CS161 Handout 04
Summer 2001 Monday, July 9th, 2001

Solution Set 1: Recurrences and Proofs

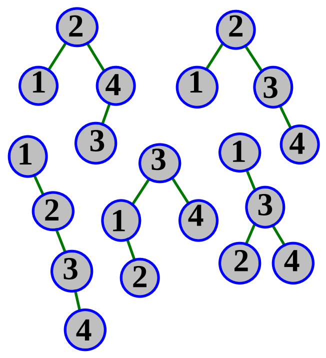
Problem 1

In CS106B or CS106X, you were introduced to a recursive data structure called the **binary search tree**, most likely using a data structure like this:

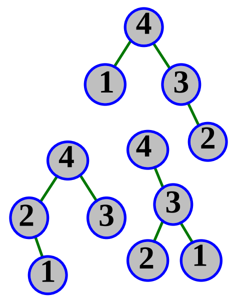
```
typedef struct node {
   int key;
   struct node *left;
   struct node *right;
} node, *tree;
```

The placement of nodes within a binary search tree were subject to the following constraints:

1. The key stored at the root of any subtree must be strictly greater than all keys stored in the left subtree.
2. The key stored at the root of any subtree must be less than or equal to all those keys stored in the right subtree.

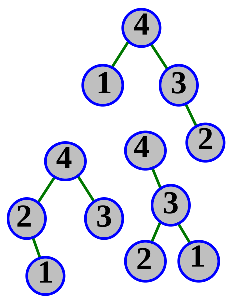


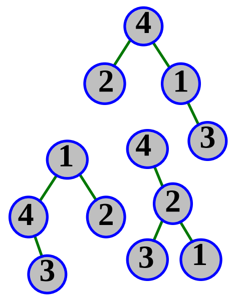
For example, consider all of the binary trees which store the keys 1, 2, 3, and 4. Among the fourteen different legal binary search trees which store these four keys are those five trees drawn on the left. Some examples of binary trees which are not binary **search** trees are drawn on the right.



A related data structure—specifically, the **heap** data structure—is also a binary tree data structure, but instead maintains a **heap** property—the property that the key of every node in the binary tree is greater than or equal to those of its children.

Consider once more those binary trees which store the keys 1, 2, 3, and 4. Drawn on the left are three of the legal heaps storing these keys, and drawn on the right are three binary trees which are **not** heaps.



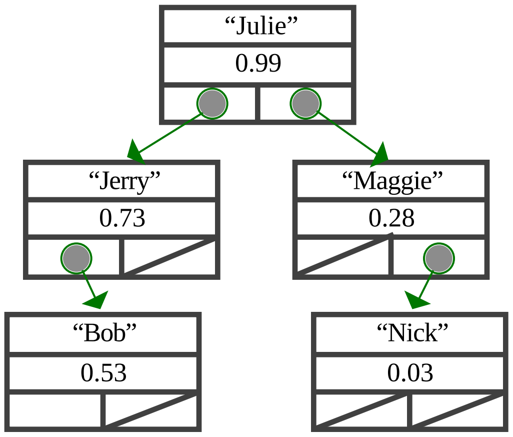


So we have binary search trees, and we have heaps. A **treap** is a binary tree data structure where each node, in addition to storing its key, stores a second value called a **priority**. A treap also maintains the binary search tree property with respect to its keys, while **simultaneously** maintaining the heap property with respect to its priorities. A key and its priority must never be separated; that is, a key and its priority are always stored in the same node. However, the location of the node within the treap depends on its relationship to other nodes.

a) Draw the treap which stores the following sets of keys and priorities:

("Jerry",0.73),("Bob",0.53),("Julie",0.99),("Nick",0.03),("Maggie",0.28)

In this case, the keys are strings and the priorities are real numbers.



