COS 226 Algorithms and Data Structures Spring 2024

Midterm Solutions

1. **Initialization.**

   Don't forget to do this.

2. **Heaps and trees.**

   (a) 35, 45

   Looking at the before heap, we can tell that the value is at most 50. Looking at the after heap, the value is at least 30. Alternatively, one can try to replace the ? with each of the values, simulate the `delMax()` operation and verify if the resulting heap matches the after heap.

   (b) 50, 70

   Substitute each of the answers for the ? to see that 50 and 70 are the only options that allow for the reconstruction of a BST. If ? = 50, then 50 is the right child of 40 and 60 is the right child of 50. If ? = 70, then 70 is the right child of 40 and 60 is the left child of 70. Note that a BST is not necessarily complete.

   (c) 21, 23, 27

   These are the only nodes that can be colored red while satisfying the color invariants and perfect black balance. To find these nodes, here is an easy way to do so: first note that 25 and 29 cannot be red, since they correspond to right leaning links, which are never red. Now, we can observe that 21 needs to be red, otherwise a path going through node 25 would have a different number of black links when going left versus when going right. Using the same logic, 27 needs to be red. So the only nodes left are 19 and 23. By considering paths going through node 23 and then going right, we can see that the number of black links in the section of the path below node 23 is 2, so the same needs to apply to a path going left on node 23. Hence, 19 needs to be black. Applying a similar logic to paths going through node 26, we can conclude that 23 needs to be red.

3. Five sorting algorithms.

   D E B F C

   D. mergesort just before the last call to `merge()`
   E. quicksort after first partitioning step
   B. selection sort after 12 iterations
   F. heapsort after heap construction phase and putting 12 keys into place
   C. insertion sort after 16 iterations

4. **Stacks and queues.**

   (a) A, E

   This method goes through the bits of a number in reverse order, so we want to use a First-In-Last-Out data structure, which is exactly a stack.

   (b) $\Theta(\log n)$

   The values of n inside the loop are $n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots, 1$ (assuming for simplicity that n is a power of 2). This sequence is of length $\Theta(\log n)$. Note that when we iterate over a stack, we are not popping any of the elements.

   (c) $\Theta(\log \log n)$

   When we insert m elements to an empty stack, the array expands $\sim \log(m)$ times: the array expends when we have 1 element, 2 elements, 4 elements, 8 elements, etc. Since we insert $m = \Theta(\log n)$ elements, the array expends $\Theta(\log \log n)$ times.

5. **Analysis of algorithms and sorting.**

   (a) $\sim 2n^2$

   Selection sort always makes $\sim \frac{1}{2}m^2$ compares on an array of length m. Here, $m = 2n$, so $\frac{1}{2}m^2 = \frac{1}{2}(2n)^2 = 2n^2$.

   (b) $\sim n^2$

   We count the number of exchanges, as for insertion sort the number of compares is at most the number of exchanges plus the size of the array. Row i below depicts the array when inserting the element at position i (in bold). The elements exchanged with it are underlined.

   1 15 2 14 3 13 4 12 5 11 6 10 7 9 8 8
   1 **15** 2 14 3 13 4 12 5 11 6 10 7 9 8 8
   1 <u>15</u> **2** 14 3 13 4 12 5 11 6 10 7 9 8 8
   1 2 <u>15</u> **14** 3 13 4 12 5 11 6 10 7 9 8 8
   1 2 <u>14 15</u> **3** 13 4 12 5 11 6 10 7 9 8 8
   1 2 3 <u>14 15</u> **13** 4 12 5 11 6 10 7 9 8 8
   1 2 3 <u>13 14 15</u> **4** 12 5 11 6 10 7 9 8 8
   1 2 3 4 <u>13 14 15</u> **12** 5 11 6 10 7 9 8 8
   ⋮

   We deduce that for a general n, the number of exchanges is

   $$0 + 0 + 1 + 1 + 2 + 2 + 3 + 3 + \cdots + (n-1) + (n-1) = 2(0 + 1 + 2 + 3 + \cdots + (n-1)).$$

   By the triangle sum, the latter sum is $\sim n^2$.

