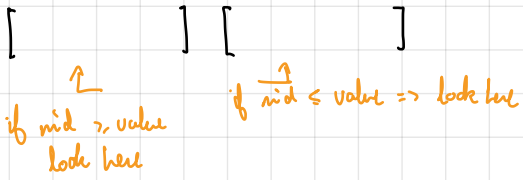
Lab 1

- Binary search → sorted list
  → no duplicates (will work)

what is value in the middle



$O(\log_2 n)$ time

cutting down search space by $\frac{1}{2}$ every time

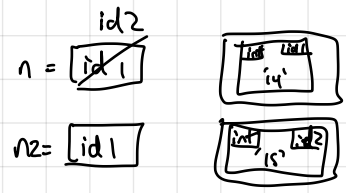
$$i_k = 2^k$$

$$2^k \geq n$$

Extra practice

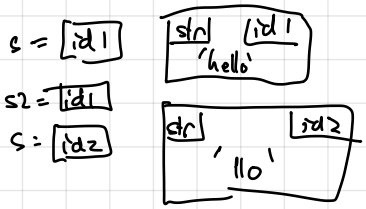
```
n = 14
n2 = n
n = 15
```

15, 14



Q2)

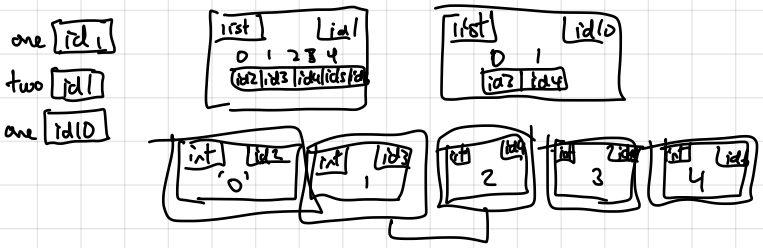
```
s = 'hello'
s2 = s
s = s[2:]
print (s, s2)
```



'llo', 'hello'

Q3)

```
one = [0, 1, 2, 3, 4]
two = one
one = one[1:3]
print (one, two)
```



[1, 2], [0, 1, 2, 3, 4]

Q4)

```
a = [1, 2, 3, 2, 9]
b = [1, 2, 3, 2, 9]
a.remove(2)
print(a, b)
```

[1, 2, 2, 9], [1, 2, 3, 2, 9]

Q5)

```
x = [1, 2, 3]
y = x
y[1] = 100
print (x, y)
```

y = x — mutate the ref to id1

[1, 100, 3], [1, 2, 3]

Q6)

```
lst = [3, 2, 7]
lst2 = lst
lst.append(99)
```

[3, 2, 7, 99], [3, 2, 7, 99]

Q7)

```
temp = [5, 10, 15]
other = 75
L = [temp, 'hey', other]
    [[5, 10, 15], 'hey', 75]
temp[1] = 99
other = 0
```

[[5, 99, 15], 'hey', other]

Q8) L = [[1, 2] . [3, 4]]
for item in L
item = 88
print
[88, 887

Q9) L = [[1, 2], [3, 4]]
for item in L

not possible since loop runs forever using methods

Q10) L = ['we', 'you', 'us']
for item in L :
item[0] = item[0].upper()

TypeError str doesn't support item assignment

Q11) L = [[1], [2], [3]]
for element in L:
element . append(8)
print(L)
[[1, 8], [2, 8], [3, 8]]

Choosing test cases

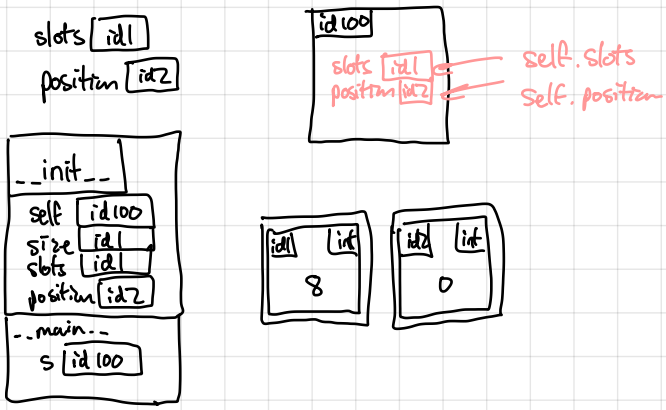
1. position of n1 in lst	3	front, back, somewhere else
n1 in lst?	2	T, F
length of lst	3	0, 1, 5
n1 == n2	2	T, F
n1 adj to n2	2	T, F
lst has only n1 in it	2	T, F

$\times$ [144]

W2 Worksheets

self refers to object

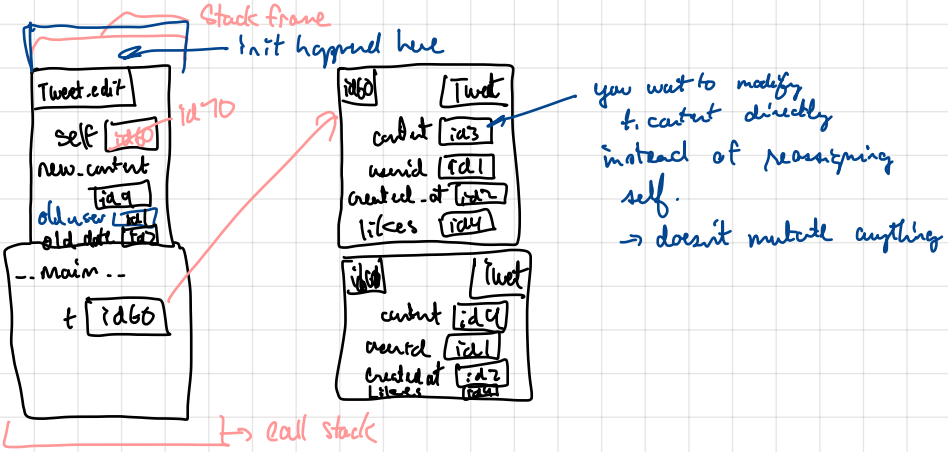
1) slots id1
position id2



ii) creates variables "slots" and "position"

iii) Attribute Error: spinner has no attribute position

3.




```
def edit(self, new_content: str) -> None:
    self.content = new_content
```

when Tweet was created, __init__ is add to call stack

Composition of Classes

```
def __init__(self, userid: str, bio: str) -> None:
    self.userid = userid
    self.bio = bio
    self.tweets = []

def tweet(self, message: str) -> None:

    t = Tweet(self.userid, date.today(), message)
    self.tweets.append(t)
```

