**DATA STRUCTURE & JAVA**

**1)Define Scanner S=new Scanner (system.in) briefly?**

In Java, we take input from user with the help of **Scanner** class. Java has a number of predefined classes which we can use.

Predefined classes are organized in the form of packages. This **Scanner** class is found in **java.util** package. So to use Scanner class, we first need to include java.util package in our program.

**import java.util.Scanner; // This will import just the Scanner class  
import java.util.\*; // This will import the entire java.util package**

After importing, we need to write the following statement in our program.

**Scanner s = new Scanner (System.in);**

Here, by writing **Scanner s**, we are declaring **s** as an object of **Scanner** class. **System.in**within the round brackets tells Java this will be System Input i.e. input will be given to the system.

Taking a value from user is quite easy. Consider the following code.

**int n;  
n = s.nextInt(); *//*s is object of Scanner class**

Here,statement **n = s.nextInt();** is used to**input value of an integer variable 'n' from user.** Here, **nextInt()** is a method of the object **s** of Scanner class.

Similarly, we can input values of other data types also. Same as nextInt() is used to input an integer value, methods to input values of other data types are listed below.

**SUM RULE**

The sum and product rules of probability refer to methods of figuring out the probability of two events, given the probabilities of each event. The sum rule is for finding the probability of either of two events that cannot occur simultaneously. The product rule is for finding the probability of both of two events that are independent.

## Explaining the Sum Rule

Write the sum rule and explain it in words. The sum rule is given by P(A + B) = P(A) + P(B). Explain that A and B are each events that could occur, but cannot occur at the same time.

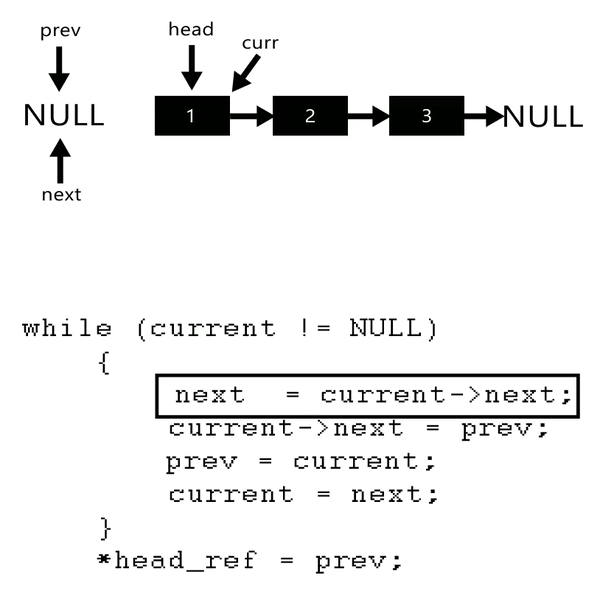
Give examples of events that cannot occur simultaneously and show how the rule works. One example: The probability that the next person walking into class will be a student and the probability that the next person will be a teacher. If the probability of the person being a student is 0.8 and the probability of the person being a teacher is 0.1, then the probability of the person being either a teacher or student is 0.8 + 0.1 = 0.9.

Give examples of events that can occur at the same time, and show how the rule fails. One example: The probability that the next flip of a coin is heads or that the next person walking into the class is a student. If the probability of heads is 0.5 and the probability of the next person being a student is 0.8, then the sum is 0.5 + 0.8 = 1.3; but probabilities must all be between 0 and 1.

## Product Rule

Write the rule and explain the meaning. The product rule is P( E\_F) = P(E)\_P(F) where E and F are events that are independent. Explain that independence means that one event occurring has no effect on the probability of the other event occurring.

