Solutions to Practice Problems for the Final Exam

Solution 1:

(a) Only (iii) is true: In an inorder traversal, internal and external node alternate with each other.

(b) 2: In AVL tree-insertion, after the first rotation operation (single or double) the subtree to which the rotation is applied has exactly the same height it did prior to the insertion. It follows that this subtree and all the others in the tree are properly balanced with respect to the AVL height criteria. Since we count a double rotation as two rotations, the answer is 2.

(c) \(O(\log n)\): In AVL-tree deletion, the subtree height may change after a rotation, and this may result in the parent’s balance factors being altered. This pattern can propagate all the way to the root.

(d) A finger search is one where, rather than starting at the root of the tree, the search starts from an existing node in the tree. For example, given a splay tree, let \(p\) be some node in the tree and let \(x\) be a key value. The function \(\text{fingerSearch}(x, p)\) performs a search for \(x\) starting at node \(p\). Ideally, such a search should take advantage of the fact that if \(p.key\) and \(x\) are similar in value, then this search is more efficient than starting from the root of the tree.

(e) If the second hash function and table size share a common factor, then the probe sequence may not visit every entry of the table, and hence insertion may fail even when there are available empty slots in the table. (For example, \(m = 10, h(x) = 3, \text{ and } g(x) = 5\), the probe sequence will consist of indices of the form \((3 + 5 \cdot i) \mod 10 = (3, 8, 3, 8, \ldots)\). If these two positions are filled, then the insertion fails.)

(f) (1) Unbalanced binary search trees - Yes. Although it may take \(O(n)\) time to find the key, inserting it involves only \(O(1)\) modifications.

(2) AVL trees - Yes. At most two rotations are performed with each insertions, so there are \(O(1)\) structural changes.

(3) AA-trees - Yes. We may perform skews and splits up to the root, for a total of \(O(\log n)\) changes.

(4) Quake-heaps - Yes. Insertion takes \(O(1)\) time, so there are only \(O(1)\) changes possible.

(5) Treaps - No. The tree’s height may be as high as \(O(n)\), and it is possible to insert a node and have it filter all the way up to the root.

(6) Splay trees - No. The tree’s height may be \(O(n)\) and if so splaying can take \(O(n)\) time.

(7) Scapegoat trees - No. If the insert triggers a major rebuild, this can involve \(O(n)\) changes.

(g) (i) A scapegoat tree’s height is guaranteed to be \(O(\log n)\).
(h) The *buddy system* has more internal fragmentation: Internal fragmentation refers to the wastage of memory within (as opposed to between) the allocated blocks. The buddy system may waste up to half of the allocated block by rounding the size up to the next power of 2.

Solution 2:

(a) Our helper function is `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 `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.

```java
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
    }
}
```

(b) 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 $O(k + \log n)$.

(c) 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.

```java
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 3:

(a) Every node stores double field weight, which stores the total weight of all the points in this cell. The initial call is \text{weightedRange}(R, \text{root}, \text{bbox}). The principal difference over the standard range counting query is that whenever the cell or point lies within the range, we add its weight (not just count) to the result.

\begin{verbatim}
double weightedRange(Rectangle R, KDNode p, Rectangle cell) {
    if (p == null) return 0 // fell out of the tree?
    else if (R.isDisjointFrom(cell)) // no overlap with range?
        return 0
    else if (R.contains(cell)) // the range contains our entire cell?
        return p.weight // include the weight of p's subtree
    else { // the range stabs this cell
        int result = 0
        if (R.contains(p.point)) // consider this point
            result += p.point.weight // include p's point's weight
        // apply recursively to children
        result += rangeCount(R, p.left, cell.leftPart(p.cutDim, p.point))
        result += rangeCount(R, p.right, cell.rightPart(p.cutDim, p.point))
        return count
    }
}
\end{verbatim}

(b) The code is structurally equivalent to the standard range-counting query. Thus, it visits exactly the same nodes as the standard range-counting query. Thus, the \(O(\sqrt{n})\) analysis applies here as well.

c) The helper is passed in the current best, and returns the updated best. The initial call is \text{frnn}(q, r, \text{root}, \text{bbox}, \text{null}). If we fall out of the tree or our cell is outside the query disk, we return best. Otherwise, we compute how close we need to be to be the new best, called viableDist. If the point in this cell is close enough, it replaces best. Finally, we recurse on our two children, updating best along the way.

\begin{verbatim}
Point frnn(Point q, double r, KDNode p, Rectangle cell, Point best) {
    if (p == null || cell.distanceTo(q) >= r) // not viable
        return best
    else {
        double bestDist = (best == null ? INFINITY : best.distanceTo(q))
        double viableDist = min(r, bestDist) // distance to be viable
        if (dist(q, p.point) < viableDist) // p.point is better?
            best = p.point // it's the new best
        // recurse on children
        best = frnn(q, r, p.left, cell.leftPart(p.cutDim, p.point), best)
        best = frnn(q, r, p.right, cell.rightPart(p.cutDim, p.point), best)
        return best
    }
}
\end{verbatim}

A further enhancement would be to order the recursive calls on the children favoring the side on which \(q\) lies.
**Solution 4:** See Fig. 1. The substring 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.

![Suffix Tree](image)

<table>
<thead>
<tr>
<th>Index</th>
<th>Substring ID</th>
<th>Index</th>
<th>Substring ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>baabaa</td>
<td>7</td>
<td>ababaa</td>
</tr>
<tr>
<td>1</td>
<td>aabaa</td>
<td>8</td>
<td>babaa</td>
</tr>
<tr>
<td>2</td>
<td>abaab</td>
<td>9</td>
<td>abaa$</td>
</tr>
<tr>
<td>3</td>
<td>baabab</td>
<td>10</td>
<td>baa$</td>
</tr>
<tr>
<td>4</td>
<td>aabab</td>
<td>11</td>
<td>aa$</td>
</tr>
<tr>
<td>5</td>
<td>ababab</td>
<td>12</td>
<td>a$</td>
</tr>
<tr>
<td>6</td>
<td>babab</td>
<td>13</td>
<td>$</td>
</tr>
</tbody>
</table>

Text: b a a b a a b a a b a a a $

Index | Substring ID | Index | Substring ID |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>aabaa</td>
<td>12</td>
<td>a$</td>
</tr>
<tr>
<td>4</td>
<td>aabab</td>
<td>0</td>
<td>baabaa</td>
</tr>
<tr>
<td>11</td>
<td>aa$</td>
<td>3</td>
<td>baabab</td>
</tr>
<tr>
<td>2</td>
<td>abaab</td>
<td>10</td>
<td>baa$</td>
</tr>
<tr>
<td>9</td>
<td>abaa$</td>
<td>8</td>
<td>babaa</td>
</tr>
<tr>
<td>7</td>
<td>ababaa</td>
<td>6</td>
<td>babab</td>
</tr>
<tr>
<td>5</td>
<td>ababab</td>
<td>13</td>
<td>$</td>
</tr>
</tbody>
</table>

**Figure 1:** Suffix tree.

**Solution 5:**

(a) 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 `printMaxK` function mentioned above, 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.

(b) 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.

**Solution 6:** We maintain two pointers $p$ (source) and $q$ (destination).

