Problem Set 5 Solutions

Reading: Chapters §18.1-18.2, 14.1-14.3, 33.1-33.3, Skip Lists Handout

There are **four** problems. Each problem is to be done on a **separate sheet** (or sheets) of paper. Mark the top of each sheet with your name, the course number, the problem number, your recitation section, the date, and the names of any students with whom you collaborated. As on previous assignments, "give an algorithm" entails providing a description, proof, and runtime analysis.

Problem 5-1. Joining and Splitting 2-3-4 Trees

The JOIN operator takes as input two 2-3-4 trees, T_1 and T_2, and an element x such that for any $y_1 \in T_1$ and $y_2 \in T_2$, we have $key[y_1] < key[x] < key[y_2]$. As output JOIN returns a 2-3-4 tree T containing the node x and all the elements of T_1 and T_2.

The SPLIT operator is like an "inverse" JOIN: given a 2-3-4 tree T and an element $x \in T$, SPLIT creates a tree T_1 consisting of all elements in $T - \{x\}$ whose keys are less than $key[x]$, and a tree T_2 consisting of all elements in $T - \{x\}$ whose keys are greater than $key[x]$.

In this problem, we will efficiently implement JOIN and SPLIT. For convenience, you may assume that all elements have unique keys.

(a) Suppose that in every node x of the 2-3-4 tree there is a new field $height[x]$ that stores the height of the subtree rooted at x. Show how to modify INSERT and DELETE to maintain the $height$ of each node while still running in $O(\log n)$ time. Remember that all leaves in a 2-3-4 tree have the same depth.

Solution: Let leaf nodes have a $height$ of 1 and internal nodes have $height[x] = 1 + height[child(x)]$. A node affected by INSERT or DELETE operations will simply recalculate their $height$ value by looking at the $height$ of their children. In both INSERT and DELETE, at most $O(\log n)$ nodes positions will be affected and each of their $height$ values can be updated in $O(1)$ time. Therefore, the added calculation cost of maintaining $height$ fields is $O(\log n)$.

A slight caveat in using this method is that we must ensure that heights are calculated from the bottom-up, otherwise there could be a case where a parent computes its height from an out of date child. Fortunately, both INSERT and DELETE recursively work from the bottom of the tree upward, so this is not an issue.

(b) Using part (a), give an $O(1 + |h_1 - h_2|)$-time JOIN algorithm, where h_1 and h_2 are the heights of the two input 2-3-4 trees.

Solution: Find the heights of T_1 and T_2. If $h_1 > h_2$, find the node z with depth $h_1 - h_2$ on the rightmost path of T_1 and insert x into z. If z is full, it will be split with a key floating up as in INSERT. Set the rightmost child of the node containing x to be the root of T_2. Now every leaf in the resulting 2-3-4 Tree has depth h_1, and the branching constraint is obeyed. The case for $h_1 < h_2$ is similar.

If $h_1 = h_2$, merge the two root nodes along with x into a "fat" node, and split the node if it is overloaded.

It takes $O(1+|h_1-h_2|)$ time to find the node z and insert x into z, and $O(1)$ time to join the smaller tree to the larger tree. Therefore the total running time is $O(1+|h_1-h_2|)$.

(c) Give an $O(\log n)$-time SPLIT algorithm. Your algorithm will take a 2-3-4 tree T and key k as input. To write your SPLIT algorithm, you should take advantage of the search path from T's root to the node that would contain k. This path will consist of a set of keys $\{k_1, \ldots, k_m\}$. Consider the left and right subtrees of each key k_i and their relationship to k. You may use your JOIN procedure from part (b) in your solution.

Solution:

1. Initialize two empty trees T_1 and T_2.
2. Search for the element k in the tree T.
3. If the search path at node k_i traverses right, INSERT k_i into T_1 and JOIN k_i's left subtrees with T_1.
4. If the search path at node k_i traverses left, INSERT k_i into T_2 and JOIN k_i's right subtrees with T_2.
5. If k is found JOIN k's left child with T_1 and its right child with T_2.
6. If a leaf node is encountered, insert any remaining elements into their appropriate tree.

