LAB 1 - Prefix-free Codes Using Binary Trees

Submission Deadline: Jan. 24, 11:59 pm
Assessment: 5% of the total course mark.

Description:

In this assignment you are required to implement prefix-free codes using binary trees with linked nodes. For this, you have to write the `Java` classes `BinTree`, `TNode`, and `Test` in the same package. The class `BinTree` uses binary trees with linked nodes to represent prefix-free codes. The class `TNode` represents the nodes of the binary tree (its source code is provided). The class `Test` should provide a thorough testing of all constructors and public methods of `BinTree`. You also need to perform the time and space complexity analysis of your algorithms.

You are not allowed to use any predefined `Java` methods other than those defined in the classes `java.util.ArrayList`, `java.lang.String` and `java.lang.Math`.

Definitions:

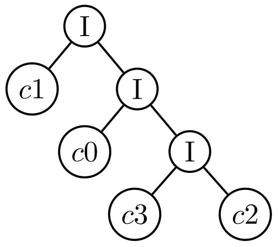
A prefix-free code (used in data compression) is a set of binary sequences (sequences of 0's and 1's) such that none of them is a prefix of another. For instance, the set $\mathcal{A} = \{0, 10, 110, 111\}$ is a prefix-free code. On the other hand, the set $\{0, 10, 100, 111\}$ is not a prefix-free code because the sequence 10 is a prefix of 100. The elements of the prefix-free code are referred to as (binary) **codewords**.

A prefix-free code can be used to **encode** a sequence of symbols from an alphabet $\mathcal{B}$ (i.e., **convert the sequence of symbols into a bitstream**) as follows. Each symbol in alphabet $\mathcal{B} = \{c0, c1, c2, \cdots, cn\}$ is assigned a distinct binary codeword. Then we can encode any sequence of symbols by replacing every symbol by the corresponding binary codeword. For instance, assume that alphabet $\mathcal{B}$ contains 4 symbols. Then they are $c0$, $c1$, $c2$, $c3$. Further, assume that symbol $c0$ is assigned 10, $c1$ is assigned 0, $c2$ is assigned 111 and $c3$ is assigned 110. Consider now the following sequence of symbols over the alphabet $\mathcal{B}$: "$c3$ $c2$ $c0$ $c3$ $c1$ $c1$ $c1$". This sequence is encoded as "110, 111, 10, 110, 0, 0, 0". The corresponding bitstream is obtained by removing the commas: "11011110110000".

Conversely, a bit sequence is **decoded** by dividing it first into non-overlapping codewords and then replacing each codeword with the corresponding alphabet symbol. For instance, the sequence "0011001010111" can be divided as follows: "0,0,110,0,10,10,111". Then the decoded alphabet sequence is "$c1$ $c1$ $c3$ $c1$ $c0$ $c0$ $c2$". **Note that the prefix-free property ensures that the binary sequence can be divided into codewords in a unique manner.**

A prefix-free code can be represented using a binary tree. Note first that any path in the tree, from some node to one of its descendants, can be regarded as a sequence of branches. By replacing any left branch with 0 and any right branch with 1 we obtain a binary sequence that represents the path. In a binary tree representation of a prefix-free code, the paths from the root to the leaves represent the codewords. Thus, the number of leaves equals the number of codewords. **Each leaf stores the alphabet symbol corresponding to the binary codeword that describes the path from the root to that leaf.**

Example: For the prefix-free code $\mathcal{A} = \{0, 10, 110, 111\}$ used to encode the symbols of alphabet $\mathcal{B}$ as described above, the binary tree is the following ("I" stands for "internal node"):



SPECIFICATIONS:

You **have to use** the following `Java` class `TNode`:

```
public class TNode {
   String data;
   TNode left;
   TNode right;

   TNode(String s, TNode l, TNode r){
       data=s;
       left=l;
       right=r;
   }
}
```

For any node corresponding to a codeword, the field `data` stores the corresponding alphabet symbol as a string. For any other node the field `data` should be `null`.

Class `BinTree` **has only two instance fields**, namely a reference to the root of the tree (a reference variable of type `TNode`) and an integer `num` that stores the number of codewords. Both have to be `private`.

Class `BinTree` contains two constructors:

- `public BinTree()` - constructs a `BinTree` that contains only the root node.
- `public BinTree(String[] a) throws IllegalArgumentException` - constructs a `BinTree` that represents the prefix-free code stored in the `String` array `a`. Each `String` in the array `a` is a binary sequence (contains only 0's and 1's) representing a codeword. The codeword stored in `a[i]` corresponds to the alphabet symbol *ci*. Thus, the tree leaf corresponding to this codeword must store the `String`: `"c" + i`. If array `a` contains two identical codewords, or a codeword that is a prefix of another one, then the set of codewords is not prefix-free and the constructor should

throw an `IllegalArgumentException` with the message "Prefix condition violated!". If any element in the array is not a binary string, the method should **throw an** `IllegalArgumentException` with the message "Invalid argument!". **You may assume that the array is nonempty.**

Example: Input array: "10", "0", "111", "110". Then 10 is the codeword for *c*0, 0 is for *c*1, 111 is for *c*2 and 110 is for *c*3. The corresponding tree is the tree at the top of the page.

