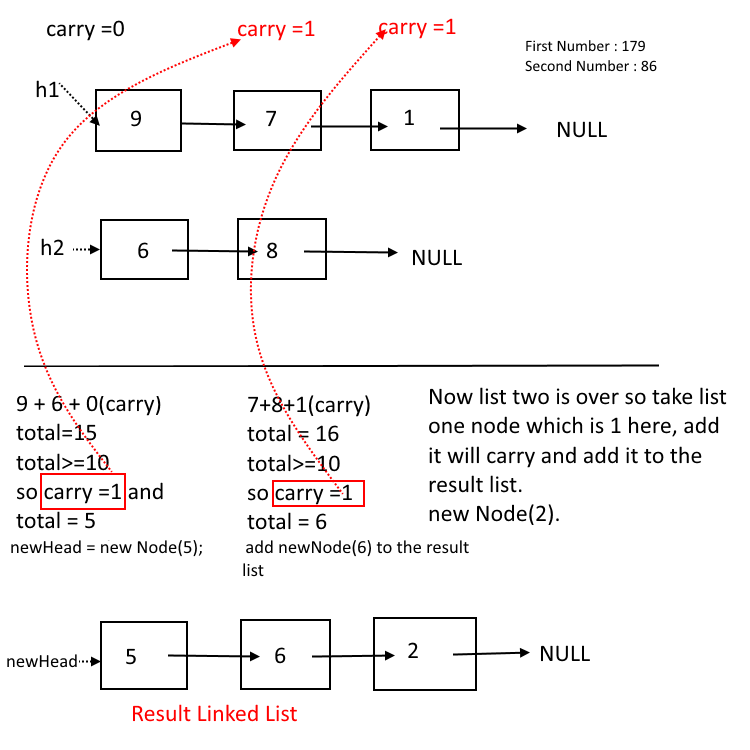
**Approach:**

1. Take a variable int **carry** =0.
2. Initialize Node **newHead** = null; and Node **curr** = null.
3. newHead will be the starting node of our result linked list and curr node will the reference to the current node on which we are working in our result linked list.
4. Navigate Both the lists simultaneously taking one node at a time.
5. Add the data of nodes and carry , say you call this as total.
6. Check if total >=10, if yes put carry =1 and total=total-10.
7. create a new node with value total, say you call it as Node 'n'.
8. check if newHead is null, if yes then and assign 'n' to newHead. Now our starting node of result linked list is fixed.
9. if newHead is not null then add 'n' to the result linked list with the help of newHead and curr.
10. Now repeat steps 4 to 9 till any one of the list gets over( considering both the list has different length, if both list has the same length then both lists gets over at the same time, you will not need step 11).
11. Now navigate the list ( whichever is remaining) and add it to the result list. (take care of the carry, see Example). *You can avoid this step by making sure that both the list has the same length adding 0 at the end of the shorter list , to see the similar implementation click here.*
12. At the End check the carry, if it is not 0, create a new node with value 1 and add it to the result linked list.

**Example:**

**First Number : 179**

**Second Number : 86**



Complete Code:

**package** LinkedListAddtionReverseOrder;

**public** **class** LinkedListAddtionReverseOrder {

**public** Node add(Node h1, Node h2){

**int** carry = 0;

Node newHead = **null**;

Node curr=**null**;

**while**(h1!=**null** && h2!=**null**){

**int** a = h1.data;

**int** b = h2.data;

**int** total = a+b+carry;

**if**(total>=10){

carry = 1;

total = total-10;

}

**if**(newHead==**null**){

newHead = **new** Node(total);

curr = newHead;

}**else**{

Node n = **new** Node(total);

curr.next = n;

curr = curr.next;

}

h1=h1.next;

h2=h2.next;

}

**while**(h1!=**null**){

**int** x= h1.data + carry;

Node n = **new** Node(x);

curr.next = n;

curr = curr.next;

h1=h1.next;

carry=0;

}

**while**(h2!=**null**){

**int** x= h2.data + carry;

Node n = **new** Node(x);

curr.next = n;

curr = curr.next;

h2=h2.next;

carry=0;

}

**if**(carry>0){

Node n = **new** Node(1);

curr.next = n;

curr = curr.next;

}

**return** newHead;

}

**public** **void** display(Node head){

Node currNode = head;

**while**(currNode!=**null**){

System.*out*.print("" + currNode.data);

currNode=currNode.next;

}

**public** **void** displayReverse(Node head){

Node currNode = head;

**if**(head==**null**){

**return**;

}

display(head.next);

System.*out*.print(head.data);

}

**public** **static** **void** main(String args[]){

LinkedListAddtionReverseOrder l = **new** LinkedListAddtionReverseOrder();

Node h1 = **new** Node(5);

h1.next= **new** Node(9);

h1.next.next = **new** Node(5);

h1.next.next.next = **new** Node(7);

System.*out*.print("First Number in REVERSE order: ");

l.display(h1);

Node h2 = **new** Node(5);

h2.next= **new** Node(9);

System.*out*.print("\n Second Number in REVERSE order : ");

l.display(h2);

Node x = l.add(h2, h1);

System.*out*.print("\n Addition in REVERSE order : ");

l.display(x);

System.*out*.print("\n Actual Result in FORWARD ORDER : ");

l.displayReverse(x);

}

}

**class** Node{

**public** **int** data;

**public** Node next;

**public** Node(**int** data){

**this**.data = data;

**this**.next = **null**;

}

}

Output :

First Number in REVERSE order: 5957

Second Number in REVERSE order : 59

Addition in REVERSE order : 0967

Actual Result in FORWARD ORDER : 9670

**Algorithms -** Add two numbers represented by a linked list, Number Stored in FORWARD order

**This post is the extension of - Two numbers represented by a linked list, Number Stored in FORWARD order**

**Objective:** Two numbers represented by a linked listwhere each node contains single digit. The digits are stored in Forward order, means head is pointing to the last digit of the number.

**Input:** Two numbers represented by Linked Lists

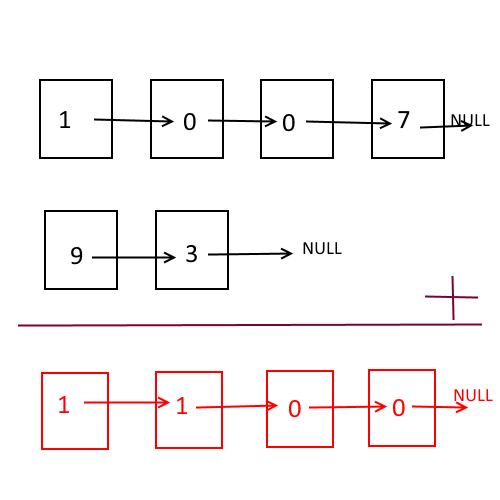
**Output:** Addition of two numbers represented by a Linked List.

**Example:**

First Number : 1007

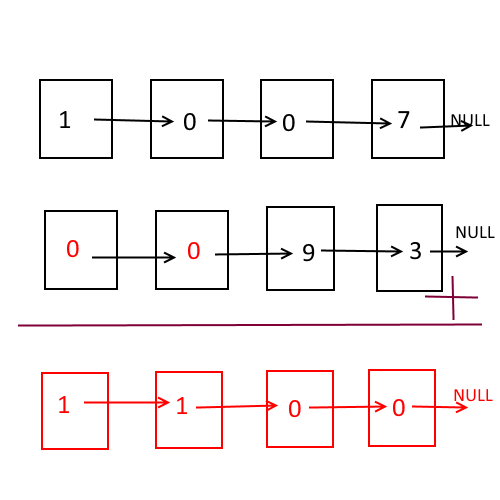
Second Number : 93

Addition : 1100

****

**Approach:**

**Recursion.**

* Get the length of both the lists.
* If lengths are not equal, make them equal by adding nodes with value 0 in front of shorter linked list.
* 
* Create a global variable carry=0.
* Create a newHead = null;
* newHead will be the starting node of our result linked list and curr node will the reference to the current node on which we are working in our result linked list.
* Now using recursion travel in both the list till the end.
* So now nodes are stores in a stack
* Now while coming back, each node will pop out from the stack in reverse order
* Take node data from both the lists add them along with carry.
* if sum is >=10 , then make carry as 1 and create a new node with sum-10
* Else just create a new Node with sum.
* Add the newly created node to the result linked list with the help of newHead.

**Complete Code:**

**package** LinkedListAddtionForwardOrder;

**public** **class** LinkedListAddtionForwardOrder {

**public** **int** carry=0;

**public** Node newHead = **null**;

**public** Node add(Node h1, Node h2){

//first we will make sure that both the Linked list has same no of nodes

// to ensure that we will append 0 in front of shorter list

**int** h1Len = getLength(h1);

**int** h2Len = getLength(h2);

**if**(h1Len>h2Len){

**int** diff = h1Len-h2Len;

**while**(diff>0){

Node n = **new** Node(0);

n.next = h2;

h2=n;

diff--;

}

}

**if**(h1Len<h2Len){

**int** diff = h2Len-h1Len;

**while**(diff>0){

Node n = **new** Node(0);

n.next = h1;

h1=n;

diff--;

}

}

**return** addBackRecursion(h1, h2);

}

**public** Node addBackRecursion(Node h1, Node h2){

**if**(h1==**null** && h2==**null**){

**return** **null**;

}

addBackRecursion(h1.next, h2.next);

**int** a = h1.data + h2.data + carry;

carry=0;

//System.out.println(a);

**if**(a>=10){

carry =1;

a = a%10;

}

Node n = **new** Node(a);

**if**(newHead==**null**){

newHead =n;

}**else**{

n.next = newHead;

newHead = n;

}

//carry=0;

**return** newHead;

}

**public** **int** getLength(Node head){

**int** len=0;

**while**(head!=**null**){

len++;

head = head.next;

}

**return** len;

}

**public** **void** display(Node head){

Node currNode = head;

**while**(currNode!=**null**){

System.*out*.print("->" + currNode.data);

currNode=currNode.next;

}

}

**public** **static** **void** main(String args[]){

LinkedListAddtionForwardOrder l = **new** LinkedListAddtionForwardOrder();

Node h1 = **new** Node(1);

h1.next= **new** Node(0);

h1.next.next = **new** Node(0);

h1.next.next.next = **new** Node(7);

System.*out*.print("First Number : ");

l.display(h1);

Node h2 = **new** Node(9);

h2.next= **new** Node(3);

System.*out*.print("\n Second Number : ");

l.display(h2);

Node x = l.add(h1, h2);

System.*out*.print("\n Addition : ");

l.display(x);

}

}

**class** Node{

**public** **int** data;

**public** Node next;

**public** Node(**int** data){

**this**.data = data;

**this**.next = **null**;

}

}

**Output:**

First Number : ->1->0->0->7

Second Number : ->9->3

Addition : ->1->1->0->0

**Algorithms -** Binary Search Tree Implementation.

Binary Tree : A data structure in which we have nodes containing data and two references to other nodes, one on the left and one on the right.

Binary Tree consist of Nodes

* Nodes are nothing but objects of a class and each node has data and a link to the left node and right node.
* Usually we call the starting node of a tree as *root*.

**class** Node{

**int** data;

Node left;

Node right;

**public** Node(**int** data){

**this**.data = data;

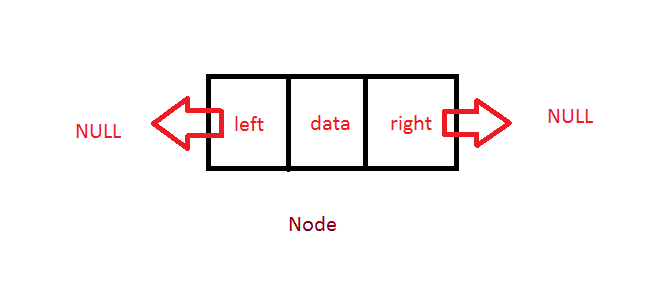
left = **null**;

right = **null**;

}

}

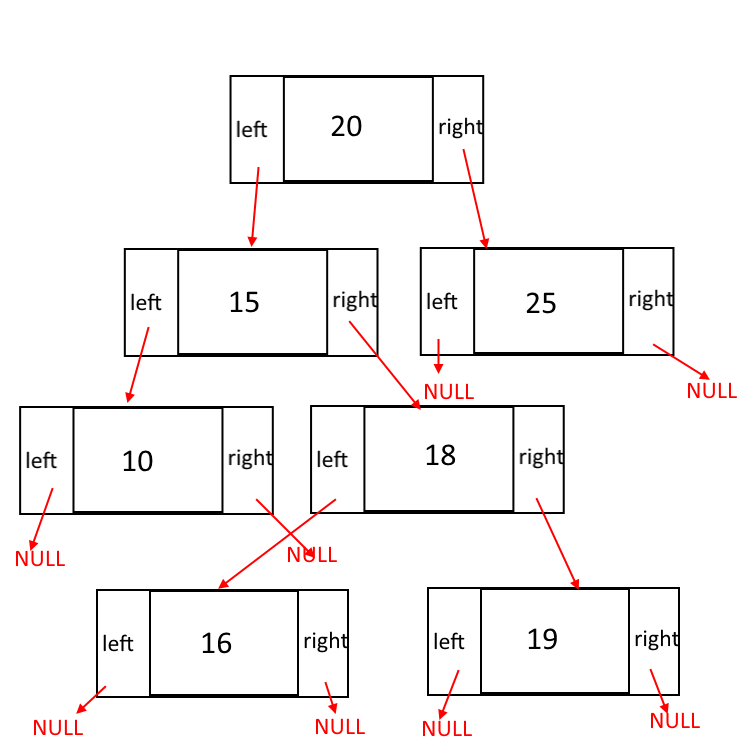
* Left and right node of a Leaf node points to NULL so you will know that you have reached to the end of the tree.



**Binary Search Tree:**

Often we call it as BST, is a type of Binary tree which has a special property.

*Nodes smaller than root goes to the left of the root and Nodes greater than root goes to the right of the root*.



**Operations:**

**Insert(int n) :** Add a node the tree with value n. Its O(lgn)

**Find(int n) :** Find a node the tree with value n. Its O(lgn)

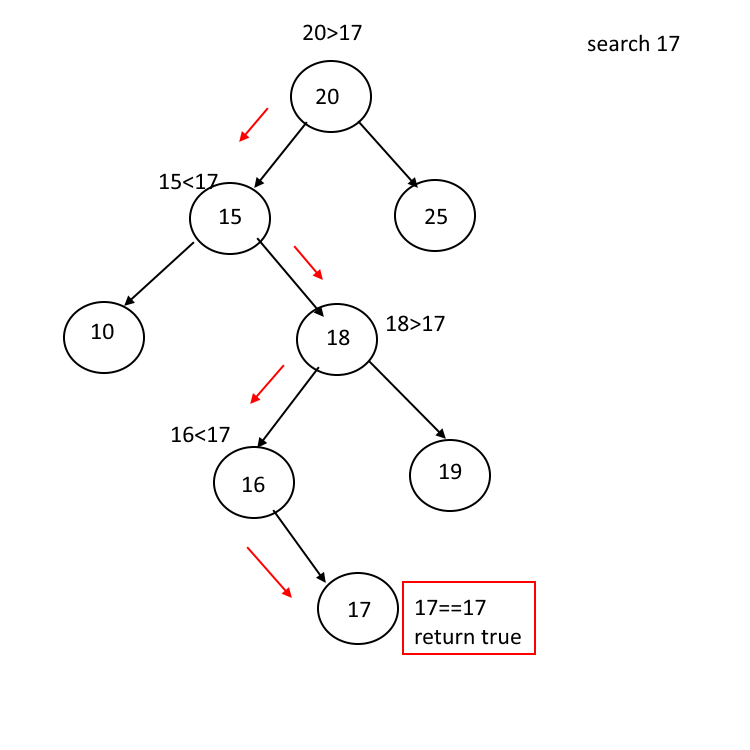
**Delete (int n)** : Delete a node the tree with value n. Its O(lgn)

**Display**(): Prints the entire tree in increasing order. O(n).

Detail Explanations for the Operations:

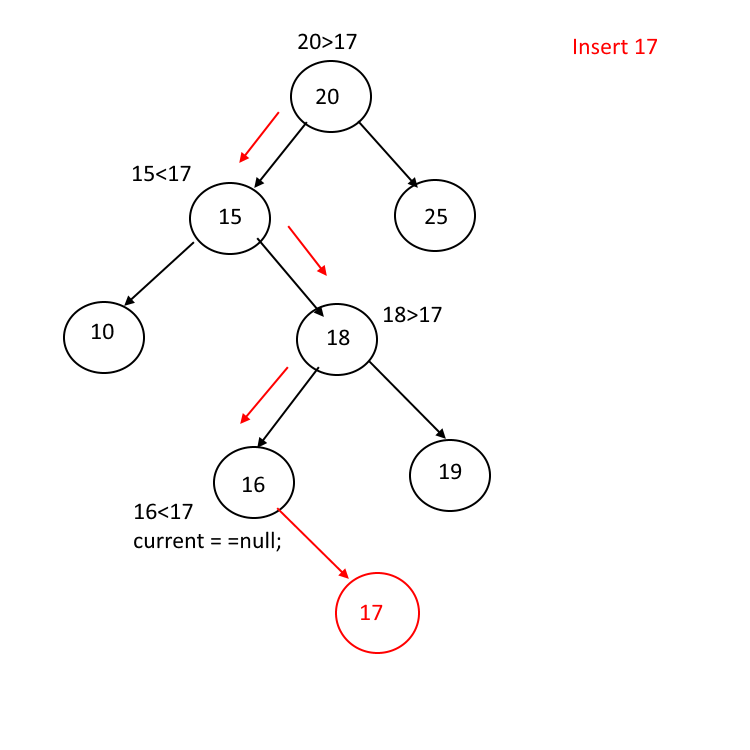
**Find(int n):**

* Its very simple operation to perform.
* start from the root and compare root.data with n
* if root.data is greater than n that means we need to go to the left of the root.
* if root.data is smaller than n that means we need to go to the right of the root.
* if any point of time root.data is equal to the n then we have found the node, return true.
* if we reach to the leaves (end of the tree) return false, we didnt find the element



**Insert(int n):**

* Very much similar to find() operation.
* To insert a node our first task is to find the place to insert the node.
* Take current = root .
* start from the current and compare root.data with n
* if current.data is greater than n that means we need to go to the left of the root.
* if current.data is smaller than n that means we need to go to the right of the root.
* if any point of time current is null that means we have reached to the leaf node, insert your node here with the help of parent node. (See code)



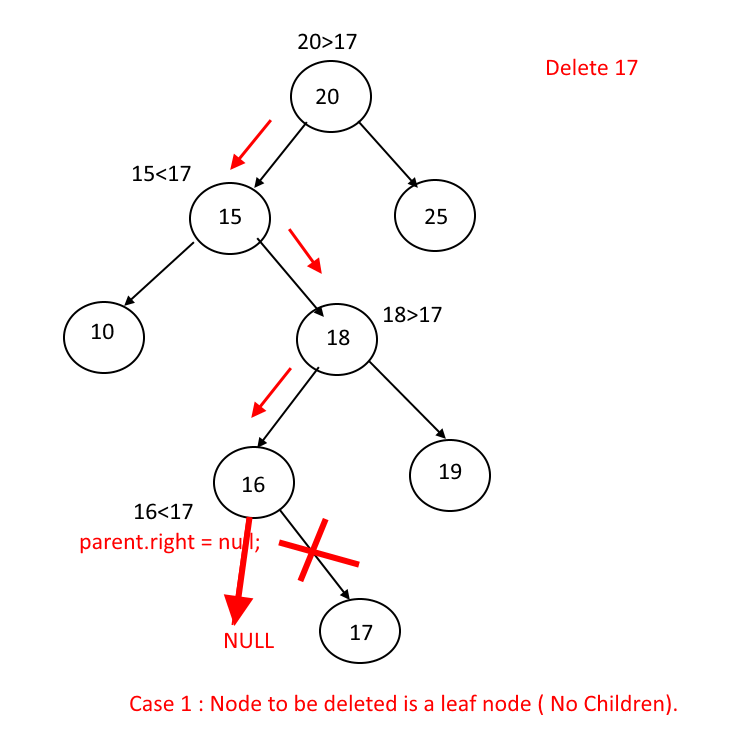
Delete(int n):

Complicated than Find() and Insert() operations. Here we have to deal with 3 cases.

* Node to be deleted is a leaf node ( No Children).
* Node to be deleted has only one child.
* Node to be deleted has two childrens.

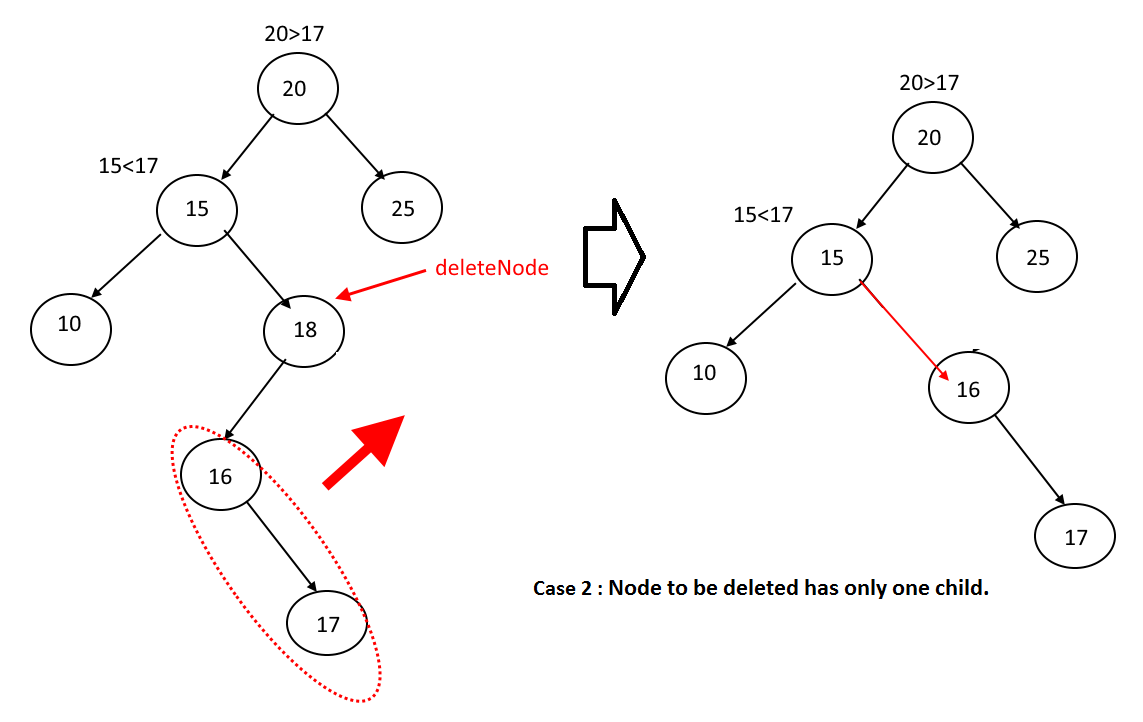
**Node to be deleted is a leaf node ( No Children).**

its a very simple case, if a node to be deleted has no children then just traverse to that node, keep track of parent node and the side in which the node exist(left or right) and set ***parent.left = null or parent.right = null;***

******

**Node to be deleted has only one child.**

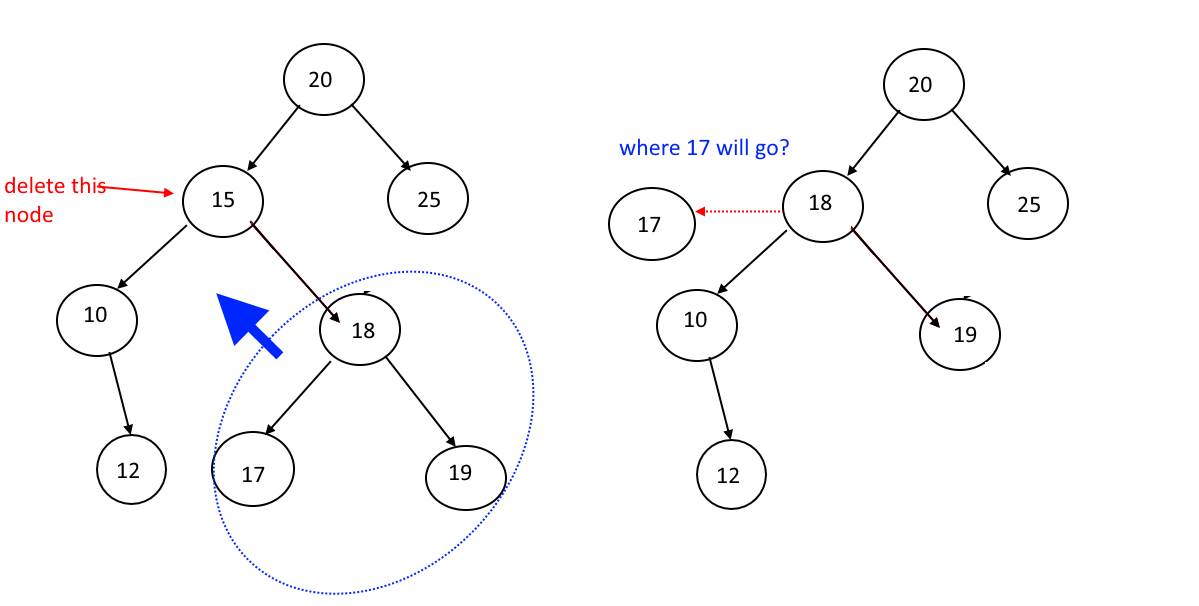
1. its a slightly complex case. if a node to be deleted(deleteNode) has only one child then just traverse to that node, keep track of parent node and the side in which the node exist(left or right).
2. check which side child is null (since it has only one child).
3. Say node to be deleted has child on its left side . Then take the entire sub tree from the left side and add it to the parent and the side on which deleteNode exist, see step 1 and example.

