UNIT–I

Introduction:Algorithm Analysis, Space and Time Complexity analysis, Asymptotic Notations.
AVL Trees – Creation, Insertion, Deletion operations.
B-Trees – Creation, Insertion, Deletion operations.

INTRODUCTION TO ALGORITHMS: WHAT IS AN ALGORITHM?

Informal Definition:

An Algorithm is any well-defined computational procedure that takes some value or set of values as Input and produces a set of values or some value as output. Thus algorithm is a sequence of computational steps that transforms the input into the output.

Formal Definition:

An Algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition,all algorithms should satisfy the following criteria.

- **INPUT**→Zero or more quantities are externally supplied.
- **OUTPUT**→Atleast one quantity is produced.
- **DEFINITENESS**→Each instruction is clear and unambiguous.
- **FINITENESS** → If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
- **EFFECTIVENESS** →Every instruction must very basic so that it can be carried out, in principle,by a person using only pencil& paper.

Issues or study ofAlgorithm:

1. How to device or design an algorithm→creating and algorithm.
2. How to express an algorithm→definiteness.
3. How to analysis an algorithm→time and space complexity.
4. How to validate an algorithm→fitness.
5. Testing the algorithm →checking for error.

AlgorithmSpecification:

Algorithm can be described in three ways.

1. **Natural language like English:**

When this way is choused care should be taken, we should ensure that each &every statement is definite.

2. **Graphic representation called flowchart:**

   This method will work well when the algorithm is small&simple.

3. **Pseudo-code Method:**

   In this method,we should typically describe algorithms as program, which resembles language like Pascal & algol.

PSEUDO-CODEFOREXPRESSINGANALGORITHM:

1. Comments begin with//and continue until the end of line.
2. Blocks are indicated with matching braces{and}.
3. An identifier begins with a letter.The data types of variables are not explicitly declared.
4. Compound data types can be formed with records. Here is an example,

```
Node. Record
{
  datatype–1data-1;
        .
        .
        .
  data type – n data – n;
  node * link;
}
```

Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period.

5. Assignment of values to variables is done using the assignment statement.

```
<Variable>:=<expression>;
```

6. There are two Boolean values TRUE and FALSE.

   →Logical Operators AND,OR,NOT

   →Relational Operators<,<=,>,>=,=,!=

7. The following looping statements are employed.

   For, while and repeat-until

While Loop:

```
While<condition>do
{
        <statement-1>
               .
               .
               .
        <statement-n>
}
```

ForLoop:

```
  Forvariable:=value-1tovalue-2stepstepdo
{
   <statement-1>
          .
          .
          .
<statement-n>
}
```

repeat-until:

```
          repeat
                  <statement-1>
                          .
                          .
                          .
                  <statement-n>
          until<condition>
```

8. A conditional statement has the following forms.

   →If<condition>then<statement>

   → If<condition> then <statement-1>

   Else <statement-1>

   Case statement:

```
Case
{
        :<condition-1>:<statement-1>
                       .
                       .
                       .
        :<condition-n>:<statement-n>
        :else:<statement-n+1>
}
```

9. Input and output are done using the instructions read&write.
10. There is only one type of procedure: Algorithm, the heading takes the form,

    Algorithm Name (Parameter lists)

→As an example, the following algorithm fields & returns the maximum of 'n' given numbers:

1. algorithmMax(A,n)
2. //A is an array of size n

```
3.  {
4.   Result:=A[1];
5.   For I:=2 to n do
6.      If A[I]>Result then
7.           Result:=A[I];
8.     Return Result;
9.}
```

In this algorithm (named Max),A & n are procedure parameters. Result & I are Local variables.

→Next we present 2 examples to illustrate the process of translation problem into an algorithm.

Selection Sort:

- Suppose we Must devise an algorithm that sorts a collection of n>=1elements of arbitrary type.
- A Simple solution given by the following.
- (From those elements that are currently unsorted,find the smallest &place it next in the sorted list.)

Algorithm:

```
1. For i:=1 to n do
2. {
3.          Examine a[I]to a[n] and suppose the smallest element is at a[j];
4.          Interchange a[I] and a[j];
5.}
```

→Finding the smallest element(sata[j])and interchanging it witha[i]

- We can solve the latter problem using the code,

-
  ```
   t:= a[i];
  a[i]:=a[j];
  a[j]:=t;
  ```
- The first subtask can be solved by assuming the minimum is a[I];checking a[I] with a[I+1],a[I+2]…….,and whenever a smaller element is found,regarding it as the new minimum.a[n]is compared with the current minimum.
- Putting all these observations together,we get the algorithm Selectionsort.

Theorem: Algorithm selection sort(a,n) correctly sorts a set of n>=1 elements .The result remains is a a[1:n]such that a[1] <= a[2] ….<=a[n].

SelectionSort:

Selection Sort begins by finding the least element in the list. This element is moved to the front. Then the least element among the remaining element is found out and put into second position.This procedure is repeated till the entire list has been studied.

Example: ListL=3,5,4,1,2

1isselected,→1,5,4,3,2

2isselected,→1,2,4,3,5

3isselected,→1,2,3,4,5

4isselected,→1,2,3,4,5

Proof:

- WefirstnotethatanyI,sayI=q,followingtheexecutionoflines6to9,itisthe case that a[q] Þ a[r],q<r<=n.
- Alsoobservethatwhen'i'becomesgreaterthanq,a[1:q]isunchanged.Hence, following the last execution of these lines(i.e.I=n).Wehavea[1]<=a[2] <=……a[n].
- We observe this point that the upper limit of the for loop in the line 4can be changed ton-1without damaging the correctness of the algorithm.

Algorithm:

```
1. Algorithmselectionsort(a,n)
2. //Sortthearraya[1:n] intonon-decreasingorder. 3.{
4.        forI:=1tondo
5.        {
6.                j:=I;
7.                fork:=i+1tondo
8.                        if(a[k]<a[j])
9.                        t:=a[I];
10.                       a[I]:=a[j];
11.                       a[j]:=t;
12.       }
13.}
```

PERFORMANCEANALYSIS:

1. **Space Complexity:**
   The space complexity of an algorithm is the amount of money it needs to run to compilation.

2. **Time Complexity:**
   The time complexity of an algorithm is the amount of computer timeit needs to run to compilation.

Space Complexity:

Space Complexity Example:

```
Algorithm abc(a,b,c)
{
Return a+b++*c+(a+b-c)/(a+b)+4.0;
}
```

→ The Space needed by each of these algorithms is seen to be the sum of the following component.

1.**A fixed part** that is independent of the characteristics (eg:number,size)of the inputs and outputs.

The part typically includes the instruction space (ie. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on.

variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referencedvariables(totheextentthatisdependsoninstancecharacteristics),and the recursion stack space.

a. The space requirement s(p) of any algorithm p may therefore be written as,

S(P) = c+ Sp(Instance characteristics)Where 'c' is a constant.

Example: Algorithm sum(a,n)

```
{
    s=0.0;
    forI=1tondo s=
    s+a[I];
```

```
            return s;
        }
```

- The problem instances for this algorithm are characterized by n,the number of elements to be summed.The space needed by ‘n’is one word, since it is of type integer.
- The space needed by ‘a’a is the space needed by variables of type array of floating point numbers.
- This is atleast‘n’words,since‘a’must be large enough to hold the‘n’elements to be summed.
- So,we obtain S sum(n)>=(n+s)[nfora[],one each forn,Ia&s]

TimeComplexity:

The time T(p) taken by a program P is the sum of the compile time and the run time(execution time)

→The compiletime does not depend on the instance characteristics.Also we may assume that a compiled program will be run several times without recompilation .This rum time is denoted by tp(instance characteristics).

→ The number of steps any problem statemnt is assigned depends on the kind of statement.

For example, comments → 0 steps.

Assignment statements →1steps.[Which does not involve any calls to other algorithms]

Interactive statement such as for,while &repeat-until→Control part of the statement.

1. **We introduce a variable, count into the program statement to increment count with initial value 0.Statement to increment count by the appropriate amount are introduced into the program.**

This is done so that each time a statement in the original program is executes count is incremented by the stepcount of that statement.

Algorithm:

```
Algorithm sum(a,n)
    {
      s=0.0;
      count=count+1; for
      I=1 to n do
{
 count=count+1;
s=s+a[I];
count=count+1;
}
count=count+1;
count=count+1;
return s;
}
```

→If the count is zero to start with, then it will be2n+3 on termination. So each invocation of sum execute a total of 2n+3 steps.

2. These cond method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.

→First determine the number of steps per execution(s/e) of the statement and the total number of times (ie., frequency)each statement is executed.

→By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.

Statement	**S/e**	**Frequency**	**Total**
1.Algorithm Sum(a,n)	0	-	0
2.{	0	-	0
3. S=0.0;	1	1	1
4. forI=1tondo	1	n+1	n+1
5. s=s+a[I];	1	n	n
6. returns;	1	1	1
7.}	0	-	0
Total			2n+3

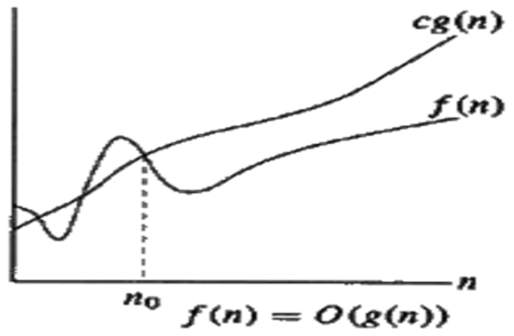
ASYMPTOTICNOTATIONS

There are different kinds of mathematical notations used to represent timecomplexity. These are called Asymptoticnotations.They are as follows:

1. Bigoh(O)notation
2. Omega(Ω)notation
3. Theta(ө)notation

1. Bigoh(O)notation:

- Bigoh(O)notation is used to represent upper bound of algorithm runtime.
- Let f(n) and g(n) are two non-negative functions
- The function f(n)=O(g(n))if and only if there exists positive constants can dn$_0$ such that f(n)≤c*g(n)for all n , n ≥ n$_0$.