Let k_l be some key less than k. Then k_l will either: (1) be a node which the search path turned right on, (2) be less than some node that the search path turned right on, or (3) be a left child of k. In all three cases, k_l will be INSERTed or JOINed into T_1. Similarly, any nodes greater than k will be placed in T_2.

The algorithm joins the subtrees as it walks down the 2-3-4 tree along the search path. Therefore the height of subtrees never increases. In other words, we have $height[T_{i-1}] \geq height[T_i]$. Searching for k takes $O(\log n)$ time. Let h_i denote the height of subtree T_i. The running time for the iterative JOIN takes:

$$\begin{aligned} O(\sum_{i=1}^{m}(1+|h_{i-1}-h_i|)) &= O(\sum_{i=1}^{m}(1+h_{i-1}-h_i)) \\ &= O(m+h_0-h_m) \\ &= O(m+\log n). \end{aligned}$$

Since a 2-3-4 tree has at most 4 branches, the algorithm can join at most 3 subtrees before the search path goes down 1 level in the 2-3-4 tree. Therefore, the number of subtrees m joined is at most 3 times the depth of the key k. Therefore $m = O(\log n)$ and the time complexity of the SPLIT operation is $O(\log n)$.

Problem 5-2. AVL Trees

An ***AVL tree*** is a binary search tree that is ***height balanced***: for each node x, the heights of the left and right subtrees of x differ by at most 1. Height is defined to be the length of the longest path from a node to any leaf in the tree rooted at that node. To implement an AVL tree, we maintain an extra field in each node: $h[x]$ is the height of node x. As for any other binary search tree T, we assume that *root*$[T]$ points to the root node.

(a) Prove that an AVL tree with n nodes has height $O(\log n)$. (*Hint:* Prove that in an AVL tree of height h, there are at least F_h nodes, where F_h is the hth Fibonacci number.)

Solution: Let $T(h)$ be the minimum number of nodes in a height balanced tree of height h. We proceed by induction. For the base cases note that $T(1) \geq T(0) \geq 1$, thus $T(1) \geq F_1$ and $T(0) \geq F_0$. Now assume that $T(h') \geq F_{h'}$ for all $h' < h$.

The root node in an AVL-tree of height h will have two children: one with height $h-1$, and the other with height at least $h-2$. The minimum number of nodes in an AVL-tree of height h can therefore be bounded in terms of $T(h-1)$ and $T(h-2)$, i.e. we have $T(h) \geq T(h-1) + T(h-2)$. By induction hypothesis, this implies $T(h) \geq F_{h-1} + F_{h-2} = F_h$.

From this and the fact that $F_h \geq 1.6^h$, it follows that an AVL tree with n nodes and height h must satisfy $n \geq 1.6^h$ i.e. $h = O(\log n)$.

(b) To insert into an AVL tree, a node is first placed in the appropriate place in binary search tree order. After this insertion, the tree may no longer be height balanced. Specifically, the heights of the left and right children of some node may differ by 2. Describe a procedure BALANCE(x), which takes a subtree rooted at x whose left and right children are height balanced and have heights that differ by at most 2, i.e., $|h[\textit{right}[x]] - h[\textit{left}[x]]| \leq 2$, and alters the subtree rooted at x to be height balanced. (*Hint:* Use rotations.)

Solution: After an insertion into an AVL tree, two generic situations may arise that lead to an unbalanced tree. Without loss of generality, these figures show only the situations when the right subtree of node A has larger height than the left subtree. The large rectangles α, β, γ, and δ represent subtrees having the heights shown. Figures 1 and 2 illustrate how to balance the tree using 1 or 2 locations.

