Solutions for Homework 4

Problem 1: 8 pts Design an algorithm for deletion in an AVL tree (lazy deletion not allowed). You have to maintain the AVL property after deletion.

Deletion in AVL trees while maintaining the AVL property is somewhat more complicated than insertion. The basic algorithm for rebalancing after deletion works similarly to the rebalancing after insertion. Whereas insertion requires at most a double rotation, deletion may require one (single or double) rotation at each level of the tree, requiring $O(\log n)$ rotations.

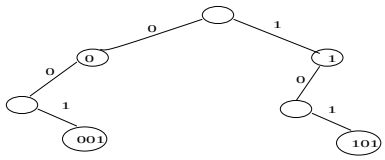
There are 2 cases illustrating the rebalancing (and 2 symmetric ones).

Case 1: Consider a tree with k_1 as the root. Its left child is k_2 and its right child is a subtree Z of height h (after removing a node in subtree Z which caused a decrease in the height). Node k_2 has a left subtree X with height $h+1$ and right subtree Y with height $h+1$ or h. Node k_1 is considered the lowest node in the tree where the AVL property is violated. Node k_1 has height $h+3$. The rebalancing requires a right rotation, where k_2 will become the root and k_1 its right child. The subtree at k_2 will have height $h+2$ or $h+3$ depending on the height of Y. If the height of k_2 is $h+2$, a new rebalancing may be required higher in the tree since the height of the root of this subtree has changed.

Case 2: Consider a tree with k_1 as the root. Its left child is k_2 and its right child is a subtree Z of height h (after removing a node in subtree Z which caused a decrease in the height). Node k_2 has a left subtree X with height h and right subtree with root k_3 with height $h+1$. Node k_3 has as left child a subtree Y' with height $h-1$ or h and a right subtree Y with height $h-1$ or h. Node k_1 is considered the lowest node in the tree where the AVL property is violated. Node k_1 has height $h+3$. The rebalancing requires a left-right double rotation, where k_3 will become the root and k_1 its left child. The subtree at k_3 will have height $h+2$. A new rebalancing may be required higher in the tree since the height of the root of this subtree has changed.

Problem 2: 8 + 10 pts Consider strings of 0s and 1s, (e.g., 0010, 101). A string s_1 is less than string s_2 if the first elements in s_1 is less than that in s_2, or if the first elements are equal in both, but the second element in s_1 is less than that in s_2, or if the first two elements are equal n both, but the third element in s_1 is less than that in s_2 and so on. In general, s_1 is less than s_2 if the first k elements are equal in both, but the $k+1$th element of s_1 is less than that of s_2 for any $k = 1, 2, ...$ For example, 0101011 is less than 01011. Also, if s_1 has length k and s_2 has length l, $k < l$ and the first k elements of s_1 and s_2 are equal, then $s_1 < s_2$. For example, 0110 is less than 01100. You have k strings. Total length of all strings is n. Give an algorithm to sort the strings in $O(n)$. For example, if you have $s_1, s_2, \ldots s_4$, and $s_1 < s_3 < s_2 < s_4$, then your algorithm should output the strings in the following sequence, s_1, s_3, s_2, s_4. Implement your algorithm in C. You may assume distinct strings.

Solution: We need to consider a binary-like tree data structure that will store the bitstrings (strings of 0s and 1s). See the next figure for more clarification. This tree stores the bit strings $0, 1, 001, 101$. When you search for a key $a = a_0 a_1 \ldots ap$, you go left at a node of depth i if $a_i = 0$ and right if $a_i = 1$. So, each node's key can be determined by traversing the path from the root to that node.



The sorting algorithm is the following: First you have to build the tree that will contain the k bitstrings. In order to store one bitstring you will need to traverse the tree from the root. The length of the string gives directly the depth of the specific bitstring in the tree. So, if the bitstring has length 4, you will need to store the value in a node of depth 4. You will have to build the intermediate nodes, if they do not appear in the tree yet. The building of the tree requires $O(n)$, where n is the total length of all strings.

After the tree buiding, you can sort the bitstring following a preorder tree traversal. Again this operation costs $O(n)$ since there are at most n nodes in the tree. We already know that if the tree consists of n nodes, then the preorder traversal takes $O(n)$.

We conclude that the sorting algorithm takes $O(n)$.

Problem 3: 8 pts Every element in a linked list consists of 2 fields, one is a string, and another is a hash value of the string. There are n elements. The size of each string is O(log n). The size of the hash value is a small constant. Design an algorithm to find an element in the list using the hash values. You may assume that at most a constant k ($k > 1$) number of elements hash into the same value. Analyze the complexity if you are not allowed to use the hash value in any way. What would your answer be for both cases if the size of a string is $O(n^2)$?

