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

ALGORITHM

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.

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.

1. **Input** → Zero or more quantities are externally supplied.
2. **Output** → At least one quantity is produced.
3. **Definiteness** → Each instruction is clear and unambiguous.
4. **Finiteness** → If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness** → Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

Examples:

1. Algorithm for *finding the maximum of n given numbers.* In this algorithm (named Max), A & n are procedure parameters. Result & i are Local variables.

```
1   Algorithm Max ( 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 Result := A[i];
7           return Result;
8    }
```

2. Algorithm for *Selection Sort* : - we can define this as

From those elements that are currently unsorted, find the smallest and place it next in the sorted list.

```
1   Algorithm SelectionSort( a, n )
2   // sort the array a[ 1 : n ] into nondecreasing order.
3   {
4           for i:= 1 to n do
5           {
6                   j := i;
7                   for k := i+1 to n do
8                           if ( a[k] < a[j] ) then j := k;
9                   t := a[i] ; a[i] := a[j]; a[j] := t;
10          }
11  }
```

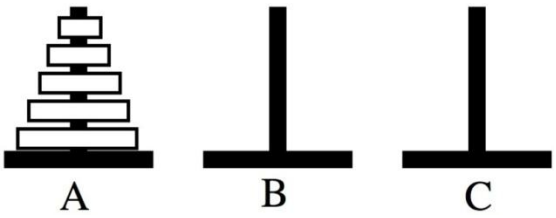
Recursive Algorithms:

- A Recursive function is a function that is defined in terms of itself.
- Similarly, 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 *Indirect Recursive* if it calls another algorithm which in turns calls 'A'.
- The Recursive mechanisms are extremely powerful. They can express complex process very clearly.
- The following 2 examples show how to develop a recursive algorithm.

Example 1 : Towers of Hanoi:

It consists of three towers, and a number of disks arranged in ascending order of size. The objective of the problem is to move the entire stack to another tower, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.



A very elegant solution results from the use of recursion.

- Assume that the number of disks is *'n'*.
- To get the largest disk to the bottom of tower B, we move the remaining *'n-1'* disks to tower C and then move the largest to tower B.
- Now we are left with the tasks of moving the disks from tower C to B.
- To do this, we have tower A and B available.
- The fact, that towers B has a disk on it can be ignored as the disks larger than the disks being moved from tower C and so any disk scan be placed on top of it.

```
1    Algorithm TowersofHanoi(n, x, y, z)
2    //Move the top 'n' disks from tower x to tower y.
3    {
4        if(n>=1) then
5        {
6            TowersofHanoi(n-1, x, z, y);
7            Write("move top disk from tower ", x ,"to top of tower " ,y);
8            Towersofhanoi(n-1, z, y, x);
9        }
10   }
```

This algorithm is invoked by ***TowersOfHanoi(n, A, B, C)***.

PERFORMANCE ANALYSIS

Computing time and storage requirement are the criteria for judging algorithms that have direct relationship to performance.

Performance evaluation can be divided into two major phases

1. a priori estimates (performance analysis) and
2. a posteriori testing (performance measurement).

Space Complexity:

Space Complexity of an algorithm is the amount of memory it needs to run to completion.

The space needed by any algorithm is the sum of the following components.

1. A **fixed part**, that is independent of the characteristics of inputs and outputs. This part includes space for instructions(code), space for simple variables, & fixed size component variables, space for constants etc.
2. A **variable part**, which consists of space needed by component variables whose size, is dependent on the particular problem instance being solved, space for reference variables and recursion stack space etc.

The Space requirement *S(P)* of an algorithm *P* may be written as

$$S(P) = c + S_P(\text{instance characteristics})$$ where *'c'* is a constant.

When analyzing the space complexity of an algorithm first we estimate $S_P(\text{instance characteristics})$. For any given problem, we need to determine which instance characteristics to use to measure the space requirements.

Example 1:

```
1  Algorithm abc(a, b, c)
2  {
3     return a+b+b*c + (a+b –c) / ( a+b) + 4.0;
4  }
```

It is characterized by values of *a, b, c*. If we assure that one word is needed to store the values of each *a, b, c , result* and also we see $S_P(\text{instance characteristics})=0$ as space needed by *abc* is independent of instance characteristics; So 4 words of space is needed by this algorithm.

Example 2:

```
1  Algorithm sum(a, n)
2  {
3     s := 0.0;
4     for i := 1 to n do
5           s := s+a[i];
6     return s;
7  }
```

This algorithm is characterized by *'n'* (number of elements to be summed). The space required for *'n'* is 1 word. The array *a[]* of float values require atleast *'n'* words.

So, we obtain

$$S_{sum}(n) \geq n+3$$

(*n* words for *a*, 1 word for each of *n, i, s*)

Example 3:

```
1  Algorithm Rsum(a, n)
2  {
3     if ( n ≤ 0 ) then  return 0.0;
4     else  return Rsum(a, n-1) + a[n];
5  }
```

Instances of this algorithm are characterized by *n*. The recursion stack space includes space for formal variables, local variables, return address (1-word). In the above algorithm each call to *Rsum* requires 3 words (for *n*, return address, pointer to *a[]*). since the depth of the recursion is *n+1*, the recursion stack space needed is $\geq 3(n+1)$

Time Complexity :

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

- The time *T(P)* taken by a program *P* is the sum of the *compile time* and *Run time*. Compile time does not depend on instance characteristics and a compiled program will be run several times without recompilation, so we concern with just run time of the program. Run time is denoted by T_P***(instance characteristics)***.

- The time complexity of an algorithm is given by the number of steps taken by the algorithm to compute the function. The number of steps is computed as a function of some subset of number of inputs and outputs and magnitudes of inputs and outputs.

- A program step is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics.

