PLEASE HAND IN

PLEASE HAND IN

UNIVERSITY OF TORONTO
Faculty of Arts and Science

Final Exam

CSC 148H1S, Winter 2019
Horton and Smith

Duration — 3 hours
Aids allowed — Provided aid sheet

Student Number: |_|_|_|_|_|_|_|_|_|_|

UTORid: |_|_|_|_|_|_|_|_|

Last (Family) Name: ________________________

First (Given) Name: ________________________

Do **not** turn this page until you have received the signal to start.
In the meantime, please fill out the section above, and read the instructions below.

- **You must earn at least 40% on this final examination in order to pass the course. Otherwise, your final course grade will be no higher than 47%.**
- This exam consists of 12 questions on 36 pages, and is DOUBLE-SIDED.
- There is an aid sheet on a separate piece of paper. You do not need to hand it in, and nothing you write on it will be marked.
- There are blank pages at the end of the exam for rough work. **Do not remove them, and do not take the exam apart.**
- You may always write helper functions unless asked not to.
- Documentation is *not* required unless asked for.
- You may write in either pen or pencil.
- As a student, you help create a fair and inclusive writing environment. If you possess an unauthorized aid during an exam, you may be charged with an academic offense.
- Good luck! We want you to do well!

# 1:	______/ 4
# 2:	______/ 9
# 3:	______/ 7
# 4:	______/12
# 5:	______/ 6
# 6:	______/ 4
# 7:	______/ 8
# 8:	______/13
# 9:	______/ 6
# 10:	______/ 8
# 11:	______/ 8
# 12:	______/ 7
TOTAL:	______/92

Question 1. [4 MARKS]

Part (a) [2 MARKS]

Suppose we have imported the `LinkedList` class from the aid sheet and executed the following code:

```
linky = LinkedList([8, 1, 9, 5, 2])
linky2 = linky
```

What is the value of each of the following expressions? If it raises an error, explain why the error occurs.

Expression	Value (or description of error)
`linky.next.next.item`	
`linky._first.next._first`	
`linky is linky2`	
`linky._first is linky2._first`	

```
MARKING:

0.5 marks each.  No further division into quarter marks.
Students do not have to know the exact error message, just explain the problem.
```

Part (b) [2 MARKS]

Suppose we have imported the `BinarySearchTree` class from the aid sheet and executed the following code:

```
x = BinarySearchTree(5)
y = BinarySearchTree(10)
t = BinarySearchTree(8)
t._left = x
t._right = y
```

What is the value of each of the following expressions? If it raises an error, describe the error.

Expression	Value (or description of error)
`t._left._right._root`	
`t._right.is_empty()`	
`t._left._left is None`	
`t._right._left is t._left._right`	

```
MARKING:

0.5 marks each.   No further division into quarter marks.
```

Question 2. [9 MARKS]

Part (a) [2 MARKS]

What is the maximum possible height of a binary tree containing 100 nodes? ____________________

What is the minimum possible height of a binary tree containing 100 nodes? ____________________

```
MARKING:

1 mark each, no part marks.
The answer must be exactly right; off by one answers get zero.
```

Part (b) [2 MARKS] Suppose I can run this code without error:

```
    x = Blorp(14, ['a', 'b', 'c'])
    print(x)
    print(x[1])
```

Name the method or methods that we know must be defined in class `Blorp`.

```
MARKING:

1 mark for saying __getitem__
    0.5 if they are close to this name but not quite correct,
    but 0 if they forget double underscores
1 mark for saying __init__
    No part marks.  This must be named exactly correctly.

Students did not have to give parameters or other details.  Ignore these if they do.

Deduct 1 mark if they say any other method is required.  Many will say __str__.
```

Part (c) [2 MARKS] Could two different binary trees (they do not have to be binary **search** trees) have this as their post-order traversal: 4 5 2 6 7 3 1?

Circle one: Yes No

If yes, draw give an example of two such trees. If no, explain why not.

```
MARKING:

- Two marks for circling Yes and also showing two different trees with that post-order
  traversal.
      You will have to traverse the trees yourself to ensure they each have the given
      traversal order.
- No marks for just circling Yes.
- I can't see a reason for part marks, but if you do, let's discuss.
```

Part (d) [2 MARKS] Can a tree have a preorder traversal that is the same as its postorder traversal?