**LINK LIST**



**EXPLANATION**

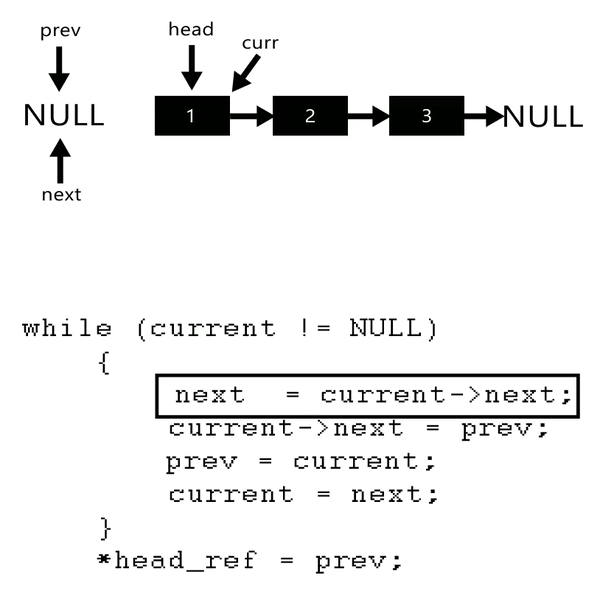
(This is the change and the function for making a link list print In reverse order)

**Step 1**:next will hold address current.next item,i.e. the next node just after current node

**Step 2**:value of current.next will be previous,i.e.the “next” address part of current.next will hold the prev address now.

**Step 3**:prev is now shifted to the value of current,and both pointing same node now.

**Step 4**:current now will store the next node leaving “prev” pointer behind.



Below

**HASHING**

HashMap is a part of java.util.package and it is for implementing hashing in java.  
Suppose I give you and array like - {1,2,1,3,4,5,6,7,8,8,1,1,1,9,9}  
and now I ask you the count of each element of the array, then what would be the naive approach?…to iterate through the array for each element and if that element is found then just increment the count for that element.  
However HashMap is used to establish a {value,key} relationship. i.e for each value you can allot a key. In this case since iterating through all the elements takes O(n\*n) complexity we can use HashMap to reduce the complexity to O(n). How??….the following program shows -

import java.util.\*;

class Hash{

public static void main (String[] args) {

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

HashMap<Integer,Integer> hm=new HashMap<Integer,Integer>();//HashMap instance created

for(int x: a){

if(hm.get(x)==null)hm.put(x,1);

else hm.put(x,hm.get(x)+1);

}

for(int x:a)System.out.println("The count of "+x+" is "+hm.get(x));

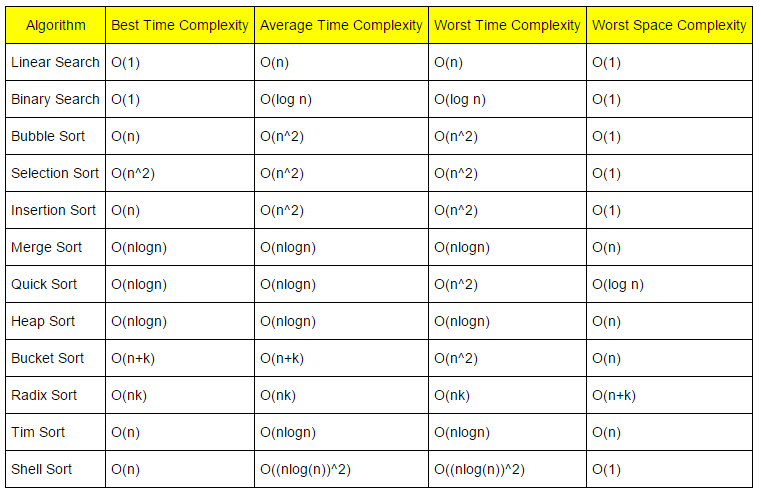
}

}

So what is happening in the above piece of code??…First I am taking an array which can be a user input array too, then I am using a HashMap object as “HashMap<Integer,Integer> hm=new HashMap<Integer,Integer>();”  
Here it means that the first parameter which is the key is going to be the integer and the second parameter is going to be the value which will store the count of the key in this case.  
Thus while iterating the array what i am doing is that first I am checking whether a particular value is present in the array or not using this line -” if(hm.get(x)==null)”, because by default the get() function of HashMap class returns null, and assigning 1 to the value for that key (i.e the particular value in the array for which we get null). This means this was the first encounter of that element and now its count is 1. Now if the get() function doesn’t returns null it means that the particular key has already been encountered so what i am doing is I am changing the value for this key to value+1 i.e I am incrementing the value for a key each time it is encountered (hm.put(x,hm.get(x)+1); the put\*+() function puts a particular value for the key here using the get() function I am getting the previous value or count of that key and putting back hm.get(x)+1 i.e previous count of x + 1 which just made the value as 2 which is now the current count of element x ).

