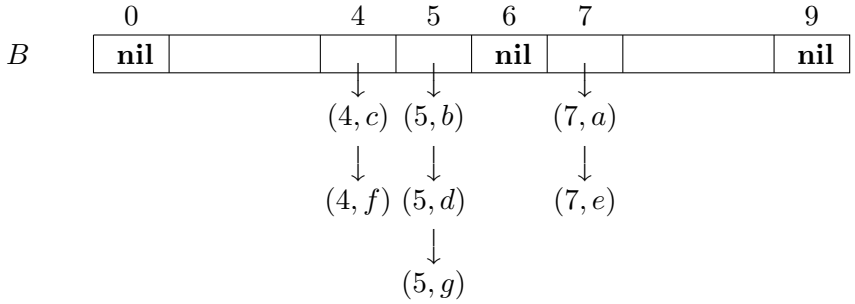
Lecture Notes, Fundamental Algorithms: Radix Sort

Most sorting algorithms are based on comparing items. However, there is an alternate approach if the values of the items lie in a restricted range, say we have n items in the range $[0, m-1]$. Then we can sort in $O(n+m)$ time. This is called Bucket Sort.

Here we suppose the input is provided in a linked list of n items. The algorithm uses an additional array, $B[0, m-1]$ of pointers. Entry $B[i]$ is called bucket i. Each bucket is initially an empty linked list, i.e. $B[i]$ has the value **nil** for all i. We then traverse the list of n items, adding an item with value v to bucket $B[v]$. The item is added at the end of the list. Finally, we concatenate the linked lists $B[0], B[1], B[2], \ldots, B[m-1]$ in this order, which yields a linked list of the items in sorted order. To be able to do these operations in $O(1)$ time per item, each bucket needs to have two pointers: one to the front of the list and one to the end of the list (only the front pointer is shown in the figure below).

Let's illustrate this on the following list of items. Each item consists of an integer key and a second character field. The list follows. $(7,a) \to (5,b) \to (4,c) \to (5,d) \to (7,e) \to (4,f) \to (5,g)$.

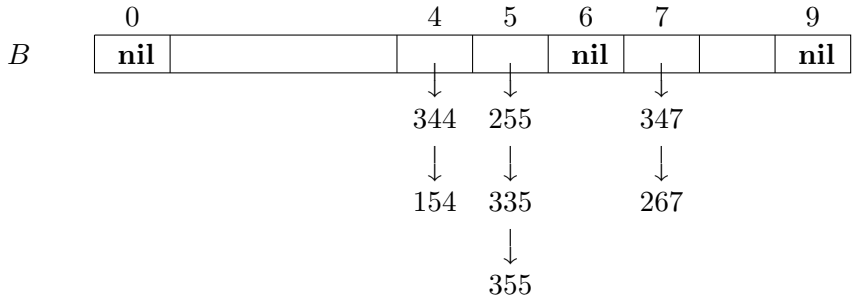
The bucket sort creates the following array $B[0:9]$ of lists (called buckets).



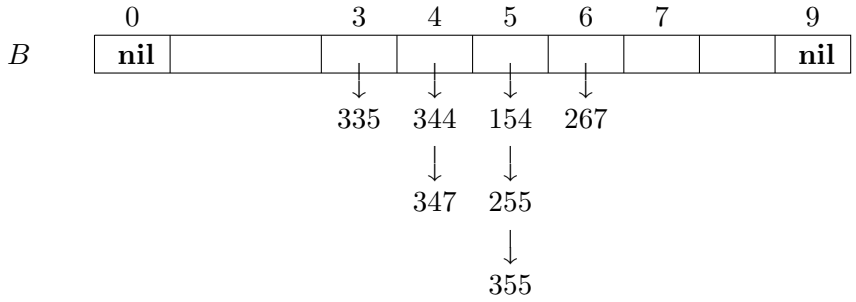
On concatenating the bucket lists, we obtain a sorted list of the items: $(4,c) \to (4,f) \to (5,b) \to (5,d) \to (5,g) \to (7,a) \to (7,e)$.

This algorithm performs $O(n+m)$ work. We need $O(m)$ work to initialize the array of buckets to **nil**, $O(n)$ work to add the items to the buckets, and a further $O(n+m)$ work to traverse and concatenate the m linked lists, some of which may be empty, but we don't know which ones are empty till we look at them.

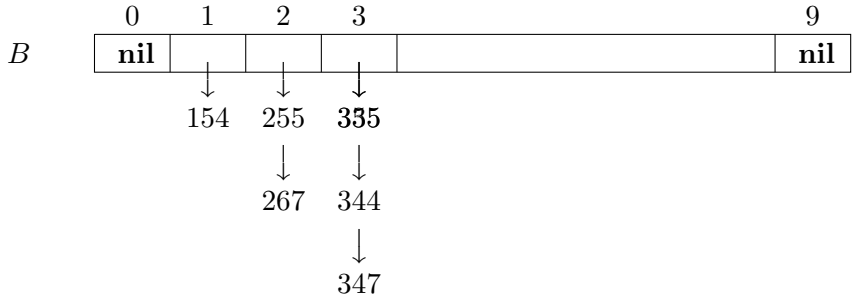
Radix sort uses bucket sort as a subroutine to keep its memory (a.k.a. space) usage down to $O(n)$ words, when $m = n^c$ for some constant c. We will first explain the radix sort algorithm when applied to sorting c-digit decimal numbers. Let the digits in a number be indexed from 1 to d going from the least significant to the most significant digit, e.g. the number 1834 has least significant digit 4 and most significant digit 1. Radix Sort sorts the numbers using d iterations of a 10 bucket Bucket Sort; the first iterations sorts w.r.t digit 1, the second w.r.t digit 2, etc. The linked list that is the output of the i-th sort will be the input to the $(i+1)$-st sort, for $i = 2, 3, \ldots, d-1$. For example given the following list of numbers, $347 \to 255 \to 344 \to 335 \to 267 \to 154 \to 355$, radix sort proceeds as follows. First it sorts on the least significant digit obtaining the following buckets:



This yields the following list, sorted on its least significant digit: $344 \to 154 \to 255 \to 335 \to 355 \to 347 \to 267$. Next, we reset all the buckets to **nil**. Now sorting on the second least significant digits yields the following bucket lists:



This yields the following list, sorted on its two least significant digits: $335 \to 344 \to 347 \to 154 \to 255 \to 355 \to 267$. Again, we reset all the buckets to **nil**. Now sorting on the most significant digit yields the following bucket lists:



The key property is that the output of the i-th Bucket Sort is sorted w.r.t. the i least significant digits. To see this property is maintained by the $(i+1)$-st sort, it suffices to ensure that the linked list ordering in each bucket keeps its items in the same relative order that they had in the input list. To do this, whenever we add an item to a bucket, the item needs to be added to the end of the bucket's list (and not the front).

Radix sort performs c bucket sorts, each of which takes $O(n+m)$ time, for a total of $O(c(n+m)$ time. If we use n buckets, and there are a constant number of "digits", then bucket sort takes $O(n)$ time.

For sufficiently large data sets, in practice, algorithms such as merge sort and quicksort will have better cache performance, which will more than compensate for the additional $\Theta(\log n)$ factor in their runtime.