CS 4800: Algorithms and Data
Due **Thursday, October 18, 2012** at **9:00am**

Homework 3

Submission Instructions: This homework will be submitted online via `git`. For an introduction to `git` by Northeastern's own Eli Barzilay, see `git.racket-lang.org/intro.html`. This includes an introduction to `git`, links to other resources, and recommendations for "best practices".

In order to submit, you will need to make a user account on `github.com`, join the course organization at `github.com/neu-cs4800f12`, and contact me via `cce@ccs.neu.edu` with your team members. I will then create a team and repository for you for the homework. Work within the team under the following guidelines:

1. You may use the `master` branch to collaborate on work within your team, and create any other branches you need.
2. You must push your solution to the `submission` branch before the deadline. You may update this branch as many times as you like up to the deadline.
3. Each program must be an executable file with the appropriate name in the repository's top directory. These programs must run successfully on `login.ccs.neu.edu`.
4. All prose solutions (proofs, math, and other explanations) must be presented in a PDF file named `solution.pdf`.
5. The `solution.pdf` file must also describe where to find the source code for each submitted program, even if the executable itself constitutes the entire source code.
6. Any other files or directories in the submission branch will be ignored.

Note about efficiency and choice of programming language: You *must* document the data structures and built-in functions you use from the language in which you write your solution. You *must* document the running time of every operation you use that is more than $O(1)$. The documented running times must be *both* correct *and* fast enough to ensure that your solution is efficient.

Exercise 1 (8 points)

Problem: For this problem we introduce the *revised* master method: Theorem 4.1 from CLRS, page 94, modified by Exercises 4.6-2 and 4.6-3 on page 106 (which you do not need to prove).

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \lg^k n)$, where $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$. *Note:* When $k = 0$, this is the same as the unrevised clause.
3. If $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. *Note:* The unrevised clause requires that $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$; this is always true when the above condition holds.

Solve the following recurrences using the revised master method, or demonstrate that they cannot be solved using the revised master method.

1. $T(n) = 2T(n/2) + n \lg n$
2. $T(n) = 3T(2n/3) + 1$
3. $T(n) = T(n/4) + 1$
4. $T(n) = 5/4 T(3n/4) + n$
5. $T(n) = 2T(n/4) + \sqrt[3]{n \lg n}$
6. $T(n) = T(n/10) + \sqrt{\frac{n}{\lg^2 n}}$
7. $T(n) = 9/4 T(2n/3) + n^2$
8. $T(n) = 2T(n/2\sqrt{2}) + \sqrt[3]{\frac{n^2}{\lg n}}$

Exercise 2 (15 points)

Problem: Solve the following problems using the graph algorithms presented in class: breadth-first search, depth-first search, topological sort, strongly-connected components, and single-source shorted paths.

1. Create a program that plays *Six Degrees of Kevin Bacon*. Given a set of movie credits, report each actor's degree of separation from Kevin Bacon. Mr. Bacon himself has a degree of 0. Anyone working with him on a movie gets a degree of 1. Anyone working with them gets a degree of 2 (if they do not already have a lower degree), and so on.

   Name your program `six-degrees`. Its input will be a list of movie credits, where each movie credit is a list of names as strings. Its output must be an object with a key for every name credited on any movie, mapped to that person's degree of separation from Kevin Bacon or `null` if no connection to Kevin Bacon can be found.

   Name the algorithm on which you base your solution and state its running time in terms of the movie credits in its input.

   Here is a sample input.

   ```
   [["Kevin Bacon","Kevin Costner"],
    ["Kevin Costner","Kevin Spacey"],
    ["Kevin Spacey,"Kevin Pollak"],
    ["Kevin Smith","Kevin Sorbo"]]
   ```

   The following text is one possible correct output for the preceding input.

```
{"Kevin Bacon":0,
 "Kevin Costner":1,
 "Kevin Spacey":2,
 "Kevin Pollak":3,
 "Kevin Smith":null,
 "Kevin Sorbo":null}
```

2. Let's say we wanted to play the game *Six Degrees of Profitability*, in which each person's connection to Kevin Bacon is measured by the total amount of money made by the movies involved. Assuming we want to find the connection involving the least amount of money, what algorithm would we need to use to find each connection, of those presented in class? State its running time in terms of the movie budgets and credits in its input.

   Extra Credit: (1 point) Either prove that the chosen algorithm solves *Six Degrees of Profitability* for all possible movies, or describe the movies for which it does not work and identify an algorithm from the text that covers all cases.

3. Create a program to form ideal homework groups in a computer science class. In an ideal homework group, each member can trust the competence and academic honesty of each other member. A student A can trust any student B that they know, as well as any other student C that student B trusts. Note that just because student A knows student B does not necessarily mean that student B knows student A.

   Name your program `ideal-groups`. Given an input that maps each student to the list of students that they know, your program must output a list of groups, in which each group is a list of names and each name is represented exactly once. Your program must output as few groups as possible.

   Name the algorithm on which you base your solution and state the running time of your solution in terms of the number of people and "A knows B" acquaintance relationships in the input.

   Here is a sample input.

```
{"Alice":["Carol","Doris"],
 "Betty":["Alice"],
 "Carol":["Betty","Ellie"],
 "Doris":["Ellie"],
 "Ellie":["Doris"],
 "Faith":[]}
```

   Here is one possible correct output for the sample input.

```
[["Alice","Betty","Carol"],
 ["Doris","Ellie"],
 ["Faith"]]
```

4. Which of the algorithms that we have seen could be used to rank each student by "trustworthiness"? Would this be asymptotically faster or slower than constructing ideal homework groups?