******

**Node to be deleted has two childrens.**

Now this is quite exciting :)

You just cannot replace the deleteNode with any of its child, Why? Lets try our a example.

****

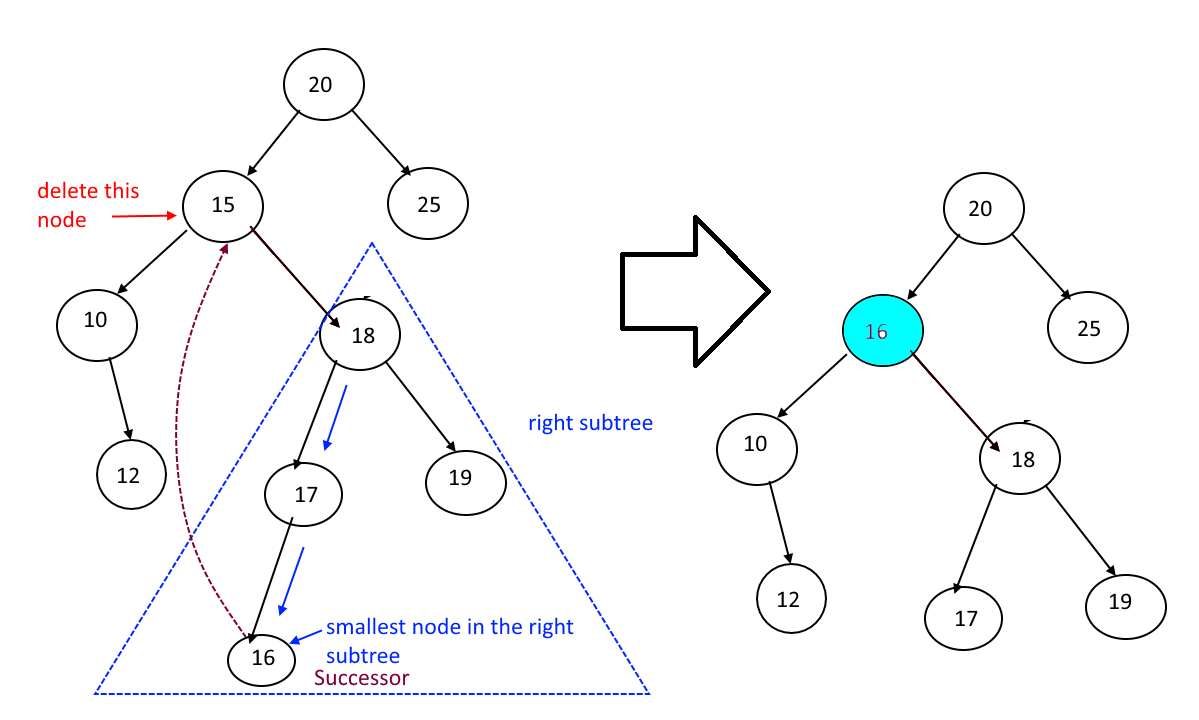
**What to do now?????**

Dont worry we have solution for this :)

**Find The Successor:**

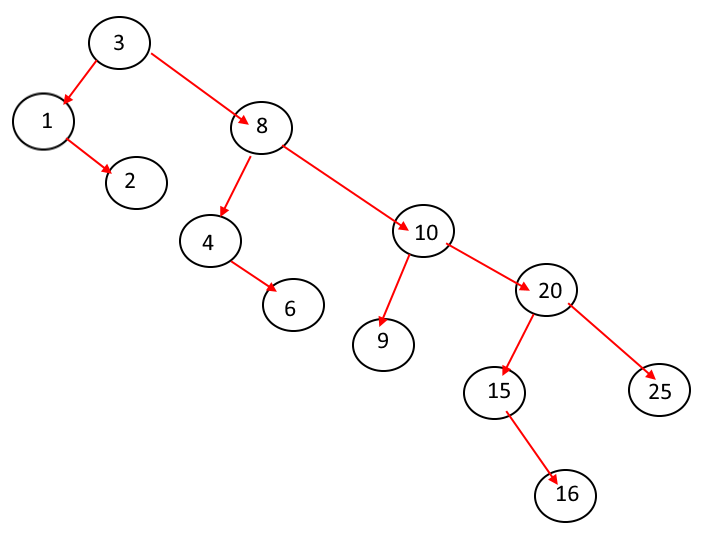
Successor is the node which will replace the deleted node. Now the question is to how to find it and where to find it.

Successor is the smaller node in the right sub tree of the node to be deleted.



Display() : To know about how we are displaying nodes in increasing order, Click Here

Complete Example :



**Complete Example Code:**

**package** binaryTree;

**public** **class** BinarySearchTree {

**public** **static** Node *root*;

**public** BinarySearchTree(){

**this**.*root* = **null**;

}

**public** **boolean** find(**int** id){

Node current = *root*;

**while**(current!=**null**){

**if**(current.data==id){

**return** **true**;

}**else** **if**(current.data>id){

current = current.left;

}**else**{

current = current.right;

}

}

**return** **false**;

}

**public** **boolean** delete(**int** id){

Node parent = *root*;

Node current = *root*;

**boolean** isLeftChild = **false**;

**while**(current.data!=id){

parent = current;

**if**(current.data>id){

isLeftChild = **true**;

current = current.left;

}**else**{

isLeftChild = **false**;

current = current.right;

}

**if**(current ==**null**){

**return** **false**;

}

}

//if i am here that means we have found the node

//Case 1: if node to be deleted has no children

**if**(current.left==**null** && current.right==**null**){

**if**(current==*root*){

*root* = **null**;

}

**if**(isLeftChild ==**true**){

parent.left = **null**;

}**else**{

parent.right = **null**;

}

}

//Case 2 : if node to be deleted has only one child

**else** **if**(current.right==**null**){

**if**(current==*root*){

*root* = current.left;

}**else** **if**(isLeftChild){

parent.left = current.left;

}**else**{

parent.right = current.left;

}

}

**else** **if**(current.left==**null**){

**if**(current==*root*){

*root* = current.right;

}**else** **if**(isLeftChild){

parent.left = current.right;

}**else**{

parent.right = current.right;

}

}**else** **if**(current.left!=**null** && current.right!=**null**){

//now we have found the minimum element in the right sub tree

Node successor = getSuccessor(current);

**if**(current==*root*){

*root* = successor;

}**else** **if**(isLeftChild){

parent.left = successor;

}**else**{

parent.right = successor;

}

successor.left = current.left;

}

**return** **true**;

}

**public** Node getSuccessor(Node deleleNode){

Node successsor =**null**;

Node successsorParent =**null**;

Node current = deleleNode.right;

**while**(current!=**null**){

successsorParent = successsor;

successsor = current;

current = current.left;

}

//check if successor has the right child, it cannot have left child for sure

// if it does have the right child, add it to the left of successorParent.

// successsorParent

**if**(successsor!=deleleNode.right){

successsorParent.left = successsor.right;

successsor.right = deleleNode.right;

}

**return** successsor;

}

**public** **void** insert(**int** id){

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

**if**(*root*==**null**){

*root* = newNode;

**return**;

}

Node current = *root*;

Node parent = **null**;

**while**(**true**){

parent = current;

**if**(id<current.data){

current = current.left;

**if**(current==**null**){

parent.left = newNode;

**return**;

}

}**else**{

current = current.right;

**if**(current==**null**){

parent.right = newNode;

**return**;

}

}

}

}

**public** **void** display(Node root){

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

display(root.left);

System.*out*.print(" " + root.data);

display(root.right);

}

}

**public** **static** **void** main(String arg[]){

BinarySearchTree b = **new** BinarySearchTree();

b.insert(3);b.insert(8);

b.insert(1);b.insert(4);b.insert(6);b.insert(2);b.insert(10);b.insert(9);

b.insert(20);b.insert(25);b.insert(15);b.insert(16);

System.*out*.println("Original Tree : ");

b.display(b.*root*);

System.*out*.println("");

System.*out*.println("Check whether Node with value 4 exists : " + b.find(4));

System.*out*.println("Delete Node with no children (2) : " + b.delete(2));

b.display(*root*);

System.*out*.println("\n Delete Node with one child (4) : " + b.delete(4));

b.display(*root*);

System.*out*.println("\n Delete Node with Two children (10) : " + b.delete(10));

b.display(*root*);

}

}

**class** Node{

**int** data;

Node left;

Node right;

**public** Node(**int** data){

**this**.data = data;

left = **null**;

right = **null**;

}

}

Output :

Original Tree :

1 2 3 4 6 8 9 10 15 16 20 25

Check whether Node with value 4 exists : true

Delete Node with no children (2) : true

1 3 4 6 8 9 10 15 16 20 25

Delete Node with one child (4) : true

1 3 6 8 9 10 15 16 20 25

Delete Node with Two children (10) : true

1 3 6 8 9 15 16 20 25

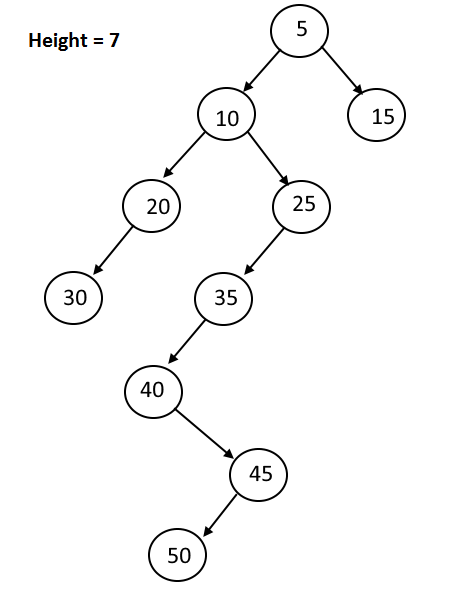
**Algorithms - Find the Height of a Binary Tree**

**Objective:** Given a binary tree, find the height of it

**Input:** A Binary Tree

**Output:** Height of a binary tree

**Example:**

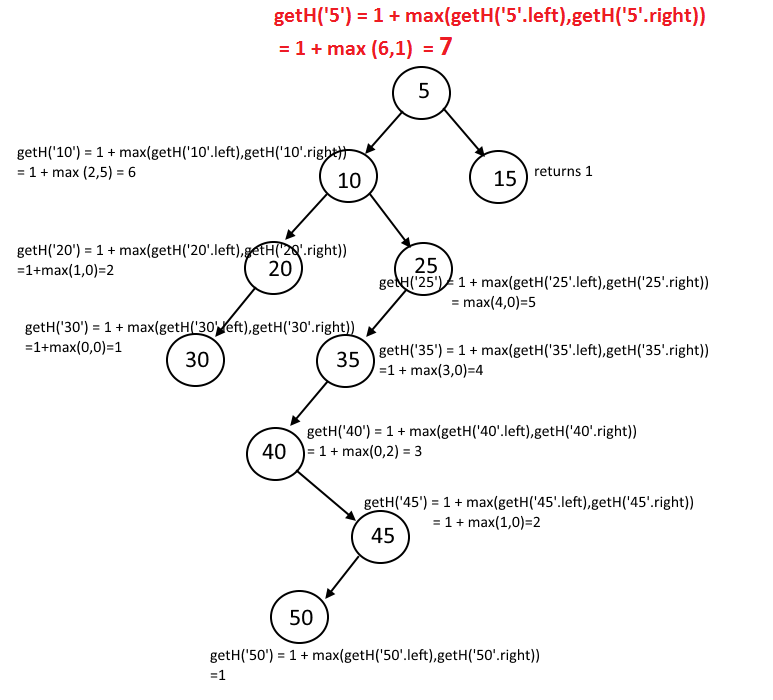
****

**Approach:**

**Recursion:**

* Get the height of left sub tree, say leftHeight
* Get the height of right sub tree, say rightHeight
* Take the Max(leftHeight, rightHeight) and add 1 for the root and return
* Call recursively.

**Time Complexity : O(log n)**

****

**Complete Code:**

**package** TreeHeight;

**public** **class** TreeHeight {

**public** **int** treeHeight(Node root){

**if**(root==**null**)**return** 0;

**return** (1+ Math.*max*(treeHeight(root.left),treeHeight(root.right)));

}

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

Node root = **new** Node(5);

root.left = **new** Node(10);

root.right = **new** Node(15);

root.left.left = **new** Node(20);

root.left.right = **new** Node(25);

root.left.left.left =**new** Node(30);

root.left.right.left = **new** Node(35);

root.left.right.left.left = **new** Node(40);

root.left.right.left.left.right = **new** Node(45);

root.left.right.left.left.right.left = **new** Node(50);

TreeHeight i = **new** TreeHeight();

System.*out*.println(i.treeHeight(root));

}

}

**class** Node{

**int** data;

Node left;

Node right;

**public** Node(**int** data){

**this**.data = data;

**this**.left = **null**;

**this**.right = **null**;

}

}

Output:

Height of the Tree is 7

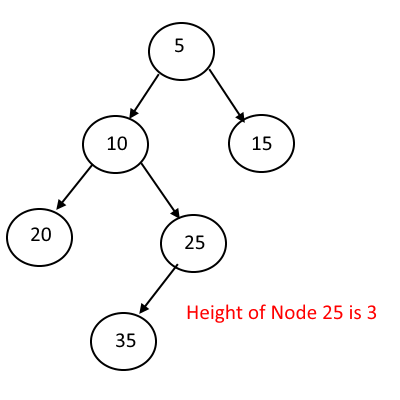
**Algorithms - Get the Height of a Node in a Binary Tree**

**Objective:** Given a binary tree, find the height of a given node in the tree.

**Input:** A Binary Tree and a node

**Output:** Height of a given node in the tree.

**Example:**



**Approach:**

**Recursion:**

* Take a variable called height =0.
* Search for that given node in the tree using recursion.
* Each time you left or right , increase the height by 1.
* Once you found the given node, return the height.
* If till the end you wont find the node, return 0

**Complete Code:**

**package** HeightOfANode;

**public** **class** NodeHeight {

**public** **int** getNodeHeight(Node root, Node x, **int** height){

**if**(root==**null**) **return** 0;

**if**(root==x) **return** height;

//check if the node is present in the left sub tree

**int** level = getNodeHeight(root.left,x,height+1);

//System.out.println(level);

**if**(level!=0) **return** level;

//check if the node is present in the right sub tree

**return** getNodeHeight(root.right,x,height+1);

}

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

Node root = **new** Node(5);

root.left = **new** Node(10);

root.right = **new** Node(15);

root.left.left = **new** Node(20);

Node x = **new** Node(25);

root.left.right = x;

root.left.right.left = **new** Node(35);

NodeHeight i = **new** NodeHeight();

System.*out*.println("Height of the Node " + x.data + " is : " + i.getNodeHeight(root,x,1));

}

}

**class** Node{

**int** data;

Node left;

Node right;

**public** Node(**int** data){

**this**.data = data;

**this**.left = **null**;

**this**.right = **null**;

}

}

Output :

Height of the Node 25 is : 3

**Algorithms - Find whether if a Given Binary Tree is Balanced?**

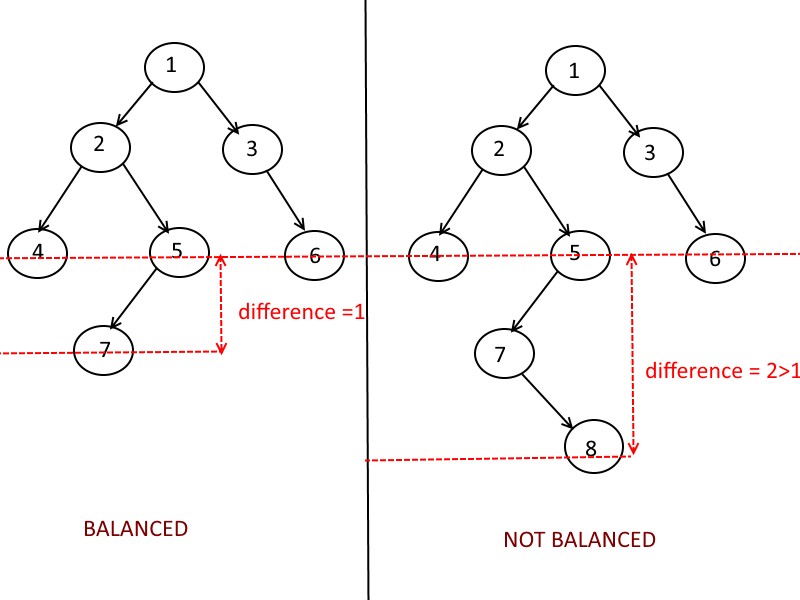
**Objective:** Given a binary tree, Find whether if a Given Binary Tree is Balanced?

**What is balanced Tree:** A balanced tree is a tree in which difference between heights of subtrees of any node in the tree is not greater than one.

**Input:** A Binary Tree

**Output:** True and false based on whether tree is balanced or not.

**Example:**

****

**Approach :**

**Naive Approach:**

**for each node of the tree, get the height of left subtree and right subtree and check the difference , if it is greater than 1, return false.**

**public** **static** **int** getHeight(Node root){

**if**(root==**null**)**return** 0;

**return** (1+ Math.*max*(*getHeight*(root.left), *getHeight*(root.right)));

}

**public** **static** **boolean** isBalancedNaive(Node root){

**if**(root==**null**)**return** **true**;

**int** heightdifference = *getHeight*(root.left)-*getHeight*(root.right);

**if**(Math.*abs*(heightdifference)>1){

**return** **false**;

}**else**{

**return** *isBalancedNaive*(root.left) && *isBalancedNaive*(root.right);

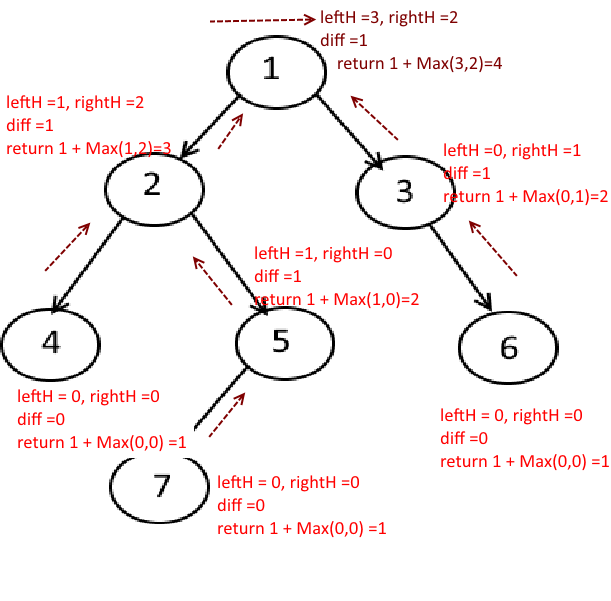
}

}

**Time complexity will be O(N^2) since for every node we will get the height of the tree which means you will come down all the way to leaves.**

**Better Solution:**

* Recursion
* Post order traversal techinque
* Travel all the way down to leaf nodes and then go up.
* while going up, calculate the left and right subtree height.
* If the difference between them is greater than 1, return -1.
* Else Max(leftHeight, rightHeight) +1 .
* Here you wont actually calulate the height of the subtrees by calling function, instead you will store the height at each level and when you go one level up, you add one to it.
* ***So Time complexity is not O(N^2) but it will ne only O(N) but it will have space complexity as O(h) where h is the height of the tree***

******

**Complete Code:**

**package** BalancedTree;

**public** **class** BalancedTree {

**public** **boolean** checkBalance(Node root){

**int** result = isBalanced(root);

**if**(result>0){

**return** **true**;

}**else**{

**return** **false**;

}

}

**public** **int** isBalanced(Node root){

**if**(root==**null**){

**return** 0;

}

**int** leftH = isBalanced(root.left);

**if**(leftH==-1){

**return** -1;

}

**int** rightH = isBalanced(root.right);

**if**(rightH==-1){

**return** -1;

}

**int** diff = leftH-rightH;

**if**(Math.*abs*(diff)>1){

**return** -1;

}

**return** 1 + Math.*max*(leftH, rightH);

}

**public** **static** **void** main(String args[]){

Node root = **new** Node(5);

root.left = **new** Node(10);

root.right = **new** Node(15);

root.left.left = **new** Node(20);

root.left.right = **new** Node(25);

root.right.left = **new** Node(30);

root.right.right = **new** Node(35);

System.*out*.println(" Is Tree Balanced : " + (**new** BalancedTree()).checkBalance(root));

root.right.right.right = **new** Node (40);

root.right.right.right.right = **new** Node (45);

System.*out*.println(" Is Tree Balanced : " + (**new** BalancedTree()).checkBalance(root));

}

}

**class** Node{

**int** data;

Node left;

Node right;

**public** Node(**int** data){

**this**.data = data;

**this**.left = **null**;

**this**.right =**null**;

}

}

**Output:**

Is Tree Balanced : true

Is Tree Balanced : false

**Algorithms - Sorted Array to Binary Search Tree of Minimal Height**

**Objective:** Given a sorted array with unique elements, Create a binary search tree with minimal height.

**Why minimal height is important :**

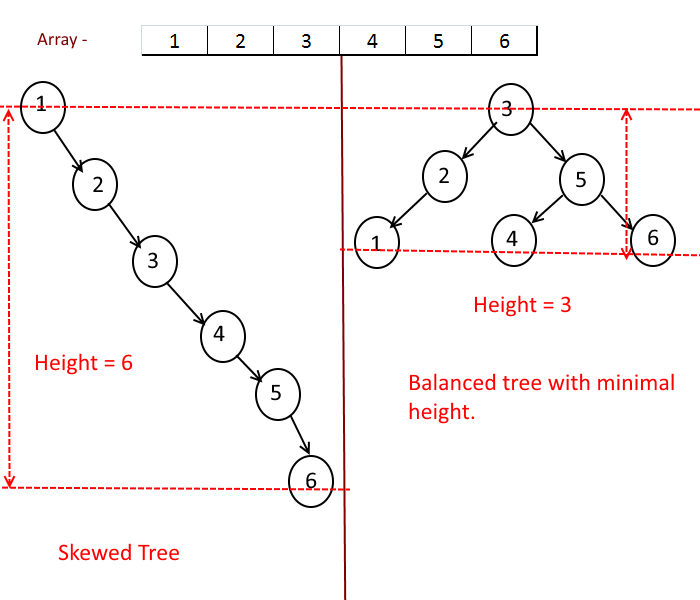
We can do the linear scan to the array and make the first element as root and insert all other elements into the tree but in that case tree will be ***skewed*** , which means all the nodes of the tree will be on the one side of the root so the height of the tree will be equal to the number of elements in the array. So here our objective is to keep the tree balanced as much as possbile.

