CMSC 420: Fall 2022

Solutions to Practice Problems for the Final Exam

Solution 1:

(a) Only (iii) is true: Both (i) and (ii) are easily seen to be false once you have enough nodes. In an *inorder traversal*, internal and external node *alternate* with each other.

(b) (iii) and (v): A replacement node is needed whenever the node containing the deleted key has two non-null children. (Replacements are never needed for leaves and may not be needed for the root if it has only one child.) Selecting the replacement only from the right subtree can lead to less balanced trees over time, and so even though it is a common convention, selecting exclusively from the right subtree is not an optimal strategy.

It is a bit surprising to note that at most one replacement will be needed per deletion. A replacement node is either the largest key in the left subtree or the smallest key in the right subtree. Such a node can have at most one child. Hence, once a replacement is performed, the node to be recursively deleted has just a single child and does not need a replacement!

(c) 2 (assuming a double rotation is counted as two): 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.

(d) $O(\log n)$: In AVL-tree deletion, rotations may propagate from the leaf to the root. Since the tree has $O(\log n)$ height, this bounds the maximum number of rotations.

(e) Skew: The skew operation enforces the right-child constraint. (In contrast, the split operation is used to enforce the condition that if a node is red, then both its children are black.)

(f) A finger search is one where, rather than starting at the root of the tree, the search starts from an existing entry in the tree. This might be the last node visited in the previous search. For example, imagine that you want to look up "house" in a dictionary (book), but just prior to this you had looked up the word "hose". Ideally, the search should exploit the fact that you are already close to the target.

(g) 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, $m = 10$, $h(x) = 3$, and $g(x) = 5$, the probe sequence will consist of indices of the form $(3 + 5 \cdot i) \bmod 10 = \langle 3, 8, 3, 8, \ldots \rangle$. If these two positions are filled, then the insertion fails.)

(h) Hashing does not support ordered dictionary operations. Operations such as finding the largest, smallest, next-larger/smaller, and range searching are not efficiently supported by hash tables, but almost all of our tree-based structures support these in $O(\log n)$ time.

(i) The reason for storing the `size2` field is for the purpose of merging blocks together. When a used block becomes free, it needs to see whether the immediately preceding block is free. The `prevInUse` bit tells us whether it is free or not, but if it is we need to find its header. The `size2` field tells us the block's size, and by offsetting by that amount, we can find the previous block's header.

(j) 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.

```
void printMaxK(Node p, int k) {                    // print max 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.

```
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 `weightedRange(R, root, 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.

```
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
}
```

(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.

Solution 4: The approach follows the standard nearest-neighbor search, but we add the additional condition that we do not visit nodes whose cell lies outside the disk's radius, that is, if the distance between `q` and the cell exceeds the disk radius `r`.

The helper is the same as for the standard nearest-neighbor search, but it is also given the disk radius `r`. Note that the best point may be `null` if no point has been found within the query disk. Otherwise, it contains the closest point seen so far that is within the query disk.

The initial helper call is `frnn(q, r, root, bbox, null)`, where `root` is the root of the tree, `bbox` is the bounding cell for the entire tree, and `null` is the initial best point. We define the *viability region* to be the disk centered at `q` whose radius is the smaller of `r` and the best point seen so far. If we fall out of the tree or our cell is outside the viable region, we return `best`. Otherwise, we check the point in this node, and update `best` appropriately. Finally, we recurse on the children, favoring the child that is closer to `q`.

```
Point frnn(Point q, double r, KDNode p, Rectangle cell, Point best) {
    double bestDist = (best == null ? INFINITY : distance(q, best))
```

```
    double viableDist = min(r, bestDist)              // distance to be viable
    if (p == null || distance(q, cell) >= viableDist) // not viable
        return best

    if (dist(q, p.point) < viableDist)                // p.point is better?
        best = p.point                                // it's the new best
    Rectangle leftCell = cell.leftPart(cd, p.point)   // child cells
    Rectangle rightCell = cell.rightPart(cd, p.point)

    if (q[cd] < p.point[cd]) {                         // q is closer to left
        best = frnn(q, r, p.left, leftCell, best)      // try left then right
        best = frnn(q, r, p.right, rightCell, best)
    } else {                                           // q is closer to right
        best = frnn(q, r, p.right, rightCell, best)    // try right then left
        best = frnn(q, r, p.left, leftCell, best)
    }
    return best
}
```

Solution 5: 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.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Text:	b	a	a	b	a	a	b	a	b	a	b	a	a	$