Note that the root of the treap is solely determined by the priority field. What particular key happens to be attached to that particular priority dictates how the remaining nodes get divided up into the left and right subtreaps.

b) Consider a collection of n ordered pairs. Assuming that all keys and priorities are distinct, prove that there always exists **exactly one** treap which accommodates the collection of n ordered pairs while meeting the treap constraints.

Scary, but true—this is a strong induction proof. It's clear that a treap with zero nodes is unique, and a treap with one node is unique. Call these our base cases.

Now assume that all treaps accommodating k or fewer nodes exist and are unique. We want to show that a treap accommodating $k+1$ pairs is also unique. We prove this by construction.

Select from among all $k+1$ nodes that node which contains the highest priority. In order to satisfy the treap property, this pair must necessarily be placed at the root of the treap. The key associated with this priority determines which subset of the k remaining nodes need be placed in the left subtreap and the right subtreap. Because the subtreaps are in fact treaps, and are each of size k or less, by the inductive hypothesis, we know that each of those two subtreaps has a unique structure. So, a treap with a unique root, a unique left subtreap, and a unique right subtreap is itself unique—there's no argument as to what it's structure may be.

Therefore, the treap accommodating some collection of n ordered pairs is always unique. Yea! Induction actually came of use for us computer scientists. Neat!

Problem 2: Fibonacci Bases

One of the most important properties of the Fibonacci numbers is the special way in which they can be used to represent integers. Using the notation $j >> k$ to mean that $j \geq k+2$, prove that every positive integer has a unique representation of the form

$$n = F_{k_1} + F_{k_2} + \cdots + F_{k_r}, \quad k_1 >> k_2 >> \cdots >> k_r >> 0 .$$

This statement is most easily proved using strong induction; therefore, let $P(n)$ represent the statement that every positive integer n has a unique representation of the form:

$$n = F_{k_1} + F_{k_2} + \cdots + F_{k_r}, \quad k_1 >> k_2 >> \cdots >> k_r >> 0 .$$

Base case: Since $F_2 = 1$, and because $F_0 = 0$ or $F_1 = 1$ are off limits (remember the constraint that $k_j >> 0$ for all j), then $1 = F_2$ is the only Fibonacci representation that exists. Therefore, $P(1)$ is true and the base case holds.

Induction: Assume that $P(c)$ is true for all positive integers $c < m$, where m is an integer—that is, assume that all integers between and including 1 through $m-1$ can be uniquely expressed as the sum of one or more Fibonacci numbers according to the above constraints. We want to show that $P(m)$ is also true; equivalently, we want to show that m can be uniquely expressed as the sum of one or more Fibonacci numbers.

First, we prove existence. That's easier, because we can find the first Fibonacci number by choosing F_{k_1} to be the largest Fibonacci number less than or equal to m, then choosing F_{k_2} to be the largest that is $\leq m - F_{k_1}$, and so on. More precisely, suppose that $\leq n - F_{k_1}$; then we have that $0 \leq m - F_k < F_{k+1} - F_k = F_{k-1}$. If m is a Fibonacci number, then $P(m)$ holds with $r = 1$ and $k_1 = k$. Otherwise, by the strong induction hypothesis, $m - F_k$ has a (unique) Fibonacci representation $F_{k_2} + F_{k_3} + \cdots + F_{k_r}$, and $P(m)$ still holds then we set $k_1 = k$, because the inequalities $F_{k_2} \leq m - F_k < F_{k-1}$ imply that $k >> k_2$.

Conversely, any representation of the form $P(m)$ implies that $F_{k_1} \leq m < F_{k_1+1}$, because the largest possible value of $F_{k_2} + F_{k_3} + \cdots + F_{k_r}$, when $k >> k_2 >> \cdots >> k_r >> 0$ is:

$$F_{k-2} + F_{k-4} + \cdots + F_{k \bmod 2+2} = F_{k-1} - 1.$$

This formula is itself easy to prove by induction on k; the left-hand side is zero whenever $k = 2$ or $k = 3$. Therefore, k_1 is the greedily chosen value described earlier, and the representation must be unique, so that $P(m)$ holds whenever $P(c)$ for integer $c < m$ holds.