Thus in this way using HashMap we can do this in O(n) time. We can use array for the process too but what if the numberss are in long range. Arrays don’t support indexes of long range. In that case we can use HashMap as HashMap<Long,Integer> hm=new HashMap<Long,Integer>()…and then the same process.

Not only Integer or Long or any predefined wrapper class but HashMap can also make a {key,value} relationship for any object whose class is defined in by the user.

**TIME COMPLEXITY CHART:**

**Small Example application of Queue, Stacks & List**

You've been to a cafeteria, right? and seen a stack of plates? When a clean plate is added to the stack, it is put on top. When a plate is removed, it is removed from the top. So it is called Last-In-First-Out (LIFO). A computer stack is like that, except it holds numbers, and you can make one out of an array or a list, if you like. (If the stack of plates has a spring underneath, they say you "push" one onto the top, and when you remove one you "pop" it off. That's where those words come from.)

In the cafeteria, go in back, where they wash dishes. They have a conveyor-belt where they put plates to be washed in one end, and they come out the other end, in the same order. That's a queue or First-In-First-Out (FIFO). You can also make one of those out of an array or a list if you like.

What are they good for? Well, suppose you have a tree data structure (which is like a real tree made of wood except it is upside down), and you want to write a program to walk completely through it, so as to print out all the leaves.

One way is to do a *depth-first* walk. You start at the trunk and go to the first branch, and then go to the first branch of that branch, and so on, until you get to a leaf, and print it. But how do you back up to get to the next branch? Well, every time you go down a branch, you "push" some information in your stack, and when you back up you "pop" it back out, and that tells you which branch to take next. That's how you keep track of which branch to do next at each point.

Another way is a *breadth-first* walk. Starting from the trunk, you number all the branches off the trunk, and put those numbers in the queue. Then you take a number out the other end, go to that branch, and for every branch coming off of *it*, you again number them (consecutively with the first) and put those in the queue. As you keep doing this you are going to visit first the branches that are 1 branch away from the trunk. Then you are going to visit all the branches that are 2 branches away from the trunk, and so on. Eventually you will get to the leaves and you can print them.

These are two very basic concepts in programming.

1. Use a queue when you want to get things out in the order that you put them in.
2. Use a stack when you want to get things out in the reverse order than you put them in.
3. Use a list when you want to get anything out, regardless of when you put them in (and when you don't want them to automatically be removed).

**Some more terms about stack, queue & List**

Arrays/lists and stacks/queues aren't mutually exclusive concepts. In fact, any stack or queue implementations you find are almost certainly using linked lists under the hood.

Array and list structures provide a description of how the data is stored, along with guarantees of the complexity of fundamental operations on the structures.

Stacks and queues give a high level description of how elements are inserted or removed. A queue is First-In-First-Out, while a stack is First-In-Last-Out.

For example, if you are implementing a message queue, you will use a queue. But the queue itself may store each message in a linked list. "Pushing" a message adds it to the front of the linked list; "popping" a message removes it from the end of the linked list.

**Linked List:** A linked queue/linked stack has flexible, high speed insertions/deletions with a reasonable implementation, but requires more storage than an array. Insertions/deletions are inexpensive at the ends of an array until you run out of space; an array implementation of a queue or stack will require more work to resize, since you'd need to copy the original into a larger structure (or fail with an overflow error).

