Problem Set 7 (due Tuesday, December 5)

1. (12 points) Reconstructing a tree

You are asked to reconstruct the reporting hierarchy of a huge company Disorganized, Inc., based on information that is complete but poorly organized. The information is available in an n-element array, in which each element of the array is a pair $(emp, boss)$ where *emp* is the name of an employee and *boss* is the supervisor of the employee, and n is the number of employees in Disorganized. You may assume that the names of all employees are distinct. For the company CEO, the *boss* entry is empty.

Your task is to compute a tree in which each node has the name of an employee *emp* and a parent field pointing to the node corresponding to the supervisor of *emp* (for the CEO, the parent field will point to NIL).

(a) Design an $O(n \log n)$ time deterministic algorithm for the problem. Justify the running time of your algorithm.

Answer: We create a tree node for each employee with parent pointers initialized to NIL, and create a list of all tree nodes, sorted according to the to the name of the employee. Then, we go through each employee and set its parent pointer to the node corresponding to the supervisor. We can find the supervisor node by doing a binary search on the sorted list.

Sorting takes $O(n \log n)$ time. Finding each parent node takes $O(\log n)$ time, for a total of $O(n \log n)$ time.

(b) Using hashing, design an expected $O(n)$ time randomized algorithm for the problem. Justify the running time of your algorithm.

Answer: Let L denote the list of nodes given (each node containing the name of an employee and their boss). To distinguish these nodes from the nodes of the tree that we will build, we refer to the nodes in the list as *list-nodes* while we refer to the nodes in the tree as *tree-nodes*. A list-node x has two fields $name[x]$ and $parentname[x]$. For a tree-node u, we will maintain three fields: (i) $name[u]$, the name of u, (ii) $parent[u]$, a pointer to the parent of u, and (iii) $children[u]$, a doubly linked list containing pointers to the children of u.

We use hash tables for efficiently searching a node. In order to ensure expected $O(1)$ time for a search, we will choose our hash function from a class of universal hash functions. In order to resolve collisions, we use chaining. Insertion and deletion take $O(1)$ time as we will maintain each chain as a doubly linked list.

In our hash table, we will store the tree-nodes that we create. There are two ways in which we can hash the tree-nodes. One is using the name of their parent. The other is through their own name. If we hash according to the parent name, there are two kinds of collisions that take place. One is when two nodes have the same parent. And the other is due to the hash function. Due to the latter kind of collisions, it is *not the case that after all the nodes* *have been hashed, a given slot contains a list of siblings.* We must keep this in mind when we do the construction of the tree. The best way, perhaps, to avoid this problem is to maintain a chain of linked lists in each slot. Each slot maintains a linked list of pointers, each pointer pointing to a linked list of children associated with a parent that maps to this slot. Suppose we are hashing a node x according to its parent's name. So we check slot $A[h[parentname[x]]]$, where A is the hash table and h is the hash function. Instead of just storing the node x into a chain in this slot, we will obtain a pointer to the linked list that maintains the children of $parentname[x]$ and then insert node x (or an associated entity) into this list. If we do not maintain a chain of linked lists, then either we may not be able to reconstruct a correct tree or we may not ensure expected linear running time.

We now give a detailed description of the algorithm in which we hash according to the given name (rather than the parent's name). This avoids some of the problems discussed in the above paragraph. Here is the algorithm:

```
TREE-CONSTRUCT(L)
1. for each list-node x in L do
2.     create new tree-node u;
3.     name[u] ← name[x];
4.     parent[u] ← nil;
5.     children[u] ← nil;
6.     hi(T, u);  (hash according to name)
7. for each node x in L do
8.     u ←   (T, name[x]);
9.     if parentname[x] ≠ nil then p ←   (T, parentname[x]);
10.    else root ← u;
11.    parent[u] ← p;
12.    insert u into children[p];
13. return root.
```

In the first for-loop, TREE-CONSTRUCT first creates a tree-node for every list-node in L and inserts the tree-node into the hash table T using universal hashing and chaining. In the second for-loop, the algorithm goes through the list-nodes again and (i) sets the parent pointer of each tree-node (except the root) to point to the appropriate tree-node that we search in the hash table, and (ii) inserts the tree-node into the children list of its parent tree-node. Finally, the algorithm returns the root. Clearly, at the end of the procedure, each tree-node u has a pointer to its parent (since step 9 is executed once for the list-node corresponding to u) and each of its children is inserted into its children linked list exactly once (since step 11 is executed exactly once for each of its children). Note that there is no recursion.

