CPSC 629: Analysis of Algorithms, Fall 2003

Solutions to Homework 2

Solution to 1 (17.1-3)

$$f(n) = n + \sum_{i=1}^{\lfloor \lg n \rfloor} (2^i - 1) = n - \lfloor \lg n \rfloor + \sum_{i=1}^{\lfloor \lg n \rfloor} 2^i < n - \lfloor \lg n \rfloor + 2n < 3n.$$

Hence, amortized cost is 3 per operation. $\square$

Solution to 2 (17.2-2)
Charge 3 for each operation. For operation i, if i is not a power of 2, the cost is paid by its charge with 2 extra credits. If $i = 2^j$ for some $j \in \mathbb{N}$, then the extra credits paid by operation $2^{j-1}+1, \ldots, 2^j - 1$ is $2(2^{j-1} - 1)$, which, together with the 3 credit of the current operation, is enough for operation i.
$\square$

Solution to 3 (17.3-2)
Define the potential function $\Phi(D_0) = 0$, and $\Phi(D_i) = 2i - 2^{1+\lfloor \lg i \rfloor}$ for $i > 0$. For operation 1,

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 2 - 2^{1+\lfloor \lg 1 \rfloor} - 0 = 1.$$

For operation i ($i > 1$), if i is not a power of 2, then

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 2i - 2^{1+\lfloor \lg i \rfloor} - [2(i-1) - 2^{1+\lfloor \lg(i-1) \rfloor}] = 3.$$

If $i = 2^j$ for some $j \in \mathbb{N}$, then

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = i + 2i - 2^{1+j} - [2(i-1) - 2^{1+j-1}] = i + 2i - 2i - 2i + 2 + i = 2.$$

Thus, the amortized cost is 3 per operation. $\square$

Solution to 4 (17.3-3)
Let the potential function be $\Phi(D_i) = \sum_{k=1}^{i} \lg k$. Then the amortized cost for INSERT is

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) \le \lg i + \sum_{k=1}^{i} \lg k - \sum_{k=1}^{i-1} \lg k = 2 \lg i \in O(\lg n).$$

The amortized cost for EXTRACT-MIN is

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) \le \lg i + 1 + \sum_{k=1}^{i-1} \lg k - \sum_{k=1}^{i} \lg k = 1 \in O(1).$$

$\square$

Solution to 5 (17.4-3)
For a TABLE-DELETE operation without contraction, the change of potential value before and after the operation is 2 or -2. Therefore, the amortized cost is

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) \le 1 + 2 = 3 \in O(1).$$

If a TABLE-DELETE operation triggers contraction, then

$$\begin{aligned}
\widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\leq num[T_i] + 1 + |2 \cdot num[T_i] - size[T_i]| - |2 \cdot num[T_{i-1}] - size[T_{i-1}]| \\
&= num[T_i] + 1 + 0 - |2 \cdot num[T_i] - 3 \cdot num[T_i]| \\
&= 1 \in O(1).
\end{aligned}$$

$\square$

Solution to 6 (17-2)

1. To perform a SEARCH on this data structure, we need to perform binary search in the sorted arrays. In worst case, we need to search all the arrays to find an element. Start from the largest array, do a binary search, if the element is not found, go to the next largest array, and so on, until the element is found or all arrays are searched. The worst case running time is $\sum_{i=0}^{k-1} i \cdot n_i \in O(\lg^2 n)$. *Remark: To search in decreasing order of array size gives better performances in some cases, but the improvement does not make much difference since worst case running time is always needed if an element does not exist in the arrays.*

2. Suppose there are n elements in the original arrays. First find the smallest array A_i that is not empty. Then insert the new element into A_i and move all the elements from $A_0, \ldots, A_{i-1}$ into A_i. In worst case, we will move all $n + 1$ elements.

   To analyze the amortized cost, the aggregation method appears to be most convenient. For a sequence of n insertions, the array A_i will be changed $n/2^i$ times (by a change, we mean from empty to full or from full to empty). Therefore, the total cost of the n operations is

   $$\sum_{i=0}^{k-1} \frac{2^i n}{2^i} = \sum_{i=0}^{k-1} n = nk \in O(n \lg n).$$

   The amortized cost is $O(\lg n)$ per insertion.

