Problem Set 4 Solutions

This problem set is due in recitation on Friday, November 2th, 2007.

Instructions:

- 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.

- You will often be called upon to “give an algorithm” to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of the essay should provide the following:

  1. A description of the algorithm in English and, if helpful, pseudo-code.
  2. At least one worked example or diagram to show more precisely how your algorithm works.
  3. A proof (or indication) of the correctness of the algorithm.
  4. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Full credit will be given only to the correct solution which are 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, prevent you from writing convoluted solutions, and also help you conceptualize the key idea of the problem.

- Be sure to know the section on “Guide to writing up homework” in the “Course Information” handout before writing your solutions.

Problem 4-1. Scheduling Seminars

Today you have to decide your schedule for seminars. There are \( n \) seminars, and each seminar has its starting time \( s_i \) and finishing time \( f_i \). You will select them greedily (according to various criteria described below), while ensuring that no chosen seminar overlaps with those previously scheduled. Your goal is to attend the maximum number of seminars.

(a) For each of the following three greedy algorithms, give a counter-example to show that they may fail to find an optimal schedule.

- The SHORTEST SEMINAR FIRST algorithm:
– Initially, choose the seminar that has the shortest duration (i.e. the one with the smallest value of \( f_i - s_i \)), and then iteratively choosing the next shortest one among remaining seminars that doesn’t conflict with the previously chosen ones. In other words, pick seminars in increasing order of \( f_i - s_i \), excluding ones that overlap previously chosen ones.

- The **Earliest Starting Seminar First** algorithm:
  – Pick seminars in increasing order of the starting time \( s_i \), among those that do not overlap with previously chosen ones.

- The **Fewest Conflicts Seminar First** algorithm:
  – Pick seminars in increasing order of the number of conflicts that they have.

**Solution:**

![Figure 1: Counter example for the shortest seminar first algorithm](image1)

![Figure 2: Counter example for the earliest starting seminar first algorithm](image2)

![Figure 3: Counter example for the fewest conflicts seminar first algorithm](image3)

(b) Find an efficient algorithm that generates an optimal schedule, and prove its correctness.

**Solution:** The earliest finishing seminar first algorithm works (i.e. pick seminars in increasing order of \( f_i \)). For proving its correctness, claim that there is an optimal schedule containing the earliest finishing seminar \((s_i, f_i)\). To prove the claim, suppose that \( S_{OPT} \) is one of the optimal schedules. We can find another optimal schedule \( S'_{OPT} \)
containing \((s_i, f_i)\) by replacing the first seminar in \(S_{OPT}\) with \((s_i, f_i)\). \((s_i, f_i)\) can be replaced without confliction because it is the earliest finishing seminar.) Now, by the claim, it is sufficient to solve the sub-problem to find an optimal schedule among those which don’t conflict with \((s_i, f_i)\). Therefore, the following algorithm finds an optimal schedule, where \(S\) is the set of seminars.

\[
\text{FINDBESTSCHEDULE}(S)
\]

1. Find the earliest seminar \((s_i, f_i)\) in \(S\).
2. Remove seminars from \(S\), which conflict with \((s_i, f_i)\).
3. Return \((s_i, f_i) \cup \text{FINDBESTSCHEDULE}(S)\)

Actually, if we think the recursive call, this algorithm is a greedy algorithm to pick seminars in increasing order of \(f_i\).

**Problem 4-2. AA tree**

An AA tree is a binary search tree that has the following properties.

1. Every node has an AA-level.
2. If a node is a leaf node, its AA-level is 1.
3. If a node is a left child, its AA-level is one less than the AA-level of its parent.
4. If a node is a right child, its AA-level is equal to or one less than the AA-level of its parent.
5. The AA-level of a node is strictly greater than the AA-level of its grand child.
6. Every node at the AA-level strictly greater than 1 has two children.

For example, the following binary search tree is an AA tree, which was drawn so that all node that have the same AA-level are located horizontally.

![AA Tree Diagram](image)

**Figure 4:** Example for the AA tree. There are nine nodes of AA-level 1, four nodes of AA-level 2, and two nodes of AA-level 3.