- The number of steps any problem statement is assigned depends on the kind of statement.
  For example, comments → 0 steps.
  Assignment statements → 1 step
  Iterative statements such as for, while & repeat-until→ 1 step for control part of the statement.

- Number of steps needed by a program to solve a particular problem is determined in two ways.

Method 1 : (Step count calculation using a variable)

- In this method, a new global variable count with initial value '0' is introduced into the program.
- Statements to increment count are also added into the program.
- Each time a statement in the original program is executed, count is incremented by the step count of the statement.

Example:

```
1   Algorithm Sum (a, n)
2   {
3      s:=0.0;
4      count :=count + 1;  //count is global; initially 0
5      for i := 1 to n do
6      {
7         count:=count+1;  //for for
8         s:=s+a[i]; count:=count+1; //for assignment
9      }
10     count :=count + 1;  //for last time of for
11    count :=count + 1;  //for the return
12    return s;
13  }
```

The change in the value of *count* by the time this program terminates is the number of steps executed by the algorithm.

The value of count is increment by ***2n*** in the for loop.

At the time of termination the value of count is ***2n+3***.

So invocation of Sum executes a total of **2n+3** steps.

Example 2:

```
1   Algorithm RSum(a, n)
2   {
3       count:=count+1; //for the if conditional
4       if ( n ≤ 0) then
5       {
6           count:=count+1;  //for the return
7           return 0.0;
8       }
9       else
10      {
11          count:=count+1; //for addition, function call, return
12          return RSum(a,n-1)+a[n];
13      }
14  }
```

Let $t_{RSum}(n)$ be the increase in the value of *count* when the algorithm terminates.

When n=0, $t_{RSum}(0) = 2$
When n>0, *count* increases by 2 plus $t_{RSum}(n-1)$

When analyzing a recursive program for its step count, we often obtain a recursive formula.

For example,

$$t_{RSum}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{RSum}(n-1) & \text{if } n > 0 \end{cases}$$

- These Recursive formulas are referred to as *recurrence relations*

To solve it, use repeated substitutions

$$\begin{aligned} t_{RSum}(n) &= 2 + t_{RSum}(n-1) \\ &= 2 + 2 + t_{RSum}(n-2) \\ &= 2(2) + t_{RSum}(n-2) \\ &: \\ &= n(2) + t_{RSum}(0) \\ &= 2n + 2, \qquad n \geq 0 \end{aligned}$$

So the step count of *RSum* is *2n+2*.
This step count is telling *the run time for a program with the change in instance characteristics*.

Method 2 : (Step count calculation by building a table)

In this method, the step count is determined by building a table. We list total number of steps contributed by each statement in the table. The table is built in this order.

1. Determine the number of steps per execution (s/e) of the statement and the total number of times (frequency) each statement is executed.
   (*The s/e of a statement is the amount by which the count changes as a result of the execution of that statement.)*
2. The total contribution of the statement is obtained by combining these two quantities.
3. Step count of the algorithm is obtained by adding the contribution of all statements.

Example 1:

	Statement	s/e	Frequency	Total Steps
1	**Algorithm** sum(*a*, *n*)	*0*	_	*0*
2	{	*0*	_	*0*
3	*s* := 0.0;	*1*	*1*	*1*
4	**for** *i* := 1 **to** *n* **do**	*1*	*n + 1*	*n + 1*
5	*s* := *s*+*a*[*i*];	*1*	*n*	*n*
6	**return** *s*;	*1*	*1*	*1*
7	}	*0*	-	*0*
	Total:			*2n + 3*

Example 2:

	Statement	s/e	Frequency n=0	Frequency n>0	Total Steps n=0	Total Steps n>0
1	**Algorithm** Rsum(*a, n*)	*0*				
2	{	*0*				
3	**if** ($n \leq 0$) **then**	*1*	*1*	*1*	*1*	*1*
4	**return** 0.0;	*1*	*1*	*0*	*1*	*0*
5	**else**	*0*				
6	**return** Rsum(*a, n-1*) + *a[n]*;	*1+x*	*0*	*1*	*0*	*1+x*
7	}	*0*				
	Total:				*2*	*2 + x*

$x = t_{RSum}(n-1)$

Example 3:

	Statement	s/e	Frequency	Total Steps
1	**Algorithm** add(*a, b, c, m, n*)	*0*	_	*0*
2	{	*0*	_	*0*
3	**for** *i* := 1 **to** *m* **do**	*1*	*m+1*	*m+1*
4	**for** *j* := 1 **to** *n* **do**	*1*	*m(n + 1)*	*m(n + 1)*
5	*c[i,j] := a[i,j]+b[i,j];*	*1*	*mn*	*mn*
6	}	*0*	-	*0*
	Total:			*2mn+2m+1*

Exercise :

1. What is an algorithm and what are the criterion that an algorithm must satisfy?
2. Define time complexity and space complexity.
3. What is space complexity? Illustrate with an example for fixed and variable part in space complexity.
4. Explain the method of determining the complexity of procedure by the step count approach. Illustrate with an example.
5. Implement iterative function for sum of array elements and find the time complexity use the increment count method.
6. Using step count method, analyze the time complexity when 2 m X n matrix added.
7. Write an algorithm for linear search and analyze the algorithm for its time complexity.
8. Give the algorithm for matrix multiplication and find the time complexity of the algorithm using step-count method.
9. Write a recursive algorithm to find the sum of first n integers and Derive its time complexity.
10. Implement an algorithm to generate Fibonacci number sequence and determine the time complexity of the algorithm using the frequency method.
11. What is the time complexity of the following function.

```
int fun() {
    for ( int i = 1; i<=n; i++ )
    {
        for ( int j = 1; j < n; j += i)
        {
            sum = sum + i * j;
        }
    }
    return(sum);
}
```

12. Give the algorithm for transpose of a matrix m X n and determine the time complexity of the algorithm by frequency-count method.