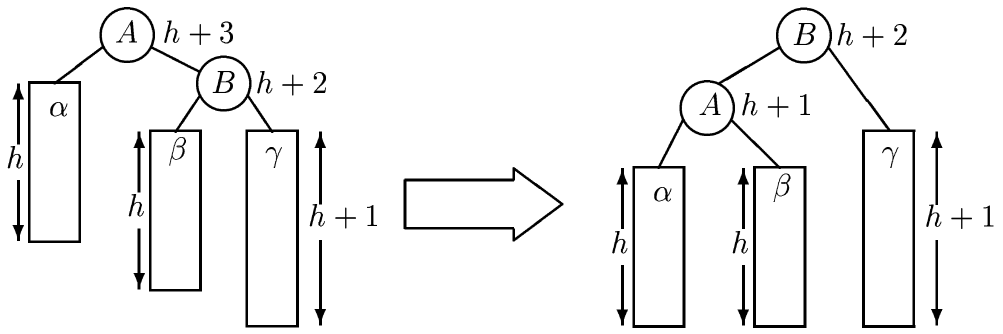

Figure 1: Unbalanced AVL tree – Case 1

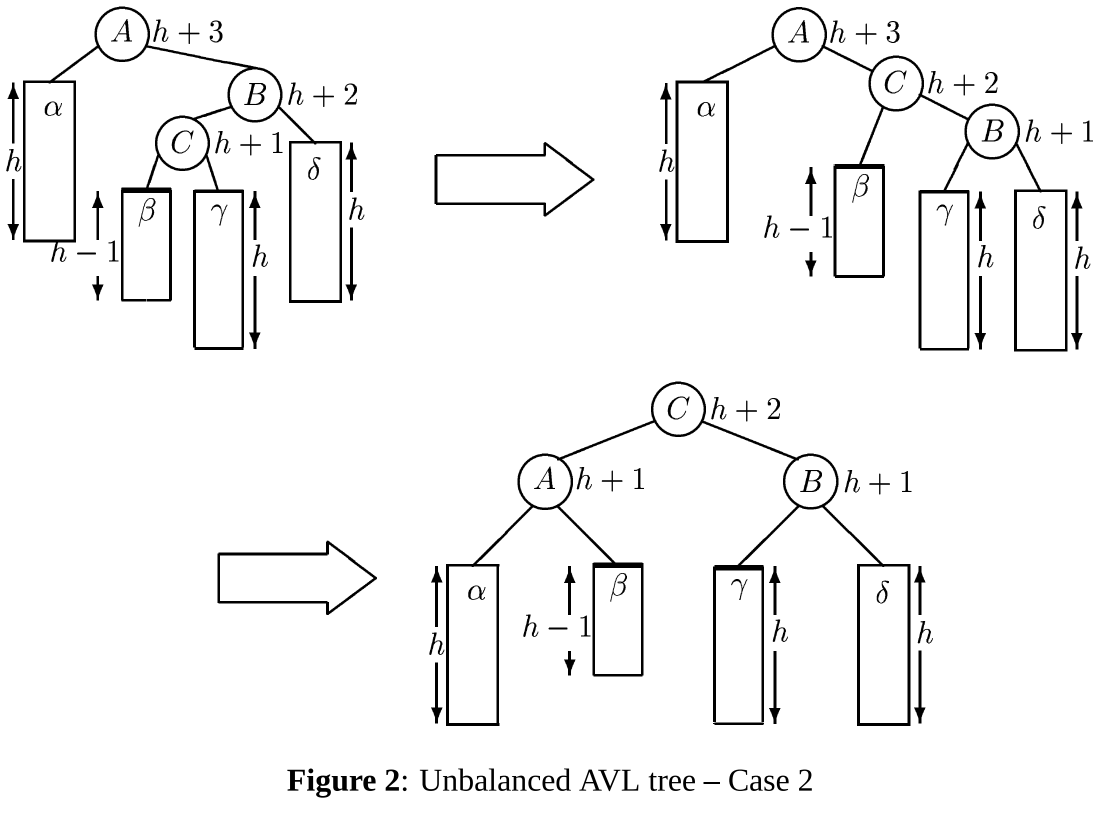

Figure 2: Unbalanced AVL tree – Case 2

Here is pseudo-code for BALANCE. The procedures RIGHT-ROTATE and LEFT-ROTATE perform the corresponding tree rotations described in Section 13.2 and return the root of the subtree after the rotation.

```
BALANCE$(x)$
 1 if $|height(left[x]) - height(right[x])| \le 1$
 2    then return $x$
 3 elseif $height(left[x]) > height(right[x])$
 4    then $y \leftarrow left[x]$
 5          if $height(left[y]) > height(right[y])$
 6            then LEFT-ROTATE$(y)$
 7          return RIGHT-ROTATE$(x)$
 8    else  $y \leftarrow right[x]$
 9          if $height(left[y]) > height(right[y])$
10            then RIGHT-ROTATE$(y)$
11          return LEFT-ROTATE$(x)$
```

