Handin 5

Thomas Vinther & Jens Kristian Refsgaard Nielsen

09-10-18

1 Radix trees

Given a set $S = \{s_j | j = 1 \dots m : s_j \text{ bit string of length } n_j\}$, denote by n the sum of the lengths of the elements from S i.e.

$$\sum_{j=1}^{m} n_j = n \tag{1.1}$$

We wish to sort the elements of S by using the standard lexicographical sort, defined by:

$$a_0 a_1 \dots a_p \leq b_0 b_1 \dots b_q \iff \tag{1.2}$$
$$(\exists j : 0 \leq j \leq \min(p,q) : a_i = b_i \text{ for } i = 0 \dots j-1 \text{ and } a_j < b_j) \vee \tag{1.3}$$
$$(p < q \text{ and } a_i = b_i \text{ for } i = 0 \dots p) \tag{1.4}$$

1.1 Building the tree

We construct the tree using the following algorithms.

Time	Pseudocode
	Build-Radix-Tree(S)
1	T.root.key = false
m	for $s_j \in$ S
$n_j + 1$	InsertRT(s_j,T,T.root)
	InsertRT($a_0 a_1 \dots a_p$,T,x)
1	if $a_0 = 0$
1	if x.left = nil
1	x.left.key = false
p+1	InsertRT($a_1 \dots a_p$,T,x.left)
1	if $a_0 = 1$
1	if x.right = nil
1	x.right.key = false
p+1	InsertRT($a_1 \dots a_p$,T,x.left)
1	if $a_0 =$ nil
1	x.key = true

Correctness: In accordance with the example in the book we take an element of our set S and finds its correct position by going left if the cipher is a 0 or right if it is a 1, and recursively calling the function without the "first"cipher. In this manner we will clearly reach the correct position of our element. Along the path we ensure that all the nodes are instanciated with the

lines 2,3,6 and 7. When we run out of ciphers the algorithm puts a true value in the node and stops.
Time: Note that for $s_j \in S$ the length n_j is equal to the height of s_j in our Radix tree. Consider firstly InsertRT$(s_j, T, T.root)$ For each passed node we make up to 4 calculations, so we have $T(s_j) \leq c_u n_j$ but on the other hand we always go all the way down, with no reuse of previous work, so $c_l n_j \leq T(s_j)$ for some small constants $c_l \leq c_u$, in total $T(s_j) = \Theta(n_j)$. Now for Build-Radix-Tree(S) we get

$$T(S) = \sum_{j=1}^{m} T(s_j) = \sum_{j=1}^{m} \Theta(n_j) = \Theta\left(\sum_{j=1}^{m} n_j\right) \overset{(1.1)}{=} \Theta(n) \tag{1.5}$$

As wanted.

1.2 Sorting the tree

To print our radix tree in sorted order we use a modified preorder treewalk algorithm.

Time	Line nr	Pseudocode
	0	preorderRTwalk(x,$a_0a_1 \dots a_k$)
1	1	if $x \neq$ nil
1	2	if x.key
1	3	print $a_0a_1 \dots a_k$
1	4	preorderRTwalk(x.left . $a_0a_1 \dots a_k0$)
1	5	preorderRTwalk(x.right , $a_0a_1 \dots a_k1$)

Correctness: From the definiton of lexicographical: (1.3) gives each node is lexicographically less than all the nodes in its subtree. (1.4) gives that each node in the left subtree is lexicographically less than all the nodes in the right subtree.
Using this argument recursively gives us a preoorder walk pattern. Which is exactly what we do in the algorithm preorderRTwalk. In the initial call we use the bit string m to keep track of our position, adding 0 when we go to the left and a 1 when going to the right, only printing when encountering a node which contains *true*, a.k.a. $m \in S$ Our algorithm will of course end when the algorithm has visited every node.
Time: As all $s_j \in S$ hav depth n_j and since $\sum_{j=1}^{m} n_j = n$. We have an "absolute" maximum of n nodes. *Theorem* 12.1 (page 288 CLRS) then gives us that our initial call starting from the root, takes O(n) time. *Theorem* 12.1 only gives this for the Inorder-Tree-Walk, but as we only have added one if-sentence and changed the order of the print, this will only change the time by a constant, which still gives a running time of $\Theta(n) = O(n) \cap \Omega(n)$.

1.3 Conclusion

Bringing it all together we start out by forming a Radix Tree with boolean keys with the Build-Radix-Tree(S) algorithm in $\Theta(n)$ time, then we use our preorder Radix Tree walk on the root with the empty bit string preorderRTwalk(T.root,"") and we get a correct print of the lexicographically sorted elements of S. This procedure takes $\Theta(n) + O(n) = \Theta(n)$ time, as wanted. $\square$