3.
```
s = User("Diane", "bio")
s.follows("Lex")
```

4.)
```
def follows(self, ~~userid: str~~ Other: User) -> None:
    """ Record that this User follows <other>

    """

    self.follows.append(other)
```

5.
```
class User:
    follows: List[User]
```

Object Oriented Design Considerations

1. misbehaved = Tweet("wwa", 01-22-25, "ullala")
   misbehaved.like(-1) ← cannot have (-ve) like
   misbehaved.edit('toolong' * 100) ← content too long

   empty user id "" is invalid

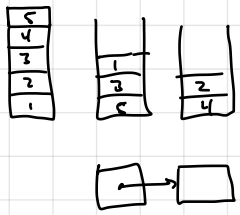
2) len(content) ≤ 280
   likes ≥ 0
   user_id is not ""
   created_at ≤ date.today() ← created_at in the future

   like precondition that n ≥ 0

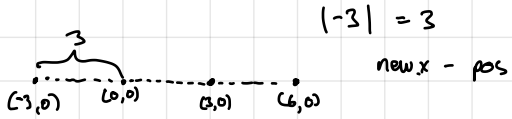
   edit
   self.content = new content[:280]
   user_id precondition that len(user_id) > 0

4.

Test 1



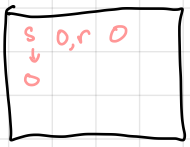
W3 SDM



$|-3| = 3$

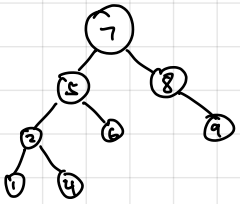
new.x - pos

$|x - pos|$



@

item = 3



$1 = \gcd(a,b) \wedge a \mid bc \Rightarrow a \mid c$

$bc = ak_1$ $\qquad c = a(\dots)$

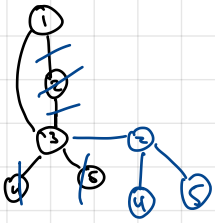
$pa + qb = \gcd(a,b) = 1$

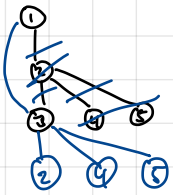
$pa + qb - bc = \gcd(a,b) - ak_1$

$pa + b(q-c) =$

Let $a, b, c \in \mathbb{Z}$

A2 swap_up





W4 Balancer

(← False

(] ← False

() ← True

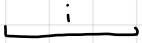
W4 Running Time Efficiency

1.

a. "Constant-time" → 1 step

b. list of length n

c. n-(i+1) items must be moved



d. (n-1)-(i+1) = n-i-2 items

i. lst.insert(0)

ii lst.pop()

e. Queue. enqueue

2. Stack 1 has push/pop 1 step
Stack 2 has push/pop n+1 steps

a. Stack 1 : 2 Stack 2: n+1, n+1+1 → 2n+3

b. Stack 1 : 5 Stack 2: 0+1, 1+1, 2+1, 3+1, 4+1 } 1+2+3+4+5 = 15

c. Stack 1: k Stack 2: $\frac{(k+1)k}{2}$

d. Stack 1 : n+n = 2n Stack 2: push: 0 1 2 ... (n-1)

4n

$\frac{n(n+1)}{2}$ steps

$\Rightarrow n(n+1) = n^2+n$

push: $\frac{n(n+1)}{2}$

pop: n n-1 n-2 ... 2 1
↓ ↓ ↓ ↓ ↓
1 2 3 n-2 n-1
n+1 n n-1 ... 3 2 +1

$$\sum_{i=1}^{n} i + n+1 - 1$$

$$= \left(\frac{n(n+1)}{2} + n\right) 2$$

$$= n(n+1) + 2n + n^2 + n$$

$$= n^2 + n + 2n + n^2 + n$$

$$= 2n^2 + 4n$$

$$\Rightarrow O(n^2)$$

W4 Stack size

1. def size(s: Stack) → int:
    count = 0
    for _ in s:
        count += 1
    return count

No → Stack is not iterable

2. def size(s: Stack) → int:
    count = 0
    while not s.is_empty():
        s.pop()
        count += 1
    return count

No, just gets rid of all items in s.
w/o replacing

3. def size(s: Stack) → int:
    return len(s._items)

~~Stack has no attribute len~~ ← accesses private instance variable

4. def size(s: Stack) → int:
    s_copy = s ← alias
    count = 0
    while not s_copy.is_empty():
        s_copy.pop()
        count += 1
    return count

Solution:

```
def size(s: Stack) -> int:
    temp = Stack()
    count = 0
    # Pop everything off <s> and onto <temp>, counting as we go.
    while not s.is_empty():
        temp.push(s.pop())
        count += 1

    # Now pop everything off <temp> and back into <s>
    while not temp.is_empty():
        s.push(temp.pop())

    # s is now restored to its state at the start of the function call
    # We consider that it was not mutated.

    return count
```

Alt solution

```
def size_v2(s: Stack) -> int:
    temp = []
    count = 0

    # pop everything off <s> and onto <temp>, counting as we go
    while not s.is_empty():
        temp.append(s.pop())
        count += 1

    # now take everything from <temp> back into <s>
    temp.reverse()
    for elem in temp:
        s.push(elem)

    # ALT to above, use list.pop
    # while len(temp) > 0:
    #     s.push(temp.pop())

    # <s> is now restored to its state at the start of the function call
    # We consider that it was not mutated.

    return count
```

W4 Stack_V2

1. push → insert at 0
   pop → remove at 0

2. the beginning of the list represents the top of this stack

3. Nothing would change

4. End if Python list
   beginning if linked list

WS Linked-list Efficiency

1. Suppose we have a Python (array-based) list of length n.

   a.

elmnt at front	O(n)
elmnt at end	O(1)
look up elm at i	O(n-i) → O(1)

   b. O(n-i)

2. Linked list

   a

elmnt at front	O(1)
elmnt at end	O(n)
look up @ index i	O(i)

   b. O(i)

3. Roughly the same
   - → traversing from the front LL → $\frac{1}{2}$ → O(n-i)
   - → traversing from the back PL → $\frac{1}{2}$ → O(i)

4a.) How many times is the node traversed?

