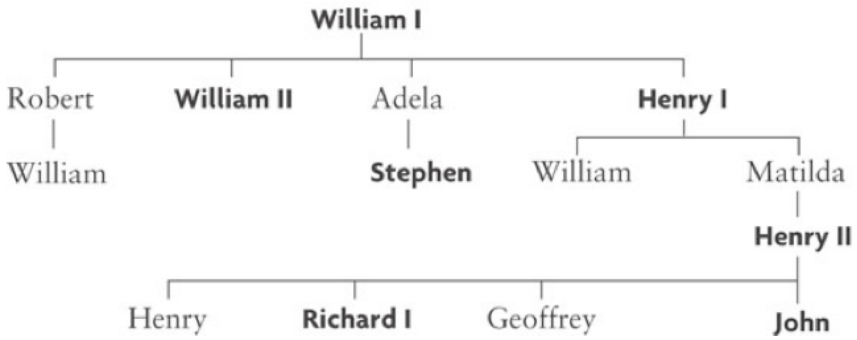
GALWAY-MAYO INSTITUTE OF TECHNOLOGY

Department of Computing & Mathematics

CP2SD – Data Structures & Algorithms
Lab 5: Trees

Trees are ubiquitous data structures in computing and are used to model real-world entities such as computer networks, adversarial zero-sum games and file systems. In this lab we will create a general tree data structure that we will use to store the partial family tree of William I, aka William the Conqueror, King of England and Duke of Normandy (William was basically a 2nd generation Viking with issues about his Nordic ancestry…). A diagrammatic representation of William's descendants tree is shown below:



- Download the Zip archive ***lab5Treess.zip*** from Moodle and extract the files into a folder on your hard disk. As the classes involved are all packaged, ensure that you create the correct directory structure and study the source code before you compile.

- Examine the class *Node.java*. *Node* is a composite reflexive object and can be used recursively, making it ideal for representing a recursive structure. In essence, this means that we can construct a tree by adding nodes to existing nodes and so on.

- Execute the class *TreeTest.java*. *TreeTest* is responsible for building the tree of William's descendants, in the ***init()*** method, and provides two mechanisms for transversing the tree structure:

    - ***dfs(Node n, int i):*** a recursive depth-first search that pushes each recursive method call onto the JVM method stack.
    - ***stackDFS(Node root):*** is a non-recursive stack-based depth-first search. Unlike the recursive method, child nodes are pushed onto a custom stack that exists in the JVM heap.

    Examine the tree structure outputted by the programme and draw out the progress of the tree transversal over the diagram of William's family tree shown above. Make sure that you understand the order of node transversal and which part of the ***dfs()*** method is controlling this.

The children of each node can be referenced, as a *Node* array, by calling the ***children()*** method. The ***for*** loop that follows the call to ***children()*** iterates over the child nodes and places each node onto the top of the stack. There are two major consequences of using a stack and a ***for*** loop in this manner:

1. The stack guarantees that the tree will be explored depth-first, i.e. by following each branch down to a leaf node and backtracking to the most recent ancestor that contains more than one child.
2. The order of the ***for*** loop determines whether the tree will be transversed from left to right or vice versa.

- Edit the ***for*** statement in the method ***dfs(Node node, int index)*** and change it to the following:

for (int i = children.length - 1; i >= 0; i--) {

After executing the class *TreeTest.java*, you should see that the tree has now been transversed from right to left instead of from left to right. Therefore, depending on the context, the order in which child nodes are arranged can be very important.

Questions & Exercises

- Replace the instance variable ***private Stack<Node> stack = new Stack<Node>()*** with a linked list as follows:

LinkedList<Node> stack = new LinkedList<Node>();

Note: you will have to import the ***java.util*** library at the top of the *TreeTest* class. Because *LinkedList* already implements all our stack methods, the *stackDFS(Node root)* method in unaffected by the changes.

- Instead of transversing a tree by exploring each branch first, down to its leaves (a depth-first search), we can also transverse a tree by examining each level (ply) at a time. This type of search is called a breadth-first search. Only a minor modification of our code is required to implement this functionality. A DFS works by pushing the most recently visited node to the top of a stack, or indeed the front of a queue (LIFO). A BFS can be implemented by adding the most recently encountered node to the back of a queue (FIFO). As *LinkedList* is implemented as a deque, all that is required is to change the statement ***stack.push(children[i])*** to ***stack.addLast(children[i])*** in the ***stackDFS(Node root)*** method.