Chapter 6

Binary Search Trees and Linear Time Sorts.

6.1 Linear Time Sorts

Counting sort. Counting sort assumes that each of the n input elements is an integer at the size at most k. It runs in $\Theta(n+k)$ time, so that when $k = O(n)$, counting sort runs in $\Theta(n)$ time. Counting sort first determines, for each input element x, the number of elements less than or equal to x. It then uses this information to place element x directly into its position in the output array. For example, if 17 elements are less than or equal to x, then x belongs in output position 17. We must modify this scheme slightly to handle the situation in which several elements have the **same value**, since we do not want them all to end up in the same position.

let B and C be new arrays at size n and k
for $i \in [0, k]$ **do**
    $C_i \leftarrow 0$
end
for $j \leftarrow [1, n]$ **do**
    $C_{Aj} \leftarrow C_{A_j} + 1$
end
for $i \in [1, k]$ **do**
    $C_i \leftarrow C_i + C_{i-1}$
end
for $i \in [n, 1]$ **do**
    $B_{C_{A_j}} \leftarrow A_j$
    $C_{A_j} \leftarrow C_{Aj}-1$ // to handle duplicate values
end
return B

Notice that the Counting sort can beat the lower bound of $\Omega(n \log n)$ only because it is not a comparison sort. In fact, no comparisons between input elements occur anywhere in the code. Instead, counting sort uses the actual values of the elements to index into an array.

An important property of the counting sort is that it is **stable**.

Stable Sort.

We will say that a sorting algorithm is stable if elements with the same value appear in the output array in the same order as they do in the input array.

Counting sort's stability is important for another reason: counting sort is often used as a subroutine in radix sort. As we shall see in the next section, in order for radix sort to work correctly, counting sort must be stable.

Radix sort Radix sort is the algorithm used by the card-sorting machines you now find only in computer museums. The cards have 80 columns, and in each column, a machine can punch a hole in one of 12 places. The sorter can be mechanically "programmed" to examine a given column of each card in a deck and distribute the card into one of 12 bins depending on which place has been punched. An operator can then gather the cards bin by bin, so that cards with the first place punched are on top of cards with the second place punched, and so on.

The Radix-sort procedure assumes that each element in the array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

for $i \in [1, d]$ **do**
  use a stable sort to sort array A on digit i
end

Correctness Proof. By induction on the column being sorted.

- Base. Where $d = 1$, the correctness follows immediately from the correctness of our base sort subroutine.

- Induction Assumption. Assume that Radix-sort is correct for any array of numbers containing at most $d - 1$ digits.

- Step. Let A' be the algorithm output. Consider $x, y \in A$. Assume without losing generality that $x > y$. Denote by x_d, y_d their d-digit and by $x_{/d}, y_{/d}$ the numbers obtained by taking only the first $d - 1$ digits of x, y. Separate in two cases:

  - If $x_d > y_d$ then a scenario in which x appear prior to y is imply contradiction to the correctness of our subroutine.
  - So consider the case in which $x_d = y_d$. In that case, it must hold that $x_{/d} > y_{/d}$. Then the appearance of x prior to y either contradicts the assumption that the base algorithm we have used is stable or that x appears before y at the end of the $d - 1$ iteration. Which contradicts the induction assumption.

The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit lies in the range 0 to k–1 (so that it can take on k possible values), and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes $\Theta(n + k)$ time. There are d passes, and so the total time for radix sort is $\Theta\left(d(n + k)\right)$.

Bucket sort. Bucket sort divides the interval [0, 1) into n equal-sized subintervals, or buckets, and then distributes the n input numbers into the buckets. Since the inputs are uniformly and independently distributed over [0, 1), we do not expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

let B[0 : n − 1] be a new array **for** $i \leftarrow [0, n–1]$ **do**
    make B_i an empty list
end
for $i \leftarrow [1, n]$ **do**
    insert A_i into list $B_{\lfloor nA_i \rfloor}$]
end
for $i \leftarrow [0, n–1]$ **do**
    sort list B_i
end
concatenate the lists $B_0, B1, .., B_{n–1}$ together and
return the concatenated lists