$1 + 2 + 3 + 4 + \ldots + n$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$O(n^2)$

5. lst.__contains__(148) returns after a single node.

   '148' is in the self.__first node

   b.) 148 is at the end of the list.

WS Traversal

lst.__getitem__(i) ~ lst[i]

1. curr1 = self._first
   curr2 = other._first

2a. curr1 is None or curr2 is None

b. while curr1 is not None and curr2 is not None
   If curr1.item != curr2.item
      return False
   curr1 = curr1.next
   curr2 = curr2.next

a) curr1 is None or curr2 is None

b.) same length when curr1 is None and curr2 is None.

c) return curr1 is None and curr2 is None

↑ They may not be equal so we check this.

__getitem__

a) how to init curr and i
   curr = self._first
   i = 0

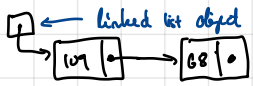
b) stopping: i == index or curr is None
   while i != index and curr is not None

c)
   curr = curr.next
   i = i + 1

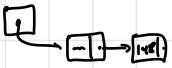
assert i == index or curr is None

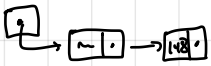
if curr is None:
   raise IndexError
else:
   return curr.item

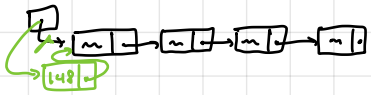
WS duration

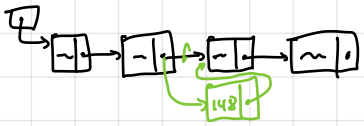
1a) 

b) len(self) == 0, index == 0 

len(self) == 1, index == 1 

len(self) == 1, index == 1 

len(self) == 4, index == 0 

len(self) == 4, index == 2 

len(self) == 4, index == 4 

2. len = 0, index 0
   1 1
   4 0

b) same value

c) index 1 node's next attribute

3.)
```
def insert(self, index: int, item: Any) -> None:
    new_node = _Node(item)
    if index == 0:
        self._first, new_node.next = new_node, self._first

    curr = self._first
    i = 0
    while curr is not None and i != index - 1
        curr = curr.next
        i = i + 1
    assert curr is None or i == index - 1

    if curr is None:
        raise IndexError
    else:
        curr.next, new_node.next = new_node, curr.next
```

W7 Nested List Mutation

```
def add_one(obj: Union[int, List]) -> None:
```

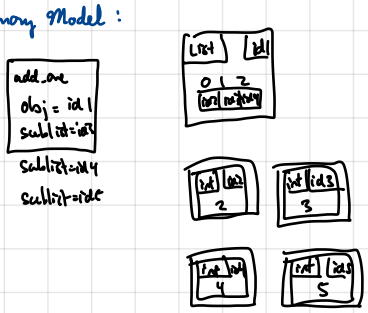
1. Base Case

```
if isinstance(obj, int):
    pass
```

2. Recursive step

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

Memory Model:



3. Loop over the indices of the list rather than elements directly → mutate elements of a list.

```
for i in range(len(obj)):
```

```
if isinstance(obj, int):
    pass
else:
    for i in range(len(obj)):  # know obj[i] is either an int or a list.
        if isinstance(obj, int):
            obj[i] += 1  # using index this way mutates list
        else:  # obj[i] is another nested list
            add_one(obj[i])
```

W7 Recursive Patterns

```
def f(obj: Union[int, list]) ->
```

1.
```
def nested_list_contains(obj: Union[int, List], item: int) -> bool:
    """Return whether the given item appears in <obj>
    Note that if <obj> is an integer, this function checks whether
    <item> is equal to <obj>.
    >>> obj = [1,2,3]
    >>> item = 1
    >>> nested_list_contains(obj, item)
    True
    >>> obj1 = [[[1]]]
    >>> item = 1
    >>> nested_list_contains(obj, item)
    True
    """
    if isinstance(obj, int):
        return obj == item
    else:
        for sublist in obj:
            if nested_list_contains(sublist, item):
                return True
        return False  # know: we didn't find it, even after looking through every sublist.
```

2.
```
def first_at_depth(obj: Union[int, list], d: int) -> Optional[int]:
    """Return the first (leftmost) item in <obj> at depth <d>.
    Return None if there is no item at depth <d>.
    Precondition: d >= 0
    >>> obj = 1
    >>> d = 0
    >>> first_at_depth(obj, d)
    1
    >>> obj = [[1]]
    >>> d = 2
    >>> first_at_depth(obj, d)
    1
    """
    if isinstance(obj, int):
        if d == 0:
            return obj
        else:
            return None
    else:  # know: obj is a list
        if d == 0:  # there is nothing at depth 0 in a list!
            return None
        d1 = d - 1
        for lst in obj:
            new = first_at_depth(lst, d1)
            if new:  # is not None   # if its not none, we know
                return new           # it's an int cause of the type contract
        # if we reach here, we know: no sublist of obj contained an int at
        # depth-1. Therefore, obj contains no int at depth d!
        return None
```

W7 Tracing Recursion

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

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

b)

sublist	flatten(sublist)	Value of s at the end of the iteration
N/A	N/A	[] (initial value of s)
[0,-1]	[0,-1]	[0,-1,-2,[[-3,[-5],-7]]]
-2	[-2]	[0,-1,-2,[[-3,[-5],-7]]]
[[-3,[-5],-7]]	[3,-5,-7]	[0,-1,-2,-3,-5,-7]

c)

d) yes

e) return [obj]

2. [1,[2,3],[4,[5]]] # Issue: we didn't check if item in already in the list!

