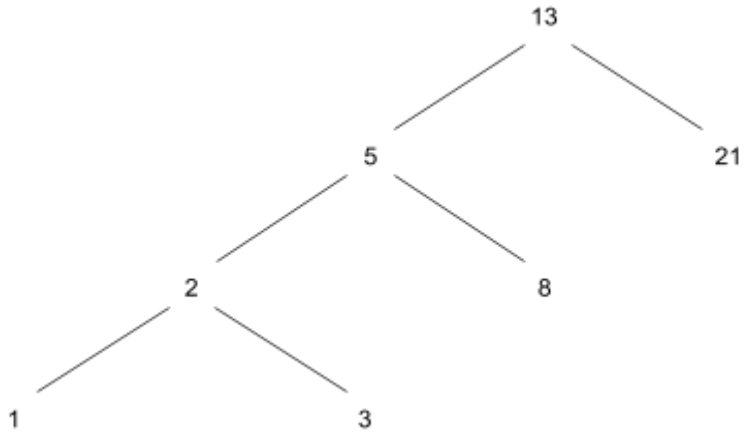
CS 111: Introduction to Computer Science

MIDTERM 2 Review Key

(3 points) **Recursive Objects: Tree**

A *Binary Search Tree* is a sorted tree data structure where each Tree node only has two branches, which are called *left* and *right*. Each Tree node's **left** branch contains zero or more Tree nodes with labels **smaller** than its label. Each Tree node's **right** branch contains zero or more Tree nodes with labels **larger** than its label.
Here is an example Binary Search Tree:



The code below defines a class called `TreeNode` that is a recursive object and has a `left` and `right` branch.

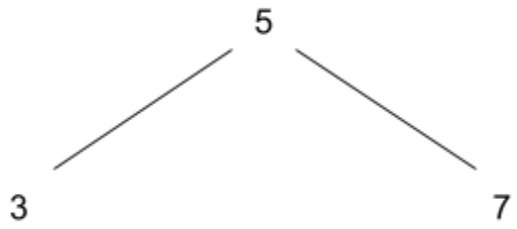
```
class TreeNode:
    """A binary tree."""
    empty = ()

    def __init__(self, label, left=empty, right=empty):
        self.label = label
        self.left = left
        self.right = right

    def is_leaf(self):
        """
        >>> t = TreeNode(1)
        >>> t.is_leaf()
        True
        >>> t = TreeNode(5, TreeNode(3), TreeNode(7))
        >>> t.is_leaf()
        False
        """
        return not self.left and not self.right
```

We want to create a function for a `TreeNode` sorted as a Binary Search Tree to return a list of the labels of the tree in sorted order – smallest to greatest. For example, the example tree (above) should return the list [1, 2, 3, 5, 8, 13, 21].

Here is the code with missing parts.

```
def inorder(curNode):
    """ Argument curNode must be a sorted binary tree
    >>> t = TreeNode(5, TreeNode(3), TreeNode(7))
    >>> inorder(t)
    [3, 5, 7]
    """
    labels = []
    if curNode != TreeNode.empty:
        labels += inorder(___(a)___)
        labels.append(___(b)___)
        labels += inorder(___(c)___)
    return labels
```

Note that the doc string above is for this tree:



(1) (1 pt) What line of code could go in blank (a)?

A. curNode.left
B. curNode.label
C. curNode.right

(2) (1 pt) What line of code could go in blank (b)?

A. curNode.left
B. curNode.label
C. curNode.right

(3) (1 pt) What line of code could go in blank (c)?

A. curNode.left
B. curNode.label
C. curNode.right

(7 points) **Regular Expressions**

(4) (2 pts) Given the pattern “[def]+QY?[4-6]”, which of the following would be **completely** matched? **Select the *two* that are correct**.

A. `"def456"`
B. `"dQ5"`
C. `"dey4"`
D. `"defQY6"`
E. `"xyzdQY4"`
F. `"efY6"`
G. `"DQY4"`

(5) (3 pts) Which of the following patterns would match “def” and “ddeff” but not “ef”? **Select the *three* that are correct**.

A. `"d+ef+"`
B. `"[def]+"`
C. `"d?[ef]"`
D. `"d[def]+"`
E. `".*ef.*"`
F. `".?def.?"`

(6) (2 pts) Which of the following patterns would **completely** match “Matthew”, “Mark”, and “John” but not **completely match** “Luke”? **Select the *two* that are correct**.

