COMP_SCI 214 (Winter 2024)
Exam 2: Solutions, Explanations, and Grading

The purpose of this document is to elaborate on the answers to the first exam, explain how things were graded and why, provide some rationale for the whole process, and set up some guidelines regarding exam-related communication. Please read this document *in full* before contacting me regarding the exam!

Communication

- Any exam-related communication should go to me (Prof. Sruti) and be via a private post to me on Piazza.
- To confirm that you have read and understood this document in full, please begin your message with the text "I have read and understood the post-exam companion." just so we're all starting on the same page.

Answers and explanations

Here are the answers to the questions on the exam, explanations for why these answers are the correct ones, and why others are not. For reference, a pdf of the exam has been uploaded to Canvas as well.

Problem 1: What Algorithm Should You Use?

For each question, the two points for were distributed as follows: 1 point for the correct algorithm and 1 point for a correct justification.

1.i

Algorithm: The correct algorithm is "Breadth-First Search" but we also accepted "Dijkstra's" algorithm.

Justification: The key idea from this problem and goal are that we are looking for the shortest path between you and the history professor in question. Since the problem did not discuss edge weights, a breadth-first search is sufficient to find the shortest path between two nodes in a graph if they are reachable. (slide 10 of SSSP lecture) Dijkstra's is also suitable for this problem if we consider all edge weights to be equal, since its intended use is to find shortest paths between vertices.

If the algorithm was "Depth-First Search", though that was incorrect, we awarded a point for justification if it explained that DFS was better for distant nodes in the graph which seemed suitable given that history is far from CS. We did not award the point for justification if it said that DFS will find a path with fewer connections as that is not correct.

1.ii

Algorithm: The correct algorithm is "Kruskal's" algorithm.

Justification: The problem is essentially trying to find an MST of all the roads in Evanston as we want to make sure we pave enough roads such that connectivity is maintained (everyone can get to where they need to go) but with minimal plowing work (since there aren't enough trucks to plow all roads). Kruskal's algorithm is the only one of the options for finding a BST.

1.iii

Algorithm: The correct algorithm is "Dijkstra's" algorithm.

Justification: The problem can be described as a weighted graph where edges are legs and weights are costs. Since the goal is to find the shortest path between one location and another, Dijkstra's is suitable given that its purpose is to find shortest paths between nodes in a weighted graph.

Problem 2: Serving Relevant News

2.i

My answer was to maintain a dictionary where the key is a city name (string) and the value is a collection of all the articles pertaining to that city (e.g., linked list, array, stack). We also accepted answers that did the above and also maintained a second dictionary where the key is the article and the value is the city with which it is associated (though this second dictionary would be redundant). Any such answers were eligible for credit.

We did not accept other answers. We specifically did not accept dictionary answers where one city was associated with only one article (no plurality) and there wer no other ADTs specified. We also did not accept answers that mentioned a concrete data structure and not the ADT.

2.ii

My answer was to maintain a UUGraph where nodes are cities and edges exist between nodes if the two cities share a border. We also accepted WUGraph with similar information or a dictionary where the key is the city and the value is the collection of its neighboring cities. Any such answers were eligible for credit.

We did not accept other answers. We specifically did not accept answers that mentioned a concrete data structure and not the ADT.

2.iii

The completion of this function required 3 components which correspond to 1 point each: (1) correctly fetching and adding the city's articles to the final list, (2) correctly fetching all the city's neighbors and adding all their articles to the list, and (3) completing the above two with correct usage of the ADTs you specified from the previous two problems. If your implementation achieves the first two components but does not use your specified ADTs or your implementation was completely incorrect, you were not eligible for the third point.

We were not particular about syntax or things like natural number nodes, which is why I mentioned you could use pseudocode in the exam. Below is my pseudocode implementation which uses a dictionary of city to articles for (i) and a UU graph for (ii).

```
def get_relevant_articles(city: str?, ADT1, ADT2) -> Cons.ListC[Article?]:

    # ADT1 is: Dictionary of cities to a linked list of articles

    # ADT2 is: UUGraph of cities to neighboring cities

    # Populate relevant_articles with articles associated with city first
    let relevant_articles = None
    for article in ADT1.get(city):
        relevant_articles = cons(article, relevant_articles)

    # Now populate relevant_articles with neighbors' articles
    for neighbor in AD2.get_neighbors(city vertex):
        for article in AD1.get(neighbor):
            relevant_articles = cons(article, relevant_articles)

    return relevant articles
```

Problem 3: Priority Queue with Binary Search Trees

This problem tested your understanding of differentiating between ADTs and concrete data structures, i.e., how do you implement priority queue ADT operations using the BST invariants? The invariants for the BST were the same ones we saw in class (note that a left or a right child has to be a valid tree, where a valid tree is just the 1st invariant again, which means a child can be empty or not).

