

---

# CS 61A

## Spring 2017

# Structure and Interpretation of Computer Programs

TEST 2 (REVISED)

---

### INSTRUCTIONS

- You have 2 hours to complete the exam.
- The exam is open book, open notes, closed computer, closed calculator.
- Mark your answers **on the exam itself**. We will *not* grade answers written on scratch paper.

|                                                                                                                      |  |
|----------------------------------------------------------------------------------------------------------------------|--|
| Last name                                                                                                            |  |
| First name                                                                                                           |  |
| Student ID number                                                                                                    |  |
| CalCentral email (@berkeley.edu)                                                                                     |  |
| TA                                                                                                                   |  |
| Name of the person to your left                                                                                      |  |
| Name of the person to your right                                                                                     |  |
| Room in which you are taking exam                                                                                    |  |
| Seat number in the exam room                                                                                         |  |
| <i>I pledge my honor that during this examination I have neither given nor received assistance.</i><br>(please sign) |  |

## Reference Material.

### # Linked Lists

```
class Link:
    """A linked list cell.
    >>> L = Link(0, Link(1))
    >>> L.first
    0
    >>> L.rest
    Link(1)
    >>> L.first = 2
    >>> L
    Link(2, Link(1))
    >>> L.rest = Link.empty
    >>> L
    Link(2)
    """
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest is Link.empty:
            return "Link({})".format(self.first)
        else:
            return "Link({}, {})".format(self.first, self.rest)
```

### # Trees

```
class Tree:
    """A tree node."""

    def __init__(self, label, branches=[]):
        for c in branches:
            assert isinstance(c, Tree)
        self.label = label
        self.branches = branches

    def is_leaf(self):
        return not self.branches
```

**1. (12 points) Pointers**

In the following problems, single boxes are variables that contain pointers, and double boxes are **Links** (see the definition of **Link** on page ??). To show that a box contains a pointer to the empty list, draw the box like this:



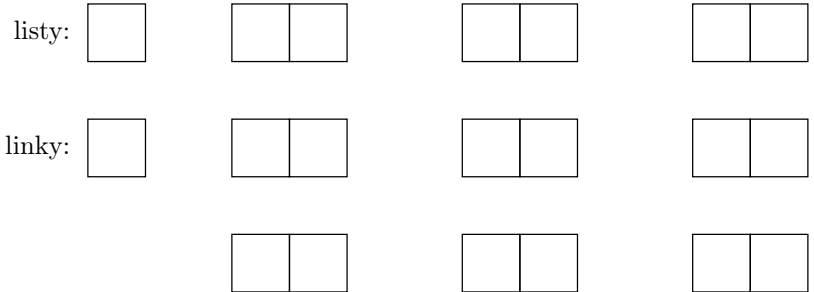
In parts (a) and (b), add arrows and values to the object skeletons on the right to show the final state of the program. Not all boxes will be used. (For examples of what kinds of box and pointer diagrams we're looking for, you might look at parts (c) and (d) first.)

**(a) (3 pt)**

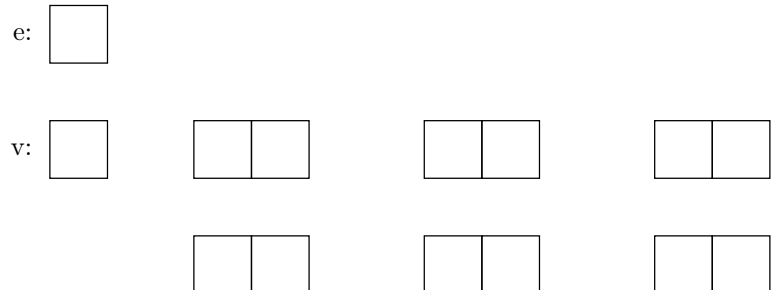
```
listy = Link(0, Link(1))
```

```
def nest(L):
    if L is Link.empty:
        return L
    N = nest(L.rest)
    L.first = Link(L.first, N)
    return L.first
```