(c) Using part (b), describe a recursive procedure AVL-INSERT(x, z), which takes a node x within an AVL tree and a newly created node z (whose key has already been filled in), and adds z to the subtree rooted at x, maintaining the property that x is the root of an AVL tree. As in TREE-INSERT from Section 12.3 in CLRS, assume that *key*$[z]$ has already been filled in and that *left*$[z] =$ NIL and *right*$[z] =$ NIL; also assume that $h[z] = 0$. Thus, to insert the node z into the AVL tree T, we call AVL-INSERT(*root*$[T], z)$.

Solution: The pseudo-code for INSERT is as follows. The idea is to recursively call INSERT on the proper subtree, and then call BALANCE to maintain the balance.

```
INSERT$(x, z)$
 1 if $x = nil$
 2    then $height[z] \leftarrow 0$
 3          return $z$
 4 if $key[z] \le key[x]$
 5    then $y \leftarrow$ INSERT$(left[x], z)$
 6          $left[x] \leftarrow y$
 7          $parent[y] \leftarrow x$
 8          $height[x] \leftarrow height[y] + 1$
 9    else  $y \leftarrow$ INSERT$(right[x], z)$
10          $right[x] \leftarrow y$
11          $parent[y] \leftarrow x$
12          $height[x] \leftarrow height[y] + 1$
13 $x \leftarrow$ BALANCE$(x)$
14 return $x$
```

(d) Show that AVL-INSERT, run on an n-node AVL tree, takes $O(\log n)$ time and performs $O(1)$ rotations.

Solution: The height of the AVL tree is $O(\log n)$. Therefore the insertion and update of the height field take $O(\log n)$ time. In addition, the BALANCE operation in part b) decreases the height of the originally unbalanced tree by 1 after the rotation. Therefore, it will not cause any proporgation of rotations to the rest of the tree. Therefore, AVL-INSERT takes $O(\log n)$ time to insert the node, and performs $O(1)$ rotations.

Problem 5-3. Order Statistics in Skip Lists

In this problem we implement the order statistics operations RANK and SEARCH-BY-RANK in a skip list. For a node x in a skip list L, RANK(x, L) gives the rank of x among the elements in the list. SEARCH-BY-RANK(k, L) is the inverse of RANK. It returns the k-th element in the skip list L. If no such node exists, it returns **nil**.

In this problem we assume that all elements in the skip list have distinct key values. Denote the top level of the skip list as level 1. You may assume a skip list L has a variable $L.depth$ that stores the number of levels in L.

(a) Show how you would augment the skip list data structure so that RANK and SEARCH-BY-RANK can be implemented with $O(\log n)$ time complexity.

Solution: Add an attribute $span[x, i]$ for each node x at each level i, which indicates the number of elements "spanned" by the pointer at x to the next element at level i. Specifically, if the rank of x is r_x and the element y following x at level i has rank r_y, then $span[x, i]$ is $r_y - r_x$. If x is the end of the list, then $span[x, i]$ is 0.

(b) Modify SEARCH, INSERT and DELETE so that the operations run in $O(\log n)$ time with the augmented data structure.

Solution:

Search: No change is needed.

Insert: Insert the element as before. In addition, for each element $p_i = prev[x, i]$, calculate its rank r_i, which can be obtained by summing up the total span traversed up to p_i. For each level i, we update the values of $span[x, i]$ and $span[p_i, i]$, by the following formula in order:

$$\begin{aligned} span[x, i] &\leftarrow span[p_i, i] - (r_1 - r_i) \\ span[p_i, i] &\leftarrow r_1 - r_i + 1 \end{aligned}$$

The expected running time is the same as the original INSERT, which is $O(\log n)$.