**What is balanced Tree:** A balanced tree is a tree in which difference between heights of subtrees of any node in the tree is not greater than one. To read more about balanced tree, click here

**Input:** A one dimensional array

**Output:** Binary Search Tree of Minimal Height

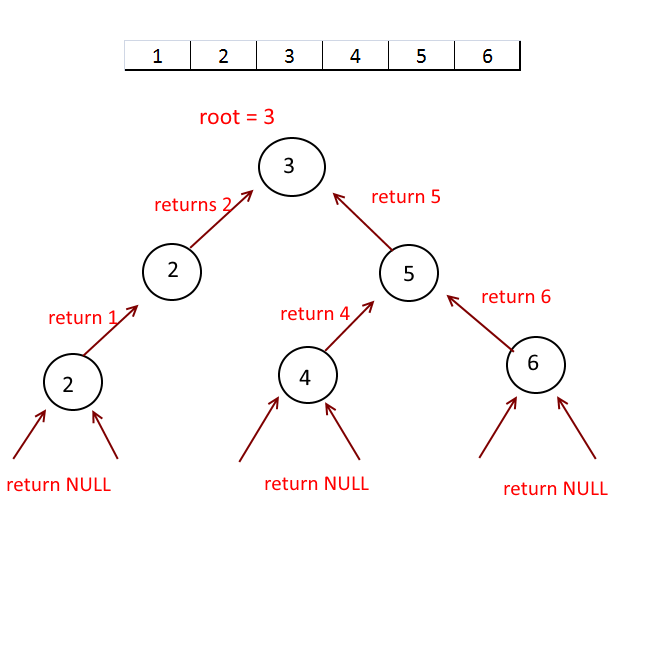
**Example:**

****

**Approach:**

**Recursion:**

* Get the middle of the array
* make it as root. *(By doing this we will ensure that half of the elements of array will be on the left side of the root and half on the right side*.)
* Take the left half of the array, call recursively and add it to root.left.
* Take the right half of the array, call recursively and add it to root.right.
* return root.

****

**Complete Code:**

**public** **class** SortedArrayToBST {

**public** BSTNode convert(**int** [] arrA, **int** start, **int** end){

**if**(start>end){

**return** **null**;

}

**int** mid = (start + end)/2;

BSTNode root = **new** BSTNode(arrA[mid]);

root.left = convert(arrA, start, mid-1);

root.right =convert(arrA, mid+1, end);

**return** root;

}

**public** **void** displayTree(BSTNode root){

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

displayTree(root.left);

System.*out*.print(" " + root.data);

displayTree(root.right);

}

}

**public** **static** **void** main(String args[]){

**int** [] arrA = {2,3,6,7,8,9,12,15,16,18,20};

SortedArrayToBST s = **new** SortedArrayToBST();

BSTNode x = s.convert(arrA, 0, arrA.length-1);

System.*out*.println("Tree Display : ");

s.displayTree(x);

}

}

**class** BSTNode{

**int** data;

BSTNode left;

BSTNode right;

**public** BSTNode(**int** data){

**this**.data = data;

left = **null**;

right = **null**;

}

}

Output:

Tree Display :

2 3 6 7 8 9 12 15 16 18 20

**Algorithms - In a Binary Tree, Create Linked Lists of all the nodes at each depth.**

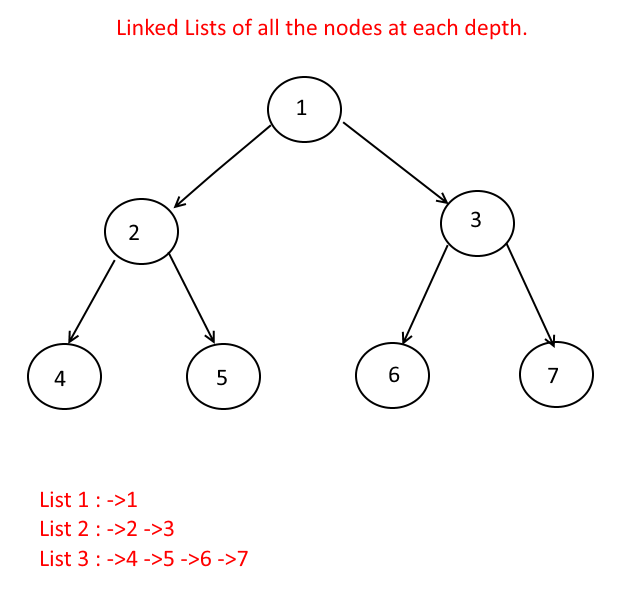
**Objective:** Given a Binary tree create Linked Lists of all the nodes at each depth , say if the tree has height k then create k linked lists.

**NOTE : This problem is very similar "Print binary tree, each level in one line"**

**Input:** A binary tree

**Output:** K linked lists if the height of tree is k. Each linked list will have all the nodes of each level.

**Example:**

****

**Approach:**

**Recursion:**

* **Create a ArrayList of Linked List Nodes.**
* **Do the level order traversal using queue(Breadth First Search). Click here to know about how to level order traversal.**
* **For getting all the nodes at each level, before you take out a node from queue, store the size of the queue in a variable, say you call it as levelNodes.**
* **Now while levelNodes>0, take out the nodes and print it and add their children into the queue. add these to a linked list**
* **After this while loop put a line break and create a new linked list**

ArrayList<ListNode> al = new ArrayList<ListNode>();

**while**(!q.isEmpty()){

levelNodes = q.size();

ListNode head = **null**;

ListNode curr = **null**;

**while**(levelNodes>0){

Node n = (Node)q.remove();

ListNode ln = **new** ListNode(n.data);

**if**(head==**null**){

head = ln;

curr = ln;

}**else**{

curr.next = ln;

curr = curr.next;

}

**if**(n.left!=**null**) q.add(n.left);

**if**(n.right!=**null**) q.add(n.right);

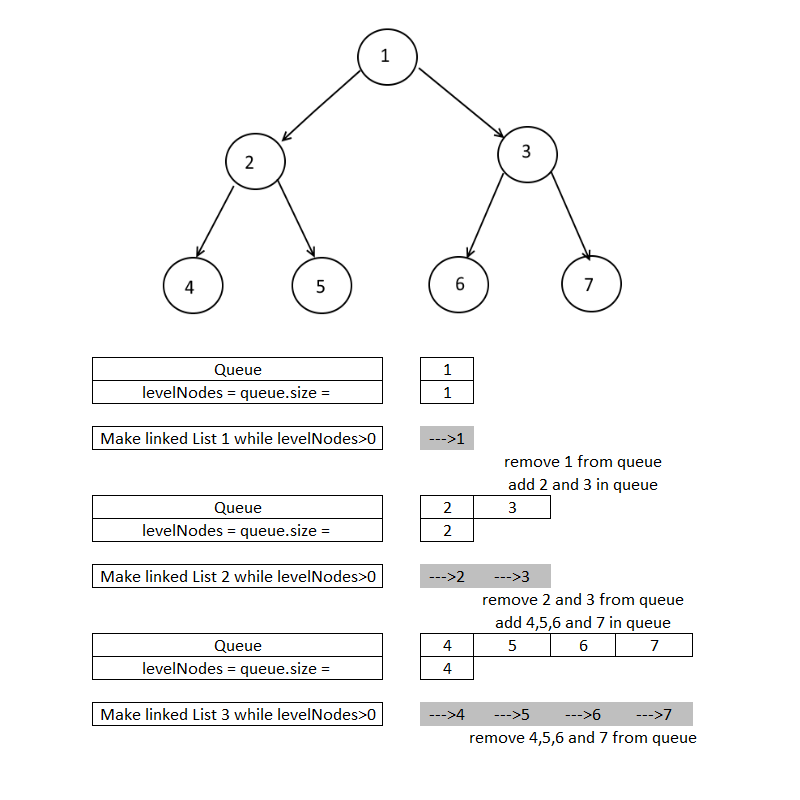
levelNodes--;

}

al.add(head);

}

* ***Since we had taken the queue size before we add new nodes, we will get the count at each level and after printing this count, put a line break, see the example below***
* **Time Complexity : O(N)**

******

**Complete Code:**

**package** LinkedListsAtEachDepth\_Tree;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.LinkedList;

**import** java.util.Queue;

**public** **class** ListAtDepthBtree {

ArrayList<ListNode> al = **new** ArrayList<ListNode>();

**public** **void** levelOrder(Node root){

**int** h = height(root);

**for**(**int** i=1;i<=h;i++){

printLevels(root,i);

System.*out*.println("");

}

}

**public** **void** printLevels(Node root, **int** h){

**if**(root==**null**) **return**;

**if**(h==1) System.*out*.print(" " + root.data);

**else**{

printLevels(root.left,h-1);

printLevels(root.right,h-1);

}

}

**public** **int** height(Node root){

**if** (root==**null**) **return** 0;

**return** 1 + Math.*max*(height(root.left),height(root.right));

}

**public** **void** levelOrderQueue(Node root){

Queue q = **new** LinkedList();

**int** levelNodes =0;

**if**(root==**null**) **return**;

q.add(root);

**while**(!q.isEmpty()){

levelNodes = q.size();

ListNode head = **null**;

ListNode curr = **null**;

**while**(levelNodes>0){

Node n = (Node)q.remove();

ListNode ln = **new** ListNode(n.data);

**if**(head==**null**){

head = ln;

curr = ln;

}**else**{

curr.next = ln;

curr = curr.next;

}

**if**(n.left!=**null**) q.add(n.left);

**if**(n.right!=**null**) q.add(n.right);

levelNodes--;

}

al.add(head);

}

display(al);

}

**public** **void** display(ArrayList al){

Iterator<ListNode> it = al.iterator();

**while**(it.hasNext()){

ListNode head = it.next();

//System.out.print("->" + head.data);

**while**(head!=**null**){

System.*out*.print("->" + head.data);

head = head.next;

}

System.*out*.println("");

}

}

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

Node root = **new** Node(5);

root.left = **new** Node(10);

root.right = **new** Node(15);

root.left.left = **new** Node(20);

root.left.right = **new** Node(25);

root.right.left = **new** Node(30);

root.right.right = **new** Node(35);

ListAtDepthBtree i = **new** ListAtDepthBtree();

//i.levelOrder(root);

i.levelOrderQueue(root);

}

}

**class** Node{

**int** data;

Node left;

Node right;

**public** Node(**int** data){

**this**.data = data;

**this**.left = **null**;

**this**.right =**null**;

}

}

**class** ListNode{

**int** data;

ListNode next;

**public** ListNode(**int** data){

**this**.data = data;

**this**.next = **null**;

}

}

Output:

->5

->10->15

->20->25->30->35

**Algorithms - Level Order Traversal, Print each level in separate line.**

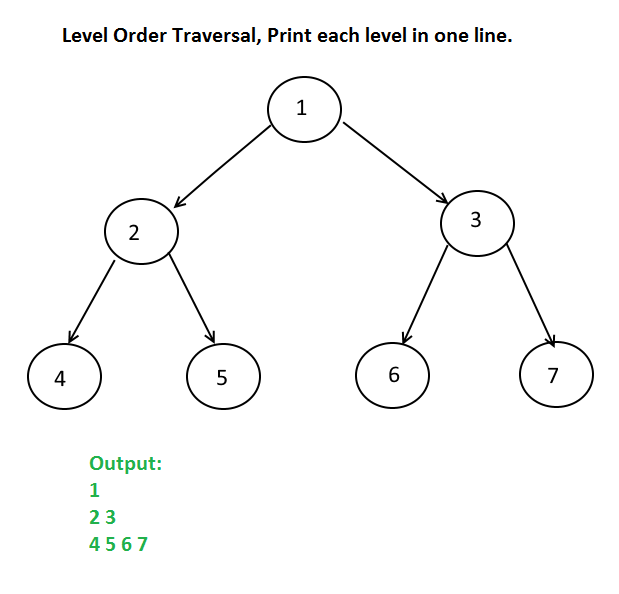
**Objective:** Given a Binary tree , Print each level of a tree in separate line.

**NOTE : This problem is very similar " Create Linked Lists of all the nodes at each depth "**

**Input:** A binary tree

**Output: Each level of binary tree, in one line**

**Example:**

****

**Approach:**

**Naive Approach:**

1. Get the height of the tree.
2. Put a for loop for each level in tree.
3. for each level in step 2, do pre order traversal and print only when height matches to the level.
4. Look at the code for better explanation

**Time Complexity : O(N^2) - because each level you are traversing the entire tree.**

**Better Solution :**

* Create a ArrayList of Linked List Nodes.
* Do the level order traversal using queue(Breadth First Search). Click here to know about how to level order traversal.
* For getting all the nodes at each level, before you take out a node from queue, store the size of the queue in a variable, say you call it as levelNodes.
* Now while levelNodes>0, take out the nodes and print it and add their children into the queue.
* After this while loop put a line break.

**Time Complexity : O(N)**

**while**(!q.isEmpty()){

levelNodes = q.size();

**while**(levelNodes>0){

Node n = (Node)q.remove();

System.*out*.print(" " + n.data);

**if**(n.left!=**null**) q.add(n.left);

**if**(n.right!=**null**) q.add(n.right);

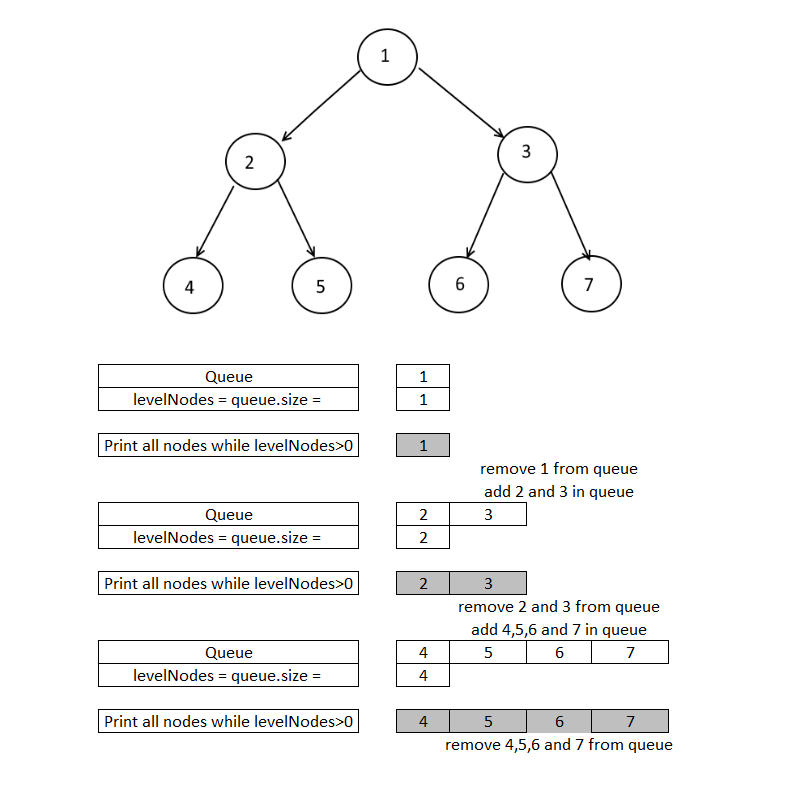
levelNodes--;

}

System.*out*.println("");

}

* ***Since we had taken the queue size before we add new nodes, we will get the count at each level and after printing this count, put a line break, see the example below***

****

**Complete Code:**

package LevelOrderTraversalByLine;

import java.util.LinkedList;

import java.util.Queue;

public class LevelOrderTraversal {

public void levelOrderNaiveApproach(Node root){

int h = height(root);

for(int i=1;i<=h;i++){

printLevels(root,i);

System.out.println("");

}

}

public void printLevels(Node root, int h){

if(root==null) return;

if(h==1) System.out.print(" " + root.data);

else{

printLevels(root.left,h-1);

printLevels(root.right,h-1);

}

}

public int height(Node root){

if (root==null) return 0;

return 1 + Math.max(height(root.left),height(root.right));

}

public void levelOrderQueue(Node root){

Queue q = new LinkedList();

int levelNodes =0;

if(root==null) return;

q.add(root);

while(!q.isEmpty()){

levelNodes = q.size();

while(levelNodes>0){

Node n = (Node)q.remove();

System.out.print(" " + n.data);

if(n.left!=null) q.add(n.left);

if(n.right!=null) q.add(n.right);

levelNodes--;

}

System.out.println("");

}

}

public static void main (String[] args) throws java.lang.Exception

{

Node root = new Node(5);

root.left = new Node(10);

root.right = new Node(15);

root.left.left = new Node(20);

root.left.right = new Node(25);

root.right.left = new Node(30);

root.right.right = new Node(35);

LevelOrderTraversal i = new LevelOrderTraversal();

System.out.println(" Output by Naive Approach : ");

i.levelOrderNaiveApproach(root);

System.out.println(" Output by Better Approach : ");

i.levelOrderQueue(root);

}

}

class Node{

int data;

Node left;

Node right;

public Node(int data){

this.data = data;

this.left = null;

this.right =null;

}

}

Output :

Output by Naive Approach :

5

10 15

20 25 30 35

Output by Better Approach :

5

10 15

20 25 30 35

**Algorithms - Inorder Successor in Binary Search Tree Using Parent link**

**Objective:** Given a Binary Seach tree in which every node has a link to its parent, find the inorder successor of a node.

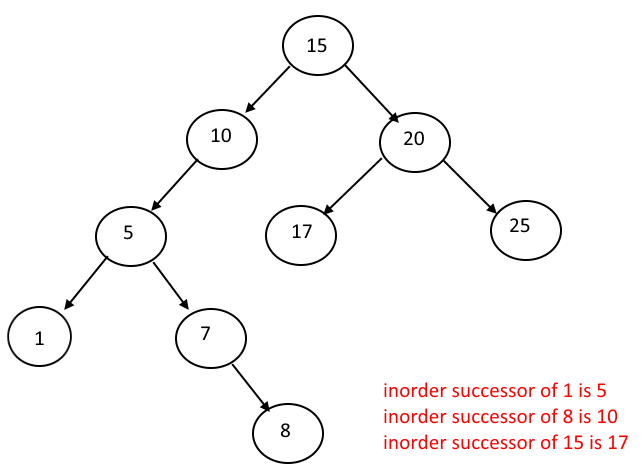
***What is Inorder Successor:*** *Inorder successor of a node is the next node in the inorder traversal of the tree. For the last node in a tree, inorder successor will be NULL*

**Similar Problems : Inorder Successor in Binary Search Tree with No Parent link**

**Input:** A binary search tree with nodes liked to its parents, a node x

**Output: The inorder successor of node x.**

**Example:**



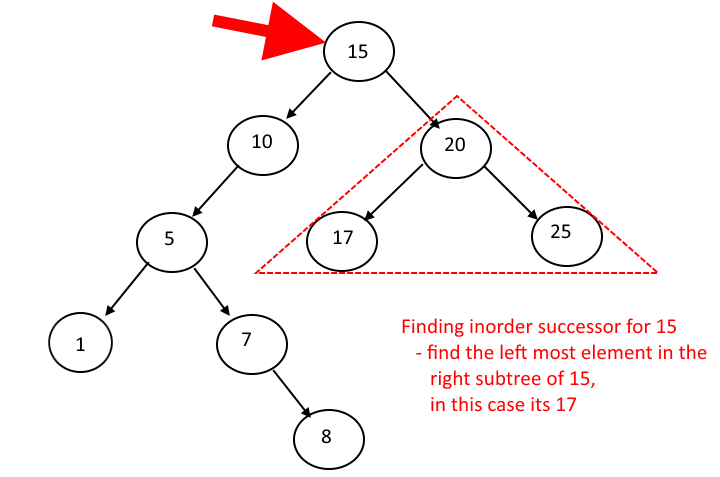
**Approach:**

**Time Complexity : O(h) , h - height of the tree**

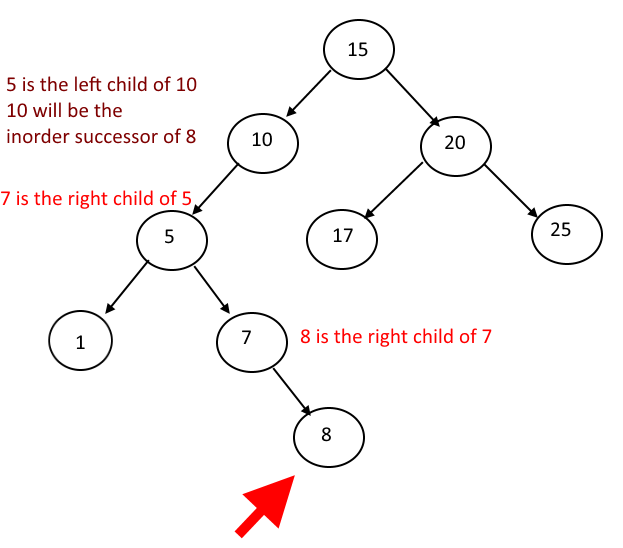
**There will be 3 cases to solve this problem**

Say the node for whose inorder successor needs to be find is **x.**

***Case 1 : If the x has a right child then its inorder successor will the left most element in the right sub tree of x.***

******

***Case 2: If the x doesnt have a right child then its inorder successor will the one of its ancestors, using parent link keep travelling up till you get the node which is the left child of its parent. Then this parent node will be the inorder successor.***

******

***Case 3: if x is the right most node in the tree then its inorder successor will be NULL.***

**Complete Code:**

**package** InorderSuccessorInBST;

**public** **class** InOrderSuccessorWithParentPointer {

**public** Node findInOrderSuccessor(Node root, Node x){

//if the right child of x is not null then in-Order successor will the left most node in

//the right sub tree of x.

