**UNIVERSITY OF TORONTO**
**Faculty of Arts and Science**

# Midterm 2 Solutions CSC148H1F

**Duration: 50 min. Instructors**: Diane Horton, David Liu. **Examination Aids**: Provided aid sheet

---

## Name:

## Student Number:

---

**Please read the following guidelines carefully.**

- Please print your name and student number on the front of the exam.
- This examination has **3** questions. There are a total of **12 pages, DOUBLE-SIDED**.
- The last page is an aid sheet that may be detached.
- You may always write helper functions/methods unless explicitly asked not to.
- Docstrings are *not* required unless explicitly asked for.

---

Take a deep breath.

This is your chance to show us

How much you've learned.

We **WANT** to give you the credit

That you've earned.

A number does not define you.

| Question | Grade | Out of |
|---|---|---|
| Q1 | | 10 |
| Q2 | | 12 |
| Q3 | | 8 |
| **Total** | | 30 |

Use this page for rough work. If you want work on this page to be marked, please indicate this clearly *at the location of the original question.*

1. **[10 marks]** Point-form responses are acceptable here. You do not need to write a lot for full marks.

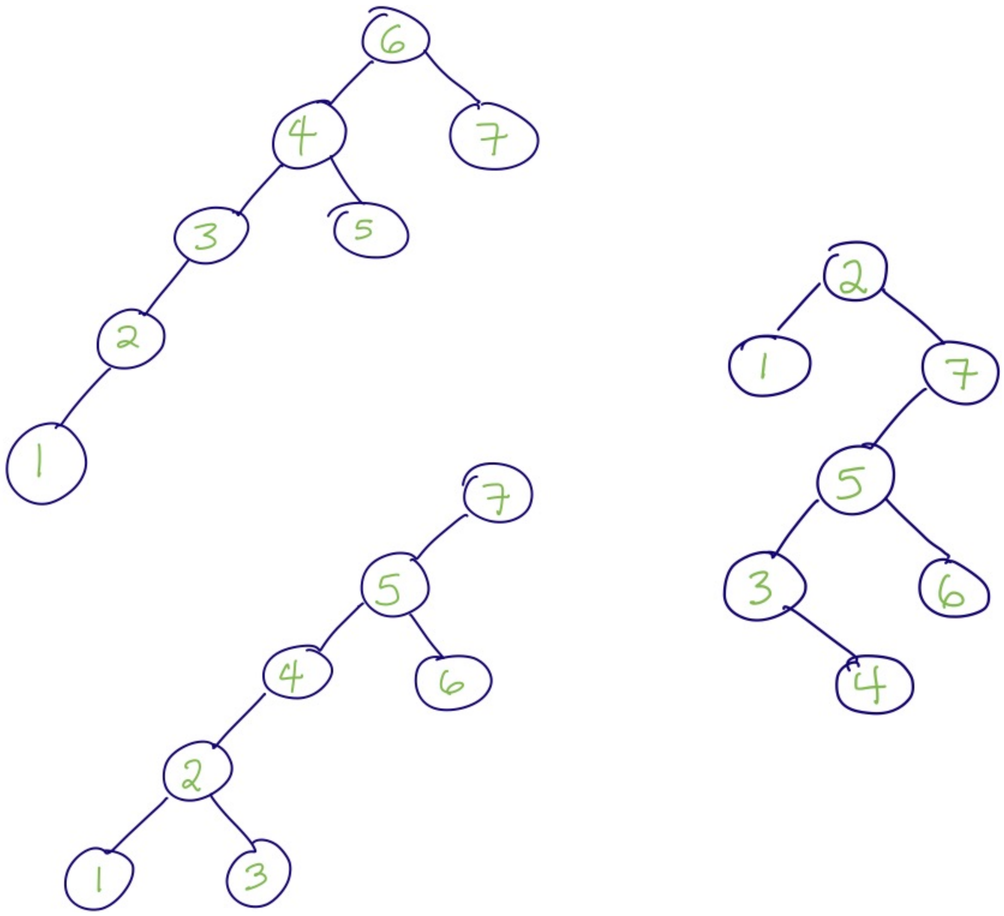
   (a) **[3 marks]** Recall that we defined the height of a tree in terms of nodes, so that the height of a tree with just a root is 1. Draw a binary search tree of height 5 containing only the numbers 1 through 7 inclusive.