Example:

If f(n)=3n+2thenprovethatf(n)=O(n)

Let f(n) =3n+2,c=4,g(n) =n if

n=1 3n+2 ≤ 4n

3(1)+2≤4(1)

3+2≤ 4

5≤4(F)

If n=2 3n+2≤4n

3(2)+2≤ 4(2)

8≤8(T)

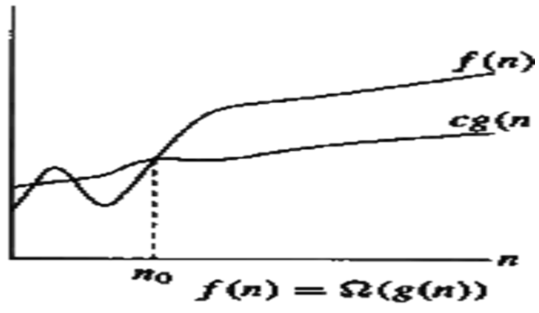
3n+2≤4nforalln ≥2

This is in the form of f(n) ≤ c*g(n) for all n ≥ n$_{0,}$where c=4, n$_0$ =2

Therefore, f(n) = O(n),

2. Omega(Ω)notation:

- Bigoh(O)notation is used to represent lower bound of algorithm runtime.
- Let f(n) and g(n) are two non-negative functions
- The function f(n)=Ω(g(n))if and only if there exists positive constants can dn$_0$ Such that f(n)≥c*g(n)for all n,n≥n$_0$.



Example

f(n)=3n+2thenprovethatf(n)=Ω(g(n))

Let f(n) =3n+2,c=3,g(n) =n if

n=1 3n+2 ≥ 3n

3(1)+2≥3(1)

5≥3(T)

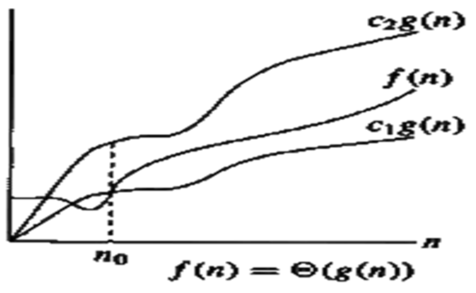
3n+2≥4nforalln ≥1

This is in the form of f(n) ≥ c*g(n) for all n ≥ n_0,where c=3, n_0 =1

Therefore, f(n) = Ω(n).

3. Theta(ɵ)notation:

- Theta(ɵ)notation is used to represent the running time between upperbound and lower bound.
- Let f(n) and g(n)be two non-negative functions.
- The function f(n) = θ(g(n)) if and only if there exists positive constants c_1, $c_2$and n_0 such that c_1*g(n) ≤ f(n)≤c_2* g(n) for all n, n≥n_0 .



Example:

f(n)=3n+2thenProvethatf(n)=θ(g(n))

Lower bound=3n+2≥3n for all n≥1

c_1=3,g(n)=n,n_0=1

Upper Bound=3n+2≤4n for all n≥2

$c2=4, g(n)=n, n0=2$

$3(n) \leq 3n+2 \leq 4(n)$ for all n, $n \geq 2$

This is in the form of $c_1*g(n) \leq f(n) \leq c_2*g(n)$ for all $n \geq n_0$ Where $c_1=3, c_2=4$, $g(n)=n$, $n_0=2$

Therefore $f(n)=\theta(n)$

POLYNOMIALVSEXPONENTIALALGORITHMS

The **time complexity**(generally referred as running time)of an algorithm is expressed as **the amount of time taken by an algorithm for some size of the input to the problem**. **Big O notation** is commonly used to express the time complexity of any algorithm as this suppresses the lower order terms and is described asymptotically. Time complexity is estimated by counting the operations(provided as instructions in a program) performed in an algorithm. Here each operation takes a fixed amount of time in execution. **Generally time complexities are classified as constant, linear, logarithmic, polynomial, exponential etc**. Among these the **polynomial and exponential are the most prominently considered** and defines the complexity of an algorithm. These two parameters for any algorithm are always influenced by size of input.

Polynomial Running Time:

An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is $O(n^k)$for some non-negative integer k,where n is the complexity of the input. Polynomial-time algorithms are said to be "fast." Most familiar mathematical operations such as addition, subtraction, multiplication, and division, as well as computing square roots, powers, and logarithms, can be performed in polynomial time. Computing the digits of most interesting mathematical constants, including pi and e, can also be done in polynomial time.

All basic arithmetic operations ((i.e.) Addition, subtraction, multiplication, division), comparison operations, sort operations are considered as polynomial time algorithms.

Exponential Running Time:

The set of problems which can be solved by an exponential time algorithms, but for which no polynomial time algorithms is known.

An algorithm is said to be exponential time, if $T(n)$ is upper bounded by $2^{\mathrm{poly}(n)}$, where poly(n) is some polynomial in n. More formally, an algorithm is exponential time if $T(n)$ is bounded by $O(2^{n^k})$ for some constant k.

Algorithms which have exponential time complexity grow much faster than polynomial algorithms.

The difference you are probably looking for happens to be where the variable is in the equation that expresses the run time. Equations that show a **polynomial time** complexity have variables in the bases of their terms.

Examples:n^3+2n^2+1.Noticenisinthebase,NOTtheexponent.

In exponential equations, the variable is in the exponent.

Examples: 2^n. As said before, exponential time grows much faster. If n is equal to 1000 (a reasonable input for an algorithm),then notice$1000^3$is 1billion,and 2^{1000}issimplyhuge! For a reference, there are about 2^{80} hydrogen atoms in the sun, this is much more than 1 billion.

AVERAGE,BEST AND WORSTCASE COMPLEXITIES

Bestcase:This analysis constraints on the input,other than size.Resulting in the fasters possible run time

Worstcase:This analysis constraints on the input,other than size.Resulting in the fasters possible run time

Averagecase: This type of analysis results in average running time over every type of input.

Complexity: Complexity refers to the rate at which the storage time grows as a function of the problem size.

ANALYSING RECURSIVE PROGRAMS.

For every recursive algorithm,we can write recurrence relation to analyse the time complexity of the algorithm.

Recurrence relation of recursive algorithms

A recurrence relation is an equation that defines a sequence where any term is defined in terms of its previous terms.

The recurrence relation for the time complexity of some problems are given below:

FibonacciNumber

$T(N)=T(N-1)+T(N-2)$

BaseConditions:$T(0)=0$and$\mathbf{T}(1)=1$

BinarySearch

$T(N)= T(N/2)+C$

BaseCondition:$T(1)=1$

MergeSort

T(N)=2T(N/2)+CN

BaseCondition:T(1)=1

Recursive Algorithm:Finding min and max in an array

T(N)=2T(N/2)+2

BaseCondition:T(1)=0and**T**(2)=1

QuickSort

T(N)=T(i)+T(N-i-1)+CN

The time taken by quick sort depends upon the distribution of the input array and partition strategy. T(i) and T(N-i-1) are two smaller sub problems after the partition where i is the number of elements that are smaller than the pivot. **CN** is the time complexity of the partition process where **C** is a constant. .

Worst Case:This is a case of the unbalanced partition where the partition process always picks the greatest or smallest element as a pivot(Think!).For the recurrence relation of the worst case scenario, we can put i = 0 in the above equation.

T(N)= T(0)+ T(N-1)+CN

Which is equivalent to

T(N)=T(N-1)+CN

BestCase:This is a case of the balanced partition where the partition process always picks the middle element as pivot.For the recurrence relation of the worstcase scenario,puti= N/2 in the above equation.

T(N)=T(N/2)+T(N/2-1)+CN

Which is equivalent to

T(N)=2T(N/2)+CN

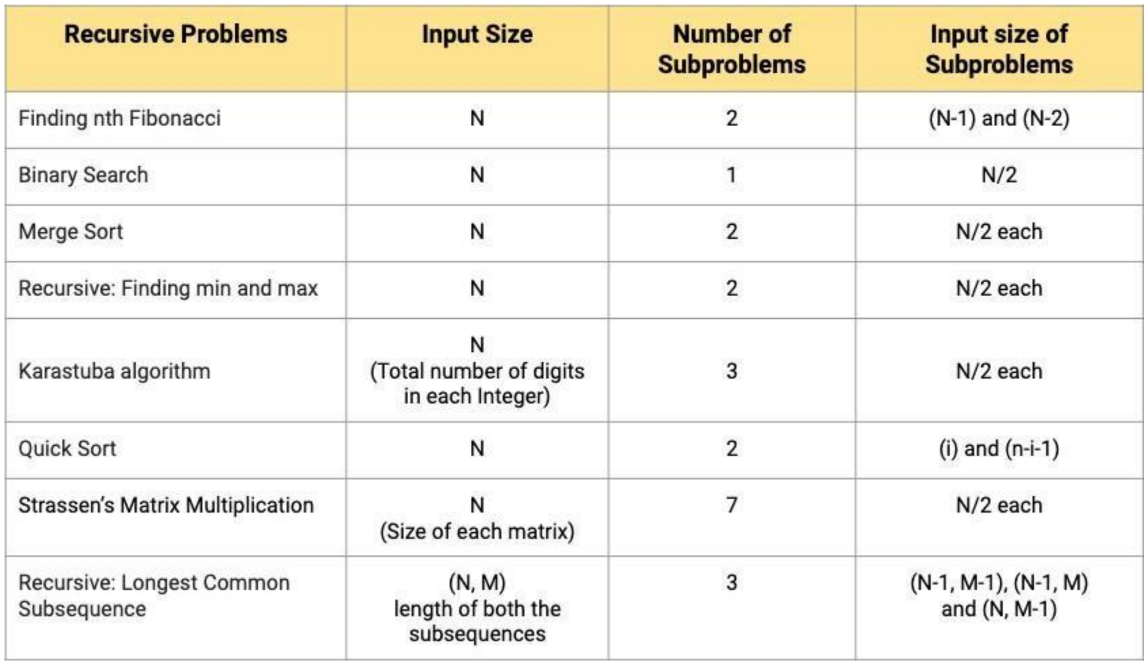
AverageCase: *For* average case analysis, we need to consider all possible permutation of input and time taken by each permutation.