For the running-time, we first consider the first for-loop. In each iteration, steps 2 through 6 take $O(1)$ time. In the second for-loop, step 8 takes expected $O(1)$ time. Steps 9 through 11 take $O(1)$ time. Also, since the children field is a doubly linked list, step 12 takes $O(1)$ time. Finally, step 13 takes $O(1)$ time. Since there are exactly n iterations for each for-loop, the expected running time is $O(n)$.

2. (12 points) Ternary Huffman coding

The Huffman code algorithm that we studied in class encoded each character into sequences over the binary alphabet $\{0, 1\}$. Suppose you have a communication mechanism with which you can transmit sequences over a ternary alphabet $\{0, 1, 2\}$. Provide a modified Huffman algorithm for compressing a file with n distinct characters with frequences $f_1, f_2, \ldots, f_n$ using $\{0, 1, 2\}$. Your algorithm should encode each character with a variable-length codeword over the values $0, 1, 2$ such that no codeword is a prefix of another codeword and so as to obtain the maximum compression. Prove that your algorithm is correct.

Answer: Suppose we are given an alphabet of size n. It is tempting to consider the following generalization of Huffman's algorithm. If $n \leq 2$, then have a root r with its children the nodes corresponding to the two characters. If $n \geq 3$, then select the 3 characters with minimum frequency (say a, b, and c) and make these characters siblings with a single parent, introduce a new corresponding to the parent assigning it with the frequency being the sum of the frequencies of a, b, and c, and then recurse.

There is a problem with the above approach, though. Notice that the above algorithm may produce a tree in which every internal node has 3 children except for the root which has 2 children; this is because n will be 2 only at the end of the computation. An optimal ternary code, however, should not have the root with only two children (for $n > 2$). For instance, consider the example with 4 characters a, b, c, and d with frequencies 1, 2, 3, and 4, respectively. The above algorithm will have a, b, and c as siblings and their parent being a sibling of d. The overall cost will be $2 * (1 + 2 + 3) + 4 = 16$. The optimal solution, however is to have a and b as siblings and their parent as sibling with c and d to give a cost of $2 * (1 + 2) + 3 + 4 = 13$.

So how do we generalize Huffman encoding? If we are guaranteed that every internal node has 3 children, then we can apply the above greedy choice. But the preceding condition may not always hold, as we saw in the case $n = 4$. However, we can see that in an optimal tree there can be only one internal node with fewer than 3 children; furthermore, such an internal node should have two children and should be an internal node with largest depth. Why?

First, any internal node u with only child v can be removed and v made the child of the parent of u, hus decreasing the cost of the tree. Now consider the case when there are two internal nodes u and v each having two children. Let u_1 and u_2 be the children of u and v_1 and v_2 be that of v. Without loss of generality, suppose that the depth of u is smaller than the depth of v. We can remove v and make v_1 the child of u and v_2 the child of the parent of v, thus decreasing the cost of the tree, again a contradiction. We leave it as an exercise to the reader to argue that if an internal node does have fewer than 3 children it must be an internal node with largest depth.

We now know that an optimal prefix-free ternary code can be one of two kinds: (i) every internal node has exactly 3 children; or (ii) there exists one internal node with 2 children, all others have 3 children, and the node with 2 children has largest depth. Given a character set of size n, can we determine which of the two types the optimal tree T is? Somewhat surprisingly, yes. Let k be the number of internal nodes in T. If T is of type (i), then the number of edges in the tree, which is $k + n - 1$, also equals $3k$. We thus get $2k = n - 1$. Since k and n are integers, it follows that n is odd. On the other hand, if T is of type (ii), then the number of edges in the tree, which is $k + n - 1$, also equals $3k - 1$. We thus get $n = 2k$. Hence, we have proved that the optimal tree is of type (i) if n is odd; otherwise, it is of type (ii).

