CSC111 Lecture 2: Introduction to Linked Lists

Hisbaan Noorani

January 13, 2021

Contents

Here are the implementations of the `LinkedList` class and the `_Node` class you saw in lecture.

```python
from __future__ import annotations
from dataclasses import dataclass
from typing import Any, Callable, Iterable, Optional, Union


@dataclass
class _Node:
    """A node in a linked list.

    Note that this is considered a "private class", one which is only meant
    to be used in this module by the LinkedList class, but not by client code.

    Instance Attributes:
      - item: The data stored in this node.
      - next: The next node in the list, if any.
    """
    item: Any
    next: Optional[_Node] = None  # By default, this node does not link to any other


class LinkedList:
    """A linked list implementation of the List ADT.
    """
    # Private Instance Attributes:
    #   - _first: The first node in this linked list, or None if this list is empty.
    _first: Optional[_Node]

    def __init__(self) -> None:
```

```
        """Initialize an empty linked list.
        """
        self._first = None
```

1 Exercise 1: The `LinkedList` and `_Node` classes

The following Python code creates three `_Node` objects and an empty `LinkedList` object.

```
>>> node1 = _Node('a')
>>> node2 = _Node('b')
>>> node3 = _Node('c')
>>> linky = LinkedList()  # linky is empty
```

1. Write code below that uses the above four variables to make `linky` refer to a linked list containing the elements `'a'`, `'b'`, and `'c'`, in that order.

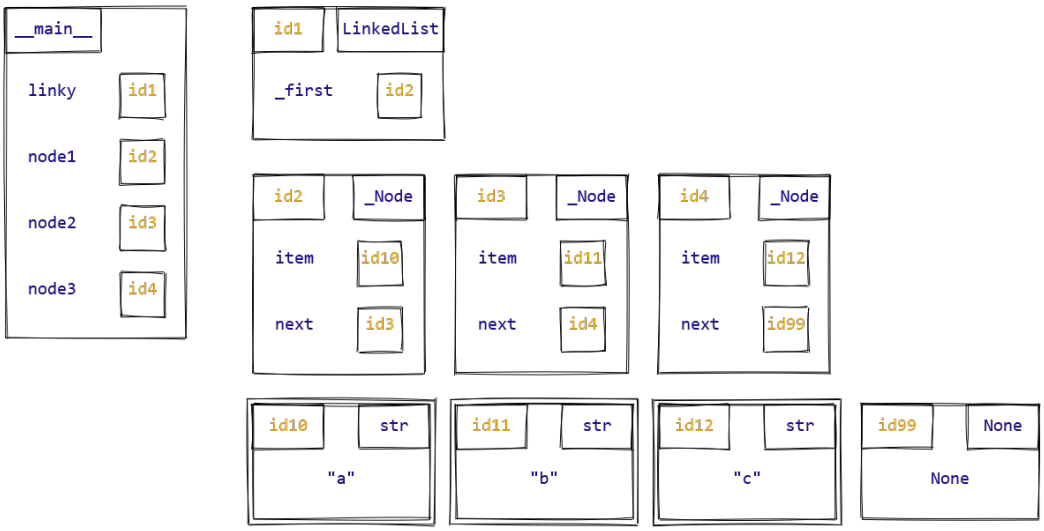
```
>>> node1.next(node2)
>>> node2.next(node3)
>>> linky._first(node1)
```

2. Assume you have executed your code from the previous question. Write the value of each of the following Python expressions. If there would be an error raised, describe the error, but don't worry about matching the exact name/wording as the Python interpreter.

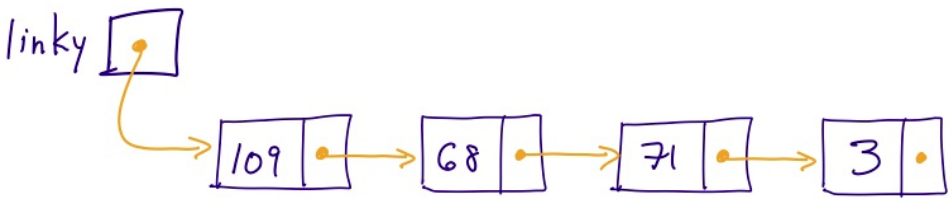
```
>>> node1.item
'a'

>>> node1.next is node2
True

>>> node1.next.item
'b'

>>> node1.next.next.item
'c'

>>> node1.item + node2.item + node3.item
'abc'

>>> node1 + node2 + node3
TypeError: unsupported operand type(s) for +: '_Node' and '_Node'

>>> type(linky)
<class 'LinkedList'>

>>> type(linky._first)
```

```
<class '_Node'>

>>> type(linky._first.item)
<class 'str'>

>>> linky._first.next.next.item
'c'

>>> linky._first.next.next.next.item
Error
```

3. Complete the memory model diagram below to show the state of memory from the above example. Remember that you can make up your own id values, as long as they're all unique. (If you are working on paper, you'll need to redraw the diagram. That's okay, take your time!)



4. Draw an abstract diagram of the linked list `linky`, using the style from lecture (with boxes and arrows).



2 Exercise 2: Linked list traversal

Recall the basic *linked list traversal pattern*:

```
curr = self._first

while curr is not None:
    ... curr.item ...  # Do something with curr.item
    curr = curr.next
```

In this exercise, you'll implement three new methods using this pattern.

1. Implement the following linked list method.

```
import math


class LinkedList:
    def maximum(self) -> int:
        """Return the maximum element in this linked list.

        Preconditions:
            - every element in this linked list is a float
            - this linked list is not empty

        >>> linky = LinkedList()
        >>> node3 = _Node(30.0)
        >>> node2 = _Node(-20.5, node3)
        >>> node1 = _Node(10.1, node2)
        >>> linky._first = node1
        >>> linky.maximum()
        30.0
        """
        largest_so_far = -math.inf
        curr = self._first

        while curr is not None:
            if largest_so_far < curr.item:
                largest_so_far = curr.item

            curr = curr.next

        return largest_so_far
```

2. David has attempted to implement the `LinkedList.__contains__` method below, but unfortunately his program has a bug.

```
class LinkedList:
    def __contains__(self, item: Any) -> bool:
        """Return whether item is in this linked list.

        >>> linky = LinkedList()
```

```
        >>> linky.__contains__(10)
        False
        >>> node2 = _Node(20)
        >>> node1 = _Node(10, node2)
        >>> linky._first = node1
        >>> linky.__contains__(20)
        True
        """
        curr = self._first

        while curr is not None:
            if curr == item:
                # We've found the item and can return early.
                return True

            curr = curr.next

        # If we reach the end of the loop without finding the item,
        # it's not in the linked list.
        return False
```

(a) What is the error in the above implementation?

`if curr == item:` → `if curr.item == item:`

(b) Which doctest example(s) will fail because of this error?

Anything that is expecting `True` will return false as `curr` will almost never be equal to `item`.

(c) How should we fix this error?

`if curr == item:` → `if curr.item == item:`

3. Finally, let's look at one more `LinkedList` method, `__getitem__`:

```
class LinkedList:
    def __getitem__(self, i: int) -> Any:
        """Return the item stored at index i in this linked list.

        Raise an IndexError if index i is out of bounds.

        Preconditions:
            - i >= 0
        """
```

To implement this method, we're going to need two variables: `curr`, to keep track of the current `_Node`, and `curr_index`, to keep track of the current *index*.

Implement this method below by using an *early return* inside the loop body, similar to `LinkedList.__contains__` above. We've started the implementation for you.

```
def __getitem__(self, i: int) -> Any:
    """Return the item stored at index i in this linked list.

    Raise an IndexError if index i is out of bounds.

    Preconditions:
        - i >= 0
    """
    curr = self._first
    curr_index = 0

    while curr is not None:
        if curr_index == i:
            # If we reach the right index, return the item
            return curr.item

        curr_index += 1
        curr = curr.next

    # If we reach this point, the list has ended BEFORE
    # we reach index i.
    raise IndexError
```

3 Additional exercises

1. Implement the following linked list method, which is very similar to `LinkedList.sum`.

```
class LinkedList:
    def __len__(self) -> int:
        """Return the number of elements in this linked list.

        >>> linky = LinkedList()
        >>> linky.__len__()
        0
        >>> node3 = _Node(30)
        >>> node2 = _Node(20, node3)
        >>> node1 = _Node(10, node2)
        >>> linky._first = node1
        >>> linky.__len__()
        3
        """
        curr = self._first
        len_so_far = 0

        while curr is not None:
            curr = curr.next
```

```
            len_so_far += 1

        return len_so_far
```

Note: this is yet another Python special method. Unsurprisingly, it gets called when you call the built-in function `len` on a `LinkedList` object. Try it!

2. Here is another linked list method, which allows you to compare the items in two different linked lists.

```
class LinkedList:
    def __eq__(self, other: LinkedList) -> bool:
        """Return whether this list and the other list are equal.

        Two lists are equal when each one has the same number of items, and
        each corresponding pair of items are equal (using == to compare).
        """
        curr_1 = self._first
        curr_2 = other._first

        # Ensure that the lengths are then same using the above len method.
        # If they're not the same length, they cannot be equal.
        # Also the loop condition only works as intended if they're the same len.
        if len(curr_1) != len(curr_2):
            return False

        while curr_1 is not None and curr_2 is not None:
            if curr_1.item != curr_2.item:
                return False

        return True
```

Implement this method by using the linked list traversal pattern, except use *two* loop variables `curr1` and `curr2`, one for each list.

For extra practice, implement this method twice: once using an early return, and once using a compound while loop condition. Which approach do you like better?

CSC111 Lecture 3: Mutating Linked Lists,

Hisbaan Noorani

January 18, 2021

Contents

1 Exercise 1: Index-based insertion

Our goal for this exercise is to extend our `LinkedList` class by implementing one of the standard mutating List ADT methods: inserting into a list by index. Here's the docstring of such a method:

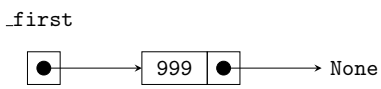
```
class LinkedList:
    def insert(self, i: int, item: Any) -> None:
        """Insert the given item at index i in this list.

        Raise IndexError if i > len(self).
        Note that adding to the end of the list (i == len(self)) is okay.

        Preconditions:
            - i >= 0

        >>> lst = LinkedList([1, 2, 10, 200])
        >>> lst.insert(2, 300)
        >>> lst.to_list()
        [1, 2, 300, 10, 200]
        """
```

Before diving into any code at all, we'll gain some useful intuition by generating some test cases for this method based on two key input properties: the length of the list, and the relationship between `index` and the length of the list.

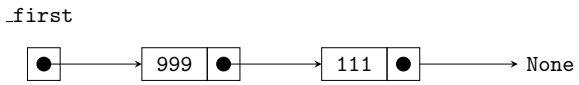
1. In the list below, modify each `self` diagram to show the state of the linked list after `999` is inserted.

   You should draw a new node containing `999`. In each case, you need to determine which arrows to modify to insert the new node into the correct location in the list.

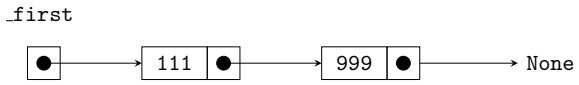
   (a) insert `999` at `0`:



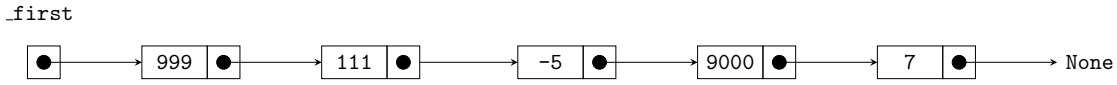
(b) insert `999` at `0`:



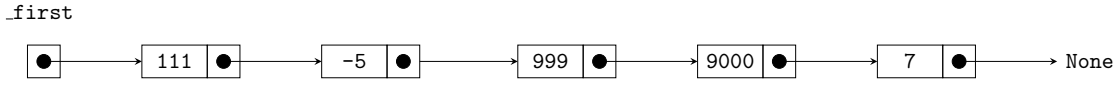
(c) insert `999` at `1`:



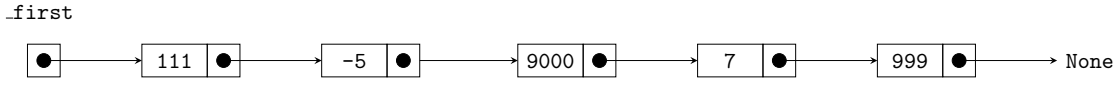
(d) insert `999` at `0`:



(e) insert `999` at `2`:



(f) insert `999` at `4`:



2. Now let's start thinking about some code. Using your diagrams as a guide, answer the following questions:

   (a) For what values of `len(self)` and/or `i` would we need to reassign `self._first` to something new?

   We would only need to reassign `self._first` if we wanted to insert at index `0`.

   (b) What is the relationship between `len(self)` and `i` that makes `insert` behave the same as `LinkedList.append` from this week's prep?

   For insert to behave like append, we want `i = len(self)`. This will mutate the `len(self) - 1` $^{\text{th}}$ node

   (c) In the `len(self) == 4, i == 2` case, which *existing* node was actually mutated? Write down the index of this node in the list. (Hint: it's *not* the node at index 2!)

   The node at index 1 would be mutated (-5). The `next` attribute of the node would have to be modified to match the new inserted node.

3. Finally, using these ideas, implement the `insert` method. Note that you should have two cases: one for when you need to mutate `self._first`, and one where you don't.

   You'll need to make use of the *linked list traversal pattern*, as well as the extra "parallel loop variable" `curr_index` that we studied last week with `LinkedList.__getitem__`.

```
1  class LinkedList:
2      def insert(self, i: int, item: Any) -> None:
3          """Insert the given item at index i in this list.
```

```
4
5           Raise IndexError if i > len(self).
6           Note that adding to the end of the list (i == len(self)) is okay.
7
8           Preconditions:
9               - i >= 0
10
11          >>> lst = LinkedList([1, 2, 10, 200])
12          >>> lst.insert(2, 300)
13          >>> lst.to_list()
14          [1, 2, 300, 10, 200]
15          """
16          new_node = _Node(item)
17
18          if i == 0:
19              new_node.next, self._first = self._first, new_node
20          else:
21              curr = self._first
22              curr_index = 0
23
24              while curr is not None:
25                  if curr_index == i - 1:
26                      new_node.next, curr.next = curr.next, new_node
27                      return
28
29              raise IndexError
```

CSC111 Lecture 4: Mutating Linked Lists, Part 2

Hisbaan Noorani

January 20, 2021

Contents

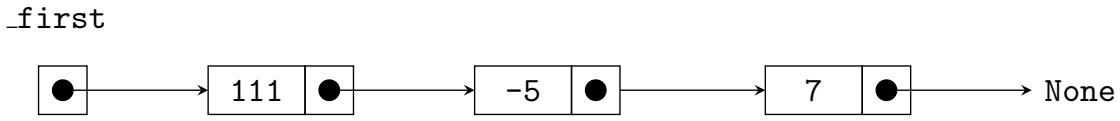
1 Exercise 1: Index-based deletion

Last class, we studied *index-based insertion* into a linked list, and implemented the `LinkedList.insert` method. Now, we're going to study *index-based deletion*, implementing the following method:

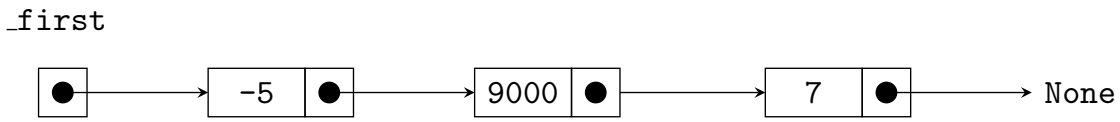
```
class LinkedList:
    def pop(self, i: int) -> Any:
        """Remove and return the item at index i.

        Raise IndexError if i >= len(self).

        Preconditions:
            - i >= 0

        >>> lst = LinkedList([1, 2, 10, 200])
        >>> lst.pop(1)
        2
        >>> lst.to_list()
        [1, 10, 200]
        >>> lst.pop(2)
        200
        >>> lst.pop(0)
        1
        >>> lst.to_list()
        [10]
        """
```

1. As we did last class, before implementing this method we'll look at some diagrams representing example inputs.

(a) Modify the linked list diagram below to show what happens when we call `LinkedList.pop` on it with `i == 2`.



(b) Repeat, but with `i == 0` instead.



2. Implement `LinkedList.pop`. As with `LinkedList.insert`, use the linked list traversal pattern with a parallel `curr_index` variable. You should have two cases based on whether `self._first` needs to be mutated or not.

```
class LinkedList:
    def pop(self, i: int) -> Any:
        """Remove and return the item at index i.

        Raise IndexError if i >= len(self).

        Preconditions:
            - i >= 0

        >>> lst = LinkedList([1, 2, 10, 200])
        >>> lst.pop(1)
        2
        >>> lst.to_list()
        [1, 10, 200]
        >>> lst.pop(2)
        200
        >>> lst.pop(0)
        1
        >>> lst.to_list()
        [10]
        """
        if i == 0:
            if self._first is None:
                # The list is empty. There is no 0th element
                raise IndexError
            else:
                # self._first is a _Node
                curr = self._first
                self._first = self._first.next
        else:
            curr == self._first
            curr_index = 0

            while not (curr is None or curr_index == i - 1)
                curr = curr.next
                curr_index += 1

```

```
            if curr is None:
                raise IndexError
            else:
                if curr.next is None:
                    # There is no node at index i
                    raise IndexError
                else:
                    # curr_index == i - 1; curr is the node at index i - 1
                    item = curr.next.item
                    curr.next = curr.next.next
                    return item
```

2 Exercise 2: Value-based deletion

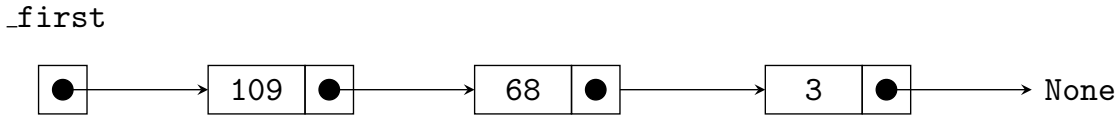
The built-in `list` data type has another type of deletion: `list.remove`, which removes the first occurrence of an item. This is known as *value-based deletion*, since the item that's removed is based on its value rather than its index.