T(N)=(**for** i=0toN-1)∑(T(i)+T(N-i-1))/N

Note: This looks mathematically complex but we can find several other intuitive ways to analyse the average case of quick sort.

Analyzing the Efficiency of Recursive Algorithms

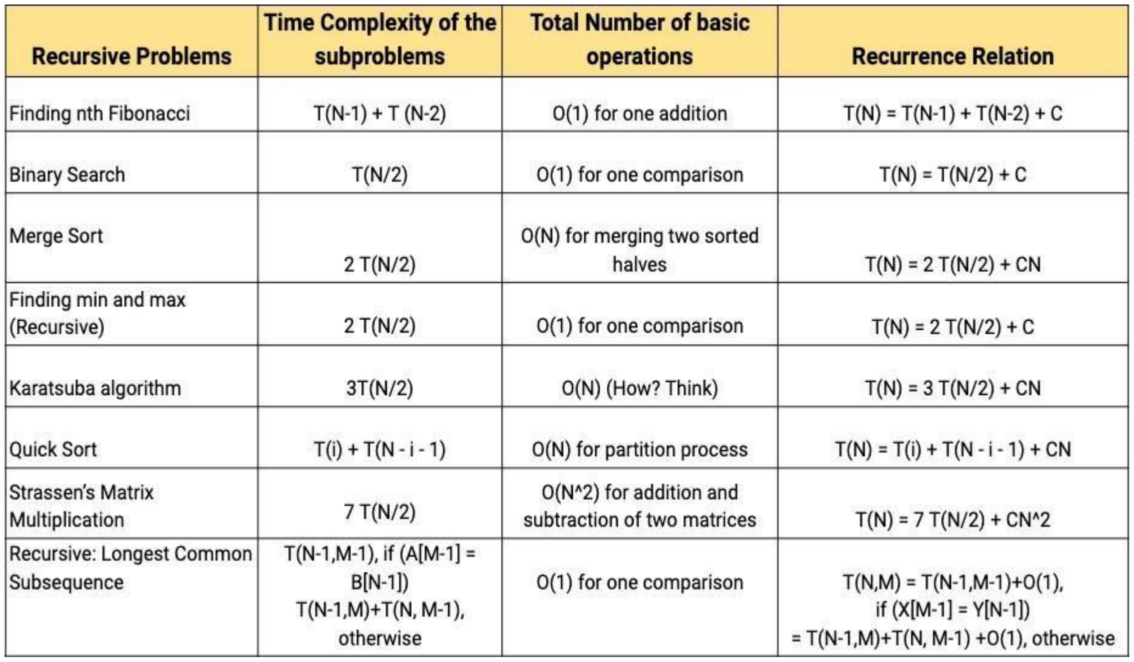
Step1: Identify the number of sub-problems and a parameter (or parameters) indicating an input's size of each sub-problem (function call with smaller input size)



Recursive Problems	Input Size	Number of Subproblems	Input size of Subproblems
Finding nth Fibonacci	N	2	(N-1) and (N-2)
Binary Search	N	1	N/2
Merge Sort	N	2	N/2 each
Recursive: Finding min and max	N	2	N/2 each
Karastuba algorithm	N (Total number of digits in each Integer)	3	N/2 each
Quick Sort	N	2	(i) and (n-i-1)
Strassen's Matrix Multiplication	N (Size of each matrix)	7	N/2 each
Recursive: Longest Common Subsequence	(N, M) length of both the subsequences	3	(N-1, M-1), (N-1, M) and (N, M-1)

Step 2: Addthetimecomplexitiesofthesub-problemsand the totalnumberofbasic operations performed at that stage of recursion.

Step3: Set up a recurrence relation,with a correct base condition, for the numberoftimes the basic operation is executed.



Recursive Problems	Time Complexity of the subproblems	Total Number of basic operations	Recurrence Relation
Finding nth Fibonacci	T(N-1) + T (N-2)	O(1) for one addition	T(N) = T(N-1) + T(N-2) + C
Binary Search	T(N/2)	O(1) for one comparison	T(N) = T(N/2) + C
Merge Sort	2 T(N/2)	O(N) for merging two sorted halves	T(N) = 2 T(N/2) + CN
Finding min and max (Recursive)	2 T(N/2)	O(1) for one comparison	T(N) = 2 T(N/2) + C
Karatsuba algorithm	3T(N/2)	O(N) (How? Think)	T(N) = 3 T(N/2) + CN
Quick Sort	T(i) + T(N - i - 1)	O(N) for partition process	T(N) = T(i) + T(N - i - 1) + CN
Strassen's Matrix Multiplication	7 T(N/2)	O(N^2) for addition and subtraction of two matrices	T(N) = 7 T(N/2) + CN^2
Recursive: Longest Common Subsequence	T(N-1,M-1), if (A[M-1] = B[N-1]) T(N-1,M)+T(N, M-1), otherwise	O(1) for one comparison	T(N,M) = T(N-1,M-1)+O(1), if (X[M-1] = Y[N-1]) = T(N-1,M)+T(N, M-1) +O(1), otherwise

Step4: Solve the recurrence or,atleast,ascertain the order of growth of its solution. There are several ways to analyse the recurrence relation but we are discussing here two popular approaches of solving recurrences:

- **Method1**:RecursionTreeMethod
- **Method2**:MasterTheorem

Method1: Recursion Tree Method

A recurrence tree is a tree where each node represents the cost of a certain recursive sub problem. We take the sum of each value of nodes to find the total complexity of the algorithm.

Steps for solving a recurrence relation

1. Draw a recursion tree based on the given recurrence relation.
2. Determine the number of levels, cost at each level and cost of the last level.
3. Add the cost of all levels and simplify the expression.

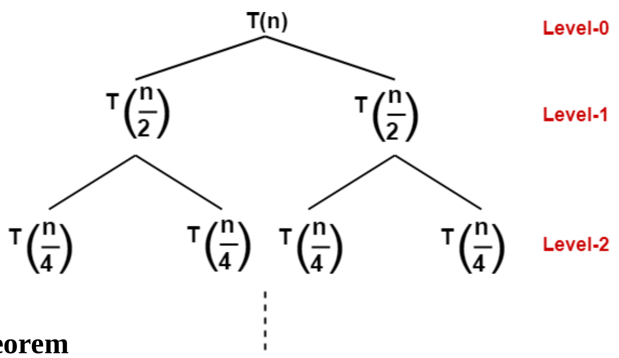
Let us solve the given recurrence relation by Recurrence Tree Method

$$T(N) = 2*T(N/2) + CN$$

From the above recurrence relation,we can find that

1. The problem of size N is divided into two sub-problems of size N/2.
2. The cost of dividing a sub-problem and then combining its solution of size N is CN.
3. Each time,the problem will be divided into half, until the size of the problem becomes 1.

The recursion tree for the above relation will be



Method2:Master theorem

Master theorem states that for a recurrence relation of form

$T(N) = aT(N/b) + f(N)$ where $a >= 1$ and $b > 1$

If**f**(N)=O(N^k)andk≥0, then

Case 1: $T(N) = O(N^{\log_b(a)})$, **if** $k < \log_b(a)$.

Case 2: $T(N) = O((N^k)*\log N)$, **if** $k = \log_b(a)$.

Case 3: $T(N) = O(N^k)$,   **if** $k > \log_b(a)$

Example 1

$T(N)=T(N/2)+C$

The above recurrence relation is of binary search.Comparing this with master theorem,we get a = 1,b = 2 and k = 0 because f(N) = C= C(N^0)

Here logb(a) = k, so we can apply case 2 of the master theorem.

$T(n) = (N^0*\log(N)) = O(\log N)$.

Example2

$T(N)=2*T(N/2)+CN$

The above recurrence relation is of **mergesort**.Comparing this with master theorem,a=2,b =2 and f(N)=CN.Comparing left and right sides off(N),we get k=1.

$\log_b(a)=\log_2(2)=1=K$

So,we can apply the case 2 of the master theorem.

$=>T(N)=O(N^1*\log(N))=O(N\log N)$.

PART-A(2Marks)

1. What is performance measurement?

Ans. Performance measurement is concerned with obtaining the space and the time requirements of a particular algorithm.

2. What is an algorithm?

Ans. An algorithm is a finite set of instructions that,if followed,accomplishes a particular task.

3.**What are the characteristics of an algorithm?**

Ans. 1) Input

2) Output

3) Definiteness

4) Finiteness

5) Effectiveness

4. What is recursive algorithm?

Ans. An algorithm is said to be recursive if the same algorithm is invoked in the body. An algorithm that calls itself is direct recursive. Algorithm A is said to be indeed recursive if it calls another algorithm, which in turn calls A.

