

1. Show how to sort S in $\Theta(n)$ time using a radix tree.

Insert each string in S into a radix tree in the following way: for each string $s \in S$, examine each symbol. Begin with a pointer to the root of the radix tree. If the symbol matches any the current node's children, move the pointer to that child and advance to the next symbol in s . Otherwise create a new child node of the current node, store the symbol in it, move the pointer to it, and advance to the next symbol in s .

To sort S , walk the tree in order, storing a stack of symbols from each node visited. Whenever a terminal node is reached, print the stack as a single string. The resulting output is the strings in S , in lexicographically sorted order.

The algorithm visits a node twice for each symbol, once during insertion and once during printing. (The two steps could be collapsed, but it doesn't affect the running time.) Thus the running time is $\Theta(n)$.

This doesn't obey the lower bound for sorting algorithms proved in class because it isn't a comparison sort. The running time is dependent on the combined length of the input strings, not on the number of strings.

2. Design a hash table data structure in which m changes in response to fluctuations in n .

Keep two hash tables, sizes m and $2m$, while $m < n \leq 2m$.

At an increment operation at which n becomes $> m$, we say an "increasing phase" begins. At a deletion operation at which n becomes $\leq 2m$, we say a "decreasing phase" begins. The phase ends when n passes out of the range $m < n \leq 2m$.

We will just explain how the increasing phase works; a decreasing phase is similar. At the beginning of an increasing phase, we have a table of size $m/2$, which we throw away, a table of size m , which we retain, and we create a new table of size $2m$. We have to initialize the new table (by filling it with zeros), and then then copy the information from the table of size m into the new table. Both of these would take time $O(m) = O(n)$ if we did them all at once, which would not satisfy our objective of making every operation $O(1)$. So we take advantage of the fact that there will be at least $O(m)$ operations before it is possible that the increasing phase ends with $n > 2m$. If the increasing phase ends because $n \leq m$, we will throw away the table of size $2m$, so if it is not finished it does not matter.

At every operation on the table (insert, delete, search), we do the actual operation itself, and then we do some work on our project of filling the table of size $2m$. For the first $m/2$ operations in the increasing phase, we do the operation on the table of size m alone, and then we initialize four positions in the table of size $2m$. After these $m/2$ operations, the new big table is completely initialized. The expected time per operation remains $O(1)$. For the next $m/2$ operations in the increasing phase, we take two addresses in the table of size m , and re-hash each of the elements stored at the two addresses into the table of size $2m$. Since the expected number of items per address is $O(1)$, the expected time per operation remains $O(1)$. During these $m/2$ operations, deletions and insertions are performed on both of the tables.

3. Non-prime p in hash functions.

3a. Show a non-prime p fails to permute.

The following table shows some values of $g(x)$ for different values of a and b :

	$a = 1, b = 0$	$a = 2, b = 0$	$a = 3, b = 0$
x	$g(x)$	$g(x)$	$g(x)$
0	0	0	0
1	1	2	3
2	2	4	0
3	3	0	3
4	4	2	0
5	5	4	3

Some values of a produce permutations, others don't.

3b. Which propositions fail for non-prime p ?

The following propositions fail for non-prime p :

ii) If $(a(x - y) \bmod p) = 0$, then either $a = 0$ or $(x - y) = 0$

iii) If $((ax + b) \bmod p) = ((ay + b) \bmod p)$ then $x = y$