1  7.2.5

Problem
Suppose that the splits at every level of quicksort are in the proportion $1 - \alpha$ to $\alpha$, where $0 < \alpha \leq 1/2$ is a constant. Show that the minimum depth of a leaf in the recursion tree is approximately $-\frac{\ln n}{\ln(\alpha)}$ and the maximum depth is approximately $-\frac{\ln n}{\ln(1-\alpha)}$. (Don’t worry about integer round-off.)

Solution
The minimum depth occurs for the path that always takes the smaller portion of the split, i.e., the nodes that take $\alpha$ proportion of work from the parent node. The first node in the path (after the root) gets $\alpha$ proportion of work (the size of data processed by this node is $\alpha n$), the second one gets $\alpha^2$ so on. The recursion bottoms out when the size of data becomes 1. Assuming the recursion ends at level $m$, we have

$$\alpha^m n = 1$$

or

$$m = \frac{\ln(1/\alpha)}{\ln(\alpha)}$$

Similar argument can be used for showing that the maximum depth is $-\frac{\ln n}{\ln(1-\alpha)}$.

2  8.3.3

Problem
Use induction to prove that radix sort works. Where does your proof need the assumption that the intermediate sort is stable?
Solution

We induce on $d$, the number of digits.

Basis If $d = 1$, then the input numbers have just 1 digit, so sorting on that digit will sort the entire array.

Induction step We use weak induction for this problem. The induction hypothesis is that radix sort works correctly and stably for all numbers with $d - 1$ digits. We need to show that it works for $d$-digit numbers. Note that Radix sort, operates separately on each digit, so we can consider radix sort of $d$ digit numbers as sort on $d - 1$ digit numbers followed by a sort on the $d^{th}$ digit. By the induction hypothesis, just before the sort of $d^{th}$, the numbers are in correct order according to the lower $d - 1$ digits.

Now say we are comparing the $d^{th}$ digits of two numbers $a$ and $b$. We write the specific digits as $a_d$ and $b_d$.

if $a_d < b_d$, then we place number $a$ before number $b$ and this is the proper order irrespective of the lower order digits.

if $a_d > b_d$, then we place number $a$ after number $b$.

if $a_d = b_d$, the sorting algorithm being stable, leaves the numbers in the order they were before. Since the $i^{th}$ digits are same, the order of the numbers of determined by the lower $i - 1$ digits. By the induction hypothesis that sort on $d - 1$ digit numbers works correctly, the numbers are in the correct order.

Loop invariant approach

The loop invariant for the radix sort is the following. At the beginning of iteration $i$, the numbers are correctly and stably sorted according to the lower $i - 1$ digits.

Initialization: Initially $i = 1$ and numbers are not sorted yet, so they are indeed sorted according to $1 - 1 = 0$ digits.

Maintainence: Assume that we are at the $i^{th}$ iteration and that numbers are sorted correctly according to last $i - 1$ digits. In the $i^{th}$ iteration, we use a stable sorting algorithm to sort the numbers based on the $i^{th}$ digit. (The argument here is similar to the induction step in the above solution). Say that we are comparing two numbers $a$ and $b$. Since the we sort on $i^{th}$ digit, we are actually comparing the $i^{th}$ digits, which we denote by $a_i$ and $b_i$.

if $a_i < b_i$, then we place number $a$ before number $b$ and this is the proper order irrespective of the lower order digits.

if $a_i > b_i$, then we place number $a$ after number $b$. 
if $a_i = b_i$, the sorting algorithm being stable, leaves the numbers in the order they were before. Since the $i^{th}$ digits are same, the order of the numbers of determined by the lower $i-1$ digits. Since the loop invariant is satisfied at the beginning of the iteration, the numbers are in the correct order.

Termination: When we are at the beginning of $d^{th}$ iteration, our loop invariant holds for last $d-1$ digits. When sort on the $d^{th}$ we maintain the correct order of numbers based on all the $d$ digits.

3 8.4-2

Problem

Explain why the worst-case running time for bucket sort is $O(n^2)$. What simple change to the algorithm preserves its linear average-case running time and makes its worst-case running time $O(n\log n)$?

Solution

The worst case for bucket sort occurs when all inputs falls into single bucket, for example. Since we use insertion sort for sorting buckets and insertion sort has a worst case of $O(n^2)$, the worst case runtime for bucket sort is $O(n^2)$.

By using an algorithm with worst case runtime of $O(n\log n)$ instead of insertion sort for sorting buckets, we can ensure that worst case is $O(n\log n)$ without affecting the average case behavior.

To see that the worst case is still $O(n\log n)$, lets consider a case where $n$ data are distributed among two buckets, $a$ elements in one bucket and $n-a$ in the other. Since we use $O(n\log n)$ sorting algorithm in each bucket, the run time for each sort is, $k\log(a) + c_2$ and $k(n-a)\log(n-a) + c_2$, where $k, c_2$ are positive constants. The total runtime is $k\log(a)+k(n-a)\log(n-a)+2c_2$. This quantity attains its maximum value when $a = 0$ or $a = n$ and the maximum value is $kn\log n + 2c_2$. Thus total runtime is still $O(n\log n)$. It is clear from this that maximum running cost occurs when data are in single bucket instead of spread in two buckets. Extending this argument, we can see that worst case for the hash table occurs when all inputs hash into the same bucket. (We also note that the expressions obtained are basically convex combinations of $n\log n$ which is a convex function and then jensen’s rule can be applied to arrive at the same conclusion).