5.What is space complexity?

Ans. The space complexity of an algorithm is the amount of memory it needs to run to completion.

6.What is time complexity?

Ans. The time complexity of an algorithm is the amount of computer time it needs to run to completion.

7.Define the asymptotic notation"BigOh"(O),"Omega"(Ω)and"theta"(ɵ)

Ans. Big Oh(O) : The function f(n)= O(g(n))iff there exist positive constants C and no such that $f(n) \leq C * g(n)$ for all n, $n \geq n0$.

Omega(Ω) : The function f(n)=Ω(g(n))iff there exist positive constant C and no such that $f(n) \geq C * g(n)$ for all n, $n \geq n0$.

theta(ɵ) :The function f(n) = ɵ (g(n)) iff there exist positive constant C1, C2, and no such that $C1*g(n) \leq f(n) \leq C2* g(n)$for all n, $n \geq n0$.

PART-B(10Marks)

1. What is asymptotic notation? Explain different types of notations with example.
2. **Solve** the following recurrence relation T(n)=7T(n/2)+cn^2
3. **Solve** the following recurrence relation $T(n) = \{2T\left(\frac{n}{2}\right) + 1, \quad and\ T(1) = 2$
4. **Define** the term algorithm and state the criteria the algorithm should satisfy.
5. If $f(n)=5n^2+6n+4$,then **prove** that f(n) is O(n^2).
6. **Use** step count method and analyze the time complexity when two n×n matrices are added.
7. **Describe** the role of space complexity and time complexity of a program?
8. **Discuss** various the asymptotic notations used for best case average case and worst case analysis of algorithms.

PART-II

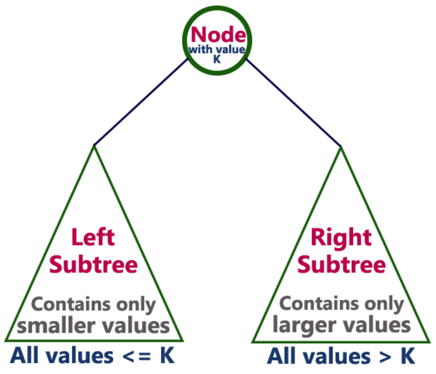
AVLTrees:*Definition and Operations, Applications.*
BTrees:*Definition and Operations.*

AVLTREES

AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree.A binary tree is said to be balanced if, the difference between the heights of left and right sub trees of every node in the tree is either-1, 0 or +1. In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either-1, 0 or +1. In an AVL tree, every node maintains an extra information known as **balance factor**.The AVL tree was introduced in the year1962 by G.M. Adelson-Velsky and E.M. Landis. An AVL tree is defined as follows...

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

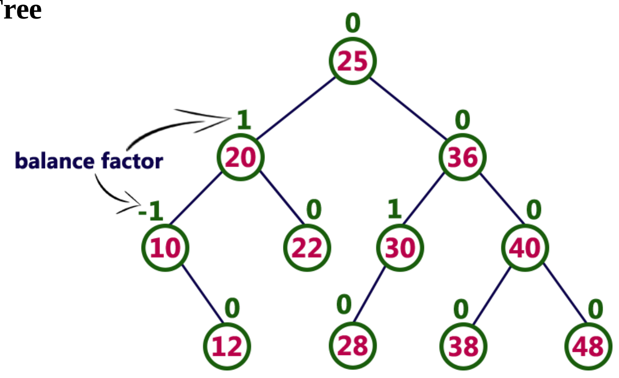
Balance factor of a node is the difference between the heights of the left and right subtrees of that node.The balance factor of a node is calculated either **height of leftsubtree-heightof**



Right sub tree(OR)**height of right sub tree-height of left sub tree**.In the following explanation, we calculate as follows...

Balance factor = height Of Left Sub tree- height Of Right Sub tree

Example of AVL Tree



The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.

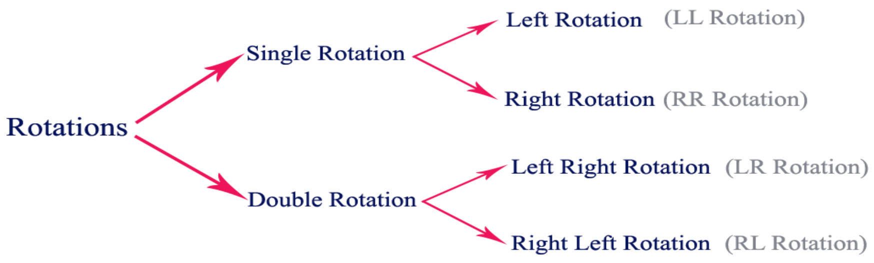
AVL Tree Rotations:

In AVL tree,after performing operations like insertion and deletion we need to check the **balance factor** of every node in the tree.If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation we use **rotation** operations to make the tree balanced.

Rotation operations are used to make the tree balanced.

Rotationistheprocessofmovingnodeseithertoleftortorighttomakethetree balanced.

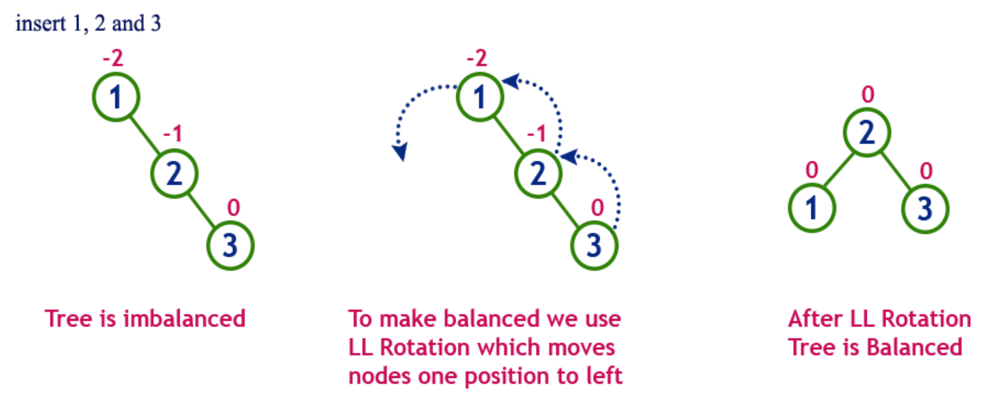
There are **four** rotations and they are classified into **two** types.



Rotations
Single Rotation
Left Rotation (LL Rotation)
Right Rotation (RR Rotation)
Double Rotation
Left Right Rotation (LR Rotation)
Right Left Rotation (RL Rotation)

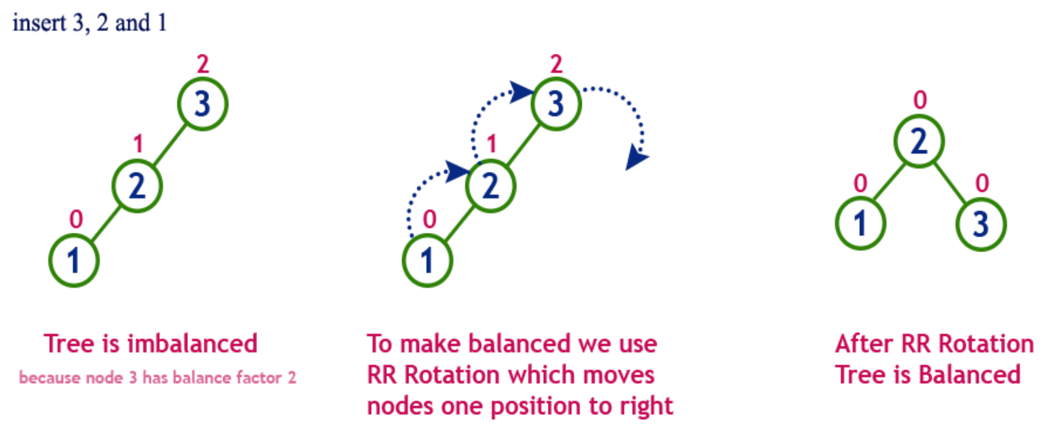
i) Single Left Rotation(LLRotation)

In LL Rotation, every node moves one position to left from the current position.To understand LL Rotation,let us consider the following insertion operation in AVL Tree...



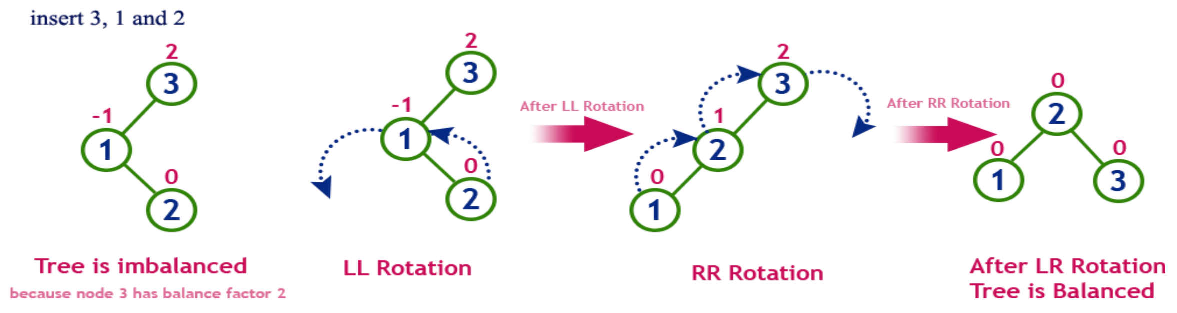
ii) Single Right Rotation(RRRotation)

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation,let us consider the following insertion operation in AVL Tree...