   (c) $\sim \frac{3}{2} n \log_2 n$

   Consider the "top-most" `merge()` that merges two subarrays of length n to one array of length $m = 2n$. This `merge()` uses $\frac{3}{4}n$ compares, as when the right subarray is exhausted, there are still $\frac{m}{4}$ elements remaining in the left subarray.

   For instance, for $n = 8$ (i.e., $m = 16$), the first `merge()` merges the sorted left half, $1, 2, 3, 4, 12, 13, 14, 15$, with the sorted right half, $5, 6, 7, 8, 8, 9, 10, 11$. To this end, it

first places the left half of the left subarray $1, 2, 3, 4$, then the entire right subarray $5, 6, 7, 8, 8, 9, 10, 11$. At this point $12 = \frac{3}{4}m$ elements were placed, each after being compared with one element. Since we exhausted the right subarray, the remaining $\frac{m}{4}$ elements in the left subarray are copied over without any compares. The same analysis is valid for any of the subsequent calls to `merge()`.

So, the number of exchanges satisfies the divide-and-conquer recurrence $T(m) = 2T(m/2) + \frac{3}{4}m$. We solve this recurrence and get $T(m) = \frac{3}{4}m \log_2 m = \frac{3}{4} \cdot 2n \log_2(2n) = \frac{3}{2}n(\log_2(n) + 1)$, which is $\sim \frac{3}{2}n \log_2(n)$.

(d) $\Theta(n^2)$

This is a worst-case array for quicksort as in every call to `partition()`, the pivot is either the smallest or the largest element in the subarray. For instance, when $n = 8$, `partition()` is first called with an array of length 16 and the pivot is 1, which is the smallest element. It partitions the array to a subarray of length 0 (elements smaller than 1) and to a subarray of length 15 (elements greater than 1). Next, `partition()` is called with an array of length 15 and the pivot is 15, which is the largest element. It partitions the array to a subarray of length 14 (elements smaller than 15) and to a subarray of length 0 (elements greater than 15). This continues and in the subsequent calls to `partition()` the arrays are of lengths $13, 12, 11, \ldots$.

More generally, on the array of length $m = 2n$, `partition()` is called with arrays of length $m, m-1, m-2, \ldots, 1$. Since `partition()` always makes $\sim m$ compares on an array of length m, we get that the total number of compares is $\sim m + (m-1) + (m-2) + \ldots + 1$. By the triangle sum, this is $\Theta(m^2) = \Theta(n^2)$.

6. **Asymptotic.**

T F F T

T: Denote $m = n^2$. For simplicity assume that m is a power of 2.
At the beginning $i = m$ and j gets the values 0 to $m-1$, so `hello` is printed m times.
Then, $i = \frac{m}{2}$ and j gets the value 0 to $\frac{m}{2} - 1$, so `hello` is printed $\frac{m}{2}$ times.
Then, $i = \frac{m}{4}$ and j gets the value 0 to $\frac{m}{4} - 1$, so `hello` is printed $\frac{m}{4}$ times.
$\vdots$

We get that `hello` is printed $m + \frac{m}{2} + \frac{m}{4} + \frac{m}{8} + \ldots$ times. Using the formula for geometric sum, the last sum is $\sim 2m$. Substitute n^2 for m to get $\sim 2n^2$.

F: We implement binary heaps as arrays for simplicity of coding, to save memory, and to have better data locality for caching. It is possible to implement a binary heaps using nodes and links while guaranteeing a running time of $O(\log n)$ for both `insert` and `delMax`. To do so, we keep a pointer to the root of the heap as well as to the last leaf. To add an key to the heap, we use the pointer to the last leaf to find the location where a new leaf holding the new key should be inserted. We then swim the new key. When deleting the maximum, we exchange the key at the root of the heap with the key at the last leaf. We then delete the last leaf, set the last leaf pointer to the previous leaf, and sink the key at the root.