As you see in Figure 4, it is convenient to think an AA tree by positioning the nodes at the same AA-level horizontally and linking between a node and its child. Then, the third property implies...
there is no left horizontal link, while a right link can go horizontally due to the fourth property.
The fifth property says that there cannot be two consecutive right horizontal links.

Our goal is to show how to insert a node into an AA tree while maintaining all properties. To insert a key into an AA tree, we first place a node with that key in the appropriate place in the binary search order using BST-INSERT, and assign 1 as its AA-level. After this insertion, the tree may no longer satisfy the third or fifth property. In Figure 4, inserting 2 will break the third property (i.e. create a left horizontal link), and inserting 37 creates two consecutive right horizontal links which breaks the fifth property.

(a) Suppose a node $x$, whose left and right child subtrees are AA trees, has the same AA-level as its left child $l$ (i.e. they form a left horizontal link). Describe a procedure to fix it, although it may create two consecutive right horizontal links. (Assume that the children of $l$ have the same AA-level.)

Solution: See Figure 6 how to fix a left horizontal link. If $x$ has a right horizontal link, it will create two consecutive right horizontal links.
(b) Suppose a node \( x \), whose left and right child subtrees are AA trees, has the same AA-level as its grand child (i.e. if \( y \) is a right child of \( x \) and \( z \) is a right child of \( y \), \( x \), \( y \) and \( z \) forms two consecutive right horizontal links). Describe a procedure to fix it at the level \( i \), although it may create a problem at the AA-level \( i + 1 \), where \( i \) is the AA-level of \( x \).

![Figure 7: Two consecutive right horizontal links problem](image)

**Solution:** Figure 8 shows how to fix two consecutive right horizontal links. Let \( w \) be the parent of \( x \) before fixing. Then \( w \) will be a parent of \( y \). If \( x \) was a left child, \( y \) will create a left horizontal link. Also, if \( x \) was a right child and \( w \) is also a right child, \( w \) and \( y \) will create two consecutive horizontal links at the level \( i + 1 \).

![Figure 8: Fixing two consecutive right horizontal links](image)

(c) Using part (a) and (b), describe a procedure \( \text{AA-INSERT}(T, z) \) that, given an AA tree \( T \) and a newly created node \( z \) (whose key has already been filled in), inserts \( z \) into \( T \).
**Solution:** It is straightforward from (a) and (b). At first, insert using BST-INSERT and assign 1 as its AA-level. It may create a problem of ‘left horizontal link’ or ‘two consecutive right horizontal links’. We can fix it using (a) or (b), and drive a problem to a higher AA-level. At the highest AA-level, fixing will make no problem.

(d) Prove that AA-INSERT, run on an \( n \)-node AA-tree, takes \( O(\log n) \) time. (Hint: Prove that an AA tree with \( n \) nodes has height \( O(\log n) \).)

**Solution:** It is enough to show its height \( h \) is \( O(\log n) \), because BST-INSERT takes \( O(h) \) time and the iterative fixing procedure in (c) ends in at most \( 2h \) steps. Properties of AA tree enforce that the longest path from the root to a leaf is no more than twice as long as the shortest path from the root to a leaf in that tree. Now define the following notation.

- \( a \): the number of nodes in a left subtree of the root
- \( b \): the number of nodes in a right subtree of the root
- \( h_a \): the length of the shortest path from the root to a leaf in a left subtree of the root
- \( h_b \): the length of the longest path from the root to a leaf in a right subtree of the root

Then, they have the following relations.

- \( n = 1 + a + b \)
- \( a \geq 2^{h_a} - 1 \)
- \( b \leq 2^{h_b} - 1 \)
- \( h_b \leq 2h_a \)

The second and third relations are from the fact that the number of nodes in a complete binary tree with a height \( h \) is \( 2^{h+1} - 1 \). Therefore,

\[
h_b \leq 2h_a \leq 2\log(a + 1) \leq 2\log n.
\]

Similarly, we can prove the length of the longest path in a left subtree of the root is also at most \( 2\log n \). This completes the proof.