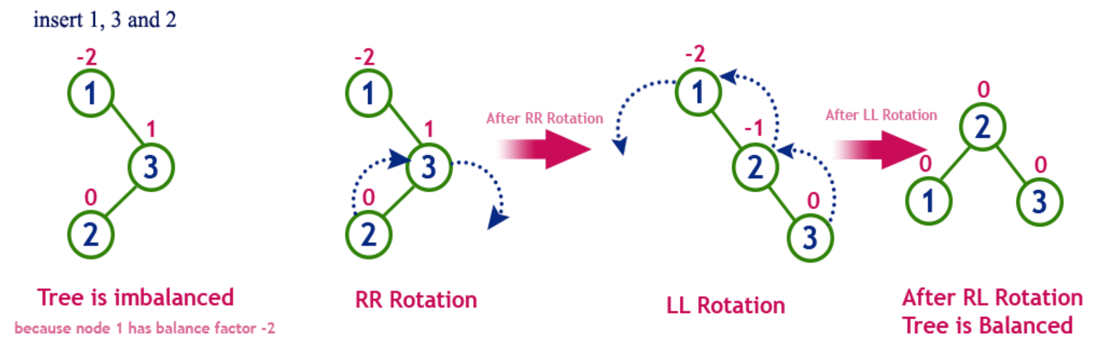
iii) LeftRight Rotation(LR Rotation)

The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...



iv) RightLeft Rotation (RLRotation)

The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...



OperationsonanAVLTree

The following operations are performed on AVLtree...

1. **Search**
2. **Insertion**
3. **Deletion**

i) Search Operation in AVLTree

In an AVL tree, the search operation is performed with **O(log n)** time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree.We use the following steps to search an element in AVL tree...

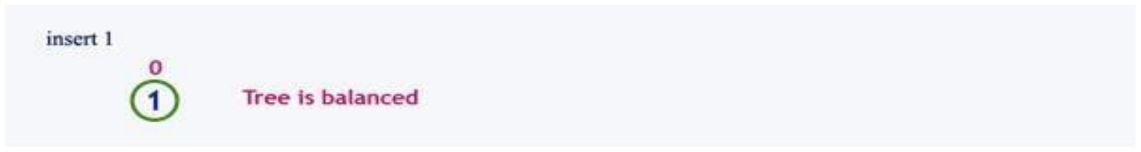
- **Step1-**Read the search element from the user.
- **Step2-**Compare the search element with the value of root node in the tree.
- **Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function
- **Step4-** If both are not matched,then check whether search element is smaller or larger than that node value.
- **Step5-**If search element is smaller,then continue the search process in left subtree.
- **Step6-**If search element is larger,then continue the search process in right subtree.
- **Step 7 -** Repeat the same until we find the exact element or until the search element is compared with the leaf node.
- **Step8-** If we reach to the node having the value equal to the search value,then display "Element is found" and terminate the function.
- **Step9-**If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found"and terminate the function.

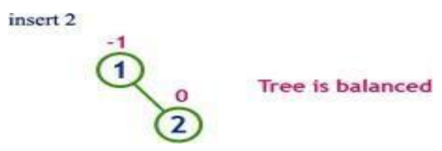
ii) Insertion OperationinAVLTree

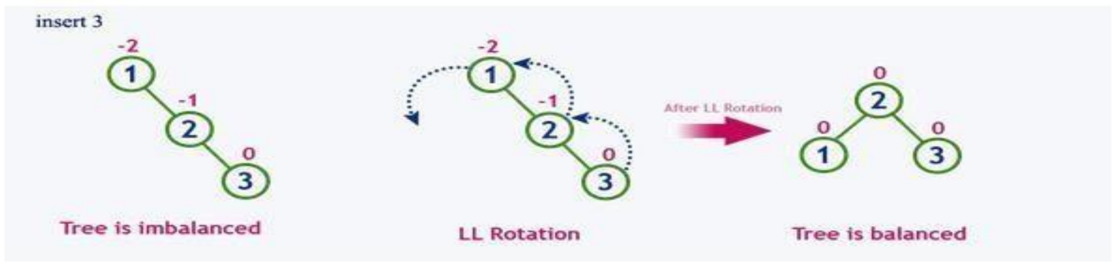
In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

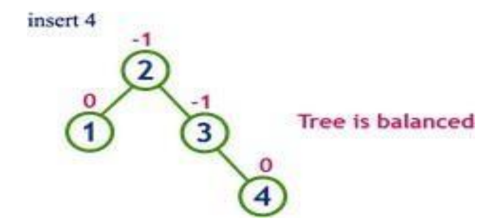
- **Step1-**Insert the new element into the tree using Binary Search Tree insertion logic.
- **Step2-**After insertion,check the **BalanceFactor** of every node.
- **Step 3 -**If the **Balance Factor** of every node is **0 or 1 or -1**then go for next operation.
- **Step 4 -** If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

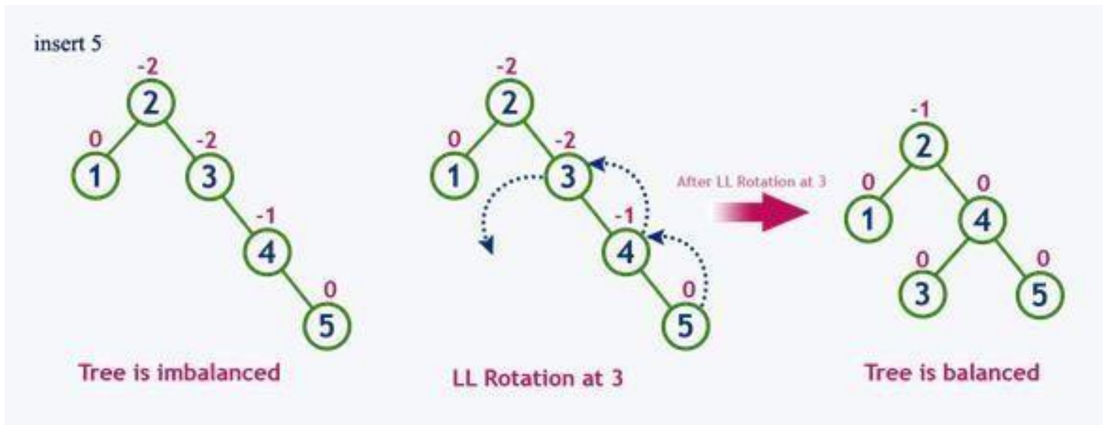
Example:Construct an AVLTree by inserting numbers from 1 to 8.

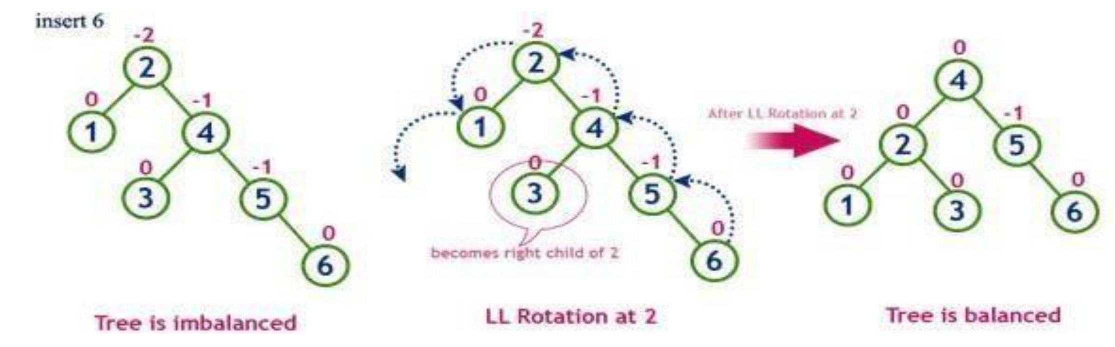


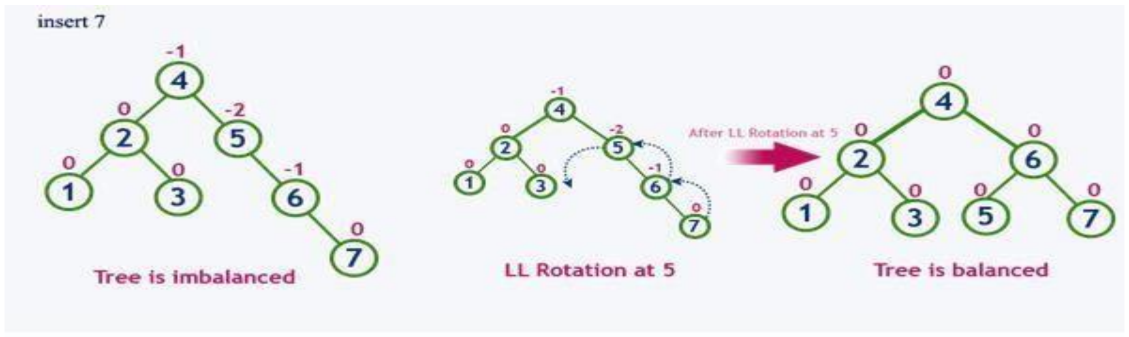


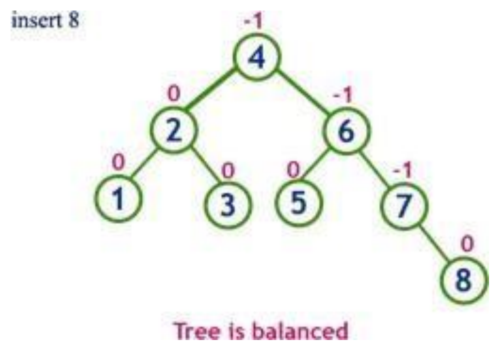

insert 3
Tree is imbalanced
LL Rotation
After LL Rotation
Tree is balanced


insert 4
Tree is balanced


insert 5
Tree is imbalanced
LL Rotation at 3
After LL Rotation at 3
Tree is balanced


insert 6
Tree is imbalanced
becomes right child of 2
LL Rotation at 2
After LL Rotation at 2
Tree is balanced


insert 7
Tree is imbalanced
LL Rotation at 5
After LL Rotation at 5
Tree is balanced



iii) Deletion Operation in AVLTree

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

The two types of rotations are **L rotation** and **R rotation.**Here,we will discuss R rotations. L rotations are the mirror images of them.

