CS 61B Fall 2024

Sorting

Discussion 11: November 12, 2024

1 All Sorts Of Sorts

Show the steps taken by each sort on the following unordered list:

`0, 4, 2, 7, 6, 1, 3, 5`

(a) Insertion sort

Solution:

```
0 | 4 2 7 6 1 3 5
0 4 | 2 7 6 1 3 5
0 2 4 | 7 6 1 3 5
0 2 4 7 | 6 1 3 5
0 2 4 6 7 | 1 3 5
0 1 2 4 6 7 | 3 5
0 1 2 3 4 6 7 | 5
0 1 2 3 4 5 6 7 |
```

(b) Selection sort

Solution:

```
0 | 4 2 7 6 1 3 5
0 1 | 2 7 6 4 3 5
0 1 2 | 7 6 4 3 5
0 1 2 3 | 6 4 7 5
0 1 2 3 4 | 6 7 5
0 1 2 3 4 5 | 7 6
0 1 2 3 4 5 6 | 7
0 1 2 3 4 5 6 7 |
```

(c) Merge sort

Solution: [0 4 2 7 6 1 3 5]

[0 4 2 7] [6 1 3 5]

[0 4] [2 7] [6 1] [3 5]

[0] [4] [2] [7] [6] [1] [3] [5]

[0 4] [2 7] [1 6] [3 5]

[0 2 4 7] [1 3 5 6]

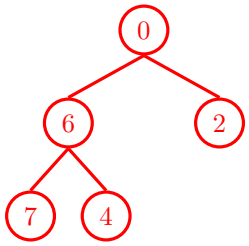
[0 1 2 3 4 5 6 7]

(d) Use heapsort to sort the following array (hint: draw out the heap). Draw out the array at each step:

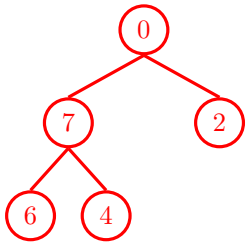
`0, 6, 2, 7, 4`

Solution:

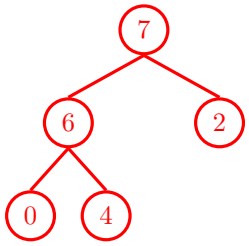
First, we need to heapify our array. We convert the current array to a max heap:



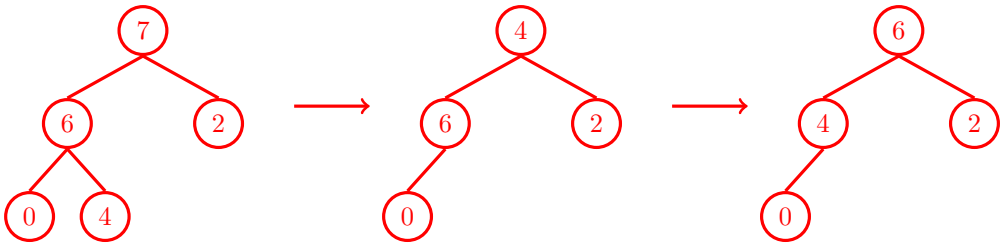
Recall that to heapify our array, we bubble down in reverse level order (bottom to top, right to left). This means we start by bubbling down 4, which in this case gives us the same heap structure. Bubbling down 7, and then 2, leaves the heap unchanged as well. Bubbling down 6 (swapping 6 and 7) then gives us the following:



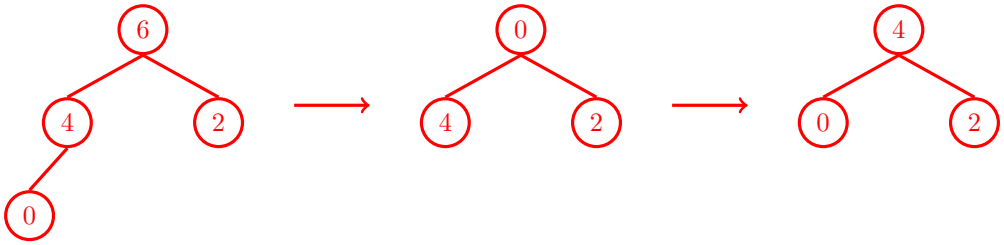
Bubbling down 0 gives us our final heap:



Note that as we heapify, we also modify the underlying array representation as well. This means that our final array looks like `[7, 6, 2, 0, 4]`. We then begin popping off the max value from the heap, placing it at the back of the array. Note that our underlying array representation doesn't consider the popped value as part of the heap any more. We start by popping off 7 and bubbling down:



Our array now looks like this: `[6, 4, 2, 0, 7]`, where the bolded section is considered sorted and not part of the heap. We the continue by popping off 6:



and the array looks like `[4, 0, 2, 6, 7]`. We then pop off 4:

and our array looks like `[2, 0, 4, 6, 7]`. In a similar fashion, we pop off 2 and 0 from our heap, resulting in `[0, 2, 4, 6, 7]` and finally our sorted array: `[0, 2, 4, 6, 7]`.