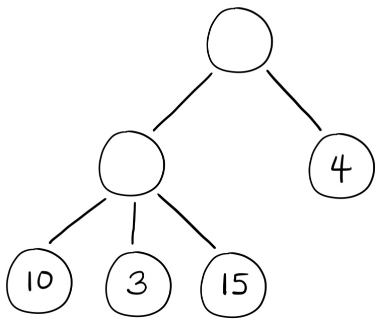
Circle one: Yes No

If yes, explain exactly which trees have this property. If no, explain why it's not possible.

```
MARKING:

Two marks for circling Yes and also saying either:
    Only one-node trees have this, or
    One-node and empty trees have this.
One mark for saying just that empty trees have this.
No marks for just circling Yes.
```

Part (e) [1 MARK] Recall from Assignment 2 that the nodes in a "treemapable" tree have a data size that must satisfy a certain rule. Fill in the missing data size values in the tree below so that it that violates this rule.



```
MARKING:

I mark for any answer where the root is not 32 and/or the left child is not 28.
They don't both have to violate the rule; it is sufficient for one to do so.
```

Question 3. [7 marks]

Consider this implementation of a helper method we defined as part of solving binary search tree deletion. It goes inside the `BinarySearchTree` class on the Aid Sheet. Unfortunately, it does not work.

```
    def extract_max(self) -> Any:
        """Remove and return the maximum item stored in this tree.

        Precondition: this tree is *non-empty*.

        >>> t1 = BinarySearchTree(20)
        >>> t2 = BinarySearchTree(30)
        >>> t3 = BinarySearchTree(40)
        >>> t2._left = t1
        >>> t2._right = t3
        >>> t4 = BinarySearchTree(50)
        >>> t4._left = t2
        >>> t4.extract_max()
        50
        >>> t4.items()
        [20, 30, 40]
        """
        if self._right.is_empty():
            max_item = self._root
            self = self._left
            return max_item
        else:
            return self._right.extract_max()
```

Part (a) [1 mark]

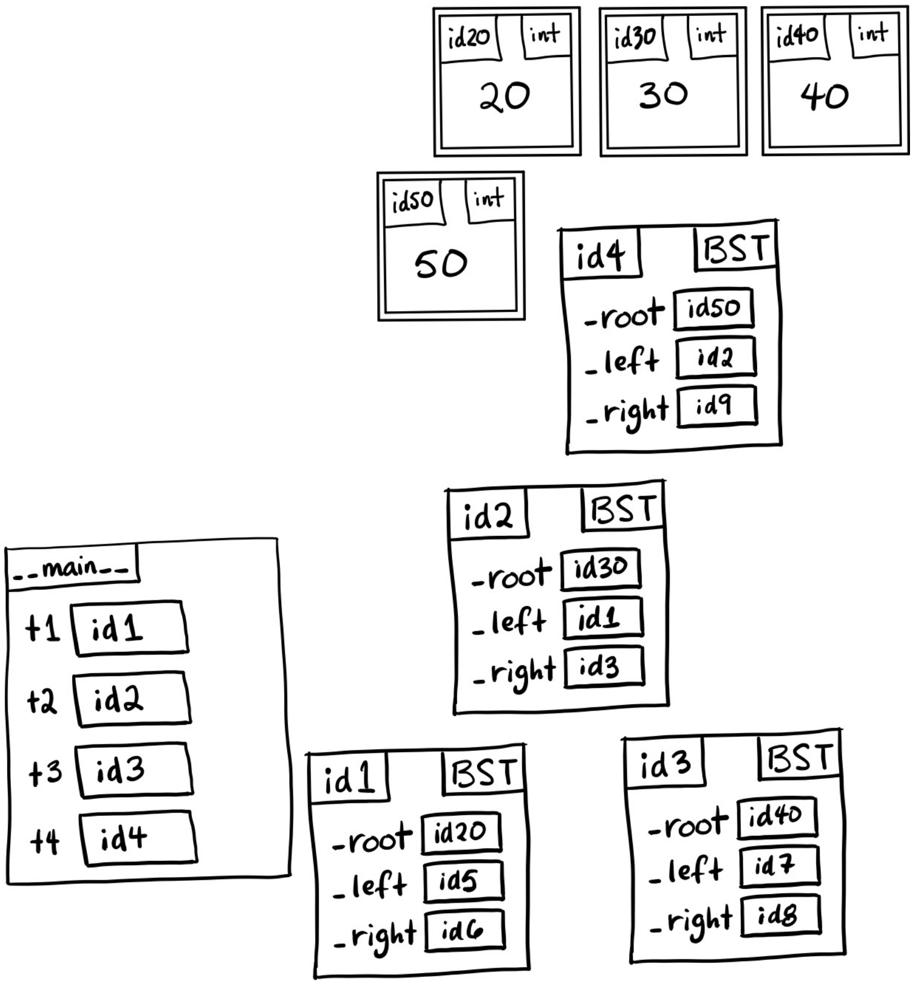
On the next page is a memory model diagram showing the state of memory right before the call to `extract_max` shown in the doctest example. To save space, we have shortened the name of the `BinarySearchTree` objects to `BST`. We have also not shown the objects that `id5`, `id6`, `id7`, `id8`, and `id9` refer to. What do you know about them?

```
MARKING:

It's okay to say just that each is an empty binary search tree -- mentioning "instance" is
not necessary.
If you see things that you think merit half marks, let's discuss.
```