[1,2,3,4,5]

Partial Trace

sublist	uniques(sublist)	Value of s at the end of the iteration
N/A	N/A	[]
1	[1]	[1]
[2,3]	[2,3]	[1,2,3]
[4,[5]]	[4,5]	[1,2,3,4,5]

Output w/ error: [[1,13],13,[13,13]] → expected value: [1,13]

[1,13]	[1,13]	[1,13]
13	[13]	[1,13,13]
[13,13]	[13]	[1,13,13,13]
	↳ why not [13,13]	

W8 Tree Deletion

1. We can't just set the _root attribute to None. Why not?
   _root → None will get rid of the entire tree. → violate RIs.

2. "Promoting a subtree"

Idea: take rightmost subtree t, make the root of t, the new root of the full tree. make the subtrees of t, become subtrees of the full tree.

Implementation:

```
def _delete_root(self) -> None:
    s = self._subtrees.pop()      # Remove rightmost subtree
    self._root = s.root           # Update self._root
    self._subtrees += s._subtrees # Update self._subtrees
```

3. "Replacing root w/ leaf"

Idea: find leftmost subtree, make leaf value the new root value.

Implementation:

```
def _delete_root(self) -> None:
    leaf = self._extract_leaf()
    self._root = leaf

def _extract_leaf(self) -> Any:
    if not self._subtrees:
        old_root = self._root
        self._root = None
        return old_root
    else:
        leaf = self._subtrees[0]._extract_leaf()

        if self._subtrees[0].is_empty():
            self._subtrees.pop(0)
        return leaf
```

Deletion ideas:

Scenario	Recursive calls	Code
Empty Tree		return False
delete item 13 in multi-node tree (Not @ root)	delete_item(12, 13)	iterate through subtrees, stop if one call returns True.
	12 ⇒ no mutation return False	↳ also return True
	13 ⇒ item removed return True	if none do
	stop recursing when one call returns True.	↳ return False

delete @ root

① single node

subtrees = []

② multiple nodes

```
if self._root == item
    self._root = None
    return True
else:
    return False
```

deletion helper

W8 Tree methods

1. Consider Method 'leaves'] no mutation → just return value

(a) Case : description / tree	(b) Recursive Calls : tree / return value	(c) Return value
- empty tree	- no recursive calls	- []
- tree of one item	- no recursive calls	- [9]
- bigger tree	6 ⇒ [6]	
	4 ⇒ [4, 3, 1]	- [6, 3, 1, 9, 5] give together all the results from the kids
	2 ⇒ [2, 9, 10, 5]	

implementation:

Use extend / append ?

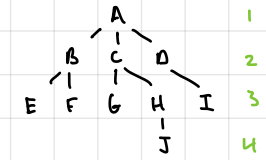
```
def leaves(self) -> list:
    if self.is_empty():                  # ← empty tree
        return []
    elif self._subtrees == []:           # ← tree of one item
        return [self._root]
    else:
        result = []                      # ← bigger trees
        for subtree in self._subtrees:   # ← does not consider root.
            result.extend(subtree.leaves())
        return result
```

2. 'average'

(a) Case : description / tree	(b) Recursive Calls : tree / return value	(c) Return value
- empty tree	- no recursive calls	- 0.0
- tree of one item	- item / 1	- item
- bigger tree	- 6 (3, 1) ⇒ 6, 1 ⇒ 9, 2 ⇒ 10, 3	- $\frac{6+3+1}{1+1+1} \Rightarrow \frac{10}{3}$

W8 Trees Nested lists

1.



1 2 3 3 2 3 3 4 2 3

['A', ['B', ['E'], ['F']], ['C', ['G'], ['H', ['J']]], ['D', ['I']]]

implementation:

```
def to_nested_list(self) -> List:
    if self.is_empty():
        return []
    elif self._subtrees.is_empty():
        return [self._root]
    else:
        s = [self._root]
        for subtree in self._subtrees:
            s.append(subtree.to_nested_list())
        return s
```

2. Tree.to_nested_list

- empty tree	no recursive calls	[]
- tree w/ one item	no recursive calls	[A]
- tree	A ⇒ [A] ⇒ [B, [E], [F]] ⇒ [C, [G], [H, [J]]] ⇒ [D, [I]]	[A]

3. Representation of tree ⇒ Representation in list

⇍

[[A]]

4. Tree.to_tree

- empty list	no recursive calls	[] empty tree
- list w/ int/string	no recursive calls	Tree('A')
- bigger list	[A, [B], [C]] ⇒ root = A ⇒ sublist.root = B sublist.root = C	

```
def to_tree(t: Union[int, list]) -> Optional[Tree]:
    if isinstance(t, int):
        return None
    elif t == []:
        return Tree(None, []) # empty Tree
    else:
        rl = t[0]
        if isinstance(rl, list):
            return None
        s = t[1:]
        st = []
        for sublist in s:
            n = to_tree(sublist)
            if n is None:
                return None
            else:
                st.append(n)
        return Tree(rl, st)
```

W9 BST deletion helper

Case 1: self is a leaf

```
def delete_root(self) -> None:
    if not (self._right or self._left):
        self._root = None
        self._right = None
        self._left = None
```

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

```
    '''
    elif not(self._right):
        self._root = self._left._root
        self._right = self._left._right
        self._left = self._left._left
    elif not(self._left):
        self._root = self._right._root
        self._right = self._right._right
        self._left = self._right._left
```

Case 3: both subtrees are non-empty

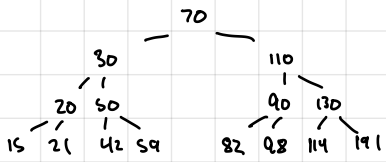
- Look for the smallest value in the right subtree to become the root.
- min from right / max from left

```
    self._root = self._left.extract_max()
```

```
def extract_max(self) -> Any:
    ''' Remove and return the max value in this tree :''
    if self._is_empty():  <- Precondition
        return None
    if self._right._is_empty():
        biggest = self._root
        self._root, self._left, self._right = self._left._root, self._left._left, self._left._right
        return biggest
    else:
        return self._right.extract_max()
```

W9 BST deletion

1.



What value is easy to delete? 15?

