CS2040S: Data Structures and Algorithms

Tutorial 2

Problem 1. Queues and Stacks

Queues have a lot of practical uses. This exercise will go through some of these uses:

(a) Last week, we learnt about the basic operations of stacks and queues. How would you implement a stack and queue in Java?

(b) A set of parentheses is said to be balanced as long as every opening parentheses "(" is closed by a closing parentheses ")". So for example the strings "()()" and "(())" are balanced but the string ")(())(" and "((" are not balanced. Using a stack, determine whether a string of parentheses are balanced.

(c) Sort an array using two queues.

(d) (Challenge) Implement a queue that allows you to get the minimum item as efficiently as possible.

Answer: The main point for this question is for them to go over Stack and Queue APIs as this might be the last time we get to focus on them. It is important for them to understand these two structures since they will be used a lot for future algorithms (e.g. DFS and BFS). Tutors are free to practice with other questions apart from these (e.g. Expression evaluation).

(a) This was covered partially in recitation. The important thing is for them to understand the associated costs with regards to an ArrayList.

(b) The idea behind this is to simply push opening parentheses onto a stack, then pop them out whenever you encounter a closing parentheses. The string will not be balanced as long as you try to pop an empty stack or your stack is non-empty after reading the last character. For extension you can try asking them how would they tackle the extra case where there are multiple different type of parentheses like { and []

(c) The näive way we can do this is by simply cycling through the queue and picking the minimum element each time and appending it to the second queue. However, we can of course do better.The idea is to do this like merge sort! We will start by sorting the pairs, then by 4 elements, so and so forth. (Will add diagram if this is unclear.)

(d) Queue with minimum : We need to add a few extra steps to enqueue and dequeue operations. We also need a pointer which will point to the current minimum element. Now, we shall do the following. We shall also have some additional pointers which are used to indicate "valleys" in the queue. Note that by valley, we refer to something that is smaller than everything in the queue **after** it. As an example consider the queue below:

Head 1 2 5 4 9 8 Tail

Here, the valleys are 1, 2, 4,and 8 (Notice how the valleys are in ascending order). We want to maintain a secondary linked list between these valleys so that whenever the current minimum is dequeued we can point to the next minimum easily.So each element will have a pointer to its previous valley and next valley. (E.g. The previous valley for 4 is 2 and the next valley is 8.).

Now this is how our operations will work. For enqueueing, we note that the newly added element instantly becomes a valley. Now, when we add it, we shall traverse the previous valleys from the tail, until we reach a valley smaller than our currently inserted element. (E.g. If we inserted 3, we would go through 8, 4 and end up at 2). As we go through the valleys, we shall remove all pointers of the nodes we have traversed, then update the next pointer of the node we stopped at to be the element we just inserted. In the above example, 8 and 4 will be removed from our valley list, and the next valley for 2 will point to 3, and the previous valley for 3 will be 2.

For dequeueing, we will not have an issue until we dequeue the current minimum. If we do dequeue the current minimum, we shall first assign the minimum pointer to the next valley, set the previous valley of the new minimum to be null, then remove the node from the queue.

For finding minimum, we simply look at the pointer to the minimum element (which is the first valley).

Problem 2. Moar Pivots!
Quicksort is pretty fast. But that was with one pivot. Can we improve it by using two pivots? What about k pivots? What would the asymptotic running time be? (That is, the algorithm is to choose the pivots at random—or perhaps, imagine you have a magic black box that gives you perfect pivots—then sort the pivots, partition the data among the pivots, and recurse on each part. You may assume whichever gives you a better performance)

Answer: Notice that partitioning now takes:

- $O(k \log k)$ time to sort the pivots.
- $O(n \log k)$ time to place each item in the right bucket (e.g., via binary search among the pivots).

So the resulting recurrence is: $T(n) = kT(n/k) + O(n \log k)$, as long as $n > k$. The solution to this recurrence is $O(n \log_k(n) \log k = O(n \log n)$, i.e., no improvement at all! Worse, doing the partition step in place becomes much more complicated, so the real costs become higher.

Except—and here's the amazing thing—we have discovered that 2 and 3 pivot QuickSort really is faster than regular QuickSort! (I might mention this in class.) Try running the experiment and see if it's true for you! Currently, the Java standard library implementation of QuickSort is a 2-pivot version!