Part (b) [3 marks]

On the next page, complete the diagram to show the state right before the method returns.



```
MARKING:

- 1 for new stack frame, properly labelled
- 1 for self with correct final value (it's okay if they don't show its initial value)
- 1 for max_item with correct value

Deduct for
- any additional changes (let's discuss)
```

Part (c) [3 marks]

Write a pytest test case that this method will fail but that a correct implementation would pass. Circle the line of code in your function where the failure will occur.

We have started the test case for you. Be sure to complete the function name and the brief docstring.

```
def test_extract_max_                                () -> None:
    """Test extract_max with
    """

MARKING:

- 1 for the setup:
  Give 0.5 if one of the three things below is incorrect or missing, and 0 if two or more are
      filling in both the method name and docstring with something reasonable.
          Both must describe the test case.
      constructing a non-empty BST.
      calling extract_max.
          It's okay if they do not save the return value, or if they save it and don't test it.
- 1 for the assert that fails:
    asserting that a mutation occurred and circling it as the source of failure.
- 1 for it being a test case that a correct implementation should pass

Deduct for
- syntax errors
```

Question 4. [12 MARKS]

You are working on software for a gym that offers exercise classes (we'll call these "sessions") and workshops. Sessions and workshops are similar, but have several differences: (1) Sessions have a level and workshops do not. (2) Sessions are taught weekly but workshops are taught once. (3) Workshop instructors must have at least one certification of any kind, but a session instructor must have the specific certification that is the name of the session. The following code demonstrates some features that we need:

```
# Create two instructors and record what they are certified to teach.
michelle = Instructor('Michelle')
michelle.add_certification('Pilates')
ruth = Instructor('Ruth')
ruth.add_certification('Pilates')
ruth.add_certification('Yoga')
# Define some sessions and workshops.
e1 = Session('Yoga', 1, 'Saturday', ruth)
e2 = Session('Pilates', 3, 'Thursday', michelle)
e3 = Workshop('Healthy spines', date(2019, 5, 3), michelle)
events = [e1, e2, e3]
# Print out a summary of each event.
for e in events:
    print(e)
```

Here is its expected output:

```
Level 1 Yoga session, with Ruth: every Saturday
Level 3 Pilates session, with Michelle: every Thursday
Healthy spines workshop, with Michelle: 2019-05-03
```

The code must raise an `UncertifiedInstructorError` where shown below:

```
e4 = Session('Yoga', 2, 'Monday', michelle)  # Error!
david = Instructor('David')
e5 = Workshop('Pilates for Runners', date(2019, 5, 5), david)  # Error!
```

On the next two pages, you will design the classes that are necessary for this code to run and produce the expected output. Your design must use inheritance effectively to capture common attributes and methods. Include the following in each class:

- A class docstring describing the class and its instance attributes. You likely won't need any representation invariants.
- Type declarations for all instance attributes. **All instance attributes must be private.**
- the header (including type contracts) for each public method in the class

Do not spend time writing any of these: method docstrings, method bodies (you don't even need to say `pass`) or doctests of any kind. There is one exception: If a method should be abstract, do write a method body so you can raise the appropriate error.