Asymptotic notations:

- The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared.
- Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.

The following asymptotic notations are mostly used to represent time complexity of algorithms.

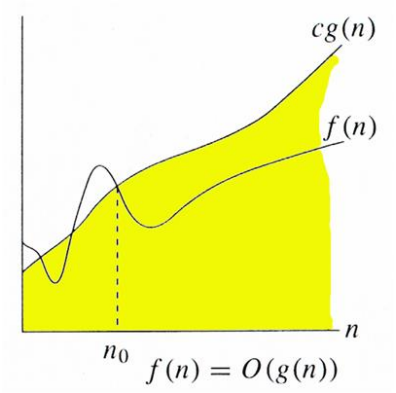
[Big-Oh] O-notation:

- The Big O notation defines an upper bound of an algorithm; it bounds a function only from above.
- Big-Oh notation is used widely to characterize running time and space bounds in terms of some parameter *n*, which varies from problem to problem.
- Constant factors and lower order terms are not included in the big-Oh notation.
- For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is $O(n^2)$. Note that $O(n^2)$ also covers linear time.

Def: The function $f(n)=O(g(n))$ iff there exists positive constants c and n_0 such that

$f(n) \le c*g(n)$ for all n, $n \ge n_0$.

Figure shows that, for all values n at and to the right of n_0, the value of function $f(n)$ is on or below $c.g(n)$.



Example : $7n - 2$ is $O(n)$
Proof: By the big-Oh definition, we need to find a real constant $c>0$ and an integer constant $n_0 \ge 1$ such that $7n - 2 \le cn$ for every integer $n \ge n_0$.
Possible choice is $c=7$ and $n_0=1$.

Example: $20n^3+10n \log n+5$ is $O(n^3)$
Proof: $20n^3+10n \log n+5 \le 35n^3$, for $n \ge 1$
In fact, any polynomial $a_k n^k + a_{k-1}n^{k-1} + \ldots + a_0$ will always be $O(n^k)$.

Here is a list of functions that are commonly encountered when analyzing algorithms. The slower growing functions are listed first. k is some arbitrary constant.

Notation	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Polylogarithmic
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(n^k)$ $(k \ge 1)$	Polynomial
$O(k^n)$ $(k > 1)$	exponential

Instead of always applying the big-Oh definition directly to obtain a big-Oh characterization, we can use the following rules to simplify notation.

Theorem: Let *d(n), e(n), f(n)* and *g(n)* be functions mapping nonnegative integers to nonnegative reals. Then

1. If *d(n)* is *O(f(n))*, then *kd(n)* is *O(f(n))*, for any constant $k > 0$.
2. If *d(n)* is *O(f(n))* and *e(n)* is *O(g(n)),* then *d(n)+e(n)* is *O(f(n)+g(n))*.
3. If *d(n)* is *O(f(n))* and *e(n)* is *O(g(n)),* then *d(n).e(n)* is *O(f(n).g(n))*.
4. If *d(n)* is *O(f(n))* and *f(n)* is *O(g(n)),* then *d(n)* is *O(g(n))*.
5. If *f(n)* is a polynomial of degree *d* (that is, $f(n)=a_0 + a_1n + ...+ a_dn^d$), then *f(n)* is $O(n^d)$.
6. n^x is $O(a^n)$ for any fixed $x>0$ and $a>1$.
7. $\log n^x$ is *O(log n)* for any fixed $x>0$.
8. $\log^x n$ is $O(n^y)$ for any fixed constants $x>0$ and $y>0$.

Example : $2n^3 + 4n^2 \log n$ is $O(n^3)$
Proof:

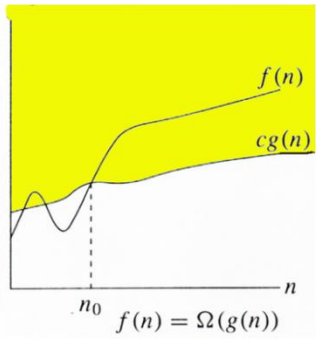
- $\log n$ is $O(n)$
- $4n^2 \log n$ is $O(4n^3)$
- $2n^3 + 4n^2 \log n$ is $O(2n^3+ 4n^3)$
- $2n^3+ 4n^3$ is $O(n^3)$
- $2n^3 + 4n^2 \log n$ is $O(n^3)$

[Omega] Ω-notation:

- Ω-notation provides an asymptotic lower bound.

Def: The function $f(n)=\Omega(g(n))$ iff there exists positive constants *c* and n_0 such that
$f(n) \geq c*g(n)$ for all *n*, $n \geq n_0$

Figure shows the intuition behind Ω-notation for all values *n* at or to the right of n_0, the value of *f(n)* is on or above *c.g(n)*.



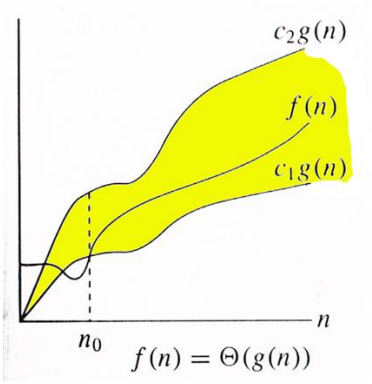
- If the running time of an algorithm is $\Omega(g(n))$, then the meaning is, "the running time on that input is atleast a constant times *g(n)*, for sufficiently large *n*".
- It says that, $\Omega(n)$ gives a lower-bound on the best-case running time of an algorithm.
- Eg: Best case running time of insertion sort is $\Omega(n)$.

