Karim Maamari

Exercises

1.

a. No such red-black tree



2.

**(a)**

The variable d refers to the maximum length of the words in W, including the padding spaces. In this case, d is equal to the length of the longest word in W plus the number of padding spaces needed to make all words the same length. Thus, d = 6 (for the word "jumps", which is the longest word in W) + 1 = 7.

The variable n refers to the number of elements in the list W. In this case, n = 9.

The variable r refers to the base used for the radix sort. Since we are sorting words in lexicographical order, we need to use the number of distinct characters in the alphabet as the base. In this case, we can assume that the alphabet includes 27 characters (26 letters and the space character). Thus, r = 27.

**(b)**

The variable d is the length of the longest binary string after the conversion. In this case, the longest word in W is "jumps" with a length of 5, so each binary string will be 5 characters long, padded with 3 zeros to make them all the same length. Therefore, d = 5 x 8 = 40.

The variable n is the number of elements to be sorted, which is the length of the list W. In this case, there are 9 elements in W, so n = 9.

The variable r is the base of the radix sort, which is the number of distinct characters that can appear in each position of the string. Since each character in the binary string can be any of 2 values (0 or 1), r = 2.

Therefore, for the ASCII representation of W, the values for d, n, and r are d = 40, n = 9, and r = 2.

**(c)**

To represent the word "jumps" with its publication date (November 19, 1562) so that it can be sorted correctly by radix sort, we can prepend the date in the format YYYYMMDD to the string. So, the resulting string would be "15621119jumps".

**(d)**

The time complexity of radix sort on the list V with all words converted to their 0-padded binary strings would be O(nd), where n is the number of words in the list V and d is the length of the longest binary string (after padding).

Since each binary string has the same length (equal to the length of the longest binary string), each pass of radix sort will take O(n) time, as there are n binary strings to be sorted in each pass. Since there are d passes required to sort the strings based on their binary representations, the total runtime will be O(nd).

The reason for this is that radix sort sorts the elements by their binary representation, one digit at a time, from the least significant digit to the most significant digit. Since all binary strings have the same length, there are d digits to sort. In each pass, radix sort sorts the elements based on a single digit of their binary representation, using a stable sorting algorithm such as bucket sort. Since there are n elements to be sorted in each pass, the pass takes O(n) time. Therefore, the total time complexity of radix sort on this list is O(nd).

**(e)**

When representing each word in the vocabulary with a one-hot vector, the value of r in radix sort becomes equal to the size of the vocabulary, which is n. Since each vector has length n, the value of d in radix sort becomes n. The number of elements being sorted is also n. Therefore, the time complexity of radix sort on the list V with one-hot vector representation is O(n^2), since the bucket sort step takes O(n) time and we perform it n times (once for each digit in the vector).

Problems

3.

The lower bound theorem for comparison-based sorting states that any comparison-based sorting algorithm requires at least O (n log n) comparisons in the worst case. To build a binary search tree with n nodes, we need to compare each node to all the nodes that came before it, which takes at least n-1 comparisons.

Therefore, building a binary search tree requires at least O(n) comparisons. Assuming we have a binary search tree algorithm that can perform INSERT operations in O (√ log n) time using a comparison-based algorithm, we could build a binary search tree with n nodes in O (n √ (log n)) time by inserting nodes one by one. However, this contradicts the lower bound of O(n) for building a binary search tree.

So, we can conclude that it is impossible to design a binary search tree that performs INSERT operations in O (√ log n) time using a comparison-based algorithm.

4.

**(a)** Pseudocode for the algorithm

1. Set left to 0 and right to n-1.

2. While left <= right:

a. Set mid to the floor of (left + right) / 2.

b. Compare the mid duck to the stick using compareToStick(mid).

c. If the mid duck is taller than the stick, set right to mid - 1.

d. If the mid duck is shorter than the stick, set left to mid + 1.

e. If the mid duck is the same height as the stick, return mid.