We will now implement an equivalent `LinkedList.remove` method:

```
class LinkedList:
    def remove(self, item: Any) -> None:
        """Remove the first occurence of item from the list.

        Raise ValueError if the item is not found in the list.
        """
```

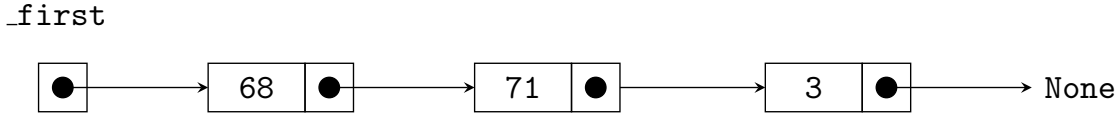
1. Suppose you want to remove `71` from the linked list below.

   (a) Modify the diagram above to show how the linked list would change when `71` is removed.



   (b) Which node needed to be mutated?
   The node at index `1` needs to be mutated (i.e. the node before `71`)

   (c) Here is the same linked list. Suppose we now wanted to remove `109`. Modify the diagram to show how the linked list needs to change.



2. Using these ideas, implement the `LinkedList.remove` method.

```python
class LinkedList:
    def remove(self, item: Any) -> None:
        """Remove the first occurence of item from the list.

        Raise ValueError if the item is not found in the list.
        """
        prev, curr = None, self._first

        while curr is not None:
            if curr.item == item:
                if prev is None:  # curr is self._first
                    self._first = curr.next
                    return
                else:  # curr is not self._first
                    prev.next = curr.next
                    return

            prev, curr = curr, curr.next

        raise ValueError
```

3 Exercise 3: Running time of linked list operations

1. Suppose we have a Python `list` of length 1,000,000, and a `LinkedList` of length 1,000,000.

   (a) If we want to *access* the item at index 500,000 in each list, would it be significantly faster for the `list`, the `LinkedList`, or about the same amount of time for both? Explain.
   Our `LinkedList` traversal pattern that we use for indexing has a running time of $\Theta(i)$ where i is the requested index. This is much slower than the build in `list` indexing method which is $\Theta(1)$. In this case, `list` would be 500,000 times faster than `LinkedList`.

   (b) If we want to *delete* the item at index 0 in each list, would it be significantly faster for the `list`, the `LinkedList`, or about the same amount of time for both? Explain.
   It would be about the same amount of time. Removing the first index does not require the use of our `LinkedList` traversal pattern. This means that the operation would have a running time of $\Theta(1)$. For the build in `list`, the running time is $\Theta(n)$. In this case, `LinkedList` would be 1,000,000 times faster than `list`.

   (c) If we want to *insert* an item at index 500,000 in each list, would it be significantly faster for the `list`, the `LinkedList`, or about the same amount of time for both? Explain.
   In this case, running time would be very similar. The `LinkedList` traversal pattern would need to be used meaning the running time would be $\Theta(\frac{n}{2})$. The `list.pop()` method would also be $\Theta(\frac{n}{2})$.

2. Consider our current implementation of `LinkedList.__init__`, which uses the `append` method:

```python
class LinkedList:
    def __init__(self, items: Iterable) -> None:
        self._first = None
        for item in items:
            self.append(item)
```

   (a) What is the running time of `LinkedList.append`, in terms of `n`, the length of the linked list?
   $RT_{\text{append}} \in \Theta(n)$. This is because we need to use the linked list traversal pattern to get to the end of the list then set the last `_Node.next` to be the new item that we are appending.

(b) Analyze the running time of this implementation of `LinkedList.__init__`, in terms of n, the length if `items`.

Remember that when taking into account the running time of a helper function, you can drop the "Theta", e.g. count a $\Theta(n)$ function as n steps.

The for loop iterates n times.

The loop body takes m steps where m is the current length of self which is changing over time. It increases by 1 each iteration.

So at iteration i, loop body takes i steps.

So then finally, $RT_{\_\_\text{init}\_\_} = \sum_{i=0}^{n-1} i = \frac{n \cdot (n-1)}{2} \in \Theta(n^2)$

CSC111 Lecture 5: Induction, Recrusion, and Nested Lists

Hisbaan Noorani

January 25, 20201

Contents

Note: today's lecture will cover a lot of new material and we likely won't have time for exercises. Instead, we have included two additional exercises for you to complete as homework after class.

1 Additional exercises

1. We saw in class that the greatest common divisor function can be defined recursively over the natural numbers as follows:

$$\gcd(a, b) = \begin{cases} a, & \text{if } b = 0 \\ \gcd(b, a \ \% \ b), & \text{if } b > 0 \end{cases}$$

   (a) Modify this definition so that it is valid for all integers (including negative numbers).
   What mathematical properties of gcd do you need to hold in order for your definition to be valid?

   (b) Implement your new recursive definition in Python. It should no longer need the preconditions $a \geq 0$ and $b \geq 0$

2. In today's lecture, we learned about a formal recursive definition of a *nested list of integers*, and used this to define a function that calculates the sum of a nested list. In this exercise, you'll practice this technique with a new function: computing the number of integers in a nested list.

   Let num_ints be a function that takes a nested list and returns the number of integers in that list.

   (a) What is num_ints(7)? num_ints(111)? num_ints(x), for an arbitrary $x \in \mathbb{Z}$?

   (b) Suppose we have three nested lists a_0, a_1, a_2, and we know the following:
   - num_ints(a_0) = 10
   - num_ints(a_1) = 1
   - num_ints(a_2) = 3

   What is num_ints($[a_0, a_1, a_2]$)?

   (c) Let $k \in \mathbb{N}$, and let $x = [a_0, a_1, \ldots, a_{k-1}]$ be a nested list (where each of the a_i is also a nested list).
   Write a formula relating num_ints(x) to the values num_ints(a_0), num_ints(a_1), ..., num_ints(a_{k-1}).

   (d) Write a recursive definition for the num_ints function:

$$\text{num\_ints}(x) = \begin{cases} \ldots \\ \ldots \end{cases}$$

   How does this recursive definition compare with the one for *sum_nested* from lecture?

   (e) Finally, implement a recursive Python function `num_ints` that takes a nested list of integers and returns the number of integers in that list.
   How does this definition compare with the one for `sum_nested` from lecture?

CSC111 Lecture 6: More with Nested Lists

Hisbaan Noorani

January 27, 2021

Contents

1 Exercise 1: Inductive Reasoning for Recursive Functions

This exercise is designed to give you practice *inductive reasoning* (aka *partial tracing*) when working with recursive functions. **Do this exercise on paper, not in PyCharm.** You're developing a new skill here: reasoning recursive code by using your brain, not relying on a computer!

1. Here is a partial implementation of a nested list function that returns a brand-new list. We have deliberately omitted the base case, since it is *not relevant* for reasoning about the recursive step.

```
def flatten(nested_list: Union[int, list]) -> list[int]:
    """Return a (non-nested) list of the integers in nested_list.

    The integers are returned in the left-to-right order they appear in
    nested_list.
    """
    if isinstance(nested_list, int):
        # Base case omitted

    else:
        result_so_far = []
        for sublist in nested_list:
            result_so_far.extend(flatten(sublist))
        return result_so_far
```

Our goal is to determine whether the recursive case is correct *without* fully tracing (or running) this code. Consider the function call `flatten([[0, -1], -2, [[-3, [-5], -7]]])`.

(a) What should `flatten([[0, -1], -2, [[-3, [-5], -7]]])` return, according to its docstring?

`[0, -1, -2, -3, -5, -7]`

(b) We'll use the loop accumulation table below to partially trace the call `flatten([[0, -1], -2, [[-3, [-5], -7]]])`. Complete the **first two columns** of this table, assuming that `flatten` works properly on each recursive call. Remember that filling out these two columns can be done *just* using the argument value and `flatten`'s docstring; you don't need to worry about the code at all!

Note: the nested list `[[0, -1], -2, [[-3, [-5], -7]]]` has just *three* sublists.

sublist	flatten(sublist)	result_so_far
N/A	N/A	[]
[0, -1]	[0, -1]	[0, -1]
-2	[-2]	[0, -1, -2]
list([-3, [-5], -7])	[-3, -5, -7]	[0, -1, -2, -3, -5, -7]

(c) Use the third column of the table to complete the partial trace of the recursive step. Remember that every time you reach a recursive call, don't trace into it—use the value you calculated in the second column!

Done in table

(d) Compare the final value of `result_so_far` with the expected return value of `flatten`. Does this match?

Yes, they match.

(e) Finally, write down an implementation of the `base case` of `flatten` directly on the code above.

`return[nested_list]`

2 Exercise 2: Designing recursive functions (1)

In this exercise, you'll get some practice implenting a new recursive function that operates on a nested list. We've broken down this exercise into various steps, following the *Recursive Function Design Recipe* we covered in lecture.

For your reference, here is the *nested list function code template*:

```
def f(nested_list: Union[int, list]) -> ...:
    if isinstance(nested_list, int):
        ...
    else:
        # With a loop
        ...
        for sublist in nested_list:
            ... f(sublist) ...

        # Or with a comprehension
        ... [f(sublist) for sublist in nested_list] ...
```

Here is the function we'll implement; read through its docstring carefully first.

```
def nested_list_contains(nested_list: Union[int, list], item: int) -> bool:
    """Return whether the given item appears in nested_list.

    If nested_list is an integer, return whether it is equal to item.
    """
```

1. *Base case example and implementation.*

   (a) Write a doctest example for this function where the argument `nested_list` is an `int`.

   ```
   >>> nested_list_contains(3, 3)
   True
   >>> nested_list_contains(2, 3)
   False
   ```

   (b) Implement the base case of this function.

   ```
   def nested_list_contains(nested_list: Union[int, list], item: int) -> bool:
       if isinstance(nested_list, int):
           return nested_list == item
       else:
           ...
   ```

2. *Recursive step examples.*

   (a) Consider the following doctest example:

   ```
   >>> nested_list_contains([ [1, [30], 40], [], 77], 50)
   False
   ```

   Complete the following table, showing each of the sublists of `[[1, [30], 40], [], 77]` in this example, as well as what each recursive call would return. Note that we've set the `item` argument to `50` in the recursive calls, since we need to search for `50` in each sublist.

`sublist`	`nested_list_contains(sublist, 50)`
`[1, [30], 40]`	`False`
`[]`	`False`
`77`	`False`

   (b) Repeat, but with the following doctest example instead:

   ```
   >>> nested_list_contains([[1, [30], 40], [], 77], 40)
   True
   ```

`sublist`	`nested_list_contains(sublist, 40)`
`[1, [30], 40]`	`True`
`[]`	`False` (Doesn't run)
`77`	`False` (Doesn't run)

3. *Generalize examples and implement the recursive step.*

   Implement the recursive step for this function. Use your answers to the previous question to determine how to *aggregate* the results of the recursive call. You can use a loop or a built-in aggregation function and comprehension.

```python
def nested_list_contains(nested_list: Union[int, list], item: int) -> bool:
    if isinstance(nested_list, int):
        ...
    else:
        # Version 1, comprehension
        # Note, this only works if you have a suitable built in agregation function
        # Dropping the square brackets, we allow early return therefore > efficiency
        return any(nested_list_contains(sublist, item) for sublist in nested_list)

        # Version 2, for loop
        for sublist in nested_list:
            if nested_list_contains(sublist, item):
                return True

        retturn False
```

3 Exercise 3: Designing recursive functions (2)

Now for something a bit more complex. We previously defined the *depth* of a nested list as the maximum number of times a list is nested within another one in the nested list. Analogously, we can define the **depth of an integer in a nested list** to be the number of nested lists enclosing the object.

For example, in the nested list `[10, [[20]], [[30], 40]]`:

- The depth of `10` is 1
- The depths of `20` and `30` are 3
- The depth of `40` is 2.

For a nested list that is a single integer x, the depth of x is 0.

Your goal is to implement the following function:

```python
def items_at_depth(nested_list: Union[int, list], d: int) -> list[int]:
    """Return the list of all integers in nested_list that have depth d.

    Preconditions:
        - d >= 0
    """
```

1. *Base case example and implementation.*

(a) Write *two* doctests where `nested_list` is an `int`, covering different possibilities for `d` that lead to different return values. Make sure you understand how the return value differs in your two doctests before moving on.

```
>>> items_at_depth(10, 0)
[10]
>>> items_at_depth(10, 1)
[]
```

(b) Implement the base case. It's okay to have a nested if statement at this point.

```
def items_at_depth(nested_list: Union[int, list], d: int) -> list[int]:
    if isinstance(nested_list, int):
        if d == 0:
            return [nested_list]
        else:
            return []
    else:
        ...
```

2. *Recursive step examples.* Now let's consider the recursive step. Let `nested_list = [10, [[20]], [[30], 40]]`.

   (a) Write *two* doctests with this nested list, one where `d = 0` and one where `d = 3`. Make sure you understand these two cases before moving on.

   (b) Now let's study the `d = 3` case in more detail. Here is the start of a recursive call table for this example, by passing `d = 3` into each recursive call. Complete the table.

sublist	items_at_depth(sublist, 3)
10	[]
[[20]]	[]
[[30], 40]	[]

   Problem! It looks like the results of the recursive calls don't actually help calculate the expected return value of `items_at_depth(nested_list, 3)`.

   We need to change the depth argument we're passing in.

   (c) If we want the items at depth `3` in our original nested list, what depth value should we pass into the recursive calls? Complete the table below.

sublist	items_at_depth(sublist, 2)
10	[]
[[20]]	[20]
[[30], 40]	[30]

3. *Generalize examples and implement the recursive step.* Finally, implement `items_at_depth`. Once again, it's okay to have a nested if statement.

```python
def items_at_depth(nested_list: Union[int, list], d: int) -> list[int]:
    if isinstance(nested_list, int):
        ...
    else:
        if d == 0:
            return []
        else:
            result_so_far = []
            for sublist in nested_list:
                result_so_far.extend(items_at_depth(sublist, d - 1))

            return results_so_far
```

4. If you followed our nested list template in this exercise, you would have ended up with nested if statements. Try rewriting your code so that there are no nested if statements, instead using `elif` branches for the different cases.

```python
def items_at_depth(nested_list: Union[int, list], d: int) -> list[int]:
    if d == 0 and isinstance(nested_list, int):
        return [nested_list]
    elif d == 0:
        return []
    elif isinstance(nested_list, int):
        return []
    else:
        results_so_far = []
        for sublist in nested_list:
            result_so_far.extend(items_at_depth(sublist, d - 1))

        return results_so_far
```

4 Additional exercises

4.1 Debugging with partial tracing

Now consider the following function and partial implementation:

```python
def uniques(nested_list: Union[int, list]) -> list[int]:
    """Return a (non-nested) list of the integers in nested_list, with no duplicates.
    """
    if isinstance(obj, int):
        # Base case omitted
    else:
        result_so_far = []
        for sublist in obj:
            result_so_far.extend(uniques(sublist))
        return result_so_far
```

It turns out that there is a problem with the recursive step in this function, and it has the insidious feature of being *sometimes correct, and sometimes incorrect.*

1. To make sure you understand this, find *two* example inputs to `uniques`: one in which partial tracing would lead to us thinking there's no error, and one in which partial tracing would lead us to find an error. Each input should have **three** sublists.

   Input that does NOT reveal an error:

   Expected output:

   Loop accuulation table (fill it in column by column, and verify that the final accumulator value matches the expected output)

`sublist`	`uniques(sublist)`	`result_so_far`
N/A	N/A	`[]`

   Input that DOES reveal an error:

   Expected output:

   Loop accuulation table (fill it in column by column, and verify that the final accumulator value doesn't match the expected output)

`sublist`	`uniques(sublist)`	`result_so_far`
N/A	N/A	`[]`

2. What you've provided above is a *counterexample* that shows that this recursive step is incorrect. This is a good start, but we'd like to go deeper. Describe in English *why* the recursive step is incorrect, i.e., what the problem with the code is.

   (Comment: we aren't asking you to fix the implementation of this method here, but you can do so as a good homework exercise.)

4.2 Mutating a nested list

In this exercise, we're going to look at a more complex form of recursion on nested lists involving *mutation*.

