Com S 228 Fall 2023

Project 4: Archived Message Reconstruction (100 pts)

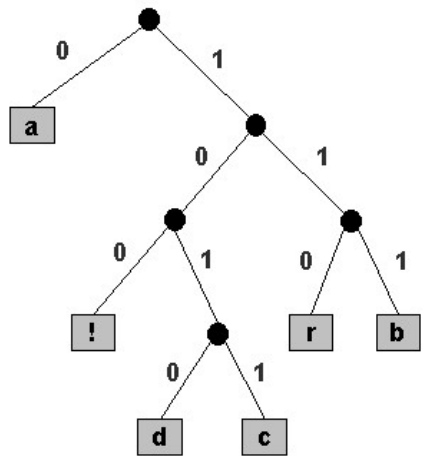
Due at 11:59 pm on **Thu, Nov. 30**

1. Problem Description

The objective of this exercise is to reconstruct/unzip a message archived with a binary-tree-based algorithm. The program should ask for a single filename at the start: "**Please enter filename to decode:** ", decode the message in the file and print it out to the console. The name of the compressed message file will end in .arch, e.g. “monalisa.arch”. The file consists of two or three lines: the first one or two lines contain the encoding scheme, and the second or third line contains the archived message.

2. Encoding

The archival algorithm uses a binary tree. The edges of the tree represent bits, and the leaf nodes contain one character each. Internal nodes are empty. An edge to a **left** child always represents a 0, and an edge to a **right** child always represents a 1. Characters are encoded by the sequence of bits along a path from the root to a particular leaf. The below tree serves as an example.



The tree on the left encodes these characters:

Character	Encoding
a	0
!	100
d	1010
c	1011
r	110
b	111

With the above encoding, the bit string:

`10110101011101101010100` is parsed as

`1011|0|1010|111|0|110|1010|100`

which is decoded as:

```
cadbard!
```

With this encoding, we can automatically infer where one character ends and another begins. That is because no character code can be the start of another character code. For example, if you have a character with the code **111**, you cannot have the codes **1** and **11**, as they would be internal nodes.

The following steps decode one character from the bit string:

```
Start at root
Repeat until at leaf
    Scan one bit
    Go to left child if 0; else go to right child
Print leaf payload
```

3. Input Format

The archive file consists of two lines: the first line contains the encoding scheme, and the second line contains the compressed string. For ease of development and to make the archive file human-readable, each bit is represented as the character '0' or '1', rather than as an actual bit from a binary file.

The encoding scheme can be represented as a string. For example, the tree from section 2 can be represented as:

```
^a^^!^dc^rb
```

where ^ indicates an internal node. The above code represents a **preorder traversal** of the tree.

The `cadbard!` message is encoded in the following file ("cadbard.arch"):

```
^a^^!^dc^rb
10110101011101101010100
```

There are four test files in HW4S2021_Test_Files.zip. **Note:** the encoding scheme representations may include a **space** character and a **newline** character, thereby breaking the tree string into *two lines*! The newline character needs to be parsed correctly if the encoding file has three lines in total.

4. Task

4.1. Read in the first line (and possibly second line, if newline is part of the tree) of the file and construct the character tree. Convert the line input into a MsgTree structure using preorder traversal. The tree should be in a class MsgTree with the following members:

```
public class MsgTree{
```

```
  public char payloadChar;
  public MsgTree left;
  public MsgTree right;

  /*Can use a static char idx to the tree string for recursive
  solution, but it is not strictly necessary*/
  private static int staticCharIdx = 0;
  //Constructor building the tree from a string
  public MsgTree(String encodingString){}
  //Constructor for a single node with null children
  public MsgTree(char payloadChar){}
  //method to print characters and their binary codes
  public static void printCodes(MsgTree root, String code){}
}
```

When building the tree, try a recursive solution where staticCharIdx tracks the location within the tree string. You can pass the same tree string during recursive calls, and update the staticCharIdx to point to the next character to be read. **Note:** if you decide to implement an iterative solution, you will receive a 15% bonus, as it is considerably more difficult. In that case, you cannot get the 5% bonus for printing statistics.

`printCodes()` performs recursive **preorder traversal** of the MsgTree and prints all the characters and their bit codes:

```
character  code
-------------------------
   c       1011
   r       110
   b       111
```

You are allowed to print the header of the table (character, code, ----) in main().

4.2. Write a method `public void decode(MsgTree codes, String msg)` to decode the message. It would print the decoded message to the console:

```
MESSAGE:
The quick brown fox jumped over the lazy dog.
```

You are allowed to print "MESSAGE:" in main().

The overall output of the program should be the output of `printCodes()` followed by the output of `decode()`:

```
character  code
-------------------------
   c       1011
   r       110
   b       111
```

```
MESSAGE:
The quick brown fox jumped over the lazy dog.
```

5. Submission

Put your classes in the `edu.iastate.cs228.hw4` package. Turn in the zip file and not your class files.

Include the Javadoc tag `@author` in each class source file. Your zip file should be named `Firstname_Lastname_HW4.zip`. No template files will be provided other than the skeleton in Section 4.1.

6. Extra credit (5% or 15%)

Print message-specific (not just encoding) statistics after the rest of the program output.

```
STATISTICS:
Avg bits/char:              8.0
Total characters:           1180
Space savings:              50.0%
```

The space savings calculation assumes that an uncompressed character is encoded with 16 bits. It is defined as `(1 – compressedBits/uncompressedBits)*100`. **compressedBits** is the sum of all characters in the message multiplied by each character's individual bits.

To earn a 15% non-cumulative bonus (either 5% for statistics or 15%), you can create an non-recursive, iterative solution for building the tree, but be advised that it will require hours of extra effort compared to the recursive solution. The bonus for early submission will stack with the 5/15% bonus.

Name your submission `Firstname_Lastname_HW4_extra.zip` if you completed the iterative solution.