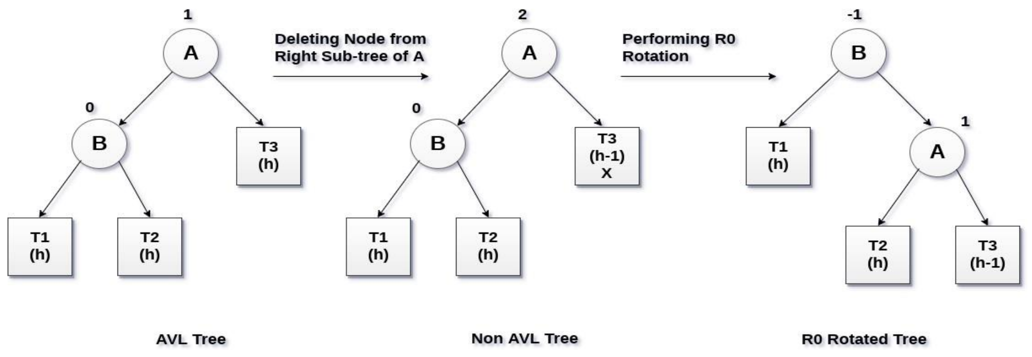
If the node which is to be deleted is present in the left sub-tree of the critical node, then L rotation needs to be applied else if, the node which is to be deleted is present in the right sub-tree of the critical node, the R rotation will be applied.

Let us consider that, A is the critical node and B is the root node of its left sub-tree. If node X,present in the right sub-tree of A,is to be deleted, then there can be three different situations:

a) R0 rotation(Node B has balance factor 0)

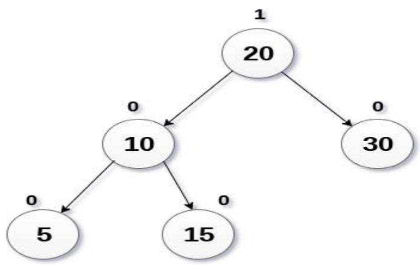
If the node B has 0 balance factor, and the balance factor of node A disturbed upon deleting the node X, then the tree will be rebalanced by rotating tree using R0 rotation.

The critical node A is moved to its right and the node B becomes the root of the tree with T1 as its left sub-tree. The sub-trees T2 and T3 becomes the left and right sub-tree of the node A. the process involved in R 0 rotation is shown in the following image.



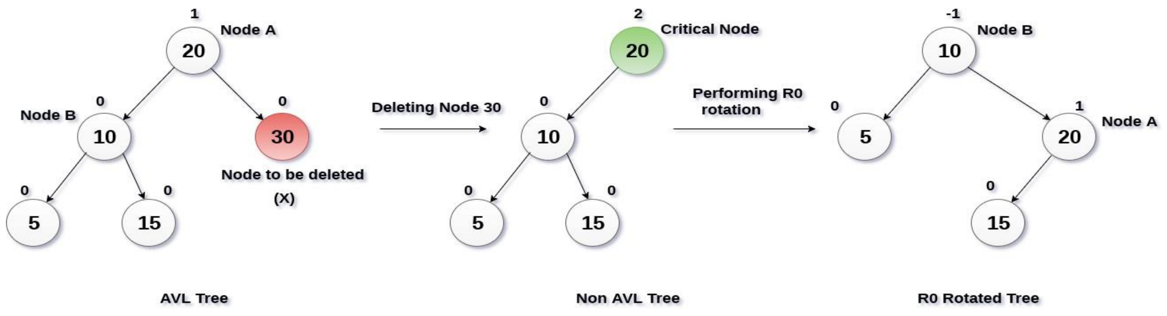
Example:

Delete the node 30 from the AVL tree shown in the following image.



Solution

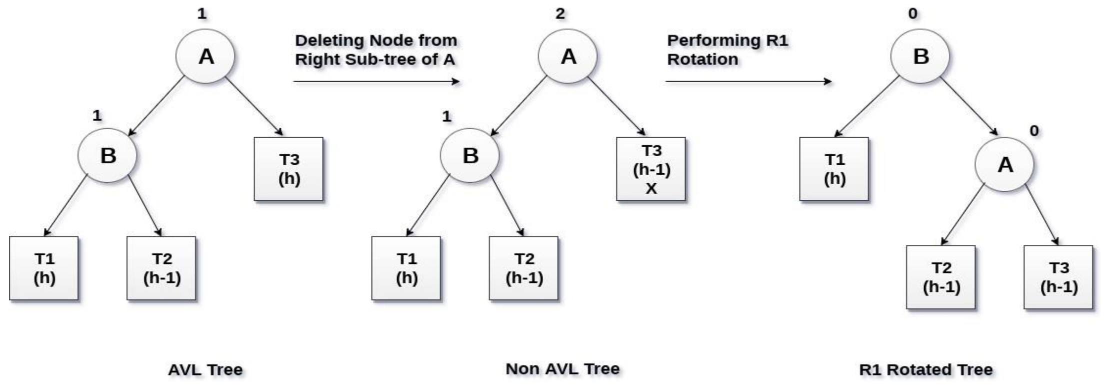
In this case, the node B has balance factor 0, therefore the tree will be rotated by using R0 rotation as shown in the following image. The node B(10) becomes the root, while the node A is moved to its right. The right child of node B will now become the left child of node A.



b) R1 Rotation(Node B has balance factor 1)

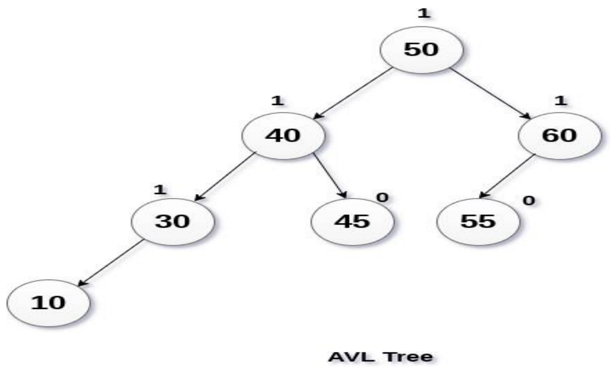
R1 Rotation is to be performed if the balance factor of Node B is 1. In R1 rotation, the critical node A is moved to its right having sub-trees T2 and T3 as its left and right child respectively.T1is to be placed as the leftsub-tree of the node B.

The process involved in R1 rotation is shown in the following image.



Example

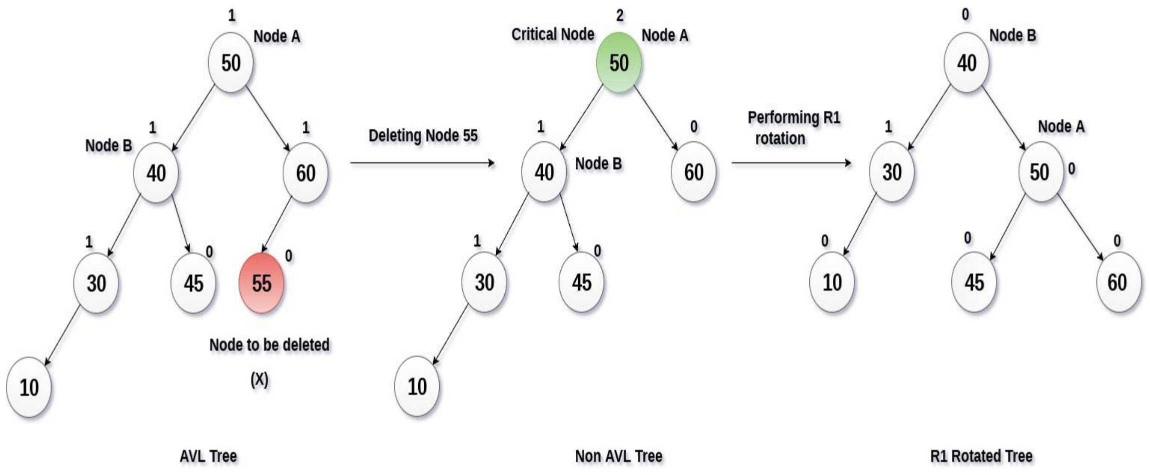
Delete Node 55 from the AVL tree shown in the following image.



Solution:

Deleting 55 from the AVL Tree disturbs the balance factor of the node 50 i.e. node A which becomes the critical node. This is the condition of R1 rotation in which, the node A will be moved to its right(shown in the image below).The right of B is now become the left of A (i.e. 45).