2. deleting 70
   - only 59 and 82 can go there

3. inorder traversal

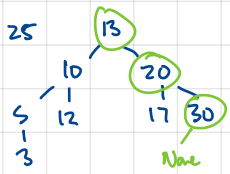
   15, 20, 21, 30, 42, 50, 59, 70, 82, 90, 98, 110, 114, 130, 191

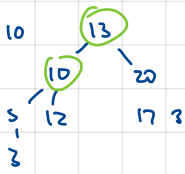
4. delete 13 from a BST

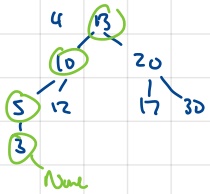
case: description / tree	Recursive calls	final outcome
- empty tree	no recursive calls	
- 13 is root	no recursive calls	
- 13 in right	delete (right)	
- 13 in left	delete (left)	

W9 BST efficiency

1.
```
def __contains__(self, item: Any) -> bool:
    if self._is_empty():
        return False
    elif item == self._root:
        return True
    elif item < self._root:
        return self._left.__contains__(item)
    else:
        return self._right.__contains__(item)
```

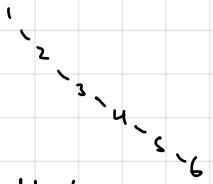






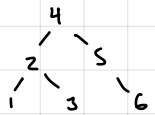
2. insertion algorithm

a) BST empty (1,2,3,4,5,6)



height = 6

b) BST empty(4,2,5,1,3,6)



height = 3

4. let $n \in \mathbb{N}$

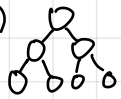
n height → worst possible height

5a BST of height 0 — $h = 1$

empty BST

b) ① node at most — $h = 1$

c)  3 nodes at most — $h = 2$

d)  7 nodes — $h = 3$

$i_0 = 0$, $i_1 = 1$, $i_2 = 3$, $i_3 = 7$

$i_k = 2^k - 1$ (closed) $\quad i_k = i_{k-1} + 2^{k-1}$ (open)

e) $2^{148} - 1$

f) $n \leq 2^h - 1$

n: total nodes; $2^h - 1$: max # of nodes

g) $n - 1 \leq 2^h$

$\log_2(n-1) \leq h$

min height of a tree w/ n nodes

W10 List Comprehensions

```
result = []
for x in lst:
    result.append(f(x))  # where f is some helper
```

~ `result = [f(x) for x in lst]`

1.
```
def sum_nested(obj: Union[int, list]) -> int:
    if isinstance(obj, int):
        return obj
    else:
        return sum([sum_nested(elem) for elem in obj])
```

```
s = sum(lst)
~ s = 0
  for x in lst:
      s += x
```

2.
```
s = init
for x in lst:  # x not necessarily a number!
    s += x

s = sum(lst, init)
```

```
# sum takes an initial value
  sometimes not a number
```

```
def flatten(obj: Union[int, list]) -> list[int]:
    if isinstance(obj, int):
        return [obj]
    else:
        flattened = [flatten(sublist) for sublist in obj]
        return sum(flattened, [])
```

3.
```
def nested_list_contains(obj: Union[int, list], item: int) -> bool:
    if isinstance(obj, int):
        return obj == item
    else:
        return any([nested_list_contains(s, item) for s in obj])
```

any := lst[0] or lst[1] or ... or lst[n-1]
all := lst[0] and lst[1] and ... and lst[n-1]

4.
```
def semi_homogeneous(obj: Union[int, list]) -> bool:
    """Return whether the given nested list is semi-homogeneous.
    A single integer and empty list are semi-homogeneous.   <- Base case
    In general, a list is semi-homogeneous <=>
        - all of its sub-nested lists are integers or all of them are lists   <- return
        - all of its sub-nested lists are semi-homogeneous
    """
    if isinstance(obj, int) or obj == []:
        return True
    else:
        all_ints = all([isinstance(elem, int) for elem in obj])
        all_lists = all([isinstance(elem, list) for elem in obj])
        all_semis = all([semi_homogeneous(elem) for elem in obj])
        return (all_ints or all_lists) and all_semis
```

WII ch_place quicksort

1. def _in_place_partition(lst : List[int]) → None:

a) small_i = 1
big_i = len(lst)

b) small_i == big_i → while small_i < big_i

c)

Swap lst[small_i] w/ lst[big_i - 1]

