Problem Set 8 - Solution

Binary Tree, Huffman Coding

---

1. * Answer the following questions in terms of *h*, the height of a binary tree:
   1. What is the **minimum** possible number of nodes in a binary tree of height *h*?
   2. A *strictly* binary tree is one in which every node has either no children or two children; in other words, there is **no** node that has exactly one child. What is the **minimum** possible number of nodes in a strictly binary tree of height *h*?
   3. A *complete* binary tree is one in which every level **but** the last has the maximum number of nodes possible at that level; the last level may have any number of nodes. What is the **minimum** possible number of nodes in a complete binary tree of height *h*?

   SOLUTION

   1. h+1 - one node at every level, and there are (h+1) levels (levels are numbered 0, 1, ..., h)
   2. Every level except the root level has 2 nodes. So, 1 + 2*h
   3. Level 0 has 2^0 nodes, level 1 has 2^1 nodes, and so on. Level h-1 has $2^{(h-1)}$ nodes. The last level has one node. The total is $2^h-1+1 = 2^h$.

---

2. Two binary trees are *isomorphic* if they have the same shape (i.e. they have identical structures.) Implement the following **recursive** method:

```
public static <T> boolean isomorphic(BTNode<T> T1, BTNode<T> T2) {
 /* your code here */
}
```

   that returns **true** if the trees rooted at T1 and T2 are isomorphic, and false otherwise. `BTNode` is defined as follows:

```
public class BTNode<T> {
    T data;
    BTNode<T> left, right;
    BTNode(T data, BTNode<T> left, BTNode<T> right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }
}
```

   SOLUTION

```
public static <T> boolean isomorphic(BTNode<T> T1, BTNode<T> T2) {
   if (T1 == null && T2 == null) return true;
   if (T1 == null || T2 == null) return false;
   if (!isomorphic(T1.left, T2.left)) return false;
   return isomorphic(T1.right, T2.right);
}
```

---

3. Exercise 9.2, page 294 of the textbook.

   You are given the following preorder and inorder traversals of a (boolean) expression tree:

```
Preorder traversal:   ||  &&  ||  a  >  b  c  <  d  e  &&  ==  !=  c  a  f  <  p  q
Inorder traversal:    a  ||  b  >  c  &&  d  <  e  ||  c  !=  a  ==  f  &&  p  <  q
```

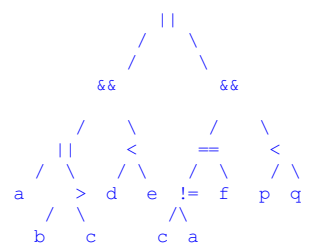
   Build the expression tree from these traverals. (The first `||` symbol in the preorder traversal matches with the later `||` in the inorder.)

   Assume the traversals are stored in string arrays. Assume a recursive build tree method that takes five parameters: the two arrays, the preorder index, the inorder low index and the inorder high index (current inorder subarray limits).

   In your tree building process, specify, for every subtree, the index into the preorder traveral array and the index limits of the inorder array.

   SOLUTION

   Here is the resulting expression tree:



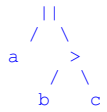
   The process:

   i = 0, lo= 0, hi = 18

   Pick first node, pre[i] in preorder as root. This matches with in[9].

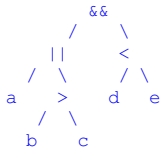
   - Therefore, for left side i = 1, lo = 0, hi = 8. I will go over the left subtree of the root before I start with the right subtree of the root. We make pre[1] = && as the root of this subtree. Clearly in[5] is the root. This seperate the left subtree into two subtrees.
     - The left side of left subtree from root has i = 2, lo = 0,hi = 4. pre[2]=|| becomes the root for this subtree which correspond to in[1].
       - The left subtree of this has i = 3,lo = 0, hi = 0, create just a leaf a.
       - But the right subtree of this has i = 4, lo = 2, hi = 4. Therefore, > becomes root for that subtree and b become left leaf and c becomes right leaf.

     After returning those, the left subtree of the left subtree from the root will become



- The right side of the left subtree from root has i = 7,lo = 6, hi= 8. pre[7]= < becomes the root of this subtree, and d becomes the left leaf and e becomes the right leaf.

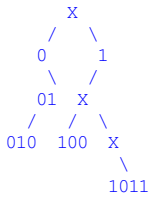
When these two subtrees of the left subtree from the root is returned, then the tree becomes



After both subtrees return, we will get the resultant expression tree.

---

4. The *radix tree* data structure shown below stores the bit strings 0,1,01,010,100, and 1011 in such a way that each left branch represents a 0 and each right branch represents a 1.



Nodes that do not have any stored bit strings will have a dummy value 'X' instead.

To find whether a bit string exists in this radix tree, start from the root, and scanning the bits of the string left to right, take a left turn if the but is 0, and a right turn if the bit is 1. If a node can be reached using this sequence of turns, and it does not contain the dummy value 'X', then the bit string is found, else it is not found.

