COS 226 Algorithms and Data Structures Spring 2023

Midterm Solutions

1. **Initialization.**

   Don't forget to do this.

2. **Memory.**

   (a) 48 bytes

   - 16 bytes for object overhead
   - 4 bytes for `int` key
   - 8 bytes for `double` value
   - 4 bytes for `int` count
   - 8 bytes for left link
   - 8 bytes for right link

   (b) ~ $48n$ bytes

   The only memory that grows as a function of n is the number of nodes.

3. **Data structures.**

   (a) 4 5

   This is union-by-size, not union-by-height.

   (b) 40 45 50

   A priority queue allows duplicate keys.

4. **Five sorting algorithms.**

   E B D C F

   E. quicksort after first partitioning step
   B. selection sort after 12 iterations
   D. mergesort (top-down) about 75% done
   C. insertion sort after 16 iterations
   F. heapsort after heap construction phase and putting 6 keys into place

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

   (a) $\sim 8n^2$

   Selection sort makes $\sim \frac{1}{2}m^2$ compares to sort any array of length m. Here, $m = 4n$.

   (b) $\sim 2n^2$

   When a B is inserted, insert sort does not make any exchanges. When an A is inserted, insertion sort makes n exchanges, one with each of the Bs to its left. There are $2n$ As, which leads to a total of $2n^2$ exchanges. The number of compares is always within m of the number of compares, where $m = 4n$ is the length of the array.

   (c) $\sim 6n$

   The first partitioning step makes $m = 4n$ compares and puts the $2n$ keys equal to the pivot in their final positions. The remaining $2n$ keys are also in their final positions; but, as far as 3-way quicksort is concerned, this is just a left (or right) subarray that still needs to be sorted. So, an additional $2n$ compares are needed for the second (and final) partitioning step.

6. **Left-leaning red–black BSTs. (6 points)**

   rotate left 18; rotate right 26; color flip 22; rotate left 14

7. **Properties of algorithms and data structures.**

   (a) A E

   The best case happens when the search key happens to be at index `n / 2`*. In this case, binary search terminates after one compare.*

   In the worst case, binary search makes $\sim \log_2 n$ compares.

   (b) A H

   If all of the keys are equal, then a delete-the-maximum operation exchanges the key in the root with a key in a leaf and sinks the key in the root down. But, the sink operation terminates after 3 compares and 0 exchanges.

   In the worst case, the number of compares is $\sim 3 \log_3 n$ because the height of the 3-way heap is $\sim \log_3 n$ and 3 compares are needed at each level for the sink operation.

   (c) A K

   If the root node contains the smallest key in the BST and the key to be inserted is smaller than that key, then insertion takes only 1 compare (and inserts the new node as the left child of the root).

   If BST were built by inserting n keys in ascending order, then its height would be $n - 1$. Inserting a new key that is larger than the largest in the BST would require $n-1$ compares.

(d) D G

The best case happens when the 2–3 tree contains all 3-nodes, except for one path of all 2-nodes, say, down the left spine. Then, the height is $\sim \log_3 n$. *When inserting a key smaller than any key in the 2–3 tree, the insertion key is compared to each of the* $\sim \log_3 n$ *keys on the left spine. (No compares are made when splitting the 4-nodes, on the way back up the 2–3 tree.)*

The worst-case happens when the 2–3 tree contains all 2-nodes, except for one path of all 3-nodes, say, down the right spine. Then, the height is $\sim \log_2 n$. *When inserting a key larger than any key in the 2–3 tree, the insertion key is compared to each of the* $\sim 2 \log_2 n$ *keys on the right spine.*

8. **Why did we do that?**

I A I I I A I I

(a) *If we implement a queue by storing the most recently added in the first node in the singly linked list (instead of the last node), then there is no efficient way to implement* dequeue*—maintaining a* `last` *pointer does not work because we can't update it efficiently after a* dequeue *operation.*

(b) *This was an arbitrary design decision.*

(c) *This is needed to avoid integer overflow when* `lo` *and* `hi` *are very large.*

(d) *This is needed for stability.*

(e) *This is needed to prevent quicksort from going quadratic on inputs with lots of duplicate keys.*

(f) *This was an arbitrary design decision.*

(g) *This is crucial to maintain symmetric order and perfect balance.*

(h) *This is needed to maintain perfect black balance.*

9. **Iteration.**

A G C H F

10. **Data-type design.**

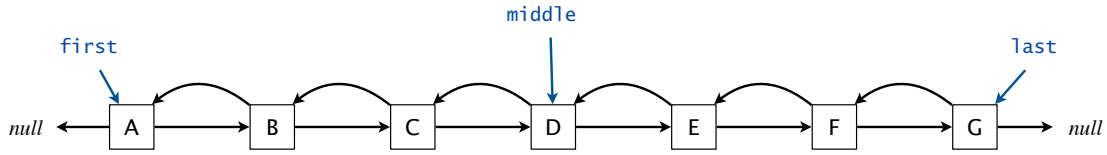
The key idea is maintain three pointers into a doubly linked list containing the items: a pointer to the first node (front), a pointer to the last node (back), and a pointer to the middle node. Since it's a doubly linked list and we are maintaining a pointer to the middle node, it's easy to delete that middle node and it's easy to update the middle pointer.

(a)

```
public class MiddleQueue<Item> {
    private Node first, last, middle;
    private int n;

    private Node {
        private Item item;
        private Node next;
        private Node prev;
    }
    ...
}
```

(b)



(c) *Add a new node containing the item to the front of the linked list, as in* `Deque`. *Advance* `middle` *to the previous node if* `n` *is odd. Increment* `n`.

(d) *Add a new node containing the item to the back of the linked list, as in* `Deque`. *Advance* `middle` *to the next node if* `n` *is even. Increment* `n`.

(e) *Delete the node from the doubly linked list. Update* `middle` *to the next node if* `n` *is even; to the previous node if* `n` *is odd. Decrement* `n`.

An alternative solution is to store the items in two `Deque` *objects, one containing the first* $\lceil n/2 \rceil$ *items in order, the other containing the last* $\lfloor n/2 \rfloor$ *items in order, so that the middle item is always the last item in the first deque. When adding an item (or removing the middle item), you may have to transfer the last item from the first deque to be the first item in the second deque (or vice versa).*