small_i += 1 big_i -= 1

2. def _in_place_partition(lst : List[int]) → None:

```
pivot = lst[0]
small_i = 1
big_i = len(lst)

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

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

3. return new index of pivot return (small_i - 1)

take in 2 add. params. start/end so
that the function only partitions the range lst[start : end]

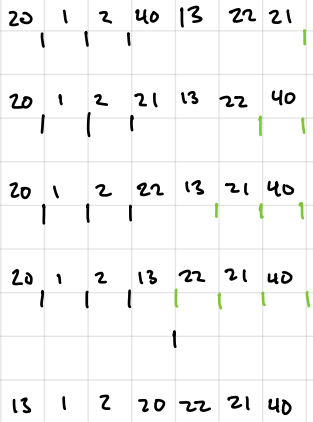
```
def _in_place_partition(lst : List[int], start : int, end : int) → int
    pivot = lst[start]
    small_i = start + 1          # make sure range (start : end)
    big_i = end                    is in the list.
```

idea: divide up lst into 3 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 example:

| := small_i iter val
| := big_i iter val



W11 sorting efficiency

1.
```
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
```

What is Big-Oh RT?

$\in O(n)$ since RT of loop is $O(n) + 1 + 1$ basic operations

2. How would answer change if each item were inserted at the front of the relevant list

`smaller.insert(0, item)`

$O(n^2)$ → since `insert(0, item)` runs $O(n)$

3.
```
def _merge(lst1: List, lst2: List) -> List:
    index1, index2 = 0, 0
    merged = []
    while index1 < len(lst1) and index2 < len(lst2):
        if lst1[index1] <= lst2[index2]:
            merged.append(lst1[index1])
            index += 1
        else:
            merged.append(lst2[index2])
            index2 += 1
    return merged + lst1[index1:] + lst2[index2:]
```

$k \geq n_1$ or $k \geq n_2$

$n_1 = len(lst1)$

$n_2 = len(lst2)$

max loop iterations for merge in terms of n_1 / n_2

max loop iter for n_1 ↓ max loop iter for n_2

$n_1 + n_2 \rightarrow O(n_1 + n_2) \rightarrow O(n)$

4. running time that depends on n_1 / n_2

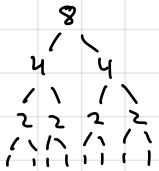
`lst[index1:]` will take `n1 - index1` steps

`lst[index2:]` will take `n2 - index2` steps

5. def mergesort(lst: List) → List:
    if len(lst) < 2:
        return lst[:]
    else:
        mid = len(lst) // 2
        left_sorted = mergesort(lst[:mid])
        right_sorted = mergesort(lst[mid:])

        return _merge(left_sorted, right_sorted)

a. len = 4 per list
b. len = 2 per list
c. len = 1 per list
d. input list length 8 4 2 1
   # mergesort calls on lists of that length 1 2 4 8
   ↳ Big Oh of mergesort



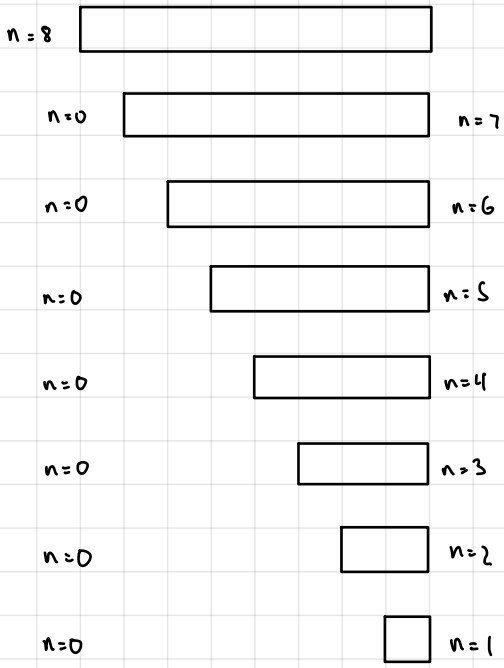
mergesort runs O(n log n)

6. def quicksort(lst: List) → List:
    if len(lst) < 2:
        return lst[:]
    else:
        pivot = lst[0]
        smaller, bigger = _partition(lst[1:], pivot)
        return quicksort(smaller) + pivot + quicksort(bigger)

a. len(smaller) = 0
   len(bigger) = 7

b. input list length
   # quicksort calls on lists of that length
   8 7 6 5 4 3 2 1 0
   1 1 1 1 1 1 1 1 1

O(n)



Mock Exam

>>> solve(20,2)

a = 20 / 2 = 10

= 2 · 10

= 20

>>> solve(100,50)

ValueError

>>> solve('science',1)

'Something is wrong'

>>> try:

'Name not defined outside'

print(solve(s,1) · 2)

10

2. in-order

→ read from left to right

b. post order

→ 3,((5,5 +), 2 *),) +

adding in terms of depth

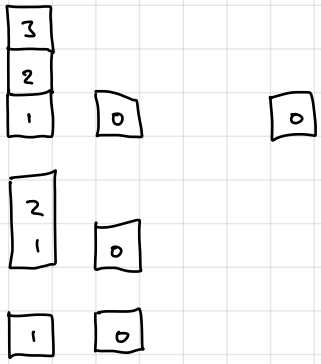
3.

```
if self._root == low:
    self._left = None
if self._root > low:
    self._left.prune
if self._root < low:
    self._right.prune
    self._root = self._right._root
    self._right = self._right._right
    self._left = self._right._left
```

4a. def insert_at_bottom(s: Stack, item: Any) → None:

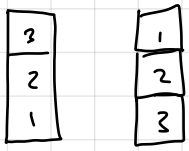
```
if s.is_empty():
    s.push(item)
else:
    temp = s.pop()
    s.insert_at_bottom(s, item)
    s.push(temp)
```

Read the instructions carefully and determine the methods that do/don't satisfy them to determine what you need to use.



4b. def mystery(s: stack) → None:

"""Pops every item of <s> and restores them in the reverse order it was deleted."""



don't assume anyone does, finish the execution all the way!

← think deeper / recursively

← respect the R.I.s.

Winter 2019

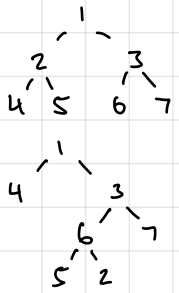
Q1) Attribute error → no attribute next
Attribute error → no attribute first
True
True

b) Attribute error, None type has no attribute root ← RI empty BinarySearchTree
↳ None
False
True → Empty BST → False
None type has no attribute root. → False

2. 100
$\lfloor \log_2 100 \rfloor + 1$

b. __init__
__getItem__ → seen by Alice

c.



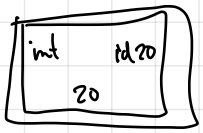
d One node trees have this
or trees w/ same values

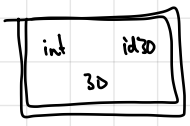
e) ——

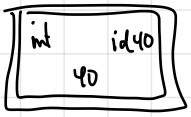
Q3) ~~None-type objects~~ → Empty BSTs.

a) __main__
t1 = id1
t2 = id2
t3 = id3
t4 = id4

they are probs greater than
SD → root.right
they don't exist → empty BST







c) def test_extract_max_fail() -> None: ← self is reassigned

"""Test that extract_max doesn't mutate self."""

bst = BinarySearchTree(3)

bst.left = BinarySearchTree(2)

max = bst.extract_max()

assert bst._root == 3

def test_extract_max_pass() -> None:

"""Test extract_max that it returns the max."""

bst = BinarySearchTree(3)

bst.left = BinarySearchTree(2)

max = bst.extract_max()

assert max == 3

Q4) class Instructor:

"""An instructor.

=== Attributes ===

_certifications: this instructor's certifications

_name: this instructor's name

"""

Attribute types

_certifications: List[str]

_name: str

def __init__(name: str) -> None:

self._name = name

self._certifications = []

def add_certification(c: str) -> None:

self._certifications.append(c)

class Session(Event)

"""a session.

=== Attributes ===

_name: the name of this session

_level: the level of this session

_date: the date of this session

_instructor: the instructor of this session

"""

_name: str

_level: int

_date: str

_instructor: Instructor

def __init__(name: str, level: int, date: str, instructor: Instructor)

if instructor.certification != name

raise Uncertified Instructor Error

else:

self._instructor = Instructor

Event.__init__(name, date)

self._level = level

Cont. class Instructor:

def get_name(self) -> str:

return self._name

def get_certs(self) -> List[str]:

return self.certifications

Cont. class Session

def get_schedule() -> str:

return f'Level {self._level} session, with {self._instructor}. every {self._date}'

class Event:

"""An event

=== Attributes ===

_name: the name of the event

_level: the level of the event

"""

Attribute types

_name: str

_date: Any

def __init__(name: str, date:) -> None:

self._name = name

self._date = date

def get_schedule() -> str:

raise Not Implemented Error

class Workshop(Event):

"""a workshop.

=== Attributes ===

_name: the name of this workshop

_date: the date of this workshop

_instructor: the instructor of this workshop.

"""

_name: str

_date: date

_instructor: Instructor

def __init__(name: str, level: int, date: date, instructor: Instructor) -> None:

Event.__init__(name, date)

if not instructor.certification:

raise Uncertified Instructor Error

else

self._instructor = instructor

def get_schedule() -> str:

return f'{self._name} workshop, with {self._instructor}: {self._date}'

