

Lecture 9

Digit-based sorting and data structures

9.1 Overview

In this short lecture we examine digit-based sorting and digit-based data structures. Our previous data structures treated keys as abstract objects that could only be examined via comparisons. The methods discussed in this lecture will instead treat them as a sequence of digits or a sequence of characters. Material in this lecture includes:

- Radix sort: a method for sorting strings or integers.
- Tries: a data structure that can be viewed as an on-the-fly version of radix sort.

9.2 Introduction

So far, we have looked at sorting and storing items whose keys can be anything at all, so long as we have a notion of “less than”. Today we will look at methods for the case that keys are natural objects like strings or integers.

To start, say we have n objects to sort whose keys are all integers in a small range: $1, \dots, r$ (e.g., say I have a stack of homeworks to sort by section). In this case, what would be a fast way to sort? One easy method is bucket-sort:

Bucket-sort:

- Make an array A of size r , of “buckets” (perhaps implemented as linked lists).
- Make one pass through the n objects. Insert object with key k into the k th bucket $A[k]$.
- Finally, make a pass through the array of buckets, concatenating as you go.

The first step of bucket-sort takes time $O(r)$, the second step takes time $O(n)$, and the last step takes time $O(r)$ or $O(r+n)$ depending on how the buckets are implemented. In any case, the total time is $O(n+r)$.

So, bucket-sort is linear time if $r = O(n)$. Notice one thing interesting about it is that by using the magic of indirect addressing, we are making an r -way decision at each step (something you can't do in the comparison model). Unfortunately, bucket-sort is only good if the range of keys is small. This leads us to our next method, radix-sort.

9.3 Radix Sort

Suppose our keys are numbers, or strings, that can be viewed as a *sequence* of digits (or characters) each in a range of size r . For example, we might have $r = 10, 26,$ or 128 . In that case, there is a natural sorting algorithm called Radix Sort we can use to sort them. (“radix” is the base r). Actually there are two versions: one that's conceptually easier called Most-Significant-First radix sort, where we go top-down digit by digit (or byte by byte), and another that's trickier to think of but easy to code called Least-Significant-First radix sort where we go in the other direction.

9.3.1 Most-significant-first (MSF) radix sort

Most-significant-first radix sort begins by sorting keys into buckets according to their most significant character or digit. For instance, if we are sorting strings, we would get a bucket for 'a', a bucket for 'b', a bucket for 'c' and so on. So, now the strings are roughly sorted in that any two strings that begin with different letters are in the correct order.¹ Now we just recursively sort each bucket that has more than one element using the same procedure (sorting into buckets by the next most significant character and so on) and then concatenate the buckets.

If we ignore time spent scanning empty buckets, then to sort n strings we just spend $O(n)$ time at each level. So, if strings are length L , then the total time is $O(nL)$. The time spent scanning empty buckets could be a problem if r is large, but if we view r as a constant, then just goes into the $O()$.

9.3.2 Least-significant-first (LSF) radix sort

Here is another idea that is easier to implement but trickier to see why it works. Let's switch from strings to numbers since that will make the method a little cleaner.

In this algorithm, we first perform a bucketsort using only the *least* significant digit. That is, we place the keys into buckets according to the ones digit and then concatenate the buckets. Next we bucketsort by the tens digit, then the hundreds and so on. This sounds weird, but the claim is that if we perform each bucketsort in a *stable* manner that doesn't rearrange the order of items that go into the same bucket (a sorting method is called **stable** if it doesn't rearrange equal keys) then this will end up correctly sorting the items. Let's see what happens on an example:

Example: Suppose our input is [28, 15, 24, 19, 13, 22].

- We first sort by the ones digit, producing: [22, 13, 24, 15, 28, 19].
- Now we sort by the tens digit using a stable bucketsort (so the relative order of items with the same tens digit remains unchanged), producing: [13, 15, 19, 22, 24, 28].

¹If we are sorting numbers, we need to pad to the left with zeros to have the correct semantics.

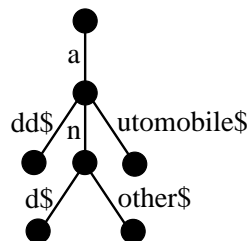
(In terms of implementation, each node has an array of child pointers, so when the picture shows an edge having a letter, like 'c', this really means that the child is the “cth” child.)

The time to do a search is only $O(\text{length of key})$. The same is true for doing an insert, if we view r as a constant. (If r is really big then we should worry about the time spent allocating the arrays of child pointers). So, this is really nice. Also, prefix searching is especially easy (e.g., if you wanted to make a text editor that did word-completion). The main drawback is that the overhead is high because you have so much pointer-following to do. E.g., what if we added “automobile” to the above trie? Also, you have a factor of r extra space since you need to have an array of size r in each node.

Since the design is so nice conceptually, people have thought about ways of making it more practical. In particular, some things you can do are:

- Compress paths that don't branch into a single edge.
- Only split when needed.

So, for instance, the set {add, automobile, and, another} would look like:



This is called a “Patricia tree” (practical algorithm to retrieve information coded in alphanumeric).