We thus have the following algorithm. If n is 1, there is nothing to do. If $n \geq 2$ and odd, then we select the 3 lowest frequency characters, make them siblings, add a parent character with frequency of their sum and repeat. Note that when we repeat, the new value of n is two less than before, implying that the new value is still odd. Similarly, if $n \geq 2$ and even, then we select the 2 lowest frequency characters, make them siblings, add a parent character with frequency of their sum and repeat. Now when we repeat, the new value of n is odd. And from now on, the algorithm will repeatedly select the 3 lowest frequency characters until there is exactly one character left.

Greedy choice: If n is odd, then we select the 3 smallest frequency characters and make them siblings. We have already argued that when n is odd, the optimal tree is of type (i). Given this, we can invoke a swapping argument similar to the binary case (Lemma 16.2) to claim that the 3 smallest frequency characters have to be siblings. When n is even, we know that the optimal tree is of type (ii). Therefore, the internal node with two children needs to be of largest depth. We invoke a swapping argument similar to the binary case to claim that the 2 smallest frequency characters have to be siblings, and their parent must be the internal node with two children.

The remainder of the proof – that once the 3 or 2 smallest frequency characters are combined, the remaining problem is to compute an optimal solution for the instance with the combined characters – is almost identical to that for the binary case the only difference is that we are working with ternary trees and the proof we went through in class is working with binary trees.

3. (12 points) Huffman encoding, entropy, and the English alphabet

(a) Obtain a table from the web giving the frequencies of the letters of the English alphabet.

(b) What is the optimum Huffman encoding of this alphabet?

(c) What is the expected number of bits per letter?

Entropy is a mathematical formulation of the uncertainty and/or the amount of information in a data set. Consider a data set D consisting of n characters, each character independently chosen from a set C according to a specified probability distribution p. That is, for $c \in C$ and $0 \leq i < n$, the probability that the ith character of D is c is $p(c)$. Note that $\sum_{c \in C} p(c) = 1$. The entropy of data set D is then defined to be

$$n \sum_{c \in C} p(c) \log_2(1/p(c)).$$

Intuitively, the entropy measures the *information-theoretic minimum* number of bits needed to represent the data set.

(d) Calculate the entropy of a corpus, that contains English letters in the same frequencies as (a). Do you think this is the limit of how much English text can be compressed? Explain.

(e) **(6 bonus points)** Prove that if all the probabilities are powers of 2 (i.e., for every c there exists an $i \geq 0$ such that $p(c) = 1/2^i$), then the expected number of bits used in the Huffman encoding of D exactly equals its entropy.

Answer (a)(b)(c)(d):

In the following table let $f(c)$ denotes the frequency of an alphabet in english (in percentage). Let $d(c)$ denotes the number of digits in the alphabet's huffman coding. Let $p(c)$ denotes the probability of occurance of an alphabet.

Character(c)	$f(c)$	Huffman Coding	$f(c)$*$d(c)$	$p(c)$	$-p(c) * log_2 p(c)$
E	11.16	101	33.48	0.112	0.353
A	8.50	0101	34.0	0.085	0.302
R	7.58	0100	30.32	0.076	0.282
I	7.54	0111	30.16	0.075	0.281
O	7.16	0110	28.64	0.072	0.272
T	6.95	0001	27.80	0.070	0.267
N	6.65	0011	26.60	0.067	0.260
S	5.74	1001	22.96	0.057	0.237
L	5.49	1111	21.96	0.055	0.230
C	4.54	1101	18.16	0.045	0.203
U	3.63	00000	18.15	0.036	0.174
D	3.38	00001	16.90	0.034	0.165
P	3.17	00100	15.85	0.032	0.158
M	3.01	00101	15.05	0.030	0.152
H	3.00	10001	15.00	0.030	0.152
G	2.47	11101	12.35	0.025	0.132
B	2.07	11001	10.35	0.021	0.116
F	1.81	100001	10.86	0.018	0.105
Y	1.78	100000	10.68	0.018	0.103
W	1.29	111001	07.74	0.013	0.081
K	1.10	111000	06.60	0.011	0.072
V	1.01	110001	06.06	0.010	0.067
X	0.29	11000010	02.32	0.003	0.024
Z	0.27	11000011	02.16	0.003	0.023
J	0.20	11000000	01.60	0.002	0.018
Q	0.20	11000001	01.60	0.002	0.018