Begin your answer below. You may assume that all necessary imports have been done.

Continue your answer below.

MARKING:

Remember that method bodies, method docstrings, or doctests (on method or class) were NOT required, except that abstract methods were to raise the appropriate error. If they wrote any of these, ignore them.

- 4 marks for headers consistent with the client code we provided. At this point, ignore whether the design uses inheritance well, just look at whether it could run with the client code.
    - There are 4 classes that *must* be defined for the client code to work as shown (it can work without any abstract parent class): `Instructor`, `Session`, `Workshop`, `UncertifiedInstructorError`.
    - These methods must be defined in class `Instructor`:
        * an initializer for `Instructor` that takes a string,
        * `add_certification`, which takes a string,
        * a method (or two) that allows client code to determine whether the instructor has any certifications at all as well as whether they have a specific certification (needed for the initializers of the other classes),
        * a method that allows the client to retrieve the instructor's name (needed for the `get_schedule` methods of the other classes).
    - These methods must be defined in class `Session`:
        * an initializer for `Session` that takes a string, an int, a string and an `Instructor`,
        * `get_schedule`, which returns a string.
    - These methods must be defined in class `Workshop`:
        * an initializer for `Workshop` that takes a string, a `date` and an `Instructor`,
        * `get_schedule`, which returns a string.
    - There are NO methods that must be defined in class `UncertifiedInstructorError`.
    - Start with 4 marks.
        * Deduct 0.5 for each method that is not defined (within the appropriate class) or is defined but has an interface incompatible with the client code.
        * Deduct 2 overall if there are no type contracts for the parameters and return value, 1 overall if they are only partly there, and 0.5 if just one is missing.
        * Deduct 0.5 if class `UncertifiedInstructorError` is missing.
        * Do not go below 0 out of 4.
        * Ignore any method bodies that they may write.
- 3 marks for good use of inheritance
    - 1 mark for having an abstract class for all gym events, with subclasses for session and workshop. Give only 0.5 if the parent class is not abstract.
      If you think you've seen another good design, talk to us.
    - 1 mark for defining the initializer in the parent class, with parameters for the common attributes (name and instructor)
    - 1 mark for defining `get_schedule` in the parent class, with the appropriate signature. and making it abstract (writing the body that raises the correct error).
- 5 marks for class docstrings and attributes

  - 1 marks if each class has a docstring saying briefly what an instance of it represents in the domain
  - 1 mark for class `Instructor` defining attributes for the following, and with a clear English description and a type contract:
    * name
    * certifications
  - 2 marks for defining attributes for the following in the appropriate place, and with a clear English description and a type contract:
    * event name
    * event instructor
    * session level
    * session day
    * workshop date

    Start with 2 and deduct 0.5 for each one that is missing or is lacking a clear English description or a type contract.
  - 1 mark for making every attribute private.
- Deduct 1 mark if the interface reveals anything about the implementation.

Question 5. [6 MARKS]

Part (a) [1 MARK]

Suppose we call quicksort's partition helper function on the list `[12, 7, 10, 8, 13, 4, 19, 3, 1]`. Recall that it uses element 0 of the list as the pivot. What two lists will it return?

```
MARKING:

2 marks if completely correct (consider any order acceptable with each list)
1 marks if there is one tiny error like a typo or omitting one number
0 otherwise
```

Part (b) [2 MARKS]

Suppose we call **quicksort** with a list of 128 elements. Consider the recursive calls made in that very first call to **quicksort**.

What is the size of the smallest list it might recurse on? $\boxed{}$

What is the size of the *largest* list it might recurse on? $\boxed{}$

Suppose we call **mergesort** with a list of 128 elements. Consider the recursive calls made in that very first call to **mergesort**.

What is the size of the smallest list it might recurse on? $\boxed{}$

What is the size of the *largest* list it might recurse on? $\boxed{}$

```
MARKING:

0.5 marks each.  Must be exactly correct.  No further division into quarter marks.
```

Part (c) [1 MARK]

Which algorithm runs faster on a sorted list than on an unsorted list? Circle one.

