Recursion

• One of the most powerful features of modern programming languages is the ability to execute recursive functions.
• A recursive function is a function that calls itself.
• The general strategy for a recursive function is to complete some small part of a large task and to use a recursive call to complete the remainder of the task.

Let’s see how we compute the factorial of a function
(factorial = the product of an integer and all the integers below it)

Factorial of n is n!. It’s just the product of the integers 1 through n.

\[ 5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \text{ or } 120 \]
\[ n! = 1 \times 2 \times 3 \times 4 \times 5 \times \ldots \times n-1 \times n \]
\[ n! = n \times (n-1)! \times \ldots \times 2 \times 1 \]
\[ n! = n \times (n-1)! \]

See what we just did! We say that computing \((n - 1)!\) is a subproblem that we solve to compute \(n!\).
So, \(5! = 5 \times 4!\)
\(4! = 4 \times 3!\)
\(3! = 3 \times 2!\)
\(2! = 2 \times 1!\)
\(1! = 1\) (By the way, \(0! = 1\)) \(\rightarrow\) this is our base case (smallest problem that can be solved)

\[ 2! = 2 \times 1! = 2 \]
\[ 3! = 3 \times 2! = 6 \]
\[ 4! = 4 \times 3! = 24 \]
\[ 5! = 5 \times 4! = 120 \]

**Factorial method**

```java
private static long factorial(int n) {
```
if (n == 1)
    return 1;
else
    return n * factorial(n-1);
}

Let's go back to the Russian dolls. Although they don't figure into any algorithms, you can see that each doll encloses all the smaller dolls (analogous to the recursive case), until the smallest doll that does not enclose any others (like the base case).

The general shape of a recursive method is:

```java
void methodA()
{
    if(<base case>) {
        <do base thing>;
        return;
    }
    <do something>
    methodA();  //do recursion
    <do something>
}
```

The base case is the smallest problem the method is designed to solve. The usual way to handle this situation is to pass a parameter to method A that tells method A whether to call itself by telling it how much work is left to be done. We use this parameter to determine if we have reached that smallest problem.

**Example: Print out the numbers from 1 to 100.**

How can we do it?
One way is to use a for or while loop.
But another way is as follows using recursion. We need to keep track of how much work is left to be done after each call, so we pass two parameters indicating the range of numbers to be printed. When that range is just one number, then we can print it and no recursion needs to be done.

```java
//The following method prints the numbers from start to end
//It assumes start <= end

public void printNumbers(int start, int end)
{
    if(start == end) {
        System.out.println(start);
    } else {
        System.out.println(start);
        printNumbers(start+1, end);
    }
}
```
If the user executes printNumbers(1,5), the numbers 1,2,3,4,5 are printed (one per line).

[Do a recursive trace diagram for this using start = 1 and end = 5].

Note the assumption that start <= end!.

Another example: Adding numbers in an array of integers.

```java
public int LinearSum(int[] A, n)
{
    if(n == 1)
        return A[0];
    else {
        return A[n-1] + LinearSum (A, n-1)
    }
}
```

Recursive Trace of LinearSum (A, n) with A = [4,3,6,2,5] and n=5

Introduction

- A binary tree is made of nodes, where each node contains a "left" reference, a "right" reference, and a data element.
- Trees that have a maximum of 2 children (but may have 0 or 1) are called binary trees.
- The topmost node in the tree is called the root.
- Any node in a binary tree has exactly one parent node (except the root)

Terminologies
• Root – Root is a node which does not have any parent.
• Parent – Node in the tree that is one step higher in hierarchy and lying on the same branch
• Child – As in the diagram above:
  o B & C are the child of A,
  o D & E are the child of B
  o F & G are the child of C.
  o A child is a node which is directly linked with its parent and it is below parent in structure.
• Sibling – Nodes that share same parent. As in the diagram above D & E are siblings
• Leaf – Nodes without children are called leaf node.

More tree terminology
• The depth/level of a node is the number of edges from the root to the node.
• The height of a node is the number of edges from the node to the deepest leaf.
• The height of a tree is a height of the root.

Binary Trees and Recursion
• A tree is a recursive data structure because each child of a node in the tree is a tree in its own.
• Recursive definition: a binary tree is either the null or empty tree or a node that has two children that are binary trees (called subtrees).

