# CS 1027 Full Review

*Recursive Solution (15) + Trace Recursive Method* (15)

```
public static int sum (int n)
{
   int result;
   if (n == 1)
      result = 1;
   else
      result = n + sum (n-1);
   return result;
}
```

- Consists of a base case, and a recursive portion. For example a root is the base case, the tree is the recursive portion.
- A recursive method continues calling itself till it reaches the base case.
- When a recursive method calls itself, an activation record is set up, and pushed onto the execution stack.

*Methods You Might Find in Queue/Lists (20) + Lists (15)*

- Queues are FIFO, first in, first out.
- Queue Methods:
  - enqueue, dequeue, first (peek), isEmpty, size, toString.

---

- Ordered List = a list ordered in alphabetic or numerical order for example.
- Unordered List = a list created by adding to the front or rear of the list, or after a specific element.
- Indexed List = elements are referenced by numeric positions.
- Circular List = next element of rear points to the front.
- Doubly Linked List = each node in the list has a reference to the next and previous element.
- List Methods:
  - removeFirst, removeLast, remove, first, last, contains, isEmpty, size, iterator, toString.
- Unordered List Methods:
  - Above + addToFront, addToRear, addAfter.
- Comparable Interface (compareTo) method, declared using Comparable<T>, where T is a generic type.
  - Comparable<T> temp = (Comparable<T>)element;
  - Object 1 > Object 2 returns +
  - Object 1 < Object 2 returns -