1. mergesort
2. quicksort
3. both
4. neither

```
MARKING:

1 mark.  No part marks.
```

Part (d) [2 MARKS]

Suppose we want to sort a list of 1,000 elements. Could quicksort be slower than mergesort?

Circle one: Yes No

If yes, describe such a scenario. If no, explain why not.

```
MARKING:

Notice:
- being sorted in either order makes quicksort slower
- a partly sorted list would also make quicksort slower, I believe, but it is unlikely
  anyone will give that answer.
- If you see another answer that you think is plausible, let's discuss.

2 marks for circling yes AND describing a correct scenario clearly.
1 mark for circling yes AND describing a correct scenario but badly
    (yet you know what they meant)
0 marks otherwise
```

Question 6. [4 marks]

Below are the implementations of the stack and queue ADT that we have studied this term. For brevity, docstrings have been omitted.

```
class Stack:
    _items: List

    def __init__(self) -> None:
        self._items = []

    def is_empty(self) -> bool:
        return self._items == []

    def push(self, item: Any) -> None:
        self._items.append(item)

    def pop(self) -> Any:
        if self.is_empty():
            raise EmptyStackError
        else:
            return self._items.pop()


class Queue:
    _items: List

    def __init__(self) -> None:
        self._items = []

    def is_empty(self) -> bool:
        return self._items == []

    def enqueue(self, item: Any) -> None:
        self._items.append(item)

    def dequeue(self) -> Optional[Any]:
        if self.is_empty():
            return None
        else:
            return self._items.pop(0)
```

Consider the following functions, which use these `Stack` and `Queue` classes:

```
def flip(s: Stack, k: int) -> None:
    """Reverse the top k items on stack s.

    Precondition: k <= len(s)
    """
    temp = Queue()
    for i in range(k):
        temp.enqueue(s.pop())
    for i in range(k):
        s.push(temp.dequeue())

def jumble(s: Stack, size: int) -> None:
    """Mystery function.

    Precondition: size is the number of items in s.
    """
    for i in range(size):
        flip(s, i)
```

Part (a) [2 MARKS]

What is the big-oh time complexity of a call to `flip` on a stack of k items?

MARKING:

- 2 marks for $O(k^2)$
- 1 mark for $O(k^2 + k)$
- 0 marks for anything else, including $O(n^2)$. The answer must be in terms of k.

Part (b) [2 MARKS]

What is the big-oh time complexity of a call to `jumble` on a stack of n items? Don't forget to include the work done by the helper method `flip`.

MARKING:

- 2 marks for $O(n^3)$
- 1 mark for $O(k^2 \times n)$ – I'm not sure
- 0 marks for anything else, including $O(k^3)$.

Question 7. [8 MARKS]

Here is a method to be added to the `LinkedList` class on the aid sheet.

```
def insert_linked_list(self, other: LinkedList, pos: int) -> None:
    """Insert <other> into this linked list immediately before position pos.

    Do not make any new nodes, just link the existing nodes in.

    Preconditions:
        0 <= pos < len(self)
        len(other) >= 1

    >>> lst1 = LinkedList([0, 1, 2, 3, 4, 5])
    >>> lst2 = LinkedList([10, 11, 12])
    >>> lst1.insert_linked_list(lst2, 4)
    >>> str(lst1)
    '[0 -> 1 -> 2 -> 3 -> 10 -> 11 -> 12 -> 4 -> 5]'
    >>> lst3 = LinkedList([99])
    >>> lst1.insert_linked_list(lst3, 0)
    >>> str(lst1)
    '[99 -> 0 -> 1 -> 2 -> 3 -> 10 -> 11 -> 12 -> 4 -> 5]'
    """
```

On the next page, assemble the lines you see below to form a correct implementation of this method.

To help you get started, we have filled in the first line. (Do not change it.)

You must use every line below exactly once, and **you may not use any other code**. When you use the `while` loops, fill in the condition needed.

Line label	
a	`else:`
b	`self._first = other._first`
c	`i = 0`
d	`i += 1`
e	`last = other._first`
f	`last.next = self._first`
g	`curr.next = other._first`
h	`curr = self._first`
i	`if pos == 0:`
j	`while` *condition* `:`
k	`curr = curr.next`
l	`last = last.next`
m	`while` *condition* `:`
n	`last.next = curr.next`

Label each line using the labels on the previous page, and indent carefully.

