Problem Set 4 Solutions

Problem 4-1. AVL Trees. An **AVL tree** is a binary search tree that is *height balanced*, i.e., for each node \( x \), the heights of the left and right subtrees of \( x \) differ by at most 1. To implement an AVL tree, we maintain an extra field in each node: \( h[x] \) is the height of the node \( x \). Our first goal is to prove that BST-SEARCH runs in \( O(\lg n) \) time on an AVL tree.

(a) Prove that an AVL tree with \( n \) nodes has height \( \Theta(\lg n) \). *(Hint: Prove that an AVL tree of height \( h \) has at least \( F_h \) nodes, where \( F_h \) is the \( h \)th Fibonacci number.)*

**Solution:** Let \( N_h \) denote the smallest possible number of nodes in an AVL tree of height \( h \). We shall show by induction that \( N_h \geq F_h \).

**Base:** Clearly, \( N_0 = 1 \geq 0 = F_0 \), and \( N_1 = 2 \geq 1 = F_1 \).

**Induction step:** Assume \( h \geq 2 \) and that for all \( k < h \), \( N_k \geq F_k \). Now, for any AVL tree of height \( h \), the height of one of the two subtrees of the root must be \( h - 1 \); the other subtree of the root must have height at least \( h - 2 \). Therefore, the number of nodes in the tree is at least \( N_{h-1} + N_{h-2} + 1 \), which by the induction hypothesis is at least \( F_{h-1} + F_{h-2} \). Since this is true for all AVL trees of height \( h \), we have that \( N_h \geq F_{h-1} + F_{h-2} = F_h \).

Thus \( N_h \geq F_h \) for all \( h \). Now recall that \( F_{i+2} \geq \phi^i \), where \( \phi > 1 \) is the golden ratio (see CLRS Exercise 3.2-7). This gives us that for \( h \geq 2 \), \( N_h \geq \phi^{h-2} \), or \( h \leq \log_\phi N_h + 2 \). Thus for \( n \geq 4 \), the height \( h \) of an AVL tree with \( n \) nodes satisfies \( h \leq \log_\phi N_h + 2 \leq \log_\phi n + 2 \), which implies that an AVL tree with \( n \) nodes has height \( O(\lg n) \). Since any binary tree on \( n \) nodes has height at least \( \lg n \), we also have that an AVL tree with \( n \) nodes has height \( \Omega(\lg n) \). This proves that the height of an AVL tree on \( n \) nodes is \( \Theta(\lg n) \).

Our remaining goal is to show how to insert a node into an AVL tree while maintaining the height-balance invariant. To insert a key into an AVL tree, we first place a node with that key in the appropriate place in the binary search tree order, using BST-INSERT. 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.

(b) Describe a procedure \( \text{BALANCE}(x) \) that takes a node \( x \), whose left and right child subtrees are height balanced and have heights that differ by at most 2, i.e.,

\[
|h[\text{right}[x]] - h[\text{left}[x]]| \leq 2,
\]

and alters the subtree rooted at \( x \) to be height balanced. *(Hint: Use rotations.)*
Problem Set 4 Solutions

Solution: We sketch how BALANCE(x) works rather than giving pseudocode.

If |h[right[x]] − h[left[x]]| ≤ 1, there is nothing to do. Suppose |h[right[x]] − h[left[x]]| = 2, and without loss of generality, assume that h[left[x]] = h and h[right[x]] = h + 2.

We need to consider three cases:

1. The right child of x is right-heavy (i.e. its right child has height h + 1 and its left child height h).
2. The right child of x is balanced (i.e. both its children have height h + 1).
3. The right child of x is left-heavy (i.e. its left child has height h + 1 and right child height h).

**Case 1** This case is illustrated below. Here the height of T₁ is h, the height of T₂ is h, and the height of T₃ is h + 1. In this case, a single left rotation at node x makes the whole tree height-balanced:

```
    x
   / \  \
T₁  y  T₃  \\
  / \     \
T₂  T₃  T₁  T₂
```

After performing the rotation we update the the values of the field h[] at nodes x and y. It is worth noting that this procedure reduces the height of the tree by 1.