Solution: First case: using the hash values. The algorithm searches into the list sequentially for finding the appropriate hash value of the string. If there exists that hash value in the list, we have to verify that the string which corresponds to this hash value is the desired one. The reason to do this, is that there may be collisions (i.e more than 2 strings hash to the same value). Checking whether the string is the desired one takes $O(\log n)$ since thhe size of the string is $O(\log n)$ characters. There can be at most k collisions according to the problem definition. So, in the worst case the algorithm spends $O(k \log n)$=$O(\log n)$ since k is constant. Total complexity is $n+O(\log n)$ in the worst case since the algorithm must search all elements if the string is the last element in the list, or if it is not in the list. The above complexity is $O(n)$.

If you are not allowed to use hash values, then the algorithm searches every element in the list and it compares the string field with the desired one. The comparison requires $O(\log n)$ since the size of each string is $O(\log n)$ characters. In the worst case this has to be done for every element in the list. So, it requires $O(n \log n)$.

We conclude that using the hash values gives more efficient algorithm.

If the size of the string is $O(n^2)$, then using similar logic as before we have: if we use the hash values it takes $n + O(n^2)$, that is $O(n^2)$.

Using just the strings and not the hash values, it takes $O(nn^2)$, that is accessing each one of the n elements with $O(n^2)$ for character comparisons. Total is $O(n^3)$.

Problem 4: 6 pts Consider a hash function, $h(k) = k \mathrm{mod} m$, where $m = 2^p - 1$. The inputs are the strings. We convert a string to an integer as follows. Let the string be $a_{k-1}a_{k-2} \ldots a_0$, then the corresponding integer is $\sum_{i=0}^{k-1} 2^{ip} \mathrm{ASCII(a_i)}$. The claim is that any 2 strings which are permutation of each other hash into the same position. Do you agree or disagree? Justify your answer.

Solution: The claim is correct.

Proof: Consider a string $a_{k-1}a_{k-2}\ldots a_0$. For simplicity, instead of ASCII(a_i) we will write A_i. The corresponding integer is

$$K = \sum_{i=0}^{k-1} 2^{ip}\mathrm{ASCII(a_i)}$$

$$\Rightarrow K = A_0 + A_1 2^p + \ldots + A_{k-1}2^{(k-1)p}$$

$$\Rightarrow h(K) = [A_0 + A_1 2^p + \ldots + A_{k-1}2^{(k-1)p}]\mathrm{modm}$$

$$\Rightarrow h(K) = [A_0\mathrm{modm} + (\mathrm{A_1 2^p})\mathrm{modm} + \ldots + (\mathrm{A_{k-1}2^{(k-1)p}})\mathrm{modm}]\mathrm{modm}$$

$$\Rightarrow h(K) = [A_0\mathrm{modm}+(\mathrm{A_1modm\ 2^p modm})\mathrm{modm}+\ldots+(\mathrm{A_{k-1}modm\ 2^{(k-1)p}modm})\mathrm{modm}]\mathrm{modm}\ (1)$$

At this point we will prove by induction that for all $k \geq 1$, $2^{kp}\mathrm{modm} = 1$.
Base case: For $k = 1$ we have

$$2^p\mathrm{modm} = 2^p\mathrm{mod}(2^p - 1)$$

$$\Rightarrow 2^p\mathrm{modm} = (2^p - 1 + 1)\mathrm{mod}(2^p - 1)$$

$$\Rightarrow 2^p\mathrm{modm} = 1$$

Let it hold for k.
We will prove it holds for $k + 1$:

$$2^{(k+1)p}\mathrm{modm} = 2^{kp+p}\mathrm{modm}$$

$$\Rightarrow 2^{(k+1)p}\mathrm{modm} = (2^{kp}\mathrm{modm}2^p\mathrm{modm})\mathrm{modm}$$

$$\Rightarrow 2^{(k+1)p}\mathrm{modm} = (1\ 1)\mathrm{modm}$$

$$\Rightarrow 2^{(k+1)p}\mathrm{modm} = 1\mathrm{modm}$$

$$\Rightarrow 2^{(k+1)p}\mathrm{modm} = 1$$

Using this fact,

$$(1) \Rightarrow h(K) = [A_0\mathrm{modm}+(\mathrm{A_1modm\ 1})\mathrm{modm}+\ldots+(\mathrm{A_{k-1}modm\ 1})\mathrm{modm}]\mathrm{modm}$$

$$\Rightarrow h(K) = (A_0 \text{modm} + \text{A}_1 \text{mod} + \ldots + \text{A}_{\text{k}-1} \text{modm}) \text{modm}$$

$$\Rightarrow h(K) = (A_0 + A_1 + \ldots + A_{k-1}) \text{modm}$$

From this we conclude that any 2 strings which are permutation of each other hash into the same position.