Line label	Line
e	`last = other._first`

```
MARKING:

- 2 for finding the last node in other.  Start with 2 and deduct the amount shown
  for each feature that is missing or incorrect (and don't go below 0 out of 2):
      0.5 initialize
      1   iterate with correct condition
      0.5 advance
      1   do all of this before splitting into cases, since the result is needed either way
          (and we are restricted from repeating the code)
- 2 for handling the case where pos is zero.  Start with 2 and deduct the amount shown
  for each feature that is missing or incorrect (and don't go below 0 out of 2):
      0.5 check the condition
      1   connect the two lists
      1   update self._first (must be done after connecting the lists!)
- 4 for handling the general case. Start with 4 and deduct the amount shown
  for each feature that is missing or incorrect (and don't go below 0 out of 4):
      0.5 initialize curr
      0.5 initialize i
      2   iterate with correct condition
      0.5 advance curr
      0.5 advance i
      0.5 afterwards, update last.next
      0.5      and curr.next

If any line is used twice, or doesn't come from the list,
      cross it out and mark as if it wasn't there.

I'm not sure how this scheme will work out as this is a new type of problem.
Be SURE to discuss it with me once you've looked at a bunch.
```

Question 8. [13 MARKS]

One problem with linked lists is that to get to a desired index, we must traverse the list node by node. Suppose we added instance attributes to keep a reference to every k^{th} node. For instance, if k is 10, we would store a reference to node 0, node 10, node 20, and so on. We could use this to get to a desired index more quickly.

Below is the beginning of a new class that will take this approach. It uses the `_Node` class on the aid sheet, and allows client code to set the value for k when initializing.

Part (a) [6 MARKS]

On the next page, write the initializer.

```
class HoppingLinkedList:
    """A linked list implementation of the List ADT with a reference to every kth item.
    """
    # === Private Attributes ===
    # _refs:
    #     References to every kth node in the list.
    # _k:
    #     The difference between the indexes of the referenced nodes.  For example,
    #     if _k is 3, we store references to the nodes at indexes 0, 3, 6, and so on.
    #     We do not store anything for indexes that are out of range of the list.
    #
    # === Representation Invariants ===
    # _k >= 1

    _refs: List[_Node]
    _k: int

    def __init__(self, k, items: list) -> None:
        """Initialize a new linked list containing the given items. The first item in
        <items> goes at the front.

        Precondition: k >= 1.

        >>> h = HoppingLinkedList(3, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
        >>> len(h._refs) == 4
        True
        >>> h._refs[0].item == 0
        True
        >>> h._refs[1].item == 3
        True
        >>> h._refs[2].item == 6
        True
        >>> h._refs[3].item == 9
        True
        """
```

Write the initializer body here:

```
MARKING:

1 for saving k
1 for initializing refs to an empty list

4 marks for building the structure, either
(a) by hand
0.5 for iterating through the items (by value or index) correctly
0.5 making a new node each time and saving the value in it
0.5 linking in the new node correctly
0.5 for handling the case of the first reference properly
2 for appending to refs if i is divisibly by k

or (b) using the LinkedList initializer
1 for building a LinkedList
1 for iterating correctly over it
      0 marks if they set their "curr" variable to the LinkedList itself rather than it's first
2 for appending to refs if i is divisibly by k
```

Part (b) [4 MARKS]

Now consider this method:

```
    def __getitem__(self, index: int) -> Any:
        """Return the item at position <index> in this list.  Does not handle slices,
        only integer indices.
        Raise an IndexError if <index> is < 0 or <index> is >= the length of this list.
```

Assume that it uses the `_refs` instance attribute well so that it is faster than a regular linked list.

1. If k is 100, what is the greatest number of elements of `_refs` that must be examined on any given call to `__getitem__`?

   If k is 100, what is the greatest number of nodes that must be examined on any given call to `__getitem__`?

2. If k is 10, what is the greatest number of elements of `_refs` that must be examined on any given call to `__getitem__`?

   If k is 10, what is the greatest number of nodes that must be examined on any given call to `__getitem__`?