**if**(x.right!=**null**){

**return** leftMostNode(x.right);

}

Node parent = x.parent;

**while**(parent!=**null** && x==parent.right){

x = parent;

parent = parent.parent;

}

**return** parent;

}

**public** Node leftMostNode(Node x){

**while**(x.left!=**null**){

x = x.left;

}

**return** x;

}

**public** **void** display(Node root){

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

display(root.left);

System.*out*.print(" " + root.data);

display(root.right);

}

}

**public** **static** **void** main(String args[]){

Node root = **new** Node(10);

root.left = **new** Node(5);

root.left.parent = root;

root.right = **new** Node(15);

root.right.parent = root;

root.left.left = **new** Node(3);

root.left.left.parent = root.left;

root.right.left = **new** Node(17);

root.right.left.parent = root.right;

Node a = **new** Node(7);

root.left.right = a;

root.left.right.parent = root.left;

InOrderSuccessorWithParentPointer b = **new** InOrderSuccessorWithParentPointer();

System.*out*.print(" Tree : ");

b.display(root);

System.*out*.println();

Node x = b.findInOrderSuccessor(root, a);

**if**(x!=**null**){

System.*out*.println("InOrder Successor of " + a.data + " is " + x.data);

}**else**{

System.*out*.println("InOrder Successor of " + a.data + " is NULL");

}

x = b.findInOrderSuccessor(root, root);

**if**(x!=**null**){

System.*out*.println("InOrder Successor of " + root.data + " is " + x.data);

}**else**{

System.*out*.println("InOrder Successor of " + root.data + " is NULL");

}

}

}

**class** Node{

**int** data;

Node left;

Node right;

Node parent;

**public** Node(**int** data){

**this**.data = data;

**this**.left = **null**;

**this**.right = **null**;

**this**.parent = **null**;

}

}

Output:

Tree : 3 5 7 10 17 15

InOrder Successor of 7 is 10

InOrder Successor of 10 is 17

**Algorithms - Inorder Successor in Binary Search Tree without Using Parent link**

**Objective:** Given a Binary Search tree, find the inorder successor of a node.

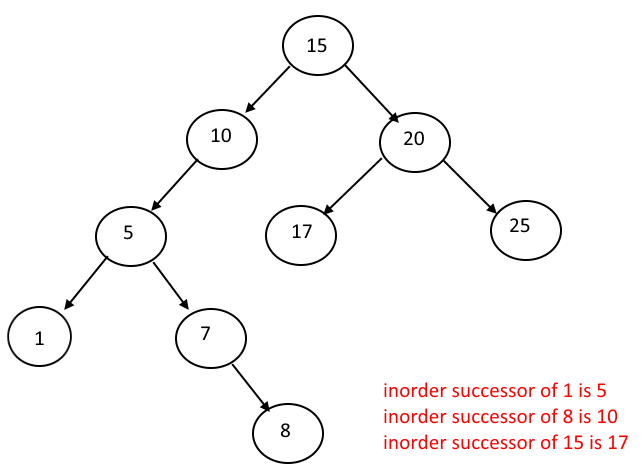
***What is Inorder Successor: Inorder successor of a node is the next node in the inorder traversal of the tree. For the last node in a tree, inorder successor will be NULL***

**Similar Problems : Inorder Successor in Binary Search Tree with parent link**

**Input:** A binary search tree, a node x

**Output: Inorder successor of node x.**

**Example:**



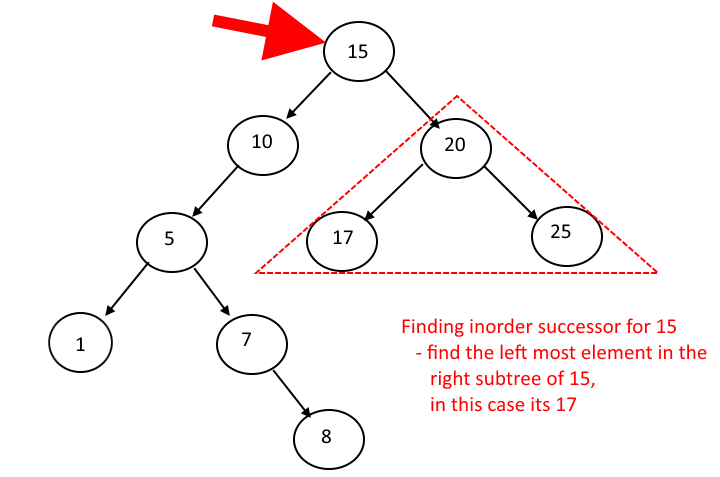
**Approach:**

**Time Complexity : O(h) , h - height of the tree**

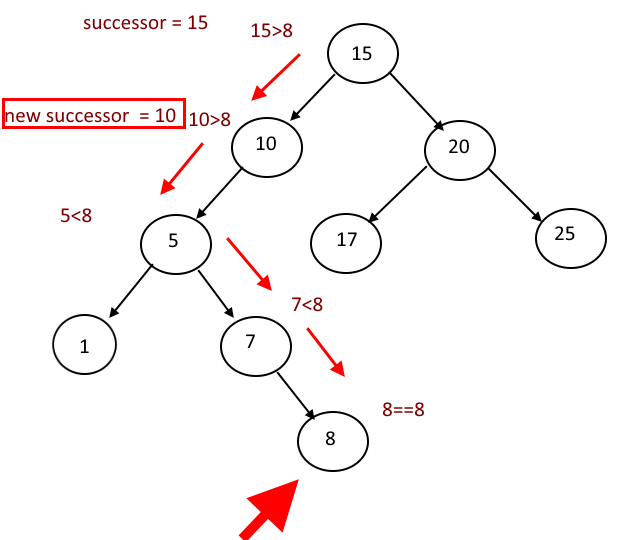
**There will be 3 cases to solve this problem**

Say the node for which inorder successor needs to be find is **x.**

***Case 1 : If the x has a right child then its inorder successor will the left most element in the right sub tree of x.***

******

***Case 2:*** *If the x doesnt have a right child then its inorder successor will the one of its ancestors,* ***use the binary search technique to find the node x, start from the root, if root is bigger than the x then go left, if root is less than x, go right. while travelling whenever you go left , store the node and call it successor***

******

***Case 3: if x is the right most node in the tree then its inorder successor will be NULL.***

**Complete Code:**

**package** InorderSuccessorInBST;

**public** **class** InOrderSuccessor {

**public** TreeNode findInOrderSuccessor(TreeNode root, TreeNode x){

//if the right child of x is not null then in-Order successor will the left most node in

//the right sub tree of x.

**if**(x.right!=**null**){

**return** leftMostTreeNode(x.right);

}

TreeNode successor = **null**;

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

**if**(root.data>x.data){

successor = root;

root = root.left;

}**else** **if**(root.data<x.data){

root = root.right;

}**else**{

**return** successor;

}

}

**return** **null**;

}

**public** **void** display(TreeNode root){

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

display(root.left);

System.*out*.print(" " + root.data);

display(root.right);

}

}

**public** **static** **void** main(String args[]){

TreeNode root = **new** TreeNode(10);

root.left = **new** TreeNode(5);

root.right = **new** TreeNode(15);

root.left.left = **new** TreeNode(3);

root.right.left = **new** TreeNode(17);

TreeNode a = **new** TreeNode(7);

root.left.right = a;

InOrderSuccessor b = **new** InOrderSuccessor();

System.*out*.print(" Tree : ");

b.display(root);

System.*out*.println();

TreeNode x = b.findInOrderSuccessor(root, a);

**if**(x!=**null**){

System.*out*.println("InOrder Successor of " + a.data + " is " + x.data);

}**else**{

System.*out*.println("InOrder Successor of " + a.data + " is NULL");

}

x = b.findInOrderSuccessor(root, root);

**if**(x!=**null**){

System.*out*.println("InOrder Successor of " + root.data + " is " + x.data);

}**else**{

System.*out*.println("InOrder Successor of " + root.data + " is NULL");

}

}

**public** TreeNode leftMostTreeNode(TreeNode x){

**while**(x.left!=**null**){

x = x.left;

}

**return** x;

}

}

**class** TreeNode{

**int** data;

TreeNode left;

TreeNode right;

**public** TreeNode(**int** data){

**this**.data = data;

**this**.left = **null**;

**this**.right = **null**;

}

}

Output:

Tree : 3 5 7 10 17 15

InOrder Successor of 7 is 10

InOrder Successor of 10 is 17

**Algorithms - Inorder Successor in Binary Tree**

**Objective:** Given a Binary tree (Not binary Search Tree ), find the inorder successor of a node.

***What is Inorder Successor:*** *Inorder successor of a node is the next node in the inorder traversal of the tree. For the last node in a tree, inorder successor will be NULL*

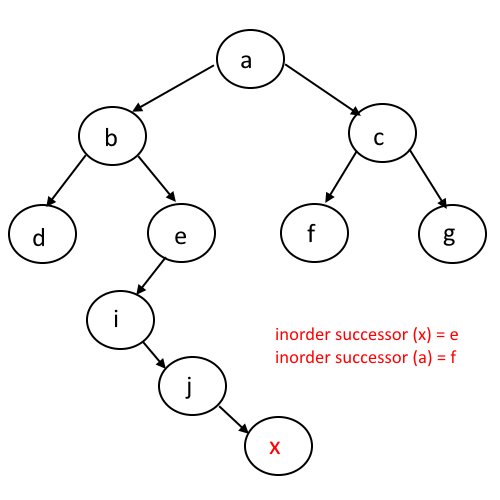
**Similar Problems : Inorder Successor in Binary Search Tree with parent link**

**Inorder Successor in Binary Search Tree without parent link**

**Input:** A binary tree, a node x

**Output: Inorder successor of node x.**

**Example:**

****

**Approach:**

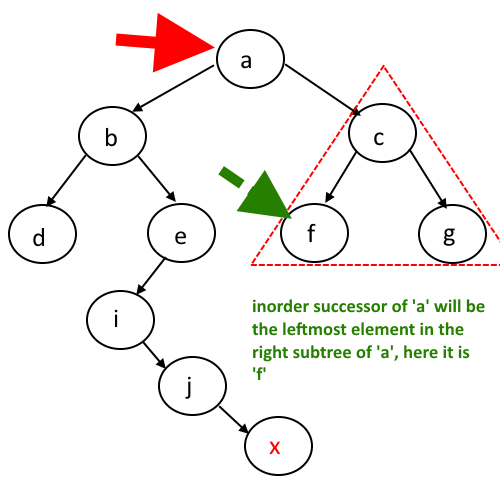
**NOTE:** since it's not a binary search tree, we cannot use binary search technique to reach to the node. we need to travel all the nodes in order to find the node for which we need to find the inorder successor.

How to search for a path of any node in binary tree.

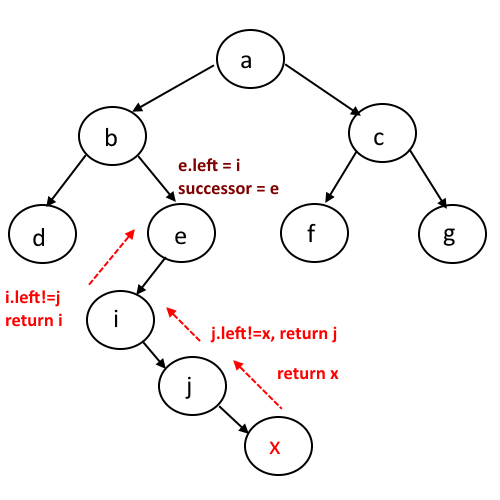
**There will be 3 cases to solve this problem**

Say the node for which inorder successor needs to be find is **x.**

***Case 1 : If the x has a right child then its inorder successor will the left most element in the right sub tree of x.***

******

1. Start from the root and compare it with x, if matched then we have found the node.
2. else go left and right.
3. recursively do step 2 and 3 till you find the node x.
4. Now when you have found the node, stop the recursion.
5. Now recursion will backtrack to the root, every recursive call will return the node itself (say it will be stored in ***n***) so when it backtracked to its parent which will be root now, check whether parent.left = n, if not , keep going up.



***Case 3: if x is the right most node in the tree then its inorder successor will be NULL.***

Complete Code:

**package** InOrderSuccessorBinaryTree;

**public** **class** InOrderSuccessorBinaryTree {

**public** **static** TreeNode *n* = **null**;

**public** **static** Boolean *nodeFound* = **false**;

**public** TreeNode inOrderSuccBiTree(TreeNode root, TreeNode x){

*nodeFound* = **false**;

**if**(x.right!=**null**){

TreeNode temp = leftMostTreeNode(x.right);

System.*out*.println("The In Order Successor for '" + x.data + "' is "+ temp.data );

**return** **null**;

}

**return** findRecur(root, x);

}

**public** **void** display(TreeNode root){

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

display(root.left);

System.*out*.print(" " + root.data);

display(root.right);

}

}

**public** TreeNode findRecur(TreeNode root, TreeNode x){

**if**(root==**null**) **return** **null**;

**if**(root==x||(*n* = findRecur(root.left,x))!=**null**||(*n*=findRecur(root.right,x))!=**null**){

*nodeFound* = **true**;

**if**(*n*!=**null**){

**if**(root.left==*n*){

System.*out*.println("The In Order Successor for '" + x.data + "' is "+ root.data );

**return** **null**;

}

}

**return** root;

}

**return** **null**;

}

**public** TreeNode leftMostTreeNode(TreeNode x){

**while**(x.left!=**null**){

x = x.left;

}

**return** x;

}

**public** **static** **void** main(String args[]){

TreeNode root = **new** TreeNode('a');

root.left = **new** TreeNode('b');

root.right = **new** TreeNode('c');

root.left.left = **new** TreeNode('d');

root.left.right = **new** TreeNode('e');

TreeNode x = **new** TreeNode('x');

root.left.right.left = **new** TreeNode('i');

root.left.right.left.right = **new** TreeNode('j');

root.left.right.left.right.right = x;

root.right.left = **new** TreeNode('f');

root.right.right = **new** TreeNode('g');

InOrderSuccessorBinaryTree i = **new** InOrderSuccessorBinaryTree();

System.*out*.print(" Tree : ");

i.display(root);

System.*out*.println();

i.inOrderSuccBiTree(root, x);

**if**(!*nodeFound*){

System.*out*.println("InOrder Successor of " + x.data + " is NULL");

}

i.inOrderSuccBiTree(root, root);

**if**(!*nodeFound*){

System.*out*.println("InOrder Successor of " + root.data + " is NULL");

}

i.inOrderSuccBiTree(root, root.right.right);

**if**(!*nodeFound*){

System.*out*.println("InOrder Successor of " + root.right.right.data + " is NULL");

}

}

}

**class** TreeNode{

**char** data;

TreeNode left;

TreeNode right;

**public** TreeNode(**char** data){

**this**.data = data;

**this**.left = **null**;

**this**.right = **null**;

}

}

Output :

Tree : d b i j x e a f c g

The In Order Successor for 'x' is e

The In Order Successor for 'a' is f

InOrder Successor of a is NULL

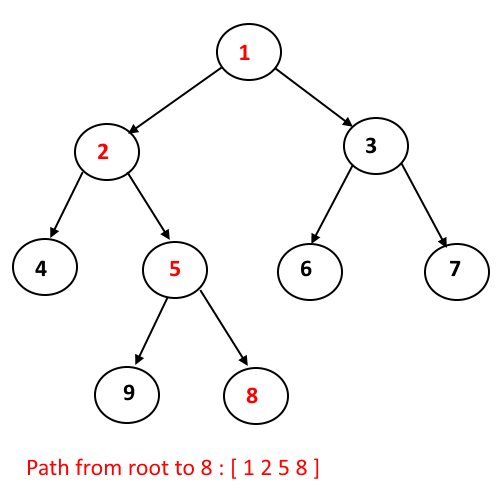
**Algorithms - Print a path from Root to Node in Binary Tree**

**Objective:** Given a Binary tree (Not binary Search Tree ), Print a path from root to a given node.

**Input:** A binary tree, a node x

**Output:** Path from root to a given node

Example :

****

**Approach :**

*since it's not a binary search tree, we cannot use binary search technique to reach to the node. we need to travel all the nodes in order to find the node*

1. Start from the root and compare it with x, if matched then we have found the node.
2. Else go left and right.
3. Recursively do step 2 and 3 till you find the node x.
4. Now when you have found the node, stop the recursion.
5. Now while going back to the root while back tracking, store the node values in the ArrayList.
6. Reverse the ArrayList and print it.

Time Complexity : O(n)

**Complete Code:**

package PrintPathRootToNode;

import java.util.ArrayList;

import java.util.Collections;

public class PrintPathRootToNode {

public static ArrayList path;

public Boolean printPath(Node root, Node dest){

if(root==null) return false;

if(root==dest||printPath(root.left,dest)||printPath(root.right,dest)){

//System.out.print(" " + root.data);

path.add(root.data);

return true;

}

return false;

}

public static void main (String[] args) throws java.lang.Exception

{

Node root = new Node(1);

root.left = new Node (2);

root.right = new Node (3);

Node n1 = new Node (4);

root.left.left = n1;

root.left.right = new Node (5);

Node n2 = new Node (8);

root.left.right.right = n2;

root.left.right.left = new Node (7);

PrintPathRootToNode i = new PrintPathRootToNode();

path = new ArrayList();

i.printPath(root,n2);

Collections.reverse(path);

System.out.println(" Path " + path);

}

}

class Node{

int data;

Node left;

Node right;

public Node (int data){

this.data = data;

left = null;

right = null;

}

}

Output :

Path [1, 2, 5, 8]

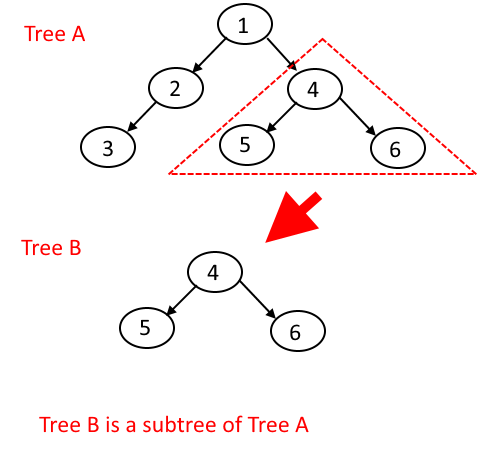
**Algorithms - Given two binary trees, check if one binary tree is a subtree of another**

**Objective:** Given two binary trees, check if one binary tree is a subtree of another

**Input:** Two binary trees

**Output:** True or false based on whether one tree is subtree of another

Example :

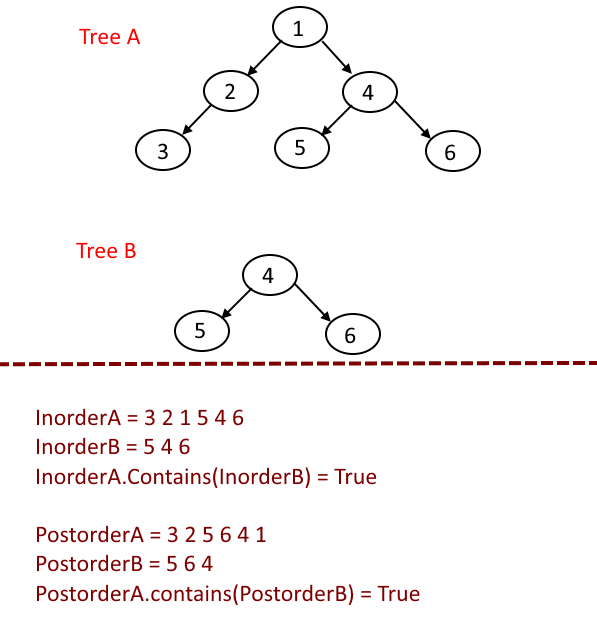


**Approach:**

***We know that to indentity any binary tree can be represented as the combination of either inorder and preorder traversal OR inorder and postorder traversal OR inorder and Level order traversal.***

* Say our trees are A and B.
* Do the inorder traveral of treeA and store it in a String say inorderA.
* Do the inorder traveral of treeB and store it in a String say inorderB.
* Do the postorder traveral of treeA and store it in a String say postorderA.
* Do the postorder traveral of treeB and store it in a String say postorderB.
* Check if inorderA contains inorderB AND postorderA contains postorderB then it means treeB is a subtree of treeA.

Time Complexity : O(n)



Complete Code:

**package** BinarytreeisSubTreeOfAnother;

**public** **class** BinarytreeisSubTreeOfAnother {

//store the inOrder and postorder traversal for both the array,

//if one array is the sub array of another array, that means one tree is the subtree of another one

**public** String inOrder(Node root, String i){

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

**return** inOrder(root.left,i) + " " + root.data + " " +inOrder(root.right,i);

}

**return** "";

}

**public** String postOrder(Node root, String i){

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

**return** postOrder(root.left,i) + " " + postOrder(root.right,i) + " " + root.data;

}

**return** "";

}

**public** **boolean** checkSubtree(Node rootA, Node rootB){

String inOrderA = inOrder(rootA,"");

String inOrderB = inOrder(rootB,"");

String postOrderA = postOrder(rootA,"");

String postOrderB = postOrder(rootB,"");

**return** (inOrderA.toLowerCase().contains(inOrderB.toLowerCase()) && postOrderA.toLowerCase().contains(postOrderB.toLowerCase()));

}

**public** **void** display(Node root){

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

display(root.left);

System.*out*.print(" " + root.data);

display(root.right);

}

}

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

Node rootA = **new** Node(1);

rootA.left = **new** Node(2);

rootA.right = **new** Node(4);

rootA.left.left = **new** Node(3);

rootA.right.right = **new** Node(6);

rootA.right.left = **new** Node(5);

Node rootB = **new** Node(4);

rootB.left = **new** Node(5);

rootB.right = **new** Node(6);

BinarytreeisSubTreeOfAnother i = **new** BinarytreeisSubTreeOfAnother();

System.*out*.print(" Tree A : ");

i.display(rootA);

System.*out*.println();

System.*out*.print(" Tree B : ");

i.display(rootB);

System.*out*.println();

System.*out*.println(i.checkSubtree(rootA,rootB));

}

}

**class** Node{

**int** data;

Node left;

Node right;

**public** Node(**int** data){

**this**.data = data;

**this**.left = **null**;

**this**.right = **null**;

}

}

Output:

Tree A : 3 2 1 5 4 6

Tree B : 5 4 6

true

**Algorithms - Find the Size of the Binary Tree**

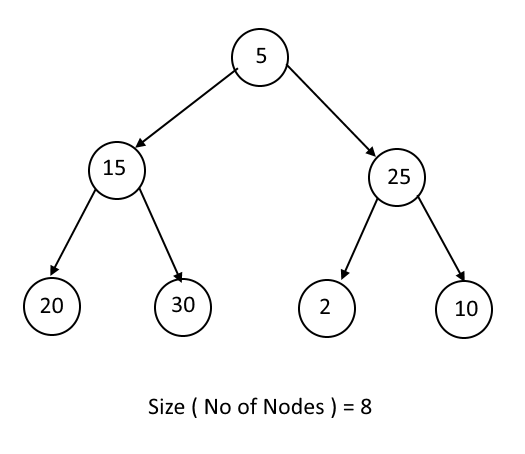
**Objective:** Given a Binary tree, Find the size of the tree.

Note : Size of the tree is number of nodes in the tree

**Input:** A Binary Tree.

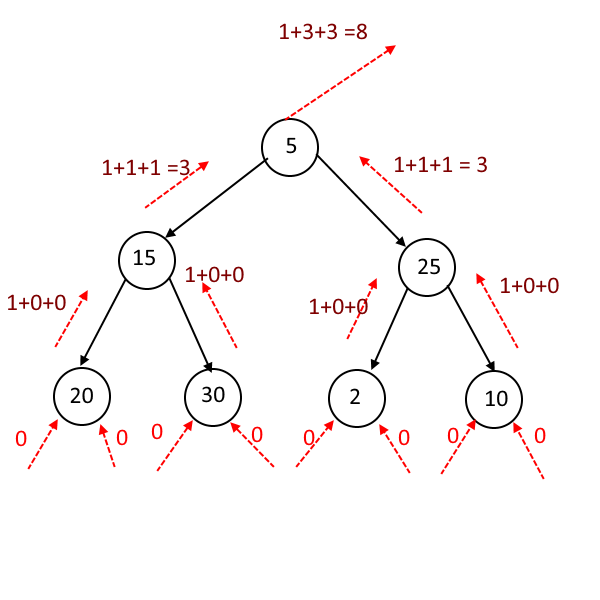
**Output:** Size of the tree.

Example :



**Approach :**

* Very Simple solution
* Start from the root.
* Size = 1 (for the root) + Size Of left Sub-Tree + Size Of right Sub-Tree
* solve the left sub-tree and right sub-tree recursively.



**Time Complexity : O(n)**

Complete Code:

**package** SizeofTree;

**public** **class** SizeofTree {

**public** **int** getSize(Node root){

**if**(root==**null**){

**return** 0;

}

**return** 1 + getSize(root.left) + getSize(root.right);

}

**public** **static** **void** main(String args[]){

Node root = **new** Node(5);

root.left = **new** Node(15);

root.right = **new** Node(25);

root.left.left = **new** Node(20);

root.left.right = **new** Node(30);

root.right.left = **new** Node(2);

root.right.right = **new** Node(10);

SizeofTree t = **new** SizeofTree();

System.*out*.println("Size of the Tree is : " + t.getSize(root));

}

}

**class** Node{

**int** data;

Node left;

Node right;

**public** Node(**int** data){

**this**.data = data;

**this**.left = **null**;

**this**.right = **null**;

}

}

Output:

Size of the Tree is : 7

**Algorithms - Determine whether given binary tree is binary search tree(BST) or not**

**Objective:** Given a Binary tree, find out whether its binary seach tree or not.

**Input:** A Binary Tree.

**Output:** True or false based on whether tree is BST ot not.

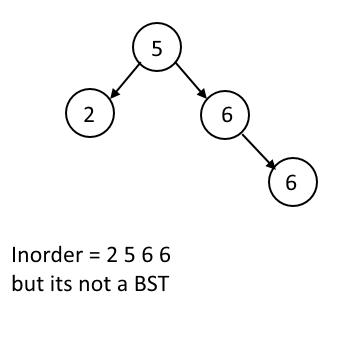
**Approach:**

**Method 1 : If tree doesn't have duplicates**

* Do the inorder traversal of the given binary tree.
* check if the previously visited node is less than the current node value.
* If yes, then return true
* Else return false.