   What is the greatest number of nodes we can have in a binary search tree of height 3? ________

   What is the fewest number of nodes we can have in a binary search tree of height 3? ________

**Solution**

For the binary seach tree: There are many possible answers. Here are three:



The easiest way to answer this question is probably to draw the shape and then add the values. Notice that placement of the values is constrained, once you have the shape.

The greatest number of nodes we can have in a binary search tree of height 3 is **7**.

The fewest number of nodes we can have in a binary search tree of height 3 is **3**.

**Marking scheme**

- **2 marks** for correct BST. Give 1 mark for partially correct answers, including errors such as:
  - Off-by-one in number of items.
  - Off-by-one in height.
  - A binary tree that violates the BST property for one or two nodes. (But give **0 marks** if the draw tree isn't even close to being a BST.)
- **0.5 marks** for each of the two numbers.

(b) **[1 mark]** Suppose we have a sorted `LinkedList`. Why would binary search be a bad technique for finding a given item?

**Solution**

Every time we want to make a comparison with an item at a particular index, we'd need to iterate through the linked list's nodes to get to that index. We'd also need to iterate through the entire linked list once just to compute the length of the linked list. This wouldn't save any time compared to just doing a linear search on the linked list.

**Marking scheme** **1 mark** for a reason that disucsses efficiency. No partial credit.

However, you can be generous in accepting reasons here. Something along the lines of "loop through at least half of the nodes in the linked list" should be enough for full marks. Students don't need to mention $\mathcal{O}(\log n)$ for Python list binary search, although you might see this come up.

(c) [**3 marks**] The aid sheet has docstrings for classes `LinkedList` and `Node`, which are relevant to this question.

Here is a linked list method you have seen before:

```
1   class LinkedList:
2       def remove(self, item: object) -> None:
3           """Remove the FIRST occurrence of <item> in this list."""
4           prev = None
5           curr = self._first
6           while curr is not None and curr.item != item:
7               prev, curr = curr, curr.next
8           # Rest of method omitted.
```

Whenever we say `blah.something`, we must know that `blah` is not `None`, otherwise the code will raise an error. For each of the following pieces of code, explain how we know that the value to the left of the dot is not `None` at the moment when it is evaluated.

Line 5: `curr = self._first`

**Solution**

`self` is a `LinkedList` object, and so it is not `None`.

Line 6: `curr.item != item`

**Solution**

The first part of the condition checks `curr is not None`, and the expression only evaluates if the first condition is true.

Line 7: `prev, curr = curr, curr.next`

**Solution**

This line is inside the `while` loop, and so the condition must be true; the condition includes the check that `curr` is not `None`.

**Marking scheme** **1 mark** per reason. You can give **0.5 marks** for unclear explanations if they're on the right track.

(d) [**3 marks**] Here is a (possibly incorrect) implementation of the `LinkedList` `remove` method.