3. If k is 1, what is the greatest number of elements of `_refs` that must be examined on any given call to `__getitem__`?

   If k is 1, what is the greatest number of nodes that must be examined on any given call to `__getitem__`?

4. Considering efficiency, what is the big advantage of setting k to 1, that is, keeping a reference to every node in the list?

   Considering efficiency, what is the big disadvantage of doing so?

```
MARKING:

0.5 for each question.

For numbers, the answer must be exact.
No further division into quarter marks if off by one, for instance.

For part 4, you will see many variations.  Keep a record of decisions and
discuss with me any case where the mark is not obvious.
```

Part (c) [3 MARKS]

Suppose that instead of making `HoppingLinkList` a brand new class, we made it a subclass of the `LinkedList` class defined on the aid sheet. And let's say that we're going to define `__getitem__` so that it uses `_refs` to run faster than in a regular linked list, but that we want to postpone writing any other code for now. We just want the class to work, and will worry about other efficiencies later.

Which of these inherited methods *must* we implement now, regardless?

`__init__`: Yes No Explain:

`insert`: Yes No Explain:

`delete`: Yes No Explain:

```
MARKING:

1 mark each for circling yes AND giving a reasonable explanation.
They don't need a lot of detail.
```

Question 9. [6 MARKS]

Suppose we want a function that will take a Python list and return a new list that has the same elements in reverse.

```
def reverse(lst: List) -> List:
    """Return a list that is the reverse of lst.  Do not mutate lst.

    >>> lst = [1, 2, 3, 4, 5]
    >>> new = reverse(lst)
    >>> new
    [5, 4, 3, 2, 1]
    >>> lst
    [1, 2, 3, 4, 5]
    """
```

Below are several implementations of `reverse`, all of which work. Beside each, write in the box its big-oh time complexity for a list of length n.

```
    answer = []
    for item in lst:
        answer.insert(0, item)
    return answer
```

```
    answer = []
    # This use of range counts down from len(lst) - 1 to 0.
    for i in range(len(lst) - 1, -1, -1):
        answer.append(lst[i])
    return answer
```

```
    # This uses the implementation of the Stack class given in Question 6.
    s = Stack()
    for item in lst:
        s.push(item)
    answer = []
    while not s.is_empty():
        answer.append(s.pop())
    return answer
```

```
MARKING:

2 marks each, for a fully correct answer; no part marks;
if you want to give part marks, let’s discuss.

Notes:
- constants must be omitted
- lower order terms must be omitted
```

Question 10. [8 MARKS]

Recall the TMTree class from Assignment 2. Here is the class docstring for a simplified version of TMTree.

Note that we have removed some of the attributes of TMTree that were present in Assignment 2. We have removed them to help guide you to a simpler solution, and you should not use those missing attributes in your solution here.

```
class TMTree:
    """A TreeMappableTree: a tree that is compatible with the treemap visualiser.

    === Private Attributes ===
    _name:
        The root value of this tree, or None if this tree is empty.
    _subtrees:
        The subtrees of this tree.
    _expanded:
        Whether or not this tree is considered expanded for visualization.

    === Representation Invariants ===
    - If _name is None, then _subtrees is empty. This represents an empty tree.

    - if _subtrees is empty, then _expanded is False
    - if _expanded is True, then _expanded can be either True or False for
      every tree in _subtrees
    - if _expanded is False, then _expanded is False for every tree in _subtrees
    """
```

We want to be able to verify that the Representation Invariants concerning _expanded hold for the entire tree rooted at a given tree. That is, that the RI holds for this tree, its subtrees, its subtrees' subtrees, and so on. On the next page, write the valid_expanded method, to be added to the TMTree class, that confirms that this holds.

This method should not mutate the tree — it just reports whether the Representation Invariant holds or not.

You can add private helper methods if you wish, and you may find it helpful to do so here.

```
    def valid_expanded(self) -> bool:
        """Return True iff the _expanded values in this tree satisfy the
        Representation Invariants concerning _expanded.
        """

MARKING:

Base case
1 mark - empty tree
1 mark - leaf case is correct
Recursive case
        1 mark - different cases depending on self._expanded
2 marks - if self._expanded, checks for valid expanded subtrees
3 marks - if not self._expanded, checks for all _expanded == False in subtrees
        deduct 1 for wrong base case

Notes
If there is another base case, do not penalize as long as it works
        Deduct if the solution mutates the tree
```

Question 11. [8 MARKS]

A nested list is defined to have "consistent depth" if it is an integer, or it is a list in which (1) each element has consistent depth, and (2) the elements have the *same* depth. Write the function below that checks for consistent depth. Read the doctest examples carefully to make sure you understand the task.