3. If we reach here, there is no duck the same height as the stick, return "No such duck".

First, we start by looking at the middle duck in the line and you compare its height to the stick using the compareToStick function. If the duck is taller than the stick, we know that any ducks to the right of it will also be taller, so we ignore them and only look at the ducks to the left of the current duck. If the duck is shorter than the stick, we know that any ducks to the left of it will also be shorter, so we ignore them and only look at the ducks to the right of the current duck. We repeat this process of looking at the middle duck and updating your search range until we either find a duck the same height as the stick or we've searched through the entire line without finding a match. If you find a duck the same height as the stick, we can tell its position in the line. If you don't find a matching duck, we can report that there is no such duck in the line. This algorithm takes O(log(n)) comparisons because it halves the search range at each step, which makes it very efficient for finding a matching duck in a large line of ducks.

**(b)** Suppose we have an algorithm that can find a duck of the same height as the stick using fewer than log(n) comparisons. Since we can only remember a constant number of ducks at any given time, the algorithm must miss comparing the stick to some ducks. If the algorithm misses comparing the stick to a duck, it can only eliminate one half of the remaining ducks as a possibility for the duck of the same height as the stick. If the duck of the same height as the stick is in the other half, the algorithm cannot find it without comparing the stick to at least one duck in that half. Therefore, the algorithm must use at least one comparison for each duck to find the duck of the same height as the stick. Since there are n ducks in total, the algorithm must use at least log(n) comparisons in total.

**(c)** We can use the element distinctness problem as an example to show that the goose's claims are false. This problem involves determining whether all the elements in a given list are distinct from each other. Assume we have n elements in the list, and we want to solve the element distinctness problem using gooseTree. We can insert all n elements into the gooseTree using gooseInsert in O(n) time. Then, for each element in the list, we can search for it in the gooseTree using gooseSearch in O(log(n)) time. If we find an element more than once, we know that the list contains duplicates and we can return "false". Otherwise, we can return "true" to indicate that all elements in the list are distinct. The lower bound for the element distinctness problem is Ω(nlog(n)) by comparison-based sorting. Therefore, any comparison-based algorithm for the element distinctness problem must take Ω(nlog(n)) time in the worst case. Since our algorithm for the element distinctness problem using gooseTree takes O(nlog(n)) time (n calls to gooseSearch, each taking O(log(n)) time), it cannot be optimal. This contradicts the goose's claim that gooseTree is better than red-black trees, which can solve the element distinctness problem optimally in O(nlog(n)) time.

Therefore, we have shown that the goose's claims are false, and gooseTree cannot be better than red-black trees in terms of worst-case time complexity.

5.

**(a)**

Using the decision tree method, each leaf of the decision tree represents a possible output sequence of elements, and each internal node represents a comparison of two elements.

The number of leaves in the decision tree is the number of possible ways to merge the two lists, which is given by the binomial coefficient (m+n choose n) or (m+n choose m), depending on how we choose to arrange the elements. This simplifies to (m+n)! / (m! \* n!).

(m+n choose n) = (m+n)! / (n! \* (m+n-n)!) = (m+n)! / (n! \* m!)

**(b)**

Let h be the height of the decision tree. Since each internal node has two children, the decision tree has at most 2 h leaves. Since there are (m+n choose n) possible ways of merging two sorted lists of length m and n, we have: 2h >= (m+n choose n)

1.Taking the logarithm base 2 of both sides, we get:

h >= log2((m+n choose n))

We have shown that (m+n choose n) = (m+n)! / (m! \* n!)

2.Substituting this into the previous inequality, we get:

h >= log2((m+n)! / (m! \* n!))

3. Applying Stirling's approximation, we get:

h >= (m+n) log2(m+n) - m log2(m) - n log2(n) + O(log(m) + log(n))

Since the decision tree has exactly (m+n-1) internal nodes, the number of comparisons required by any comparison-based merging algorithm is at least the height of the decision tree, i.e., Ω (n log (1 + m/n)).