CS61B Lecture 28

Friday, April 3, 2020

Sorting by Selection

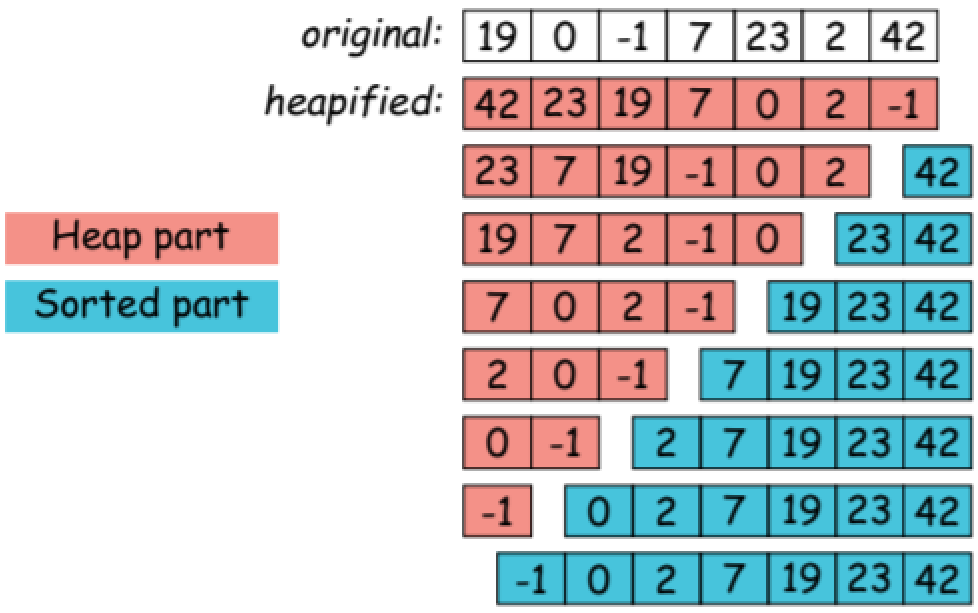
Let's look at another use for priority queues, known as **heapsort**, where we will keep selecting the smallest (or largest) element. This given as

$$
 O(N lg N)
$$

$$O(N lg N)$$

$$O(N lg N)$$

algorithm (N remove-first operations). Since we remove items from the end of the heap, we can use that area to accumulate the result:



Before, we created heaps by insertion to an initially empty heap. If we are given an array of unheaped data to start with, there is a faster procedure, assuming our heap is indexed from 0:

```
void heapify(int[] arr) {
    int N = arr.length;
    for (int k = N / 2; k >= 0; k -= 1) {
        for (int p = k, c = 0; 2*p + 1 < N; p = c) {
            reheapify downward from p;
        }
    }
}
```

At each iteration of the `p` loop, only the element at p might be out of order with respect to its descendants, so re-heap-ifying downward will restore the subtree rooted at p to proper heap ordering.

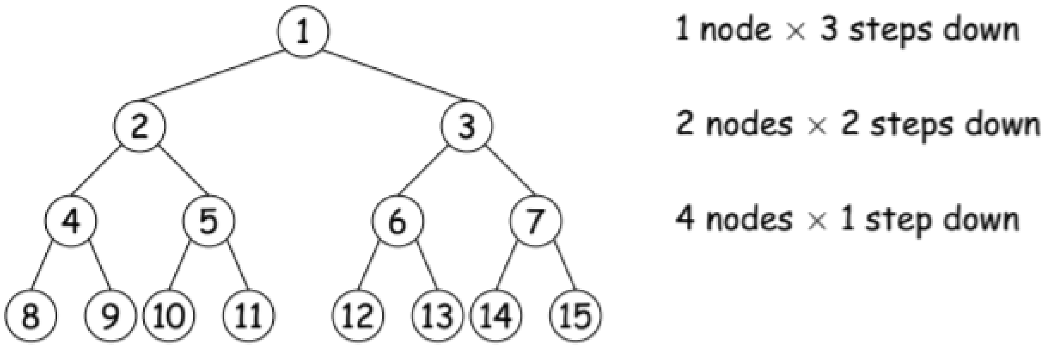
It looks like the procedure for re-inserting an element after the top element of the heap is removed, repeated N/2 times, but instead of being:

$$\Theta(NlgN)$$

it is:

$$\Theta(N)$$

Cost of creating a heap



In general, worst-case cost for a heap with h + 1 levels is:

$$\begin{gathered}2^0 \times h + 2^1 \times (h-1) + \ldots + 2^{h-1} \times 1 \\ = (2^0 + 2^1 + 2^2 + \ldots + 2^{h-1}) + (2^0 + 2^1 + \ldots 2^{h-2} + \ldots + 2^0) \\ = (2^h - 1) + (2^{h-1} - 1) + \ldots (2^1 - 1) \\ = 2^{h+1} - 1 - h \in \Theta(2^h) = \Theta(N)\end{gathered}$$

Alas, since the rest of heapsort still takes

$$\Theta(NlgN)$$

this does not improve its asymptotic cost.