Hint: The interface to the function does not convey all of information that you need during the recursion. On the next page, write a helper function that provides the necessary information.

```
def consistent_depth(obj: Union[int, list]) -> bool:
    """Return True iff obj is nested to a consistent depth
    throughout.

    >>> consistent_depth(6)
    True
    >>> consistent_depth([1, 2, 3, 4])
    True
    >>> consistent_depth([1, 2, [3], 4])
    False
    >>> consistent_depth([[1], [2, 3], [4]])
    True
    >>> consistent_depth([1, [2, 3], 4])
    False
    >>> consistent_depth([[[1]], [[2], [3], [4], []]])
    True
    >>> consistent_depth([[1], [[2], [3], [4], []]])
    False
    """
```

Write your helper function here. Including a docstring will help us give part marks if your function is not completely correct.

```
MARKING:

=== Overall design:

2 marks for defining a helper function with an interface that reports both the boolean and the dep
      These can be combined (as we did) or separate (as we discussed).
      They do not have to write a docstring.

1 mark for consistent_depth using the helper to report the correct answer.

=== Inside the helper:

0.5 for correctly handling the case where obj is a list

0.5 for correctly handling the case where obj is an empty list

4 for the recursive case:
      1 for recursing on each sub-nested list correctly
      1 for returning a value indicating No if any sub-nested list does so
      1 for returning a value indicating No if the depths returned by ALL recursive calls
            don’t match.
            This requires keeping track of one of the depths to benchmark against
      1 for returning the correct depth

Additionally
      If there is another base case, do not penalize as long as it works
      If there are problems not captured in this marking scheme, deduct.  Let’s discuss
```

Question 12. [7 marks]

Here's an implementation of a `__contains__` method for the `Tree` class on the aid sheet:

```
def __contains__(self, item: Any) -> bool:
    """Return True iff this tree contains <item>.

    >>> t = Tree(1, [Tree(-2, []), Tree(10, []), Tree(-30, [])])
    >>> t.__contains__(-30)
    True
    >>> t.__contains__(148)
    False
    """
    if self.is_empty():
        return False
    elif self._subtrees == []:
        return self._root == item
    else:
        found = False
        for subtree in self._subtrees:
            found = subtree.__contains__(item)
            if found:
                return True
        return False
```

Part (a) [1 mark] Give an example of the special syntax that `__contains__` allows us to use.

```
MARKING:

No part marks are likely, but if you think you have a case for some, let's discuss.

Must use "in" exactly, with the proper syntax.

They do not have to define whatever tree they are using in the expression, as long as
it's obvious that they mean it to be a tree.
```

Part (b) [4 MARKS] Our implementation of `__contains__` does not always do what its docstring says. Below, define a doctest example that it will fail and one that it will pass. Both examples should pass on a correct implementation of `__contains__`. To assist us, draw the tree that each doctest defines.

Doctest that will fail: Tree:

Doctest that will pass: Tree:

```
MARKING:

Two marks for each:
      1 for a tree diagram and (in the doctest) a test of a leaf / non-leaf
      1 for (in the docstest) properly building the tree.
            They can't get this second point if the test case is incorrect.

If they don't draw the tree, don't deduct; just figure it out from the doctest.
```

Part (c) [2 MARKS]

Correct the method, using the simplest changes possible. Write your answer directly on the code above.

```
MARKING:

1 for adding a condition that returns True if the root is the item
      They can remove the original elif, or
      leave it in and add the new condition either before or after it.
1 for changing nothing else
```

[Use the space below for rough work. This page will not be marked unless you clearly indicate the part of your work that you want us to mark.]

[Use the space below for rough work. This page will not be marked unless you clearly indicate the part of your work that you want us to mark.]

[Use the space below for rough work. This page will not be marked unless you clearly indicate the part of your work that you want us to mark.]