A. `"(matthew|mark|john)"`
B. `"([A-K]|[M-Z])\w*"`
C. `"[MJ]?\w+"`
D. `"((Ma|Jo)[trh].*)?"`
E. `"\w[aeiou]\w+"`
F. `"\w+"`

(6 points) **Searching a Sorted List**

Given a sorted list of `n` integers, how do we determine if some target value is in the list? We could iterate and search through the entire list until we find the target or exhaust the search.

(7) (Extra Credit - 1 pt) Given the class discussion about Efficiency, what would the run-time bound be for the search described above?

A. Quadratic, $O(n^2)$
B. Linear, $O(n)$
C. Logarithmic, $O(\log_2 n)$
D. Constant, $O(1)$

A binary search is performed by using a Divide and Conquer algorithm. Divide and Conquer algorithms divide the list into two smaller lists and then recursively operate on the sublists. With binary search, the algorithm divides the list into two sublists, where one sublist has smaller numbers and the second sublist has larger numbers. Binary search then discards the sublist that is too big or too small for the target value being searched. Binary search only continues to search the sublist that might actually contain the target value. This decision to discard a sublist is made with just one comparison.

For example, given the following sorted list: [1, 4, 6, 8, 9, 10, 11, 12], there are eight elements in sorted order. If we are trying to search this list for the value 10, then Binary Search will first split the list into two sublists: [1, 4, 6, 8] and [9, 10, 11, 12]. The target value, 10, can only be found in the numbers >= 9, so we discard the first sublist and recursively search on the second sublist.

(8) (Extra Credit - 1 pt) Given the class discussion about Efficiency, what would the run-time bound be for the described binary search?

A. Quadratic, $O(n^2)$
B. Linear, $O(n)$
C. Logarithmic, $O(\log_2 n)$
D. Constant, $O(1)$

The code on the next page is a *recursive* implementation of Binary Search (with some lines missing). Determine what should fill in the blanks.

```
# Recursive implementation of the binary search algorithm to return
# the position (index) of `target` in sub-list sorted_list[left…high_index]
def binarySearch(sorted_list, low_index, high_index, target):

    # Base condition (search space is exhausted)
    if low_index > high_index:
        return -1

    # find the middle value in the search space and compares it with the target
    mid_index = (low_index + high_index) // 2

    # Base condition (a target is found)
    if target == sorted_list[mid_index]:
        return  (a)

    # discard all elements in the high_index search space, including the middle
element
    elif target < sorted_list[mid_index]:
        return binarySearch(    (b)    )

    # discard all elements in the low_index search space, including the middle
element
    else:
        return binarySearch(    (c)    )
```

(9) (1 pt) What line of code could go in blank (a)?

A. `sorted_list`
B. `mid_index`
C. `low_index`
D. `high_index`
E. `sorted_list[mid_index]`
F. `sorted_list[low_index]`
G. `sorted_list[high_index]`

(10) (1 pt) What line of code could go in blank (b)?

A. `sorted_list, low_index, mid_index - 1, target`
B. `sorted_list, mid_index + 1, high_index, target`
C. `sorted_list, low_index, high_index, target`
D. `sorted_list, high_index, low_index, target`

(11) (1 pt) What line of code could go in blank (c)?

A. `sorted_list, low_index, mid_index - 1, target`
B. `sorted_list, mid_index + 1, high_index, target`
C. `sorted_list, low_index, high_index, target`
D. `sorted_list, high_index, low_index, target`

Recall from our class discussion that all recursive algorithms can also be written iteratively (and vice versa). The code on this page is an *iterative* implementation of Binary Search (with some lines missing).

```
# Function to determine if a `target` exists in the sorted list `sorted_list`
# or not using a binary search algorithm. Returns index or -1 if not found.
def binarySearch(sorted_list, target):
    # search space is sorted_list[low_index…high_index]
    (low_index, high_index) = (0, len(sorted_list) - 1)
    # loop till the search space is exhausted
    while low_index <= high_index:
        # find the middle value in the search space and compares it with the
target
        mid_index = (low_index + high_index) // 2
        # target is found
        if target == sorted_list[mid_index]:
            return __(d)__
        # discard elements in the high_index search space, including the middle
element
        elif target < sorted_list[mid_index]:
            high_index = __(e)__
        # discard elements in the lower search space, including the middle
element
        else:
            low_index = __(f)__
     # `target` doesn't exist in the list
    return -1
```

(12) (1 pt) What line of code could go in blank (d)?