Merge sorting

The idea behind merge sorting is to divide data into 2 equal parts, recursively sort halves, and then merge the results. The algorithmic complexity is:

$$\Theta(NlgN)$$

Mergesort is good for external sorting:

- First break data into small enough chunks to fit in memory and sort.
- We then repeatedly merge bigger and bigger sequences.

We can merge `K` sequences of arbitrary size on secondary storage using

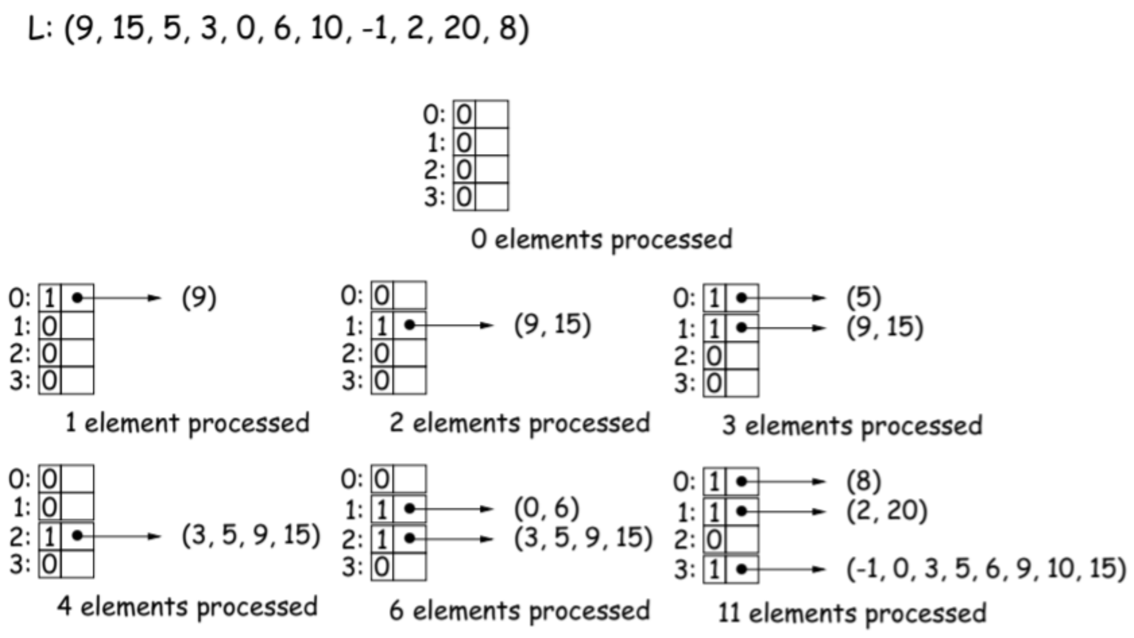
$$\Theta(K)$$

memory complexity:

```
Data[] V = new Data[K];
for (all i; while V is not sorted) {
    // set V[i] to the first data item of sequence i;
    // Find k so that V[k] is the smallest;
    // Output V[k], and read new value into V[k] if present;
}
```

Internal Merge Sort

To perform internal sorting, we can use a binomial comb to orchestrate it:



What is the motivation for doing a binomial comb over an in-place merge sort? Well, as it turns out, if we have two sorted arrays, doing an in-place merge sort is harder than it sounds, and if we were to follow all the steps, it turns out to be quite similar to the procedure of using a binomial comb.

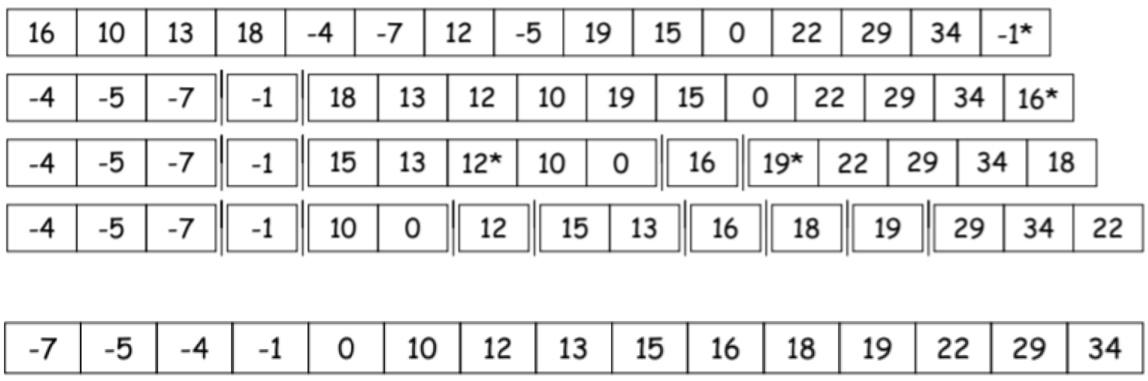
Quicksort: speed through probability

The idea behind quicksort is to partition data into pieces: everything greater than some pivot value at the high end of the sequence, and everything less than or equal to on the low end. We repeat this recursively on the high and low pieces.