2 Crystal Has Been Waiting For This

Claire and Ada, two alumni 61B TAs, are trying to sort the TAs by height so they can snap a photo. Can you help them out?

```
public class TA {
    private String name;
    private int height;

    public TA(String name, int height) {
        this.name = name;
        this.height = height;
    }
}
```

(a) Implement a `TAComparator` below such that it compares two TAs' height. Recall that a `Comparator`'s `compare` method returns a negative number when `o1` is "less than" `o2`, positive number when `o1` is "greater than" `o2`, and 0 when they are the same.

Solution:

```
public class TAComparator implements Comparator<TA> {
    @Override
    public int compare(TA o1, TA o2) {
        if (o1.height < o2.height) {
            return -1;
        } else if (o1.height > o2.height) {
            return 1;
        }
        return 0;
    }
}
```

Alternatively, you could also just do:

```
@Override
public int compare(TA o1, TA o2) {
    return o1.height - o2.height;
}
```

or

```
@Override
public int compare(TA o1, TA o2) {
    return Integer.compare(o1.height, o2.height);
}
```

(b) Anniyat suggests that we use Quicksort with our comparator. Given the following list of TAs, who would make the worst pivot? What about the best pivot?

```
TA ethan = new TA("Ethan", 6);
TA ronnie = new TA("Ronnie", 9001);
TA aditya = new TA("Aditya", 1);
TA elana = new TA("Elana", 5);
TA sree = new TA("Sree", 7);
TA kevin = new TA("Kevin", 25);
TA elaine = new TA("Elaine", 9);
TA daniel = new TA("Daniel", 4);
TA teresa = new TA("Teresa", 8);
TA diego = new TA("Diego", 8);
```

Solution: Generally speaking, the worst pivot for Quicksort on a collection is that collection's minimum and maximum values, because the sublists would be partitioned very poorly. The worst pivots in the list are therefore Ronnie (maximum height) and Aditya (minimum height). On the other hand, the best pivot for Quicksort on a collection is that collection's median value. Since there are an even number of elements, we would say that Sree, Teresa, or Diego would all make good pivots.

(c) Diego points out that even though he got in line after Teresa, he ended up in front of Teresa in the sorted list produced by Quicksort (which he doesn't like because that makes it seem like he's shorter than Teresa)! How might we ensure that Diego ends up behind Teresa?

Solution: If we want to preserve ordering of same-valued elements in the original collection when sorted, we should use a stable sort like insertion sort or merge sort! Technically speaking, there is a stable Quicksort possible, but we generally don't use that version.

(d) Our TAs have just been sorted by height, but suddenly Vika and Wilson come running in late! Which sort will do the most minimal work to get them in their correct spots, and what is the additional runtime it will take (ie. not including the runtime for sorting all the other TAs first)?

Solution: Insertion sort: it is the most efficient on an already-sorted or nearly-sorted list (ie. on a completely sorted list, it will make a linear pass and terminate without any swaps, as opposed to something like Quicksort, merge sort, or heapsort, which would indiscriminately try and sort the list without checking if it is already sorted).

We can get everyone in sorted order by tacking on Vika and Wilson to the end of the already-sorted list of TAs, and running insertion sort starting with them (instead of the beginning of the list). At worst, we'd have to make two linear passes (ie. Vika and Wilson are the two shortest TAs), so our overall runtime to get everyone sorted again would be $\Theta(N)$.

3 Zero One Two-Step

(a) Given an array that only contains 0's, 1's and 2's, write an algorithm to sort it in linear time without creating a new array. You may want to use the provided helper method, `swap`.

Hint: Consider how Hoare partitioning rearranges elements in an array.

Solution: The solution below is designed in the style of comparison swaps we have seen thus far (indeed, we see it has flavor similar to quicksort with 1 as the pivot, and other sorts that use swapping). Note that there is also a completely valid counting sort approach, but counting sorts are not the focus of this discussion.

```
public static void specialSort(int[] arr) {
    int front = 0;
    int back = arr.length - 1;
    int curr = 0;
```

Solution:

```
    while (curr <= back) {
        if (arr[curr] < 1) {
            swap(arr, curr, front);
            front += 1;
            curr += 1;
        } else if (arr[curr] > 1) {
            swap(arr,  curr, back);
            back -= 1;
        } else {
            curr += 1;
        }
    }

private static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

(b) We just wrote a linear time sort, how cool! Why can't we always use this sort, even though it has better runtime than Mergesort or Quicksort?

Solution: While our algorithm is super cool, we were only able to write it because we knew there were exactly 3 possible values that could be in our array! For the general case, when the collections we're sorting have much more variety, we can't make these kinds of guarantees.

(c) The sort we wrote above is also "in place". What does it mean to sort "in place", and why would we want this?

Solution: In general, we consider a sorting operation to be done in place if it does not require significant extra space. "Significant extra space" is often defined as linear or greater, with respect to the number of elements we are sorting. For example, if our sorting algorithm requires making a whole new array and copying elements over to it, then it is NOT in place because we had to allocate significant

space for this new array. Some algorithms that can be implemented in place are selection sort, insertion sort, and heap sort. Mergesort technically can be implemented in place, but it's rather complex.

Doing operations in place is beneficial because we typically want to use as little memory/computer resources as possible. Though in this class, we focus on time efficiency, space efficiency is important too in the real world!