1. Given the following bit strings:

   ```
   1011, 01, 0010, 1010, 011, 1000, 0101
   ```

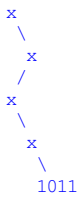
   Starting with an empty radix tree, build it up to store these strings, showing the radix tree after *each* bit string is inserted. (To insert a new string you may have to insert more than one new node in the tree built thus far.)
2. How many units of time did it take to build this tree? Treat taking a turn at an existing branch, and inserting a new branch as basic unit time operations.
3. How many units of time would it take to *lexicographically sort* the bit strings in this radix tree, after all the strings have been inserted? Use the same basic operations as in the previous question. The output of the sort should be:

   ```
   0010  01  0101  011  1000  1010  1011
   ```

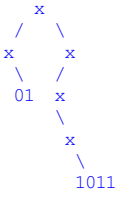
   (Lexicographic sort is like alphabetic sort, 0 precedes 1)
4. How many units of time would it take in the worst case to insert a new *k*-bit string into a radix tree? (ANY radix tree, not the specific one above.)
5. How many units of time would it take in the worst case to insert an arbitrary number of bit strings whose total length is *n* bits?

SOLUTION

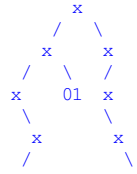
1. After inserting 1011:



   After inserting 01:

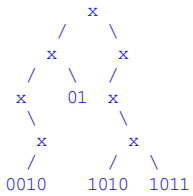


   After inserting 0010:

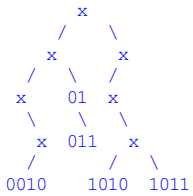


```
0010       1011
```

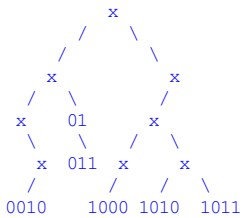
After inserting 1010:



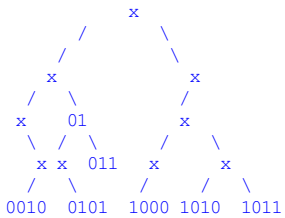
After inserting 011:



After inserting 1000:



After inserting 0101:



2. The number of turns plus new branches is equal to the length of the string being added. So total units of time is 4 + 2 + 4 + 4 + 3 + 4 + 4 = 25.
3. The lexicographic sort is equivalent to a preorder traversal on the radix tree, printing only at the nodes that have values. The first value that is printed, 0010, will need 4 turns starting from the root. Then the preorder traversal will eventually print 01, after backtracking to the parent of 01 and then taking a right turn to get to 01. Backtracking does not count for turns since it is implemented automatically in the recursion.

   The turns for all strings are as follows:

```
   0010: 4  (from root)
   01:   1  (right subtree from great-grandparent of 0010)
   0101: 2  (left subtree from 01)
   011:  1  (right subtree from 01)
   1000: 4  (from root)
   1010: 2  (right subtree from grandparent of 1000)
   1011: 1  (right subtree from parent of 1010)
-------------
  Total: 15
-------------
```

4. To insert a binary string of *k* bits would require *k* turns/new branches, so *k* units of time.
5. For an arbitrary number of bit strings whose total length is *n*, the total number of turns/new branches would be *n*, so *n* units of time.

---

5. Exercise 9.4, page 295 of the textbook.

   1. Build a Huffman tree for the following set of characters, given their frequencies:

```
R   C   L   B   H   A   E
6   6   6  10  15  20  37
```

   2. Using this Huffman tree, encode the following text:

```
CLEARHEARBARE
```

   3. What is the average code length?
   4. If it takes 7 bits to represent a character without encoding, then for the above text, what is the ratio of the encoded length to the unencoded?
   5. Decode the following (the string has been broken up into 7-bit chunks for readability):

```
1111011  1010111  1101110  0010011  111000
```

SOLUTION

1. The probabilities of occurence of the characters, listed in ascending order:

```
R    C    L    B    H    A    E
```

```
0.06 0.06 0.06 0.1  0.15  0.2  0.37

        1.0
     0 /    \ 1
      /      \
     E       0.63
          0 /   \ 1
           /     \
        0.27     0.36
      0 /  \1   0/   \1
       /    \   /     \
     0.12    H 0.16    A
   0 /   \1  0/   \1
    /     \  /     \
   R      C  L      B
```

```
R 1000
C 1001
L 1100
B 1101
A 111
H 101
E 0
```

2. `100111000111100010101111000110111110000`
3. `1*0.37 + 4*0.06 + 4*0.06 + 3*0.15 + 4*0.06 + 4*0.10 + 3*0.20 = 2.54`
4. Length of unencoded representation using 7 bits per character is 7*13=91, while length of representation using Huffman codes is 39. The ratio of encoded to unencoded is 39/91.
5. `AHBEABLECAR`