Example: 3n+2 is $\Omega(n)$.
Proof: $3n+2 \geq 3n$ for $n \geq 1$ $(c=3, n0=1)$

[Theta] Θ - notation :

Def: The function $f(n) = \Theta(g(n))$ iff there exists positive constants c_1, c_2 and n_0 such that
$c1*g(n) \leq f(n) \leq c2*g(n)$ for all *n*, $n \geq n_0$

Following figure gives an intuitive picture of functions *f(n)* and *g(n)*, where $f(n) = \Theta(g(n))$.



- For all values of n at and to the right of n_0, the value of $f(n)$ lies at or above $c1.g(n)$ and at or below $c2.g(n)$.
- For all $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within constant factors.
- We can say that $g(n)$ is an asymptotically tight bound for $f(n)$.

Example :

$3n + 2 = \Theta(n)$

find c_1, c_2, n_0 such that

$$\left.\begin{aligned} 3n + 2 \geq c_1.g(n) \\ 3n + 2 \leq c_2.g(n) \end{aligned}\right\} \quad \forall n \geq n_0$$

$c_1 = 3, c_2 = 4, n_0 = 2$ satisfies the above.

$\therefore 3n + 2 = \Theta(n)$

[Little-oh] o-notation :

- O-notation provides asymptotic upper bound. It may or may not be asymptotically tight.
- The bound $2n^2 = O(n^2)$ is asymptotically tight, but bound $2n = O(n^2)$ is not.

- o-notation is used to denote an upper bound that is not asymptotically tight.

Def: The function $f(n)=o(g(n))$ iff there exists a constant $n_0>0$ such that $0 \leq f(n) \leq c.g(n)$ for all $n>n_0$ and for any positive constant $c>0$.

The function $f(n)=o(g(n))$ iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

Example: The function $3n+2 = o(n^2)$

Since $\lim_{n\to\infty} \frac{3n+2}{n^2} = 0$

[little-omega] ω-notation:

- ω-notation is used to denote a lower bound that is not asymptotically tight.
- By analogy, ω-notation to Ω-*notation* as *o-notation* to *O-notation*.

Def: The function $f(n) = \omega(g(n))$ iff there exists a constant $n_0>0$ such that $0 \leq c.g(n) \leq f(n)$ for all $n \geq n_0$ and for any positive constant $c>0$.

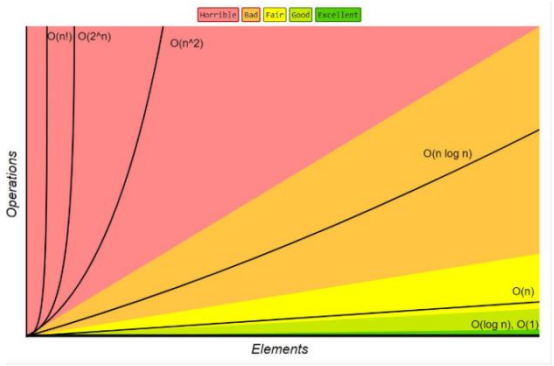
The function $f(n) = \omega(g(n))$ iff $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

Practical complexities

- Time complexity of an algorithm is generally some function of the instance characteristics.
- This function is useful
  - in determining how the time requirements vary as the instance characteristics change.
  - in comparing two algorithm P and Q which perform the same task,
- Assume that algorithm P has complexity $\Theta(n)$ and algorithm Q has complexity $\Theta(n^2)$. we can assert that algorithm P is faster than algorithm Q for sufficiently large n.

Following table shows how various functions grow with n.

n	log n	n	n log n	n2	n3	2n
1	0	1	0	1	1	2
2	1	2	2	4	8	4
4	2	4	8	16	64	16
8	3	8	24	64	512	256
16	4	16	64	256	4096	65536
32	5	32	160	1024	32768	4294967296
64	6	64	384	4096	262144	1.84467E+19
128	7	128	896	16384	2097152	3.40282E+38
256	8	256	2048	65536	16777216	1.15792E+77
512	9	512	4608	262144	134217728	1.3408E+154



It is very clear from the table that the function 2^n grows very rapidly with n.

eg:
Let a computer can execute 1 billion steps per second
And n = 40
If algorithm needs 2^n steps for execution, then the number of steps needed = $1.1 * 10^{12}$ which takes 18.3 minutes.

Important Questions :

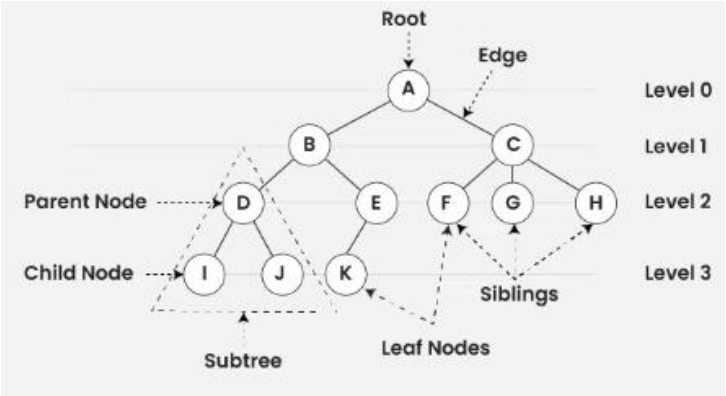
1. What are asymptotic notations? And give its properties.
2. Give the definition and graphical representation of asymptotic notations.
3. Describe and define any three asymptotic notations.
4. Give the Big – O notation definition and briefly discuss with suitable example.
5. Define Little Oh notation with example.
6. Write about big oh notation and also discuss its properties.
7. Define Omega notation
8. Explain Omega and Theta notations.
9. Compare Big-oh notation and Little-oh notation. Illustrate with an example.
10. Differentiate between Bigoh and omega notation with example.
11. Show that the following equalities are incorrect with suitable notations.

$$10\,n^2 + 9 = O(n)$$
$$n^2 \log n = \theta(n^2)$$
$$n^2 / \log n = \theta(n^2)$$
$$n^3 2^n + 5\,n^2 3^n = O(n^3 2^n)$$

12. Describe best case, average case and worst case efficiency of an algorithm.
13. Find big-oh and little-oh notation for $f(n) = 7\,n^3 + 50\,n^2 + 200$
14. What are different mathematical notations used for algorithm analysis

Tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.