**Case 2** This case looks similar to that illustrated in Case 1 above, except that the height of T₂ here is h + 1. It can be verified that in this case too, a single left rotation at x makes the tree height-balanced. Again, after the rotation we update the values of the height field h[] at nodes x and y. In this case, however, the height of the tree remains the same.

**Case 3** This case is illustrated below. Here the heights of T₁ and T₄ are h; one of the trees T₂ or T₃ has height h, while the other has height h or h − 1.

```
    x
   / \  \
T₁  y  \ 
  /     z
T₂  \\  T₃  \\
 \    \  \  
 \   T₄
```

We make the tree above height-balanced by performing a sequence of two rotations.
First we perform a right rotation at node $y$, and then a left rotation at node $x$:

\[
\begin{array}{ccc}
 T_1 & & T_2 \\
 \searrow & \searrow & \nearrow \\
 x & z & \Rightarrow & y \\
 \searrow & \searrow & \searrow & \nearrow \\
 T_3 & T_4 & T_1 & T_2 \\
 \end{array}
\]

Note that the sequence of two rotations described above reduces the height of the tree by 1 and makes the whole tree height-balanced. After performing the rotations we update the values of field $h[]$ at nodes $x$, $y$ and $z$.

Since $\text{BALANCE}(x)$ involves 1 or 2 rotations and a constant number of height field updates, its running time is clearly $O(1)$.

(c) Using part (b), describe a procedure $\text{AVL-INSERT}(T, z)$ that, given an AVL tree $T$ and a newly created node $z$ (whose key has already been filled in), inserts $z$ into $T$.

**Solution:** The idea behind $\text{AVL-INSERT}(T, z)$ is the following. We first insert node $z$ into $T$ using the standard procedure $\text{TREE-INSERT}(T, z)$, which ignores the fact that $T$ is an AVL tree and makes the insertion as into a usual binary search tree. Then, we follow the path up the tree starting from $z$ to $\text{root}[T]$, updating the heights of nodes and performing balancing (if necessary) along the way. Below is the pseudo-code.

1. $\text{AVL-INSERT}(T, z)$
2. $\text{TREE-INSERT}(T, z)$
3. $h[z] \leftarrow 0$
4. $z \leftarrow p[z]$
5. while $z \neq \text{nil}$
   6. $z' \leftarrow p[z]$
   7. if $\text{left}[z] = \text{nil}$ $l \leftarrow 0$ else $l \leftarrow h[\text{left}[z]]$
   8. if $\text{right}[z] = \text{nil}$ $r \leftarrow 0$ else $r \leftarrow h[\text{right}[z]]$
   9. if $|l - r| \leq 1$
      10. $h[z] = \max(l, r) + 1$
   11. else
      12. $\text{BALANCE}(z)$
      13. $z \leftarrow z'$

(d) Prove that $\text{AVL-INSERT}$, run on an $n$-node AVL tree, takes $O(\log n)$ time and performs $O(1)$ rotations.
Problem Set 4 Solutions

Solution: Part (a) implies that every path from the root of an AVL tree to its leaf has length $O(\log n)$. On a call to AVL-INSERT we traverse some such path (from the newly inserted leaf to the root), performing a constant amount of work at each level. Therefore the total running time of AVL-INSERT is $O(\log n)$.

We now argue that every invocation of AVL-INSERT results in at most one invocation of BALANCE. In particular, since we start with an AVL tree, any invocation of BALANCE during the AVL-INSERT procedure must fall in Case 1 or Case 3 (see part (b)). (To see this, note that if a call to BALANCE($x$) fell in Case 2, then it would not possible to obtain an AVL tree by removing one node from the subtree rooted at $x$.) Since in each of these two cases, a call to BALANCE($x$) reduces the height of the tree rooted at $x$ by 1, after such a call the height of the subtree rooted at $x$ becomes the same as that before the insertion of the new node, and the tree is again an AVL tree (i.e. no more balancing is required). Thus, the total number of rotations performed on a call to AVL-INSERT is at most two.