A. `mid_index / 2`
B. `mid_index`
C. `mid_index * 2`

(13) (1 pt) What line of code could go in blank (e)?

A. `mid_index`
B. `mid_index - 1`
C. `mid_index + 1`
D. `mid_index * 2`

(14) (1 pt) What line of code could go in blank (f)?

A. `mid_index`
B. `mid_index + 1`
C. `mid_index - 1`
D. `mid_index * 2`

(5 points) **Generators for Recursive Objects: Link**

The code below defines a class called `Link` that is a recursive object. Note that the member variable that points to the next `Link` is called `next`, instead of `rest` as was shown in the lecture slides (using the name "next" is common practice with linked lists). The code uses `label` instead of `first` as shown in the slides.

```
class Link:
    """A linked list."""
    empty = ()

    def __init__(self, label, next=empty):
        assert next is Link.empty or isinstance(next, Link)
        self.label = label
        self.next = next
```

The function below takes an argument of type `Link` and creates a generator that will give the `label` of each `Link` in the order of the linked list.

```
def inorder(linked_list):
    """
    >>> ll = Link(1, Link(2, Link(3, Link(5, Link(8, Link(13))))))
    >>> gen = inorder(ll)
    >>> next(gen)
    1
    >>> next(gen)  # Second call
    2
    >>> next(gen)  # Third call
    3
    """
    lnk = linked_list
    while __(a)__:
      yield __(b)__
      ____(c)____
```

(15) (1 pt) What line of code could go in blank (a)?

A. `lnk is Link.empty`
B. `lnk is not Link.empty`
C. `lnk.next is not Link.empty`
D. `lnk.next is Link.empty`

(16) (2 pts) What line of code could go in blank (b)?

A. `linked_list`
B. `lnk`
C. `lnk.label`
D. `lnk.next`

(17) (2 pts) What line of code could go in blank (c)?

A. `lnk = lnk.value`
B. `lnk = lnk.next`
C. `lnk = linked_list`
D. `lnk = linked_list.next`
E. `lnk = linked_list.value`

Extra credit (2 pts): Suppose we want to create a generator to return the reverse order of the list. This can be performed using recursion. Complete the following code for extra credit.

```
def reverse(linked_list):
    """
    >>> ll = Link(1, Link(2, Link(3, Link(5, Link(8, Link(13))))))
    >>> gen = reverse(ll)
    >>> next(gen)
    13
    >>> next(gen)  # Second call
    8
    >>> next(gen)  # Third call
    5
    """
    if linked_list is not Link.empty:
      yield from _____(d)
      yield _____(e)
```

(18) (extra credit - 1 pts) What line of code could go in blank (d)?

A. `reverse(linked_list.value)`
B. `reverse(linked_list.next)`
C. `reverse(linked_list)`

(19) (extra credit - 1 pts) What line of code could go in blank (e)?

A. `linked_list.next`
B. `linked_list`
C. `linked_list.label`

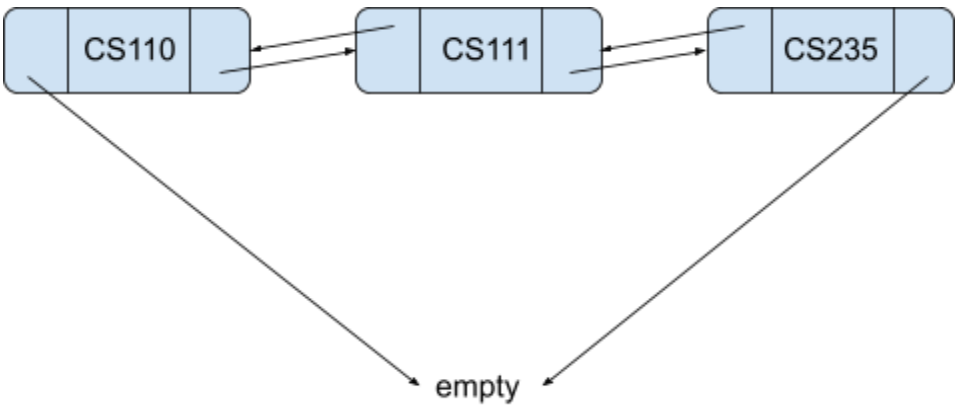
(7 points) **WWPD Recursive Objects: Doubly Linked List**