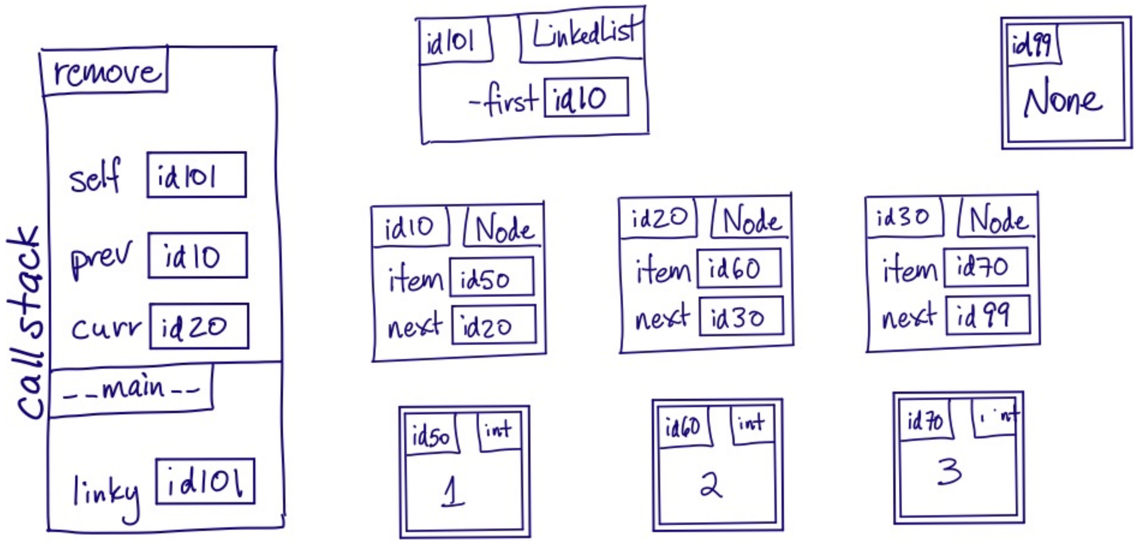
```
def remove(self, item: object) -> None:
    """Remove the FIRST occurrence of <item> in this list."""
    prev = None
    curr = self._first
    while curr is not None and curr.item != item:
        prev, curr = curr, curr.next

    # The diagram below shows the state of memory right before this line:
    curr = curr.next
```

Suppose we run the following code:

```
>>> linky = LinkedList([1, 2, 3])       # str(linky) == '[1 -> 2 -> 3]'
>>> linky.remove(2)
>>> # What would str(linky) return now?
```

Below, we have drawn a memory model diagram showing the state of the program's memory during the method call `linky.remove(2)`, immediately *before* the line `curr = curr.next`.

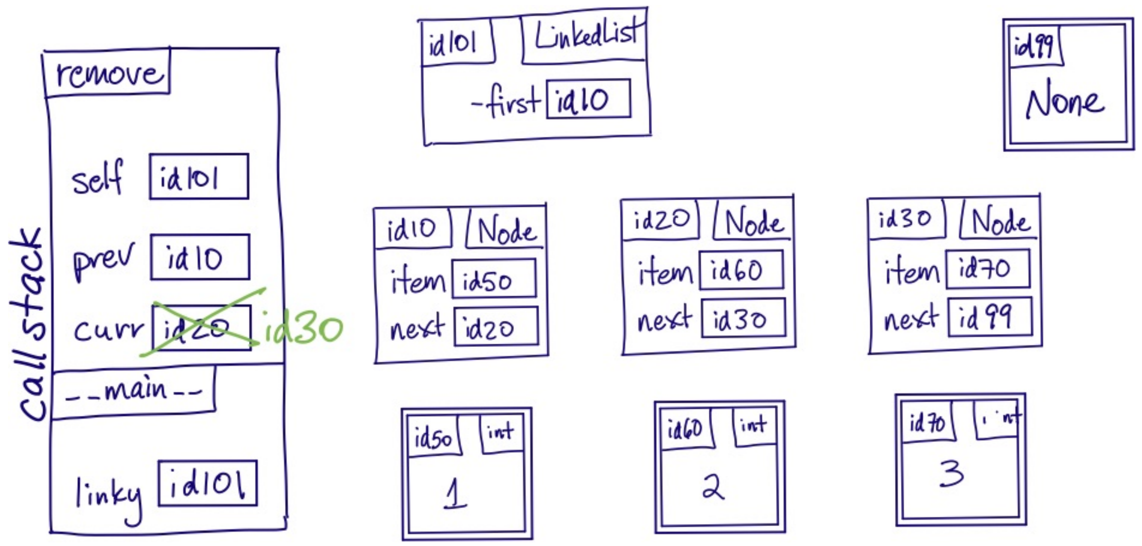


(i) Modify the diagram to show the state of memory immediately after the line `curr = curr.next` is executed.

(ii) In the space below, write what `str(linky)` would return if we called it *after* the call to `linky.remove(2)` is over.

**Solution**

The only change to the diagram is that `curr` now refers to `id30` instead of `id20`:



`str(linky)` would output `'[1 -> 2 -> 3]'`, i.e., the linked list wouldn't actually be mutated.