Problem 4-2. Merging of data structures. In this problem, you will be asked to design efficient algorithms to merge various kinds of data structures. When we talk about the running time of an algorithm for merging two structures of sizes $n_1$ and $n_2$, we mean the running time as a function of the sum $n = n_1 + n_2$ of the inputs lengths. In particular, a merging algorithm runs in linear time if it takes $O(n) = O(n_1 + n_2)$ time.

(a) Give a linear-time algorithm to merge two sorted linked lists $L_1$ and $L_2$ into a new sorted linked list $L$.

Solution: Note that we have already seen an efficient algorithm to merge two sorted lists in this class. In fact, such procedure is used as a subroutine in the MERGE-SORT algorithm. We keep a pointer to each of the input lists. We build up the new list by consecutively adding the minimal element that is currently observed in input lists to the new list (and shifting the corresponding pointer).

The following procedure gets pointers to input lists, and returns a pointer to the new (merged) list. \texttt{copy($x$)} makes a copy of the node stored $x$ and returns a pointer it. \texttt{next[$x$]} returns the pointer to the next node in the list. \texttt{key[$x$]} returns the value stored at node $x$.

1 \texttt{MERGEL($x$, $y$)}
2 \hspace{1em} if ($x = \text{nil}$) and ($y = \text{nil}$) return nil
3 \hspace{1em} if ($x = \text{nil}$) \quad $z = \text{copy($y$)}$; \quad $y \leftarrow \text{next[$y$]}$; \quad next[$z$] = \text{MergeL(}$\text{nil}$, $y$)$; \quad \text{return $z$};
4 \hspace{1em} if ($y = \text{nil}$) \quad $z = \text{copy($x$)}$; \quad $x \leftarrow \text{next[$x$]}$; \quad next[$z$] = \text{MergeL(}$x$, $\text{nil}$)$; \quad \text{return $z$};
5 \hspace{1em} $xv \leftarrow \text{key[$x$]}$
6 \hspace{1em} $yv \leftarrow \text{key[$y$]}$
7 \hspace{1em} if ($xv < yv$) \quad $z = \text{copy($x$)}$; \quad $x \leftarrow \text{next[$x$]}$; \quad next[$z$] = \text{MergeL(}$x$, $y$)$; \quad \text{return $z$};
8 \hspace{1em} else \quad $z = \text{copy($y$)}$; \quad $y \leftarrow \text{next[$y$]}$; \quad next[$z$] = \text{MergeL(}$x$, $y$)$; \quad \text{return $z$};
It is easy to verify that the procedure above takes linear time since on every iteration we add one element of lists $x$ and $y$ to the new list.

(b) Call a binary search tree $T$ nearly balanced if there exists some integer $k$ such that every path from the root of $T$ to a leaf of $T$ has length either $k$ or $k + 1$. Give a linear-time algorithm to merge two binary search trees $T_1$ and $T_2$ into a nearly balanced binary search tree $T$. You can assume that all keys in the binary trees are distinct.

Solution: Our algorithm works in three steps. On the first step we convert each of the input binary trees into a sorted list via an inorder tree walk. This part is standard and takes linear time [CLRS, p. 255]. On the second step we merge the sorted lists via a procedure from part a. Finally on the third step we turn a (merged) sorted list into a nearly balanced binary search tree. Below we argue that the last step can be performed in linear time.

Let $n$ be the length of the sorted list $L$. Let $k$ be such that $2^k - 1 \leq n < 2^{k+1} - 1$. Let $n = 2^k - 1 + s$. We are going to convert $L$ into a binary tree $T$ that has first $k$ levels completely filled up, and the $(k + 1)$-st level containing $s$ elements. First we build a binary tree $T$ with the topology specified above. By default, we set all the key values in our tree to 0.

Next we fill up tree $T$ with values from the list $L$, by consecutively inserting the smallest element of $L$ into $T$. After each insertion we trivially shift the pointer in $L$, and make a call to TREE-SUCCESSOR to shift the pointer in $T$. We rely on the fact that TREE-SUCCESSOR finds the successor of a given key without ever comparing the keys. Let $l$ be our pointer in $L$ and $x$ be our pointer in $T$. We assume that initially $x$ points to the left-most node in $T$, i.e., to the node that should contain the smallest key.