The topmost node of the tree is called the root, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.

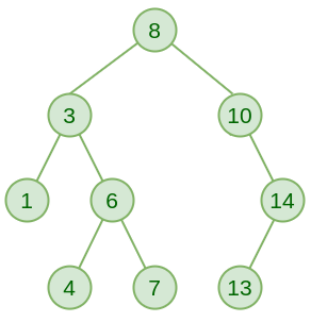


Basic Terminologies in Tree Data Structure:

- **Parent Node**: The node which is an immediate predecessor of a node is called the parent node of that node.
- **Child Node**: The node which is the immediate successor of a node is called the child node of that node.
- **Root Node**: The topmost node of a tree or the node which does not have any parent node is called the root node.
- **Leaf Node** or **External Node**: The nodes which do not have any child nodes are called leaf nodes.
- **Ancestor of a Node**: Any predecessor nodes on the path of the root to that node are called Ancestors of that node.
- **Descendant**: A node x is a descendant of another node y if and only if y is an ancestor of x.
- **Sibling**: Children of the same parent node are called siblings.
- **Level of a node**: The count of edges on the path from the root node to that node. The root node has level 0.
- **Internal node**: A node with at least one child is called Internal Node.
- **Height of a node**: The height of a node can be defined as the length of the longest path from the node to a leaf node of the tree.

Binary Tree is a non-linear and hierarchical data structure where each node has at most two children referred to as the left child and the right child.

Binary Search Tree is a data structure used in computer science for organizing and storing data in a sorted manner. Binary search tree follows all properties of binary tree and for every node, its left subtree contains values less than the node and the right subtree contains values greater than the node. This hierarchical structure allows for efficient Searching, Insertion, and Deletion operations on the data stored in the tree.



Disadvantages of Binary Search Tree (BST):

- Not self-balancing: Unbalanced BSTs can lead to poor performance
- Worst-case time complexity: In the worst case, BSTs can have a linear time complexity for searching and insertion
- Not suitable for large datasets: BSTs can become inefficient for very large datasets

A binary tree of height h is ideally height-balanced if every leaf has depth h or $h - 1$, and every node of $depth < h - 1$ has two children.

Self-Balancing Binary Search Trees *are* **height-balanced** binary search trees that automatically keep the height as small as possible when insertion and deletion operations are performed on the tree.

The height is typically maintained in order of $\boldsymbol{log\ N}$ so that all operations take $\boldsymbol{O(log\ N)}$ time on average.

Examples: The most common examples of self-balancing binary search trees are

- AVL Tree
- Red Black Tree and
- Splay Tree

AVL Trees

AVL trees are ideally height-balanced binary search trees. The name "AVL" comes from the names of the two mathematicians, *Adelson-Velski* and *Landis*, who devised them.

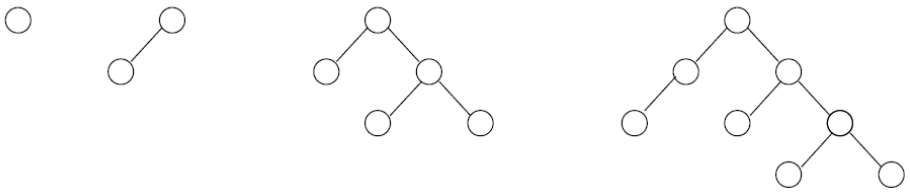
Def*: A binary tree is height-balanced if the heights of left and right subtrees of every node differ by at most one. An AVL tree is a height-balanced binary search tree.*

By convention,

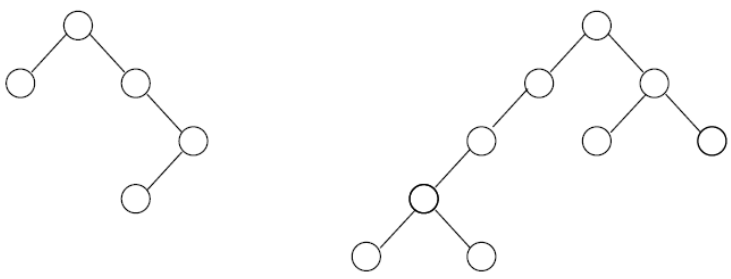
- the height of an empty binary tree is -1;
- the height of a tree consisting of a single node is 0.

In simple words, an AVL tree can be defined as a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.

Examples of AVL trees



Non-Examples of AVL trees



Balance Factor:

The difference between the heights of the left subtree and the right subtree for any node is known as the balance factor of the node.

Balance factor def – Let h_R and h_L be the heights of the right and left subtrees of a node m in a binary tree resepectively. The balance factor of m is defined as $BF(m) = h_R - h_L$.

For an AVL tree, the balance factor of any node is $-1, 0, or + 1$.

Basic operations on AVL Tree include:

- Insertion
- Searching
- Deletion

Insertions, Deletions, Search operations can be done in O(log n) time
The algorithms for insertion and deletion are a bit complex

Key Points of AVL tree:

- It is a binary search tree
- It is height balanced tree
- It is a binary tree in which the height difference between the left subtree and right subtree is atmost one

Characteristics of AVL Tree:

- It follows the general properties of a Binary Search Tree.
- Each subtree of the tree is balanced, i.e., the difference between the height of the left and right subtrees is at most 1.
- The tree balances itself when a new node is inserted. Therefore, the insertion operation is time-consuming

Application of AVL Tree:

- Most in-memory sets and dictionaries are stored using AVL trees.
- Database applications, where insertions and deletions are less common but frequent data lookups are necessary, also frequently employ AVL trees.
- In addition to database applications, it is employed in other applications that call for better searching.

