Easy, Med, Hard, Ninja

June 26.2015

Please pair with someone who programs in the same language as you.

A. Easy – 3 exercises.

a. Create a function that prints out a singly-linked list on the console from head to tail. Use the unicode right arrow symbol between each node. For simplicity assume that nodes contain a string inside them as their data element.

```
void printLinkedList(Node head) {
     …
}
```

Output:

```
Apple⟶Boat⟶Cat⟶Dog⟶Eye
A
Z⟶12⟶M⟶String1
```

b. Write a function that creates a linked list with random letters(stored as strings) inside it. The method takes in a length range and letter range. It will return the head to the new list. Assume all incoming is valid and numNodes>=1 and minLetter<=maxLetter. Use printLinkedList to test your function.

```
Node createRandomList(  int numNodes
                        char minLetter,
                        char maxLetter) {
     …
}
```

Output:

createRandomList(6,A,C): A⟶A⟶C⟶B⟶A⟶C
createRandomList(5,C,C): C⟶C⟶C⟶C⟶C
createRandomList(2,A,Z): F⟶R

c. Adapt your createRandomList method so that it creates palindromic random linked lists.

```
Node createPalindromeList(   int numNodes,
                             char minLetter,
                             char maxLetter) {
    …
}
```

```
Output:    createPalindromeList(3,A,C):   A→B→A
           createPalindromeList(4,C,C):   C→C→C→C
           createPalindromeList(5,F,R):   F→N→P→N→F
           createRandomList(2,G,G):       G→G
```

B. Medium – 2 Exercises

a. Write methods to perform run-length encoding(aka RLE) and decoding on a linked list of capital letters. Let the encoding work as shown by the following examples (use the $ symbol to separate the number from the letter):

```
B→A→A→A       RLE-encodes to: B→3$A
B→A→A→Z→Z     RLE-encodes to: B→2$A→2$Z
B→R→Z→M       RLE-encodes to: B→R→Z→M

5$A           RLE-decodes to: A→A→A→A→A
3$B→C         RLE-decodes to: B→B→B→C
```

For simplicity, the method will return a brand new list and leave the original list intact. Use your printLinkedList to test the method.

```
    Node rleEncodeList(Node head) {
         …
    }

    Node rleDecodeList(Node head) {
         …
    }
```

b. Write a method to delete a key from the first occurrence from left to right in an RLE-encoded list. If the key isn't present, do nothing. Use printLinked list to test.

```
void deleteFromRLEList(char key, Node list) {
    ...
}
```

before:	list = 3$B ⟶ A	deleteFromRLEList('A', list)
after:	list = 3$B	
before:	list = B ⟶ 10$A ⟶ C ⟶ A	deleteFromRLEList('A', list)
after:	list = B ⟶ 9$A ⟶ C ⟶ A	
before:	list = B ⟶ A	deleteFromRLEList('B', list)
after:	list = A	

C. Hard – 2 Exercises

a. Adapt/type in the following code into your IDE to print out binary trees on your console:

Node.java:

```java
public class Node {
    String data; Node left, right;
    Node(String aData) { data = aData; }
    Node(String aData,String leftData,String rightData) {
        data = aData; left = new Node(leftData); right = new Node(rightData);
    }
    public void printTree() {
        if (right != null) right.printTree(true, "");
        printNodeValue();
        if (left != null) left.printTree(false, "");
    }
    private void printNodeValue() {
        System.out.print((data==null ? "<null>" : data) + '\n');
    }
    private void printTree(boolean isRight, String indent) {
        if (right != null) {
            right.printTree(true, indent + (isRight ? "        " : " |      "));
        }
        System.out.print(indent + (isRight ? " ┌" : " └") + "────── ");
        printNodeValue();
        if (left != null) {
            left.printTree(false, indent + (isRight ? " |      " : "        "));
        }
    }
}
```

Main.java:

```java
public class Main {
    public static void main(String[] args) {
        Node root = new Node("root","a","b");
        root.left.left = new Node("c");
        root.left.right = new Node("d");
        root.right.left = new Node("e");
        root.right.right = new Node("f",null,"g");
        root.printTree();
    }
}
```

Output:

```
                 ┌────── g
         ┌────── f
         |       └────── <null>
 ┌────── b
 |       └────── e
root
 |       ┌────── d
 └────── a
         └────── c
```

```
Node.cc:

class Node {
        public:
            const char *data; Node *left, *right;
            Node(const char *aData) {
                data = aData; left = right = 0;
            }
            Node(const char *aData,const char *leftNodeData,const char *rightNodeData) {
                data = aData;
                left = new Node(leftNodeData);
                right = new Node(rightNodeData);
            }
            void printTree() {
                if (right) right->printTree(true, string(""));
                printNodeValue();
                if (left) left->printTree(false, string(""));
            }
            void printNodeValue() {
                if ( data ) {
                    cout<<data<<'\n';
                } else {
                    cout<<"<null>"<<'\n';
                }
            }
            void printTree(bool isRight, const string &indent) {
                if (right) {
                    right->printTree(true, indent + (isRight ? "        " : " |      "));
                }
                cout << indent << (isRight ? " ┌" : " └") << "───────── ";
                printNodeValue();
                if (left) {
                    left->printTree(false, indent + (isRight ? " |      " : "        "));
                }
            }
        };

int main() {
    Node *root = new Node("root","a","b");
    root->left->left = new Node("c");
    root->left->right = new Node("d");
    root->right->left = new Node("e");
    root->right->right = new Node("f",0,"g");
    root->printTree();
    return 0;
}
```

b. Make a method to generate random sparse trees with random capital letters as nodes. Assume numNodes>=1.

```
        Node makeRandomTree(int numNodes)      {
                        …
        }
```

c. Write a method to serialize and de-serialize a binary tree to a vector (array). Assume the data in the node is a string. Use the print() and makeRandomTree() methods to test your implementation.