The running example we'll use for this worksheet is the following function:

```
1  def add_one(nested_list: Union[int, list]) -> None:
2      """Add one to every number stored in nested_list.
3
4      Do nothing if nested_list is an int.
```

```
    If nested_list is a list, *mutate* it to change the numbers stored.

    >>> lst0 = 1
    >>> add_one(lst0)
    >>> lst0
    1
    >>> lst1 = []
    >>> add_one(lst1)
    >>> lst1
    []
    >>> lst2 = [1, [2, 3], [[[5]]]]
    >>> add_one(lst2)
    >>> lst2
    [2, [3, 4], [[[6]]]]
    """
    # if isinstance(nested_list, int):
    #     ...
    # else:
    #     for sublist in nested_list:
    #         ... add_one(sublist) ...
```

1. To start, think about the *base case* for this function. Implement it in the space below.

   Hint: read the docstring carefully—it tells you exactly what to do.

```
if isinstance(nested_list, int):
```

2. Now for the recursive step. The limitation of the standard `for sublist in nested_list` loop is that while it can mutate individual sublists of `nested_list` that are lists, it can't modify any *integer* element of `nested_list` directly.

   To make sure you understand this, suppose `nested_list = [1, 2, 3]`, and we run the following code on it:

```
else:
    # nested_list = [1, 2, 3]
    for sublist in nested_list:
        sublist = sublist + 1
```

   What would be the result of executing this code?

   Hint: draw a memory model diagram to trace the function.

3. In general, if we want to mutate elements of a list, we loop over the *indexes* of the list rather than its elements directly:

```
for i in range(0, len(nested_list)):
```

Using this idea, implement `add_one` in the space below.

Hint: you'll need different cases in your loop for when `obj[i]` is an integer or a list.

```
def add_one(nested_list: Union[int, list]) -> None:
    """Add one to every number stored in nested_list.

    Do nothing if nested_list is an int.

    If nested_list is a list, *mutate* it to change the numbers stored.
    """
    if isinstance(nested_list, int):
        # Write your answer to Question 1 here.
    else:
        for i in range(0, len(nested_list)):
```

CSC111 Lecture 7: Trees and Recursion

Hisbaan Noorani

February 1, 2021

Contents

Note: if you wish, you may write the code in this exercise in the same file as your prep from this week.

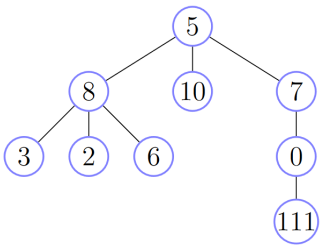
For your reference, here is our definition for the `Tree` class:

```python
from __future__ import annotations
from typing import Any, Optional


class Tree:
    """A recursive tree data structure.

    Note the relationship between this class and RecursiveList; the only major
    difference is that _rest has been replaced by _subtrees to handle multiple
    recursive sub-parts.

    Representation Invariants:
        - self._root is not None or self._subtrees == []
    """
    # Private Instance Attributes:
    #   - _root:
    #       The item stored at this tree's root, or None if the tree is empty.
    #   - _subtrees:
    #       The list of subtrees of this tree. This attribute is empty when
    #       self._root is None (representing an empty tree). However, this attribute
    #       may be empty when self._root is not None, which represents a tree consisting
    #       of just one item.
    _root: Optional[Any]
    _subtrees: list[Tree]

    def __init__(self, root: Optional[Any], subtrees: list[Tree]) -> None:
        """Initialize a new Tree with the given root value and subtrees.
```

```
        If root is None, the tree is empty.

        Preconditions:
            - root is not none or subtrees == []
        """
        self._root = root
        self._subtrees = subtrees

    def is_empty(self) -> bool:
        """Return whether this tree is empty.

        >>> t1 = Tree(None, [])
        >>> t1.is_empty()
        True
        >>> t2 = Tree(3, [])
        >>> t2.is_empty()
        False
        """
        return self._root is None
```

1 Exercise 1: Designing Recursive Tree Functions

Consider the following definition.[1]
Let T be a tree, and x be a value in the tree. The **depth** of x in T is the distance (counting links) between the item and the root of the tree, inclusive.



For example, in the above tree:

- The root value 5 has depth **zero**.
- The children of the root, 8, 7, and 10, have depth **one**
- The value 111 has depth **three**.

Your task is to implement the following `Tree` method.

```
class Tree:
    def first_at_depth(self, d: int) -> Optional[Any]:
```

```
        """Return the leftmost value at depth d in this tree.

        Return None if there are NO items at depth d in this tree.

        Preconditions:
            - d >= 0  # depth 0 is the root of the tree
        """
```

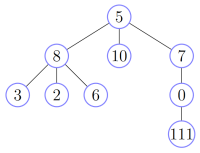
1. (*base cases*) First, let's consider some base cases. Complete each of the following doctests. If `None` would be returned, write down `None`, even though nothing would actually be printed in the Python console (this is just to help you remember when reviewing!).

```
>>> empty = Tree(None, [])
>>> empty.first_at_depth(0)
None

>>> empty.first_at_depth(10)
None

>>> single = Tree(111, [])
>>> single.first_at_depth(0)
111

>>> single.first_at_depth(3)
None
```

2. (*base cases*) Implement the base cases below (don't worry if the size-one case might end up being redundant—you can double-check this after implementing the recursive step).

```
class Tree:
    def first_at_depth(self, d: int) -> Optional[Any]:
        """Return the leftmost value at depth d in this tree.

        Return None if there are NO items at depth d in this tree.

        Preconditions:
            - d >= 0
        """
        if self.is_empty():
            return None
        elif not self._sublists:
            if d == 0:
                return self._root
            return None
        else:
            ...
```

3. (*recursive step*) Now suppose we have a variable `tree` that refers to the following tree:



(a) What should `tree.first_at_depth(0)` return?

`5`

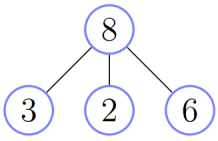
(b) What should `tree.first_at_depth(1)` return?

`8`

(c) Now let's investigate `tree.first_at_depth(1)` recursively.

Fill in the recursive call table below, choosing the right "depth" argument so that the return values of the recursive calls are actually useful for computing `tree.first_at_depth(1)`. (Write `None` if that's what is returned.)

Subtree

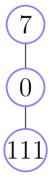


`subtree.first_at_depth(1) = 3`

Subtree

`subtree.first_at_depth(1) = None`

Subtree



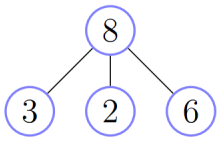
`subtree.first_at_depth(1) = None`

(d) What should `tree.first_at_depth(3)` return?

`111`

(e) Once again, fill in the recursive call table below, this time to compute `tree.first_at_depth(3)`.

Subtree

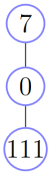


`tree.first_at_depth(3) = None`

Subtree

`tree.first_at_depth(3) = None`

Subtree



`tree.first_at_depth(3) = 111`

(f) Finally, implement the recursive step below.

```
class Tree:
    def first_at_depth(self, d: int) -> Optional[Any]:
        """Return the leftmost value at depth d in this tree.

        Return None if there are NO items at depth d in this tree.

        Preconditions:
            - d >= 0
        """
        if self.is_empty():
            return None
        # elif not self._sublists:
        #     if d == 0:
        #         return self._root
        #     return None
        else:
            if d == 0:
                return self._root
            else:
                for subtree in self._subtrees:
                    result = subtree.first_at_Depth(d - 1)
                    if result is not None:
                        return result
                return none
```

(g) Is the size-one base case you implemented above redundant? If so, modify your code to remove it. If not, leave a comment explaining why not.

Yes, it is redundant. The `if d == 0` is covered in the next case on line `17`. The `return None` is covered by the other `return None` on line 24 as the for loop will not trigger since there are not `subtree` elements if `self._subtrees == []`.

2 Additional exercises

1. Implement a method `Tree.items_at_depth`, which returns *all* items at a given depth in the tree.

2. Consider the following definition. The **branching factor** of an item in a tree is its number of children (or equivalently, its number of subtrees). In Artificial Intelligence, one of the most important properties of a tree is the average branching factor of its *internal values* (i.e., its non-leaf values).

   Implement a method `Tree.branching_factor` that computes the average branching factor of the internal values in a tree, returning `0.0` if the tree has no internal values.

---

1. This is analogous to a definition we gave for nested lists last week!

CSC111 Lecture 8: Tree Mutation and Efficiency

Hisbaan Noorani

February 3, 2021

Contents

1 Exercise 1: Tree Deletion

We've seen that when deleting an item from a tree, the bulk of the work comes when you've already found the item, that is, you are "at" a subtree where the item is in the root, and you need to delete it. This is the code we developed in lecture:

```
class Tree:
    def remove(self, item: Any) -> bool:
        """Delete *one* occurrence of the given item from this tree.

        Do nothing if the item is not in this tree.
        Return whether the given item was deleted.
        """
        if self.is_empty():
            return False
        elif self._root == item:
            self._delete_root()
            return True
        else:
            for subtree in self._subtrees:
                if subtree.remove(item):
                    # Call an update function to remove empty subtrees
                    # self._remove_empty_subtrees()

                    # Check whether subtree is empty
                    if subtree.is_empty():
                        list.remove(self._subtrees, subtree)
                    return True
            return False
```

Our goal is to complete this function by implementing the helper `Tree._delete_root`:

```
class Tree:
    def _delete_root(self) -> None:
        """Remove the root item of this tree.

        Preconditions:
            - not self.is_empty()
        """
```

1. We can't always set the `self._root` attribute to `None`. When can we, and when must we do something else?

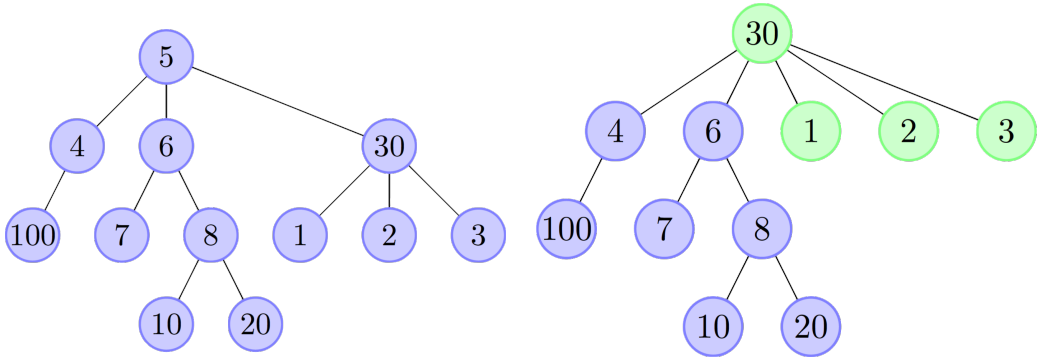
   Setting `self._root` to `None` would violate a representation invariant of the `Tree` class if the node has subtrees. Setting it to `None` would also delete the whole branch beneath it. So instead of setting it to none, we have to change it in some other way (see below).

Next, we'll look at two strategies for replacing `self._root` with a new value from somewhere else in the tree.

1.1 Strategy 1: "Promoting" a subtree

Idea: to delete the root, take the rightmost subtree t_1, and make the root of t_1 the new root of the full tree, and make the subtrees of t_1 become subtrees of the full tree.[1]

Before: After:



Implement `Tree._delete_root` using this approach.

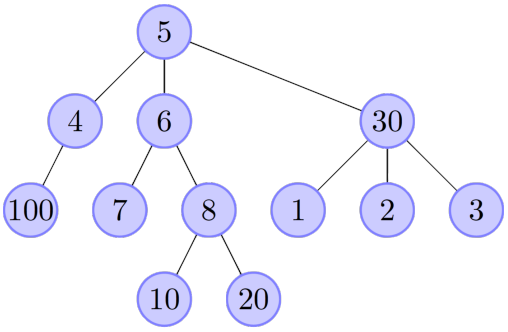
```
class Tree:
    def _delete_root(self) -> None:
        """Remove the root item of this tree.

        Preconditions:
            - not self.is_empty()
        """
        if self._subtrees == []:
            self._root == None
```

```
        else:
            last_subtree = self._subtrees.pop()
            self._root = last_subtree._root
            self._subtrees.extend(last_subtree._subtrees)
```

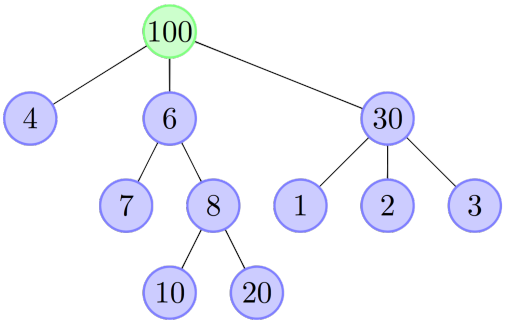
2 Strategy 2: Replace the root with a leaf

Idea: to delete the root, find the leftmost *leaf* of the tree, and move that leaf so that it becomes the new root value. No other values in the tree should move.[2]

Before:



After:



Implement `Tree._delete_root` using this approach. We recommend using an additional helper method to recursive remove and return the leftmost leaf in a tree.

```
class Tree:
    def _delete_root(self) -> None:
        """Remove the root item of this tree.

        Preconditions:
            - not self.is_empty()
        """
        self._root = self._extract_leaf()

        # Or we could use a while loop
        # withotu the helper method
        prev, curr = None, self._root
        while not self._subtress:
            prev, curr = curr, curr._subtrees[0]

        self._root = curr
        prev.



    def _extract_leaf(self) -> Any:
```

```
        """Remove and return the leftmost leaf in this tree.

        Precondiditons
            - not self.is_empty()
        """
        if self._subtrees = []:
            root = self._root
            self._root = None
            return root

        return self._subtrees[0]._extract_leaf()
```

Instead of leaving the leaf as a subtree with `None`, we want to make all of our methods forbit this. We added this as a representation invariant:

```
all(not subtree.is_empty() for subtree in self._subtrees)
```

3 Additional exercises

1. Write a new method `Tree.remove_all` that deletes *every* occurrence of the given item from a tree. As with linked lists, you'll need to be careful here about the order in which you check the items and mutate the tree so that you don't accidentally skip some occurrences of the item.

2. Consider the following `Tree` method:

```
class Tree:
    def leftmost(self) -> Optional[Any]:
        if self.is_empty():
            return None
        elif self._subtrees == []:
            return self._first
        else:
            return self._subtrees[0].leftmost()
```

   Suppose the variable `tree` refers to the same example tree from lecture when we analysed the running time of `Tree.__len__`.

   (a) Draw the recursive call diagram when we call `tree.leftmost()`. The diagram should look different than the one for `Tree.__len__`!

   (b) What is the exact non-recursive running time of the `Tree.leftmost` method?

   (c) Using your answers to parts (a) and (b), compute the exact running time of `tree.leftmost()` (for this specific `tree` variable).

   (d) Let $n \in \mathbb{N}$. Describe a tree of size n such that `Tree.leftmost` would take $\Theta(n)$ time for that tree.[3]

(e) Let $n \in \mathbb{N}$. Describe a tree of size n such that `Tree.leftmost` would take $\Theta(1)$ time for that tree.

---

1. We could have also chosen to "promote" the leftmost subtree, or some other subtree.
2. We could have also chosen to use any other leaf to replace the root.
3. To use the terminology from CSC110, you are describing an *input family* with this running time.

CSC111 Lecture 9: Introduction to Binary Search Trees

Hisbaan Noorani

February 8, 2021

Contents

1 Exercise 1: Implementing `BinarySearchTree.insert`

In this exercise, we will implement a method to insert items into a binary search tree. Here is the docstring for the `BinarySearchTree.insert` method:

```
class BinarySearchTree:
    def insert(self, item: Any) -> None:
        """Insert <item> into this tree.

        Do not change positions of any other values.

        >>> bst = BinarySearchTree(10)
        >>> bst.insert(3)
        >>> bst.insert(20)
        >>> bst._root
        10
        >>> bst._left._root
        3
        >>> bst._right._root
        20
        """
```

When we implement the `insert` method, we must ensure that we maintain the representation invariants of the `BinarySearchTree` class:

```
class BinarySearchTree:
    """Binary Search Tree class.

    Representation Invariants:
      - (self._root is None) == (self._left is None)
      - (self._root is None) == (self._right is None)
      - (BST Property) if self._root is not None, then
```

```
        all items in self._left are <= self._root, and
        all items in self._right are >= self._root
    """
```

1. First, we will implement the base case for the method.

   (a) Suppose you have an empty binary search tree. Based on the representation invariants, which of `_root`, `_left`, and `_right` are `None`?

   For an empty tree, all three are `None`.

   (b) Suppose you have a binary search tree with only a single value. Based on the representation invariants, what are the values of `_root`, `_left`, and `_right` are `None`?

   None of the three are `None`. `_root` will be the value, `_left` and `_right` will be `BinarySearchTree(None)`.