**Marking scheme**

- **1 mark** for changing `curr` on the diagram.
- **1 mark** for *not* changing anything else on the diagram.
- **1 mark** for the correct output.
  - No partial credit, including if students give the expected output of `'[1 -> 3]'`.

2. **[12 marks]** Consider the following nested list function. Suggestion: Drawing arcs between matching opening and closing list brackets may help you to see what it is counting.

```
def num_lists(obj: Union[int, List]) -> int:
    """Return the number of list objects in the given nested list.
    If obj is a list itself, include it in the count.

    >>> num_lists(4)
    0
    >>> num_lists([1, 2, 3])
    1
    >>> num_lists([1, [2], [[3, 4]]])
    4      # The four lists are:    [1, [2], [[3, 4]]],    [2],    [[3, 4]],    and    [3, 4].
    """
```

(a) **[3 marks]** Suppose we have the following nested list `lst` (we've added some extra whitespace for readability):

```
lst = [
    [1, [3, 4]],
    2,
    [],
    [5, 6, [[7]]]
]
```

Note that `lst` has length 4, and so we say that it has four sub-nested lists. Complete the table below.

**Solution**

| Sub-nested list of `lst` | Correct return value of `num_lists` on the sub-nested list |
|---|---|
| `[1, [3, 4]]` | 2 |
| `2` | 0 |
| `[]` | 1 |
| `[5, 6, [[7]]]` | 3 |

**Marking scheme**

- **1 mark** for the three correct sub-nested lists.
- **0.5 marks each** for the correct return values for `2` and `[]`.
- **1 mark** for the correct return value for `[5, 6, [[7]]]`.

(b) **[1 mark]** What should `num_lists(lst)` return if called with the value for `lst` shown in part (a)?

**Solution**

7

**Marking scheme** **1 mark** for correct answer. No partial credit here (even if answer is consistent with an incorrect table above).

(c) [**2 marks**] Explain, *in English*, how to compute `num_lists` on a nested list from the recursive calls on its sub-nested lists.

**Solution**

Add up all of the return values from the recursive calls on each sub-nested list, and then add one.

*Note*: of course, this only applies when the input is a list. If it's an integer, then 0 should be returned; no recursive calls are made.

**Marking scheme** **1 mark** for "add up recursive calls", and **1 mark** for adding one.

- Don't be afraid to deduct a mark if the statement is unclear. We're explicitly looking for good communication here.
- You can give **1 mark** for an otherwise incorrect answer, if consistent with parts (a) and (b).
- Students don't need to talk about the case when the input is an integer. But if they do, certainly reward this with the generic "Good!" comment.

(d) [**6 marks**] In the space below, implement the `num_lists` function using recursion. You may *not* define any helper functions here.

```
def num_lists(obj: Union[int, List]) -> int:
    """Return the number of list objects in the given nested list.
    If obj is a list itself, include it in the count.

    >>> num_lists(4)
    0
    >>> num_lists([1, 2, 3])
    1
    >>> num_lists([1, [2], [[3, 4]]])
    4  # The four lists are:   [1, [2], [[3, 4]]],   [2],   [[3, 4]],   and   [3, 4].
    """
```

**Solution**

```
    if isinstance(obj, int):
        return 0
    else:
        num = 1  # For the outermost list
        for lst_i in obj:
            num += num_lists(lst_i)
        return num
```

**Marking scheme**

- **2 marks** for base case.
  - **1 mark** for correct base case check (`isinstance`).
  - **1 mark** for correct return value in the base case.
  - Don't penalize students for using `type()` instead of `isinstance()`, but leave a comment.
- **4 marks** for recursive step.
  - **1 mark** for looping through `obj`.
  - **1 mark** for making the correct recursive call, and using it to compute the sum.
  - **1 mark** for counting `obj` itself. (We do this by initializing `num` to 1 instead of 0).
  - **1 mark** for a `return` statement with a correct value.
  - `return 1 + sum([num_lists(lst_i) for lst_i in obj])` is correct.

3. [**8 marks**] The aid sheet has the docstring for class `Tree`, which is relevant to this question.