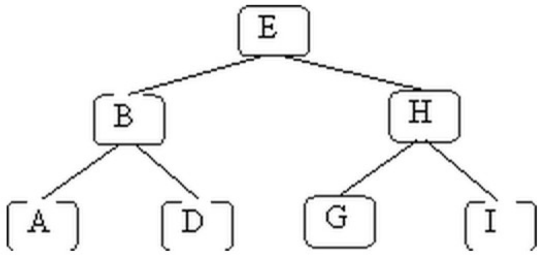
```
ArrayList<String> serializeTree(Node root) {
     ...
}

Node deserializeTree(ArrayList<String> vector) {
     ...
}
```

D. Ninja – 3 Exercises – Last Exercise is Ninja Challenge

a. Copy/implement a recursive method to the Node class in section C, that identifies whether a tree is a BST tree or not. The definition of a BST tree is:

1. Let k = parent's key.
2. All children to parent's left have keys <= k.
3. All children to parent's right have keys >= k.

For simplicity use the string comparator and use capital letters for nodes.
Example from https://www.cs.auckland.ac.nz/~jmor159/PLDS210/niemann/s_bin.htm :



Node.java:
```
public class Node {
    ...
    boolean isBST() {
        return isBST(this);
    }
    boolean isBST(Node node)
    {   Node prev = null;
        if (node != null)
        {   if (!isBST(node.left)) return false;
            if (  prev != null &&
                  node.data.compareTo(node.data)<=0 )
                return false;
            prev = node;
            return isBST(node.right);
        }
        return true;
    }
}
```

Node.cc

```
...

bool isBST() {

    return isBST(this);
}
bool isBST(Node *node)
{   Node *prev = 0;
```

```
        if (node)
        {   if (!isBST(node->left)) return false;
            if (   prev && prev->data &&
                    strcmp(prev->data,node->data)<=0 )
                return false;
            prev = node;
            return isBST(node->right);
        }
        return true;
    }
```

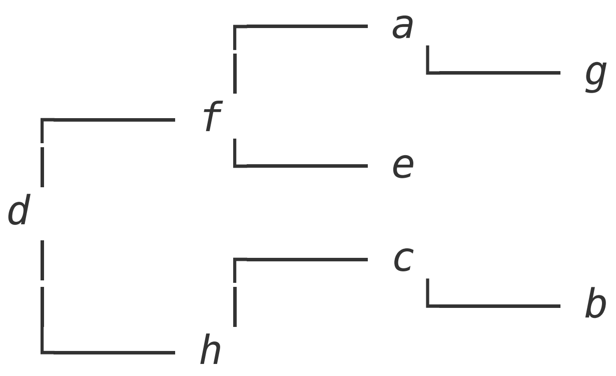
b. Copy this a function to generate a BST from a sorted array of capital letters.

```
public static Node sortedArrayToBST(String arr[]) {
    return sortedArrayToBST(arr,0,arr.length-1);
}
public static Node sortedArrayToBST(String arr[], int start, int end) {
      if (start > end) return null;
      int mid = (start + end)/2;
      Node node = new Node(arr[mid]);
      node.left =  sortedArrayToBST(arr, start, mid-1);
      node.right = sortedArrayToBST(arr, mid+1, end);
      return node;
}
```

c.   From Elements of Programming Interviews. Problem 15.16.

"Test if a binary tree is an almost BST.

Define a binary tree with [capital-letter]-labeled nodes to an almost BST if it does not satisfy the BST property, but there exists a pair such that swapping the keys at the nodes makes the resulting binary tree a BST." See example below for an "almost" BST example:



So if only 1 single swap where "h" were swapped "a", this would be a BST. Hence, it's an "almost" BST.

Design an algorithm that takes as input a binary tree with capital-letter labeled nodes, and determines if it an almost BST. If it is an almost BST, reconstruct the original BST.

Answer to Ninja problem:

"The brute-force approach is to create an array of nodes as they appear in an inorder traversal. We traverse this array. If it is already sorted according to the key at each node, the binary tree is already a BST. Otherwise, for it to be an almost BST, there must be a pair of array entries which after swapping result in a sorted array. Such a pair can be identified via a par of nested for-loops with O(n^2) time complexity, where n is the number of nodes in the tree.

A better approach is to traverse the array with a single for loop and find nodes that are out of order. Specifically, if the a pair that needs to be swapped is adjacent in the array, there will be a single entry which is smaller than its predecessor. Otherwise, there are two such entries. In either case, we perform a swap.

We can circumvent the O(n) additional space of the array by performing an in-order traversal and implicitly searching for the out-of-order nodes. By the earlier argument, a binary tree is an almost BST if there is a pair of keys which must be swapped to get a sorted order. The corresponding BST for the binary tree is derived by swapping these keys. If the two keys that must be swapped correspond to a node and its successor then we simply exchange these. Otherwise there must be two inverted node pairs. Let these node pairs be (u,v) and (w,x). In this case we swap the key at x with the key at u....
The code below used an explicit variable to track the predecessor of the first node that is out of place through the traversal. Note that any inorder traversal – recursive, iterative, etc., can be used.

https://github.com/epibook/epibook.github.io/blob/master/solutions/java/src/main/java/com/epi/ReconstructAlmostBst.java

https://github.com/epibook/epibook.github.io/blob/master/solutions/cpp/reconstruct-almost-bst.cc

Note: The answer uses integer keys. It would need to be adapted to use strings.