```
class UnidentifiedCharacterError(Exception):
    """An unidentified character error."""

    pass
```

Q5)

a) [7, 10, 8, 4, 3, 1]
[13, 19]

b) 1, 128 0, 127 no pivot.

$\frac{128}{2} = 64$

$\frac{128}{2} = 64$

c) merge sort

d) yes,
an unsorted list a sorted list
makes quicksort slower

Q6)

a) $O(k + k^2) \rightarrow O(k^2)$

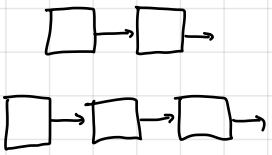
b) $$\sum_{k=1}^{n} k + k^2 = \sum_{k=1}^{n} k + \sum_{k=1}^{n} k^2$$

$$= \frac{n(n-1)}{2} + \frac{n(n-1)(2n-1)}{6}$$

$$= n^3$$

Q7) e. `last = other._first`

```
i    if pos == 0:
f        last.next = self._first
b        self._first = other._first
a    else:
c        i = 0
h        curr = self._first
j        while curr is not None and i < pos - 1:
k            curr = curr.next
d            i += 1
m        while last.next is not None:
l            last = last.next

n        last.next = curr.next
g        curr.next = other._first
```



Q8)

```
def __init__(self, k, items: list) -> None:
    self._k = k
    self._refs = []
    self._first = _Node(items[0])
    curr = self._first
    for i in range(1, len(items)):
        new_node = _Node(items[i])
        curr.next = new_node
        curr = curr.next
        if i % k == 0:
            self._refs.append(new_node)
```

C) `__init__` : define the class' attributes and reference to superclass

insert/delete : polymorphic code → changes what the method does override

b)

n//100 elems ✓ 1000/100 = 10

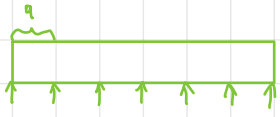
100 nodes 99

n//10 elems

10 nodes 9

n elems → advantage that RT ∈ O(1)

1 node disadvantage that space ∈ O(n)



Q9)

$\sum_{i=1}^{n} i \in O(n^2)$

$O(n)$

$n + n \in O(n)$

Q10)

```
def valid_expanded(self) -> bool:
    if self.is_empty():
        return True
    if not self._subtrees:
        return self._expanded == False
    else:
        if self.expanded:
            return any(subtree._expanded for subtree in self._subtrees)
        else:
            return all(subtree._expanded == False for subtree in self._subtrees)
```

Q11)

```
def consistent_depth(obj: Union[int, list]) -> bool:
    if isinstance(obj, int):
        return True
    else:
        c, depth = helper(obj)
        if depth:
            return c
```

```
def helper(depth: int) -> Tuple[bool, int]:
    if isinstance(obj, int):
        return True, 0
    else:
        depths = []
        depth = 1
        for elem in obj:
            c, d = helper(elem)
            depth = depth + d
            depths.append(depth)

        return all(d for d in depths), depth
```

Q(12) a) t = Tree(1, [7])

>>> 1 in t

True

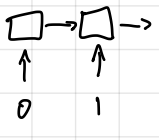
b) fail

December 2017

1a) since it returns None,
even if it mutates number/branch,
it doesn't return that mutation.

1b) that is an int
self.thing is an int.

1c) curr is None or curr > index



>>> c = LinkedList([0,1])
>>> c[2] == IndexError
curr has no object. item

Not the right error.

2. 38, 51, 40, 46, 11, 25, 29, 15, 5

b) composition := combining two functions
such that the output of
one function is the input of
another.

c) abstract class: a class that may be used
to implement more specific subclasses
of that same type so methods
are polymorphic / inherited.

d) '_' before the attribute name

restricts access by other classes/functions.

3. Class Design

classes : Student → name, num

Undergrad → name, num

- str

Grad → name, num, supervisors

- str
- add_committee_member
- committee_size

- complete_course()
- num_credits()
- average()

```
class Student:
    """ A student
    == Attributes ==
    - name : This student's name
    - num : This student's number
    - total : The total score of this student in all subjects
    - courses : The courses this student has taken.
    """
    name : str
    num : int
    total : int
    courses : list[str]
    def __init__(name : str, num : int) -> None:
        self.name = name
        self.num = num
        self.total = 0
        self.courses = []

