UNIVERSITY OF TORONTO
Faculty of Arts and Science

Midterm 2 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 **10 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? _____

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

(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`

Line 6: `curr.item != item`

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

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

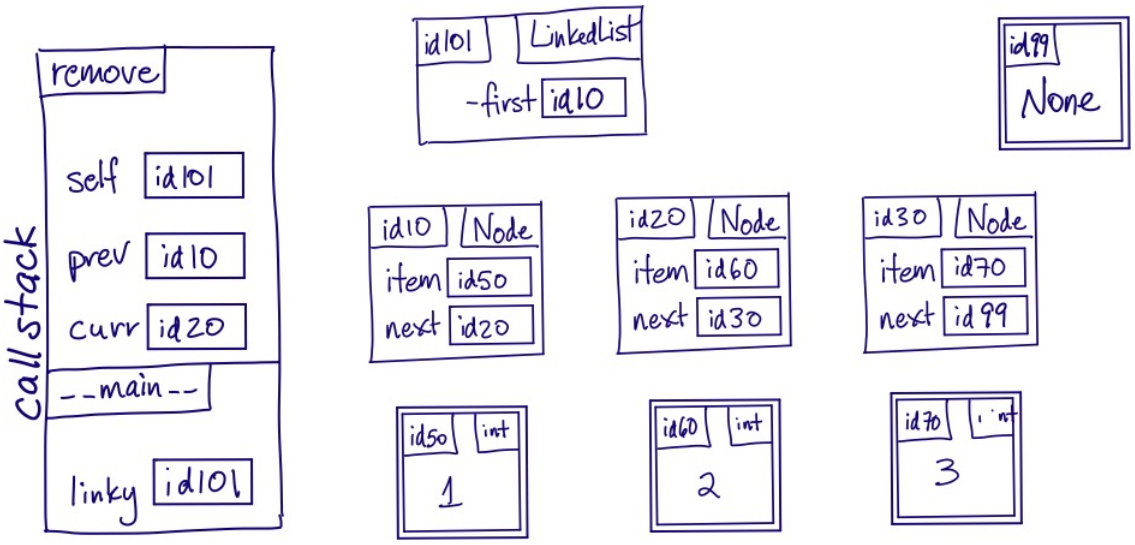
```
def remove(self, item: object) -> None:
    """Remove the FIRST occurrence of <item> in this list."""
    prev = None
    curr = self._first
    while curr is not None and curr.item != item:
        prev, curr = curr, curr.next

    # 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.

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.

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

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

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

(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].
    """
```

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**.

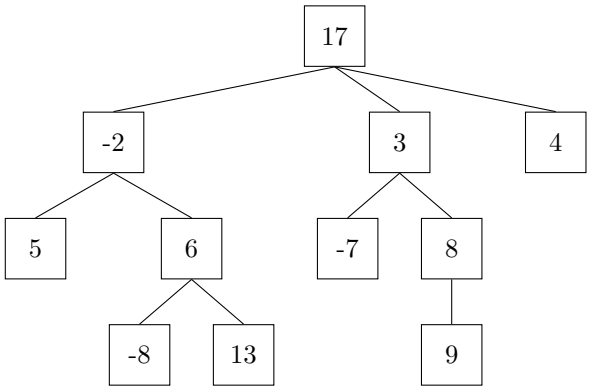
   Consider the `Tree` method `truncate`, with the docstring below:

```
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)`

(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.

```
def truncate(self, d: int) -> None:
    if self.is_empty():









    else:
        if d == 1:









        else:
            # In this branch, we know that self is non-empty and d > 1.
```

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.*