3.i

We discussed and drew out the approach for `find_min` in class (slide 17) when learning about Binary Search Trees (BSTs). Therefore, this problem is merely implementing that algorithm in code.

The approach to `find_min` is to start looking from the root, and just keep looking to the left child of the current node until we find the minimal element. Due to the second BST invariant, the minimal element can only be the left- and bottom-most element in the tree. If a node we are looking at does not have a left child, then this node is the minimal element.

This can be done recursively or iteratively. Since the given code did not have a loop, it has to be recursive. The base case is when we know to stop "looking", which in this case is when the node we are looking at does not have a left child according to the approach above. The recursive case is when the node we are looking at has a left child which means we are clearly not at the left-most element yet; in this case we recurse down the left-subtree of the node we're looking at. Below are two possible correct solutions.

```
def find_min(self) -> X:
    if self.size == 0:
        error('PQ is empty')
    def
        find_min_helper(node):
         if not node.left: #
             Base case
              return node.data
         else: # Recursive
             case
              return
                  find_min_helper(node.left)
    return
        find_min_helper(self.root)
```

```
def find_min(self) -> X:
    if self.size == 0:
        error('PQ is empty')
    def
        find_min_helper(node):
         if node.left: #
             Recursive case
              return
                  find_min_helper(node.left)
         else: # Base case
              return node.data
    return
        find_min_helper(self.root)
```

The two points were distributed as follows: 1 point for correctly implementing the base case and 1 point for correctly implementing the recursive case.

Notes on other answers we accepted:

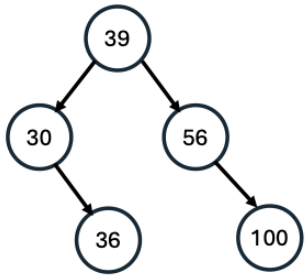
- Although there was no need to do any comparisons with `self.lt?`, we gave a point for the recursive case if it looked like this: `if self.lt?(node.left.data, node.data): return find_min_helper(node.left)`. The mistake here is that it doesn't consider that node.left might be `None`, but logically has the right idea. We did not award a point if the comparison flipped the order of items being compared.
- We were okay with returning `node` instead of `node.data` or if you referred to `node` as `root` throughout.

We did not accept any other answers.

3.ii

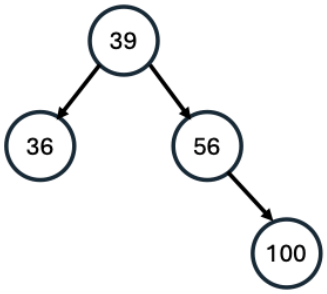
In this question, you were expected to do a simple BST insertion according to what we saw in class (slides 18-19), and for which we saw pseudocode. Note that there were only two invariants of the BST, and they did not include a height invariant, hence, no rotations should be done.

Below is the final BST after doing such an insertion. If your answer matched the below diagram exactly (there is only one possibility), it was eligible for full credit.



3.iii

In this question, you were expected to remove the minimal element from the BST and produce a valid BST after that removal. Since we know the minimal element is the left- and bottom-most element, the final tree should not have the node 30 in it. Below is a possible valid BST with 30 removed, however, we accepted any answer in which the node with 30 was removed and all the other invariants of the BST were satisfied.

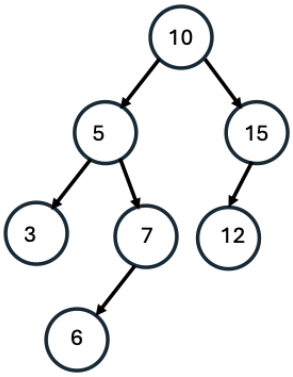


Any other answer was not eligible for credit. A common mistake we saw was when the tree removed the root element of 36 instead of the minimal element. This mistake likely arose from treating the BST like a binary heap, which is precisely what is being tested.

3.iv

This function seems to be recursive, hence to compute complexity, as we learned from class, we need to roughly count the number of recursive calls being made and see if there is a relation to the input size.

Consider the binary search tree below; we will use this as an example to reason about the complexity.



Below are all the recursive calls we do in order:

1. Call on root node 10
2. Call on 10's left child 5
3. Call on 5's left child 3
4. Call on 3's left child None
5. Call on 3's right child None
6. Call on 5's right child 7
7. Call on 7's left child 6
8. Call on 6's left child None
9. Call on 6's right child None
10. Call on 7's right child None
11. Call on 10's right child 15

12. Call on 15's left child 12
13. Call on 12's left child None
14. Call on 12's right child None
15. Call on 15's right child None

7 of those recursive calls are to add the 7 nodes into the final array so $O(n)$ work is being done here. The remaining recursive calls hit the base case if the node didn't exist (when the left or right child was recursively visited on a leaf node or a node with only one child). The number of leaf nodes in a complete tree is $(n+1)/2$ (a known formula I didn't expect you to know) and since we do 2 recursive calls for each leaf node, this amounts to an upper bound of $(8/2) * 2 = 8$ additional recursive calls. This formula is still linear in n, so $O(n)$ additional work is being done through these remaining recursive calls. There are a few additional recursive calls possible made on the right or left child of a node that only has one chlld. Those additional recursive calls will also be in order n.

