Sample Solution for CS3323 Fall 2006 Assignment 5 **(58 marks)**

Due Monday, Nov. 27, by 5pm.

- Assignments should be handed in by placing them in the CS3323 bin on E level of Gillin Hall.

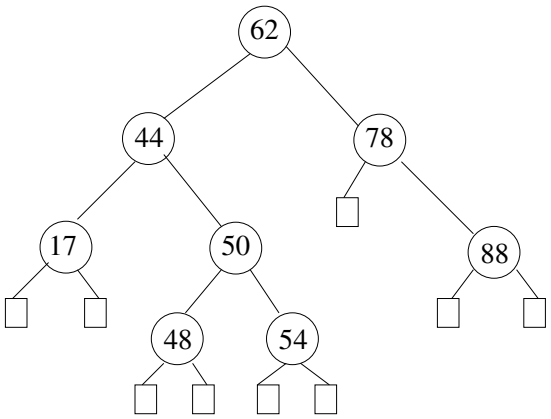

Figure 1: AVL tree for Question 1.

1.

(a) **(5 marks)** Draw the AVL tree resulting from the insertion of an item with a key 52 into the AVL tree of Figure 1 (show the tree after each rotation, if any).

(b) **(5 marks)** Draw the AVL tree resulting from the removal of the item with key 62 from the AVL tree of Figure 1 (show the tree after each rotation, if any).

(c) **(8 marks)** Draw an example of an AVL tree such that a single remove operation causes rotations to propagate all the way to the root. (Use triangles to represent subtreees that are not affected by this operations.) The height of your example tree should be at least 5. Show the tree after each rotation.

solution: In a generic sense, we are looking for a certain type of subtree (of arbitrary height) for which removing an element causes the entire subtree to decrease in height. Consider the tree rooted at z:

If T were to decrease in height by one (to k), that would cause this whole (sub)tree to become unbalanced, causing a rotation and resulting in the following post-rotation tree:

Note that before the rotation, the tree was of height $k + 3$, but now it is $k + 2$: it has shrunk by one level. Now, in a bit of handwaving, we blithely said that T would decrease in height by one; with that we constructed a larger tree which would also decrease in height by one (after a rotation). The base case of this little induction is where $k = 0$ and T is just a single node which is deleted; from which we can construct any number of larger trees by performing the construction with T equal to a smaller tree with this property.

So here is a concrete example, which is (not coincidentally) also the most correct homework answer: an AVL tree of height 5 which, upon the deletion of one node, causes two rotations. (The externals are omitted because that would just get so messy.)

The leftmost unmarked node is by itself the subtree T in the first iteration of the construction (T_1); the other subtrees U, V, W are null here. This is a textbook example of an AVL rotation, and clearly when this node is removed, the height of the tree rooted at z_1 will shrink by one. So if we consider the subtree rooted at z_1 to be itself the subtree T_2, for the second iteration, we can construct around it a larger tree which would have to undergo further rotation.

Note that I called that the "most correct" answer to the problem; this is because, if I wanted to, I could plug this tree into yet another iteration of the algorithm, and come out with a height-7 tree which would require three rotations. Ad infinitum. A slightly less correct solution would be if y_2's left subtree (a.k.a. U_2) were one level higher; then the tree would require two rotations, but wouldn't really be a general example. (No points were deducted for this, however.)

There are, of course, a few valid variations; for example, z, y, and x can be related in any of the four ways leading up to a rotation (either single or double). Instead of the anonymous 3-node subtrees we used for U_2, V_2, and W_2, we could have used 2-node subtrees, and all the conditions would still hold. And, of course, we could make our base case with $k_1 = 1$ instead of $k_1 = 0$, which wouldn't have affected the analysis, but would have made the diagrams a lot messier. (It also would have made the tree of height 6.) Speaking of messy diagrams: the problem did allow for answers which specified irrelevant subtrees with triangles; thus, we could reasonably have substituted triangles of height 2 for the three anonymous 3-node subtrees, but the tree is small enough that it's also not unreasonable to just draw them out.

Analysis: this construction generates trees such that the number of required rotations $r = \frac{h-1}{2}$ (assuming $k_1 = 0$). The height h of an AVL tree is always $\Theta(\log n)$. Thus this tree (and all generated like it) will require $\Theta(\log n)$ rotations.

2. In this problem, assume that letter A is equivalent to 0. The subscripts do not affect the value of the keys (which are letters).

(a) **(5 marks)** Give the contents of the hash table that results when keys E_1 A S_1 Y Q U E_2 S_2 T I O N are inserted in that order into an initially empty 13-item hash table using linear probing (use $h(k) = k \bmod 13$ for the hash function for the k-th letter of the alphabeth).

Solution: In linear probing, we place an element into $h(k)$th slot or the first slot available after it (if $H[h(k)]$ is already full).

First, a small helper table for the English alphabeth:

```
Key    0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 Letter A B C D E F G H I J  K  L  M  N  O  P  Q  R  S  T  U  V
W  X  Y  Z mod 13 0 1 2 3 4 5 6 7 8 9 10 11 12  0  1  2  3  4  5  6
7  8  9 10 11 12
```

Probes listing:

```
E  A  S  Y  Q  U  E  S  T  I  O  N 4  0  5 11  3  7  4  5  6  8  1
0
                  5  6  7  9     1
                  6  7  8 10     2
                     8  9
```

So the hashtable ends up being:

0	1	2	3	4	5	6	7	8	9	10	11	12
A	O	N	Q	E_1	S_1	E_2	U	S_2	T	I	Y	

(b) **(5 marks)** Give the contents of the hash table that results when keys E_1 A S_1 Y Q U E_2 S_2 T I O N are inserted in that order into an initially empty 13-item hash table using double hasing (use $h(k) = k \bmod 13$ for the hash function for the k-th letter of the alphabeth, and $h'(k) = 1 + (k \bmod 11)$ for secondary hashing function).

