Solutions to the Final Exam

Solution 1:

(1.1) A “deque” is short for “Double-Ended QUEue,” but it is a homonym with “deck,” and like a deck of cards, items can be inserted or deleted from either end.

(1.2) Min: 2 (a linear chain of nodes), Max: $n - 1$ (a star consisting of a single central node with $n - 1$ children branching off)

(1.3) True: The inserted key is the rightmost node of the tree, and hence any imbalance it causes will be right-right heavy. After this rotation, no other rotations are needed.

(1.4) False: Although the number of nodes on the search path cannot be larger, the number of comparisons can be higher because each 3-node stores two keys, and we may need to compare against both. To see this more explicitly, consider both an AVL and 2-3 tree that are complete binary trees. Now, replace the rightmost search path in the 2-3 tree with 3-nodes. If we search for the largest key, the number of comparisons in the AVL tree will be $\lceil \lg n \rceil$, but in the 2-3 tree it will roughly twice as large.

(1.5) The problem with Zigs alone is that they are not guaranteed to improve the tree’s structure. As seen in lecture, if you are given a degenerate tree, performing Zig rotations from the leaf level results again in a degenerate tree.

(1.6) In a B-tree of order 15, nodes have 8–15 children, except the root which 2–15.

(1.7) See Fig. 1(a) for the case of an imbalanced tree and (b) for the case of a tree where splits do not alternate. In the first case, points are inserted in left to right (bottom to top) order. The line $\ell$ intersects roughly $n/2$ leaf cells. In the second case, points are inserted in a balanced manner (each time splitting at the median coordinate), but we only split along $x$, never $y$. Any horizontal line intersects all the cells.

![Figure 1: Stabbing cells in a kd-tree.](image)

(Note that it is not sufficient to present an example where the number of stabbed nodes exceeds $\sqrt{n}$ exactly. It is necessary to present an example where it is clear how to continue adding points so that the count exceeds $c\sqrt{n}$, for any positive constant $c$.)
In general, \( O(n \log n) \) time is needed to sort the keys. However, by performing an inorder traversal, we can extract the sorted keys in \( O(n) \) time. The rest of the construction runs in time proportional to the size of the tree. This is because we can retrieve the median element and extract the subarrays on either side of the median constant time.

The \textit{treap data structure} was a special case of the geometric data structures Cartesian trees and priority search trees (where the key played the role of the \( x \)-coordinates and the priorities played the roles of the \( y \)-coordinates).

\textbf{Solution 2:}

(2.1) Preorder: \texttt{d b g c e h j a i f}

(2.2) Inorder: \texttt{g e c h b d a j f i}

(2.3) Crazy: \texttt{d j i i f j a d b b g c h c e}

(2.4) The traversal prints each node, traverses the right subtree recursively, traverses the left subtree recursively, and then prints the key again.

\begin{verbatim}
void traverse(Node p) {
    if (p != null) {
        print(p.key)  
        traverse(p.right)
        traverse(p.left)
        print(p.key)
    }
}
\end{verbatim}

\textbf{Solution 3:}  See Fig. 2. The substrict identifiers are shown (in suffix order) in the upper left. They are sorted lexicographically in the lower left. The final suffix tree is shown on the right.

\textbf{Solution 4:}

(4.1) Our helper function is \texttt{printMaxK(Node p, int k)}, which prints the largest \( k \) nodes from the subtree rooted at \( p \). If \( k \) is not positive, we print nothing. The initial call is \texttt{printMaxK(root, k)}. Subtracting the size of the right subtree from \( k \) leaves the number of nodes remaining to be printed. (The remainder may be negative, but if so, nothing is printed.)

Because we invoke the function on left, then this node, then right, the keys will be printed in ascending order.

\begin{verbatim}
void printMaxK(Node p, int k) {
    if (p != null && k > 0) { // something to print?
        int rightSize = (p.right == null ? 0 : p.right.size) // size of p.right
        int remainder = k - rightSize // remainder after p.right
        if (remainder > 0)
            printMaxK(p.left, remainder - 1) // print left keys
        print(p.key) // print this node
        printMaxK(p.right, k) // print right keys
    }
}
\end{verbatim}
We assert that the running time is $O(k + \log n)$. To see this, observe that there are two ways we might visit a node. First, we visit it to print its key. The number of such nodes is $k$, and (since we do $O(1)$ work in each node) the time spent visiting all these nodes is $O(k)$. Otherwise, we visit the node but do not print its contents. This happens when the right subtree has $k$ or fewer keys. If so, we make a recursive call on its right subtree only. Since the tree’s height is $O(\log n)$, the number of times we can do this is $O(\log n)$. So, the total running time is $O(k + \log n)$.