Index	Substring ID	Index	Substring ID
0	baabaa	7	ababaa
1	aabaa	8	babaa
2	abaab	9	abaa$
3	baabab	10	baa$
4	aabab	11	aa$
5	ababab	12	a$
6	babab	13	$

Index	Substring ID	Index	Substring ID
1	aabaa	12	a$
4	aabab	0	baabaa
11	aa$	3	baabab
2	abaab	10	baa$
9	abaa$	8	babaa
7	ababaa	6	babab
5	ababab	13	$

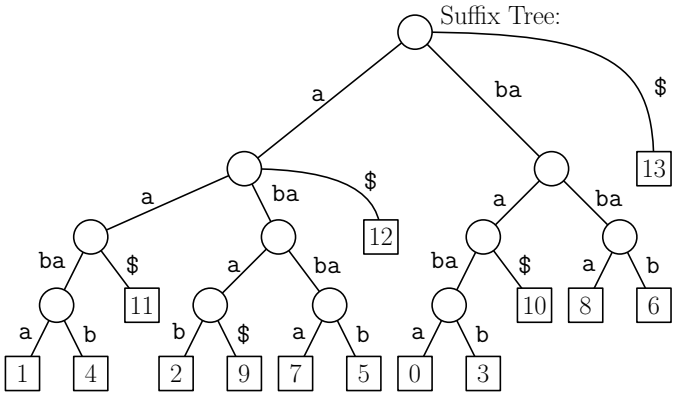

Figure 1: Suffix tree.

Solution 6:

(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 7: Before presenting the solution, let us make some useful observations. Recall that each segment has left and right endpoints (x_i^-, y_i) and (x_i^+, y_i) and the query segment has x-coordinate q_x, and the y-coordinates of the lower and upper endpoints are q_y^- and q_y^+.

First, in order to intersect the query segment we must have $q_y^- \leq y_i \leq q_y^+$, that is, the data-set segment's y-coordinate must lie between the upper and lower endpoints of the query segment. Next, $x_i^- \leq q_x$, since the left endpoint of the data-set segment must lie to the left of the query segment. Symmetrically, $x_i^+ \geq q_x$, since the right endpoint of the data-set segment must lie to the right of the query segment.

(a) We build a 3-layer range-tree structure, to test each of these conditions. (The order does not matter, since we are just counting. Note that we cannot just build a single layer for the x-conditions. This is because there are two independent x values, x_i^- and x_i^+ and each needs to be filtered independently against q_x.)

Let us treat each horizontal line segment as a point (y_i, x_i^-, x_i^+) in $\mathbb{R}^3$. Given the above observations, we can model the query segment as a 3-dimensional range, given by the three constraints

$$y_i \in [q_y^-, q_y^+], \qquad x_i^- \in [-\infty, q_x], \qquad \text{and} \qquad x_i^+ \in [q_x, +\infty].$$

This is just a standard 3-dimensional range tree, which has space $O(n \log^2 n)$. .

(b) Queries are answered as they would be for any standard 3-dimensional range tree. We search the first layer (sorted by y_i) to identify a set of $O(\log n)$ canonical nodes such that the points in these subtrees have y_i coordinates that define a disjoint cover of $[q_y^-, q_y^+]$. Next, for each node from the first layer, we access the 2nd layer auxiliary trees to identify a set of $O(\log n)$ canonical nodes such that the x_i^- coordinates of the these points form a disjoint cover of $[-\infty, q_x]$. Finally, for each of the nodes from the 2nd-layer search, we access the 3rd-layer auxiliary trees to identify a set of $O(\log n)$ canonical nodes such that the x_i^+ coordinates of the these points form a disjoint cover of $[q_x, +\infty]$. We sum the sizes of all these subtrees and return the result as the final answer.

This is just a standard 3-layer range tree search for n points, which has query time is $O(\log^3 n)$.

Solution 8: We maintain two pointers `p` (source) and `q` (destination). When we encounter an allocated block (`p.inUse`) we copy this block's contents to the destination. We set the `prevInUse` to 1, since we assume there will be no gaps after compression. We then increment the pointers to the source and destination by the block size. When we are done, we have one huge block leftover free block at the end. We set its `inUse` to 0, its `prevInUse` to 1, set its block sizes, and we return a pointer to the head of this block.

```
(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.size2
    return q;                               // return pointer to this block
}
```

Solution 9: This is all an exercise in bit manipulation.