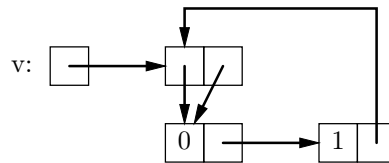
```
linky = nest(listy)
```

**(b) (3 pt)**

```
v = Link(0, Link(1, Link(2)))
e = v.rest.rest
e.rest = v.rest
v.rest.rest = v
v.rest = Link.empty
```



- (c) (3 pt) Show Python code that will produce the situation shown in the diagram. (An arrow pointing to a `Link` may be shown as pointing anywhere on the double box for that `Link`.)

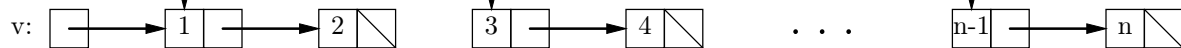
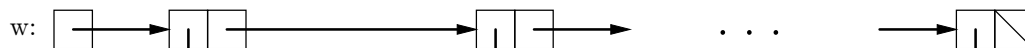
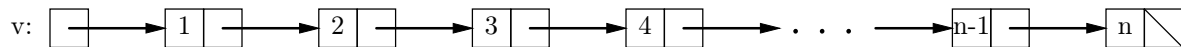


```
v = Link(_____)
```

```
v._____ = _____
```

```
v._____ = _____
```

- (d) (3 pt) Show Python code that converts the situation shown above the line into that shown below the line. Assume  $n$  is even.



```

def split2(L):
    """Assuming that linked list L has even length, breaks L into
    two-element linked lists, and returns a linked list of those lists."""

    if _____:

        return _____
    else:

        result = Link(L, _____)

        _____ = Link.empty

        return result

w = split2(v)
  
```

**2. (6 points) Complexity**

(a) (1.5 pt) Indicate which of the following assertions are true by circling the letters for those statements. An assertion such as  $\Theta(f(n)) \subseteq \Theta(g(n))$  means “any function that is in  $\Theta(f(n))$  is also in  $\Theta(g(n))$ ,” and  $\Theta(f(n)) = \Theta(g(n))$  if and only if  $\Theta(f(n)) \subseteq \Theta(g(n))$  and  $\Theta(g(n)) \subseteq \Theta(f(n))$ .

- A. If  $f(n) \in \Theta(1)$  and  $g(n) \in \Theta(1)$ , then  $\Theta(|f(n)| + |g(n)|) \in \Theta(1)$ .
- B. If  $\Theta(f(n)) = \Theta(g(n))$ , and  $g(n) > 0$  everywhere, then  $f(n)/g(n) \in \Theta(1)$ .
- C.  $\Theta(x^2) \subseteq \Theta(x^3)$ .
- D.  $\Theta(2^x) \subseteq \Theta(2^x + x^2)$ .
- E. If  $f(n) \in \Theta(1000 \cdot x^3)$ , then  $f(20) > 800$ .

(b) (1.5 pt) Consider the function

```
def num_kinks(L):
    c = 0
    i = 0
    while i < len(L):
        j = i
        while j < len(L):
            while j < len(L):
                if kink(L[i], L[j]):
                    c += 1
                    break
                j += 1
            j += 1
        i += 1
    return c
```

Circle the order of growth that best describes the worst-case execution time (measured by the number of calls to `kink`) of a call to `num_kinks` as a function of  $N$ , the length of `L`.

- A.  $\Theta(\log N)$
- B.  $\Theta(N)$
- C.  $\Theta(N^2)$
- D.  $\Theta(N^3)$
- E.  $\Theta(2^N)$

(c) (1.5 pt) Consider the following function on **Trees**

```
def count_subtrees(T, p):
    if p(T.label):
        return 1
    else:
        return sum([count_subtrees(child, p) for child in T.branches])
```

Assuming that the maximum number of children of any node is 3, circle the order of growth that best describes the worst-case execution time (measured by the number of calls to `p`) of a call to `count_subtrees` as a function of  $N$ , the number of nodes in `T`.