Advantages of AVL Tree:

- AVL trees can self-balance.
- It also provides faster search operations.
- AVL trees also have balancing capabilities with a different type of rotation
- Better searching time complexity than other trees, such as the binary Tree.
- Height must not be greater than log(N), where N is the total number of nodes in the Tree.

Disadvantages of AVL Tree:

- AVL trees are difficult to implement

Maximum & Minimum number of Nodes
Maximum number of nodes = $2^{H+1} - 1$

$$Minimum\ number\ of\ nodes\ of\ height\ H\ = \\ min\ no\ of\ nodes\ of\ height\ (H-1)\ +\ min\ no\ of\ nodes\ of\ height(H-2)\ +\ 1$$

where $H(0) = 1\ \ and\ H(1) = 2$

Insertion Operation

- To insert a key x into an AVL tree T, let us first insert x in T as in ordinary binary search trees.
- When we do an insertion, we need to update all the balancing information for the nodes on the path back to the root.
- The insertion operation is potentially difficult, because inserting a node could violate the AVL tree property.
- If this is the case, then the property has to be restored before the insertion step is considered over.
- The restoration can be done with a simple modification to the tree, known as a ***rotation***.
- After an insertion, only nodes that are on the path from the insertion point to the root might have their balance altered because only those nodes have their subtrees altered.
- As we follow the path up to the root and update the balancing information, we may find a node whose new balance violates the AVL condition.
- Let us call the node that must be rebalanced A. Since any node has at most two children, and a height imbalance requires that A's two subtrees' heights differ by two, it is easy

to see that a violation might occur in four cases:

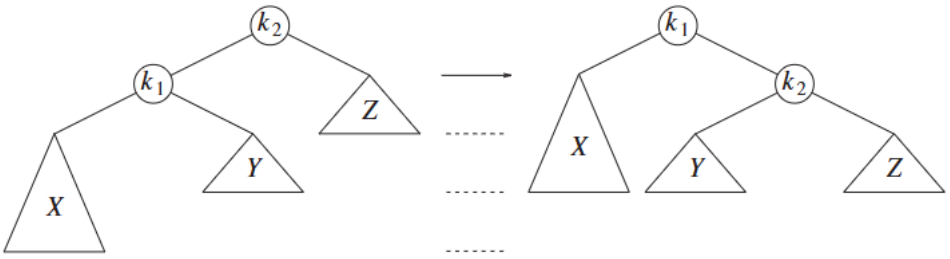
1. An insertion into the left subtree of the left child of A
2. An insertion into the right subtree of the left child of A
3. An insertion into the left subtree of the right child of A
4. An insertion into the right subtree of the right child of A

for cases 1 and 4, the insertion occurs on the outside, and is fixed by a single rotation.
for cases 2 and 3, the insertion occurs inside, and is fixed by double rotation.

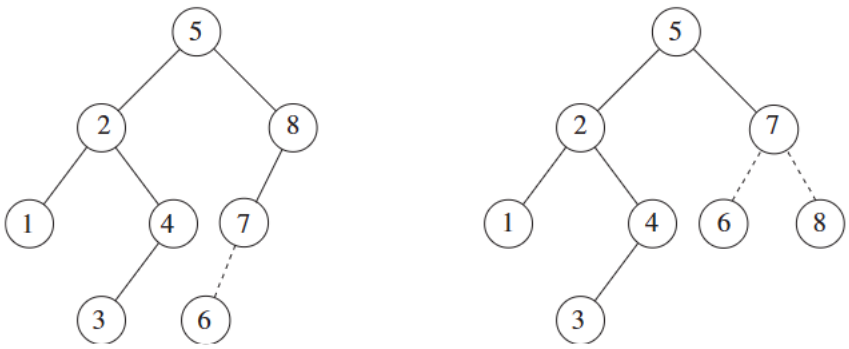
Single Rotations

LL Rotaion

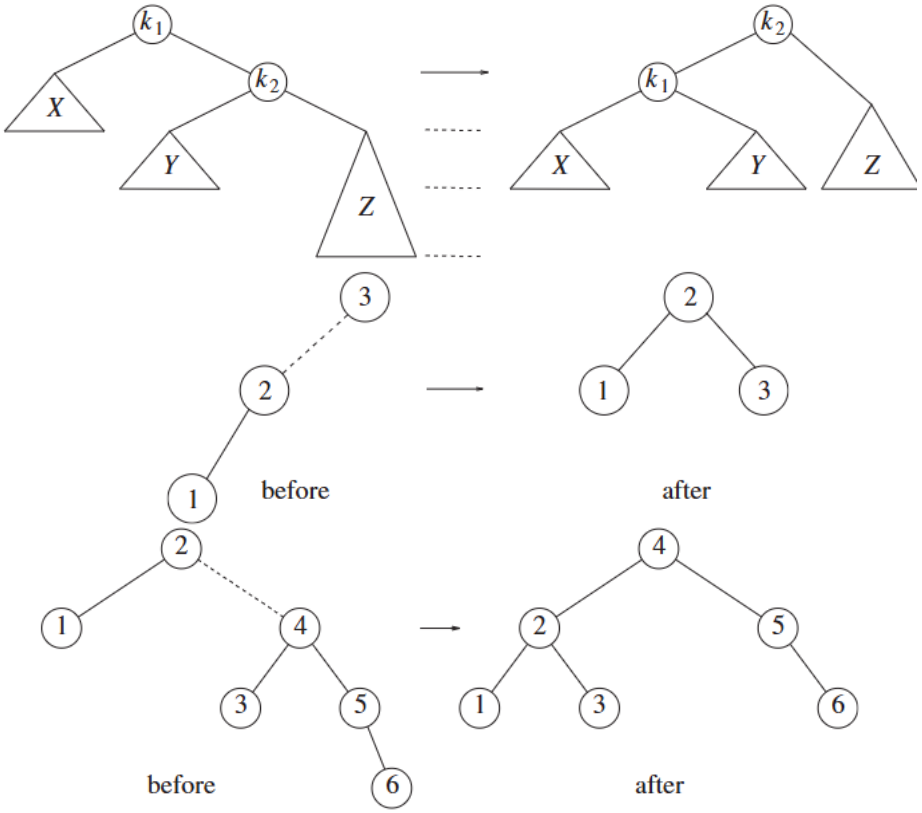
Single rotation to fix Case 1



AVL property destroyed by insertion of 6, then fixed by a single rotation is shown below