Totally, we have $O(n + n + n) = O(n)$.

Another simpler way to reason about this is: we know that every time the function is called, as long as the base case is not hit, we are doing some work to add it into the array; so we can count the number of times we do that work to add an element into the final array (i.e., the number of recursive calls where the recursive case is hit). In the above calls, only 7 of the calls hit the recursive call which is directly n.

3.v

With a binary heap implementation, this function could simply return the internal array **if** it was already sorted. However, heap ordering only maintains ordering between a parent and its children, it doesn't guarantee any sortedness across all children in a level of the tree. So the internal array needs to be sorted using a sorting algorithm. The fastest sorting algorithm we saw ran in $O(nlog\ n)$, where heap sort might be the most intuitive since it can just use the priority queue's `insert` and `remove_min`.

If your answer correctly said $O(nlog\ n)$ **and** contained the above reasoning, it was eligible for credit. Any other answer was not eligible for credit.

Problem 4: Recording Paths in Graphs

This problem may feel reminiscent of Problem 2 in the practice exam. Just like in that problem, here we need to keep track of vertices that we are exploring in the current path (i.e., they are pushed onto the stack), and need to "untrack" vertices that are not part of the path we are exploring once we backtrack (i.e., they are popped off the stack).

There are three changes below that needed to be made to enable path recording. Each of these, if done correctly, was eligible for 1 of the 3 points.

- Inside `dfs_helper`, `v` needs to be pushed onto the stack before the check if we've arrived at our target. The two possible answers eligible for credit here were: insert `path.push(v)` between line 6 and 7 with same indentation as 7, or `path.push(v)` between line 7 and 8 with same indentation as line 7 or 8.
- Inside `dfs_helper`, `v` needs to be popped off the stack once we know that exploring all its neighbors was a dead-end. When it returns `False`, that's when we know it's about to backtrack to `v`'s predecessors making `v` no longer part of the current path being explored. Therefore, if your answer inserted between lines 14 and 15 with same indentation as line 15, a `path.pop()`, it was eligible for credit.
- `dfs` has a contract that returns an instance of a stack, and so we need to make sure this function returns the `path` stack. Since `path` is already defined within `dfs` and has been updated by `dfs_helper`, it should already contain the correct path. Therefore, the easiest and correct fix that was eliglible for credit was: replace line 18 with "`return path`" or "`if connected: return path else return ListStack()`". Since line 13 called `dfs_helper` and evaluated whether its true or false, we rely on the `bool?` contract set by this function, so any changes in returning from this function would have caused more issues. If all your changes to achieve this goal were within `dfs_helper` only, unless it handled

all return values (the contract, all three return statements, and checking the result of `dfs_helper` on line 13), it was not eligible for credit. If you deleted additional lines that would prevent the helper function from running (e.g., line 17), it was also not eligible for credit.

Seeing your graded exam

When we release the grades, we will also be giving you access to the graded scan of your exam on the GradeScope platform. There, you will be able to see your answers, and how they were graded.

The five possible outcomes for the exam—positive, neutral, single-negative, double-negative, or triple-negative modifier—will be denoted on Canvas as a score of 1, 0, -1, -2, and -3 on the exam assignment, respectively. The project improvement modifier will be a separate item in the gradebook.

If you have questions about the exam, see Section "Communication" of this document.

Did we make a mistake?

While we are very careful when grading exams, with multiple graders looking at each exam, and lots of double-checking along the way, it is possible we may have made a mistake when grading.

Things that are grading mistakes:

- Clerical errors
- Not giving credit for an answer that is *the same* or *exactly equivalent* as one that we explicitly said we accepted

Things that are *not* grading mistakes:

- Disagreeing with a decision we made
- Being unhappy with your grade

If you're not sure whether something is a grading mistake or not, feel free to ask us on Piazza.

If we *did* make a mistake when grading your exam, we will fix it. Please let us know by filing a grading correction request here: https://forms.gle/wyYRBtNDRfog5oiFA

Bear in mind:

- Grading correction requests must be targeted: we do not do blanket regrades of an exam.
- Grading correction requests which are not about genuine grading mistakes will be ignored.
- Because scores on the exam get turned into modifiers, not all grade deltas are significant. For example, a score of 13 and a score of 14 both correspond to a neutral modifier. Submitting a grading correction request to go from 13 to 14 is not useful.
- Decisions on grading corrections (or lack thereof) are final.

All correction requests must be in by *Friday March 15th, 5pm.* I won't be able to consider late requests.