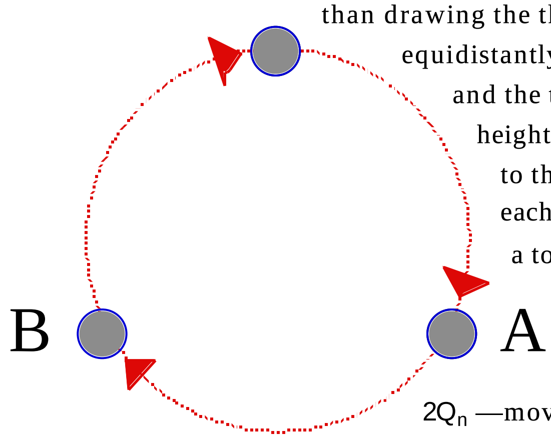
So, by the principle of mathematical induction, $P(n)$ is true for all integers $n \geq 1$. Whew!

Problem 3: Circular Towers

Let Q_n be the minimum number of moves needed to transfer a tower of n disks from A to B, if all moves must be clockwise—that is, from A to B, or from B to the other peg, or from the other peg to A. Also, let R_n be the minimum number of moves needed to go from B back to A under this restriction. Justify why:

$$Q_n = \begin{cases} 0; & \text{if } n = 0; \\ 2R_{n-1} + 1; & \text{if } n > 0; \end{cases} \quad R_n = \begin{cases} 0; & \text{if } n = 0; \\ Q_n + Q_{n-1} + 1; & \text{if } n > 0; \end{cases}$$

You needn't solve these recurrences.



Picture time! I want to emphasize the clockwise constraint of the problem; rather than drawing the three pegs side by side in a line, I place them equidistantly on a circle, the A peg at 4 o'clock, the B peg at 8 o'clock, and the third peg at high noon. Also note: moving a tower of height n from peg A to B is the same as moving a tower from B to the other peg or from the other peg to A. By definition, each requires Q_n individual disk moves. Similarly, moving a tower of height n from B to A is identical to moving any tower of height n from A to the other peg of from the other peg to B, and by definition that requires R_n moves. Note that R_n isn't necessarily equal to $2Q_n$ —moving a tower of height n from B to A doesn't necessarily require that the tower be fully and intermittently transferred to the other peg.

Anyway, to move the tower of height n from A to B, we need to move all but the largest disk to the other peg so that the largest disk can be transferred from A directly to B. Only after we've placed the largest disk where it needs to be should we transfer the tower of height n - 1 from the other peg to B. Moving a tower of height n from A to the other peg requires, by definition, R_{n-1} disk moves. Then we require a single disk move to move the largest disk, and then an additional R_{n-1} disk moves to finish off the job. That justifies the following:

$$Q_n = 0 \qquad n = 0$$
$$Q_n = 2R_{n-1} + 1 \quad n > 0$$

Now, trying to move a tower of height n from B back to A clearly requires that the largest disk be moved twice—once from B to the other peg, and once more from the other peg to A. That means that the tower of height n - 1 has to be moved our of the way two times, and then moved a third time to place it on the destination peg. We must make R_{n-1} disk moves to move the top n-1 disks to peg A, so that the largest disk can be moved to the other peg. Then we must make Q_{n-1} moves to transfer the (n-1)-tower from B back to A, so that the largest disk can be moved from the other peg to A. Finally, we can make R_{n-1} disk moves to move the (n-1)-tower from B to A. The total number of disk moves is then given as:

$$R_n = 0 \qquad n = 0$$
$$R_n = R_{n-1} + 1 + Q_{n-1} + 1 + R_{n-1} \quad n > 0$$

But we know $Q_n = 2R_{n-1} + 1$ from the first half of the solution, so that:

$$R_n = 0 \qquad n = 0$$
$$R_n = Q_n + Q_{n-1} + 1 \quad n > 0$$

There it is.