(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 kth order bit of x (where the least significant bit is bit 0), that is, `(1<<k)^x`. For example:

$$\texttt{sibling}(2, 24) = \texttt{sibling}(2, 011000_2) = 000100_2 \ \texttt{^}\ 011000_2 = 011100_2 = 28.$$

which means that blocks 24 and 28 are siblings at level 2. In Fig. 2, we show other examples. Blocks $0 = 000000_2$ and $8 = 000100_2$ are siblings at level 3 since they differ in bit position 3. Blocks $32 = 100000_2$ and $48 = 110000_2$ are siblings at level 4 since they differ only in bit position 4.

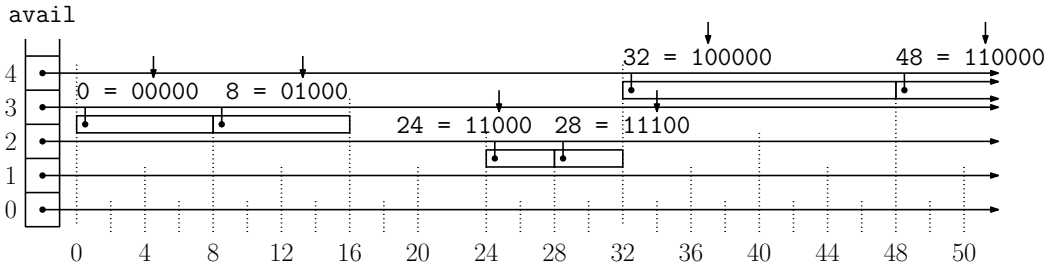

Figure 2: Buddies and bits.

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

$$\texttt{parent}(2, 12) = \texttt{parent}(2, 001100_2) = 000100_2 \ \texttt{\&}\ 001100_2 = 001000_2 = 8.$$

(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:

$$\texttt{right}(2, 12) = \texttt{sibling}(1, 12) = 000010_2 \ \texttt{\&}\ 001100_2 = 001110_2 = 14.$$

Solution 10:

(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 `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 (deposting 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 ≈ 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.

Solution 11:

(a) We will show that the amortized time $\alpha = 7/3 = 2.333\ldots$. Each time we perform an insertion, we receive α tokens. One of these tokens will be used to pay for the insertion, and the remaining $\alpha - 1$ are put in a bank account to pay for the next expansion. Let us assume that we have just expanded a table of size m resulting in a new table of size $m' = 4m$, which contains $3m/4$ entries. In order to induce the next expansion, the total number of entries must grow to $(3/4)m' = (3/4)(4m) = 3m$. This means that the number of new insertions is at least $3m - (3m/4) = (9/4)m$. Through these insertions we have collected $(9/4)m(\alpha - 1)$ tokens. We need to have enough tokens to pay the expansion cost, which is $3m$. Therefore, α must satisfy:

$$\frac{9m}{4}(\alpha - 1) \;\geq\; 3m \quad\Longrightarrow\quad \alpha \;\geq\; 1 + \frac{4}{3} \;=\; \frac{7}{3},$$

as desired.

Aside: We can generalize this. Let $0 < \lambda < 1$ denote the load factor when the expansion is triggered, and let $\beta > 1$ denote the expansion factor. Let us assume that we have just expanded a table of size m resulting in a new table of size $m' = \beta m$, which contains λm entries. In order to induce the next expansion, the total number of entries must grow to $\lambda m' = \lambda(\beta m)$. This means that the number of insertions is at least $\lambda\beta m - \lambda m = \lambda(\beta - 1)m$. Through these insertions we have collected $\lambda(\beta - 1)m(\alpha - 1)$ tokens. We need to have enough tokens to pay the expansion cost, which is $\lambda\beta m$. Therefore, α must satisfy:

$$\lambda(\beta - 1)m(\alpha - 1) \;\geq\; \lambda\beta m \quad\Longrightarrow\quad \alpha \;\geq\; 1 + \frac{\beta}{\beta - 1} \;=\; \frac{2\beta - 1}{\beta - 1}.$$

(It is interesting that the amortized cost does not depend on λ. When $\beta = 4$, this yields $\alpha = 7/3$, as expected.)

(b) To decrease the amortized cost, we should *increase* the expansion factor, since this reduces the frequency with which expansions take place (but does not increase their cost). This increase has the negative side effect that we may waste more space if we never fill up the expanded table. For example, if we expanded the table by a factor of 400 instead of 4, expansions would be very infrequent, but the final expansion could potentially waste a lot of space.