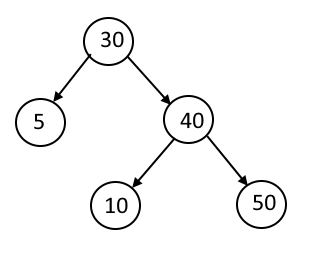
Time Complexity : O(n)

The above method works only when tree doesn't have any duplicates. see figure,



**Method 2: The Max/Min Solution :**

When we talk about binary search tree we talk about one property i.e leftChild.data <=root.data<rightChild, but checking alone this property at every node is not gonna work out, want to know why, see this example



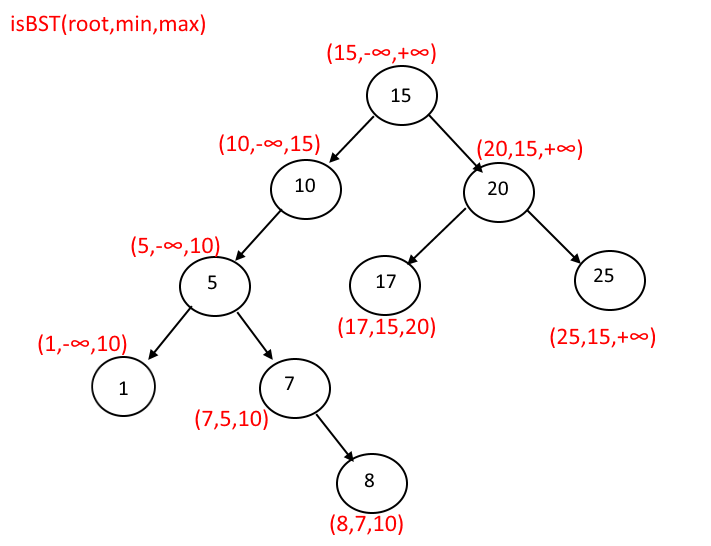
Now in the above example you can see that when you validate the Node 40, 10<=40<50 so it will return true nad when you validate the Node 30, 5<=30<40, will return true but as you can see that this tree is not BST since 10 is smaller than 30 so it should be on the left of the 30. Actually all the nodes less than 30 should be on the left and all the nodes greater than 30 should be on the right.

**How would you achieve that???**

Your root value can have any value between -∞ to + ∞. When you validate the right child of 30, it can take any value between 30 and + ∞. When you validate the left child of 30, it can take any value between - ∞ and 30. likewsie when you validate the left child of 40, it can take any value between 30 and 40.

So the idea is *Pass the minimum and maximum values between which the node's value must lie*.

we start with the range minimum = Integer.MIN\_VALUE and maximum = Interger.MAX\_VALUE, so when checking the left node of 30(root), minimum will be = Integer.MIN\_VALUE and maximum = 30, so on. See the figure for better understanding.



**Complete Code for Both the methods:**

**package** isBST;

**public** **class** isBST {

**public** **static** Node *prevNode* = **null**;

//method 1: do inOrder and check if it is in ascending order

**public** **boolean** isBST1(Node root){

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

**if**(!isBST1(root.left)) **return** **false**;

**if**(*prevNode*!=**null** && *prevNode*.data>=root.data){

**return** **false**;

}

*prevNode* = root;

**return** isBST1(root.right);

}

**return** **true**;

}

// //method 2

**public** **boolean** isBST2(Node root, **int** min, **int** max){

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

**if**(root.data>max || root.data<=min){

**return** **false**;

}

**if**(isBST2(root.left, min, root.data)==**false**||isBST2(root.right, root.data,max)==**false**){

**return** **false**;

}

**return** **true**;

}**else**{

**return** **true**;

}

}

**public** **void** inorder(Node root){

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

inorder(root.left);

System.*out*.print(" " + root.data);

inorder(root.right);

}

}

**public** **static** **void** main(String args[]){

isBST i = **new** isBST();

Node root = **new** Node(20);

root.left = **new** Node(10);

root.right = **new** Node(30);

root.left.left = **new** Node(5);

root.left.right = **new** Node(15);

root.right.left = **new** Node(25);

root.right.right = **new** Node(35);

System.*out*.println("Tree is " );

i.inorder(root);

System.*out*.println();

System.*out*.println("is Tree BST ?? METHOD 1 : " + i.isBST1(root));

System.*out*.println("is Tree BST ?? METHOD 2 : " + i.isBST2(root, Integer.*MIN\_VALUE*, Integer.*MAX\_VALUE*));

root.left.right.right = **new** Node(40);

System.*out*.println("Tree is " );

i.inorder(root);

System.*out*.println();

System.*out*.println("is Tree BST ?? METHOD 1 : " + i.isBST1(root));

System.*out*.println("is Tree BST ?? METHOD 2 : " + i.isBST2(root, Integer.*MIN\_VALUE*, Integer.*MAX\_VALUE*));

}

}

**class** Node{

**int** data;

Node left;

Node right;

**public** Node(**int** data){

**this**.data = data;

left = **null**;

right = **null**;

}

}

**Output :**

Tree is

5 10 15 20 25 30 35

is Tree BST ?? METHOD 1 : true

is Tree BST ?? METHOD 2 : true

Tree is

5 10 15 40 20 25 30 35

is Tree BST ?? METHOD 1 : false

is Tree BST ?? METHOD 2 : false

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Algorithms - Given a sorted array, find the smallest integer which cannot be represented by sum any subset of the array.**

**Objective:** Given a sorted array of positive integers, find out the smallest integer which cannot be represented as the sum of any subset of the array

**Input:** A Sorted Array

**Output:** The smallest number which cannot be represented as the sum of any subset of the given array

Examples :

Array {1,1,3,4,6,7,9} smallest Number : 32

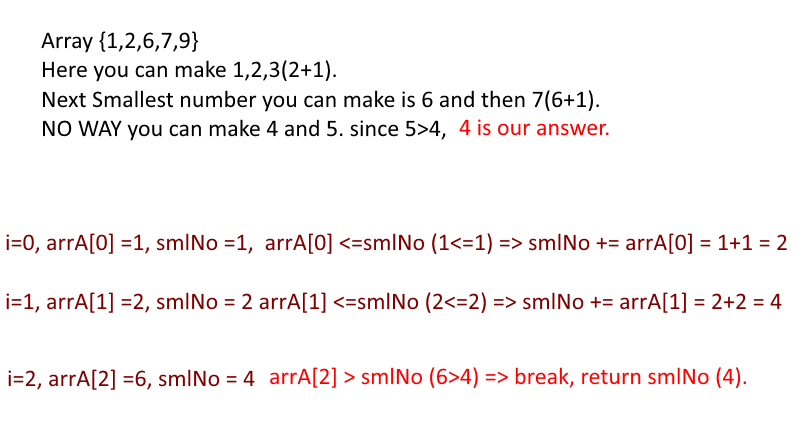
Array {1,1,1,1,1} smallest Number : 6

Array {2,3,6,7} smallest Number : 1

Array {1,2,6,7,9} smallest Number : 4

**Approach:**

* If 1 is not present in the array, our answer is 1.
* So take a variable "smlNumber" and assign 1 to it.
* Now we need to find the gap between the array elements which cannot be represented as sum of any subset of array.
* To find that keep adding the array elements to *smlNumber* and check it current array element and if at any point *smlNumber*<current array element that means we have found the gap. return *smlNumber.*
* See figure

****

**Complete Code:**

**package** newInterviewQuestions;

**public** **class** SmallestIntegerInSortedArray {

**public** **int** find(**int** [] arrA){

**int** smlNumber = 1;

**for**(**int** i = 0;i<arrA.length;i++){

**if**(arrA[i]<=smlNumber){

smlNumber += arrA[i];

}**else**{

**break**;

}

}

**return** smlNumber;

}

**public** **static** **void** main(String arg[]){

SmallestIntegerInSortedArray i = **new** SmallestIntegerInSortedArray();

System.*out*.println("Smallest Positive Integer that cant be represented by the sum of any subset of following arrays are : ");

**int** [] arrA = { 1,1,3,4,6,7,9};

System.*out*.println("{1,1,3,4,6,7,9} - " + i.find(arrA));

**int** [] arrB = {1,1,1,1,1};

System.*out*.println("{1,1,1,1,1} -> " + i.find(arrB));

**int** [] arrC = {2,3,6,7};

System.*out*.println("{2,3,6,7} -> " + i.find(arrC));

**int** [] arrD = {1,2,6,7,9};

System.*out*.println("{1,2,6,7,9} -> " + i.find(arrD));

}

}

Output:

Smallest Positive Integer that cant be represented by the sum of any subset of following arrays are :

{1,1,3,4,6,7,9} - 32

{1,1,1,1,1} -> 6

{2,3,6,7} -> 1

{1,2,6,7,9} -> 4

**Algorithms - Find the first repeated element in an array by its index**

**Objective:** Given an array of integers, find out the first repeated element. First repeated element means the element occurs atleast twice and has smallest index.

**Input:** An Array

**Output:** The first repeated element

Examples :

Array {1,1,3,4,6,7,9} smallest Number : 1

Array {7,2,2,3,7} smallest Number : 7

Array {5,3,3,3,3,3,3,3,4,4,4,5,3} smallest Number : 5

**Approach:**

**Naive Solution :**

Use two for loops. Time Complexity O(N2).

**Better Solution: Using HastSet, Time Complexity O(N).**

* Take a variable say *index = -1.*
* Initialize a HashSet.
* Navigate the array from right to left taking one element at a time
* if HashSet doesnt contain element, add it
* if HashSet contains then update the index with current index in navigation.
* At the end index will be updated by the element which is repeated and has the lowest index.

Complete Code:

**package** newInterviewQuestions;

**import** java.util.HashSet;

**public** **class** FirstRepeatingelement {

**public** **int** find(**int** [] arrA){

**int** index = -1;

HashSet<Integer> hs = **new** HashSet<>();

**for**(**int** i = arrA.length-1;i>=0;i--){

**if**(hs.contains(arrA[i])){

index = i;

}**else**{

hs.add(arrA[i]);

}

}

**return** arrA[index];

}

**public** **static** **void** main(String args[]){

**int** [] a = {1,2,5,7,5,3,10,2};

FirstRepeatingelement f = **new** FirstRepeatingelement();

System.*out*.println("{1,2,5,7,5,3,10,2}");

System.*out*.println("first repeated element by index is : " + f.find(a));

}

}

Output:

{1,2,5,7,5,3,10,2}

first repeated element by index is : 2

**Algorithms - Find all common numbers in given three sorted arrays.**

**Objective:** Given three sorted(ascending order) arrays of integers, find out all the common elements in them.

**Input:** Three sorted arrays.

**Output:** All the common elements.

Examples :

Array A = {1,2,3,4,5,6,7,8,9,10};

Array B = {1,3,5,6,7,8,12};

Array C = {2,3,4,5,8,9};

Common Elements are 3,5,8

**Approach:**

* Very Simple Solution.
* Navigate all three arrays(A,B,C) simultaneously using indexes say, i,j,k.
* if(A[i]==B[j]==C[k]) then print A[i] and do i++, j++, k++.
* if not then compare all A[i],B[j],C[k] and which ever is smaller, increase its index.
* Stop when any of these array gets over

Complete Code:

**package** Arrays;

**public** **class** FindCommonElement3Arrays {

**public** **void** findCommon(**int** [] A, **int** [] B, **int** [] C){

**int** i=0,j=0,k=0;

**while**(i<A.length && j<B.length && k<C.length){

**if**((A[i]==B[j])&& (B[j]==C[k])){

System.*out*.print(A[i] + " ");

i++;j++;k++;

}**else** **if**((A[i]<=B[j])&& (A[i]<=C[k])){

i++;

}**else** **if**((B[j]<=A[i])&& (B[j]<=C[k])){

j++;

}**else**{

k++;

}

}

}

**public** **static** **void** main(String args[]){

**int** [] A = {1,2,3,4,5,6,7,8,9,10};

**int** [] B = {1,3,5,6,7,8,12};

**int** [] C = {2,3,4,5,8,9};

FindCommonElement3Arrays f = **new** FindCommonElement3Arrays();

System.*out*.print("Common Elements are : ");

f.findCommon(A, B, C);

}

}

Output:

Common Elements are : 3 5 8

**Algorithms - Find the number of occurrences of a number in a given sorted array.**

**Objective:** Given a sorted(ascending order) arrays of integers, find out the number of occurences of a number in that array

**Input:** A sorted array arrA[] and a number x.

**Output:** number of occurrences of 'x' in array arrA[].

Examples :

Array - {1,2,2,2,2,2,2,2,3,4,5,5,6}

x = 2

Output : No of Occurrences of number 2 is 7

**Approach:**

**Method 1: O(n)**

Do the linear scan and count the occurrences.

**Method 2: Binary search Technique : O(lgn)**

**Steps:**

* Find the first occurrence of x in the array. say it is startPoint.
* Find the last occurrrence of x in the array, say it is endPoint.
* Calculate the number of occurrence = endPoint-startPoint+1

**How do you find the first occurrence of a number using binary search.**

Put one additional condition in the step when array[mid]==x, just check whether the element prior to mid is smaller than the x, if yes then only return the mid index, this will give you the first occurrence. handle the boundary conditions.

**How do you find the last occurrence of a number using binary search.**

Put one additional condition in the step when array[mid]==x, just check whether the element after the mid is greater than the x, if yes then only return the mid index, this will give you the last occurrence. handle the boundary conditions.

Complete Code:

**package** Arrays;

**public** **class** OccurrencesInArray {

**public** **int** findOccurrences(**int** [] arrA, **int** x){

**int** count = 0;

**int** startPoint = findFirstOccurrence(arrA,x,0,arrA.length-1);

**if**(startPoint<0){

**return** -1;

}

**int** endPoint = findLastOccurrence(arrA, x, 0, arrA.length-1);

count = endPoint-startPoint+1;

**return** count;

}

**public** **int** findFirstOccurrence(**int** [] arrA, **int** x,**int** start, **int** end ){

**if**(end>=start){

**int** mid = (start+end)/2;

**if**((mid==0||(arrA[mid-1]<x)) && arrA[mid]==x){

**return** mid;

}**else** **if**(arrA[mid]<x){

**return** findFirstOccurrence(arrA, x, mid+1, end);

}**else**{

**return** findFirstOccurrence(arrA, x, start, mid-1);

}

}**else** **return** -1;

}

**public** **int** findLastOccurrence(**int** [] arrA, **int** x,**int** start, **int** end ){

**if**(end>=start){

**int** mid = (start+end)/2;

**if**((mid==arrA.length-1||arrA[mid+1]>x) &&(arrA[mid]==x)){

**return** mid;

}**else** **if**(arrA[mid]>x){

**return** findLastOccurrence(arrA, x, start, mid-1);

}**else**{

**return** findLastOccurrence(arrA, x, mid+1, end);

}

}**else** **return** -1;

}

**public** **static** **void** main(String args[]){

**int** [] arrA = {1,2,2,2,2,2,2,2,3,4,5,5,6};

**int** x = 2;

OccurrencesInArray i = **new** OccurrencesInArray();

**int** r = i.findOccurrences(arrA, x);

System.*out*.println("No of Occurrences of number " + x + " is : " + r);

}

}

Output:

No of Occurrences of number 2 is : 7

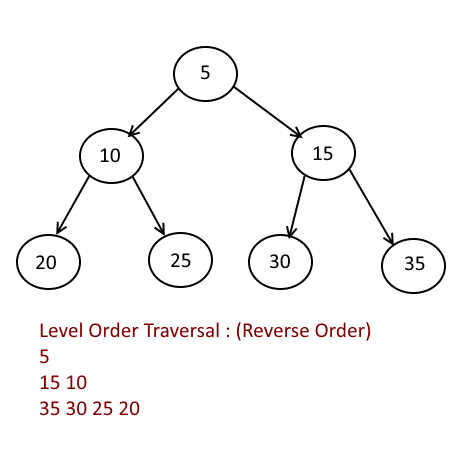
**Algorithms - Level Order Traversal, Print Nodes at each level in reverse order**

**Objective:** Given a binary Tree, Do Level Order Traversal, Print Nodes at each level in reverse order

**Input:** A Binary Tree

**Output:** Level Order Traversal, Print Nodes at each level in reverse order

Example:



**Approach:**

The solution of this problem is very simple, Do level order traversal and in recursive calls first add the right child and then left child.

Read this solution " Level Order Traversal, Print each level in separate line" and implement the above line.

Complete Code:

package LevelOrderTraversalByLine;

import java.util.LinkedList;

import java.util.Queue;

public class LevelOrderTraversal {

public void levelOrderNaiveApproach(Node root){

int h = height(root);

for(int i=1;i<=h;i++){

printLevels(root,i);

System.out.println("");

}

}

public void printLevels(Node root, int h){

if(root==null) return;

if(h==1) System.out.print(" " + root.data);

else{

printLevels(root.left,h-1);

printLevels(root.right,h-1);

}

}

public int height(Node root){

if (root==null) return 0;

return 1 + Math.max(height(root.left),height(root.right));

}

public void levelOrderQueue(Node root){

Queue q = new LinkedList();

int levelNodes =0;

if(root==null) return;

q.add(root);

while(!q.isEmpty()){

levelNodes = q.size();

while(levelNodes>0){

Node n = (Node)q.remove();

System.out.print(" " + n.data);

if(n.left!=null) q.add(n.left);

if(n.right!=null) q.add(n.right);

levelNodes--;

}

System.out.println("");

}

}

public static void main (String[] args) throws java.lang.Exception

{

Node root = new Node(5);

root.left = new Node(10);

root.right = new Node(15);

root.left.left = new Node(20);

root.left.right = new Node(25);

root.right.left = new Node(30);

root.right.right = new Node(35);

LevelOrderTraversal i = new LevelOrderTraversal();

System.out.println(" Output by Naive Approach : ");

i.levelOrderNaiveApproach(root);

System.out.println(" Output by Better Approach : ");

i.levelOrderQueue(root);

}

}

class Node{

int data;

Node left;

Node right;

public Node(int data){

this.data = data;

this.left = null;

this.right =null;

}

}

Output by Naive Approach :

5

10 15

20 25 30 35

Output by Better Approach :

5

10 15

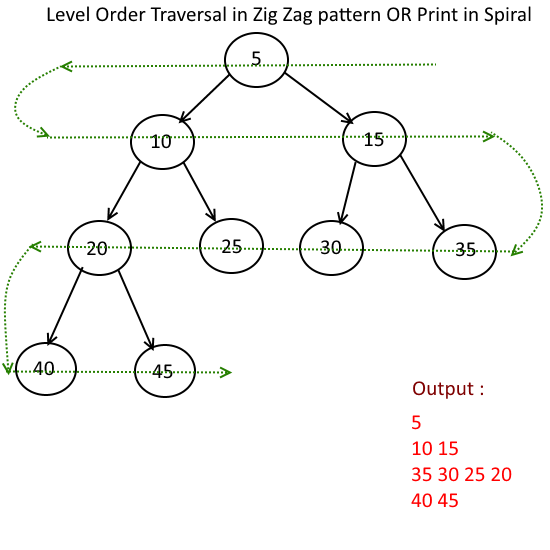
20 25 30 35

**Algorithms - Level Order Traversal in Zig Zag pattern OR Print in Spiral Pattern**

**Objective:** Given a binary Tree, Do Level Order Traversal in Zig Zag pattern OR Print in Spiral

**Input:** A Binary Tree

**Output:** Order Traversal in Zig Zag pattern OR Print in Spiral.



**Better Solution :**

* Idea is to take all nodes at each level and print them forward and reverse order alternatively.
* Create a ArrayList al.
* Do the level order traversal using queue(Breadth First Search). Click here to know about how to level order traversal.
* For getting all the nodes at each level, before you take out a node from queue, store the size of the queue in a variable, say you call it as levelNodes.
* Now while levelNodes>0, take out the nodes and add it to the array list and add their children into the queue.
* Alternatively print the array list in forward and backward order.
* After this while loop put a line break.

**Time Complexity : O(N)**

**Complete Code:**

package ZigZagTraversal;

import java.util.ArrayList;

import java.util.Collections;

import java.util.LinkedList;

import java.util.Queue;

public class ZigZagTraversal {

ArrayList<Integer> al = new ArrayList<>();

public static boolean evenLevel = false;

public void levelOrderQueue(Node root){

Queue q = new LinkedList();

int levelNodes =0;

if(root==null) return;

q.add(root);

while(!q.isEmpty()){

levelNodes = q.size();

al.clear();

while(levelNodes>0){

Node n = (Node)q.remove();

al.add(n.data);

//System.out.print(" " + n.data);

if(n.left!=null) q.add(n.left);

if(n.right!=null) q.add(n.right);

levelNodes--;

}

if(evenLevel){

System.out.print(al);

evenLevel = !evenLevel;

}else{

Collections.reverse(al);

System.out.print(al);

evenLevel = !evenLevel;

}

System.out.println("");

}

}

public static void main (String[] args) throws java.lang.Exception

{

Node root = new Node(5);

root.left = new Node(10);

root.right = new Node(15);

root.left.left = new Node(20);

root.left.right = new Node(25);

root.right.left = new Node(30);

root.right.right = new Node(35);

root.left.left.left = new Node(40);

root.left.left.right = new Node(45);

ZigZagTraversal i = new ZigZagTraversal();

System.out.println(" Spiral Print of a Tree : ");

i.levelOrderQueue(root);

}

}

class Node{

int data;

Node left;

Node right;

public Node(int data){

this.data = data;

this.left = null;

this.right =null;

}

}

**Output:**

Spiral Print of a Tree :

[5]

[10, 15]

[35, 30, 25, 20]

[40, 45]

**Algorithms - Find the maximum width of a binary tree.**

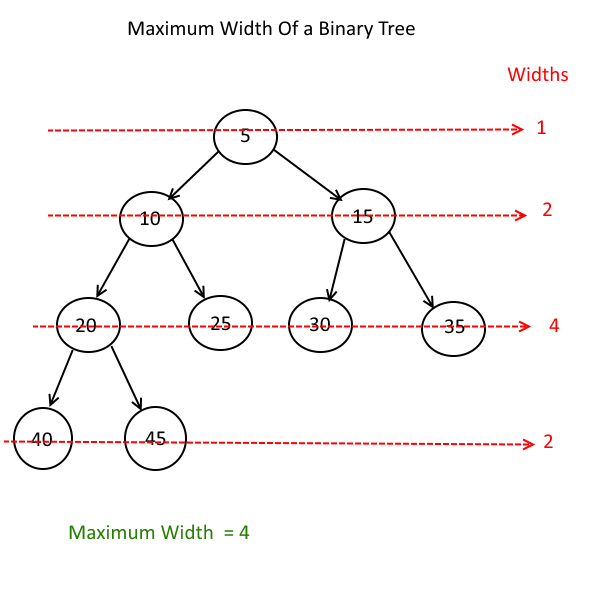
**Objective:** Given a binary Tree, Find the maximum width.

**Note:** Maximum width of a tree is nothing but the Max(nodes at each level).

**Input:** A Binary Tree

**Output:** Maximum width of a given tree.

Example:



**Approach:**

The solution of this problem is very simple, Do level order traversal and in recursive calls first add the right child and then left child and count the number of nodes at each level and keep track of Max of them and at the end return the max.

Read this solution " Level Order Traversal, Print each level in separate line" and implement the above approach.

Complete Code:

**package** MaxWidthOfTree;

**import** java.util.LinkedList;

**import** java.util.Queue;

**public** **class** MaxWidthOfTree {

**public** **static** **int** *maxWidth*=0;

**public** **int** findMaxWidth(Node root){

Queue q = **new** LinkedList<>();

**int** levelNodes =0;

**if**(root==**null**) **return** 0;

q.add(root);

**while**(!q.isEmpty()){

levelNodes = q.size();

**if**(levelNodes>*maxWidth*){

*maxWidth* = levelNodes;

}

**while**(levelNodes>0){

Node n = (Node)q.remove();

**if**(n.left!=**null**) q.add(n.left);

**if**(n.right!=**null**) q.add(n.right);

levelNodes--;

}

}

**return** *maxWidth*;

}

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

Node root = **new** Node(5);

root.left = **new** Node(10);

root.right = **new** Node(15);

root.left.left = **new** Node(20);

root.left.right = **new** Node(25);

root.right.left = **new** Node(30);

root.right.right = **new** Node(35);

MaxWidthOfTree i = **new** MaxWidthOfTree();

**int** x = i.findMaxWidth(root);

System.*out*.println("Maximum Width of a binary Tree is : " + x);

}

}

**class** Node{

**int** data;

Node left;

Node right;

**public** Node(**int** data){

**this**.data = data;

**this**.left = **null**;

**this**.right =**null**;

}

}

Output:

Maximum Width of a binary Tree is : 4

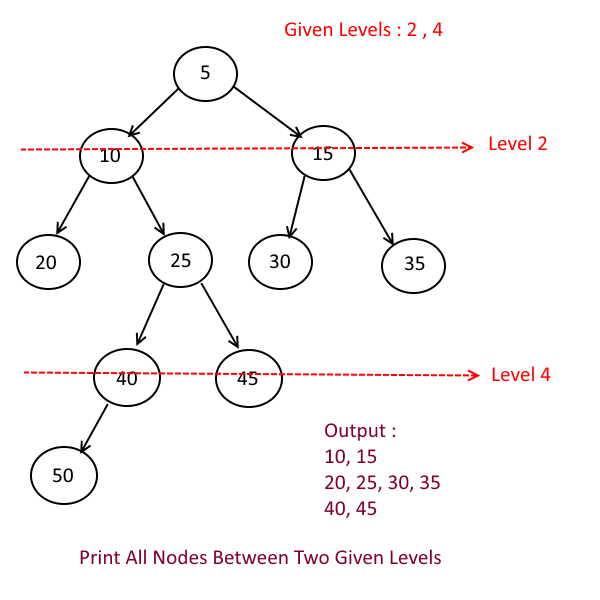
**Algorithms - Print All Nodes Between Two Given Levels**

**Objective:** Given a binary Tree and two levels, Print all the nodes between those.

**Input:** A Binary Tree and two levels.

**Output:** Print all the nodes between given levels.

Example:



**Approach:**

* The solution of this problem is very simple, Do level order traversal and in recursive calls first add the right child and then left child.
* Take a global variable say, *currLevel* which will store the current level while doing Level Order traversal.
* Print only nodes between given levels.

