*Introduction to Algorithms: 6.006*
Massachusetts Institute of Technology — February 18th, 2010
Professors Piotr Indyk and David Karger — Handout 3

# Problem Set 2

This problem set is divided into two parts: Part A problems are theory questions, and Part B problems are programming tasks.

**Part A questions** are due **Tuesday, March 2nd** at **11:59PM**.

**Part B questions** are due **Thursday, March 4th** at **11:59PM**.

Solutions should be turned in through the course website. Your solution to Part A should be in PDF form using LaTeX or scanned handwritten solutions. Your solution to Part B should be two valid Python files which runs from the command line, together with one PDF file containing your solutions to part (a), (b), (e) and the optional part (f).

Templates for writing up solutions in LaTeX are available on the course website.

Remember, your goal is to communicate. Full credit will be given only to the correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

## Part A: Due Tuesday, March 2nd

1. **(14 points)** Building a Balanced Search Tree from a Sorted List

   You are given a sorted Python list containing $n$ distinct numbers.

   (a) **(4 points)** Show how to construct a binary search tree containing the same numbers. The tree should be roughly balanced (its height should be $O(\log n)$) and the running time of your algorithm should be $O(n)$.

   **Solution:** Let `constructBST(L,i,j)` be the function that takes the list $L$ with starting index $i$ and ending index $j$ and creates its corresponding balanced binary search tree, and returns the root. `constructBST(L,0,n-1)` will select the middle node of $L$ and make it the root of the tree, then it recursively calls `constructBST(L,0,n/2-1)` and `constructBST(L,n/2+1,n-1)` to get the left and right child of the root respectively.

   (b) **(5 points)** Argue that your algorithm returns a tree of height $O(\log n)$. Note: It is probably easier to prove an absolute bound (such as $1 + \log n$ or $2 \log n$) than to use asymptotic notation in the argument.

   **Solution:** For any node $n$ in the tree returned by `constructBST`, the heights of the left and right subtree of $n$ can only differ by 1. The algorithm always selects the middle element to make the root, therefore the number of nodes in the left and right subtree size can at worst differ by 1, and since the same algorithm (which only depends on the size) is called upon both the sublists the heights of the left and right subtree can atmost

differ by 1. Let $T$ be the tree returned by the algorithm and $T_1$ and $T_2$ be the left and right subtree of the root node of $T$, and $h(T)$ denotes the height of the tree $T$. We have
$h(T) = max(h(T_1), h(T_2)) + 1$
$max(h(T_1), h(T_2)) \leq h(T_1) + 1$,
$h(T) \leq h(T_1) + 2$
The algorithm runs for $\log n$ steps as every time the list gets halved, so
$h(T) \leq 2 \log n$.

(c) **(5 points)** Argue that the algorithm runs in $O(n)$ time (here it is hard to avoid the asymptotic notation, so use asymptotic notation).

**Solution:** Since every node in the list $L$ is visited exactly once and there are constant number of operations done per visited node, the algorithm takes $O(n)$ time. We can also write the recurrence equation for the running time
$T(n) = 2 \cdot T(n/2) + \Theta(1)$, which gives us a running time of $\Theta(n)$.

2. **(18 points)** Range Queries for a Balanced Search Tree

We use balanced binary search trees for keeping a directory of MIT students. We assume that the names of students have bounded constant length so that we can compare two different names in $O(1)$ time. Let $n$ denotes the number of students. Say we have a binary search tree with students' last names as the keys with lexicographic dictionary ordering. In this problem we are going to use the balanced search tree to answer some range queries of the students' last names.

For example, if the tree contains 5 names {ABC, ABD, ADA, ADB, ADC}, then all the $5$ names are (inclusively) between ABC and ADC. There are two names, namely ABC and ABD, start with the prefix x=AB.

(a) **(6 points)** Given two strings $a$ and $b$ with $a < b$, give an algorithm that returns all the nodes whose key values are (inclusively) between $a$ and $b$. If the total number of such nodes is $k$, then the running time of your algorithm should be $O(k + \log n)$.

**Solution:** Consider the following procedure for traversing the tree.

```
traverse(tree, (a, b)):
    if tree is empty then
        return
    x := the key at the root of tree
    if (a ≤ x and x ≤ b) then
        traverse(left subtree of tree, (a, x))
        output(x)
        traverse(right subtree of tree, (x, b))
    elseif x > b then
        traverse(left subtree of tree, (a, b))
    else
        traverse(right subtree of tree, (a, b))
```

It suffices to run `traverse`(the entire balanced BST, $(a, b)$) to find all the nodes whose names are between $a$ and $b$. The algorithm has the following properties:

- It finds all nodes whose names are in $[a, b]$, because it includes the rootnode of any tree if the rootnode has its name in $[a, b]$ and it never excludes a subtree that could contain a name in $[a, b]$.
- It only outputs names that are in $[a, b]$.
- The running time is $O(k + \log n)$, because at each depth it visits at most two nodes whose names are not in $[a, b]$ (namely, the visits use for checking the two ends). Therefore, it visits at most $2 \cdot O(\log n) + k$ nodes.