```c
(void*) compact(void* start, void* end) { // compact memory from start to end-1
    void* p = start; // p points to source block
    void* q = start; // q points to destination block
    while (p < end) {
        if (p->inUse) { // allocated block?
            memcpy(q, p, p->size); // copy to destination
            q->prevInUse = 1; // previous block is in-use
            q += p->size; // increment destination pointer
            // (no need to set q.size or q.inUse, since they are copied from p)
        }
        p += p->size; // advance to the next block
    }
    // everything copied - now q points to the remaining available block
    q->inUse = 0; // this block is available
    q->prevInUse = 1; // previous block is in-use
    int blockSize = p - q; // size of this final block
    q->size = blockSize; // set q.size
    *(q + q->size -1) = blockSize; // ... and q.size
    return q; // return pointer to this block
}
```

**Solution 7:**

(a) `boolean isValid(int k, int x)`: A block at level $k$ starts at address $x$ if $x$ is a multiple of $2^k$, or equivalently its $k$ lowest-order bits are all zero. That is, ($bitMask(k) \& x) == 0$.

(b) `int sibling(int k, int x)`: As given in class, this comes about by complementing the $k$-th order bit of $x$ (where the least significant bit is bit 0), that is, $(1<<k)^{x}$. For example:

```
parent(2, 12) = parent(2, 0011002) = 0001002 \& 0011002 = 0010002 = 8.
```

which means that blocks 24 and 28 are siblings at level 2. In Fig. 2, we show other examples. Blocks $0 = 0000002$ and $8 = 0001002$ are siblings at level 3 since they differ in bit position 3. Blocks 32 = 1000002 and 48 = 1100002 are siblings at level 4 since they differ only in bit position 4.

(c) `int parent(int k, int x)`: To obtain the parent, we zero out the $k$-order bit, that is, $\{\neg(1<<k) \& x\}$. For example:

```
p0011002 = parent(2, 0011002) = 0001002 \& 0011002 = 0010002 = 8.
```
(d) **int left(int k, int x):** The left child’s starting address is the same as the parent’s starting address, so this is just \( x \).

(e) **int right(int k, int x):** The right child’s starting address is the sibling of the left child’s starting address at level \( k - 1 \), that is \( (1<<((k-1)))^x \). For example:

\[
right(2, 12) = sibling(1, 12) = 000010_2 \& 001100_2 = 001110_2 = 14.
\]

**Solution 8:**

(a) **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 \text{null}, for a running time of \( n + 1 \).

(b) **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 \( \approx 1.5 \).

(c) **Expected \( O(m/(m-k)) \):** The probability that any element was erased is \( k/m \). Therefore, the probability that any accessed element is not erased is \( p = 1 - k/m = (m-k)/m \). Basic probability theory teaches us that if a coin has probability \( p \) of coming up heads, then in expectation, you will need to flip the coin \( 1/p \) times before seeing heads. In our case, this means that we expect to visit \( 1/p = m/(m-k) \) entries before finding an unerased entry.