**Read this solution " Level Order Traversal, Print each level in separate line" and implement the above approach.**

**Complete Code:**

**package** NodesBwTwoGivenLevels;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.LinkedList;

**import** java.util.Queue;

**public** **class** NodesBwTwoGivenLevels {

**public** **static** **int** *currLevel* = 1;

**public** **void** levelOrderQueue(Node root, **int** low, **int** high){

Queue q = **new** LinkedList();

**int** levelNodes =0;

**if**(root==**null**) **return**;

q.add(root);

**while**(!q.isEmpty()){

levelNodes = q.size();

**while**(levelNodes>0){

Node n = (Node)q.remove();

**if**(*currLevel*>=low && *currLevel*<=high){

System.*out*.print(" " + n.data);

}

**if**(n.left!=**null**) q.add(n.left);

**if**(n.right!=**null**) q.add(n.right);

levelNodes--;

}

**if**(*currLevel*>=low && *currLevel*<=high){

System.*out*.println("");

}

*currLevel*++;

}

}

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

Node root = **new** Node(5);

root.left = **new** Node(10);

root.right = **new** Node(15);

root.left.left = **new** Node(20);

root.left.right = **new** Node(25);

root.right.left = **new** Node(30);

root.right.right = **new** Node(35);

root.left.right.left = **new** Node(40);

root.left.right.right = **new** Node(45);

root.left.right.left.left = **new** Node(50);

NodesBwTwoGivenLevels i = **new** NodesBwTwoGivenLevels();

**int** low = 2;

**int** high = 3;

System.*out*.println("Print all nodes between nodes " + low + " and " + high);

i.levelOrderQueue(root,low,high);

}

}

**class** Node{

**int** data;

Node left;

Node right;

**public** Node(**int** data){

**this**.data = data;

**this**.left = **null**;

**this**.right =**null**;

}

}

Output:

Print all nodes between nodes 2 and 4

10 15

20 25 30 35

40 45

**Algorithms - Sort an Array such that the odd numbers appear first followed by the even numbers . The odd numbers in as**cending **order and the even numbers in descending order.**

**Objective:** Given an array of intergers, sort it such that the odd numbers appear first followed by the even numbers . The odd numbers in ascending order and the even numbers in descending order.

**Input:** An Array of Integers

Example:

Input Array : 1 2 3 4 5 6 7 8 9 10

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

**Appraoch:**

* First separate the odd numbers and even numbers, put odd numbers first followed by even numbers using quick sort technique.
* Now sort the odd numbers in ascending order
* Sort the even numbers in descending order.

**Complete Code:**

**package** Arrays;

**public** **class** DescendingOddAscendingEven {

**public** **static** **int**[] *arrA*;

**public** **void** solve() {

separateOddEven(0, *arrA*.length - 1);

**int** i =0;

**while**(*arrA*[i]%2!=0){

i++;

**if**(i==*arrA*.length-1)**break**;

}

**if**(i!=*arrA*.length-1){

customizedSort(0, i-1);

}**else**{

customizedSort(0, i);

}

customizedSort(i, *arrA*.length-1);

}

**public** **void** separateOddEven(**int** low, **int** high) {

**int** i = low;

**int** j = high;

**while** (i < j) {

**while** (i<high && *arrA*[i] % 2 != 0) {

i++;

}

**while** (j>low && *arrA*[j] % 2 == 0) {

j--;

}

**if** (i <= j) {

swap(i, j);

i++;

j--;

}

}

}

**public** **void** customizedSort(**int** low, **int** high) {

**int** mid = (low + high) / 2;

**int** left = low;

**int** right = high;

**int** pivot = *arrA*[mid]; // select middle element as pivot

**while** (left <= right) {

**if** (*arrA*[left] % 2 == 0) {

**while** (*arrA*[left] > pivot)

left++;// find element which is greater than pivot

**while** (*arrA*[right] < pivot)

right--;

} **else** **if** (*arrA*[left] % 2 != 0) {

**while** (*arrA*[left] < pivot)

left++;// find element which is greater than pivot

**while** (*arrA*[right] > pivot)

right--;

}

**if** (left <= right) {

**int** temp = *arrA*[left];

*arrA*[left] = *arrA*[right];

*arrA*[right] = temp;

left++;

right--;

}

}

// Recursion on left and right of the pivot

**if** (low < right)

customizedSort(low, right);

**if** (left < high)

customizedSort(left, high);

}

**public** **void** swap(**int** i, **int** j) {

**if** (i < j) {

**int** temp = *arrA*[i];

*arrA*[i] = *arrA*[j];

*arrA*[j] = temp;

}

}

**public** **void** printArray(**int**[] arrA) {

**for** (**int** i = 0; i < arrA.length; i++) {

System.*out*.print(" " + arrA[i]);

}

}

**public** **static** **void** main(String args[]) {

*arrA* = **new** **int**[] {1,2,3,4,5,6,7,8,9,10};

DescendingOddAscendingEven d = **new** DescendingOddAscendingEven();

System.*out*.println("Original Array : ");

d.printArray(*arrA*);

System.*out*.println("\nOutput Array : ");

d.solve();

d.printArray(*arrA*);

}

}

Output:

Original Array :

1 2 3 4 5 6 7 8 9 10

Output Array :

1 3 5 7 9 10 8 6 4 2

**Algorithms - Sort an Given Array in the order defined by another array**

**Objective:** Given an array of intergers, sort it according to the order defined by another array.

**Input:** An Array of Integers

Example:

Input Array : 2 6 9 1 4 4 2 1 10 13 5 7 8

Defined Array : 1 2 4 6

Output : 1 1 2 2 4 4 6 5 7 8 9 10 13

**Appraoch:**

**Method 1: Sort and Search**

1. Create an Aux array and copy all the elements of arrA
2. Create another boolean array, visited[] of size arrA[]
3. Initialize visited[] = false
4. Sort the Aux array using Merge sort.
5. Navigate through the arrB, taking one element at a time, say x
6. perform binary search of x on Aux array and find the first occurrence of x
7. if x is found, copy it to arrA, make visited[] = true
8. Do linear navigation on Aux array till you find x, copy all these to arrA and mark visited[] = true
9. When arrB is over, copy rest of the elements from Aux to arrA.

**Method 2: Using Hashing**

1. Insert all the elements of arrA in hash Table,
2. Element as key and its count as its value
3. Navigate through arrB, check element's presence in Hash table
4. If it is present then takes its value (count) and insert into arrA.
5. Once arrB is completed, take rest of the elements from Hash table
6. Sort Them and insert into arrB.

**Complete Code:**

**package** Arrays;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.Hashtable;

**import** java.util.Set;

**import** java.util.Iterator;

**import** interviewQuestion.MergeSort;

**public** **class** SortArrayDefinedByOtherArray {

**public** **int** [] SortAndSearch(**int** [] arrA, **int** [] arrB){

//create an Aux array and copy all the elements of arrA

// create another boolean array, visited[] of size arrA[]

// Initialize visited[] = false

// Sort the Aux array using Merge sort.

// Navigate through the arrB, taking one element at a time, say x

// 1. perform binary search of x on Aux array and find the first occurrence of x

// 2. if x is found, copy it to arrA, make visited[] = true

// 3. Do linear navigation on Aux array till you find x, copy all these to arrA and mark visited[] = true

// When arrB is over, copy rest of the elements from Aux to arrA.

**int** oIndex = -1;

**boolean** [] visited = **new** **boolean** [arrA.length];

**for**(**int** i=0;i<visited.length;i++){

visited[i] = **false**;

}

**int** [] Aux = **new** **int**[arrA.length];

**for**(**int** i=0;i<arrA.length;i++){

Aux[i] = arrA[i];

}

System.*out*.println("Original Array : ");

displayArray(arrA);

System.*out*.println("\nDefined Array : ");

displayArray(arrB);

MergeSort m = **new** MergeSort(Aux);

Aux = m.mergeSorting();

**for**(**int** i=0;i<arrB.length;i++){

**int** x = arrB[i];

**int** index = findFirstOccurrence(Aux, x, 0, Aux.length-1);

**if**(index>=0){

arrA[++oIndex]=Aux[index];

visited[index] = **true**;

//oIndex++;

**while**(Aux[++index]==x){

arrA[++oIndex]=Aux[index];

visited[index] = **true**;

}

}

}

**for**(**int** i=0;i<Aux.length;i++){

**if**(visited[i]==**false**){

arrA[++oIndex] = Aux[i];

}

}

**return** arrA;

}

**public** **int** findFirstOccurrence(**int** [] arrA, **int** x, **int** start, **int** end){

**if**(end>=start){

**int** mid = (start+end)/2;

**if**((mid==0||(arrA[mid-1]<x)) && arrA[mid]==x){

**return** mid;

}**else** **if**(arrA[mid]<x){

**return** findFirstOccurrence(arrA, x, mid+1, end);

}**else**{

**return** findFirstOccurrence(arrA, x, start, mid-1);

}

}**else** **return** -1;

}

**public** **int** [] usingHashing(**int** [] arrA, **int** [] arrB){

//Insert all the elements of arrA in hash Table,

//Element as key and its count as its value

//Navigate through arrB, check element's presence in Hash table

//if it is present then takes its value (count) and insert into arrA.

//Once arrB is completed, take rest of the elements from Hash table

// Sort Them and insert into arrB.

**int** resIndex = -1;

Hashtable<Integer, Integer> h = **new** Hashtable<>();

**for**(**int** i=0;i<arrA.length;i++){

**if**(h.containsKey(arrA[i])){

**int** count = h.get(arrA[i]);

count++;

h.put(arrA[i], count);

}**else**{

h.put(arrA[i], 1);

}

}

**for**(**int** i=0;i<arrB.length;i++){

**if**(h.containsKey(arrB[i])){

**int** count = h.get(arrB[i]);

**while**(count>0){

arrA[++resIndex] = arrB[i];

count--;

}

h.remove(arrB[i]);

}

}

ArrayList<Integer> al = **new** ArrayList<Integer>();

Set<Integer> keys = h.keySet();

**for**(Integer x:keys){

al.add(x);

}

Collections.*sort*(al);

Iterator<Integer> it = al.iterator();

**while**(it.hasNext()){

arrA[++resIndex] = it.next();

}

**return** arrA;

}

**public** **void** displayArray(**int** [] b){

**for**(**int** i=0;i<b.length;i++){

System.*out*.print(" " + b[i]);

}

}

**public** **static** **void** main(String args[]){

**int** [] arrA = {2,6,9,1,4,4,2,1,10,13,5,7,8};

**int** [] arrB = {1,2,4,6};

SortArrayDefinedByOtherArray s = **new** SortArrayDefinedByOtherArray();

**int** [] res = s.SortAndSearch(arrA, arrB);

System.*out*.print("\n Output Array using Sort and search: ");

s.displayArray(res);

System.*out*.print("\n Output Array using Hashing : ");

**int** [] res1 = s.usingHashing(arrA, arrB);

s.displayArray(res1);

}

}

Output:

Original Array :

2 6 9 1 4 4 2 1 10 13 5 7 8

Defined Array :

1 2 4 6

Output Array using Sort and search: 1 1 2 2 4 4 6 5 7 8 9 10 13

Output Array using Hashing : 1 1 2 2 4 4 6 5 7 8 9 10 13

**Algorithms - Convert InFix Expression to PostFix Expresssion**

**Objective:** Given an Infix Expression in String, Convert it to Post Fix Expression.

**Input:** An String represents Infix expression.

Example:

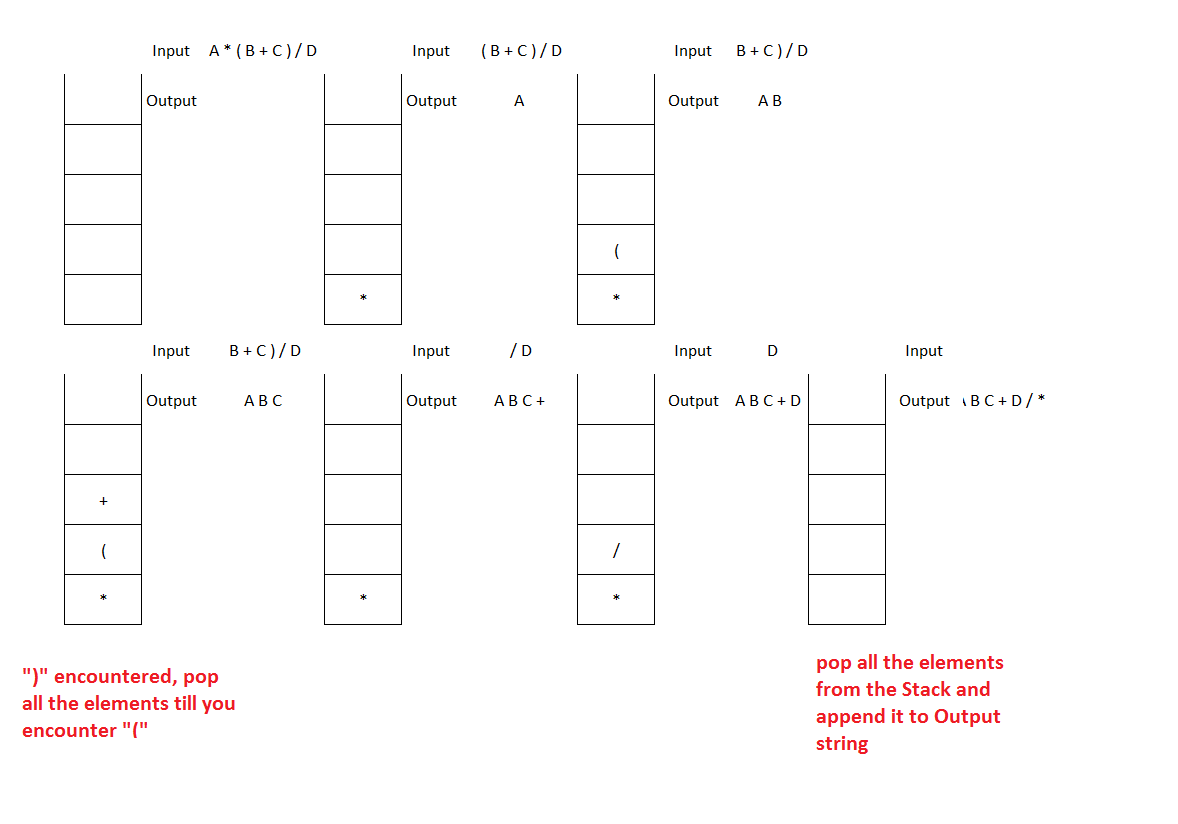
Input - Infix Expression : A\*(B+C)/D

Output - PostFix Expression : A B \* C D / +

**Appraoch:**

**Use Stack**

**take one character at**

****

**Algorithms – Print All The Permutations Of a String**

**Objective:** Given a String, print all the permutations of it.

**Input:** A String

**Output:** Print all the permutations of a string

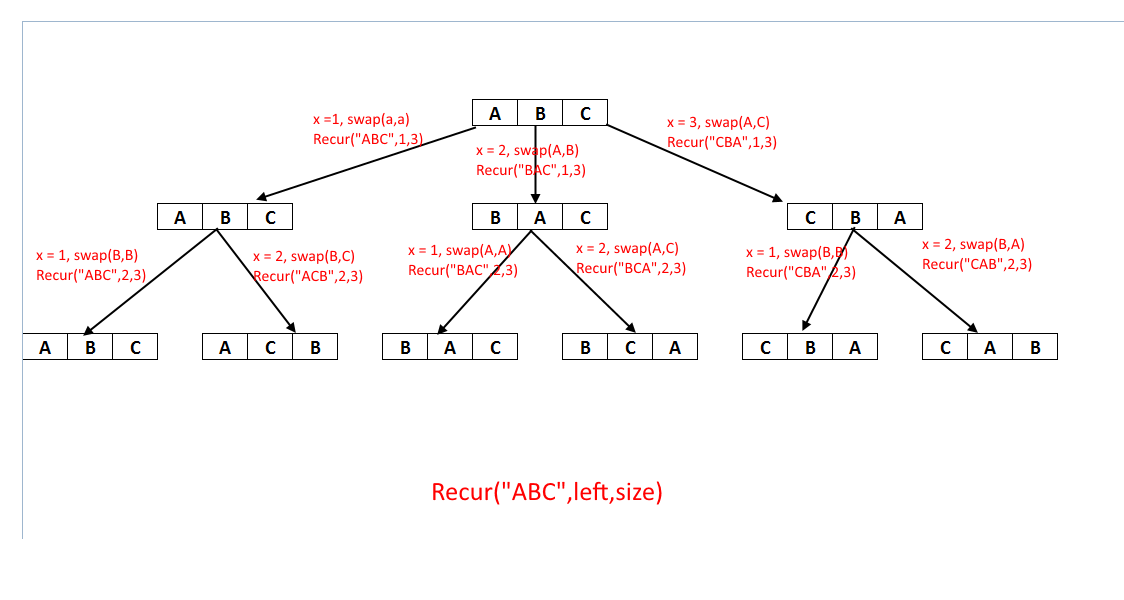
Example:

Input : abc

**Output:** abc acb bac bca cba cab

**Approach:**

* Take one character at a time and fix it at the first position. (*use swap to put every character at the first position*)
* make recursive call to rest of the characters.
* use swap to revert the string back to its original form fo next iteration.



**Complete Code:**

package interviewQuestion;

//

import java.util.\*;

import java.lang.\*;

import java.io.\*;

public class StringPermutations {

private char[] arrA;

public void permutation(char[] arrA, int left, int size) {

int x;

if (left == size) {

for (int i = 0; i < arrA.length; i++) {

System.out.print(arrA[i]);

}

System.out.print(" ");

} else {

for (x = left; x < size; x++) {

swap(arrA, left, x);

permutation(arrA, left + 1, size);

swap(arrA, left, x);

}

}

}

public void swap(char[] arrA, int i, int j) {

char temp = arrA[i];

arrA[i] = arrA[j];

arrA[j] = temp;

}

public static void main(String[] args) throws java.lang.Exception {

// your code goes here

String s = "abc";

char[] arrCh = s.toCharArray();

StringPermutations i = new StringPermutations();

i.permutation(arrCh, 0, arrCh.length);

}

}

Output:

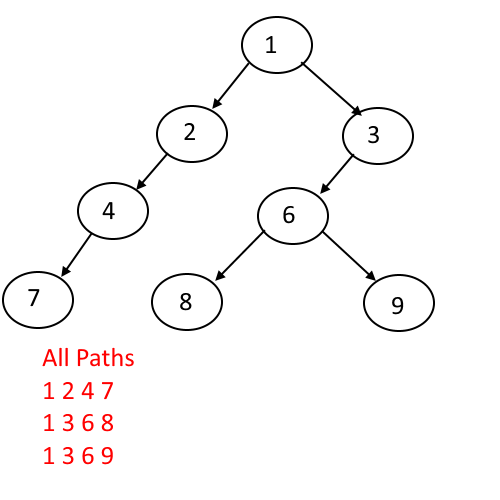
abc acb bac bca cba cab

**Algorithms – Print Paths from root to all leaf nodes in a binary tree.**

**Objective:** Given a binary tree, Print paths from root to all leaf nodes

**Input:** A binary tree

Example:



**Approach:**

* Do the preOrder traversal.
* Store the current node in an array, say Path[] and maintain the length say pathLength.
* check if you are at leaf node (both the left and right child of parent will be null).
* If not then keep traversing by making recursive calls to root.left and root.right. Pass the path and pathLen as parameter so that at each level have its own copy of path and and its length.
* If you hit the leaf node then print the array.
* See Code

**Complete Code:**

**package** PrintAllLeafPaths;

**public** **class** PrintAllLeafPaths {

**int** [] paths = **new** **int**[500];

**int** pathLength =0;

**public** **void** printPaths(Node root, **int** [] path, **int** pathLen){

**if**(root==**null**){

**return**;

}

path[pathLen++]= root.data;

**if**(root.left==**null** && root.right==**null**){

print(path,pathLen);

}

**else**{

printPaths(root.left,path,pathLen);

printPaths(root.right,path,pathLen);

}

}

**public** **void** print(**int** [] path, **int** pathLen){

**for**(**int** i=0;i<pathLen;i++){

System.*out*.print(path[i]+ " ");

}

System.*out*.println();

}

**public** **static** **void** main(String[] arg){

Node root = **new** Node(1);

root.left = **new** Node (2);

root.right = **new** Node (3);

root.left.left = **new** Node (4);

root.left.left.left = **new** Node (7);

root.right.left = **new** Node (6);

root.right.left.left = **new** Node (8);

root.right.left.right = **new** Node (9);

**int** [] path = **new** **int** [100];

PrintAllLeafPaths p = **new** PrintAllLeafPaths();

p.printPaths(root, path, 0);

}

}

**class** Node{

**int** data;

Node left;

Node right;

**public** Node (**int** data){

**this**.data = data;

left = **null**;

right = **null**;

}

}

Output:

1 2 4 7

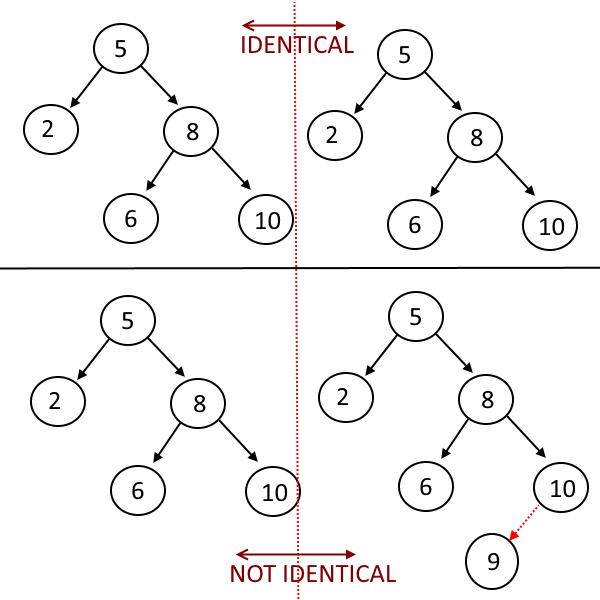
1 3 6 8

1 3 6 9

**Algorithms - Check if Two BST's are Identical**

**Objective:** Given Two binary Search Trees, Check if both are identical.

**Input:** Two binary Search Trees



**Approach:**

* Travesre both trees at the same time, starting from root.
* Check if roots are not null and data are matched, if not, return false.
* Make recursice calls to root.left and root.right.
* If any of the tree gets over and other is not , return false.
* if both traversal of both trees ends at the same time, return true
* see code.