A *doubly* linked list is a list where each List node has a reference to both the next List node and the previous List node, which are stored as `next` and `prev` respectively in the object. See the code below and answer the questions.

```
class DoubleLink:
    """A doubly linked list."""
    empty = ()

    def __init__(self, label, next=empty, prev=empty):
        assert next is DoubleLink.empty or isinstance(next, DoubleLink)
        assert prev is DoubleLink.empty or isinstance(prev, DoubleLink)
        self.label = label
        self.next = next
        if next is not DoubleLink.empty:
          next.prev = self
        self.prev = prev
        if prev is not DoubleLink.empty:
          prev.next = self
```

Suppose we run the following code to create a `DoubleLink` list (graphically shown below)

```
>>> n5 = DoubleLink("CS235")
>>> n3 = DoubleLink("CS110")
>>> n4 = DoubleLink("CS111", n5, n3)
>>> start = n3
```



(20) (1 pt) What would Python display? `>>> start.label`

A. CS110
B. CS111
C. CS235

(21) (1 pt) What would Python display? `>>> start.next.label`

A. CS110
B. CS111
C. CS235

(22) (1 pt) What would Python display? `>>> start.next.next.label`

A. CS110
B. CS111
C. CS235

(23) (2 pts) What would Python display? `>>> start.next.next.prev.label`

A. CS110
B. CS111
C. CS235

(24) (2 pts) What would Python display? `>>> start.next.next.prev.prev.label`

A. CS110
B. CS111
C. CS235

(2 points) **Testing**

Consider the following simple class.

```
class Grid:
    def __init__(self, width, height):
        assert width > 0 and height > 0
        self.height = height
        self.width = width

    def in_bounds(self, x, y):
        if x < 0 or x >= self.width:
            return False
        if y < 0 or y >= self.height:
            return False
        return True
```

(25) (1 pt) Which of the following docstrings would **NOT** be a valid doctest for the in_bounds function?

A.
```
>>> b = Grid(4,13)
>>> b.in_bounds(0,0)
True
```

B.
```
>>> b = Grid(3,4)
>>> b.in_bounds(3,4)
True
```

C.
```
>>> b = Grid(2,2)
>>> b.in_bounds(5,1)
False
```

D.
```
>>> b = Grid(10,30)
>>> b.in_bounds(-2,0)
False
```

E.
```
>>> b = Grid(20, 20)
>>> b.in_bounds(19,19)
True
```

(26) (1 pt) Which of the following would be a good pytest function to verify that the in_bounds function properly responds to a negative y value?

A.
```
def test_negative_y_false():
    b = Grid(20, 20)
    assert b.in_bounds(-1,5) == False
```

B.
```
def test_negative_y_false():
    b = Grid(20, 20)
    assert b.in_bounds(20,-5) == False
```

C.
```
def test_negative_y_false():
    b = Grid(20, 20)
    assert b.in_bounds(0,0) == True
```

D.
```
def test_negative_y_false():
    b = Grid(20, 20)
    assert b.in_bounds(-1,-5) == False
```

E.
```
def test_negative_y_false():
    b = Grid(20, 20)
    assert b.in_bounds(17,-1) == False
```

(27) (Extra Credit - 1 pt) What is the minimum number of tests we would need to write, assuming each test exercises one condition, to test that in_bounds() responds properly to the possible input combination for x & y, both valid and invalid?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6
G. 7
H. 8

(10 points) **Language Parsing - Postfix Notation**

In the calculator project, you parsed the input expressions to create a syntax tree, and then evaluated that tree to find the value of the expression that was provided. However, once we have the syntax tree, we can do more than just evaluate the expression. For example, we could use that syntax tree to rewrite the expression into another language or syntax. For example if we had a syntax tree of a python program, it would be possible to rewrite that program in C++ or Java just from the syntax tree (assuming there were translations for everything).