**Stack:** Stacks are used in cache based applications, like recently opened/used application will comes up. Queues are used in deleting/remove the data, like first inserted old data needs to be deleted at first, to save memory before inserting new data.

# Tree - Terminology

In linear data structure data is organized in sequential order and in non-linear data structure data is organized in random order. A tree is a very popular non-linear data structure used in a wide range of applications. A tree data structure can be defined as follows...

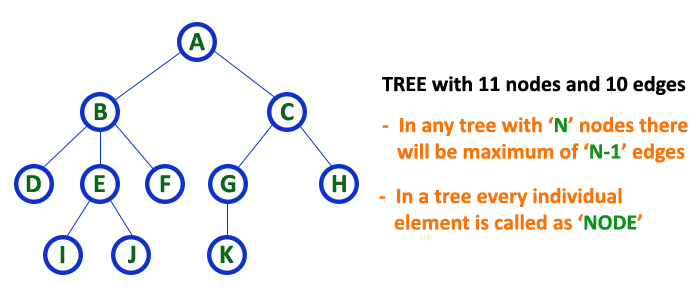
**Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.**

A tree data structure can also be defined as follows...

**Tree data structure is a collection of data (Node) which is organized in hierarchical structure recursively**

In tree data structure, every individual element is called as **Node**. Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure.  
  
In a tree data structure, if we have **N** number of nodes then we can have a maximum of **N-1** number of links.

# Example

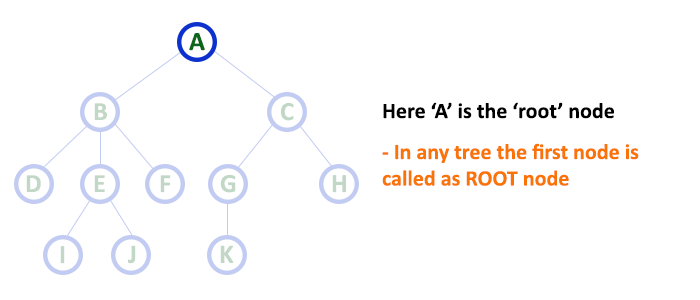


# Terminology

In a tree data structure, we use the following terminology...

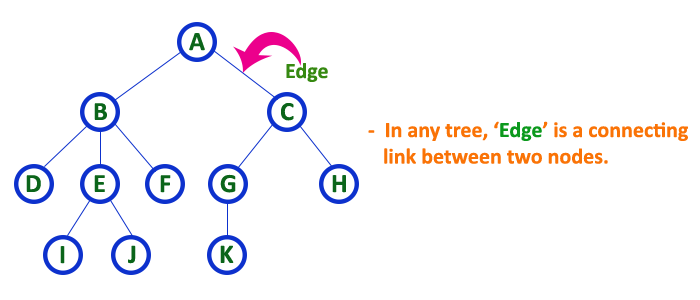
# 1. Root

In a tree data structure, the first node is called as **Root Node**. Every tree must have a root node. We can say that the root node is the origin of the tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.



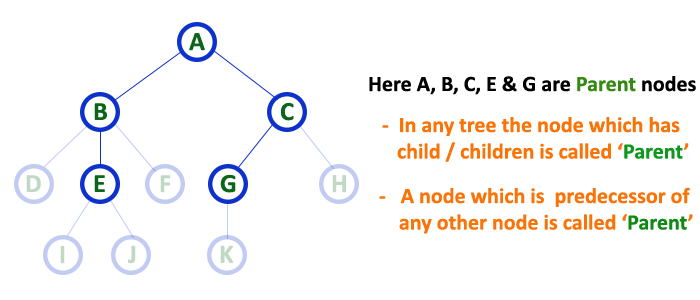
# 2. Edge

In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



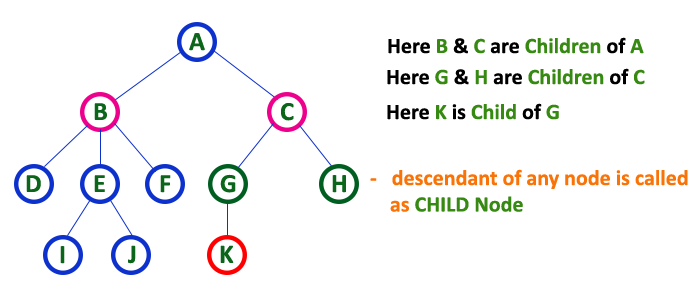
# 3. Parent

In a tree data structure, the node which is a predecessor of any node is called as **PARENT NODE**. In simple words, the node which has a branch from it to any other node is called a parent node. Parent node can also be defined as "**The node which has child / children**".