(b) **(6 points)** Give an algorithm that outputs a list of all the nodes whose keys starts with a given prefix $x$ in $O(k + \log n)$ time, where $k$ is the number of nodes in the list.

**Solution:** Consider the following procedure for traversing the tree.

```
traverse(tree, prefix):
    if tree is empty then
        return
    name := the last name at the root of tree
    if prefix is a prefix of name then
        traverse(left subtree of tree, prefix)
        output(name)
        traverse(right subtree of tree, prefix)
    elseif prefix < name then
        traverse(left subtree of tree, prefix)
    else
        traverse(right subtree of tree, prefix)
```

It suffices to run `traverse`(the entire balanced BST, $x$) to find all last names with prefix $x$. The algorithm has the following properties:

- It finds all names with prefix $x$, because it never excludes a subtree that could contain a name with prefix $x$.
- It only outputs names with $x$ as a prefix.
- The running time is $O(k + \log n)$, because at each depth it visits at most two nodes for which $x$ is not a prefix. Therefore, it visits at most $2 \cdot O(\log n) + k$ nodes.

(c) **(6 points)** Give an algorithm to count the number of nodes whose keys start with a given prefix $x$ in $O(\log n)$ time, independent of the number of such nodes. You are allowed to augment the binary search tree nodes.

**Solution:** We can use a similar augmentation of Number of nodes in the left subtree and right subtree which we used for finding the Rank of a node (covered in the recitation). First traverse down the tree to find a node $x_{first}$ which starts with the prefix $x$. Then traverse the left subtree of $x_{first}$ to find the minimum node $x_{min}$(lexicographically) which starts with prefix $x$. Traverse the right subtree of $x_{first}$ to find the largest node

$x_{max}$ (lexicographically) that starts with prefix $x$. $x_{first}$, $x_{min}$ and $x_{max}$ can be found in $O(\log n)$ time. Using the augmentation we can find the ranks of $x_{min}$ and $x_{max}$. The required number of nodes is equal to $\mathsf{Rank}(x_{max}) - \mathsf{Rank}(x_{min}) + 1$.

3. **(18 points)** Collision Resolution

   Assume simple uniform hashing in the entire problem.

   (a) **(6 points)** Consider a hash table with $m$ slots that uses chaining for collision resolution. The table is initially empty. What is the probability that, after four keys are inserted, there is a chain of size $4$?

   **Solution:** For a chain of size $4$, all the four keys should hash to the same value. The first key can hash to any slot in the table. The second, third and the fourth key need to hash to the same slot. The probability of hashing to a particular slot with $m$ slots under simple uniform hashing is $1/m$, so the required probability is $1/m \times 1/m \times 1/m = 1/m^3$.

   (b) **(6 points)** Consider a hash table with $m$ slots that uses open addressing with linear probing. The table is initially empty. A key $k_1$ is inserted into the table, followed by key $k_2$. What is the probability that inserting key $k_3$ requires three probes?

   **Solution:** Inserting $k_3$ in the table would require three probes iff the first two probes hit occupied slots. Since we have two keys in the table $k_1$ and $k_2$, they need to be consecutive and $k_3$ should hash to the one which is above of the two. So we have two cases:

   i. $k_2$ is above of $k_1$ : $k_1$ is free to hash to any slot and $k_2$ should hash to the slot above $k_1$. So the probability for this case is $1/m$.

   ii. $k_2$ is below $k_1$: $k_1$ is free to hash to any slot. Now for $k_2$ to hash to the next slot, it can either hash to the same slot occupied by $k_1$ or it can hash directly to the next slot. So $k_2$ has 2 places to hash to, therefore the probability of this case is $2/m$.

   Now for $k_3$ to require 3 probes for insertion, it should hash to the key above of the two $k_1$ and $k_2$, so the total probability $k_3$ hashing to the above slot is: $(1/m+2/m)\times 1/m = 3/m^2$.

   (c) **(6 points)** Suppose you have a hash table where the load-factor $\alpha$ is related to the number $n$ of elements in the table by the following formula:

$$\alpha = 1 - \frac{1}{\log n}.$$

   If you resolve collisions by open addressing, what is the expected time for an unsuccessful search in terms of $n$?

   **Solution:** According to materials covered in recitation as well as Theorem 11.6 in CLRS (2nd edition), the expected time taken by an unsuccessful search in open

addressing (under the Uniform Hashing assumption) is $T(\alpha) \leq \frac{1}{1-\alpha}$. Plugging in $\alpha(n) = 1 - 1/\log n$, we get

$$T(\alpha) \leq \log n.$$