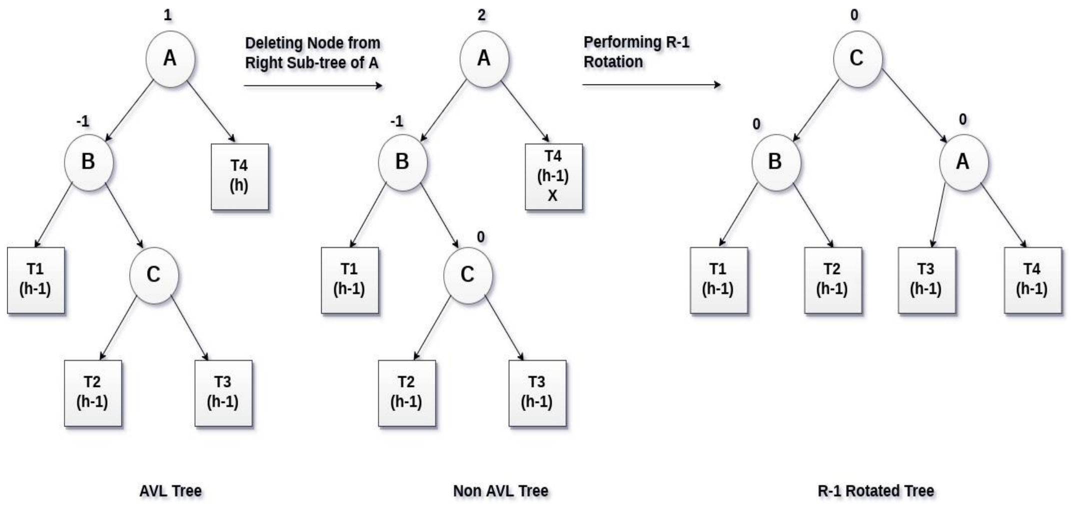
The process involved in the solution is shown in the following image.



c) R-1 Rotation(Node B has balance factor-1)

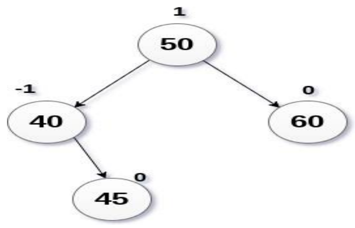
R-1 rotation is to be performed if the node B has balance factor -1. This case is treated in the same way as LR rotation.In this case,the node C,which is the right child of node B, becomes the root node of the tree with B and A as its left and right children respectively.

The sub-trees T1,T2 becomes the left and right sub-trees of B whereas,T3,T4 become the left and right sub-trees of A. The process involved in R-1 rotation is shown in the following image.



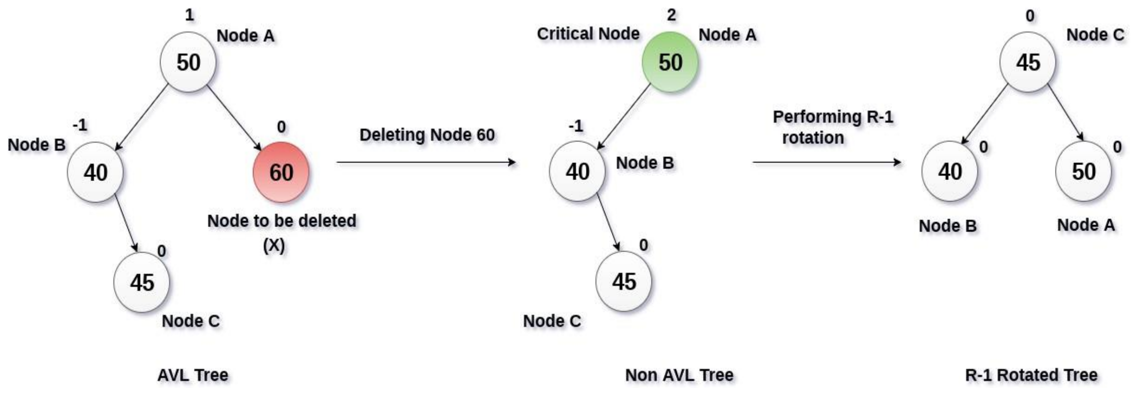
Example

Delete the node 60 from the AVL tree shown in the following image.



Solution:

in this case, node B has balance factor-1. Deleting the node 60, disturbs the balance factor of the node 50 therefore, it needs to be R-1 rotated. The node C i.e.45 becomes the root of the tree with the node B(40)andA(50)as its left and right child.



Applications of AVL Trees

AVL trees are applied in the following situations:

- There are few insertion and deletion operations
- Short search time is needed
- Input data is sorted or nearly sorted

AVL tree structures can be used in situations which require fast searching. But, the large cost of re balancing may limit the usefulness.

BTree

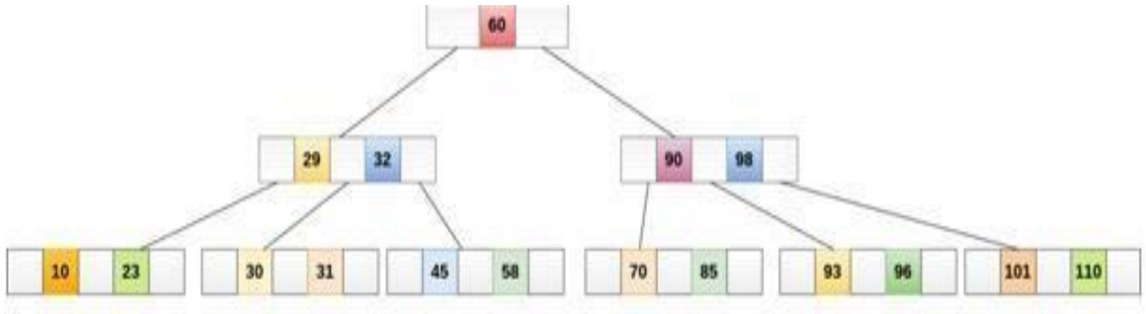
B Tree is a specialized m-way tree that can be widely used for disk access. AB-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain atleast m/2 children.
3. The root nodes must have atleast 2 nodes.
4. All leaf nodes must beat the same level.

It is not necessary that, all the nodes contain the same number of children but,each node must have m/2 number of nodes.

A B tree of order 4 is shown in the following image.



While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

Operations of B Trees

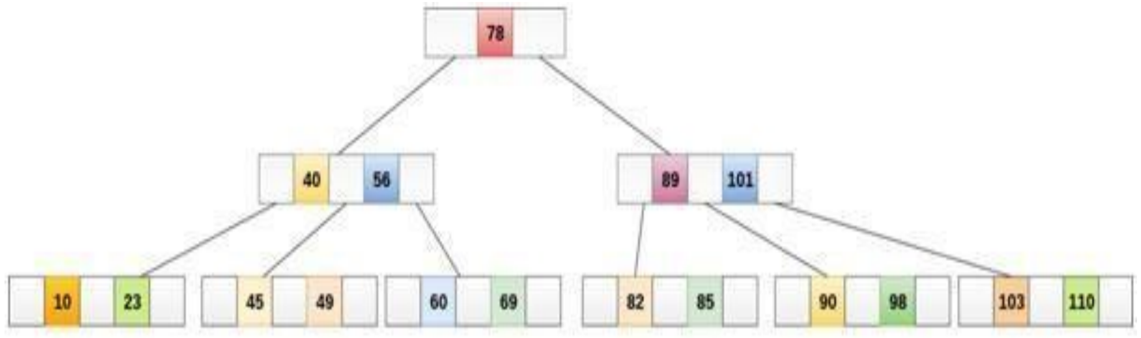
1. Searching
2. Insertion
3. Deletion

i) Searching:

Searching in B Trees is similar to that in Binary search tree. For example,if we search for an item 49 in the following B Tree.The process will something like following:

1. Compare item 49 with root node78. since49<78 hence,move to its leftsub-tree.
2. Since,40<49<56,traverse right sub-treeof40.
3. 49>45,move to right.Compare49.
4. Match found,return.

Searching in a B tree depends upon the height of the tree.The search algorithm m takesO(log n) time to search any element in a B tree.



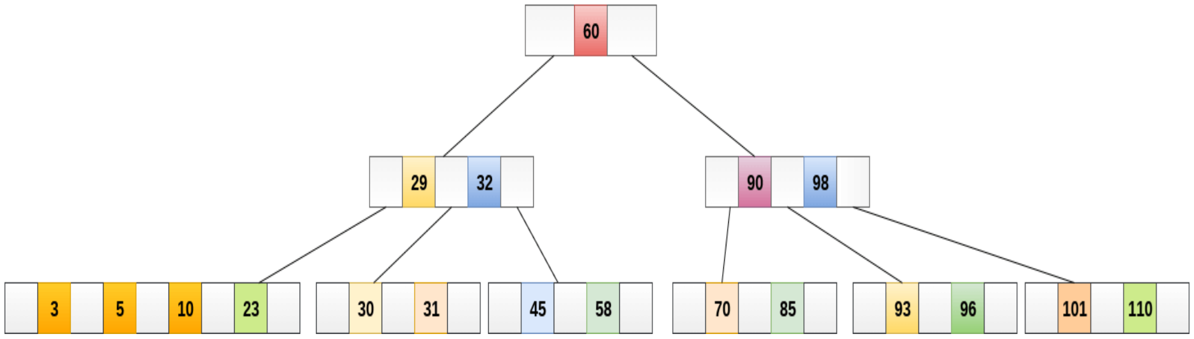
ii) Inserting

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

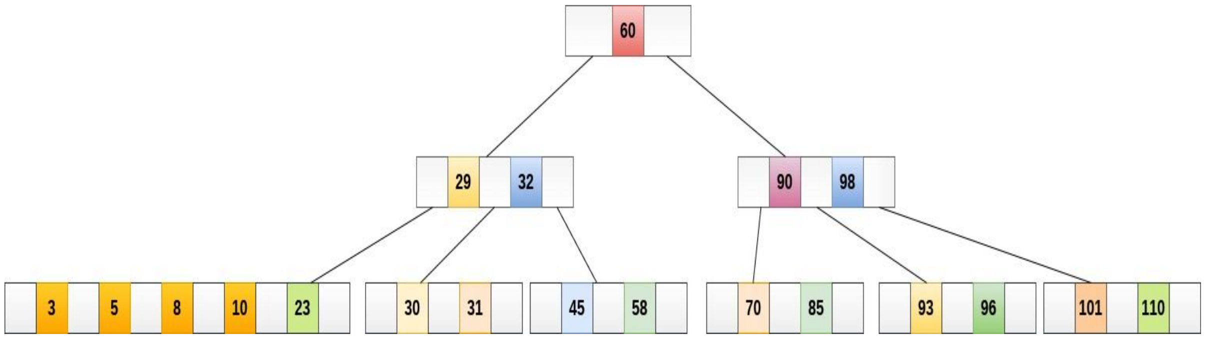
1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else,if the leaf node contains m-1keys,then follow the following steps.
    - Insert the new element in the increasing order of elements.
    - Split the node into the two nodes at the median.
    - Push the median element upto its parent node.
    - If the parent node also contain m-1 number of keys, then split it too by following the same steps.