Solution: As per section 7.3.5 of the book, if h maps some key k to a bucket A[i] with $i = h(k)$, that is already occupied, then we iteratively try the buckets A[$(i + j \cdot h'(k)) \bmod 13$] next, for $j = 1, 2, 3, \ldots$.

```
E  A  S  Y  Q  U  E  S  T  I  O  N 4  0  5 11  3  7  4  5  6  8  1
0
                  0  0     4  5  3
                  9  8     0  9  6
                           9  0  9
                           5  4 12
                           1  8  2
                             12
```

0	1	2	3	4	5	6	7	8	9	10	11	12
A	I	N	Q	E_1	S_1	T	U	S_2	E_2		Y	O

3. (a) **(5 marks)** What is the worst-case running time for inserting n items into an initially empty hash table, where collisions are resolved by chaining? What if each sequence is stored in sorted order? Assume that a hash table is an array of link-based sequences.

   Solution: The worst-case running time in the first case is $O(n)$, since for every item, we do two things, both of which take constant amount of work: compute the hash function h of its key ($O(1)$ time) and insert this item in front of the $h(k)$th sequence (this take $O(1)$ time in a link-based sequence).

   If each sequence (chain) is stored in sorted order, then insertion in such a chain could potentially take as much as $O(\underline{sizeofsequence})$ time. The worst-case happens when we try to insert n keys into a hashtable in sorted order and all of these keys are mapped to the same i by the hash function. In this case, we would have to make n increasing-key insertions into a initially empty ordered link-based sequence. I.e., for each key we have to scan down the sequence to find the item's proper place (last) which would take time $O(j)$ for jth insertion. The total time is then $O(1 + 2 + \ldots + n) = O(n^2)$.

   (b) **(5 marks)** How many probes are involved when double hashing is used to build a table consisting of n equal keys? Consider each successful or unsuccessful attempt to place an element in a hash to be a single probe.

   Solution: Double hashing hashes key k according to the index $H(k, j) = (h(k) + j \cdot h'(k)) \bmod N$ where $j = 0, 1, 2, \ldots$ and N is the size of the hash table. Since we are concerned with lots of equal keys in this problem, let us see what kind of js would it take to make $H(k, j_1) = H(k, j_2)$ (let's drop the first argument for now, as all the keys in this problem are equal anyway).

$$(h(k) + j_1 \cdot h'(k)) \bmod N = (h(k) + j_2 \cdot h'(k)) \bmod N \Longrightarrow (j_1 - j_2) \cdot h'(k) \bmod N = 0$$

This means that either $h'(k)$ is divisible by N or $(j_1 - j_2)$ is. The former is impossible (or at least, it does not make any sense, because then $H(k, j) = h(k) \bmod N$ is independent of j and our probing mechanism is permanently stuck trying to place an item in the same slot. So $(j_1 - j_2)$ is our only other option, i.e. double-hashing on the same key would require at least N probes before it hashes a key into the same index as before.

Note that n is bound to be less then N, because otherwise the table is just too small for the input set anyway. So if all n items have the same key, the $i$the item being inserted is guaranteed to have $i-1$ unsuccessful probes and then successful ith probe (i.e. i probes altogether). So the total number of probes for all elements is going to be $1 + 2 + \ldots + n = n * (n + 1)/2$.

4. (a) **(5 marks)** Draw the merge tree for an execution of the merge-sort algorithm on on the following input sequence: $(2, 5, 16, 4, 10, 23, 39, 18, 26, 15)$.

   (b) **(5 marks)** Draw the quick-sort tree for an execution of the quick-sort algorithm on the input sequence from problem 1(a) (like in Figure 6.8).

   Suppose we modify the deterministic version of the quick-sort algorithm so that, instead of selecting the last element in an n-element sequence as the pivot, we choose the element at rank $\lfloor n/2 \rfloor$.

   (c) **(5 marks)** Draw the quick-sort tree for an execution of this modified quick-sort algorithm on the input sequence from problem 1(a).

   (d) **(5 marks)** What is the running time of this version of quick-sort on a sequence that is already sorted?

   Solution: In a sorted sequence, this version of quicksort would always split the sequence into two almost equal sorted pieces since the pivot is always smack in the middle of it. Let $T(n)$ be the worst-case running time of this quicksort algorithm on a sequence of size n. Then

$$T_q(n) = T_q(\lfloor \frac{n}{2} \rfloor) + T_q(\lfloor \frac{n}{2} \rfloor) + cn = 2T_q(\lfloor \frac{n}{2} \rfloor) + cn$$

   Note that this is almost exactly the same equation as we have for the standart analysis of merge-sort in section 6.1.3, except merge-sort has running time $T_m(n) = T_m(\lceil n/2 \rceil) + T_m(\lfloor n/2 \rfloor) + cn$. Obviously $T_q(n) = O(T_m(n))$ given the same constant c and the same running time given $n \leq 1$. Since $T_m(n) = O(n \log n)$, $T_q(n) = O(O(n \log n)) = O(n \log n)$.

   There are other ways to analyze this, like considering the height of the quicksort tree. This tree is as tall as the number of times we can halve and then take floor

of n before hitting 1. This is trivially smaller then the number of times we can halve n before hitting 1 (i.e. the smallest number k such that $n/2^k \leq 1$) which is $\lceil \log n \rceil$. So now we have a tree of height $O(\log(n))$ where at each level of the tree we process each element of the sequence at most once. Therefore the running time of modified quick-sort on a sorted sequence of size n is $O(n \log n)$.