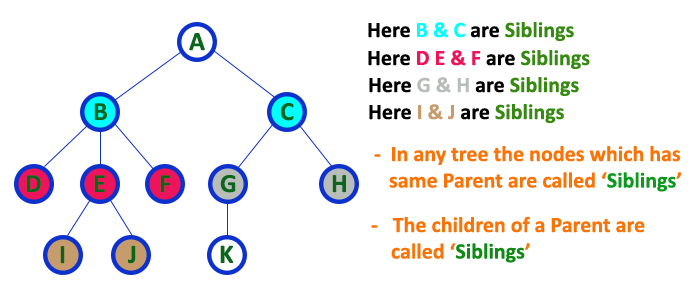
# 4. Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



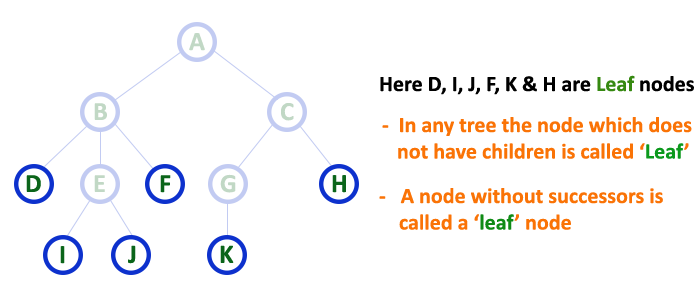
# 5. Siblings

In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with the same parent are called Sibling nodes.



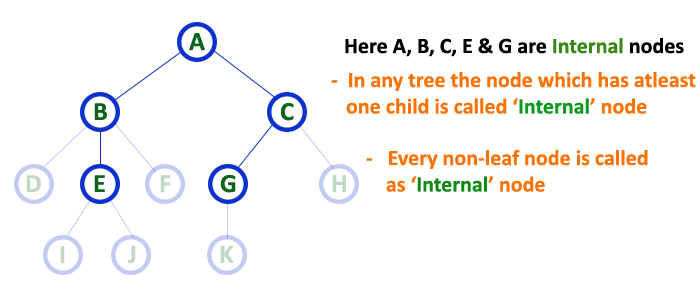
# 6. Leaf

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.  
  
In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.



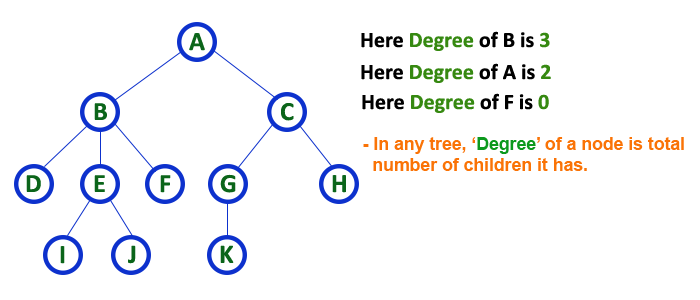
# 7. Internal Nodes

In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.  
  
In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**.**The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as '**Non-Terminal**' nodes.