    def complete_course(s:str, i : int) -> None:
        """ Records that this student completed course <s>
        with mark <i>. """
    def num_credits() -> int :
        """ Returns the number of credits this student earned. """
    def average() -> int
        """ Returns this student's average
    def __str__() -> str
        """ An abstract method. Returns str representation of
        this student """
        raise NotImplementedError
```

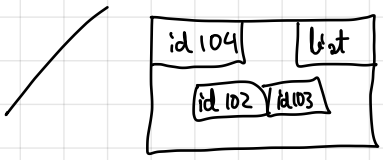
```
class Grad(Student):
    """ A grad student.
    == Attributes ==
    - name : this grad's name
    - num : this grad's number
    - courses : this grad's courses
    - total : this grad's total score
    - members : this grad's members
    """
    name : str
    num : int
    total : int
    courses : list[str]
    members : list[str]
    def __init__(name : str, num : int, member : str) -> None:
        """ Initialize this grad student's attributes """
    def __str__() -> None
        """ Return a
```

5. roughly the same
in a python list we traverse $\frac{1}{2}$ of the list
in a linked list we traverse $\frac{1}{2}$ of the list

b) $O(\log_2 n)$
each list is split into two
for loop iterates proportional to $\frac{1}{2^k}$
so we get that the number of iterations
is $= \log_2 n$

c) copying a list acts like a sorted list
- there are better algos for sorting a sorted list.

5)



m2 id 104

b. yes
== only compares types
no
is only compares mem addresses
c. yes
index slicing modifies the list directly
no
m1 is affected is well

d) O(n) the for loop only iterates over
items in the matrix not the sublists.

6) >>> str(linky.bisect(2))
'[3 -> 4]'

2) >>> linky2 = LinkedList([1])
>>> str(linky2.bisect(1))
IndexError

curr.next is None or index > i

curr is not None and index < i

```
def bisect(self, i: int) -> LinkedList:
    curr = self._first
    index = 0

    while curr.next is not None and index - 1 < i:
        curr = curr.next
        index += 1
    temp = curr.next
    curr.next = None
    items = []
    while temp is not None:
        items.append(curr.item)
    new_list = LinkedList(items)
    return new_list
```

7. def colour_valid(self) -> bool:
    if self.children:
        return len(self.children)==4 and self.colour == None
    elif not self.children:
        return self.colour is not None

8.) Attribute error

self._left must be defined as an empty tree
    -right

b) log n

at each point we are making 2^h recursive calls

so
$2^{h-1}=n$
$h=\log_2 n + 1$

c) def is_bst(self) -> bool:
    if self.is_empty():
        return True
    elif self._left.is_empty() and self._right.is_empty():
        return True
    else:
        left = self._left.is_bst()
        right = self._right.is_bst()
        return left and right and (self._root)

9.)

0	0
1	0
4	0
9	6

1: doesn't consider the empty tree
and returns 1

2: depth is >= given depth
so the recursive call should do +1.
results in depths >= returning +1.

10. yes since 1235 - 1231 = 4
which is a buyable option.

```
def buyable(n: int) -> bool:
    if n == 0:
        return True
    elif n == 4 or n == 25 or n == 6:
        return True:
    else:
        if buyable(n-6):
            return True
        if buyable(n-4):
            return True
        if buyable(n-25):
            return True
```

December 2016

1. a) list

b) int

mutable → ability to change values stored in memory

immutable → cannot do this

b) depth first search of adjacent

breadth first search

c) first → find takes linear time as each node is visited once.

d) abstract class

- a class that represents a framework for objects which share similar features

e) inheritance - subclass has access to parents methods / attributes

composition - subclasses contain other classes within it

inheritance is useful when there are two objects with similar functions

class sink and class shower both perform wash hands.

composition is useful when we require properties of objects for our class to work.

2.) $O(m+n)$ $O(n^2)$ $O(a^2)$ $O(b)$ $O(n)$ $O(na^2)$

3. t = None ← an object that is None doesn't have attribute x defined.

t.x

b. def hi(x: int) → int ← recursion is infinite or exceeds the available space.

hi(x)

hi(x)

c. class x: ← method is not defined.

def what():

raise Not implemented error

x = X()

x.what

4. No, No, Yes, No

b. 10
10 ← attribute error

5. Classes: spreadsheet
cell

```
class Cell:
    """ a cell.
    === Attributes ===
    _x: the x coord of this cell
    _y: the y coord of this cell
    """
    _x: int
    _y: int
    def __init__(x: int, y: int) -> None:
        """Initialize this cells attributes."""
```

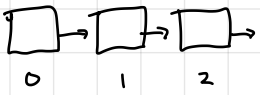
5. `s = Spreadsheet(cols=["Year", "Sales", "Expenses"],`

```
s.record(coords=(0,1), 2015)
s.record((0,2), 2016)
s.record((1,1), 125000)
s.record((1,2), 150000)
s.record((2,1), 100000)
s.record((2,2), 100000)
s.average("Sales")
```

d.) _cols contains list of strings
if _cols are none so are rows.
if _cols are not none, rows are
init below index >= 1.

```
class Spreadsheet

    def __init__(cols: List[str],
        """Initialize this spreadsheets attributes"""

    def average(col: str) -> int
        """Return the average of the total sum in this <col>"""

    def record(t: tuple[int,int], s: str) -> None:
        """Record this <s> in cell <t>."""

    _cols: the columns of this spreadsheet
    _rows: the rows of this spreadsheet
    # Attribute types
    _cols: List[str]
    _rows: List[List[int]]
```

6. last = other._first
while last.next is not None:
    last = last.next
curr = self._first
index = 0
while curr.next is not None and 1 < index:
    curr = curr.next
last.next = curr.next
curr.next = other._first



6) O(m+n)

7)(A) current is None or current.item < previous.item

(B) current is not None but current.item < previous.item
previous is not None and points to current
all nodes before point to previous

(C) current is none ~~but either or~~ current.item < previous.item
current.item >= previous.item
since current is none
previous is the last node in this LL.

"""Return the last item in this list
if this item is less than all others
else return None.
>>> s = LinkedListL([5,4,3,2,1])
s.mystery()
1

c. No, since in the case curr is not None
curr.item < prev.item
and in the case curr is None
curr.item doesn't exist so the ambiguity is ruled out.

d. O(1): len(self) < 2.

e O(n): has to iterate and compare

8. def distribution(self):
    if self.is_empty:
        return {}
    else:
        d = {self._root: 1}
        ld = self._left.distribution()
        rd = self._right.distribution()

        return absorb(d, absorb(ld, rd))

9. def nested_sum(obj):

→ nested_sum([4, [1,2,3], [10, [20]], 4])

0

changes: sum = 0 → line 4
sum += nested_sum(lst[i]) → line 6