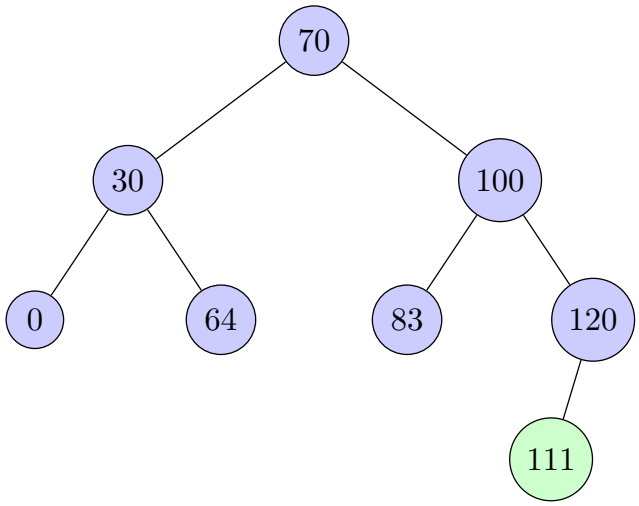
   (c) Complete the base case for `BinarySearchTree.insert`.

```python
class BinarySearchTree:
    def insert(self, item: Any) -> None:
        if self.is_empty():
            self._root = item
            self._left = BinarySearchTree(None)
            self._right = BinarySearchTree(None)
        else:
            ...
```

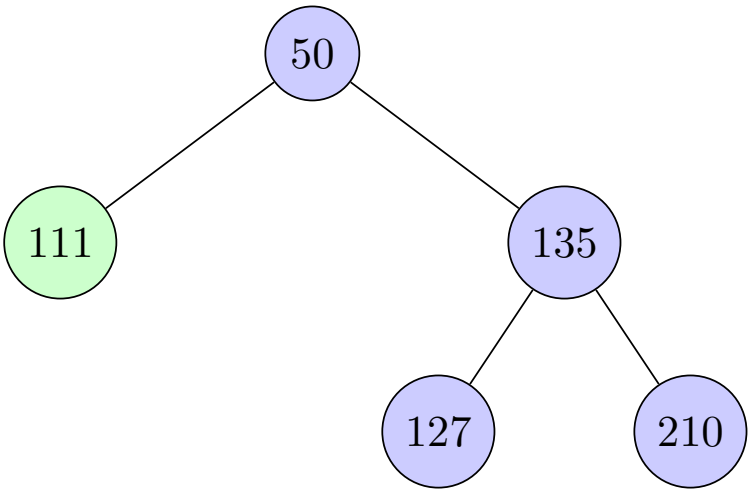
2. Before we write any code for the recursive step, we will look at some examples. In our implementation of `BinarySearchTree.insert`, we will insert a new item into the tree by creating a new *leaf* with the desired value.

   For each of the following binary search trees, insert the value `111` by adding a new leaf, ensuring that the resulting tree still satisfies the BST Property. There may be more than one correct answer!

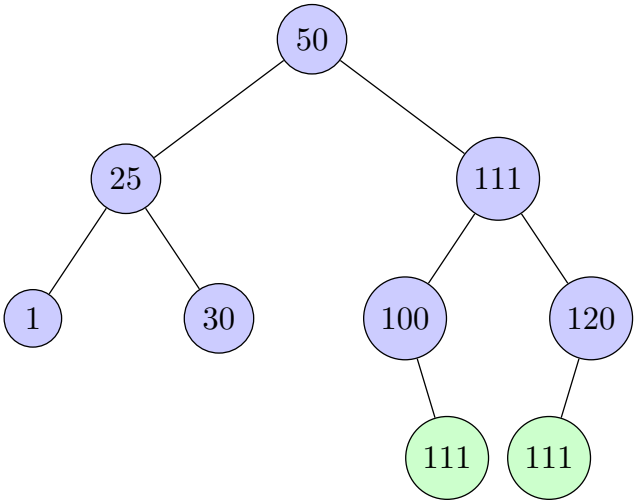
   (a) Insert to the left of `120`



(b) Insert on hte left side of `120`.



(c) Insert on the right of `100` or on the left of `120`.



3. Suppose we want to insert `111` into a non-empty binary search tree by creating a new leaf with the value `111`. We must traverse through one of the subtrees of the binary search tree until we reach an empty binary search tree (the `_left` or `_right` attribute of one of the existing leaves).

   (a) If the `_root` of the binary search tree is less than `111`, in which subtree should the new leaf be created?

   (b) If the `_root` of the binary search tree is greater than `111`, in which subtree should the new leaf be created?

   (c) If the `_root` of the binary search tree is equal to `111`, in which subtree should the new leaf be created?

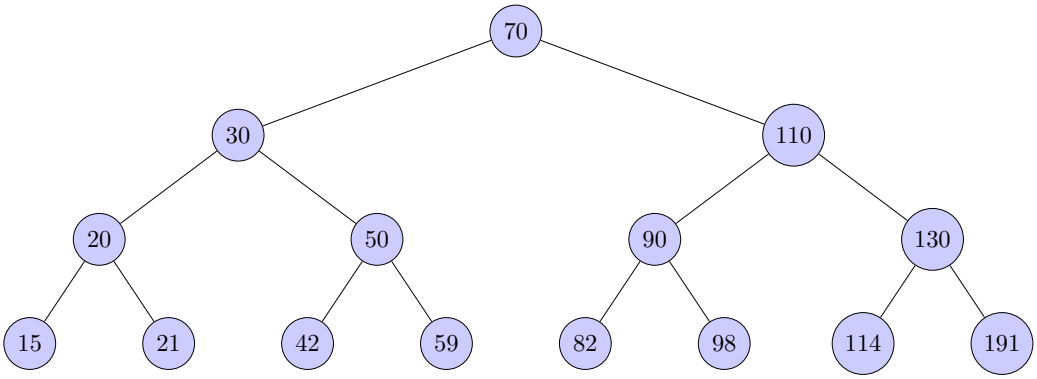
(d) Complete the recursive step for `insert` (you can use a nested if statement, or you can use an `elif` like you saw in `BinarySearchTree.__contains__`).

```
class BinarySearchTree:
    def insert(self, item: Any) -> None:
        """...."""
        if self.is_empty():
            self._root = item
            self._left = BinarySearchTree(None)
            self._right = BinarySearchTree(None)
        elif item == self._root:
            self._left.insert(item)
        elif item < self._root:
            self._left.insert(item)
        elif item > self._root:
            self._right.insert(item)
```

2 Exercise 2: Deletion from a binary search tree (preview)

Before we write any code for BST deletion, we will look at an example to gain some intuition about how we could delete values from a binary search tree.

1. Suppose you have to delete a value from the BST below. What would be an extremely easy value to delete, without changing the rest of the tree?



   One of the leafs!

2. Suppose instead you have to delete the tree's root, 70. Ugh. One strategy is to keep most of the tree structure as is, but to find another value in the tree that can be used to replace the 70.

   (a) Could 110 replace the 70?
   No. $90 < 110$

   (b) Could 20 replace the 70?
   No. $30 > 20$

(c) Could 98 replace the 70?
No. $90 < 98$

(d) Exactly which values can replace the 70?
$59 \leq x \leq 82$

CSC111 Lecture 10: Binary Search Tree Deletion and Running-Time Analysis

Hisbaan Noorani

February 10, 2021

Contents

1 Exercise 1: Implementing `BinarySearchTree._delete_root`

We saw in lecture that we can implement the `BinarySearchTree.remove` method as follows, using a helper to do the "hard" part:

```
class BinarySearchTree:
    def remove(self, item: Any) -> None:
        """Remove *one* occurrence of <item> from this BST.

        Do nothing if <item> is not in the BST.
        """
        if self.is_empty():
            pass
        elif item == self._root:
            self._delete_root()
        elif item < self._root:
            self._left.remove(item)
        else:
            self._right.remove(item)


    def _delete_root(self) -> None:
        """Remove the root of this BST.

        Preconditions:
```

```
            - not self.is_empty()
        """
```

In this exercise, we'll lead you through the cases to develop an implementation of the `BinarySearchTree._delete` in a similar fashion to what we did in lecture for `Tree._delete_root` last week.

1.1 Case 1: `self` is a leaf

Suppose `self` is a leaf. Implement this case below, by filling both:

1. The if condition to check whether `self` is a leaf.
2. The if branch to update `self`'s instance attributes to delete the root.

```
class BinarySearchTree:
    def _delete_root(self) -> None:
        # Case 1: this BST is a leaf
        if self._left.is_empty() and self._right).is_empty():
            self._root = None
            self._left = None
            self._right = None
```

1.2 Case 2: exactly one of `self`'s subtrees are empty

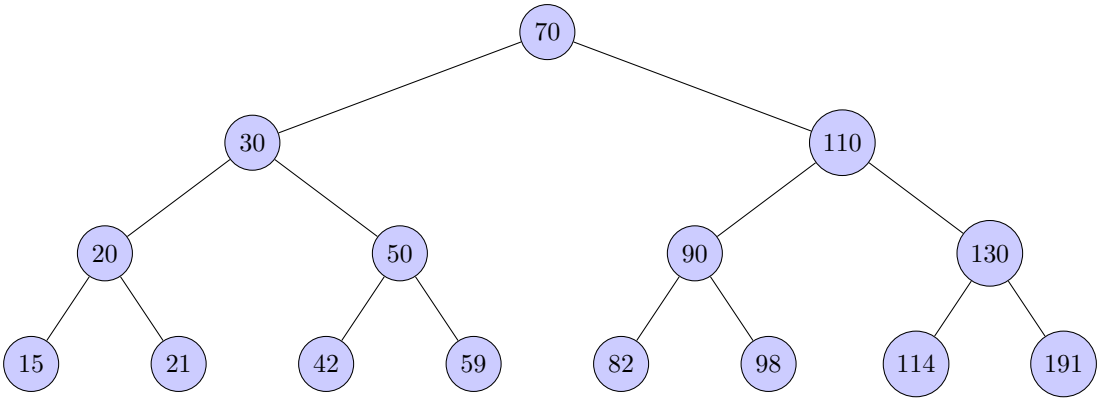
Suppose we want to delete the root of a BST where one of the subtrees is empty. The simplest approach is to "promote the non-empty subtree", similar to the technique from last week. Review this idea, and then fill in the conditions and implementations of each `elif`.

Hint: this is actually easier than what we did trees last week, since you can just reassign all three instance attributes—no need to `list.extend` the list of subtrees.

```
class BinarySearchTree:
    def _delete_root(self) -> None:
        # Case 1: this BST is a leaf
        if self._left.is_empty() and self._right).is_empty():
            self._root = None
            self._left = None
            self._right = None
        # Case 2a: empty left subtree, non-empty right subtree
        elif self._left.is_empty():
            self._root, self._left, self._right =\
                self._right._root, self._right._left, self._right._right
        # Case 2b: non-empty left subtree, empty right subtree
        elif self._right.is_empty():
            self._root, self._left, self._right =\
                self._left._root, self._left._left, self._left._right
```

1.3 Case 3: both subtrees are non-empty

Consdier the following BST, whose left and right subtrees are *both* non-empty.



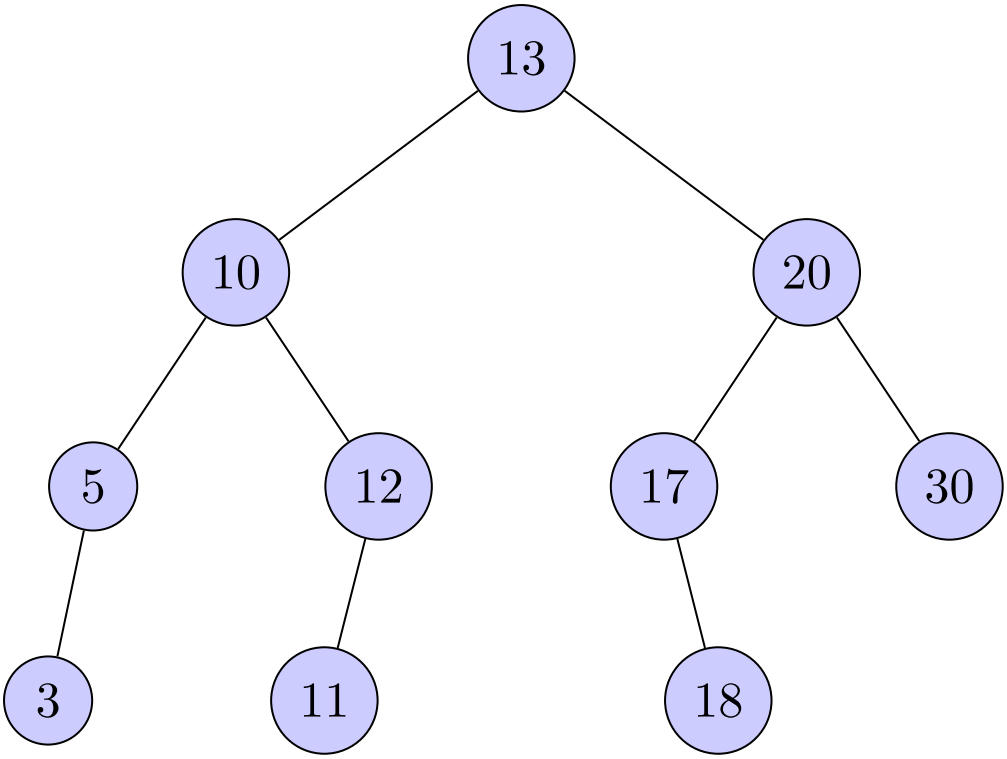
1. Suppose we want to delete the root 70 by replacing it with a value from one of its subtrees. Which possible values we could use to replace the root and maintain the BST property? (*This should be review from yesterday.*

2. Since there are two possible values, you have a choice about which one you want to pick. Complete the final case of `BinarySearchTree._delete_root`, using a helper method to extract your desired value.

```
class BinarySearchTree:
    def _delete_root(self) -> None:
        # Case 1: this BST is a leaf
        if self._left.is_empty() and self._right).is_empty():
            self._root = None
            self._left = None
            self._right = None
        # Case 2a: empty left subtree, non-empty right subtree
        elif self._left.is_empty():
            self._root, self._left, self._right =\
                self._right._root, self._right._left, self._right._right
        # Case 2b: non-empty left subtree, empty right subtree
        elif self._right.is_empty():
            self._root, self._left, self._right =\
                self._left._root, self._left._left, self._left._right
        # Case 3: non-empty left and right subtrees
        else:
            # if we take the max from the left subtree, it will be a good contender for replac
            self._root = self._left.extract_max()

    # For practice, implement the same method but with extract min instead of max
    def _extract_max(self) -> Any:
        if self._right.is_empty():
            max_item = self._root
```

```
            # Promote the left subtree
            self._root, self._left, self._right =\
                self._right._root, self._right._left, self._right._right

            return max_item
        else:
            return self._right._extract_max()
```

3. Check your assumptions: did you assume that the value you were extracting is a leaf? Consider deleting the root of the following tree:



Notice that if your chosen value is *not* a leaf, one of its subtrees must be empty—you can use the "promote a subtree" strategy in this case!

2 Exercise 2: Efficiency of `BinarySearchTree.__contains__`

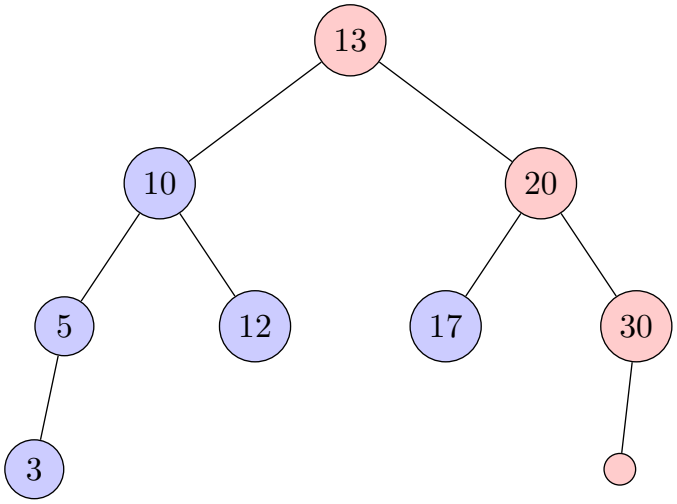
Recall our implementation of `BinarySearchTree.__contains__`:

```
class BinarySearchTree:
    def __contains__(self, item: Any) -> bool:
        if self.is_empty():
            return False
```

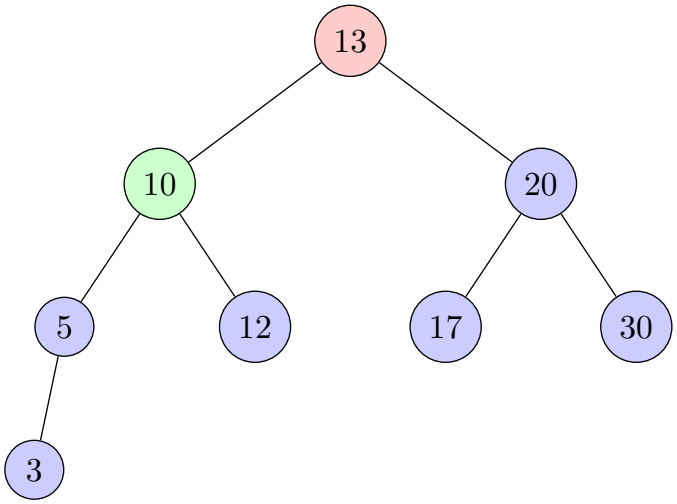
```
5           elif item == self._root:
6               return True
7           elif item < self._root:
8               return self._left.__contains__(item)
9           else:
10              return self._right.__contains__(item)
```

Crucially, after comparing the item against the root of the BST, only *one* subtree is recursed into. To make sure you understand this idea, suppose we have the following BST, and perform three searches, for the items `25`, `10`, and `4`, respectively. Circle all the values that are compared against the target item when we do each search. *Also, draw a new circle for any empty subtree where a recursive call is made.*

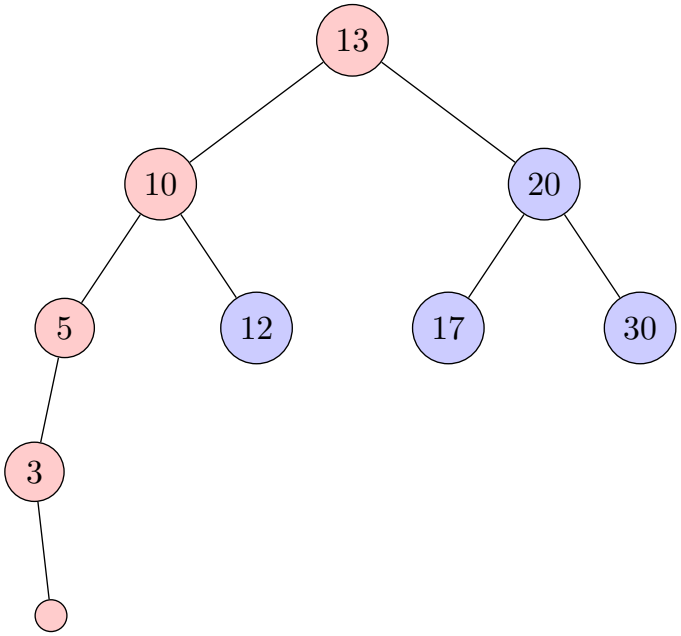
1. Search for `25`:



2. Search for `10`:



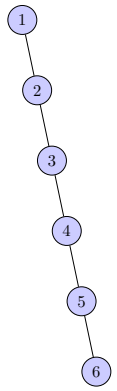
1. Search for `4`:



3 Exercise 3: Investigating the height of binary search trees

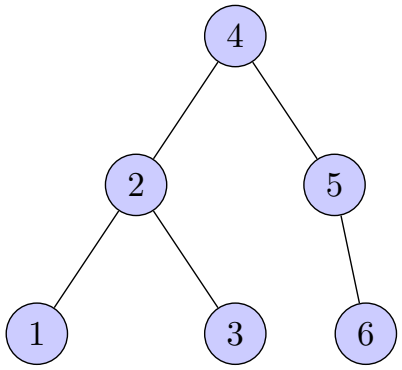
1. Recall that the *BST insertion algorithm* always inserts a new item into an "empty spot" at the bottom of a BST, so that the new item is a leaf of the tree.