**Problem 4-3. Relation between Birthday and IQ**

You are researching the relation between Birthday and IQ, and maintain a database of many people’s birthday and IQ. You would like to be able to quickly answer questions of the form: “What was the average IQ among people born between day \( a \) and day \( b \)?” Your goal is to create an efficient dynamic data structure that the following operations:

- \( \text{INSERT}(i, j) \) inserts a person with birthday \( i \) and IQ \( j \). You may assume both \( i \) and \( j \) are positive integers.
• \textsc{Average}(a, b) returns the average IQ among people born between days \(a\) and \(b\) inclusively. (Assume \(a < b\).)

You decide to create your data structure by augmenting a balanced binary search tree such as AA trees, red-black trees or 2-3 trees.

(a) Describe the data contained in each node of your augmented BST.

Solution: For each node \(x\), store the two extra fields.

\begin{itemize}
  \item \textsc{sumIQ}[x] = the sum IQ over all people in the subtree rooted at \(x\).
  \item \textsc{num}[x] = the number of people in the subtree rooted at \(x\).
\end{itemize}

(b) Describe your implementation of \textsc{Insert} operation.

Solution: For any balanced binary search tree you choose, we can implement the augmented \textsc{Insert} operation if the fields \textsc{sumIQ}[x] and \textsc{num}[x] can be computed from the fields at the children of \(x\). They can be computed as follows:

\begin{itemize}
  \item \textsc{sumIQ}[x] = \textsc{sumIQ}[\textsc{left}[x]] + \textsc{iq}[x] + \textsc{sumIQ}[\textsc{right}[x]]
  \item \textsc{num}[x] = \textsc{num}[\textsc{left}[x]] + 1 + \textsc{num}[\textsc{right}[x]]
\end{itemize}

(c) Describe your implementation of \textsc{Average} operation. Prove its correctness, and analyze its running time in term of the total number \(n\) of data inserted so far.

Solution: At first, implement \textsc{sumIQAfter}(x, a) and \textsc{sumIQBefore}(x, b) which return the sum of IQ over all people in the subtree rooted at \(x\) who were born after day \(a\) and before day \(b\) respectively.

\begin{verbatim}
\textsc{sumIQAfter}(x, a)
1. \textbf{if} \(x = \text{nil}\) \textbf{return} 0
2. \textbf{if} \(\text{key}[x] < a\) \textbf{then return} \textsc{sumIQAfter}(\textsc{right}[x], a)
3. \textbf{else return} \textsc{sumIQAfter}(\textsc{left}[x], a) + \textsc{iq}[x] + \textsc{sumIQ}[\textsc{right}[x]]
\end{verbatim}

\begin{verbatim}
\textsc{sumIQBefore}(x, b)
1. \textbf{if} \(x = \text{nil}\) \textbf{return} 0
2. \textbf{if} \(\text{key}[x] > b\) \textbf{then return} \textsc{sumIQBefore}(\textsc{left}[x], b)
3. \textbf{else return} \textsc{sumIQBefore}(\textsc{right}[x], a) + \textsc{iq}[x] + \textsc{sumIQ}[\textsc{left}[x]]
\end{verbatim}
Using these two sub-routines, $\text{SUMIQBETWEEN}(a, b)$ which returns the sum of IQ over all people born between day $a$ and day $b$ ($a < b$) can be computed as:

$$\text{SUMIQBETWEEN}(a, b) = \text{SUMIQAFTER}(\text{root}, a) + \text{SUMIQBEFORE}(\text{root}, b) - \text{SUMIQ}[\text{root}].$$

Similarly, we can implement $\text{NUMBETWEEN}(a, b)$ which returns the number of people born between day $a$ and day $b$ ($a < b$). Hence, $\text{AVERAGE}(a, b)$ is computed as:

$$\text{AVERAGE}(a, b) = \frac{\text{SUMIQBETWEEN}(a, b)}{\text{NUMBETWEEN}(a, b)}.$$  

The running time is $\Theta(h) = \Theta(\log n)$, where $h$ is the height of the balanced binary search tree, because both $\text{SUMIQAFTER}(\text{root}, a)$ and $\text{SUMIQAFTER}(\text{root}, b)$ take $\Theta(h)$ time.