Complete Code:

**package** findIdentialBST;

**public** **class** TwoIdenticalBST {

**public** **boolean** identicalBSTs(Node rootA, Node rootB){

**if**((rootA==**null**)&&(rootB==**null**)){

**return** **true**;

}**else** **if**((rootA!=**null** && rootB==**null**)||(rootA==**null** && rootB!=**null**)){

**return** **false**;

}**else** **if**(rootA.data==rootB.data){

**return** identicalBSTs(rootA.left, rootB.left) && identicalBSTs(rootA.right, rootB.right);

}**else**{

**return** **false**;

}

}

**public** **static** **void** main(String args[]){

BST a = **new** BST();

a.insert(2);a.insert(4);a.insert(1);a.insert(3);a.insert(5);

BST b = **new** BST();

b.insert(2);b.insert(4);b.insert(1);b.insert(3);b.insert(5);

TwoIdenticalBST t = **new** TwoIdenticalBST();

System.*out*.println(t.identicalBSTs(a.root, b.root));

a.insert(11);b.insert(12);

System.*out*.println(t.identicalBSTs(a.root, b.root));

}

}

**class** Node{

**int** data;

Node left;

Node right;

**public** Node(**int** data){

**this**.data = data;

left = **null**;

right = **null**;

}

}

**class** BST{

**public** Node root=**null**;

**public** **boolean** find(**int** id){

Node current = root;

**while**(current!=**null**){

**if**(current.data==id){

**return** **true**;

}**else** **if**(current.data>id){

current = current.left;

}**else**{

current = current.right;

}

}

**return** **false**;

}

**public** **void** insert(**int** id){

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

**if**(root==**null**){

root = newNode;

**return**;

}

Node current = root;

Node parent = **null**;

**while**(**true**){

parent = current;

**if**(id<current.data){

current = current.left;

**if**(current==**null**){

parent.left = newNode;

**return**;

}

}**else**{

current = current.right;

**if**(current==**null**){

parent.right = newNode;

**return**;

}

}

}

}

}

Output

true

false

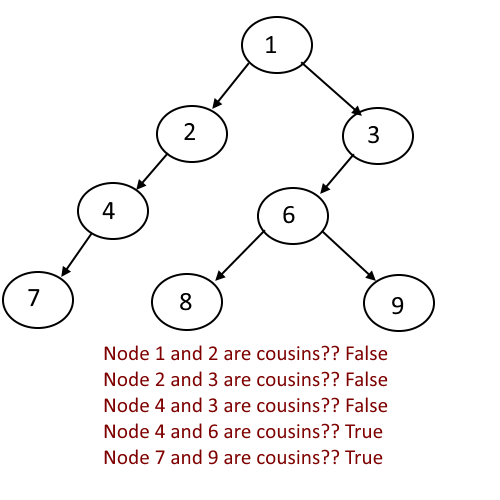
**Algorithms - In a Binary Tree, Check if two nodes are cousins**

**Objective:** Given a binary tree and two nodes, Check if they are cousins

**Input:** A binary tree and two nodes

**Cousin Nodes:** Cousin nodes are the nodes who are at the same level in the tree and whose parents are siblings.

**Example:**



**Approach:**

* Check the height of both the nodes, if heights are different then return false.
* Check if both the nodes has the same parent, if yes then return false.
* else return true.

Complete Code:

**package** CheckIiTwoNodesAreCousins;

**public** **class** CheckIiTwoNodesAreCousins {

**public** **boolean** areCousins(Node root, Node x, Node y) {

// get the heights of both the nodes and return false if heights ate not

// same

**if** (getHeight(root, x, 1) != getHeight(root, y, 1))

**return** **false**;

**else** {

// Now check if or not parents are same for both the node, if not ,

// return true

**if** (sameParents(root, x, y) == **false**) {

**return** **true**;

} **else** {

**return** **false**;

}

}

}

**public** **int** getHeight(Node root, Node x, **int** height) {

**if** (root == **null**)

**return** 0;

**if** (root == x)

**return** height;

**int** level = getHeight(root.left, x, height + 1);

**if** (level != 0)

**return** level;

**return** getHeight(root.right, x, height + 1);

}

**public** **boolean** sameParents(Node root, Node x, Node y) {

**if** (root == **null**)

**return** **false**;

**return** ((root.left == x && root.right == y)

|| (root.left == y && root.right == x)

|| sameParents(root.left, x, y) || sameParents(root.right, x, y));

}

**public** **static** **void** main(String[] args) **throws** java.lang.Exception {

Node root = **new** Node(1);

Node x1 = **new** Node(2);

Node y1 = **new** Node(3);

root.left = x1;

root.right = y1;

root.left.left = **new** Node(4);

root.right.left = **new** Node(6);

Node x2 = **new** Node(7);

Node y2 = **new** Node(9);

root.right.left.left = **new** Node(8);

root.right.left.right = y2;

root.left.left.left = x2;

CheckIiTwoNodesAreCousins i = **new** CheckIiTwoNodesAreCousins();

System.*out*.println("Node " + x1.data + " and Node " + y1.data + " are cousins??? " + i.areCousins(root, x1, y1));

System.*out*.println("Node " + x2.data + " and Node " + y2.data + " are cousins??? " + i.areCousins(root, x2, y2));

}

}

**class** Node {

**int** data;

Node left;

Node right;

**public** Node(**int** data) {

**this**.data = data;

**this**.left = **null**;

**this**.right = **null**;

}

}

Output:

Node 2 and Node 3 are cousins??? false

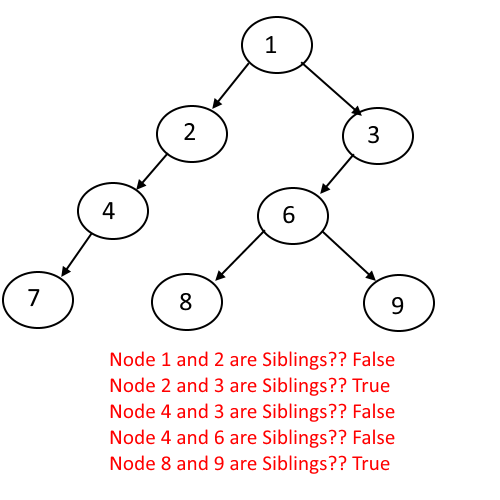
Node 7 and Node 9 are cousins??? true

**Algorithms - In a Binary Tree, Check if Two nodes has the same parent or are siblings**

**Objective:** In a Binary Tree, Check if Two nodes has the same parent or are siblings

**Input:** A binary tree and two nodes

**Example:**

****

**Approach:**

* Given, root, Node x, Node y.
* Check if x and y are childs of root. (*root.left==x && root.right==y) ||root.left==y && root.right==x*)
* if yes then return true.
* Else make a recursive call to root.left and root.right

Complete Code:

**package** CheckIfTwoNodesAreSiblings;

**public** **class** CheckIfTwoNodesAreSiblings {

**public** **boolean** sameParents(Node root, Node x, Node y){

**if**(root==**null**) **return** **false**;

**return** ((root.left==x && root.right==y) ||root.left==y && root.right==x ||sameParents(root.left,x,y) || sameParents(root.right,x,y));

}

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

Node root = **new** Node(1);

Node x1 = **new** Node(2);

Node y1 = **new** Node(3);

root.left = x1;

root.right = y1;

root.left.left = **new** Node(4);

root.right.left = **new** Node(6);

Node x2 = **new** Node(7);

Node y2 = **new** Node(9);

root.right.left.left = **new** Node(8);

root.right.left.right = y2;

root.left.left.left = x2;

CheckIfTwoNodesAreSiblings i = **new** CheckIfTwoNodesAreSiblings();

System.*out*.println("Node " + x1.data + " and Node " + y1.data + " are siblings??? " + i.sameParents(root, x1, y1));

System.*out*.println("Node " + x2.data + " and Node " + y2.data + " are siblings??? " + i.sameParents(root, x2, y2));

}

}

**class** Node{

**int** data;

Node left;

Node right;

**public** Node(**int** data){

**this**.data = data;

**this**.left = **null**;

**this**.right = **null**;

}

}

Output:

Node 2 and Node 3 are siblings??? true

Node 7 and Node 9 are siblings??? false

**Algorithms - Print Right View of a given binary tree**

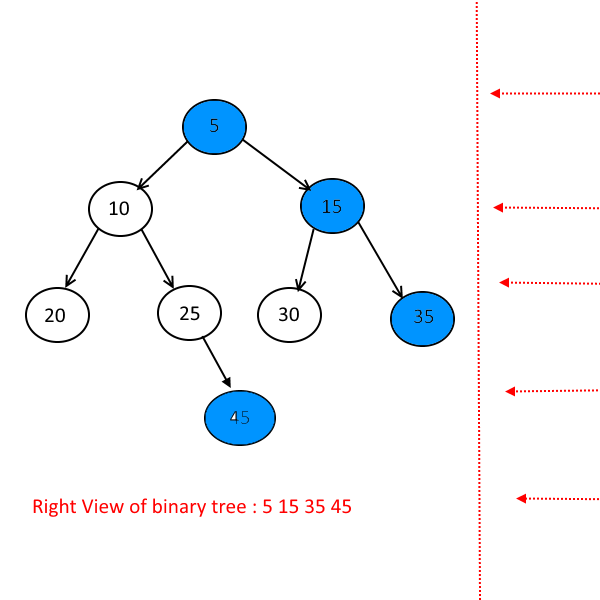
**Objective:** In a Binary Tree, print right view of it

**Input:** A binary tree.

*What is Right View of a binary Tree*

When just look at the tree from the right side , all the nodes you can see will be the right view of the tree.

**Example:**

****

**Approach :**

**Method 1 :**

* Traverse the tree from right to left
* Print the first node you encounter
* Take two variables , currentLevel=0 and nextLevel=1
* As soon as you change level , change the currentLevel = nextLevel
* Print only when current level<nextLevel so this way you will print only the first element
* For rest of the nodes on the the level currentLevel and nextLevel are equal so it wont print

**Method 2:**

Do the Level order traversal and print the last node value

Complete Code for Method 1:

**package** RightViewOfTree;

**public** **class** RightViewOfTree {

// method 1 : recursion :

//traverse the tree from right to left

// print the first node you encounter

//take two variables , currentLevel=0 and nextLevel=1

//as soon as you change level , change the currentLevel = nextLevel

//print only when current level<nextLevel so this way you will print only the first element

//for rest of the nodes on the the level currentLevel and nextLevel are equal so it wont print

//Method 2:

//do the Level order traversal and print the last node value

**public** **static** **int** *currentLevel* =0;

**public** **void** rightViewRecur(Node root, **int** nextLevel){

**if**(root==**null**) **return**;

**if**(*currentLevel*<nextLevel){

System.*out*.print (" " + root.data);

*currentLevel* = nextLevel;

}

rightViewRecur(root.right,nextLevel+1);

rightViewRecur(root.left,nextLevel+1);

}

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

Node root = **new** Node(5);

root.left = **new** Node(10);

root.right = **new** Node(15);

root.left.left = **new** Node(20);

root.left.right = **new** Node(25);

root.right.left = **new** Node(30);

root.right.right = **new** Node(35);

root.left.right.right = **new** Node(45);

RightViewOfTree i = **new** RightViewOfTree();

i.rightViewRecur(root,1);

}

}

**class** Node{

**int** data;

Node left;

Node right;

**public** Node(**int** data){

**this**.data = data;

**this**.left = **null**;

**this**.right =**null**;

}

}

Output:

5 15 35 45

**Algorithms - Print Left View of a given binary tree**

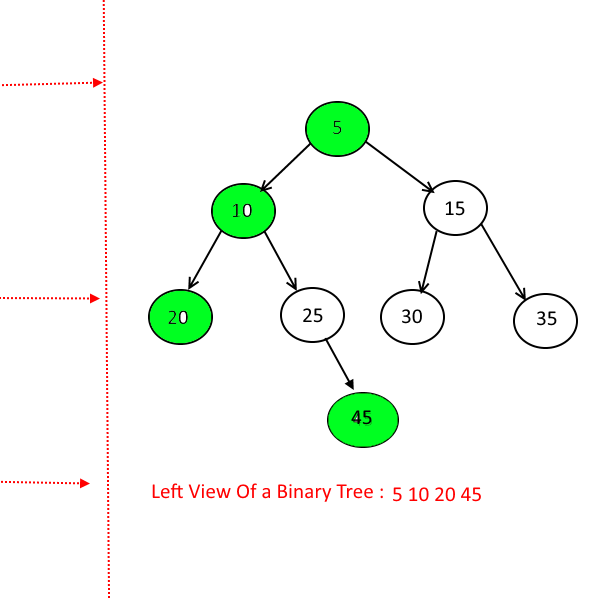
**Objective:** In a Binary Tree, print left view of it

**Input:** A binary tree.

*What is left View of a binary Tree*

When just look at the tree from the left side , all the nodes you can see will be the left view of the tree.

**Example:**

****

**Appraoch:**

**Method 1:**

* Traverse the tree from left to right
* Print the first node you encounter
* Take two variables , currentLevel=0 and nextLevel=1
* As soon as you change level , change the currentLevel = nextLevel
* Print only when current level<nextLevel so this way you will print only the first element
* For rest of the nodes on the the level currentLevel and nextLevel are equal so it wont print

**Method 2:**

Do the Level order traversal and print the first node value

Complete Code:

**package** LeftViewOfTree;

**public** **class** LeftViewOfTree {

**public** **static** **int** *index* = 0;

**public** **void** Method2\_levelOrder(Node root) {

**int** h = height(root);

**for** (**int** i = 1; i <= h; i++) {

*index* = 1;

printLevels(root, i);

// System.out.println("");

}

}

**public** **void** printLevels(Node root, **int** h) {

**if** (root == **null**)

**return**;

**if** (h == 1 && *index* == 1) {

System.*out*.print(" " + root.data);

*index*++;

} **else** {

printLevels(root.left, h - 1);

printLevels(root.right, h - 1);

}

}

**public** **static** **int** *currentLevel* =0;

**public** **void** leftViewRecur(Node root, **int** nextLevel){

**if**(root==**null**) **return**;

**if**(*currentLevel*<nextLevel){

System.*out*.print (" " + root.data);

*currentLevel* = nextLevel;

}

leftViewRecur(root.left,nextLevel+1);

leftViewRecur(root.right,nextLevel+1);

}

**public** **int** height(Node root) {

**if** (root == **null**)

**return** 0;

**return** 1 + Math.*max*(height(root.left), height(root.right));

}

**public** **static** **void** main(String[] args) **throws** java.lang.Exception {

Node root = **new** Node(5);

root.left = **new** Node(10);

root.right = **new** Node(15);

root.left.left = **new** Node(20);

root.left.right = **new** Node(25);

root.right.left = **new** Node(30);

root.right.right = **new** Node(35);

root.left.right.right = **new** Node(45);

LeftViewOfTree i = **new** LeftViewOfTree();

System.*out*.println("METHOD 1: ");

i.leftViewRecur(root, 1);

System.*out*.println("\nMETHOD 2 : Using Level Order, Left view ");

i.Method2\_levelOrder(root);

}

}

**class** Node {

**int** data;

Node left;

Node right;

**public** Node(**int** data) {

**this**.data = data;

**this**.left = **null**;

**this**.right = **null**;

}

}

Output:

METHOD 1:

5 10 20 45

METHOD 2 : Using Level Order, Left view

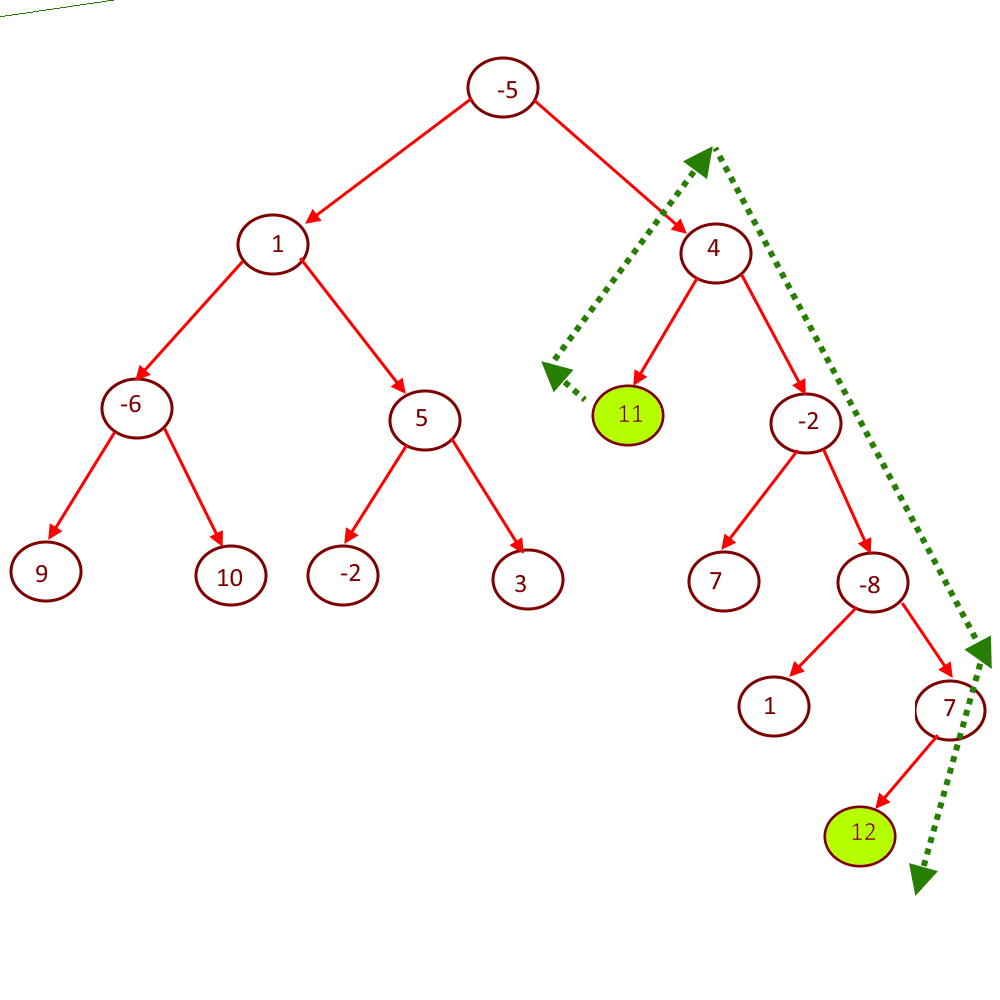
5 10 20 45

**Algorithms - Given a binary tree, Find the maximum path sum between Two Leaves**

**Objective:** Given a binary tree, Find the maximum path sum from one leaf node to another.

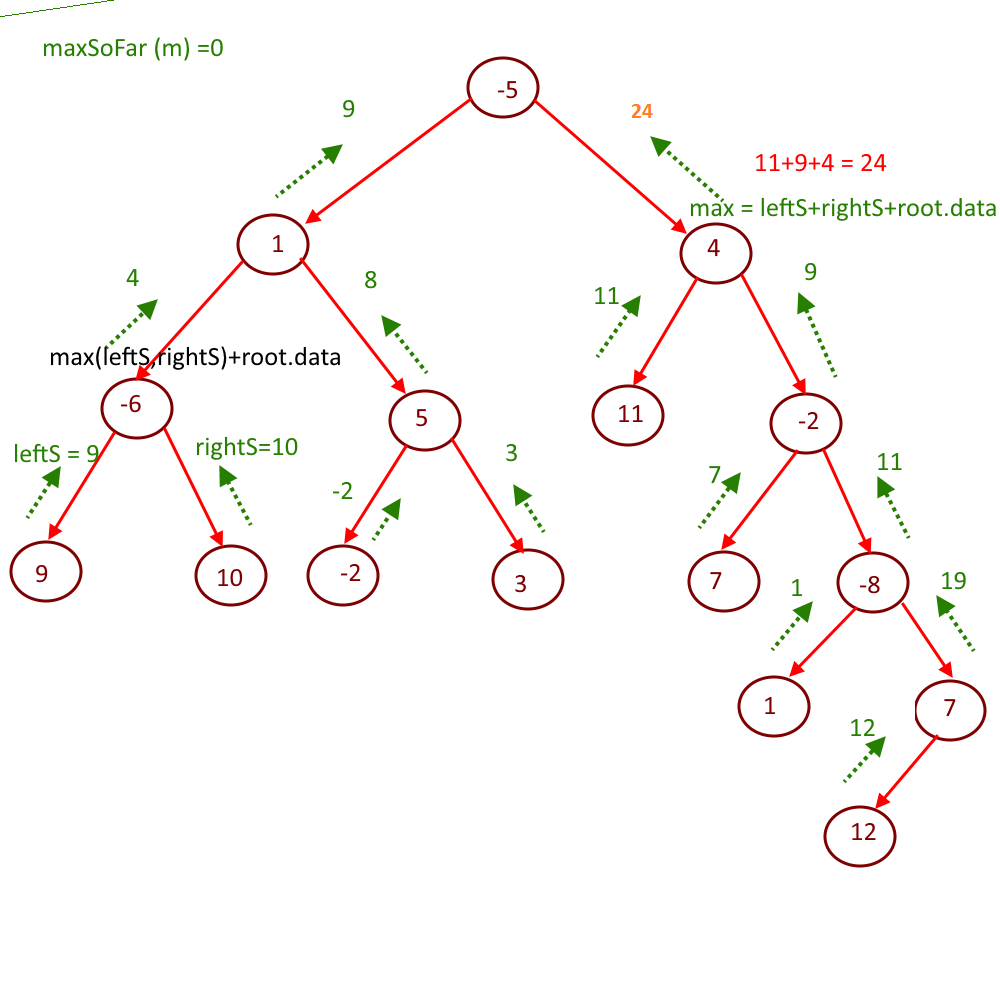
**Input:** A binary tree.

**Example:**

****

Approach:

* Now we will calculate the max path sum between two leaves node
* So our max path will be either on the left sub tree OR
* Our max path will be either on the right sub tree OR
* Our max path will have some part in left and some part in right and passes through through the root
* Take a variable say, "maxSoFar=0" this will our final result.
* Do postOrder traversal, This will give you result from left and right subtree
* Now at each node calcuate sumCurrent =Max of (result of leftSubtree,result of RightSubtree, result of leftSubtree+result of RightSubtree + Root data)
* if(maxSoFar<sumCurrent) then maxSoFar = sumCurrent
* At each node return max(result of leftSubtree,result of RightSubtree)+root.data;
* See Picture



Complete Code:

**package** MaxPathSumBwTwoLeaves;

//Now we will calculate the max path sum between two leaves node

//So our max path will be either on the left sub tree OR

//our max path will be either on the right sub tree OR

//Our max path will have some part in left and some part in right and passes through through the root

//Take a variable say, "maxSoFar=0" this will our final result.