   (a) Suppose we start with an empty BST, and then insert the items `1, 2, 3, 4, 5, 6` into the BST, in that order. Draw the final BST.



   Height: 5

(b) Suppose we start with an empty BST, and then insert the items `4, 2, 3, 5, 1, 6` into the BST, in that order. Draw the final BST.



Height: 2

2. Write down the *height* of each BST you drew in the previous question. (You can do so beside or underneath the diagram.)

3. Let $n \in \mathbb{N}$ and suppose we want a BST that contains the values `1, 2, ..., n`. What is the *maximum* possible height of the BST?

   $n - 1$ (as illustrated by 1a)

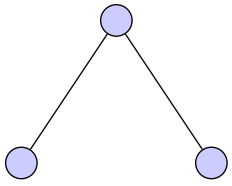
4. Now suppose we want to find the *minimum* possible height of the BST. We're going to investigate this in an indirect way: for every height h, we're going to investigate what the maximum number of values can be stored in a binary tree of height n. Let's try to find a pattern.

   (a) Suppose we have a BST of height **0**. What's the maximum possible number of values in the BST? (Hint: there's only one kind of BST with height 0.)

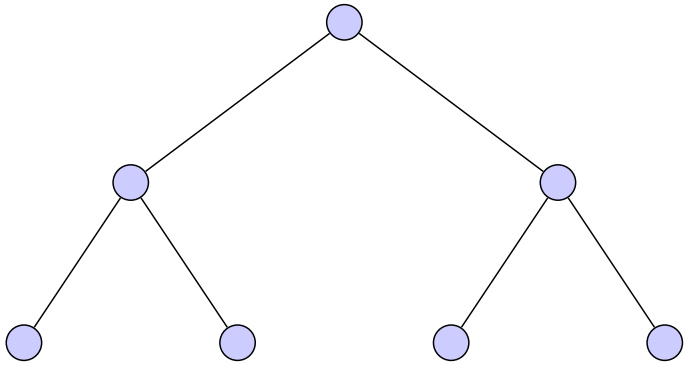
   Max Values: $1 = 2^0 = 2^1 - 1$

   (b) Suppose we have a BST of height **1**. What's the maximum possible number of values in the BST? (Draw an example.)

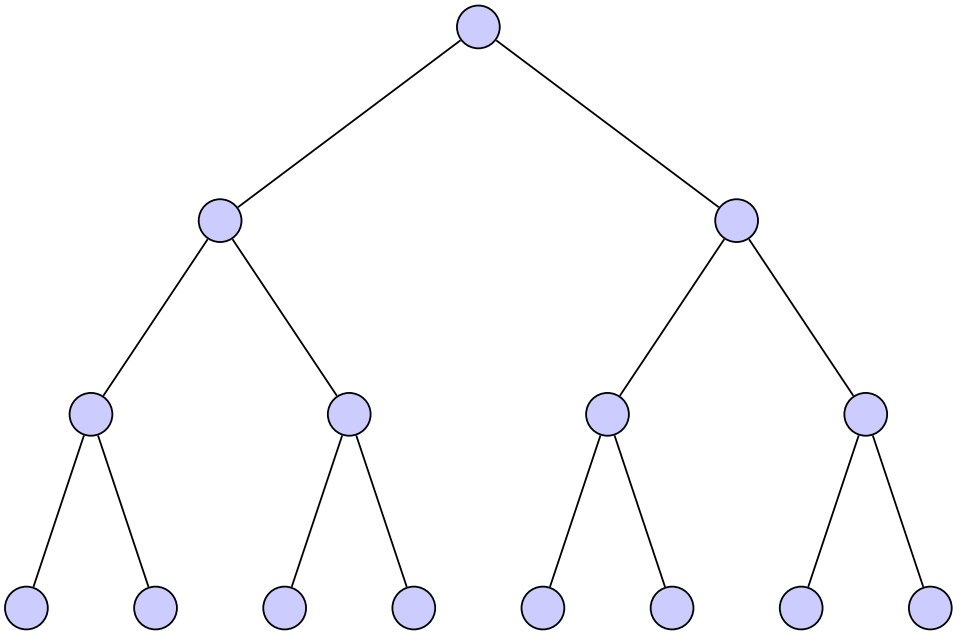
   Max Values: $3 = 2^0 + 2^1 = 2^2 - 1$

(c) Suppose we have a BST of height **2**. What's the maximum possible number of values in the BST? (Draw an example.)



Max Values: $7 = 2^0 + 2^1 + 2^2 = 2^3 - 1$

(d) Repeat for height 3.



Max Values: $15 = 2^0 + 2^1 + 2^2 + 2^3 = 2^4 - 1$

(e) Suppose we have a BST of height **111**. What's the maximum possible number of values in the BST? (*Don't* draw an example, but find a pattern from your previous answers.)

Max Values: $2^0 + 2^1 + 2^2 + \cdots + 2^{110} = \sum_{i=0}^{110} 2^i = 2^{111} - 1$

(f) Suppose we have a BST of height h and that contains n values. Write down an inequality of the form $n \leq ...$ to relate n and h. (This is a generalization of your work so far.)

$n \leq 2^h - 1$

(g) Finally, take your inequality from the previous part and isolate h. This produces the answer to our original question: what is the minimum height of a BST with n values?

$$\begin{aligned} n+1 &\leq 2^h \\ \log(n+1) &\leq h\log(2) \\ \frac{\log(n+1)}{\log(2)} &\leq h \\ h &\geq \frac{\log(n+1)}{\log(2)} \end{aligned}$$

CSC111 Lecture 11: Introduction to Abstract Syntax Trees

Hisbaan Noorani

February 22, 2021

Contents

1 Exercise 1: Representing assignment statements

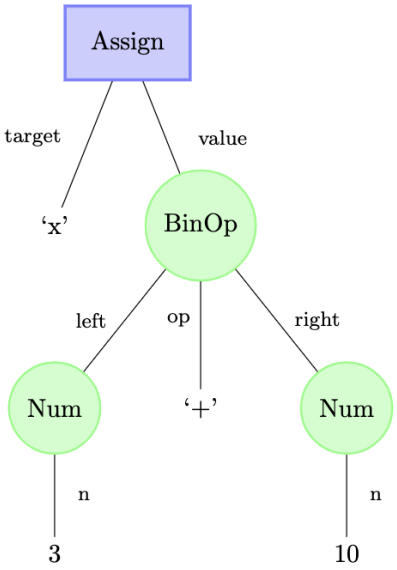
In lecture, we introduced the following class to represent assignment statements in abstract syntax trees.

```python
class Assign(statement):
    """An assignment statement (with a single target).

    Instance Attributes:
      - target: the variable name on the left-hand side of the equals sign
      - value: the expression on the right-hand side of the equals sign
    """
    target: str
    value: Expr

    def __init__(self, target: str, value: Expr) -> None:
        """Initialize a new Assign node."""
        self.target = target
        self.value = value
```

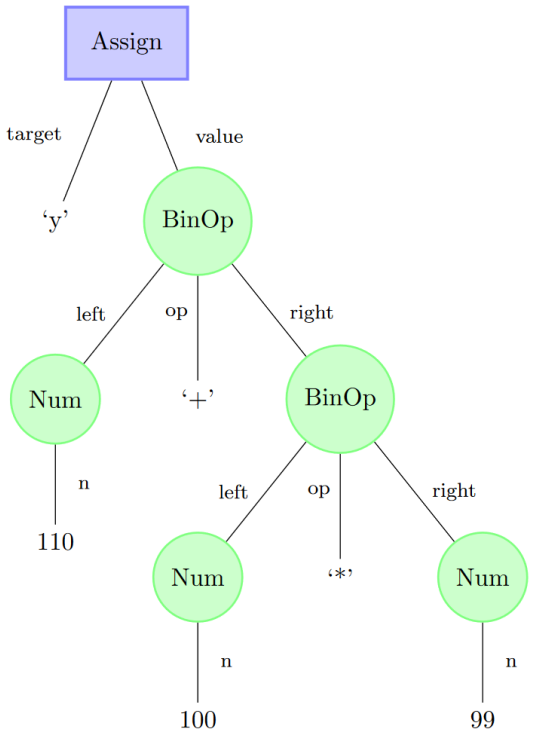
First, make sure you understand this class by answering the following questions.

1. Draw an abstract syntax tree diagram that represents the following Python statement.

```python
x = 10 + 3
```



2. Write an AST expression using `Assign` (and other AST types) to represent the following diagram.



```
Assign('y', BinOp(Num(10), '+', BinOp(Num(100), '*', Num(99)))
```

3. Finally, implement the `Assign.evaluate` method. This method should *mutate* its `env` argument, and shouldn't return anything.

```
class Assign(statement):
    def evaluate(self, env: dict[str, Any]) -> None:
        """Evaluate this statement.

        This does the following: evaluate the right-hand side expression,
        and then update <env> to store a binding between this statement's
        target and the corresponding value.

        >>> stmt = Assign('x', BinOp(Num(10), '+', Num(3)))
        >>> env = {}
        >>> stmt.evaluate(env)
        >>> env['x']
        13
        """
        env[self.target] = self.value.evaluate()
```

2 Additional exercises

1. Let's create a variation of the `Assign` class to support *parallel assignment*. Read through the following class.

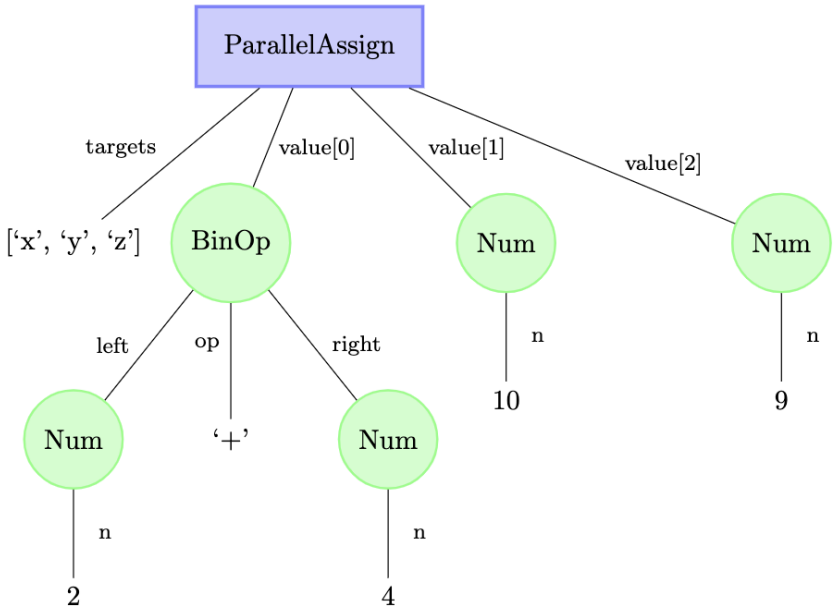
```
class ParallelAssign(Statement):
    """A parallel assignment statement.

    Instance Attributes:
        - targets: the variable names being assigned to---the left-hand side of the =
        - values: the expressions being assigned---the right-hand side of the =
    """
    targets: list[str]
    values: list[Expr]

    def __init__(self, targets: list[str], values: list[Expr]) -> None:
        """Initialize a new ParallelAssign node."""
        self.targets = targets
        self.values = values
```

To make sure you understand this class, answer the following questions.

(a) Draw the abstract syntax tree diagram that represents the following Python statement.

```
ParallelAssign(['x', 'y'],
               [BinOp(Num(10), '+', Num(3)), Num(-4.5)])
```

(b) Write an AST expression using `Assign` (and other AST types) to represent the following diagram.



(c) Now, implement the `ParallelAssign.evaluate` method.

```
class ParallelAssign:
    def evaluate(self, env: dict[str, Any]) -> None:
        """Evaluate this statement.

        This does the following: evaluate each expression on the right-hand side
        and then bind each target to its corresponding expression.

        Raise a ValueError if the lengths of self.targets and self.values are
        not equal.

        >>> stmt = ParallelAssign(['x', 'y'],
        ...                       [BinOp(Num(10), '+', Num(3)), Num(-4.5)])
        >>> env = {}
        >>> stmt.evaluate(env)
        >>> env['x']
        13
        >>> env['y']
        -4.5
        """
```

CSC111 Lecture 12: Abstract Syntax Trees, Continued

Hisbaan Noorani

February 24, 2021

Contents

Note: if you are working in PyCharm, you can use the starter file lecture12.py on Quercus for this exercise.

1 Exercise 1: If statements

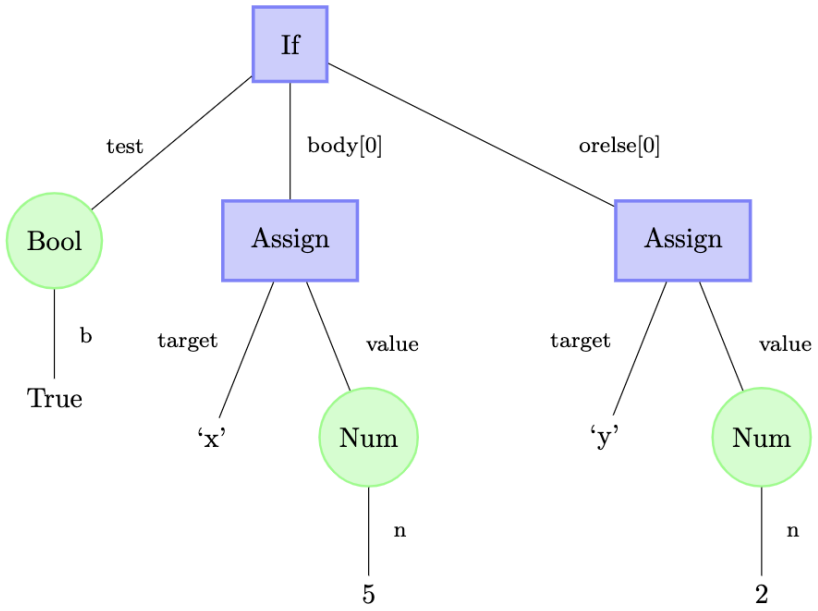
Now that you've seen the `Module` class in lecture, you are ready to implement *compound statements* whose bodies consist of multiple statements. In this exercise, we'll cover if statements.

First, review this new class to represent if statements.

```
class If(Statement):
    """An if statement.

    This is a statement of the form:

        if <test>:
            <body>
        else:
            <orelse>

    Instance Attributes:
        - test: The condition expression of this if statement.
        - body: A sequence of statements to evaluate if the condition is True.
        - orelse: A sequence of statements to evaluate if the condition is False.
    """
    test: Expr
    body: list[Statement]
    orelse: list[Statement]

    def __init__(self, test: Expr, body: list[Statement],
                 orelse: list[Statement]) -> None:
```

```
22          self.test = test
23          self.body = body
24          self.orelse = orelse
```

1. Write an AST expression using `If` (and other AST types) to represent the following diagram.




```
>>> If(Bool(b),
...    [Assign('x', Num(1))],
...    [Assign('y', Num(0))])
```

2. Write an AST expression using `If` (and other AST types) to represent the following Python code.

```
if x < 100:
    print(x)
else:
    y = x + 2
    x = 1
```

```
>>> If(Compare(Name('x'), [('<', Num(100))]),
...    [Print(Name('x'))],
...    [Assign('y', BinOp(Name('x'), '+', Num(2))), Assign('x', Num(1))]
...    )
```

3. In Python, the else branch is optional. How would we represent an if statement with no `else` branc using our `If` class?