Students are encouraged to try run a few experiments on their own with regards to this.

Problem 3. It's Not Just About Time
What if your goal is to minimize the number of times data is written, rather than the number of

comparisons? Assume you want your algorithm to be in-place. What is a good algorithm in that case? Assume for now you do not care about comparisons at all. One case where this is important is if you have very large data to sort: comparing is relatively cheap (as you only have to look at a small prefix of the data to decide the order, in most cases), but moving is expensive (because you have to re-write a large file.)

Answer: Selection Sort! Only $2n$ writes needed. You might want to bring out the fact this is a reason people might use Selection Sort. In extension to this, tutors might talk about other situations where other sorting algorithms learnt might be preferred over the asymptotically faster Merge Sort and Quicksort.

Problem 4. But Wait There's More...
Continuing on from Problem 3, now your goal is to keep $O(n)$ writes, but with only $O(n \log n)$ cost for reads. (This turns out to be important for non-volatile NVRAM memory where writing takes longer than reading, but both matter.) For now, do not worry about the algorithm being in-place, but be sure to count every single write operation. (E.g., if you create an auxiliary array and write an integer to that array, it counts.)

Answer : Now, the first idea might be to use Selection Sort in this instance. However, we note that Selection Sort would not have an $O(n \log n)$ cost of reads. So we need to be a bit smarter about this. What we shall do is the following.

1. First we shall choose $k = n/\log n$ pivots. This means we would have $n/\log n$ buckets which we can partition our array into. The size of each bucket will be $\log n$.

2. Now we run selection sort on each pivot separately.

Why does this work? Well for one thing we note that the number of writes still remains $O(n)$. The partitioning will take $O(n)$ writes and running the selection sort on each bucket would also take $O(\log n)$ per bucket, hence it will still be $O(n)$ writes overall. Hence, the only thing we need to check at this point is whether we can do this without doing more than $O(n \log n)$ reads. We find out that this is the case since:

1. It takes $O(n)$ to sort the pivots (Intuitively true since we have $n/\log n$ pivots. The recurrence of merge sort or quicksort on this would ensure we can do this in the given time). The number of writes in this case would still be $O(n)$. Try and work out the recurrence!

2. It takes $O(\log n)$ for us to binary search for the correct bucket to put the rest of elements in. Hence overall this will take $O(n \log n)$ time. Now over here we note that we are still doing $O(n)$ writes since there are only $O(n)$ elements to be written.

3. Lastly note that it will take $O(n \log n)$ to sort all the buckets. This is because each bucket takes $O(\log^2 n)$ time and we have $n/\log n$ buckets overall. Hence, the time will still be $O(n \log n)$ overall.

Problem 5. Child Jumble
Your aunt and uncle recently asked you to help out with your cousin's birthday party. Alas, your

cousin is three years old, and so that means spending several hours with twenty rambunctious three year olds, as they race back and forth, covering the floors with paint and hitting each other with plastic beach balls. Finally it is over. You are now left with twenty toddlers that each need to find their shoes. And you have a pile of shoes that all look about the same. They are not helpful. (Between exhaustion, too much sugar, and being hit on the head too many times, they are only semiconscious.)

Luckily, their feet (and shoes) are all slightly different size. Unfortunately, they are all very similar, and it is very hard to compare two pairs of shoes or two pairs of feet to decide which is bigger. (Have you ever tried asking a grumpy and tired toddler to line up their feet carefully with another toddler to determine which has bigger feet?) So you cannot compare shoes to shoes or feet to feet.

The only thing you can do is you can have a toddler try on a pair of shoes. When you do this, you can figure out whether the shoes fit, or if they are too big, or if they are too small. That is the only operation you can do.

Come up with an efficient algorithm to match each child to their shoes. Give the time complexity of your algorithm in terms of the number of children.

Answer : This is a classic QuickSort problem, often presented in terms of nuts-and-bolts (instead of kids). The basic solution: choose a random pair of shoes (e.g., the red Reeboks), and use it to partition the kids into "bigger" and "smaller" piles. Along the way, you find one kid ("Alex") for whom the red Reeboks fit. Now use Alex to partition the shoes into two piles: those that are too big for Alex and those that are too small for Alex. Match the big shoes to the big kids, the small shoes to the small kids, and recurse on the two piles. If you choose, the "pivot" shoes at random, you will get exactly the QuickSort recurrence.