In this question, we are going to take our calculator expressions, which were written in what is known as prefix notation, where the operator precedes the operands, and convert them to postfix notation, where the operands precede the operator. Some examples:

If we had the following calculator expression for 1 + 2:
(+ 1 2)
It would be written in postfix notation as
1 2 +

The expression for (3 * 6) - (1 + 2):
(- (* 3 6) (+ 1 2))
Would be written as
3 6 * 1 2 + -

Notice that the postfix notation **doesn't require any parentheses**. When evaluating, each operator simply uses the two operands before it and replaces operands and operator with the value of the operation at its position in the expression as it is evaluated. Thus that last expression would evaluate like this:

3 6 * 1 2 + -
18 1 2 + -
18 3 -
15

(28) (1 pt) What would the postfix notation be for the expression (* 3 2)

A. `3 * 2`
B. `(3 2 *)`
C. `2 3 *`
D. `* 2 3`
E. `3 2 *`

(29) (1 pt) What would the postfix notation be for the expression (+ 2 (* 7 5))

A. `2 + (7 * 5)`
B. `(5 * 7) + 2`
C. `((7 5 *) 2 +)`
D. `2 7 5 * +`
E. `2 + (7 5 *)`

Let's build a function that can traverse our calculator syntax tree and write out the expression in postfix notation. Remember that our syntax tree is composed of Pair objects where *first* is either an operator or a Pair object containing a sub expression and *rest* is either the next Pair object in the sub-expression or nil which represents the end of that expression. Thus our calculator expression (+ (* 3 2) 1) would be represented as:

```
Pair('+', Pair(Pair('*', Pair(3, Pair(2, nil))), Pair(1, nil)))
```

and would be written in postfix notation as:

```
3 2 * 1 +
```

Here's the framework for the code to write this out

```
def postfix(syntax_tree):
    """
    syntax_tree is a calculator syntax tree represented by Pair objects.
This
    function returns a string representing the calculator expression in
    postfix notation.  E.g. the calculator expression (+ (* 3 2) 1) would
    be represented by the string '3 2 * 1 +'.
    >>> postfix(Pair('+',Pair(Pair('*',Pair(3,Pair(2 nil))),Pair(1,nil))))
    '3 2 * 1 +'
    """
    if syntax_tree is nil:
        _____(a)_____
    if isinstance(syntax_tree, (int, float)):
        _____(b)_____
    if isinstance(syntax_tree, Pair):
        if _____(c)_____:
            return postfix(__(d)__) + postfix(__(e)__) + f"{__(f)__} "
        else:
            return postfix(syntax_tree.first)
```

Use the syntax tree for (+ 1 (* 3 2)) (given in the docstring) to help you figure out the code that should fill in the blanks and answer the following questions:

(30) (2 pts) What is/are the base case(s) for this recursive function? Select all that apply

A. syntax_tree is a Pair object
B. syntax_tree is the nil object
C. syntax_tree is one of the operators (+, -, *, /)
D. syntax_tree is an integer or floating point number

(31) (1 pt) What should be returned in the case where t is nil? What code should go in line (a)?

A. `return syntax_tree`
B. `return ""`
C. `return 0`
D. `return None`

(32) (1 pt) What should be returned in the case where t is a floating point or integer number? What code should go in line (b)?

A. `return 0`
B. `return none`
C. `return syntax_tree`
D. `return f"{syntax_tree}"`
E. `return f"{syntax_tree} "`

(33) (1 pt) How do we check to see if the item in the current pair is one of our operators? What code should go in line (c)?

A. `syntax_tree in ['+', '-', '*', '/']`
B. `syntax_tree in [+, -, *, /]`
C. `syntax_tree.first in ['+', '-', '*', '/']`
D. `syntax_tree.first in [+, -, *, /]`

If the current item is one of our operators, we need to return the first operand, the second operand, and then the operator. The following questions ask you to fill in the items for lines (d), (e), * (f) to make that happen.

(34) (1 pt) What value should go in line (d)?

A. `0`
B. `None`
C. `nil`
D. `syntax_tree`
E. `syntax_tree.first`
F. `syntax_tree.rest`
G. `syntax_tree.rest.rest`

(35) (1 pt) What value should go in line (e)?

A. `0`
B. `None`
C. `nil`
D. `syntax_tree`
E. `syntax_tree.first`
F. `syntax_tree.rest`
G. `syntax_tree.rest.rest`

(36) (1 pt) What value should go in line (f)?

A. `0`
B. `nil`
C. `None`
D. `syntax_tree`
E. `syntax_tree.first`
F. `syntax_tree.rest`
G. `syntax_tree.rest.rest`