The helper function is called `printEvenOdd(Node p, int index)`, where `index` indicates the index of this key in the sequence. We print a key if the index value is odd, and we increment the index each time we visit a node. We return the updated index after visiting a subtree (which is a bit sneaky). The initial call is `printEvenOdd(root, 1)`. It easy to see that this runs in $O(n)$ time.

```
int printEvenOdd(Node p, int index) {
    if (p == null) return index // nothing to print
    else
        index = printEvenOdd(p.left, index) // print left subtree
        if (index % 2 == 1) print(p.key) // print current if odd
        index += 1
        return printEvenOdd(p.right, index) // print the right subtree
}
```
Solution 5:

(5.1) Worst-case $n + 1$: In the worst case, the user performs $n$ pushes and erases them all. In this case the pop operation skips over all $n$ of the erased elements and returns null, for a running time of $n + 1$.

(5.2) Amortized 1.5: Before giving the formal proof, here is an intuitive argument. The expensive operations are skips of erased elements performed during a pop operation. In order to skip an erased node, it must first be pushed and then erased. If we charge an additional $\frac{1}{2}$ token for each push and erase, we have enough tokens accumulated to pay for each skip of an erased elements.)

We will employ a standard token-based analysis. We charge 1.5 tokens for each operation. Each push and erasure takes 1 unit of actual time, and this means that we place half a token in the bank for each. Whenever a pop comes along, we skip over some number of elements. In order to skip over an element, it must have been pushed (depositing half a token) and it must have been erased (depositing half a token), and together, the $\frac{1}{2} + \frac{1}{2} = 1$ token pays for the time needed to skip this one element. We also use one token for the pop of the final unerased item.

Is 1.5 tight? Yes. This can be seen if you push $n$ entries (for a huge value $n$), erase them all, and do a single pop. The total number of operations is $m = n + n + 1 = 2n + 1$. The total work is $n + n + (n + 1) = 3n + 1$. Averaging over the $m$ operations, the amortized cost is $(3n + 1)/m = (3n + 1)/(2n + 1)$. If $n$ is large, this is $\approx 1.5$.

(5.3) Expected $O(m/k)$: The probability that any element was erased is $k/m$. Therefore, the expected number of erased elements we will skip over before finding an unerased element is roughly $m/k$. Allowing one more unit of time for the final pop of the unerased item, this yields an expected time of at most $1 + m/k = O(m/k)$.

Solution 6:

(6.1) To answer orthogonal top-$k$ queries, the preprocessing consists of building a 3-layer structure. The first two layers consist of a standard 2D range tree based on the $(x, y)$-coordinates of the points. This yields a structure with space $O(n \log^2 n)$. For each node of this structure, we create a third layer consisting of a simple list inversely sorted by the ratings.

Given any query region $R$, we know by standard results on range trees that we can express the set of all points lying within $R$ as the disjoint union of a collection $O(\log^2 n)$ subtrees and these can be computed in $O(\log^2 n)$ time. For each subtree, we access the auxiliary third-level structure, sorted on rating to select the $k$ largest elements from each. This yields a total of $O(k \log^2 n)$ elements. In the same time, we can extract the $k$ largest elements. (We could also use the solution of Problem 4 for the ratings-sorted trees, but this yields a slightly worse running time of $O((\log^2 n)(k + \log n)) = O(k \log^2 n + \log^3 n).)

You might wonder whether the third layer is necessary. The problem with trying to solve the problem with just a two-layer structure (sorted say on $x$ and then $y$) is that the points within the auxiliary subtrees are not sorted by rating. A single subtree may contain $O(n)$ elements, and filtering out the largest $k$ will generally take $O(n)$ time, which will be way too slow.
We break the annulus up into four rectangles, and apply an orthogonal top-$k$ query to each. This yields up to $4k$ elements. Among these, we select the largest $k$, which can be done in additional $O(k)$ time. The overall space and query time is the same as for 6.1.

You might wonder whether it is possible to apply the trick treating the annulus as a difference of two squares. That is, we first identify the points lying within the large (radius $r_2$) square and then filter out the points in the smaller (radius $r_1$) square. While this works for counting, where we can take differences, it does not work for the $k$-largest. The problem with this is that the inner square may contain a huge number of elements (e.g., $O(n)$), and these are larger than the elements in the annulus. The time needed to filter these out (processing them point by point) would be $O(n)$, which is way too slow.