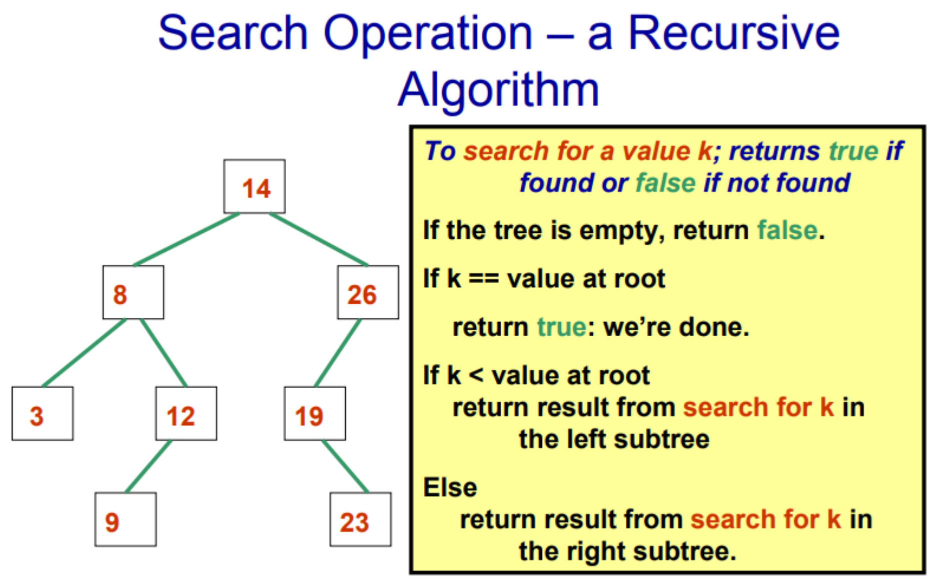
*Using pathToRoot/LCA/shortestPath (15)*

```java
public Iterator<T> pathToRoot(T targetElement) throws ElementNotFoundException {

     // Create an empty List.
     ArrayUnorderedList<T> tempList = pathToRootAgain(targetElement, root);

     // If the list remains empty, throw an exception.
     if (tempList == null)
          throw new ElementNotFoundException("pathToRoot not found.");

     // Return an iteration of the list obtained from the pathToRootAgain file.
     return tempList.iterator();
}


public ArrayUnorderedList<T> pathToRootAgain(T targetElement, BinaryTreeNode<T> next)
         throws ElementNotFoundException {

    // Assuming that the tree doesn't end after one node:
    if (next != null)
    {
        // If the next element is the target element create a temp list to store the element and return it.
        if (next.getElement().equals(targetElement)) {
            ArrayUnorderedList<T> tempListA = new ArrayUnorderedList<T>();
            tempListA.addToRear(next.getElement());
            return tempListA;
        }

        // Otherwise make another temp list and recursively search left then right of the tree for the
        // specific element.
        ArrayUnorderedList<T> tempListB = pathToRootAgain(targetElement, next.left);
        if (tempListB == null) {
            tempListB = pathToRootAgain(targetElement, next.right);
        }
        if (tempListB == null)
            return null;
        tempListB.addToRear(next.getElement());
        // Return the list containing the path to the specified node.
        return tempListB;
    }
    // If the tree does end in one node:
    else
        return null;
}

 public T lowestCommonAncestor(T targetOne, T targetTwo) throws ElementNotFoundException {

     // Initialize the iteration of both targets paths to the root.
     Iterator<T> tempList1 = pathToRoot(targetOne);
     Iterator<T> tempList2 = pathToRoot(targetTwo);
     // The temporary lists that will hold the iterated values above.
     ArrayUnorderedList<T> tempListA = new ArrayUnorderedList<T>();
     ArrayUnorderedList<T> tempListB = new ArrayUnorderedList<T>();
     // Placeholder for the LCA or lowest common ancestor.
     T lowestCommonAncestorResult = null;

     // While the first target's list path has values:
     while (tempList1.hasNext()) {
         tempListA.addToFront(tempList1.next());
     }

     // While the second target's list path has values:
     while (tempList2.hasNext()) {
         tempListB.addToFront(tempList2.next());
     }

     // Create an iteration of the new lists.
     Iterator<T> tempIter1 = tempListA.iterator();
     Iterator<T> tempIter2 = tempListB.iterator();

     // Check to see at each point as they move up the try, where the element matches.
     while (tempIter1.hasNext() && tempIter2.hasNext()) {
         T tempElement1 = tempIter1.next();
         T tempElement2 = tempIter2.next();
         if (tempElement1.equals(tempElement2)) {
             lowestCommonAncestorResult = tempElement1;
         }
     }
     return lowestCommonAncestorResult;
 }
```

```java
public Iterator<T> shortestPath(T targetOne, T targetTwo) throws ElementNotFoundException {
    ArrayUnorderedList<T> list1 = new ArrayUnorderedList<T>();
    ArrayUnorderedList<T> list2 = new ArrayUnorderedList<T>();
    T lca = lowestCommonAncestor(targetOne, targetTwo);
    Iterator<T> t1_path = pathToRoot(targetOne);
    Iterator<T> t2_path = pathToRoot(targetTwo);

    if (targetOne == null || targetTwo == null) {
        throw new ElementNotFoundException("binary tree"); // throw an exception if either target elements are null
    }

    while (t1_path.hasNext()) { // add elements in t1_path to list1
        list1.addToRear(t1_path.next());
    }

    while (!list1.last().equals(lca)) { // remove unnecessary elements from root until the lowest common ancestor so
                                        // that it makes sense
        list1.removeLast();
    }

    list1.removeLast(); // remove the overlapping element

    while (t2_path.hasNext()) { // add reversed path from the second target element to root
        list2.addToFront(t2_path.next());
    }

    while (!list2.first().equals(lca)) { // remove unnecessary elements from root until the LCA so that it makes
                                         // sense
        list2.removeFirst();
    }

    for (T i : list2) { // combine the two lists so that they connect and show the shortest path between
                        // the target elements
        list1.addToRear(i);
    }

    return list1.iterator();
}
```

*Recursive Tree Method (20)*

- Root = the origin element of the tree.
- Internal Node = an element in the tree.
- Leaf Node = a node without children (an ending node).
- Degree of a Node = the number of children a node has.
- Degree of a Tree = the largest degree of a node in the tree.
- General Tree = a tree whose nodes have no maximum child limit.
- N-ary Tree = a tree whose nodes have a maximum of N children.
- Binary Tree = a tree whose nodes have a maximum of 2 children.
- Binary Search Tree = a tree in which each left child is smaller then its parent node and right child is greater.




---

Things to Know in General: + (20) MC

---

*Time complexity*

- O(1) = constant = running time of the statement will not change with input (usually 1 loop).
- O(N) = linear = running time of the loops is proportional to a limiter N.
- O($N^2$) = quadratic = running time of the loops has it so if N doubles, run time increases by N*N.
- O(logn) = log = proportional to the number of times N Is divided by 2.
- Essentially, check how many times the code would run if n = 0, n = 1, n = 2, and n = 3.