   We can use an empty list for the `orelse` branch.

4. Implement the `If.evaluate` method. Note that you use can use if statements in your implementation (similar to how we used the `+` and `*` operators to implement `BinOp`).

```
class If:
    def evaluate(self, env: Dict[str, Any]) -> None:
        """Evaluate this statement.

        Preconditions:
            - self.test evaluates to a boolean

        >>> stmt = If(Bool(True),
        ...           [Assign('x', Num(1))],
        ...           [Assign('y', Num(0))])
        ...
        >>> env = {}
        >>> stmt.evaluate(env)
        >>> env
        {'x': 1}
        """
        test_val = self.test.evaluate(env)

        if test_val:
            for statement in self.body:
                statment.evaluate(env)
        else:
            for statement in self.orelse:
                statment.evaluate(env)
```

2 Exercise 2: For loops (over ranges)

Please take a moment to review the `ForRange` class we introduced in lecture.

```
class ForRange(Statement):
    """A for loop that loops over a range of numbers.

        for <target> in range(<start>, <stop>):
            <body>

    Instance Attributes:
        - target: The loop variable.
        - start: The start for the range (inclusive).
        - stop: The end of the range (this is *exclusive*, so <stop> is not included
                in the loop).
```

```
        - body: The statements to execute in the loop body.
    """
    target: str
    start: Expr
    stop: Expr
    body: list[Statement]

    def __init__(self, target: str, start: Expr, stop: Expr,
                 body: list[Statement]) -> None:
        """Initialize a new ForRange node."""
        self.target = target
        self.start = start
        self.stop = stop
        self.body = body
```

1. Write the Python code that corresponds to the following AST expression.

```
>>> ForRange('x',
...          Num(1),
...          BinOp(Num(2), '+', Num(3)),
...          [Print(Name('x'))])
```

```
for x in range(1, 2 + 3):
    print(x)
```

2. Write the `Module` AST expression that corresponds to the following Python code:

```
sum_so_far = 0

for n in range(1, 10):
    sum_so_far = sum_so_far + n

print(sum_so_far)
```

Hint: Your `Module` body should have three elements (an `Assign`, a `ForRange`, and a `Print`).

```
>>> Module([
...     Assign('sum_so_far', Num(0)),
...     ForRange('n', Num(1), Num(10),
...              [Assign('sum_so_far', BinOp(Name('sum_so_far'), '+', Name('n')))]),
...     Print(Name('x'))
... ])
```

3. Finally, implement the `ForRange.evaluate` method. Think carefully about how you make the loop variable accessible when you evaluate the statements in the loop body—you'll need to update the variable environment. Note that you use can use for loops in your implementation!

```python
class ForRange:
    def evaluate(self, env: Dict[str, Any]) -> None:
        """Evaluate this statement.

        Preconditions:
            - self.start and self.stop evaluate to integers

        >>> statement = ForRange('x', Num(1), BinOp(Num(2), '+', Num(3)),
        ...                      [Print(Name('x'))])
        >>> statement.evaluate({})
        1
        2
        3
        4
        """
        start_val = self.start.evaluate(env)
        stop_val = self.stop.evaluate(env)

        for i in range(start_val, stop_val):
            # Need to assign self.target to have value i
            # NOTE: only use one of the two versions

            # Version 1:
            env[self.target] = i

            # Version 2:
            Assign(self.target, Num(i)).evaluate(env)

            for statement in self.body:
                statement.evaluate(env)
```

3 Additional exercises

1. Let's add some *type-checking* to the methods we implemented in this exercise.

   (a) Modify the `If.evaluate` method to raise a `TypeError` if its condition doesn't evaluate to a boolean. (This is technically stricter than real Python.)

```python
class If:
    def evaluate(self, env: Dict[str, Any]) -> None:
        """Evaluate this statement.

        Preconditions:
            - self.test evaluates to a boolean

        >>> stmt = If(Bool(True),
```

```
        ...             [Assign('x', Num(1))],
        ...             [Assign('y', Num(0))])
        ...
        >>> env = {}
        >>> stmt.evaluate(env)
        >>> env
        {'x': 1}
        """
        test_val = self.test.evaluate(env)

        if not isinstance(test_val, bool):
            raise TypeError

        if test_val:
            for statement in self.body:
                statment.evaluate(env)
        else:
            for statement in self.orelse:
                statment.evaluate(env)
```

(b) Modify the `ForRange.evaluate` method to raise a `TypeError` if its start or stop expressions don't evaluate to an integer.

2. Modify the `If` class to support an arbitrary number of `elif` branches. You may add additional instance attributes.

3. Design and implement a new AST `Statement` subclass to represent a while loop.

4. Design and implement a new AST `Expr` subclass to represent a list comprehension over a range of numbers.

CSC111 Lecture 13: Introduction to Graphs

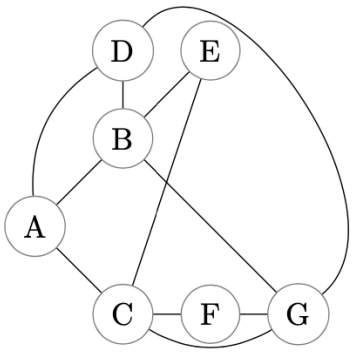
Hisbaan Noorani

March 1, 2021

Contents

1 Exercise 1: Graph terminology review

One of the tricky things about learning graphs is that there's a lot of terminology to understand. This exercise will give you the opportunity to practice using this terminology on a concrete example. Consider the graph below:



1. How many vertices does this graph have?

   There are 7 vertices: $V = \{A, B, C, D, E, F, G\}$.

2. How many edges does this graph have?

   There are 11 edges: $E = \{\{A, B\}, \{A, C\}, \{A, D\}, \{B, D\}, \{B, E\}, \{B, G\}, \{C, E\}, \{C, F\}, \{C, G\}, \{D, G\}, \{F, G\}\}$.

3. List all the vertices that are adjacent to vertex G.

   $\{B, C, D, F\}$.

4. Find a *path* that goes through all vertices of the graph. (Remember that a path cannot have any duplicate vertices.)

   One such path is A, D, G, F, C, E, B.

2 Exercise 2: A property of vertex degrees

Recall that the **degree** of a vertex v, denoted $d(v)$, is its number of neighbours.

Answer the following questions about this definition.

1. Let $G = (V, E)$ be the graph from Exercise 1. Complete the table below showing the *degree* of each vertex. We've done the first row for you.

Vertex	Degree
A	3
B	4
C	4
D	3
E	2
F	2
G	4

2. What is the *sum of the degrees* of all the vertices in the above table?

   $3 + 4 + 4 + 3 + 2 + 2 + 4 = 22$

3. Compare your answer to Question 2 and the *number of edges* of this graph (Question 2 in Exercise 1). What do you notice?

   The sum of the degrees of all the vertices in the graph is equal to double the number of edges in the graph. $\sum_{v \in V} d(v) = 2 \cdot |E|$.

4. Prove the following graph property: for all graphs $G = (V, E)$, $\sum_{v \in V} d(v) = 2 \cdot |E|$. Your proof body can consist of a short explanation written in English.

   Note: $\sum_{v \in V}$ means "sum over all vertices v in V".

   WTS $\forall G = (V, E), \sum_{v \in V} d(v) = 2 \cdot |E|$

   Proof:

   Let $G = (V, E)$ be an arbitrary graph

   For a vertex v, $d(v)$, is the number of edges that "touch" (incident) that vertex.

   We know that each edge touches exactly 2 vertices. This means each edge will be counted by exactly two of the $d(v)$ expressions in the summation. 1 egde will represent 2 total degrees. 4 edges will represent 8 total degrees.

   Thus we know that the sum of the $d(v)$ (over all $v \in V$) is equal to double the number of edges. ■

3 Additional exercises

1. Let $G = (V, E)$ be a graph, and assume that for all $v \in V$, $d(v) \leq 5$. Find and prove a good upper bound (exact, not asymptotic) on the total number of edges, $|E|$, in terms of the number of vertices, $|V|$.

   Formally, you can think of this as proving the following statement (after filling in the blank):

   $$\forall G = (V, E),\ \big(\forall v \in V,\ d(v) \leq 5\big) \Rightarrow |E| \leq \_\_\_\_\_$$

   See the lecture slide starting titled "Graphs and induction" for the proof

CSC111 Lecture 14: Representing Graphs in Python

Hisbaan Noorani

March 3, 2021

Contents

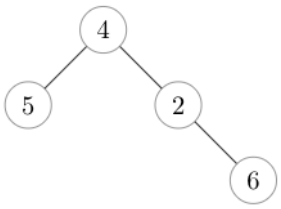
For your reference, here are the `_Vertex` and `Graph` classes we introduced in lecture.

```python
from __future__ import annotations
from typing import Any


class _Vertex:
    """A vertex in a graph.

    Instance Attributes:
        - item: The data stored in this vertex.
        - neighbours: The vertices that are adjacent to this vertex.
    """
    item: Any
    neighbours: set[_Vertex]

    def __init__(self, item: Any, neighbours: set[_Vertex]) -> None:
        """Initialize a new vertex with the given item and neighbours."""
        self.item = item
        self.neighbours = neighbours


class Graph:
    """A graph.
    """
    # Private Instance Attributes:
    #     - _vertices:
    #         A collection of the vertices contained in this graph.
    #         Maps item to _Vertex object.
    _vertices: dict[Any, _Vertex]

```

```
    def __init__(self) -> None:
        """Initialize an empty graph (no vertices or edges)."""
        self._vertices = {}

    def add_vertex(self, item: Any) -> None:
        """Add a vertex with the given item to this graph.

        The new vertex is not adjacent to any other vertices.

        Preconditions:
            - item not in self._vertices
        """
        self._vertices[item] = _Vertex(item, set())

    def add_edge(self, item1: Any, item2: Any) -> None:
        """Add an edge between the two vertices with the given items in this graph.

        Raise a ValueError if item1 or item2 do not appear as vertices in this graph.

        Preconditions:
            - item1 != item2
        """
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            v2 = self._vertices[item2]

            # Add the new edge
            v1.neighbours.add(v2)
            v2.neighbours.add(v1)
        else:
            # We didn't find an existing vertex for both items.
            raise ValueError
```

1 Exercise 1: Reviewing the `Graph` implementation

1. Consider the following graph:



Write the Python code that we could use to represent this graph. We've started by creat-

ing an empty `Graph` for you, which you should mutate with calls to `Graph.add_vertex` and `Graph.add_edge`:

```
>>> graph = Graph()
>>> graph.add_vertex(2)
>>> graph.add_vertex(4)
>>> graph.add_vertex(5)
>>> graph.add_vertex(6)
>>> graph.add_edge(2, 4)
>>> graph.add_edge(2, 6)
>>> graph.add_edge(4, 5)
```

2. Complete the following function, which creates and returns a graph of `n` vertices where every vertex is adjacent to every other vertex.

```
def complete_graph(n: int) -> Graph:
    """Return a graph of n vertices where all pairs of vertices are adjacent.

    The vertex items are the numbers 0 through n - 1, inclusive.

    Preconditions:
        - n >= 0
    """
    graph_so_far = Graph()

    for i in range(0, n):
        graph_so_far.add_vertex(i)

        for j in range(0, i):
            graph_so_far.add_edge(i, j)

    return graph_so_far
```

3. Finally, add two representation invariants to the `_Vertex` class to represent the following restrictions on edges in a graph:

   - a vertex cannot be a neighbour of itself
   - edges are symmetric: for any vertex v, all of its neighbours have v in their `neighbours` set

```
class _Vertex:
    """A vertex in a graph.

    Instance Attributes:
        - item: The data stored in this vertex.
        - neighbours: The vertices that are adjacent to this vertex.

```

```
    Representation Invariants:
        - self not in self.neighbours
        - all(self in n.neighbours for n in self.neighbours)
    """
    item: Any
    neighbours: set[_Vertex]
```

2 Exercise 2: Writing `Graph` functions

In this exercise, you'll get some practice writing some methods to operate on our new `_Vertex` and `Graph` data types.

1. Implement the `Graph` method below.

```
class Graph:
    def adjacent(self, item1: Any, item2: Any) -> bool:
        """Return whether item1 and item2 are adjacent vertices in this graph.

        Return False if item1 or item2 do not appear as vertices in this graph.
        """
        if item1 in self._vertices and item2 in self._vertices:
            v1 = self._vertices[item1]
            v2 = self._vertices[item2]

            # Return whether the two vertices have an edge
            # This will handle them both being in the graph
            # but not being neighbours
            return v1 in v2.neighbours and v2 in v1.neighbours

        # Return False when they are not in the graph
        return False
```

2. Implement the `Graph` method below.

   Hint: use the statement from Lecture 13 Exercises, Part 2 Q4

```
class Graph:
    def num_edges(self) -> int:
        """Return the number of edges in this graph."""
        total_degree = 0

        for item in self._vertices:
            total_degree += len(self._vertices[item].neighbours)

        # Bonus comprehension implementation!
        # total_degree = sum(len(self._vertices[item].neighbours)
        #                    for item in self._vertices)
```

```
12
13          return total_degree // 2
14
15          # You could even do this all in a oneliner:
16          return sum(len(self._vertices[item].neighbours)
17                     for item in self._vertices) // 2
```

3 Additional exercises

1. Write a `Graph` method that returns the maximum degree of a vertex in the graph (assuming the graph has at least one vertex).

2. Write a `Graph` method that returns a list of all edges (represented as `sets` of items) in the graph. Don't worry about order in the list.

3. (*harder*) A **triangle** in a graph is a set of three vertices that are all adjacent to each other. Write a `Graph` method that returns a list of all triangles in the graph.

CSC111 Lecture 16: Graph Connectivity and Spanning Trees

Hisbaan Noorani

March 19, 2021

Contents

1 Exercise 1: Proving "Lemma 2"

Here's the lemma we just saw in lecture:

Lemma 2: Let $G = (V, E)$ be a graph. If there exists an edge $e \in E$ such that $G - e$ is connected, then that edge e is in a cycle in G, the original graph.

Prove this lemma. *Hint*: the proof body is actually quite short, and can again be written in English! Take the edge e from the assumption and label its endpoints u and v. What can you say about these two vertices in the graph $G - e$?

Proof.

Let $G = (V, E)$ be a graph, and assume there exists an edge e such that $G - e$ is connected. We want to prove e is in a cycle in G.

Let u and v be the endpoints of e. Since we assumed that $G - e$ is connected, there must be a path between u and v in $G - e$. So then let the path equal $u, v_1, v_2, v_3, \ldots, v$

Then the sequence $u, v_1, v_2, v_3, \ldots, v, u$ is a cycle in G that contains e. ■

2 Exercise 2: Proving "Theorem 2 (number of edges in a tree)"

Here's the theorem we just saw in lecture:

Theorem 2 (number of edges in a tree). Let $G = (V, E)$ be a tree with at least one vertex. Then $|E| = |V| - 1$.

Translation into predicate logic, using a variable n:

$$\forall n \in \mathbb{Z}^+,\ \forall G = (V, E), \big(|V| = n \wedge G \text{ is a tree}\big) \implies |E| = n - 1$$

The induction predicate $P(n)$ translates to, "Every tree with n vertices has $n-1$ edges".

In this exercise, you'll prove this theorem using induction on n, the number of vertices in the tree. We've started the proof structure for you.

Proof.

Base case: let $n = 1$.

Let $G = (V, E)$ be a graph, and assume $|V| = 1$ and G is a tree. We want to prove that $|E| = 1 - 1 = 0$

Since there is only one vertex, there cannot be any edges. So $|E| = 0$, as needed ■.

Induction step: let $k \in \mathbb{Z}^+$ and assume that $P(k)$ holds, i.e., every tree with k vertices has $k-1$ edges. We need to prove that $P(k+1)$ is true.

[NOTE: you may assume that every tree with ≥ 2 vertices has at least one vertex of degree 1. Consider removing such a vertex.]

Let $G = (V, E)$. Assume G is a tree, and $|V| = k + 1$. We want to prove that $|E| = k$

Let v be a vertex with degree 1 (from the NOTE above).

Let $G' = (V', E')$ be the graph obtained by removing v from G.

Then G' has no cycles (because G had no cycles), and is still connected (as we only removed v which had degree 1).

This means that G' is a tree. Also it has k vertices, since we just removed 1 vertex from G. By the induction hypothesis, $|E'| = k - 1$.

And hten for G since we removed one edge, we have:

$|E| = |E'| + 1 = (k-1) + 1 = k$ ■

3 Additional exercises

1. Prove that every tree with ≥ 2 vertices has at least one vertex of degree 1. (*Hint*: consider a path of maximum length in the tree. Pick an endpoint, and prove that it must have degree 1.)
2. Prove that every tree with ≥ 2 vertices has at least *two* vertices of degree 1.
3. Prove or disprove: every tree with ≥ 2 vertices has at least *three* vertices of degree 1.