For speed, we stop when the pieces are "small enough" and do insertion sort instead. Why? Insertion sort has low constant factors. by design, no item will move out of its piece, so when the pieces are small, the number of inversions is too.

We do however, have to choose our pivot well. One good example would the median of the first, last and middle items of the sequence. A well-chosen pivot will result in a fairly quick sorting algorithm in the usual case (not the worst case, however).

In this example, we continue until the pieces of size less than or equal to 4. The pivots for the next step are starred, and we arrange to move pivot to dividing line each time, and the last step in insertion sort:



Performance

If our choice of pivots is good, we will divide the data in two each time:

$$\Theta(NlgN)$$

with a good constant factor relative to merge or heap sort. If the choice of pivots is bad, most items on one side each time, taking:

$$\Theta(N^2)$$

In the best case, quicksort takes:

$$\Omega(NlgN)$$

thus, insertion sort is better for nearly ordered input sets.

An interesting point: randomly shuffling the data before sorting makes:

$$\Omega(N^2)$$

very unlikely! The probability is quite low of encountering the best case.

Quick Selection

For a given `K`, if we want to find the `K`-th smallest element in the data, how can we do it?

The obvious method would be to sort, select the `K`-th smallest element, which would take:

$$\Theta(NlgN)$$

If `K` is less than or equal to some constant, we could easily do it in linear time by going through the array and keeping the `K` smallest items.

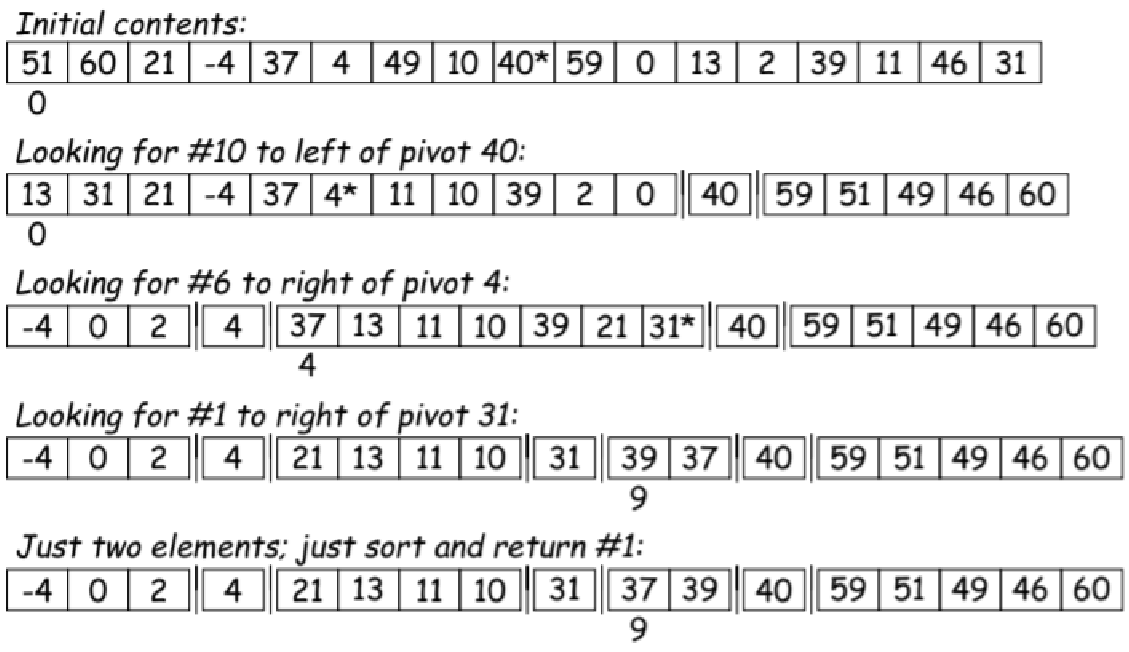
We can do a little better (or worse by some metrics) by having a high probability of

$$\Theta(N)$$

time for all `K` by adapting quicksort:

- Partition around some pivot, p, as in quicksort, arrange that pivot ends up at dividing line.
- Suppose that in the result, pivot is at index m, all elements ≤ pivot have indices ≤ m.
- If m = `K`, you're done: p is answer.
- If m > `K`, recursively select `K`-th from left half of sequence. – If m < k, recursively select `(k – m – 1)`-th from right half of sequence.

Here's an example: find item 10 in the sorted array:



Performance of Selection

For this algorithm, if `M` is roughly in the middle each time, the cost is:

$$\begin{aligned} C(N) &= \begin{cases} 1 & \text{if } N = 1, \\ N + C(\frac{N}{2}) & \textit{otherwise} \end{cases} \\ &= N + \frac{N}{2} + \ldots + 1 \\ &= 2N - 1 \in \Theta(N) \end{aligned}$$

But in the worst case, you get:

$$\Theta(N^2)$$

just like quicksort.

By another, non-obvious algorithm, we can actually achieve linear worst case time for any `K`, which you will see in CS170.