Exercise 3 (25 points)

Problem: A *2-3-4 tree* is a left-to-right ordered search tree in which each node has 2, 3, or 4 children and 1, 2, or 3 keys. Leaves have no keys or children; every leaf in a 2-3-4 tree must occur at the same depth. 2-3-4 trees are a special case of *B-trees*, which in brief description are search trees with a varying number of children per node.

Implement *immutable* 2-3-4 trees in which the contents of a tree never change. Perform updates by constructing a new tree with different contents, *sharing* as much of the structure of the old tree as possible. Immutable data structures have the benefit that they are *persistent*: the contents of previous versions of the data structure are always available, regardless of future updates. Furthermore, because of sharing, multiple copies of an immutable data structure can often be maintained using very little allocated space.

Name your program `2-3-4`. Its input must be a list of 2-3-4 tree operations; its output must be the list of results of those operations.

1. The first operation to implement is *insertion*. Your program must start with an empty 2-3-4 tree; insertion adds individual elements to the tree. Insertion operations in the input will be given with the form `{"insert":`*int*`}`, specifying the new integer element. The result of each insertion operation is a 2-3-4 tree matching the following grammar, with keys and children in left-to-right order.

   ```
   tree  =  null
         |  {"two": [tree,int,tree]}
         |  {"three": [tree,int,tree,int,tree]}
         |  {"four": [tree,int,tree,int,tree,int,tree]}
   ```

   The insertion operation starts by finding the appropriate point to insert the new key into the lowest layer of nodes. We then *add* the new key to the node with empty leaves on either side of it.

   We add a key with surrounding subtrees to a node by the following steps. If the node has 1 or 2 keys, simply add the new key. If the node has 3 keys, split the 4 total keys into two nodes with 1 and 2 keys and 1 remaining key between them. Recursively add the remaining key to the parent node with the two new child nodes on either side of it. If the root node of the tree needs to spit, create a new root node with the extra key, using thw two halves of the old root node as the children of the new root.

   Prove that when given a valid 2-3-4 tree as input, insertion produces a valid 2-3-4 tree (ordered, appropriate number of keys/children per node, and all leaves at the same depth). Prove that insertion allocates at most $O(n)$ nodes for an input tree with n keys.

2. The second operation is *deletion*. Deletion operations are provided with the form `{"delete":`*int*`}`, specifying the element to remove. The result is a 2-3-4 tree with the same output form as in insertion.

   Deletion from a 2-3-4 tree proceeds by traversing down the tree to find the key to remove. At every step below the root, we ensure we are recurring into a node with 2 or 3 keys. If the target child node has only 1 key, we first "borrow" keys and children from a sibling. If either sibling of the target node has more than the minimum number of keys and children, we rotate an extra key and child into the target. Otherwise, if both (or the only) sibling of the child has the minimum number of keys, we join the two children together around the key between them. (When the current node is the root, and the root has only 1 key, this can result in deleting the root node.) When we reach the key in question, if we are at a bottom-layer node, we simply remove the key. Otherwise, we swap the key to remove with its immediate successor or predecessor at a leaf and recursively delete that leaf key.

Extra Credit: (2 points) Ensure that your deletion operation works in a single pass down the tree. Specifically, when swapping an element with its successor or predecessor and recursively deleting the swapped leaf key, perform the swap and delete in a single pass down (and back up) the tree rather than performing separate swap and delete operations.

3. The third operation is testing for *membership* of a given key. Membership operations are provided with the form {``member'':*int*}, specifying the element to test. The result of the operation must be `true` or `false` as appropriate.

   Prove that membership requires $O(n)$ comparisons for a tree with n elements. Prove that after an insert or delete operation, the result of membership tests does not change except for the element inserted or deleted.

4. The fourth operation must *build* a 2-3-4 tree containing a given sequence of elements. The input specifying this operation takes the form {``build'': [*int*,...]}, and its output is the newly created tree. This operation ignores and replaces the previous tree.

   Extra Credit: (4 points) Implement your build operation to run in $O(n)$ time when its input is already sorted.

5. The fifth operation is *recovery* of a prior tree. Given input of the form {``recover'':*int*}, the result is the tree that existed prior to the operation n steps ago for the given number n.

Here is an example input.

```
[{"build":[10,20]},
 {"insert":30},
 {"delete":20},
 {"recover":1},
 {"member":20}]
```

And here is one possible correct output for the example input.

```
[{"two":[null,10,null,20,null]},
 {"three":[null,10,null,20,null,30,null]},
 {"two":[null,10,null,30,null]},
 {"three":[null,10,null,20,null,30,null]},
 true]
```

Extra Credit: (10 points) Implement a second version of the program, `2-3-4-mutable`, that implements *mutable* 2-3-4 trees with insert and delete operations. The insert operation must only allocate new nodes when inserting the first key into an empty tree and when splitting a node upon adding a 4th key. The delete operation must execute a single pass down the tree. Specifically, when swapping an internal key with a leaf and then deleting a leaf, the swap and delete must be performed in a single pass downward. Once at the leaf, only $O(1)$ time may be spent accessing ancestor nodes to complete the swap.