The second constructor can use the following private method

- `private void insertCodeword(String symbol, String binary) throws IllegalArgumentException` - inserts a node in the `BinTree` that represents the codeword `binary`, which is assigned to `symbol`. You may assume that `symbol` is a valid symbol, `binary` is a valid binary string, and the tree is non-empty (i.e., it contains at least the root node) and it represents a prefix code. This method reads the characters in the binary string and follows the corresponding path in the tree starting at the root. If the tree ends before the path is completed, the method should add additional nodes as internal nodes. The last node to be added is a leaf that should store the corresponding symbol. This method should also check whether any node encountered on the path is a node representing a codeword. If this happens, this means that the encountered codeword is a prefix of the binary string that was input (so the code we attempt to store is not prefix free) and the method should **throw an** `IllegalArgumentException` with the message "Invalid argument!".

Class `BinTree` contains the following public methods (**you may declare additional** `private` **methods, but not** `public` **methods**):

- `public void printTree()` - visits the tree nodes through an **inorder** traversal and prints "I " for each internal node, and the symbol stored in the leaf (in `data`) followed by space for each leaf. It invokes a recursive private method `printTree()`.

  Recall that the inorder traversal first visits recursively the left subtree, next visits the root, and last visits recursively the right subtree.

  For the tree in our example, this method will print: c1 I c0 I c3 I c2

  You **have to use** the following `Java` code:

```
public void printTree(){ printTree(root);}

private void printTree(TNode t){
   if(t!=null){
       printTree(t.left);
       if(t.data == null )
           System.out.print("I ");
```

```
            else
                System.out.print(t.data + " ");
            printTree(t.right);
        }
    }
```

- `public int getNumberOfCodewords()` - returns the number of codewords stored in the tree.
- `public int height()` - returns the height of the tree. It invokes the recursive private method `height()`.
- `public ArrayList<String> getCodewords()` - returns an `ArrayList<String>` object that stores the codewords in lexicographical order[1]. Each item in the list is a codeword (a string of 0's and 1's). This can be done using an inorder, preorder or postorder traversal of the tree, while also keeping track of the path from the root to the visited node (as an additional parameter passed to the recursive method). When visiting a node that represents a codeword (i.e., `data != null`), the codeword (i.e. the string representing the current path) should be inserted at the end of the list using a method in the `ArrayList` class.
- `public ArrayList<String> decode(String s)`. The input string `s` contains only 0's and 1's). This method outputs the sequence of alphabet symbols obtained by decoding the binary sequence `s`. Each alphabet symbol has to be stored as a separate item in the list. The order of items is important here. You may assume that `s` is a nonempty, valid binary sequence that can be divided into codewords.
- `public String toString()` - returns the string representation of the prefix-free code as a sequence of pairs (symbol, codeword) listed in increasing order of the symbol index, separated by empty spaces and ending with an empty space. For our example, this string is "(c0,10) (c1,0) (c2,111) (c3, 110) ".
- `public String[] toArray()` - returns an array representation of the binary tree. You have to use the convention given in COMP ENG 2SI3 for the representation of binary trees using arrays. The first array element stores `null`. Any array element corresponding to a missing tree node stores `null`. Any array element corresponding to an internal node stores `"I"`. Any array element corresponding to a leaf stores the codeword corresponding to that leaf.
  This method can be implemented as follows. Find first the height h of the tree. Next allocate an array of `String` objects of size 2^{h+1}, with all elements equal to `null`. Next perform a tree traversal (inorder, preorder or postorder) keeeping track of the path to the current node. The current path indicates at which array position the node has to be stored.

You have to include meaningful comments at key points in your code.

For each method, you have to indicate in a comment the time and space complexities of the algorithm and include a brief, but clear justification.

[1]The lexicographical order of two binary strings is the same as their alphabetical order if we replace '0' by 'a' and 1 by 'b'.

Your algorithms have to be efficient.

In addition, you have to prepare a test class `Test` that performs a thorough testing of all the constructors and public methods.

Bonus: You can earn a bonus of up to 0.5% of the course mark if you implement efficiently the following method:

◇ `public void optimize()`. If the tree is not a full binary tree (i.e., where each internal node has two children), then the lengths of some codewords can be reduced by removing some bits, while the prefix-free condition is still maintained. This method checks if the tree is full and if it is not, it performs these changes (i.e., it reduces the lengths of some codewords and adapts the tree) until the tree becomes full. You may assume that the tree stores at least two codewords.

Include a brief justification of the time and space complexities. Provide a thorough testing.

Submission Instructions: Submit the source code for the `Java` class `BinTree` and for the test class as well as the output of your test class, each in a text file.

Include your name and student number in the names of the files. For instance, if your name is "Ellen Davis" and the student number is 12345 then the file should be named "BinTreeEllenDavis12345.txt", "Lab1TestEllenDavis12345.txt", "Lab1OutputEllenDavis12345.txt".