Complete Tree

• A complete binary tree is a binary tree such that
  o every level of the tree has the maximum number of nodes possible except possibly the deepest level, and
  o at the deepest level, the nodes are as far left as possible.

```
    o
   / \ 
  o   o
 / \ / \ 
 o   o   o
```

complete tree

• A fully complete binary tree has \( n \) nodes. What is the height of the tree?
  \( O(\lg n) \)

```
   e
  / \  
 m   p
 / \ / \  
 d f n v
```

<table>
<thead>
<tr>
<th>Level</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2^0</td>
</tr>
<tr>
<td>1</td>
<td>2^1</td>
</tr>
<tr>
<td>2</td>
<td>2^2</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>( k )</td>
<td>2^k</td>
</tr>
</tbody>
</table>

• Number of nodes: \( 2^{(k+1)} - 1 \) nodes

\( K \) - depth of tree
Example, at root, number of nodes will be \( 2^{(0+1)} - 1 = 1 \) node

Applications
Trees are mainly used to represent data containing a hierarchical relationship between elements, e.g. records, family trees and table of contents.

Traversals
• A traversal is a process that visits all the nodes in the tree. Since a tree is a nonlinear data structure, there is no unique traversal.
Note: nonlinear data structures do not have their elements in a sequence

Example: Consider this tree

InOrder traversal
• Visit the left subtree first
• Visit the node.
• Visit the right subtree.
InOrder - 9, 5, 1, 7, 2, 12, 8, 4, 3, 11

PostOrder traversal
• Visit the left subtree first
• Visit the right subtree
• Visit the node.
PostOrder - 9, 1, 2, 12, 7, 5, 3, 11, 4, 8

PreOrder traversal
• Visit the node
• Visit the left subtree first
• Visit the right subtree.
PreOrder - 8, 5, 9, 7, 1, 12, 2, 4, 11, 3

In the next picture we demonstrate the order of node visitation. Number 1 denote the first node in a particular traversal and 7 denote the last node.
Binary Tree Implementation

1. Array based Implementation

- Root: [0]
- Data for a non-root node: [i]
- Parent of node in array position i: [(i-1)/2]
- Left child of node in array position i: array position [2i+1]
- Right child of node in array position i: array position [2i+2]

2. Linked/Node-based Implementation

*(Shaffer Book – Section 5.3.1)*
public class IntTree {

.
.
.

}

Build the TreeNode class which stores an integer at each node

private static class TreeNode {
    private int data;
    private TreeNode left;
    private TreeNode right;

    public TreeNode( int d, TreeNode l, TreeNode r ) {
        this.data = d;
        this.left = l;
        this.right = r;
    }
}

User:

TreeNode tree1 = new TreeNode( 5, null, null );
TreeNode tree2 = new TreeNode( 7, null, null );
TreeNode root = new TreeNode( 3, tree1, tree2 );

The key is never to allow a Node to have null children. That is, make sure that every Node
object has exactly two children objects. Then there is always an object to invoke your method
on so the totalSum method for Nodes doesn't need to test for null.

So we create a special kind of Tree node called "NullNode" that has its own version of the
totalSum method. Note that the constructor for NullNode just calls the super's constructor
using d and null and null as arguments, since those values are not important. What's important
is that it has its own implementation of totalSum. Note that it has null for left and right pointer
values, but it doesn't need to test for null in totalSum since its version of totalSum is not recursive.

```java
public class Node {
    int data;
    Node left, right;

    public Node(int d, Node l, Node r) {
        this.data = d; this.left = l; this.right = r;
    }

    public int totalSum() {
        return value + left.totalSum() + right.totalSum();
    }
}

public class NullNode extends Node {
    public NullNode() {
        super(0, null, null);
    }

    public int totalSum() {
        return 0;
    }
}
```

**Binary Search Trees**

A binary search tree is a binary tree in which the key field value of the root node is greater than the key field values of all of the nodes in the root's left subtree, and less than the key field values of all of the nodes in the root's right subtree. In addition, each subtree in the tree is also a binary search tree.
Example

- The value of the key field of the tree's root node, 50, is greater than all the keys in its left subtree (40, 35, 47, and 43), and it is also less than all the keys in its right subtree (63, 55, 70, 68, and 80).
- In addition, all subtrees in the tree are also binary trees, which can be verified by inspecting them.
- For example, consider the subtree whose root node is 63. This subtree is also a binary search tree because all keys in 63's left subtree (55) are less than 63, and all of the keys in the root's right subtree (70, 68, and 80) are greater than 63.

BST combines the flexibility of insertion in linked lists with the efficiency of search in an ordered array.

- **Binary Search Tree Operations**
  - Add
  - Search
  - Delete