Example:

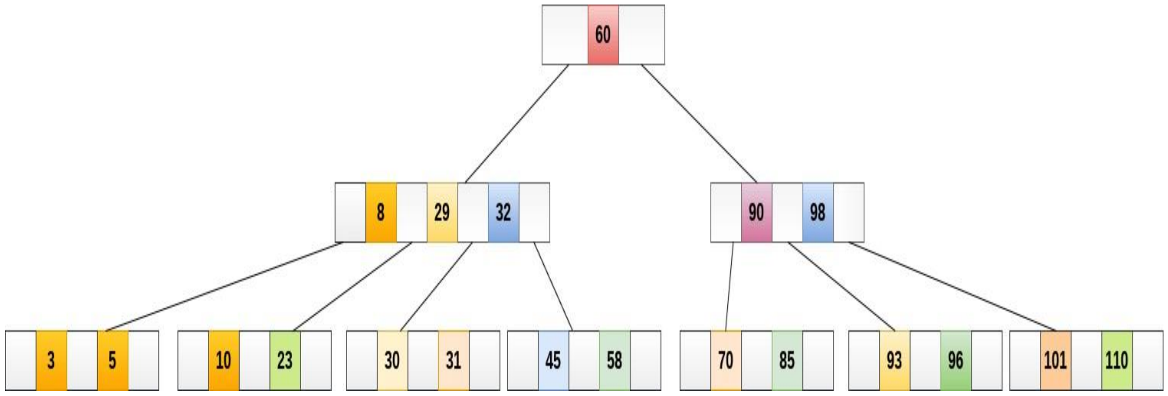
Insert the node 8 into the B Tree of order 5 shown in the following image.



8 will be inserted to the right of 5,therefore insert 8.



The node, now contain 5 keys which is greater than (5 -1 = 4) keys. Therefore split the node from the median i.e.8 and push it up to its parent node shown as follows.



iii) Deletion

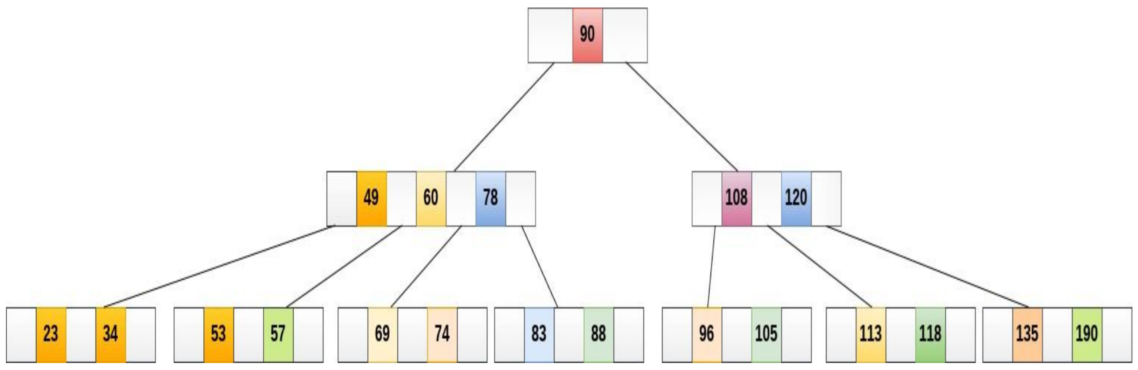
Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.
    - If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
    - If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
4. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parentis left with less than m/2 nodes then, apply the above process on the parent too.

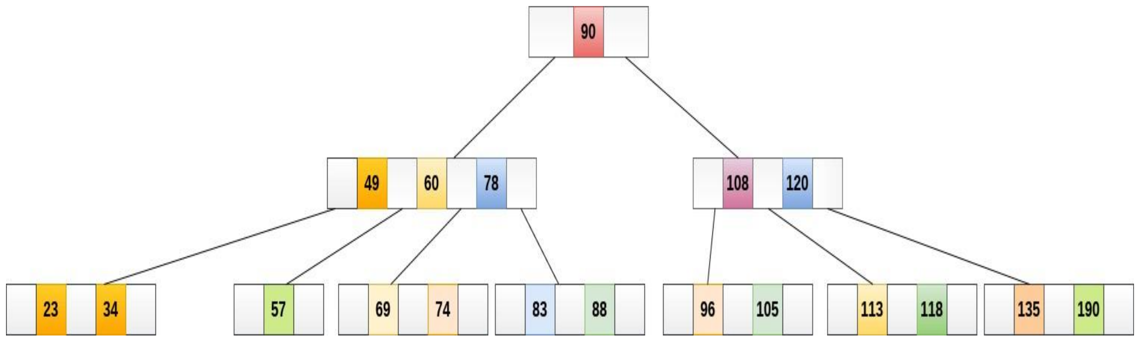
If the the node which is to be deleted is an internal node,then replace the node with its in- order successor or predecessor.Since,successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Example1

Deletethenode53fromtheBTree oforder5showninthefollowingfigure.

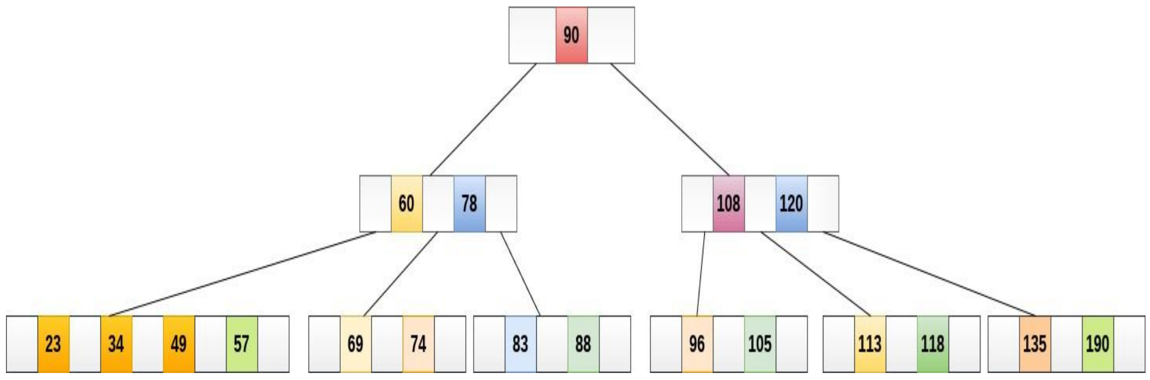


53 is present in the right child of element 49.Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore,merge it with the left sibling and intervening element of parent i.e. 49.

The final B tree is shown as follows.



Application of Btree:

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

Searching an un-indexed and unsorted database containing n key values needs O(n) running time in worst case. However, if we use B Tree to index this database, it will be searched in O(log n) time in worst case.

PART-A(2Marks)

1. **Define AVL Tree.**

   Ans:AVL stands for Adelson-Velskii and Landis.An AVL tree is a binary search tree which has the following properties:

   1. The sub-trees of every node differ in height by at most one.

   2. Every sub-tree is an AVLtree.

   Search time is O(logn).Addition and deletion operations also take O(logn)time.

4. **What do you mean by balanced trees?**

   Ans: Balanced trees have the structure of binary trees and obey binary search tree properties.Apart from these properties,they have some special constraints,which differ from one data structure to another. However, these constraints are aimed only at reducing the height of the tree,because this factor determines the time complexity. Eg: AVL trees, Splay trees.

5. **What are the categories of AVL rotations?**

   Ans: Let A be the nearest ancestor of the newly inserted nod which has the balancing factor±2. Then the rotations can be classified into the following four categories:

   Left-Left: The newly inserted node is in the left sub tree of the left child of A.

   Right-Right: The newly inserted node is in the right sub tree of the right child of A.

   Left-Right: The newly inserted node is in the right sub tree of the left child of A.

   Right-Left: The newly inserted node is in the left sub tree of the right child of A.

6. **What do you mean by balance factor of a node in AVL tree?**

   Ans: The height of left sub tree minus height of right sub tree is called balance factor of a node in AVL tree. The balance factor may be either 0 or +1 or -1.The height of an empty tree is -1.

7. **Whatis‘B’Tree?**

   Ans: A B-tree is a tree data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. Unlike self-balancing binary search trees, it is optimized for systems that read and write large blocks of data. It is most commonly used in database and file systems.

PART-B(10Marks)

1. Explain the AVL tree insertion and deletion with suitable example.
2. Describe the algorithm is used to perform single and double rotation on AVL tree.
3. Create a AVL TREE for the following numbers start from an empty binary search tree.45,26,10,60,70,30,40
4. Delete keys10,60 and 45one after the other and show the trees at eac