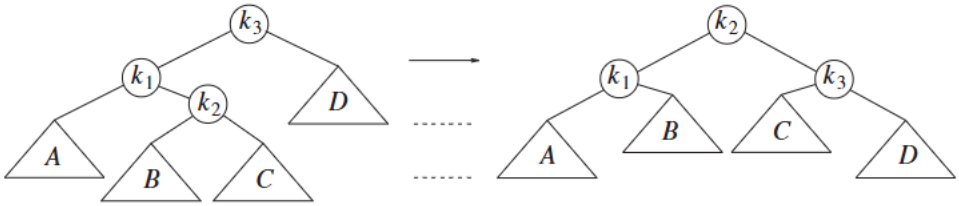


RR Rotation
Single rotation to fix case 4



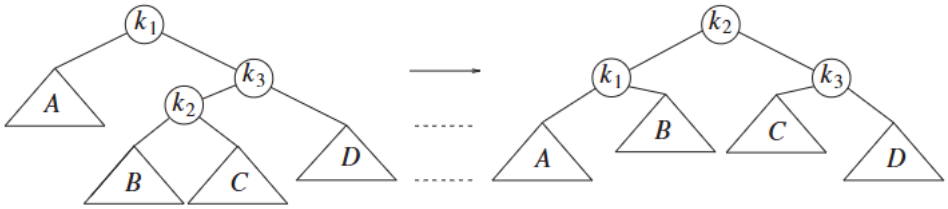
Double Rotations

for the cases 2 or 3, a single rotation does not help. double rotation solves this issue.
Left - Right rotation to fix case 2.



The above diagram suggests, exactly one of tree B or C is two level deeper than D. This rotation places k2 as the new root. This forces k1 to be k2's left child and k3 to be its right child. the resulting locations of the subtrees is based on k1 and k3. The resulting tree satisfies the AVL tree property.

Right - Left rotation to fix case 3



- The programming details are fairly straightforward except that there are several cases.

- To insert a new node with item X into an AVL tree T, we recursively insert X into the appropriate subtree of T (let us call this T_{LR}).
- If the height of T_{LR} does not change, then we are done. Otherwise, if a height imbalance appears in T, we do the appropriate single or double rotation depending on X and the items in T and T_{LR}, update the heights (making the connection from the rest of the tree above), and we are done.

AVL tree node structure

```
class AVLNode {
    int key;
    int height;
    AVLNode left, right;

    AVLNode(int key) {
        this.key = key;
        this.height = 1;
        this.left = this.right = null;
    }
}
```

Insertion Algorithm

The insertion algorithm for an **AVL Tree** follows these main steps:

1. **Standard BST Insertion** – Similar to a Binary Search Tree (BST), we traverse the tree to find the correct position for insertion.
2. **Height Update & Balance Factor Calculation** – Once inserted, we update the height and check the balance factor of each affected node.
3. **Rotations for Rebalancing** – If the tree becomes unbalanced, we perform **one or two rotations** to restore balance.

```
Algorithm insert (root, key){
    if root is null:
        return new AVLNode(key)

    if key < root.key
        root.left = insert(root.left, key)
    else if key > root.key
        root.right = insert(root.right, key)
    else
        return root  // duplicates not allowed

    root.height = 1 + max(height(root.left), height(root.right))

    balance = getBalance(root)

    if balance > 1 and key < root.left.key
        return rightRotate(root)  // left-left case

    if balance < -1 and key > root.right.key
        return leftRotate(root)  // right-right case
```

```
    if balance > 1 and key > root.left.key{
        root.left = leftRotate(root.left)
        return rightRotate(root)  // left-right case
    }

    if balance < -1 and key < root.right.key{
        root.right = rightRotate(root.right)
        return leftRotate(root)  // right-left case
    }
    return root
}
```

```
Algorithm rightRotate(y){
    x = y.left
    T2 = x.right

    x.right = y
    y.left = T2

    y.height = max(height(y.left), height(y.right)) + 1
    x.height = max(height(x.left), height(x.right)) + 1

    return x  // new root
}

Algorithm leftRotate(x){
    y = x.right
    T2 = y.left

    y.left = x
    x.right = T2

    x.height = max(height(x.left), height(x.right)) + 1
    y.height = max(height(y.left), height(y.right)) + 1

    return y  // new root
}
```

Time Complexity Analysis:

1. Finding the Insertion Position (BST Insertion)
   - Since an AVL tree is a balanced binary search tree, its height is $O(log\ n)$.
   - The insertion follows the same logic as BST insertion, which requires traversing $O(log\ n)$ levels in the worst case.
2. Updating Heights & Balance Factors
   - After inserting the new node, we backtrack to the root, updating the height and balance factor at each node. Since the tree has $O(log\ n)$ height, we perform these updates $O(log\ n)$ times.
3. Rotations for Rebalancing

- If the tree is unbalanced, we perform rotations:
  - Single Rotation (Left or Right) – $O(1)$ time.
  - Double Rotation (Left-Right or Right-Left) – Two single rotations, so $O(1) + O(1) = O(1)$.
- Since rebalancing is needed only on the path from the inserted node to the root, at most $O(log\ n)$ rotations occur.