3. Observe that the DELETE cannot be done in time better than SEARCH. Indeed, there is no way to implement DELETE in better than linear time, unless we impose some relationship between elements in different arrays. Consider a deletion that removes an item at the center of the largest array A_{k-1}. After the deletion, at least one element e must be moved from other arrays to A_{k-1}. Since there is no relationship between e and other elements in A_{k-1}, reshuffling A_{k-1} into sorted order make take time $2^{k-2} \in O(n)$. Since each deletion may take this much time, the amortized cost is the same.

   For a simple implementation, first find the smallest array A_i that is not empty. Then find the element that is to be deleted. Assume it is in array A_j. Remove the item from A_j. Move one element from A_i to A_j (make no change if $i = j$), and break A_i into $i - 1$ arrays $A_0, \ldots, A_{i-1}$. Finally, rearrange the array A_j into order. As explained before, the DELETE operation takes $O(n)$ in worse case.

$\square$

Solution to 7 (17-3)

1. First sort elements in the subtree rooted at x. Given a binary search tree, it can be done in linear time. Supposed the sorted elements are $s_1, \ldots, s_k$. Select $s_{\lfloor k/2 \rfloor}$ as the root and divide the remaining elements into two halves. Recursively build the tree for the halves. Eventually every element is assigned to a node. It is easy to see that the procedure takes linear time and needs linear auxiliary space.

2. Let $f(n)$ be the time of performing a search in an α-balanced tree of n elements, then $f(n) \leq f(\alpha n) + O(1)$. Solve the recurrence relation gives $f(n) \in O(\log_{1/\alpha} n) = O(\lg n)$.

3. For every node x in a 1/2-balanced, we have

$$size[left[x]] + size[right[x]] + 1 = size[x],$$

and

$$size[left[x]] \leq 1/2 \cdot size[x],$$
$$size[right[x]] \leq 1/2 \cdot size[x].$$

Thus

$$|size[left[x]] - size[right[x]]| \leq 1.$$

Therefore $\Delta(x) = 0$, which implies that a 1/2-balanced tree as potential 0. It is also clear that summation of absolute values are nonnegative.

4. If a rebuild is triggered at a node x, then x is no longer α-balanced before the rebuild. Therefore the potential of x before the rebuild is

$$\Delta(x) = |size[left[x]] - size[right[x]]| \geq \alpha \cdot size[x] - (size[x] - \alpha \cdot size[x]) = (2\alpha - 1)size[x].$$

Also observe that the potential of the subtree rooted at x will have potential 0 after rebuild. The potential is decreased after the rebuild by at least $\Delta(x)$. In order to cover the cost of rebuild, it suffice to satisfy

$$c \cdot \Delta(x) \geq size[x],$$

since $\Phi(T_x) \geq c \cdot \Delta(x)$. This implies that $c(2\alpha - 1) \geq 1$, or $c \geq 1/(2\alpha - 1)$.

5. Let $\widehat{c_i}$ be the amortized cost of an operation of inserting or deleting a node in an n node α-balanced tree. If the operation does not trigger a rebuild, then

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

By result of (b),

$$c_i \in O(\lg n).$$

Similarly, the maximum height of the tree is also $O(\lg n)$. Then either inserting or deleting a node v in the tree will change the $\Delta(x)$ for every node x on the path from the node v to the root by one. Therefore,

$$\Phi(D_i) - \Phi(D_{i-1}) \leq c \cdot O(\lg n) \in O(\lg n).$$

If the operation triggers a rebuild, let $c_i = c_i' + c_i''$, where c_i' is the cost of insertion of deletion, and c_i'' is the cost of rebuild. Let D_i' be the state after the node is inserted or deleted, but before the rebuild is called. Then

$$\widehat{c_i} = c_i' + c_i'' + [\Phi(D_i) - \Phi(D_i')] + [\Phi(D_i') - \Phi(D_{i-1})]$$

Similar to the first case,

$$\Phi(D_i') - \Phi(D_{i-1}) \leq O(\lg n).$$

$\Phi(D_i')$ is the potential before the rebuild and $\Phi(D_i)$ is the potential after the deletion, and $\Phi(D_i') - \Phi(D_i)$ is the decrease of the potential after rebuild. By the result of 4,

$$c_i'' \leq \Phi(D_i') - \Phi(D_i).$$

Therefore

$$\widehat{c_i} = c_i' + c_i'' + [\Phi(D_i) - \Phi(D_i')] + [\Phi(D_i') - \Phi(D_{i-1})] \leq O(\lg n) + c_i'' - [\Phi(D_i') - \Phi(D_i)] + O(\lg n) \leq O(\lg n).$$

□

Solution to 8 (18.2-6)
If the search within a node is implement by binary search, then the time taken by the **while** loop of line 2-3 in the search algorithm presented on page 442 is $O(\lg t)$. Then the total CPU time is $O(t \cdot h) = O(\lg t \cdot \log_t n) = O(\lg n)$. $\square$

Solution to 9 (18-2)

1. At each node of the tree, keep a field $height[x]$, such that $height[x] = 0$ if x is a leave, otherwise $hight[x] =$ the length of a path from x to a leave. With a few changes, the field is maintain within the procedure of splitting or merging. Therefore, the asymptotic running time of any operation is the same.