CSC111 Lecture 17: Iterative Sorting Algorithms, Part 1

Hisbaan Noorani

March 15, 2021

Contents

1 Exercise 1: Implementing selection sort

Here is the skeleton of a selection sort algorithm we developed in lecture:

```
def selection_sort(lst: list) -> None:
    """Sort the given list using the selection sort algorithm.

    Note that this is a *mutating* function.

    >>> lst = [3, 7, 2, 5]
    >>> selection_sort(lst)
    >>> lst
    [2, 3, 5, 7]
    """
    for i in range(0, len(lst)):
        # Loop invariants
        #   - lst[:i] is sorted
        #   - if i > 0, lst[i - 1] is less than all items in lst[i:]

        # Find the index of the smallest item in lst[i:] and swap that
        # item with the item at index i.
        index_of_smallest = _min_index(lst, i)
        lst[index_of_smallest], lst[i] = lst[i], lst[index_of_smallest]


def _min_index(lst: list, i: int) -> int:
    """Return the index of the smallest item in lst[i:].

    In the case of ties, return the smaller index (i.e., the index that appears first).
```

```
    Preconditions:
        - 0 <= i <= len(lst) - 1

    >>> _min_index([2, 7, 3, 5], 1)
    2
    """
    min_index_so_far = i
    for x in range(i + 1, len(lst)):
        if lst[x] < lst[min_index_so_far]:
            min_index_so_far = x

    return min_index_so_far
```

Complete this implementation by implementing the helper function `_min_index`. Hint: this is similar to one of the functions you implemented on this week's prep!

2 Exercise 2: Running-time analysis

1. Analyse the running time of the helper function `_min_index` in terms of n, the length of the input `lst`, and/or i, the second argument.

   We will assume that the smallest item is at the end of the list for a worst case running time analysis.

```
def _min_index(lst: list, i: int) -> int:
    min_index_so_far = i                    # 1 step
    for x in range(i + 1, len(lst)):        # iterates n − i − 1 times
        if lst[x] < lst[min_index_so_far]:  # 1 step for whole if blcok
            min_index_so_far = x

    return min_index_so_far                 # 1 step
```

   Therefore an upper bound for the running time is $2 + n - i - 1 = 1 + n - i \in \mathcal{O}(n - i)$.

   Next, we will can find an input family that proves a lower bound, but we will ommit it in this example.

   Therefore, since we have found both an upper and lower bound that match, $RT_{\texttt{\_min\_index}} \in \Theta(n - i)$.

2. Analyse the running time of `selection_sort`.

   We will assume that the list to be sorted is in reverse order, thus slection sort will take as long as possible.

```
def selection_sort(lst: list) -> None:
    for i in range(0, len(lst)):                # n steps
        index_of_smallest = _min_index(lst, i)  # n − i steps
        lst[index_of_smallest], lst[i] =\       # 1 step
            lst[i], lst[index_of_smallest]
```

The loop runs n times for $i = 0, 1, \ldots, n-1$

The iteration i takes $n - i$ steps (because of `_min_index(lst, i)`) plus one step for constant time operations.

Therefore $RT_{\texttt{selection\_sort}} = \sum_{i=0}^{n-1}(n - i + 1) \in \Theta(n^2)$.

3 Additional exercises

1. Translate the two loop invariants in `selection_sort` into Python assert statements. You can use `is_sorted=/=is_sorted_sublist` from this week's prep.

   (One version is included in the Course Notes, but it's a good exercise for you to try it yourself without looking there first!)

CSC111 Lecture 18: Iterative Sorting Algorithms, Part 2

Hisbaan Noorani

March 17, 2021

Contents

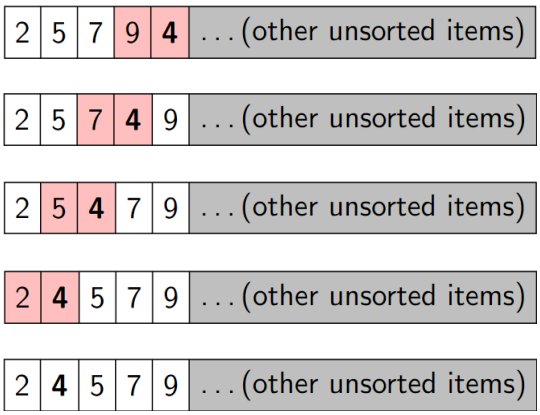
1 Exercise 1: Implementing `_insert`

In lecture, we implemented the `insertion_sort` algorithm using a helper, `_insert`.

```
def insertion_sort(lst: list) -> None:
    """Sort the given list using the insertion sort algorithm.
    """
    for i in range(len(lst)):
        # Invariants:
        # - lst[:i] is sorted
        _insert(lst, i)


def _insert(lst: list, i: int) -> None:
    """Move lst[i] so that lst[:i + 1] is sorted.

    Preconditions:
        - 0 <= i < len(lst)
        - lst[:i] is sorted

    >>> lst = [7, 3, 5, 2]
    >>> _insert(lst, 1)
    >>> lst
    [3, 7, 5, 2]
    """
    j = i
    while j == 0 or lst[j] < lst[j - 1]:
        # Do the swap
```

```
25          lst[j], lst[j - 1] = lst[j - 1], lst[j]
26          j = j - 1
```

One way to efficiently implement `_insert` is to repeatedly swap the element at index `i` with the one to its left until it reaches its correct spot in the sorted list.



Using this idea, implement the `_insert` helper function. *Hint*: this is similar to a function from this week's prep.

2 Exercise 2: Running-time analysis

Your implementation of `_insert` (or the one we saw in class) has a spread of running times, since the number of loop iterations depends on the value of `lst[i]` and the other list items before it. This means that we'll need to do a worst-case running-time analysis for our code.

1. Find (with analysis) a good asymptotic upper bound on the worst-case running time of `_insert`, in terms of n, the size of the input list, and/or i, the value of the second argument.

   Assume a reversed list. This means that the list will be in the form $[n, n-1, \ldots, 1, 0]$.

   This means that we will always need to swap a given element to the start of the list.

   This gives `_insert` an upper bound for a worst case running time of $\mathcal{O}(i)$

2. Find an input family for `_insert` whose asymptotic running time matches the upper bound you found in the previous question. To save time, you do *not* need to analyse the running time for this input family, just describe what the input family is. This lets you conclude a Theta bound for the worst-case running time of `_insert`.

   Let $i \in \mathbb{N}$ and assume $i < n$.

   Let `lst` $= [4, 3, 2, 1, 0]$

   This input family gives `_insert` a lower bound for a worst case running time of $\Omega(i)$

   This, combined with the upper bound gives a worst case running time of $\Theta(i)$

3. Now, refer to the `insertion_sort` implementation at the top of this exercise. Find (with analysis) a good asymptotic upper bound on the worst-case running time of `_insert`, in terms of n, the size of the input list.

   The `for` loop runs n times, for $i = 0, 1, \ldots, n-1$.

   Each iteration takes at most i steps.

   Since i is increasing each iteration, this leads to an upper bound for a worst case running time of $\mathcal{O}(n^2)$.

4. Finally, find an input family for `insertion_sort` whose asymptotic running time matches the upper bound you found in the previous question. To save time, you do *not* need to analyse the running time for this input family, just describe what the input family is.

   This lets you conclude a Theta bound for the worst-case running time of `insertion_sort`.

   Hint: look carefully at the property you used for the input family in Question 2, and try to pick a list so that this property holds for *every* index `i`.

   Let $n \in \mathbb{N}$.

   Let `lst` $= [n-1, n-2, \ldots, 1, 0]$.

   In this case, for every $i \in \mathbb{N}$ with $i < n$, `lst[i]` is always less than every element in `lst[:i]`.

   This leads to a lower bound for a worst case running time of $\Omega(n^2)$.

   This, in combination with the upper bound gives a worst case running time of $\Theta(n^2)$.

3 Exercise 3: Saving `key` values

Here is the start of a *memoized* version of insertion sort that we started in lecture. Your task: complete this algorithm by implementing a new `_insert_memoized` helper function.

Hint: this function is very similar to `_insert_key`, except for the places where `key` is called.

```
def insertion_sort_memoized(lst: list, key: Optional[Callable] = None) -> None:
    """Sort the given list using the insertion sort algorithm.

    If key is not None, sort the items by their corresponding return value when passed to key.
    Use a dictionary to keep track of "key" values, so that the function is called only once per
    list element.

    Note that this is a *mutating* function.

    >>> lst = ['cat', 'octopus', 'hi', 'david']
    >>> insertion_sort_memoized(lst, key=len)
    >>> lst
    ['hi', 'cat', 'david', 'octopus']
    >>> lst2 = ['cat', 'octopus', 'hi', 'david']
    >>> insertion_sort_memoized(lst2)
    >>> lst2
    ['cat', 'david', 'hi', 'octopus']
    """
```

```
    # Use this variable to keep track of the saved "key" values
    # across the different calls to _insert.
    key_values = {}

    for i in range(0, len(lst)):
        _insert_memoized(lst, i, key, key_values)

# Define the _insert_memoized helper below.
# Hint: You'll need to modify the _insert_key helper function to take an
# additional dictionary argument.

def _insert_memoized(lst: list, i: int, key: Optional[Callable] = None, key_values: dict) -> None:
    for j in range(i, 0, -1):  # This goes from i down to 1
        if key is None:
            if lst[j - 1] <= lst[j]:
                return
            else:
                # Swap lst[j - 1] and lst[j]
                lst[j - 1], lst[j] = lst[j], lst[j - 1]
        else:
            if lst[j - 1] not in key_values:
                key_values[lst[j - 1] = key(lst[j - 1])
            if lst[j] not in key_values:
                key_values[lst[j] = key(lst[j])

            if key_values[lst[j - 1]] <= key_values(lst[j]):
                return
            else:
                # Swap lst[j - 1] and lst[j]
                lst[j - 1], lst[j] = lst[j], lst[j - 1]#+end_src
```

4 Additional exercises

1. Implement “key” and memoized versions of the selection sort algorithm from last class.

CSC111 Lecture 19: Recursive Sorting Algorithms, Part 1

Hisbaan Noorani

March 22, 2021

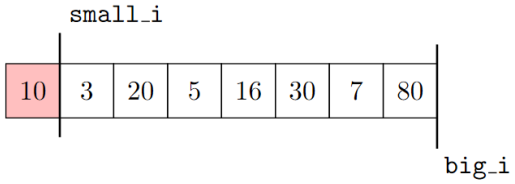
Contents

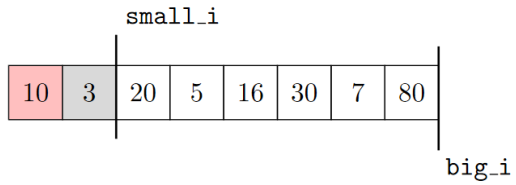
1 Exercise 1: In-place partitioning

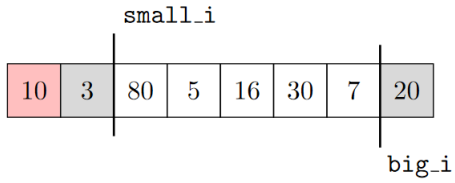
We're going to study how to implement an *in-place* version of `_partition` that mutates `lst` directly, without creating any new lists. This is the key step to implementing an in-place version of quicksort.

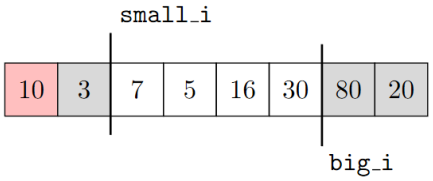
```
def _in_place_partition(lst: list) -> None:
    """Mutate lst so that it is partitioned with pivot lst[0].

    Let pivot = lst[0]. The elements of lst are moved around so that lst looks like

        [x1, x2, ... x_m, pivot, y1, y2, ... y_n],

    where each of the x's is <= pivot, and each of the y's is > pivot.

    >>> lst = [10, 3, 20, 5, 16, 30, 7, 100]
    >>> _in_place_partition(lst)  # pivot is 10
    >>> lst[3]  # the 10 is now at index 3
    10
    >>> set(lst[:3]) == {3, 5, 7}
    True
    >>> set(lst[4:]) == {16, 20, 30, 100}
    True
    """
```

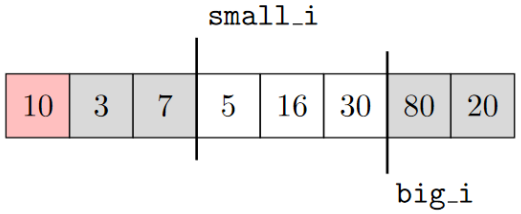
For your reference, there are the diagrams of the example we saw in lecture:

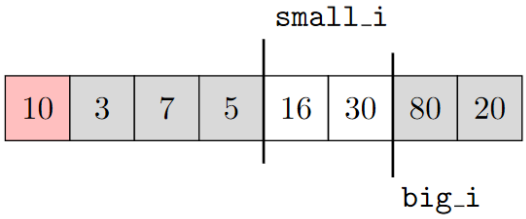


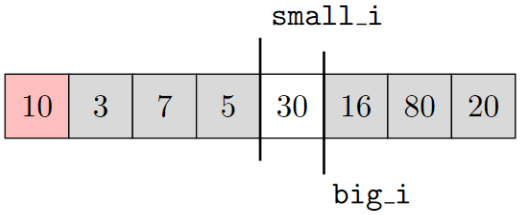

small_i
10
3
20
5
16
30
7
80
big_i

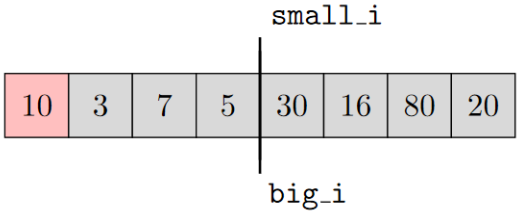

small_i
10
3
80
5
16
30
7
20
big_i

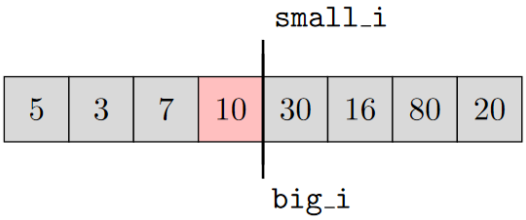

small_i
10
3
7
5
16
30
80
20
big_i


small_i
10
3
7
5
16
30
80
20
big_i


small_i
10
3
7
5
16
30
80
20
big_i


small_i
10
3
7
5
30
16
80
20
big_i


small_i
10
3
7
5
30
16
80
20
big_i



1. The key idea to implement this function is to divide up `lst` into three parts using indexes `small_i` and `big_i`:

   - `lst[1:small_i]` contains the elements that are known to be $<$ `lst[0]` (the "smaller partition")
   - `lst[big_i:]` contains the elements that are known to be $>$ `lst[0]` (the "bigger partition")
   - `lst[small_i:big_i]` contains the elements that have not yet been compared to the pivot (the "unsorted section")

   `_in_place_partition` uses a loop to go through the elements of the list and compare each one to the pivot, building up either the "smaller" or the "larger partition" and shrinking the "unsorted section."

   (a) When we first begin the algorithm, we know that `lst[0]` is the pivot, but have not yet compared it against any other list element. What should the initial values of `small_i` and `big_i` be?

   `small_i = 1`

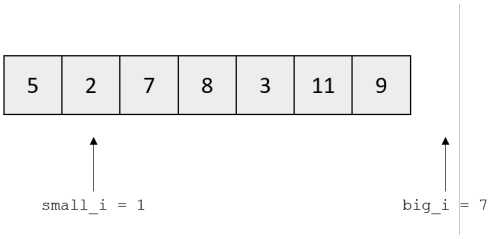
   `big_i = len(lst)`

   (b) We need to check the items one at a time, until every item has been compared against the pivot. What is the relationship between `small_i` and `big_i` when we have finished checking every item?

   Once we have finished checking every single item, `small_i = big_i`. If we were to do some sort of recursive implementation (which we don't need to do) then this would likely be our base case.

   (c) On each loop iteration, the element `lst[small_i]` is compared to the pivot.

   - Suppose `lst[small_i]` is less than or equal to the pivot, as in the diagram below.

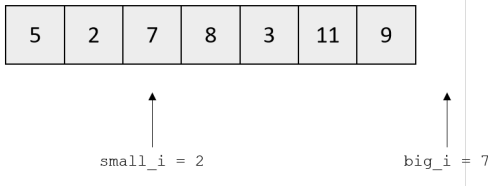


   Which partition should `lst[small_i]` belong to? Write the code to update `small_i` to add this element to the correct the partition.

   `lst[small_i]` should be a part of the $<$ partition. In this case, we will only increment the `small_i` variable.

```
small_i += 1
```

- Now suppose `lst[small_i]` is greater than the pivot, as in the diagram below.



  Which partition should `lst[small_i]` belong to? Write the code to swap this element and update `small_i` or `big_i` to add this element to the correct partition.
  `lst[small_i]` should be a part of the $>$ partition. In this case, we will swap the items.

```
lst[small_i], lst[big_i - 1] = lst[big_i - 1], lst[small_i]
big_i -= 1
```

2. Next, implement `_in_place_partition`. Make sure you're confident in your answers to the previous questionsbefore writing the main loop!

```
def _in_place_partition(lst: list) -> None:
    # Initialize variables
    pivot = lst[0]
    small_i = 1
    big_i = len(lst)

    # The main loop
    while small_i < big_i:
        # Loop invariants
        assert all(lst[j] <= pivot for j in range(1, small_i))
        assert all(lst[j] > pivot for j in range(big_i, len(lst)))

        if lst[small_i] <= pivot:
            # Increase the "smaller" partition
            small_i +=1
        else:
            # Swap lst[small_i] to back and increase the "bigger" partition
            lst[small_i], lst[big_i - 1] = lst[big_i], lst[small_i]
            big_i -= 1

    # At this point, all elements have been compared.
    # The final step is to move the pivot into its correct position in the list
    # (after the "smaller" partition, but before the "bigger" partition).
    lst[0], lst[small_i - 1] = lst[small_i - 1], lst[0]
    # Could have used big_i as well, since they're equal
```

3. We're going to need to make this helper a bit more general in order to use it in our a modified quicksort algorithm. If you still have time, complete the following modifications to your implementation of `_in_place_partition`.