F: Consider the following compare-based sorting algorithm: insert the n elements to a BST and return the in-order traversal of the BST. Recall that an in-order traversal can be done in linear time. So, if insert can be done using $O(\sqrt{\log n})$ compares in the worst case, the above

sorting algorithm uses $O(n\sqrt{\log n})$ compares (note that search and delete are not used by this algorithm), which violates the sorting lower bound.

T: Copy the $n \times n$ array to an array of length n^2 such that the first line is copied first, then the second line, etc. Observe that this array is in acceding order and therefore we can use a binary search. Note that the binary search can also be done in-place (without copying to a second array).

7. **Algorithm design.**

We scan both array from left to right using two indices i and j, like in `merge()`. If the keys in `a[i]` and `b[j]` are different, we increment our count and the index of the smaller key. Otherwise, we increment both indices. For reference, here is a full Java implementation.

```
public static int diff(int[] a, int[] b)
{
     int i = 0, j = 0, count = 0;

     while (i < a.length && j < b.length)
     {
          if (a[i] == b[j]) { i++; j++; }
          else if (a[i] < b[j]) { count++; i++; }
          else { count++; j++; }
     }

     return count + (a.length - i) + (b.length - j);
}
```

8. **Data structure design.**

Like in the *Percolation* assignment, we use a boolean array of length n to remember what sites are open. We also keep a *union-find* data structure, implemented as *weighted quick-union*, to represent the maximal consecutive sequences of open site as disjoint sets.

On an `open(i)` request, we mark i as open in our array. In addition, if site $i-1$ is open, we merge that set containing i and the set containing $i-1$. If site $i+1$ is open, we also merge the set containing i and the set containing $i+1$.

On an `openLen(i)` request, we return the size of the disjoint set in the union-find object that contains i. To this end, we add the method `int subsetSize(int i)` to the API of weighted quick-union. `subsetSize(i)` returns `size[find(i)]`. Recall that `size` is an instance variable in the `WeightedQuickUnionUF` class and that `size[j]` is the number of elements in subtree rooted at j.

For reference, here is a possible Java implementation (for clarity, we do not check the validity of the inputs).

```
public class OpenSites {

    private final int n;
    private final boolean[] isOpen;
    private final WeightedQuickUnionUF uf;

    public OpenSites(int n)
    {
        this.n = n;
        isOpen = new boolean[n];
        uf = new WeightedQuickUnionUF(n);
    }

    public void open(int i)
    {
        isOpen[i] = true;

        if (i > 0 && isOpen[i - 1])
            uf.union(i - 1, i);

        if (i < n - 1 && isOpen[i + 1])
            uf.union(i, i + 1);
    }

    public int openLen(int i)
    {
        if (!isOpen[i])
            return 0;

        return uf.subsetSize(i);
    }
}
```

9. **EXTRA CREDIT.**

It might be tempting to try to modify the weighted quick union data structure to support a "unlink" operation, but this is not easy. A better approach is to use a completely different data structure, namely a balanced binary search tree.

Maintain a balanced BST containing the indices of all the sites that are currently blocked. This means that in the constructor we start by inserting all the 0 through $n-1$ indices to the BST (since they are all initially blocked). Now, opening a site consists of removing its index from the BST and blocking a site consists of adding its index to a BST.

To implement the `openLen(i)` method we need the following observation: the indices of the endpoints of the set containing element i are exactly the smallest blocked element that is greater than i and the largest blocked element that is less than i. But this is exactly what the `floor` and `ceil` methods in the BST give us. So we can return `bst.ceil(i) - bst.floor(i) - 1`.

```
public class OpenSites {

    private final int n;
    private final RedBlackBST<Integer, Boolean> bst;
    // We don't actually need the Boolean value, but we have to pick one

    public OpenSites(int n)
    {
        this.n = n;
        for (int i = 0; i < n; i++)
            bst.put(i, true);
    }

    public void open(int i)
    {
        bst.delete(i);
    }

    public void block(int i)
    {
        bst.put(i, true);
    }

    public int openLen(int i)
    {
        if (bst.contains(i))
            return 0;

        return bst.ceil(i) - bst.floor(i) - 1;
    }
}
```