2. If $h' > h''$, then insert k into the rightmost node N' at height $h'' + 1$ of T'. If N' is full, extract its largest key k', fill k into its space, and recursively insert k' into the parent of N'. Finally, attach the tree T'' to the right of k. If $h' = h''$, create a new root with k, and attach T' and T'' to the left and right of k. If $h' < h''$, insert k into the leftmost node at height $h' + 1$ of T''. The procedure is the similar to the first case. Since the procedure has at most $|h' - h''| + 1$ recursions, each takes $O(1)$ time. The total time for joining is $O(|h' - h''| + 1)$.

3. Suppose the path p has length l, then $P = \{c_{a_1}[x_1], \ldots, c_{a_l}[x_l], k\}$, where $a_i \in \{1, 2, 3, 4\}$. For $i = 1, \ldots, l$, do:

   (a) if $c_i = 1$, then $k_i = NIL$ and $T'_i = \emptyset$;
   (b) if $c_i = 2$, then $k_i = key_1[x_i]$ and $T'_i = c_1[x_i]$;
   (c) if $c_i = 3$, then $k_i = key_2[x_i]$ and $T'_i =$ a tree formed by a root containing $key_1[x_i]$ and two subtrees $c_1[x_i]$ and $c_2[x_i]$;
   (d) if $c_i = 4$, then $k_i = key_3[x_i]$ and $T'_i =$ a tree formed by a root containing $key_1[x_i]$ and $key_2[x_i]$, and three subtrees $c_1[x_i]$, $c_2[x_i]$, and $c_3[x_i]$.

   Finally consider the node x' containing k. If $k = key_1[x']$, then $T'_{l+1} = c_1[x']$; If $k = key_2[x']$, then $T'_{l+1} =$ a tree formed by a root containing $key_1[x']$ and two subtrees $c_1[x']$ and $c_2[x']$; If $k = key_3[x']$, then $T'_{l+1} =$ a tree formed by a root containing $key_1[x']$ and $key_2[x']$, and three subtrees $c_1[x']$, $c_2[x']$ and $c_3[x']$. After tress $T_1, \ldots, T'_{l+1}$ and keys $k'_1, \ldots, k'_l$ are generated, remove all $T'_i = \emptyset$ and $k_i = NIL$. The result is a set of trees $\{T'_0, \ldots, T'_m\}$ and a set of keys $\{k'_1, \ldots, k'_m\}$. By the properties of a 2-3-4 tree, these are the desired sets. For $i = 1, \ldots, m$, the height of T'_{i-1} is greater or equal to the height of T'_i. The procedure to break S'' into sets of trees and keys is symmetric.

4. To split a tree, first find a path from the root to the key $k = key[x]$. Then break the set S' into sets of trees $\{T'_0, \ldots, T'_m\}$ and keys $k'_1, \ldots, k'_l$ as described above. Next for i from m to 1, do $join(T'_i, k_i, T'_{i-1})$. The result is a 2-3-4 tree containing the set S'. Do the same for S'' and create another tree for S''. Finally, return S' and S''. To analyze the running time, observe that the time to join two trees of hight h' and h'' is $O(1 + |h' - h''|)$ by (b). Then the total time to join all the trees in S' is

   $$\sum_{i=1}^{m}(1 + |h(T'_{i-1}) - h(T'_i)|) = m + \sum_{i=1}^{m}[h(T'_{i-1}) - h(T'_i] = m + h(T'_0) - h(T'_m) \in O(\lg n),$$

   since $m, h(T'_0) \leq h(T) \in O(\lg n)$, $h(T'_m) \geq 0$. Since the time for finding the path, breaking S' and S'' is also bounded by $O(\lg n)$, the running time for splitting is $O(\lg n)$.

$\square$

Note: The solutions given here are terse, and in some cases, incomplete. Your answers should be complete and have more details. But you will lose points if they are too long or too complex.