*Reversing a Queue*

```
public void reverse(Queue q)
{
    Stack s = new Stack();  //create a stack

    //while the queue is not empty
    while(!q.isEmpty())
    {  //add the elements of the queue onto a stack
       s.push(q.serve());
    }

    //while the stack is not empty
    while(!s.isEmpty())
    { //add the elements in the stack back to the queue
      q.append(s.pop());
    }

}
```

*Recursive Solution on String Reversal*

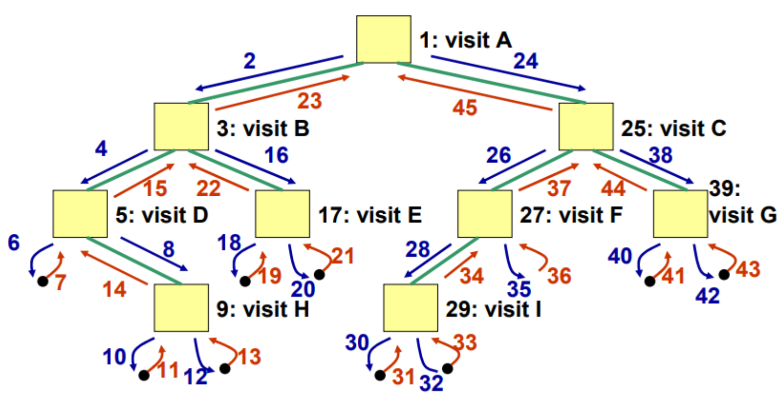
**Solution:**

```
ArrayStack<String> reverse = new ArrayStack<String>();
for (int i=0; i<userTokens.length; i++)
  reverse.push(userTokens[i]);
while(!reverse.isEmpty())
  System.out.print(reverse.pop() + " ");
}
}
```

*Traversal + Distinguish the Differences between Preorder/Inorder/Postorder (tree)*

- Preorder Traversal = Check if empty, start at the root, visit each node, and its children (left then right for trees and children).

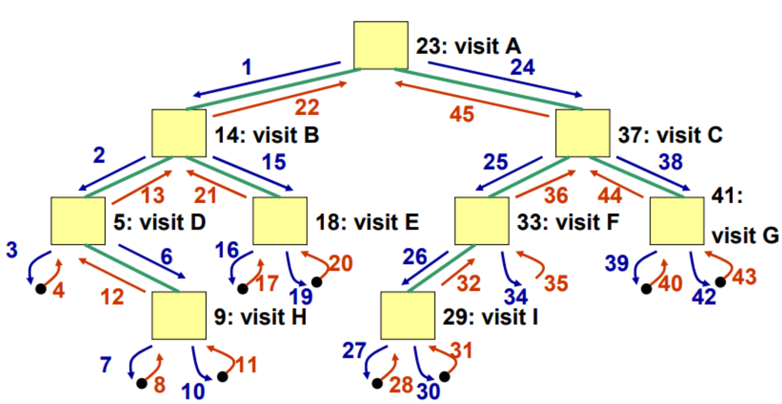
## Preorder Traversal




Nodes are visited in the order ABDHECFIG

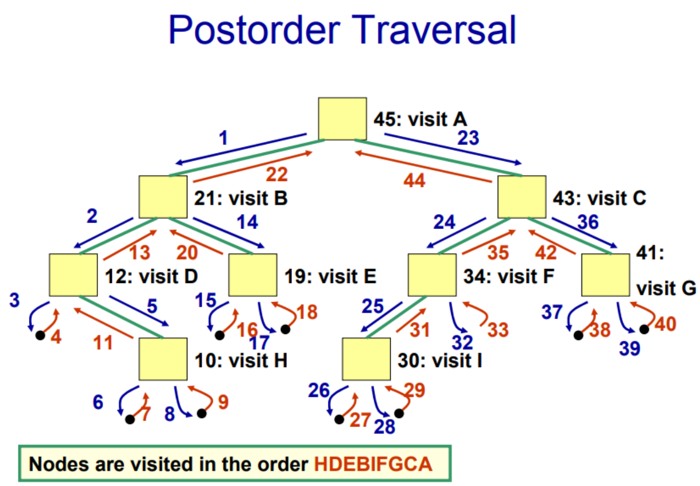
- Inorder Traversal = Check if empty, start at the left child of the root, visit the left child of each node, then the node (if there is no left child), then remaining nodes. Left then right tree.