Overall Time Complexity - Each step contributes to the total complexity:
- BST insertion: $O(log\ n)$
- Height & balance updates: $O(log\ n)$
- Rotations: $O(1)$ per node, at most $O(log\ n)$ nodes are affected → $O(log\ n)$

Thus, the worst-case time complexity of AVL tree insertion is: $O(\log n)$

This ensures efficient insertion, unlike an unbalanced BST which can degrade to O(n) in the worst case.

Deletion algorithm

The deletion process in an **AVL Tree** involves three main steps:
1. **Standard BST Deletion** – Locate the node, remove it, and adjust pointers.
2. **Height Update & Balance Factor Calculation** – After deletion, update the height and balance factor of each affected node.
3. **Rotations for Rebalancing** – If the tree becomes unbalanced, perform **one or two rotations** to restore balance.

```
Algorithm Delete(root, key):
    if root is null:
        return root

    if key < root.key {
        root.left = delete(root.left, key)
    else if key > root.key:
        root.right = delete(root.right, key)
    else {
        if root.left is null{
            temp = root.right
            root = null
            return temp
        }
        else if root.right is null{
            temp = root.left
            root = null
            return temp
        }
        temp = minValueNode(root.right)
        root.key = temp.key
        root.right = delete(root.right, temp.key)
    }
    if root is null:
        return root
```

```
    root.height = max(height(root.left), height(root.right)) + 1

    balance = getBalance(root)

    if balance > 1 and getBalance(root.left) >= 0
        return rightRotate(root)  // left-left case

    if balance > 1 and getBalance(root.left) < 0{
        root.left = leftRotate(root.left)
        return rightRotate(root)  // left-right case }

    if balance < -1 and getBalance(root.right) <= 0
        return leftRotate(root)  // right-right case

    if balance < -1 and getBalance(root.right) > 0{
        root.right = rightRotate(root.right)
        return leftRotate(root)  // right-left case
    }
   return root
}

Algorithm minValueNode(node){
    current = node
    while current.left is not null:
        current = current.left
    return current
}
```

Time complexity Analysis

1. Finding and Deleting the Node (BST Deletion)
   - Searching for the node to delete takes $O(log\ n)$ time (since an AVL tree is balanced).
   - Actual deletion has three cases:
     - Leaf Node (No children) – Delete directly, $O(1)$.
     - One Child – Replace with its child, $O(1)$.
     - Two Children – Replace with the inorder successor (smallest in the right subtree) or inorder predecessor (largest in the left subtree), which requires one additional $O(log\ n)$ traversal.
   - Worst-case complexity for this step: $O(log\ n)$

2. Updating Heights & Balance Factors
   - After deletion, we traverse back up to the root, updating the height and computing balance factors.
   - Since the tree height is $O(log\ n)$, these updates occur at most $O(log\ n)$ times.
   - Worst-case complexity for this step: $O(log\ n)$

3. Rotations for Rebalancing
   - After deletion, the AVL tree may become unbalanced at multiple levels.
   - To restore balance, we perform:
     - Single Rotation (Left or Right) – $O(1)$.
     - Double Rotation (Left-Right or Right-Left) – $O(1) + O(1) = O(1)$.

- Since imbalance may propagate up to the root, we may perform $O(log\ n)$ rotations in the worst case.
- Worst-case complexity for this step: $O(log\ n)$

Overall Time Complexity

Each step contributes to the total complexity:

- BST deletion: $O(log\ n)$
- Height & balance updates: $O(log\ n)$
- Rotations: $O(log\ n)$ (at most one per level)

Thus, the worst-case time complexity of AVL tree deletion is: $O(\log n)$

This ensures efficient deletion, unlike an unbalanced BST where deletion could take $O(n)$ in the worst case.

Search algorithm

```
algorithm search(root, key):
    if root is null or root.key == key:
        return root

    if key < root.key:
        return search(root.left, key)

    return search(root.right, key)
```

time complexity analysis of search algorithm

- Since an AVL tree is balanced, its height is $O(log\ n)$.
- In the worst case, we traverse from the root to the deepest leaf, which takes at most $O(log\ n)$ steps.
- Worst-case complexity for this step: $O(log\ n)$

B Trees

Motivation for B-Trees

- Index structures for large datasets cannot be stored in main memory.
- Storing it on disk requires different approach to efficiency
- Assume that we use an AVL tree to store about 20 million records. It is a very deep binary tress with lot of disk accesses (nearly 24)
- Solution is to use more branches and thus reduce the height of the tree!
- As branching increases, depth decreases

Definition of B-Tree

A B-tree of order m is an m-way tree (i.e., a tree where each node may have up to m children) in which:

1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
2. all leaves are on the same level
3. all non-leaf nodes except the root have at least $\left\lceil \frac{m}{2} \right\rceil$ children
4. the root is either a leaf node, or it has from two to m children
5. a leaf node contains no more than m – 1 keys

The number m should always be odd.

Application of B-Tree:

B-trees are commonly used in applications where large amounts of data need to be stored and retrieved efficiently. Some of the specific applications of B-trees include:

- Databases: B-trees are widely used in databases to store indexes that allow for efficient searching and retrieval of data.
- File systems: B-trees are used in file systems to organize and store files efficiently.
- Operating systems: B-trees are used in operating systems to manage memory efficiently.
- Network routers: B-trees are used in network routers to efficiently route packets through the network.
- DNS servers: B-trees are used in Domain Name System (DNS) servers to store and retrieve information about domain names.
- Compiler symbol tables: B-trees are used in compilers to store symbol tables that allow for efficient compilation of code.