The expected number of bits per letter $= \frac{\sum_c f(c)*d(c)}{\sum_c f(c)} = 4.27$

The entropy of an n-letter text file would be $n * \sum_c -p(c) * log_2 p(c) \approx 4.27n$. Per-letter entropy is 4.27, matching Huffman's coding size.

The Huffman coding produced is the limit of how much the english text can be compressed if *we are compressing one character at a time*, as the expected number of bits is almost equal to entropy. Entropy is basically lower bound on the number of bits needed to represent the characters with code words and in our coding we have achieved that.

But, one can achieve much better compression of English text if we do not do character-level compression. We can take significant advantage of commonly occurring words like "the" and also take into account repetitions of phrases that may be specific to the text. Most modern compression schemes, like LZW, take these issues into account and handily outperform Huffman's coding when compressing English text.

4. (12 points) Security of a simplified RSA

Suppose that instead of using $N = pq$ in the RSA cryptosystem, we simply use a prime modulus p.

As in RSA, we would have an encryption exponent e relatively prime to $p-1$ and encryption of a message $m \bmod p$ would be $m^e \bmod p$. Prove that this new cryptosystem is not secure, by giving an efficient algorithm to decrypt: that is, an algorithm that given p, e, and $m^e \bmod p$ as input, computes $m \bmod p$. Justify the correctness of your algorithm and analyze the running time of your decryption algorithm.

Answer: Since e is relatively prime to $p-1$, it follows that there exist integers x and y such that $ex + (p-1)y = 1$. In fact, we can find x and y by the Extended Euclid algorithm. So we have $m = m^{ex} \cdot m^{(p-1)y}$. Taking both sides mod$p$, we get

$$\begin{aligned} m &= m^{ex} \cdot m^{(p-1)y} \bmod p \\ &= ((m^{ex} \bmod p) \cdot (m^{(p-1)y} \bmod p)) \bmod p \\ &= m^{ex} \bmod p \\ &= (m^e \bmod p)^x \bmod p. \end{aligned}$$

(The third equality follows from Fermat's Little Theorem.) Thus, we can compute the message $m \bmod p$ by simply raising m to the power of x, modulus p.

5. (12 points) RSA and digital signatures

A digital signature scheme has two components, `sign` and *verify*. Digital signatures can be implemented using RSA. The `sign` procedure takes a message M and a secret key d corresponding to an RSA triple (N, d, e), then outputs a signature $\sigma = M^d \bmod N$. The *verify* procedure takes a public key (N, e), a signature σ, and a message M, then returns "true" if σ could have been created using a consistent RSA triple.

(a) Signing involves decryption, and is therefore risky. Show that if Bob agrees to sign anything he is asked to, Eve can take advantage of this and decrypt any message sent by Alice to Bob.

Answer: Eve can simply ask Bob to sign every message that Alice sends to him. The signed message, in fact, would be the original text sent by Alice.

(b) Suppose that Bob is more careful and refuses to sign messages if their signatures look suspiciously like text. Describe a way in which Eve can nevertheless still decrypt messages from Alice to Bob, by getting Bob to sign messages whose signatures look random.

Answer: Suppose Alice has sent a message M to Bob by encrypting it as $M^e \bmod N$. Eve selects a number m at random from $\{0, 1, 2, \ldots, N\}$. With high probability m has a multiplicative inverse m^{-1}. Eve gets Bob to sign two messages $(M^e \cdot m) \bmod N$ and $m^{-1} \bmod N$. Since m is chosen at random, both signatures $(M^e \cdot m)^d \bmod N$ and $(m^{-1})^d \bmod N$ are likely to look random. From these two signatures, Eve can obtain $M \bmod N$ by simply multiplying the two signatures since $m \cdot m^{-1} = 1 \bmod N$ and $M^{ed} = M \bmod N$.