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:

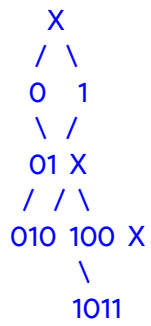
```
public class BTNode<T> {
   T data;
   BTNode<T> left, right;
   BTNode(T data, BTNode<T> left, BTNode<T> right) {
      this.data = data;
      this.left = left;
      this.right = right;
   }
}
```

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. 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

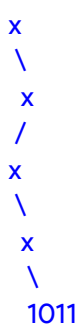
   Starting with an empty radix tree, build it up to store these strings, showing the radix tree after *each* bit string is inserted. (To insert a new string you may have to insert more than one new node in the tree built thus far.)
2. How many units of time did it take to build this tree? Treat taking 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

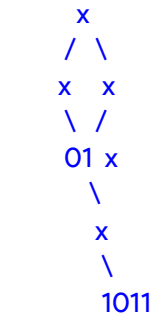
   (Lexicographic sort is like alphabetic sort, 0 precedes 1)
4. How many units of time would it take in the worst case to insert a new *k*-bit string into a radix tree? (ANY radix tree, not the specific one above.)
5. How many units of time would it take in the worst case to insert an arbitrary number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

SOLUTION

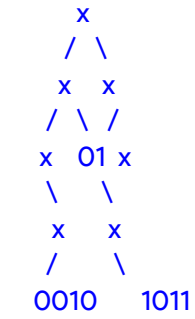
1. After inserting 1011:



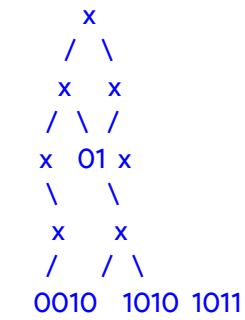
   After inserting 01:



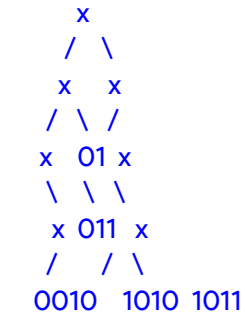
After inserting 0010:



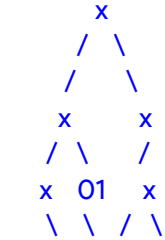
After inserting 1010:

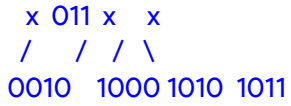


After inserting 011:

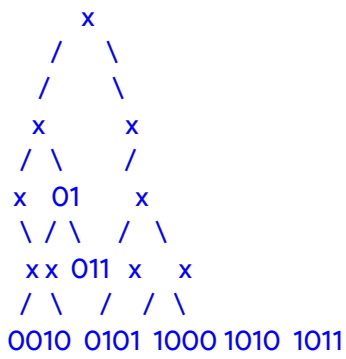


After inserting 1000:





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)

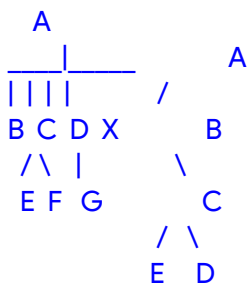
   Total: 15

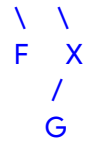
4. To insert a binary string of *k* bits would require *k* turns/new branches, so *k* units of time.
5. For an arbitrary number of bit strings whose total length is *n*, the total number of turns/new branches would be *n*, so *n* units of time.

---

4. A general tree is one in which a node can have any number of children. General trees are used to model hierarchies. Think of a company hierarchy with a CEO at the root, then presidents of various units that report to her at the next level, then various vice-presidents who report to the presidents at the next level, and so on. Computer file systems are hierarchies as well, with a root folder, with other folders and files recursively nested under the root.

   Every general tree can be represented as a binary tree. Here's an example of how:



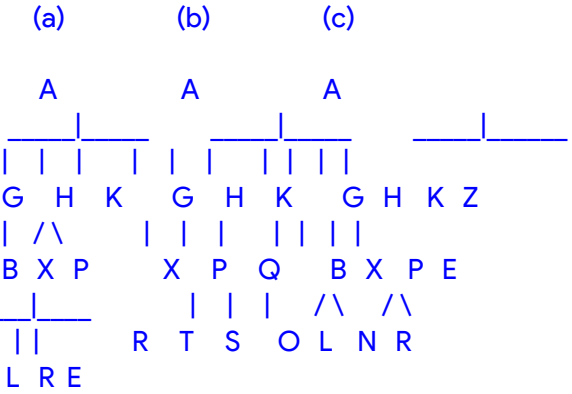


For every node in the general tree:

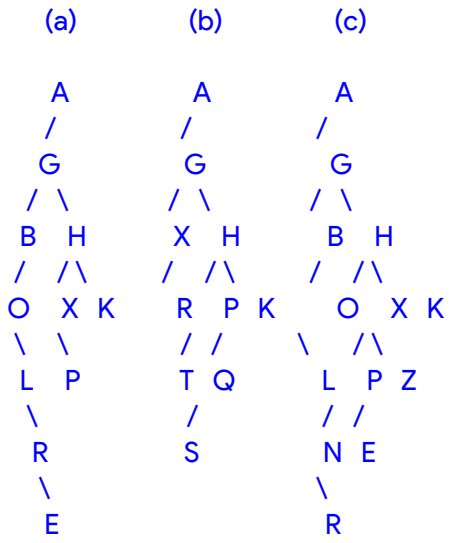
- The first child of the node becomes the left child of the same node in the binary tree
- The second child of the node becomes the right child of the first
- The third child of the node becomes the right child of the second, and so on

1. Exercise 9.8 of text

   Draw the binary tree representations of the following general trees:

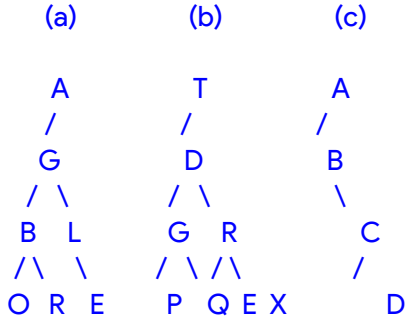


   SOLUTION

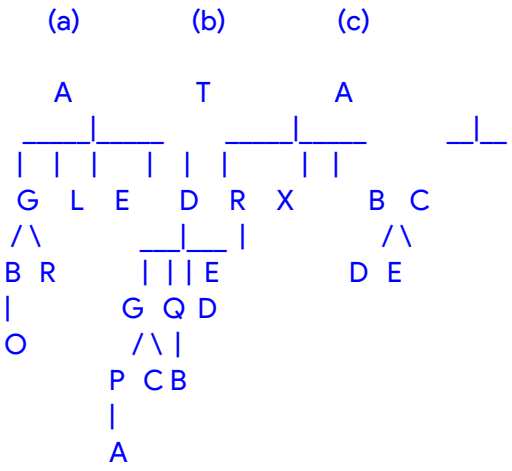


2. Exercise 9.9 of text

   Draw the general trees represented by the following binary trees:



SOLUTION



---

5. Suppose the equivalent binary tree for a general tree is built out of the following binary tree nodes:

```
public class BTNode<T> {
  T data;
  BTNode<T> left, right, parent;
  ...
}
```

Complete the following methods

1. Given a pointer to a node x, find and return the node that would be x's parent in the general tree:

```
public static <T> BTNode<T> genTreeParent(BTNode<T> x) {
   /* COMPLETE THIS METHOD */
}
```

SOLUTION

```
public static <T> BTNode<T> genTreeParent(BTNode<T> x) {
  while (x.parent != null) {
    if (x == x.parent.left) {
      return x.parent;
    }
    x = x.parent;
  }
  return null;
}
```

2. * Given a pointer to a node x, and an integer k, find and return the node that would be the k-th child of x (k=1 if first child, etc.), in the general tree. The method should thrown an exception if there is no k-th child:

```
public static <T> BTNode<T> genTreekthChild(BTNode<T> x, int x)
throws NoSuchElementException {
```

```
    /* COMPLETE THIS METHOD */
  }
```

SOLUTION

```
public static <T> BTNode<T> genTreekthChild(BTNode<T> x, int k)
throws NoSuchElementException {
  if (k <= 0 || x.left == null) {
    throw new NoSuchElementException();
  }
  x = x.left; k--;
  while (k > 0 && x.right != null) {
    x = x.right;
    k--;
  }
  if (k > 0 && x.right == null) { // k > #children
    throw new NoSuchElementException();
  }
  return x;
}
```

---

6. * Suppose you are given the following binary tree class definition, which uses the BTNode<T> class of the previous exercise.

```
public class BinaryTree<T> {
  private BTNode<T> root;
  ...
}
```

Add the following methods to this class so that applications can do an inorder traversal one node at a time:

```
// returns the first node that would be visited in an inorder traversal
// null if tree is empty
public BTNode<T> firstInorder() {
   /* COMPLETE THIS METHOD */
}
```

SOLUTION

```
// returns the first node that would be visited in an inorder traversal;
// returns null if tree is empty
public BTNode<T> firstInorder() {
  if (root == null) {
    return null;
  }
  // left most node in tree is the first node in inorder
  BTNode<T> prev=root, ptr=root.left;
  while (ptr != null) {
    prev = ptr;
    ptr = ptr.left;
  }
  return prev;
}
```

and

```
// returns the next node that would be visited in an inorder traversal;
// returns null if there is no next node
public BTNode<T> nextInorder(BTNode<T> currentNode) {
  if (currentNode == null) { // playing defense here
    return null;
  }
  // if there is a right subtree, then right turn, and left all the way to bottom
  if (currentNode.right != null) {
    BTNode<T> ptr=currentNode.right;
    while (ptr.left != null) {
      ptr = ptr.left;
    }
    return ptr;
  }
  // no right subtree
  BTNode<T> p = currentNode.parent;
  while (p != null && p.right == currentNode) {
    currentNode = p;
    p = p.parent;
  }
  return p;
}
```

For instance, an application would call these methods like this to do the inorder traversal:

```
BinaryTree<String> strBT = new BinaryTree<String>();
// insert a bunch of strings into strBST
...
// do inorder traversal, one node at a time
BTNode<String> node = strBST.firstInorder();
while (node != null) {
  node = strBST.nextInorder(node);
}
```

---

7. 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?

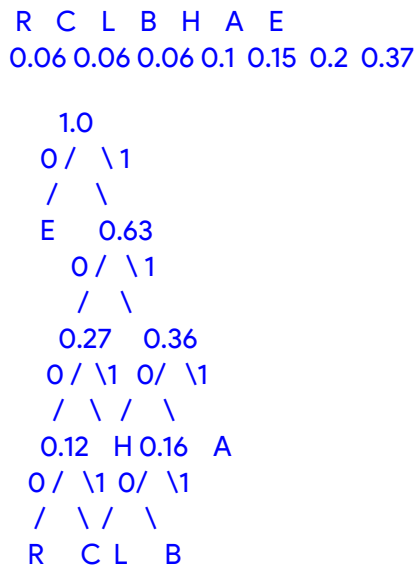
    4. If it takes 7 bits to represent a character without encoding, then for the above text, what is the ratio of the encoded length to the unencoded?

    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 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