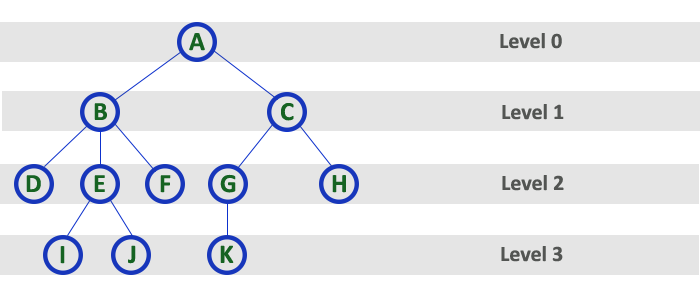
# 8. Degree

In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'



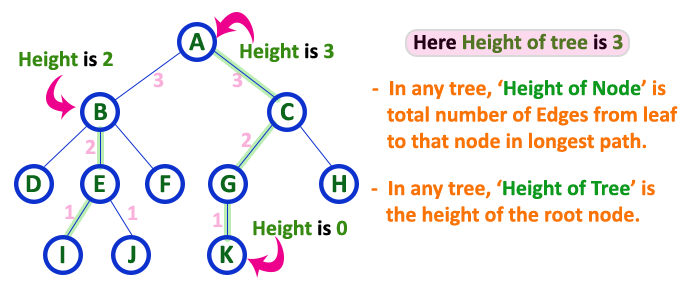
# 9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



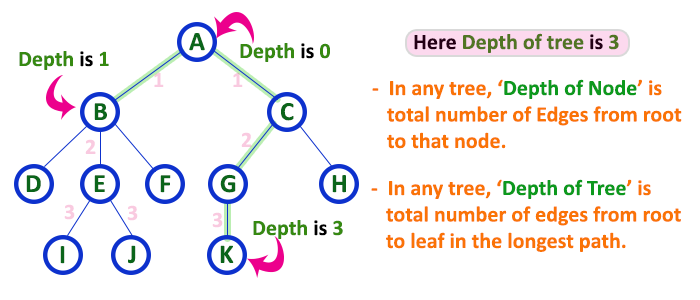
# 10. Height

In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree**. In a tree, **height of all leaf nodes is '0'.**



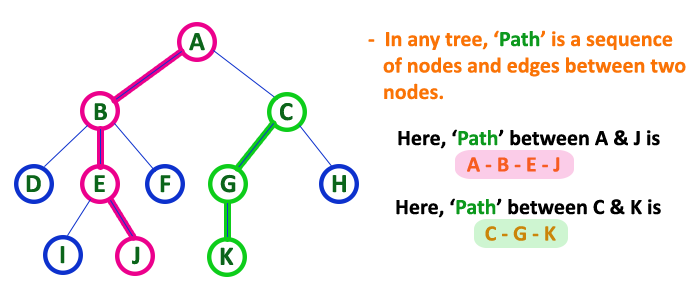
# 11. Depth

In a tree data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**



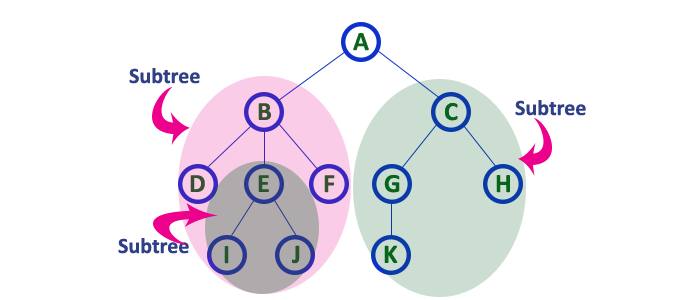
# 12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4**.



# 13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



The first 21 Fibonacci numbers *Fn* are:[[2]](https://en.wikipedia.org/wiki/Fibonacci_number#cite_note-oeis-2)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *F*0 | *F*1 | *F*2 | *F*3 | *F*4 | *F*5 | *F*6 | *F*7 | *F*8 | *F*9 | *F*10 | *F*11 | *F*12 | *F*13 | *F*14 | *F*15 | *F*16 | *F*17 | *F*18 | *F*19 | *F*20 |
| 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233 | 377 | 610 | 987 | 1597 | 2584 | 4181 | 6765 |

The sequence can also be extended to negative index *n* using the re-arranged recurrence relation