## Inorder Traversal

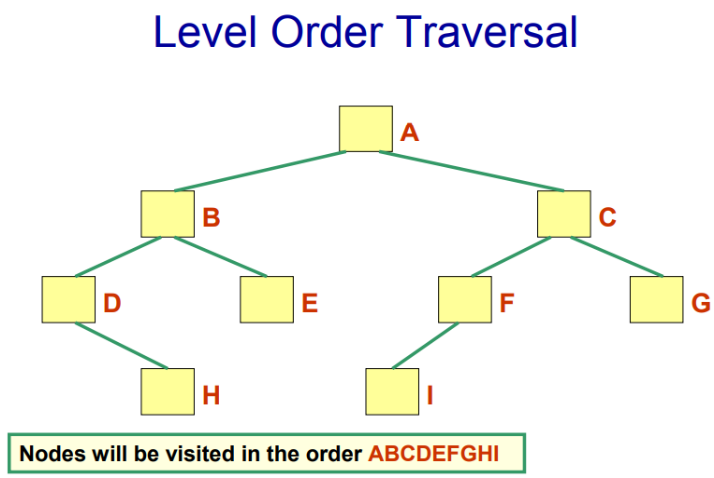



Nodes are visited in the order DHBEAIFCG

- Postorder Traversal = Check if empty, start at the left child of the root, visit the child of each node, then the node, and finally the root node. Left then right tree.




- Level Order Traversal = Check if empty, start at the root, visit the nodes at each level, from left to right.




- A tree either starts empty or with a root, or with a root and subtrees and leaves.

- The height of a tree is the length of the longest path from root to leaf, with an empty tree having a height of -1.

- The level of the root node is 0.

- During recursive iteration, the java execution stack keeps track of the current position. During normal iteration, the programmer must keep track.

- Using a **stack** as the container for a tree iteration gives a preorder traversal, while using a **queue** gives a level order traversal.

*Sorting*

- Insertion Sort = insert a value to the point it belongs in the order, and then move the value that used to be there 1 space back.

- In-Place Insertion Sort = same as insertion sort but can't use other data structures.

- Selection Sort = Find the smallest element in the unsorted list; swap it with the front of the unsorted list. Find the smallest element in the now smaller unsorted list and repeat.
    - Or remove the smallest from the list and enqueue it to the end of the sorted list.

- In-Place Selection Sort = swap the smallest element with the 1st element in the unsorted list and continue. Not using any extra data structures.

- Quick Sort = Pick an element around the middle of the list as a pivot. All items are stored in a container for items smaller then the pivot or one for those greater than the pivot; and one for equal items. Each container is recursively sorted and then added in the list in relation to the pivot.

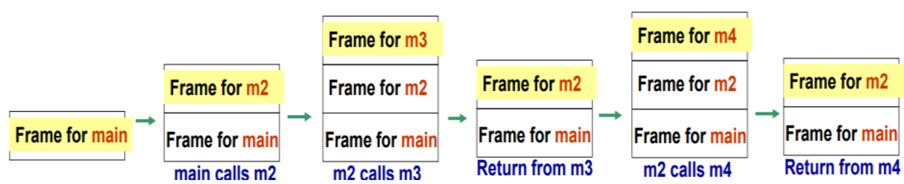
- Selection Sort has a time complexity of $O(n^2)$.

- In-Place Selection Sort has an $O(n^2)$ worst case time complexity, and a $O(n \log_2 n)$ complexity.

*Extra + What's Going on With Call Stack During Recursion?*

- Execution Stack / Call Stack = the memory space used while a method is being run.

- Activation Record / Call Frame = when a method is called, a reference is put onto the execution stack. This contains the:
    - Return address.
    - Given parameters.
    - Local variables.
    - Returns.




- Dynamic Heap = information stores about each **object**, including its local variables and the type of the object.

- Static Heap = contains one copy of each **class**, interface, static variable and static method named.

- Deallocation = once a method returns a value, the activation record is popped off the execution stack

- For example, if a new object is created, memory is allocated to the heap for creation, and reference memory is put on the call stack.

- When a ***new*** variable or method is called, memory is allocated on the execution stack. Afterwards the object is created in the heap.

- An object stays on the heap after it is made even without a reference variable, but garbage collection occurs in the execution stack.
- == → is a reference comparison, i.e. both objects point to the same memory location.
- .equals() → evaluates to the comparison of values in the object.
- Parent class = child class, and child class = parent class.
- / 0 = error.
- Anything after an error catch does not occur.
- Make sure to read through the code and check for /0 or other redundancies that would waste my time.
- Integer i = new Integer(1); → sets i to equal the address of new integer(1), which stores the value 1.
- Objects go in the heap, local variables go in the execution stack.
- Creating a variable referencing another variable, and changing it, changes the original variable, just the reference.
- Remember changing a variable in a method doesn’t change it outside the method, unless you return it and say the original variable equals the new one.