   (a) Modify your implementation so that it returns the *new index of the pivot* in the list (so that when `_in_place_partition` is called in quicksort, we know where the partitions start and end).

```
def _in_place_partition(lst: list) -> None:
    # Initialize variables
    pivot = lst[0]
    small_i = 1
    big_i = len(lst)

    while small_i < big_i:
        assert all(lst[j] <= pivot for j in range(1, small_i))
        assert all(lst[j] > pivot for j in range(big_i, len(lst)))

        if lst[small_i] <= pivot:
            small_i +=1
        else:
            lst[small_i], lst[big_i - 1] = lst[big_i], lst[small_i]
            big_i -= 1

    lst[0], lst[small_i - 1] = lst[small_i - 1], lst[0]
    return small_i - 1
```

   (b) Modify your implementation so that it takes in two additional parameters `b` and `e`, so that the function only partitions the range `lst[b:e]` rather than the entire list.

```
def _in_place_partition(lst: list, b: int, e: int) -> int:
    # Initialize variables
    pivot = lst[b]
    small_i = b + 1
    big_i = e

    while small_i < big_i:
        # assert all(lst[j] <= pivot for j in range(1, small_i))
        # assert all(lst[j] > pivot for j in range(big_i, len(lst)))

        if lst[small_i] <= pivot:
            small_i +=1
        else:
            lst[small_i], lst[big_i - 1] = lst[big_i], lst[small_i]
            big_i -= 1

    lst[b], lst[small_i - 1] = lst[small_i - 1], lst[b]
    return small_i - 1
```

CSC111 Lecture 20: Recursive Sorting Algorithms, Part 2

Hisbaan Noorani

March 24, 2021

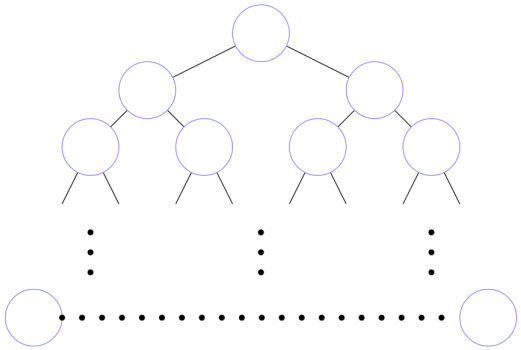
Contents

1 Exercise 1: Running-time analysis for mergesort

Here is our mergesort implementation.

```
def mergesort(lst: list) -> list:
    if len(lst) < 2:
        return lst.copy()  # Use the list.copy method to return a new list object
    else:
        # Divide the list into two parts, and sort them recursively.
        mid = len(lst) // 2
        left_sorted = mergesort(lst[:mid])
        right_sorted = mergesort(lst[mid:])

        # Merge the two sorted halves. Using a helper here!
        return _merge(left_sorted, right_sorted)
```

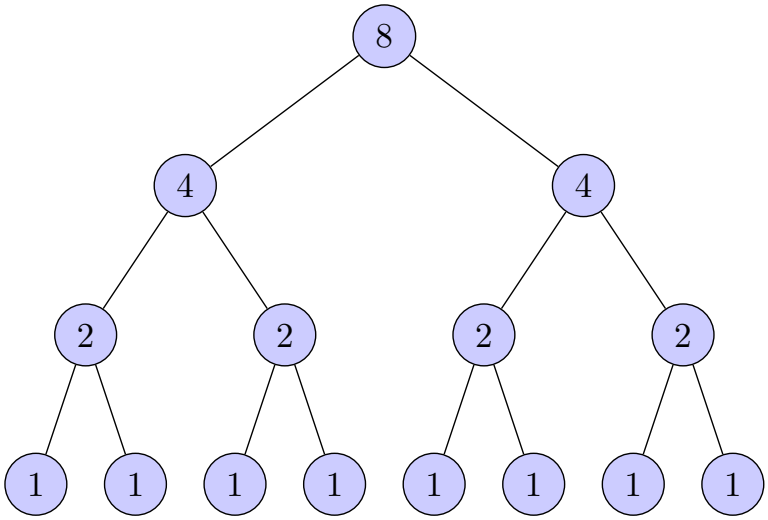
We saw in lecture that the recursive call tree for `mergesort` is a binary tree like the following:



In this exercise, you'll complete a running-time analysis for mergesort.

1. Suppose we call `mergesort` on a list of length **8**. Draw the corresponding recursion diagram, and inside each node write down the non-recursive running time of the call, which we count as equal to the *size of the input list* for that call.

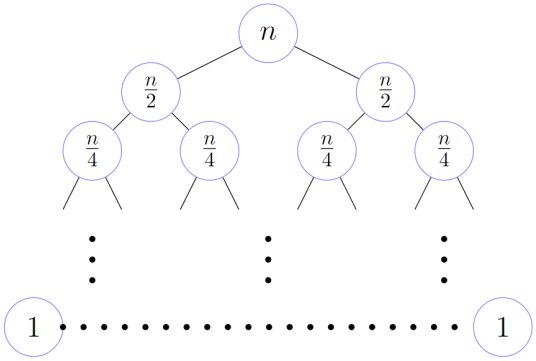
   (For example, the root of the tree should contain an "8", and its two children should be "4"s.)



2. Compute the total of the numbers in your above diagram. This gives you the total running time for `mergesort` on a list of length 8.

   The total of the numbres in the above diagram is $8 \cdot 1 + 4 \cdot 2 + 2 \cdot 4 + 1 \cdot 8 - 32$.

3. Now suppose we have a list of length n, where n is a power of 2. Fill in the recursion diagram below with the corresponding non-recursive running times. The root node should be filled in with n.



4. Compute the total of the numbers in your above diagram, again assuming n is a power of 2.

   Hint: consider the sum of the numbers in each *level* of the tree.

   The total of the numbers in the above diagram is n multiplied by the height of the tree which is $\log n$ therefore, it is $n \log n$.

2 Exercise 2: Quicksort running time and uneven partitions

Now consider the quicksort algorithm.

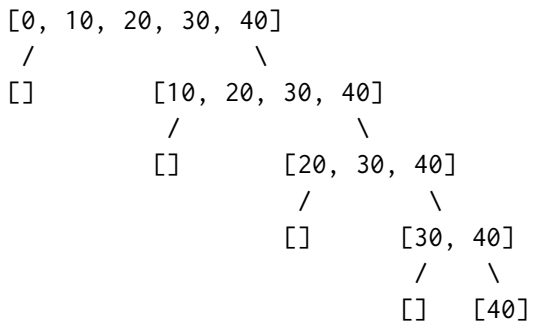
```
1  def quicksort(lst: list) -> list:
2      if len(lst) < 2:
3          return lst.copy()
4      else:
5          pivot = lst[0]
6          smaller, bigger = _partition(lst[1:], pivot)
7
8          smaller_sorted = quicksort(smaller)
9          bigger_sorted = quicksort(bigger)
10
11         return smaller_sorted + [pivot] + bigger_sorted
```

Its recursive step also makes two recursive calls, but unlike mergesort, the input list lengths are not necessarily always the same size.

1. Suppose we call `quicksort([0, 10, 20, 30, 40])`. After the `_partition` call, what are `smaller` and `bigger`?

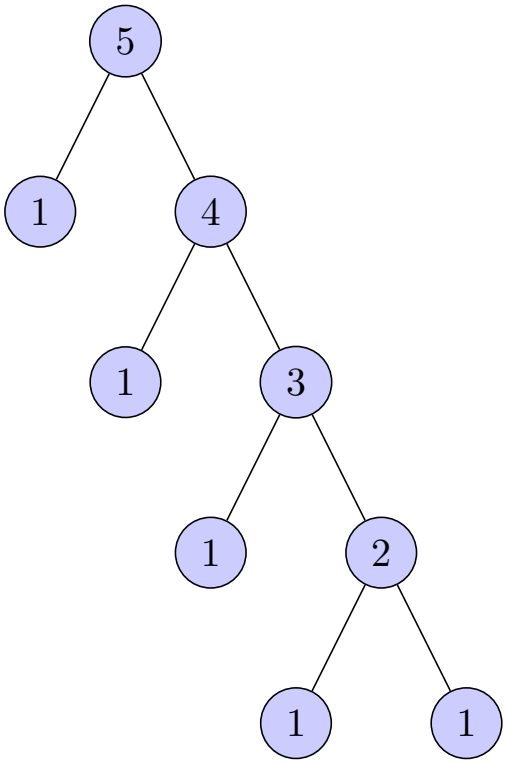
   `smaller = [], bigger = [10, 20, 30, 40]`

2. Draw a recursion tree showing the *inputs* to each recursive call, when we call `quicksort([0, 10, 20, 30, 40])`. We've started the first two levels for you below. The node containing represents a recursive call on an empty list.



3. Now redraw the recursion tree, but with the *non-recursive running time* in each node.

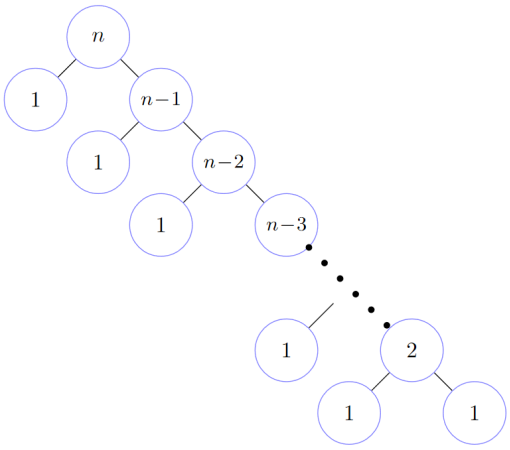
   - For an empty list, the non-recursive running time is 1.
   - For a non-empty list, the non-recursive running time is the *length of the list*.



4. Add up all of the numbers in the above diagram.

   $5 + 1 + 4 + 1 + 3 + 1 + 2 + 1 + 1 = 19$

5. Generalize the above calculation for a call to quicksort with a list of length n, when the chosen pivot is always the *smallest* element in the list.



   This leads to a running time of $\Theta(n^2)$

3 Additional exercises

1. Here is the implementation of the helper function `_partition` used by quicksort.

```python
def _partition(lst: list, pivot: Any) -> tuple[list, list]:
    smaller = []
    bigger = []

    for item in lst:
        if item <= pivot:
            smaller.append(item)
        else:
            bigger.append(item)

    return smaller, bigger
```

(a) Analyse the running time of `_partition` in terms of n, the length of its input list.

(b) How would your analysis change if each item were inserted at the *front* of the relevant list instead, for example: `smaller.insert(0, item)`?

2. Analyse the running time of your `_in_place_partition` implementation from last class.

CSC111 Lecture 22: Average-Case Running Time Analysis

Hisbaan Noorani

March 31, 2021

Contents

Our worksheet exercises today will focus on analysing the average-case running time for the following implementation of the *linear search* algorithm:

```
def search(lst: list, x: Any) -> bool:
    """Return whether x is in lst."""
    for item in lst:
        if item == x:
            return True
    return False
```

1 Exercise 1: Counting binary lists

In lecture, we began an average-case running time analysis for `search` on binary lists, i.e., lists where each element is either 0 or 1. Let $n \in \mathbb{N}$. We define the set of inputs $\mathcal{I}_n$ to be the set of binary lists of length n.

1. Suppose $n = 3$. Write down $\mathcal{I}_3$, i.e., the set of binary lists of length 3. You should have **eight** lists in total.

   `{[0, 0, 0,], [0, 0, 1], [0, 1, 0], [0, 1, 1], [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]}`

2. Now suppose n is an arbitrary natural number. Find an expression (in terms of n) for $|\mathcal{I}_n|$, the size of $\mathcal{I}_n$. (You don't need to formally prove that this expression is correct, but try to briefly justify this expression for yourself.)

   2^n. Here we have $n = 3$, and we were able to generate 8 lists. We cannot have done more than this.

$$\underbrace{0/1 \quad 0/1 \quad 0/1 \quad \ldots \quad 0/1}_{n}$$

2 Exercise 2: Partitioning $\mathcal{I}_n$

Here is the definition of the partitions of $\mathcal{I}_n$ from lecture.

- For each $n \in \mathbb{N}$ and each $i \in \{0, 1, \ldots, n-1\}$, let $S_{n,i}$ denote the set of all binary lists of length n where the first `0` occurs in index i. More precisely, every list `lst` in $S_{n,i}$ satisfies the following two properties:

  1. `lst[i] =` 0=
  2. For all $j \in \{0, 1, \ldots, i-1\}$, `lst[j] =` 1=

- For each $n \in \mathbb{N}$, let $S_{n,n}$ denote the set of binary lists of length n that do not contain a `0` at all.

1. To make sure you understand the definitions of these partitions, write down the corresponding set of binary lists for each of the following.

   (a) $S_{3,0}$
   `{[0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1]}`

   (b) $S_{3,1}$
   `{[1, 0, 0], [1, 0, 1]}`

   (c) $S_{3,2}$
   `{[1, 1, 0]}`

   (d) $S_{3,3}$
   `{[1, 1, 1]}`

2. Now find expressions for each of the following. Once again, make sure you can briefly justify your answers, but no need for formal proofs.

   (a) $|S_{n,n}|$
   $\underbrace{1 \quad 1 \quad 1 \quad \ldots \quad 1}_{n} \implies 1$

   (b) $|S_{n,i}|$, where $i \in \{0, 1, \ldots, n-1\}$ (your expression should be in terms of i and n)
   $\underbrace{1 \quad 1 \quad 1 \quad \ldots \quad 1 \quad 0}_{i+1} \quad \ldots \quad \underbrace{0/1 \quad \ldots \quad 0/1}_{n-i-1} \implies 2^{-i-1}$

3. Recall that the purpose of this partitioning is that all input lists `lst` in the same partition have the same running time for `search(lst, 0)`:

   - Every input in $S_{n,i}$ (for $i \in \{0, 1, \ldots, n-1\}$) takes $i + 1$ steps
   - Every in $S_{n,n}$ takes $n + 1$ steps (technically this is compatible with the previous bullet point, setting $i = n$)

Using this and your answers to the previous question, simplify the expression below. *Tips:*

Use the formula $\sum_{i=0}^{m-1} i \cdot r^i = \frac{m \cdot r^m}{r-1} - \frac{r(r^m-1)}{(r-1)^2}$, valid for all $m \in \mathbb{Z}^+$ and $r \in \mathbb{R}$ where $r \neq 1$.

b. Use the change of variable $i' = i + 1$ to help see how to use the above formula.

Expression to simplify:

$$\sum_{i=0}^{n} |S_{n,i}| \times \big(\text{running time of } \texttt{search(lst, 0)} \text{ when } \texttt{lst} \in S_{n,i}\big)$$

Let $m = n + 1$, $r = \frac{1}{2}$

$$\begin{aligned}
&= \sum_{i=0}^{n} |S_{n,i}| \times (i+1) \\
&= \sum_{i=0}^{n-1} \left[2^{n-i-1} \times (i+1)\right] + 1 \times (n+1) \\
&= \sum_{i'=1}^{n} \left[2^{n-i'} \times i'\right] + (n+1) \\
&= 2^n \left(\sum_{i'=1}^{n} \left(\frac{1}{2}\right)^{i'} \times i'\right) + (n+1) \\
&= 2^n \left[\sum_{i'=1}^{n} \frac{(n+1)\left(\frac{1}{2}\right)^{n+1}}{\frac{1}{2}-1} - \frac{\frac{1}{2}\left(\left(\frac{1}{2}\right)^n + 1\right) - 1}{\left(\frac{1}{2}-1\right)^2}\right] + (n+1) \\
&= \dots \\
&= 2^{n+1} - 1
\end{aligned}$$

This last expression gives you the total running time of `search` over all inputs in $\mathcal{I}_n$! We'll continue lecture by using this to calculate the average-case running time of `search` for this input set, but if you still have time feel free to work it out yourself.

3 Additional exercises

1. Generalize our average-case running time analysis for `search` for the input set consisting of all lists of length n where every element is a 0, 1, or 2, and `x = 0`.

2. Let $m \in \mathbb{Z}^+$. Generalize our average-case running time analysis for `search` for the input set consisting of all lists of length n where every element is a number between 0 and $m - 1$, inclusive, and `x = 0`.

3. Analyse the average-case running time of `search` when the input set is the list of all **permutations** of the numbers $\{0, 1, \dots, n-1\}$ and `x = 0`.

4. Consider the following algorithm for checking whether a string is a palindrome:

```python
def is_palindrome(s: str) -> bool:
    """Return whether s is a palindrome."""
    mid = len(s) // 2
    for i in range(0, mid):
        if s[i] != s[len(s) - 1 - i]:
            return False

    return True
```

Analyse the average-case running time of `is_palindrome` on the input set $\mathcal{I}_n$ consisting of all *binary strings* of length n (i.e., strings that contain only the characters `'0'` and `'1'`).