Delete: Delete the element as before. For each previous element $p_i = prev[x, i]$ of x at each level i, we update the values $span[p_i, i]$, by the following formula:

$$span[p_i, i] \leftarrow span[p_i, i] + span[x, i] + 1$$

The expected running time is the same as the original DELETE, which is $O(\log n)$.

(c) Give $O(\log n)$ implementations for RANK and SEARCH-BY-RANK.

Solution:

```
RANK($x, L$):
1  $y \leftarrow L$
2  $r \leftarrow 0$
3  $level \leftarrow 1$
4  while $level \leq L.depth$
5     while $key[next[y, level]] \leq key[x]$
6           $r \leftarrow r + span[y, level]$
7           $y \leftarrow next[y, level]$
8     if $y = x$ return $r$
9     else $level \leftarrow level + 1$
10 return null
```

In the procedure RANK, the loop invariant is $r = rank(y)$, which holds just before entering the loop. Inside the loop, whenever r gets increased, the pointer y jumps ahead to the element with exactly the same distance. Therefore the invariant is preserved. If x is in the skip list, the procedure returns r when $y = x$, so it returns the rank of x. Otherwise, the procedure cannot find y and returns null. The running time is the same as SEARCH, which is $O(\log n)$ expected time.

```
SEARCH-BY-RANK($k, l$):
1  $y \leftarrow L$
2  $r \leftarrow k$
3  $level \leftarrow 1$
4  while $r > 0$
5     if $level > L.depth$ return null
6     if $span[y, level] < r$
7           $r \leftarrow r - span[y, level]$
8           $y \leftarrow next[y, level]$
9     else $level \leftarrow level + 1$
10 return $y$
```

The procedure SEARCH-BY-RANK search for the element and keeps track of total span sum during the process. The invariant is $r + rank(y) = k$ and $r \geq 0$. The invariant is preserved in the loop because whenever r decreased, the pointer y jumps forward by the same amount. The program exits the loop when $r = 0$, so it follows that $rank(y) = k$. Therefore the program returns y as the queried element. The analysis of running time is the similar to SEARCH, which is $O(\log n)$ expected time.

Problem 5-4. Convex Layers

Given a set Q of points in the plane, we define the ***convex layers*** of Q inductively. The first convex layer of Q consists of those points in Q that are vertices of CH(Q). For $i > 1$, define Q_i to consist of the points of Q with all points in convex layers $1, 2, \ldots, i-1$ removed. Then the ith convex layer of Q is CH(Q_i) if $Q_i \neq \emptyset$ and is undefined otherwise.

(a) Give an $O(n^2)$-time algorithm to find the convex layers of a set on n points. Hint: Refer to CLRS chapter 33.3.

Solution: Jarvis's march to find each convex layer. If there are k convex layers and the ith layer contains l_i points, the total running time is

$$O(nl_i + nl_2 + \cdots + nl_k) = O(n^2) \; ,$$

since $\sum_{i=1}^{k} l_i = n$.

(b) Suppose we are given an unsorted array of n real values A. Let the array B contain the values of A in descending sorted order. Give a linear time algorithm to convert A to a set of points Q, such that each convex layer Q_i can be translated to $B[i]$ in constant time. In other words, give a linear time *reduction* from the sorting problem to the convex layer problem.

Solution: We reduce sorting to computing the convex layers by showing that given a set of n numbers to sort, we can construct in linear time a set of points whose convex layers can be interpreted in linear time to give the numbers in sorted order. Given A we put three points into set Q for each $A[j]$: $(0, 0)$, $(j, 0)$, and $(0, A[j])$. Q has n convex layers, where Q_i is a triangle with vertices $(0, 0)$, $(n + i - 1, 0)$, and $(0, B[i])$. Thus, it is trivial to convert a given convex layer its corresponding sorted value.

This gives a lower bound on the time to find convex layers. If we could determine convex layers in $o(n \log n)$ time, then we could sort in $o(n \log n)$ time. Therefore, finding convex layers takes $\Omega(n \log n)$ time.