4 11.1-4

Problem

We wish to implement a dictionary by using direct addressing on a huge array. At the start, the array entries may contain garbage, and initializing the entire
array is impractical because of its size. Describe a scheme for implementing a
direct-address dictionary on a huge array. Each stored object should use \(O(1)\)
space; the operations SEARCH, INSERT, and DELETE should take \(O(1)\) time
each; and initializing the data structure should take \(O(1)\) time. (Hint: Use an
additional array, treated somewhat like a stack whose size is the number of keys
actually stored in the dictionary, to help determine whether a given entry in the
huge array is valid or not.)

Solution
We use a huge array \(T\) and an verifier array \(V\) together. While the array \(V\) could
grow up to maximum size of array \(T\), the size changes dynamically as keys are
added/deleted. When array \(T\) is allocated, no attempt is made to initialize the
entries. Instead we use the verifier array \(V\) to verify the entries in \(T\) as follows.

Let \(n_V\) be the number of elements in array \(V\), (which is same as number of
keys in \(T\)). Let the first entry be at index 1. When adding a new object \(x\) with
key \(k_{\text{new}}\) to \(T\), we add a reference to the object to \(V\), at index \(j = n_V + 1\).
Then we add the object \(x\) to the array \(T\) at location \(x.\text{key}\), i.e., \(T[x.\text{key}] = x\)
and \(x.\text{verifier} = j\). Note that we assume existence of a field \text{verifier}\ in
object \(x\). It is important to check that entry in \(T\) and the corresponding entry
in verifier \(V\), reference each other and this cycle of reference provides us with
required verification we need. When this verification is successful, we know we
are dealing with a legitimate entry.

When we lookup an object with key \(k\), we check for this verification \((T[k] = x, x.\text{verifier} = j \text{ and } V[j] = x \text{ with } x.\text{key} = k)\) and only when the
verification is successful, we obtain the object.

Deletion, of key \(k\) for example, is a bit tricky. We need to perform three
tasks. First is to break the verification cycle. This is pretty trivial, We just need
to set the \(T[k] = 0\). However the entry \(j\) correponding to key \(k\) in \(V\) is still
there. We cant just set the value to 0, as it would leave an empty space in \(V\).
We fix this problem by exchanging this entry with the last entry, i.e., exchange
\(V[j]\) with \(V[n_V]\). Finally we need to fix the verification cycle for the object
we just moved into index \(j\). Note that all this is constant time effort.

The three key operations of a dictionary are implemented as follows.

// insert object x, with key x.key into the dictionary T
Insert(T, x) {
    n_V = n_V + 1
    // adding the up reference to the array T
    V[n_V] = ptr to x
    T[x.key] = x
    // add a down reference to the verifier. we assume a field in object x
    x.verifer = n_V
}

// return the object corresponding to the key k. If none found return null.
Search(T, k) {
    x = T[k]
    j = x.verifier
    if ( j < 1 or j > n_V )
        return null
    obj = V[j]
    // verify that verifier points to the same object
    if ( obj.key == k )
        return obj
    else
        return null
}

// delete the key 'k' from dictionary T
Delete(T, k) {
    x = T[k]
    // reset the entry in T
    T[k] = 0
    last = n_V
    lastObj = T[last]
    // delete the entry in V..
    exchange V[j] <-> V[last]
    n_V = n_V - 1
    // fix the verifier for the lastObj
    lastObj.verifier = j
}

5 Lexicographic sort

Problem
Given n integer pairs (r_i, c_i), sort them in lexicographic order:
(r_i, c_i) <= (r_j, c_j) if r_i < r_j or ( r_i == r_j and c_i <= c_j )
Give a high-level description of your algorithms. What is its running time?

Solution
Use a sorting algorithm to sort first based on c_i. Then use a stable sorting
algorithm to sort output from first sort, this time using r_i as keys. Note that
the integer pair is treated as a single entity, though we use c_i or r_i as keys.
Using an $O(n \log n)$ algorithms for both sorts, the run time first sort is $kn\log n$ and the runtime for second sort is $kn\log n$ since both use the same algorithm.
Adding up, we have the total runtime $2kn\log n$, which is still $O(n \log n)$.

Note that counting sort cannot be used here as there is no information on
the range of the numbers.
Solution 2

You can also use any comparison sorting algorithm, by using extended definition for the key comparison. We use the following key comparison method for ordering the numbers.

\[
\begin{align*}
\text{if } & \ r_i == r_j \text{ and } c_i == c_j \\
& \text{then} \\
& \quad \text{both numbers are same} \\
\text{if } & \ r_i < r_j \text{ or } (r_i == r_j \text{ and } c_i <= c_j) \\
& \text{then} \\
& \quad \text{first pair is smaller} \\
\text{else} \\
& \quad \text{first pair is bigger}
\end{align*}
\]

You can extend both techniques to comparison on k-tuples. For the first method, we make k passes using stable sorting $O(n \log n)$ algorithm for a total runtime of $O(kn \log n)$. For the second method, the comparison method is $O(k)$ and since we make $O(n \log n)$ comparisons, we get $O(kn \log n)$ for the runtime.