Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — September 23, 2010
Professors Konstantinos Daskalakis and Patrick Jaillet — Handout 2

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, October 5th** at **11:59PM**.

Part B questions are due **Thursday, October 7th** 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 a valid Python file, together with one PDF file containing your solutions to the two theoretical questions in Part B.

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, October 5th

1. **(15 points)** Building a Balanced Search Tree from a Sorted List

   (a) **(5 points)** Given a list of n numbers, we can build a binary search tree containing these numbers by starting with an empty tree and inserting the numbers from the list one by one into the tree. By employing appropriate rebalancing procedures, e.g. as in AVL-trees, the total time needed to build this tree will be $O(n \log n)$.

   Now assume that the elements in this list of n numbers given to you are already sorted. Show how to construct in $O(n)$ time a binary search tree containing these numbers. The tree should be roughly balanced (its height should be $O(\log n)$)

   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. **(20 points)** Implementing the Runway Reservation System

(a) **(5 points)** Explain how to implement the runway reservation system such that all operations – checking whether a request to land at time t is valid, inserting a landing time into the system, and deleting a landing time from the system – take $O(\log n)$ time, where n is the total number of scheduled landing times in the system before the operation. Remember, a requested time t is valid if there are no scheduled landings within < 3 minutes of t. Give algorithms for the three operations and analyze their running time.

Solution: The runway reservation system should be implemented using a balanced BST, such as an AVL tree, where the nodes in the tree represent reserved landing times. To check whether a requested time t is valid, we can find the location where we would insert it into the AVL, and from that location find the predecessor and successor. If both the predecessor and the successor differ form t by at least 3 minutes, then t is valid; otherwise it is not. Inserting into an AVL tree and finding a successor and predecessor each take $O(\log n)$ time, so the total time is $O(\log n)$. Inserting a node into the tree and deleting a node from the tree can be done using the normal delete and insert methods shown in class (with rotations to maintain the balance). These methods take $O(\log n)$ time in AVL trees.

(b) **(5 points)** Given two times t_1 and t_2 with $t_1 < t_2$, give an algorithm that returns all the scheduled landing times that are (inclusively) between t_1 and t_2. If the total number of such landing times 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, $(t_1, t_2)$):
    if tree is empty then
        return
    x := the key at the root of tree
    if ($t_1 \leq x$ and $x \leq t_2$) then
        traverse(left subtree of tree, $(t_1, x)$)
        output(x)
        traverse(right subtree of tree, $(x, t_2)$)
    elseif $x > t_2$ then
        traverse(left subtree of tree, $(t_1, t_2)$)
    else
        traverse(right subtree of tree, $(t_1, t_2)$)
```

It suffices to run `traverse`(the entire balanced BST, (t_1, t_2)) to find all the nodes whose times are between t_1 and t_2. The algorithm has the following properties:

- It finds all nodes whose times are in $[t_1, t_2]$, because it includes the rootnode of any tree if the rootnode has its time in $[t_1, t_2]$ and it never excludes a subtree that could contain a time in $[t_1, t_2]$.
- It only outputs times that are in $[t_1, t_2]$.
- The running time is $O(k + \log n)$, because at each depth it visits at most two nodes whose times are not in $[t_1, t_2]$ (namely, the visits used for checking the two ends). Therefore, it visits at most $2 \cdot O(\log n) + k$ nodes.

(c) **(5 points)** Given two times t_1 and t_2 with $t_1 < t_2$, give an algorithm to count the number of scheduled landing times that are (inclusively) between t_1 and t_2 in $O(\log n)$ time, independent of the number of such landing times. You are allowed to augment the binary search tree nodes.

Solution: We can augment each node with the number of nodes in its left subtree. First search for t_1, and if it is in the tree set x_{first} to the node with value t_1. If it is not in the tree, find the successor of the location where it would be inserted in the tree, and set x_{first} to be that node. Then, search for t_2, and if it is in the tree set x_{last} to be the node with value t_2, otherwise set it to be the predecessor of the place where t_2 would be inserted. x_{first} and x_{last} can be found in $O(\log n)$ time.

Using the augmentation we can find the ranks of x_{first} and x_{last}. The required number of nodes is equal to $\mathsf{Rank}(x_{first}) - \mathsf{Rank}(x_{last}) + 1$.

For those who did not see Rank in recitation, the algorithm takes a node x as input and returns the number of nodes in the tree whose keys are $\leq key[x]$. Its code can be found in chapter 14.1 of CLRS. The algorithm starts at x and count $size(left[x]) + 1$ and travels up the tree to the root node. Along the way any time that it moves up and left to a node y, it adds $size(left[y]) + 1$ to the count. At the end, it returns the count.

(d) **(5 points)** Given a requested time t which is invalid, give an $O(\log n)$-time algorithm to find the smallest time t_2 such that $t_2 > t$ and t_2 is valid. You are allowed to augment the binary search tree nodes.

Solution: One solution would be to augment each node x with aug[x] = the number of "spaces" in the node's subtree, where a node has a "space" if the difference between its key and the key of its successor is at least 6 – as in there is a space between it and the next node to insert another time. (The maximum node in the tree always has a space.) Then if a request t is invalid, consider the following algorithm:

```
next-valid(tree, $t$):
    $x$ = the successor of the location $t$ would be inserted into if it were valid
    while $x != $ root(tree):
        if aug[$x$]-aug[left[$x$]] $\geq 1$: (only care about spaces in $x$ and right[$x$])
            if aug[$x$]-aug[right[$x$]] = 1 : $y = x$ ($x$ has a space)
            else: $y =$ the smallest node with a space in right[$x$]
            return key[$y$] + 3
        (find the first ancestor such that $x$ is in its left subtree)
        while $x =$ right(p[$x$]):
            x = p[x]
        x = p[x]
```

This algorithm finds the smallest node with key greater than t that has a space, and returns that node's key + 3. Finding the successor takes $O(\log n)$ time. Finding the first ancestor that is reached by moving up and right and that has a space in its right subtree or itself requires travelling up the tree at most once, so $O(\log n)$ time. Finding the smallest node with a space in a subtree rooted at node x can be done by checking whether there are any spaces in left[x], and if so continuing to search in left[x]; if not, checking if x has a space and if so returning it; if not continuing into right[x]. This requires travelling down the subtree at most once, which takes $O(\log n)$ time. The total running time of the algorithm is $O(\log n)$.

3. **(15 points)** Collision Resolution

Assume simple uniform hashing in the entire problem.

(a) **(5 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 exactly 3?

Solution: For the three keys that collide, the probability that they do is $1/m^2$, the probability that the last two hash into the same slot as the first one. The probability that the fourth key does not hash into that slot is $\frac{m-1}{m}$. So the probability of a chain of three for a given three keys is $\frac{m-1}{m^3}$, and there are $\binom{4}{3} = 4$ ways of choosing these three keys out of the four, so the total probability of a chain of exactly three is $4\frac{m-1}{m^3}$.

(b) **(5 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, and then key k_3. What is the probability that the total number of probes while inserting these keys is at least 4?

Solution: Since the insertion of each element requires at least one probe, for there to be 4 probes, there needs to be at least one collision. If the first two keys collide, we do not even need to know where the third key hashes to – there is already a collision. The probability that the first two keys collide is $1/m$. If the first two keys do not collide, then the third key needs to hash to one of the two slots that the first two keys hash to in order for there to be a collision. The probability of this happening is the product of the probability that the first two keys do not collide and the probability that the third key goes into one of the two slots that the first two keys hash to: $\frac{m-1}{m} \cdot \frac{2}{m} = \frac{2(m-1)}{m^2}$. The total probability of having at least 4 probes is $1/m + \frac{2(m-1)}{m^2} = \frac{3m-2}{m^2}$.

(c) **(5 points)** Suppose you have a hash table where the load-factor α is related to the number n of elements in the table by the following formula:

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

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 $\alpha(n) = 1 - 1/\sqrt{n}$, we get

$$T(\alpha) \leq \frac{1}{\frac{1}{\sqrt{n}}} = \sqrt{n}.$$

Part B: Due Thursday, October 7th

(50 points) Remote Error Correction

Prof. Daskalakis went to Mordor to give a talk. He was planning to work on Problem Set 3 (PS3) for 6.006 during his visit. Unfortunately, his computer broke and corrupted the latest copy of PS3. A small fraction of the lines of the file was affected. Prof. Daskalakis could just download the latest copy of the file, but Mordor (The Land of Shady ISPs) is infamous for slow and pricey Internet access (2 magic rings per each transferred and received byte).

Help Prof. Daskalakis [redacted] recover the file and prepare PS3 on time! Design an interactive protocol for detecting and correcting corrupted characters that uses little communication. The file `file_transfer_skeleton.py` has three unimplemented functions. Your goal in this problem is to implement them. (Feel free to add helper methods.)

1. First, Prof. Daskalakis and Prof. Jaillet, who is in Cambridge, need a good hash function which would compute a hash value for any contiguous subset of lines of a file.

   (5 points) Implement a division hash in the function `hash_function`, which returns the hash value of a string.

   (20 points) The function `compute_node_values` should use your function `hash_function` to precompute all useful hash values for the old file and the new file. The hash values should be kept in `HashOldFile` and `HashNewFile`, respectively. In order to compute these values efficiently you should use the idea behind a rolling hash, i.e. instead of computing each value from scratch, exploit the fact that if you know the hash values of two blocks of lines, then with a constant number of operations you can compute the hash of the concatenation of these two blocks of lines.

2. **(4 points)** Suppose the two copies of the file differ in some number of consecutive lines. Let n be the total number of lines in the file. How can Prof. Daskalakis and Prof. Jaillet use binary search to find the start and end of the corruption by exchanging only $O(\log n)$ hash values?

   (4 points) Let W be the number of corrupted lines (not necessarily consecutive). What can they do to find all corrupted lines by exchanging only $O(W \cdot \log n)$ hash values?

   (17 points) The function `binary_check_hash` should simulate the protocol. It should call the function `compare` to compare two hash values, and the function `send_words` whenever a line is corrupted and has to be retrieved from the server.