//Do postOrder traversal, This will give you result from left and right subtree

//Now at each node calcuate sumCurrent =Max of (result of leftSubtree,result of RightSubtree, result of leftSubtree+result of RightSubtree + Root data)

//if(maxSoFar<sumCurrent) then maxSoFar = sumCurrent

//at each node return max(result of leftSubtree,result of RightSubtree)+root.data;

//See Picture

**public** **class** MaxPathSumBwTwoLeaves {

**public** **static** **int** *maxSoFar* =0;

**public** **int** maxPathSum(Node root){

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

**int** leftS = maxPathSum(root.left);

**int** rightS = maxPathSum(root.right);

**int** sumCurrent = Math.*max*(leftS+rightS+root.data , Math.*max*(leftS, rightS));

**if**(*maxSoFar*<sumCurrent){

*maxSoFar* = sumCurrent;

}

**return** Math.*max*(leftS,rightS)+root.data;

}

**else** **return** 0;

}

**public** **void** inorder(Node root){

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

inorder(root.left);

System.*out*.print(" " + root.data);

inorder(root.right);

}

}

**public** **static** **void** main(String args[]){

Node root = **new** Node(-5);

root.left = **new** Node(1);

root.right = **new** Node(4);

root.left.left = **new** Node(-6);

root.left.right = **new** Node(5);

root.left.right.left = **new** Node(-2);

root.left.right.right = **new** Node(3);

root.left.left.left = **new** Node(9);

root.left.left.right = **new** Node(10);

root.right.left = **new** Node(11);

root.right.right = **new** Node(-2);

root.right.right.right = **new** Node(-8);

root.right.right.left = **new** Node(7);

root.right.right.right.left = **new** Node(1);

root.right.right.right.right = **new** Node(7);

root.right.right.right.right.left = **new** Node(12);

MaxPathSumBwTwoLeaves m = **new** MaxPathSumBwTwoLeaves();

m.maxPathSum(root);

System.*out*.println("Max Path Sum Between Two Leaves is " + *maxSoFar*);

//m.inorder(root);

}

}

**class** Node{

**int** data;

Node left;

Node right;

**public** Node (**int** data){

**this**.data = data;

left = **null**;

right = **null**;

}

}

Output:

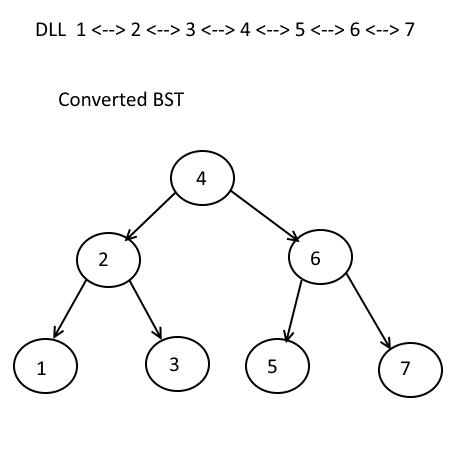
Max Path Sum Between Two Leaves is 24

**Algorithms - Convert a Sorted Doubly Linked List to Balanced BST.**

**Objective:** Given a sorted doubly linked list, convert it into Balanced binary search tree

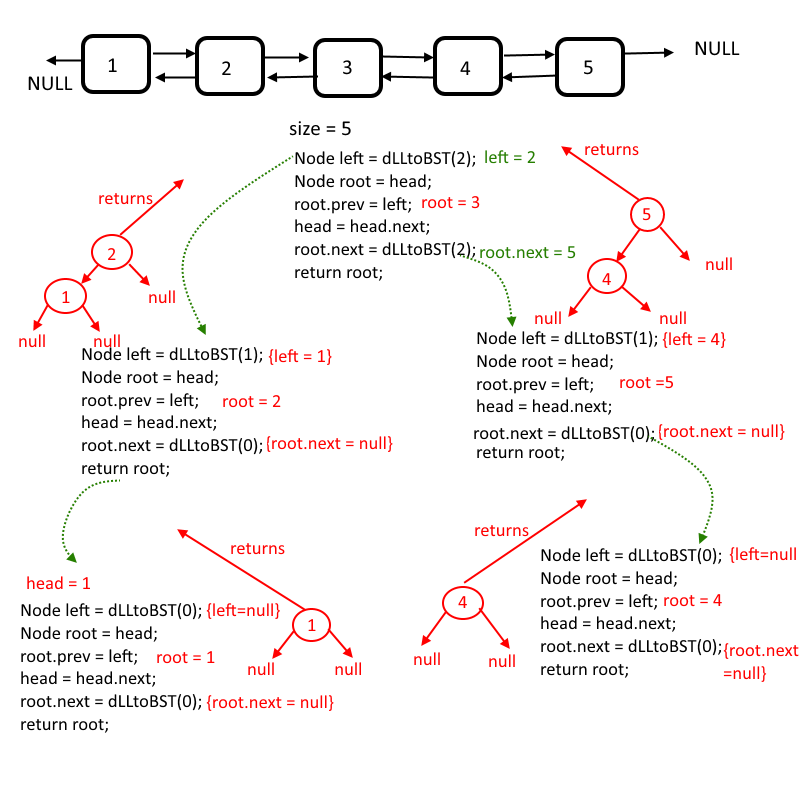
**Input:** A Doubly Linked List

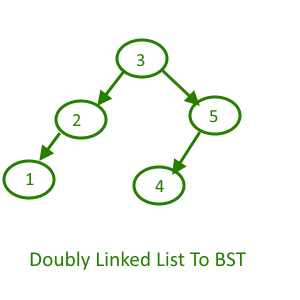
**Example:**

****

**Approach:**

1. Get the size of the Doubly Linked list.
2. Take left n/2 nodes and recursively construct left subtree
3. Make the middle node as root and assign the left subtree( constructed in step 2) to root's left.
4. Recursively construct right subtree and link it to the the right of root made in step 3.
5. See picture below





Complete Code:

**package** DLLtoBST;

**public** **class** DLLToBST {

**public** **static** Node *head* = **null**;

**public** **static** Node *tail* = **null**;

**public** **static** **int** *size* = 0;

**public** Node root;

**public** **void** add(**int** data) {

Node n = **new** Node(data);

**if** (*head* == **null**) {

*head* = n;

*tail* = n;

} **else** {

*head*.prev = n;

n.next = *head*;

*head* = n;

}

*size*++;

}

**public** Node dLLtoBST(**int** size) {

**if** (size <= 0) {

**return** **null**;

}

Node left = dLLtoBST(size / 2);

Node root = *head*;

root.prev = left;

*head* = *head*.next;

root.next = dLLtoBST(size-(size / 2)-1);

**return** root;

}

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

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

inOrder(root.prev);

System.*out*.print(" " + root.data);

inOrder(root.next);

}

}

**public** **void** printDLL(Node head) {

Node curr = head;

**while** (curr != **null**) {

System.*out*.print(" " + curr.data);

curr = curr.next;

}

System.*out*.println();

}

**public** **static** **void** main(String args[]) {

DLLToBST r = **new** DLLToBST();

r.add(9);

r.add(8);

r.add(7);

r.add(6);

r.add(5);

r.add(4);

r.add(3);

r.add(2);

r.add(1);

Node h = *head*;

System.*out*.println("DLL is : ");

r.printDLL(h);

Node x = r.dLLtoBST(*size*);

System.*out*.println("Inorder traversal of contructed BST");

r.inOrder(x);

}

}

**class** Node {

**int** data;

Node next;

Node prev;

**public** Node(**int** data) {

**this**.data = data;

**this**.next = **null**;

**this**.prev = **null**;

}

}

Output:

DLL is :

1 2 3 4 5 6 7 8 9

Inorder traversal of contructed BST

1 2 3 4 5 6 7 8 9

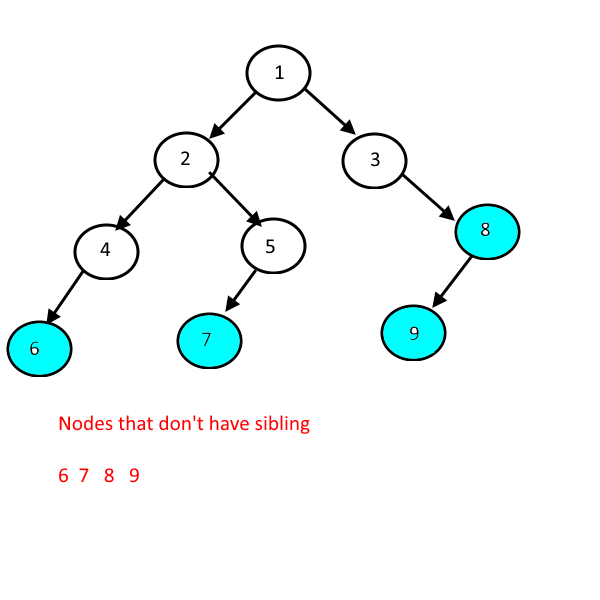
**Algorithms - Given a binary tree, Print All the Nodes that don't have siblings.**

**Objective:** Given a binary tree, Print All the Nodes that don't have siblings.

Note: sibling node is the node which has the same parent, so you need to print the nodes who is a single child of his parent.

**Input:** A binary tree.

**Example:**

****

Approach:

* Do the inorder traversal.
* check if node has only child, if yes then print that child.

Complete Code:

**package** PrintNoSiblingsNodes;

**public** **class** PrintNoSiblingsNodes {

**public** **void** printOnlyLeaves(Node root){

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

printOnlyLeaves(root.left);

**if**((root.left==**null**) && (root.right!=**null**)){

System.*out*.print(" " + root.right.data);

}

**if**((root.left!=**null**) && (root.right==**null**)){

System.*out*.print(" " + root.left.data);

}

printOnlyLeaves(root.right);

}

}

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

Node root = **new** Node(1);

root.left = **new** Node(2);

root.right = **new** Node(3);

root.left.left = **new** Node(4);

root.left.right = **new** Node(5);

root.left.right.left = **new** Node(7);

root.left.left.left = **new** Node(6);

root.right.right = **new** Node(8);

root.right.right.left = **new** Node(9);

PrintNoSiblingsNodes i = **new** PrintNoSiblingsNodes();

System.*out*.print("Nodes with No Siblings: ");

i.printOnlyLeaves(root);

}

}

**class** Node{

**int** data;

Node left;

Node right;

**public** Node(**int** data){

**this**.data = data;

**this**.left = **null**;

**this**.right =**null**;

}

}

Output:

Nodes with No Siblings: 6 7 8 9

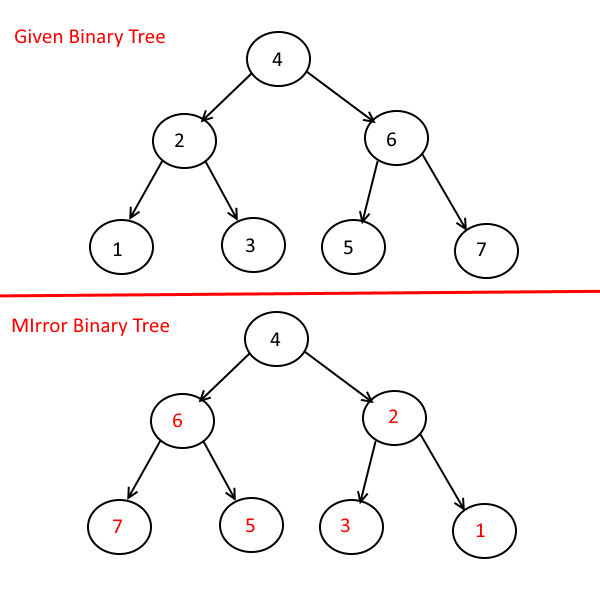
**Algorithms - Given a binary tree, Convert it into its Mirror Tree**

**Objective:** Given a binary tree, Convert it into its Mirror Tree

Mirror Tree: Mirror Tree of a binary tree is where left and right child of every node of given binary tree is interexchnaged.

**Input:** A binary tree.

**Example:**



**Approah:**

* Do the PreOrder Traversal.
* Starting from the root, swap the left and right childs.
* Recursively call the same procedure in the leftsubtree and right subtree.

Complete Code:

**package** MirrorBinaryTree;

**public** **class** MirrorTree {

**public** **void** mirror(Node root){

print(root);

Node x = mirrorTree(root);

System.*out*.print("\n Mirror Image ");

print(x);

}

**public** Node mirrorTree(Node root){

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

Node t = root.left;

root.left = root.right;

root.right = t;

mirrorTree(root.right);

mirrorTree(root.left);

}

**return** root;

}

**public** **void** print(Node root){

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

print(root.left);

System.*out*.print(" " + root.data);

print(root.right);

}

}

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

Node root = **new** Node(4);

root.left = **new** Node(2);

root.right = **new** Node(6);

root.left.left = **new** Node(1);

root.left.right = **new** Node(3);

root.right.left = **new** Node(5);

root.right.right = **new** Node(7);

MirrorTree i = **new** MirrorTree();

i.mirror(root);

}

}

**class** Node{

**int** data;

Node left;

Node right;

**public** Node(**int** data){

**this**.data = data;

**this**.left = **null**;

**this**.right =**null**;

}

}

**Output:**

1 2 3 4 5 6 7

Mirror Image 7 6 5 4 3 2 1

**Algorithms - Given a Sorted Array, Convert it into its Balanced Binary search Tree.**

**Objective:** You have been given a sorted array of intergers, you need to convert it into balanced binary search tree.

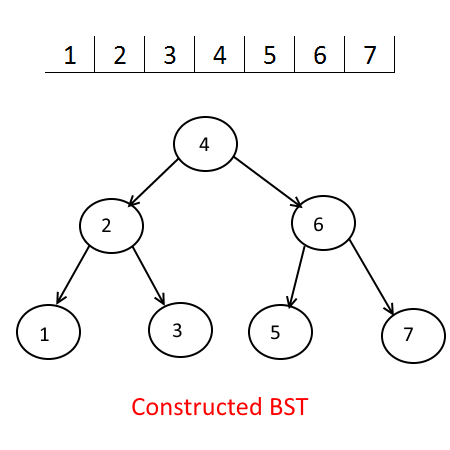
*Why balanced binary tree is important:*

*You can also create first element as root and insert all other elements to the right of the tree because array is in increasing order but this constructed tree won't be balanced tree, it wil be skwed tree and to perform operations on this tree will be O(n) not O(logn)*

**Input:** An sorted array of integers

**Output:** Balanced Binary Tree

**Example:**

****

**Approach:**

* Its a simple appraoch.
* Make the middle element of the array and make it root.
* Recusively solve left half of the array and add it to root.left.
* Recusively solve right half of the array and add it to root.right.

Complete Code:

**package** SortedArrayToBST;

**public** **class** SortedArrayToBST {

**public** BSTNode convert(**int** [] arrA, **int** start, **int** end){

**if**(start>end){

**return** **null**;

}

**int** mid = (start + end)/2;

BSTNode root = **new** BSTNode(arrA[mid]);

root.left = convert(arrA, start, mid-1);

root.right =convert(arrA, mid+1, end);

**return** root;

}

**public** **void** displayTree(BSTNode root){

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

displayTree(root.left);

System.*out*.print(" " + root.data);

displayTree(root.right);

}

}

**public** **static** **void** main(String args[]){

**int** [] arrA = {2,3,6,7,8,9,12,15,16,18,20};

SortedArrayToBST s = **new** SortedArrayToBST();

BSTNode x = s.convert(arrA, 0, arrA.length-1);

System.*out*.println("Tree Display : ");

s.displayTree(x);

}

}

**class** BSTNode{

**int** data;

BSTNode left;

BSTNode right;

**public** BSTNode(**int** data){

**this**.data = data;

left = **null**;

right = **null**;

}

}

**Output:**

Tree Display :

2 3 6 7 8 9 12 15 16 18 20

**Algorithms - Given a Sorted Singly Linked List Array, Convert it into its Balanced Binary search Tree.**

**Objective:** You have been given a sorted singly List, you need to convert it into balanced binary search tree.

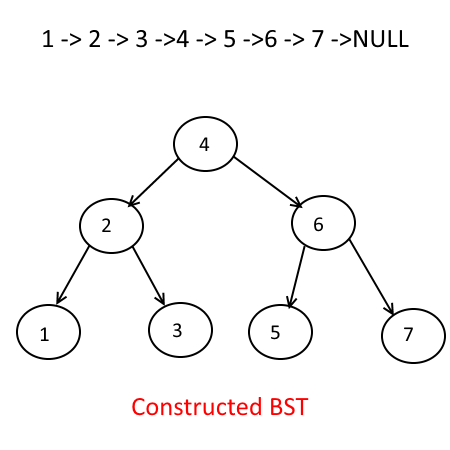
*Why balanced binary tree is important:*

*You can also create first node as root and insert all other nodes to the right of the tree because List is in increasing order but this constructed tree won't be balanced tree, it wil be skwed tree and to perform operations on this tree will be O(n) not O(logn).*

**Input:** An sorted Singly Linked List

**Output:** Balanced Binary Tree

**Example:**

****

**Appraoch:**

* Say *mid* is the middle node in the linked list.
* Recursively construct left subtree from ***start to mid-1***
* Make the middle node as root and assign the left subtree to it.
* Recursively construct right subtree from ***mid+1 to end.***
* Assign the right subtree to root.

NOTE: This problem is very smililar to " **Given a Sorted Array, Convert it into its Balanced Binary search Tree"** and **"Convert a Sorted Doubly Linked List to Balanced BST" .** Details explanation is given here

Complete Code:

**package** SortedLLToBST;

**public** **class** SortedLLToBST {

**public** **static** Node *head* = **null**;

**public** BSTNode LLToBST(**int** start, **int** end){

**if**(start>end)**return** **null**;

**int** mid = (start+end)/2;

BSTNode leftChild = LLToBST(start,mid-1);

BSTNode root = **new** BSTNode(*head*.data);

root.left = leftChild;

*head* = *head*.next;

root.right = LLToBST(mid+1,end);

**return** root;

}

**public** **int** getSize(){

Node curr = *head*;

**int** size =0;

**while**(curr!=**null**){

curr=curr.next;

size++;

}

**return** size;

}

**public** **void** inorder(BSTNode root){

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

inorder(root.left);

System.*out*.print(" " + root.data);

inorder(root.right);

}

}

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

*head* = **new** Node(1);

*head*.next = **new** Node(2);

*head*.next.next = **new** Node(3);

*head*.next.next.next = **new** Node(4);

*head*.next.next.next.next = **new** Node(5);

Node tail = **new** Node(6);

*head*.next.next.next.next.next = tail;

SortedLLToBST i = **new** SortedLLToBST();

BSTNode x = i.LLToBST(1,i.getSize()) ;

System.*out*.print("Constructed BST is :");

i.inorder(x);

}

}

**class** Node{

**int** data;

Node next;

**public** Node(**int** data){

**this**.data = data;

next = **null**;

}

}

**class** BSTNode{

**int** data;

BSTNode left;

BSTNode right;

**public** BSTNode(**int** data){

**this**.data = data;

left = **null**;

right = **null**;

}

}

**Output:**

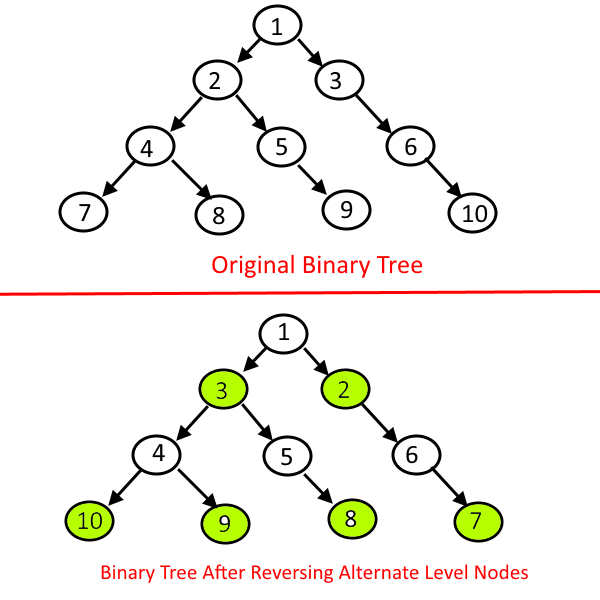
Constructed BST is : 1 2 3 4 5 6

**Algorithms - Reverse Alternate levels of a given binary tree.**

**Objective:** **-** Reverse Alternate levels of a given binary tree

**Input:** A binary tree

**Example:**

****

**Appraoch:**

* Do the inorder traversal and store all the alternate level nodes in an ArrayList.
* Reverse the ArrayList
* Do another inorder traversal and place the reversed array list in the same order in which it was fetched in step one.
* Look at the code for clear explanation.

Complete Code:

**Note**: *levelOrderQueue* function is used for printing each level in new line**.**

**package** ReverseAlternateLevelsOfBinaryTree;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.LinkedList;

**import** java.util.Queue;

**public** **class** ReverseAlternateLevelsOfBinaryTree {

**public** **static** ArrayList *al*;

**public** **void** storeAlterNateLevels(Node root,**int** level){

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

storeAlterNateLevels(root.left,level+1);

**if**(level%2!=0){

*al*.add(root.data);

}

storeAlterNateLevels(root.right,level+1);

}

}

**public** **void** reverseAlterNateLevels(Node root,**int** level){

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

reverseAlterNateLevels(root.left,level+1);

**if**(level%2!=0){

root.data = (Integer)*al*.remove(0);

}

reverseAlterNateLevels(root.right,level+1);

}

}

**public** **void** levelOrderQueue(Node root){

Queue q = **new** LinkedList();

**int** levelNodes =0;

**if**(root==**null**) **return**;

q.add(root);

**while**(!q.isEmpty()){

levelNodes = q.size();

**while**(levelNodes>0){

Node n = (Node)q.remove();

System.*out*.print(" " + n.data);

**if**(n.left!=**null**) q.add(n.left);

**if**(n.right!=**null**) q.add(n.right);

levelNodes--;

}

System.*out*.println("");

}

}

**public** **void** inorder(Node root){

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

inorder(root.left);

System.*out*.print(" " + root.data);

inorder(root.right);

}

}

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

Node root = **new** Node(1);

root.left = **new** Node (2);

root.right = **new** Node (3);

root.left.left = **new** Node (4);

root.left.right = **new** Node (5);

root.right.left = **new** Node (6);

root.right.right = **new** Node (7);

root.left.left.left = **new** Node (8);

root.left.left.right = **new** Node (9);

root.left.right.left = **new** Node (10);

root.left.right.right = **new** Node (11);

root.right.left.left = **new** Node (12);

root.right.left.right = **new** Node (13);

root.right.right.left = **new** Node (14);

root.right.right.right = **new** Node (15);

ReverseAlternateLevelsOfBinaryTree i = **new** ReverseAlternateLevelsOfBinaryTree();

*al* = **new** ArrayList();

System.*out*.println(" Orininal Tree");

//i.inorder(root);

i.levelOrderQueue(root);

i.storeAlterNateLevels(root,0);

// System.out.println(" LIst " + al);

Collections.*reverse*(*al*);

i.reverseAlterNateLevels(root,0);

System.*out*.println("\n New Tree, Alternate Levels Reversed..");

//i.inorder(root);

i.levelOrderQueue(root);

}

}

**class** Node{

**int** data;

Node left;

Node right;

**public** Node (**int** data){

**this**.data = data;

left = **null**;

right = **null**;

}

}

Output:

Orininal Tree

1

2 3

4 5 6 7

8 9 10 11 12 13 14 15

New Tree, Alternate Levels Reversed..

1

3 2

4 5 6 7

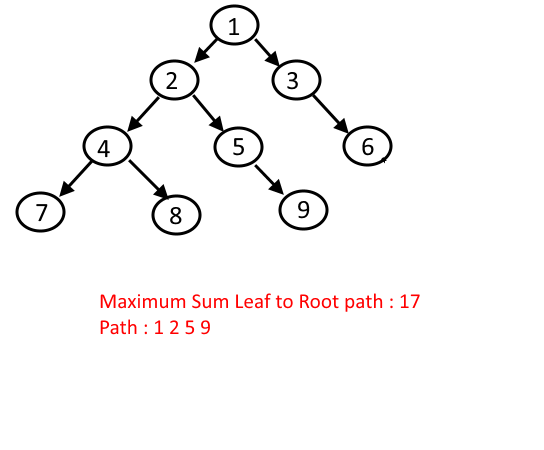
15 14 13 12 11 10 9 8

**Algorithms -** Find the Maximum Sum Leaf to Root path in a Given Binary Tree

**Objective:** **-** Find the maximum sum leaf to root path in a Binary Tree. Means in all the paths from root to leaves, find the path which has the maximum sum.

**Input:** A binary tree

**Example:**

****

**Appraoch:**

This solution will be divided into two parts

**Find the leaf which has the maximum sum from root.**

* Take a global variable maxLeaf and maxSum. (this maxLeaf will the node which has the maximum sum path and maxSum will the maximum sum.)
* Do the preorder traversal
* At each node, maintain a another variable sum = sum + root.data.
* When ever you reach to any leaf, check if sum>maxSum, if yes then update the maxLeaf and maxSum.

**Print the path from root to that leaf.**

Please refer this link to print the path.

Complete Code:

**package** MaxSumLeafToRoot;

**public** **class** MaxSumLeafToRoot {

**static** **int** *maxSum* = -100000;

**static** Node *maxLeaf*=**null**;

**static** **int** *currentSum* =0;

**public** **void** maxSum(Node root, **int** sum){

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

sum=sum+root.data;

**if**(sum>*maxSum* && root.left==**null** && root.right==**null**){

*maxLeaf* = root;

*maxSum* = sum;

}

// System.out.println("Sum " + sum);

maxSum(root.left,sum);

maxSum(root.right,sum);

}

}

**public** Boolean path(Node root, Node leaf){

**if**(root==**null**) **return** **false**;

**if** ((root == leaf) || path(root.left, leaf) ||path(root.right, leaf) )

{

System.*out*.print(" " + root.data);

**return** **true**;

}

**return** **false**;

}

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

Node root = **new** Node(1);

root.left = **new** Node (2);

root.right = **new** Node (3);

root.left.left = **new** Node (4);

root.left.right = **new** Node (5);

root.right.left = **new** Node (6);

root.right.left.left = **new** Node (8);

MaxSumLeafToRoot i = **new** MaxSumLeafToRoot();

i.maxSum(root,0);

System.*out*.println("Maximum Sum Leaf to Root path " + maxSum);

System.*out*.print("Path :");

i.path(root,*maxLeaf*);

}

}

**class** Node{

**int** data;

Node left;

Node right;

**public** Node (**int** data){

**this**.data = data;

left = **null**;

right = **null**;

}

}

Output:

Maximum Sum Leaf to Root path :18

Path : 8 6 3 1

**Algorithms -** Make a Binary Tree from Given Inorder and Preorder Traveral.

**Objective:** **-** Find the maximum sum leaf to root path in a Binary Tree. Means in all the paths from root to leaves, find the path which has the maximum sum.

**Input:** A binary tree

**Example:**