1. FILLTREE($x, l$)
2. if ($l = \text{nil}$) return ;
3. else key[$x$] = key[$l$]; $l \leftarrow \text{next}[l]$;
4. $x \leftarrow \text{TREE-SUCCESSOR}(x)$; FILLTREE($x, l$); return ;

It remains to notice that the above procedure runs in linear time, since while executing a sequence of calls to TREE-SUCCESSOR starting with the minimum element we traverse each edge in $T$ at most twice.

(c) Give a linear-time algorithm to merge two well formed red-black trees into a new well formed red-black tree. You can assume that all keys in the red-black trees are distinct.

Solution: Our algorithm works in two steps. On the first step we ignore the fact that out trees are red-black and merge them as usual binary search trees into a nearly balanced binary search tree using the procedure developed in part (b). The resulting
Problem Set 4 Solutions

tree has either depth $k$ or $k + 1$. Levels 1 through $k$ are completely filled up, and level $k + 1$ is either empty or partially filled up. On the second step we convert a nearly balanced binary search tree into a red black tree by assigning colors to nodes of the tree. We do not modify the tree structure.

We color the nodes of a nearly balanced binary tree according to the following rule: A node is colored red if it is located at level $k + 1$. All other nodes are colored black. It is easy to verify that such coloring can be done in linear time and satisfies the coloring constraints of a red black tree.

(d) Give an $O(\lg n)$-time algorithm to merge two 2-3-4 trees $T_1$ and $T_2$ into a new 2-3-4 tree $T$, assuming that all keys stored in $T_1$ are smaller than all keys stored in $T_2$. You can assume that all keys in the 2-3-4 trees are distinct.

Solution: Let $h_1$ be the height of $T_1$ and $h_2$ be the height of $T_2$. Note that both $h_1$ and $h_2$ can be computed in $O(\lg n)$ time. Without loss of generality assume that $h_1 \leq h_2$. The high-level idea behind our merging algorithm is to walk down the left-most path in $T_2$ until we reach a node $V$ at height $h_1$ and merge the root of $T_1$ with the node $V$. Clearly, there is a number of technical details that we need to take care about. In fact merging the nodes may be problematic for three reasons:

1. The node $V$ may be full.
2. Even if $V$ is not full it may not have enough empty slots in the array $c[V]$ to store pointers to all subtrees of the root of $T_1$.
3. When adding a new subtree to the beginning of the array $c[V]$ we need to add a new key to the beginning of the array $key[V]$.

We take care of the first two issues by:

1. Splitting full nodes when we walk from the root of $T_2$ to the node $V$. This is exactly what is done in the 2-3-4-TREE-INSERT procedure.
2. Inserting the subtrees of the root of $T_1$ into $c[V]$ one by one.

The last issue is merely a technicality. Below is a more formal description of our algorithm:

1. $m \leftarrow \text{GET-MAXIMUM}(\text{root}[T_1])$
2. $\text{2-3-4-TREE-DELETE}(\text{root}[T_1], m)$
3. $h_1 \leftarrow \text{HEIGHT}(\text{root}[T_1])$
4. For $i = n[\text{root}[T_1]] + 1$ Downto 1
5. If $i \neq n[\text{root}[T_1]] + 1$ $m \leftarrow key_i[\text{root}[T_1]]$
6. Walk down the left-most path in $T_2$ splitting every full node along the way, until you reach a node $V$ at height $h_1$
7. Shift all the values in array $c[V]$ and $key[V]$ by one;
8. Set $c_1[V] = c_i[\text{root}[T_1]]$ and $key_1[V] \leftarrow m$
It remains to notice that the number of loop iterations made by our algorithm is equal to the number of children of the root of $T_1$, which is at most 4. Since every iteration takes $O(lg n)$ time, the total running time is therefore $O(lg n)$. 