Recall that the *depth* of an item in a tree is equal to the distance between it and the root inclusive, counting items. So the root of a tree has **depth 1**.

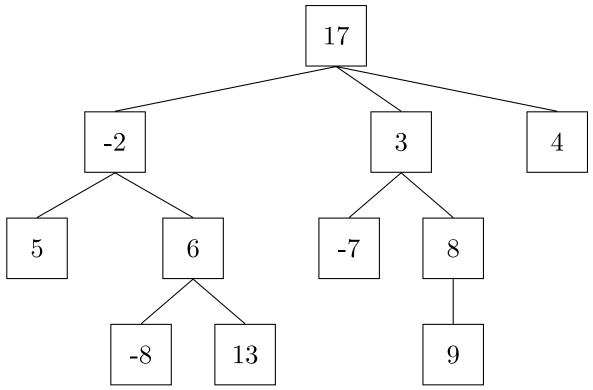
Consider the `Tree` method `truncate`, with the docstring below:

```
class Tree:
    def truncate(self, d: int) -> None:
        """Remove all values in the tree that are at depth <d> or greater.

        Precondition: d >= 1.

        Notes:
        1. Calling truncate with d = 1 always results in an empty tree.
        2. Calling truncate when d is greater than the tree's height (number of levels)
           does not change the tree at all.
        """
```

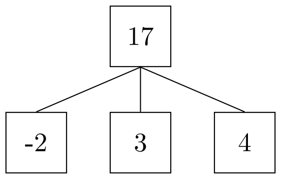
(a) [**2 marks**] Suppose we have a variable `t` that is a `Tree` instance representing the following tree:



Here are two calls to `t.truncate` with our initial tree `t`, shown above, but with different values of `d`. Below each call, draw what `t` would look like after the call. If the tree would be empty, write "empty".

`t.truncate(3)`    `t.truncate(1)`

**Solution**



empty

**Marking scheme 1 mark** per correct answer. No partial credit, except if *both* diagrams are off by one (i.e., think that we remove at depth `> d` rather than `>= d`), then you can give 1 out of 2.

(b) [**6 marks**] In the space below, implement the `truncate` method using recursion. You may not use any `Tree` methods other than `is_empty`, but you may access all `Tree` attributes. You may *not* define any helper methods here. We have given you some code to help you get started.

**Solution**

```
def truncate(self, d: int) -> None:
    if self.is_empty():
        pass  # 'return' is also acceptable here
    else:
        if d == 1:
            # Turn the tree into an empty tree.
            self._root = None
            self._subtrees = []
        else:
            # In this branch, we know that self is non-empty and d > 1.
            for subtree in self._subtrees:
                subtree.truncate(d - 1)

            # ALTERNATE VERSION, which removes empty subtrees from self._subtrees.
            # This is *not* required by the representation invariants given on
            # the aid sheet, but useful in practice.
            new_subtrees = []
            for subtree in self._subtrees:
                subtree.truncate(d - 1)
                if not subtree.is_empty():
                    new_subtrees.append(subtree)
            self._subtrees = new_subtrees
```

**Marking scheme**

- **1 mark** for correct behaviour when the tree is empty.
- **2 marks** for turning the tree into an empty tree.
  - Deduct **1 mark** if students only modify `self._root`, without modifying `self._subtrees` as well.
- **1 mark** for looping through `self._subtrees`.
- **1 mark** for the correct recursive call (with argument `d - 1`).
- **1 mark** for correctly handling the case when the tree is just a single node.
  - This can be handled with just the main loop, as in our solution (i.e., `self._subtrees` is empty). But students might handle it separately. If they do so correctly, they should get this mark, even if they don't have a loop or any recursive calls.
- **Deduct 1 mark** if students return something. If they write `return None`, leave a comment but don't deduct.
- Give **at most 4 marks** if students define a helper method. We expect this to be quite rare.
- Ask for guidance if you see students using a forbidden `Tree` method.
- If you can tell students are trying to remove empty subtrees from `self._subtrees` but they do so incorrectly, leave a comment but *don't* penalize this.

Use this page for rough work. If you want work on this page to be marked, please indicate this clearly *at the location of the original question.*