- A.  $\Theta(\log N)$
- B.  $\Theta(N)$
- C.  $\Theta(N^2)$
- D.  $\Theta(N^3)$
- E.  $\Theta(3^N)$

(d) (1.5 pt) For the same function as in part (c) above, and again assuming that the maximum number of children of any node is 3, circle the order of growth that best describes the worst-case execution time (measured by the number of calls to `p`) of a call to `count_subtrees` as a function of  $H$ , the height of `T`.

- A.  $\Theta(\log H)$
- B.  $\Theta(H)$
- C.  $\Theta(H^2)$
- D.  $\Theta(H^3)$
- E.  $\Theta(3^H)$ .

### 3. (1 points) From the Sum of Human Knowledge

This Renaissance composer, famous for his harmonically innovative madrigals, was also infamous for murdering his wife and her lover and thereafter having himself beaten regularly by one of his servants. Who was he?

**4. (8 points) OOPs!**

For each of the expressions in the table below, write the output displayed by the interactive Python interpreter when the expression is evaluated. The output may have multiple lines. If an error occurs, write “ERROR”. No answer requires more than 3 lines. (It’s possible that all of them require even fewer.)

```
class Thing:
    id = 0

    def fidget(self, n):
        print(n, "A", self.id)

    def fuss(self, x, n):
        print(n, "B")
        self.fidget(n)
        x.fidget(n)

    def twitch(self, n):
        self.waffle(n)

class Gadget(Thing):
    id = 1

class Whatsit(Gadget):
    def fidget(self, n):
        print(n, "D", self.id)

    def waffle(self, n):
        print(n, "D")

    def fiddle(self, x, n):
        x.waffle(n)

t1 = Thing()
t2 = Gadget()
t3 = Whatsit()
t4 = Whatsit()
t3.id = 2
```

| Expression                   | Interactive Output |
|------------------------------|--------------------|
| t3.fidget(1)                 |                    |
| t4.fidget(2)                 |                    |
| t4.fuss(t1, 3)               |                    |
| t4.fiddle(t4, 4)             |                    |
| t4.fiddle(t1, 5)             |                    |
| Thing.id = 3<br>t1.fidget(6) |                    |

**5. (8 points) Inflections**

Fill in the definition of class `Wrinkles` to conform to its doc comment. You need not use all the lines shown.

```
class Wrinkles:
```

"""An object that contains a sequence of items and a predicate (true/false function) and that, when iterated over, produces adjacent pairs of items in the sequence that satisfy the predicate.

```
>>> w = Wrinkles([1, 2, 3, 2, 4, 8, 5, 4], lambda x, y: x > y)
```

```
>>> for p in w:
```

```
... print(p)
```

(3, 2)

(8, 5)

(5, 4)

|| || ||

```
def __init__(self, L, wrinkle):
```

```
self._L = L
```

```
self._wrinkle = wrinkle
```

---

-----

---

-----

---

---

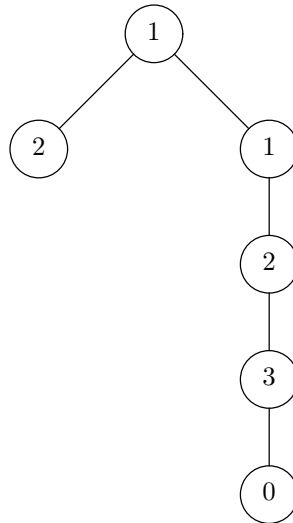
---

---

---

**6. (8 points) Tree Paths**

Given a tree,  $t$ , find the length of the longest downward sequence of node labels in the tree that are increasing consecutive integers. For example, in this tree, the longest such sequence has three labels (1, 2, 3):



As illustrated, the longest sequence can start and end anywhere in the tree, not just the root. (**Hint:** don't forget there's a `max` function.) [The original skeleton was flawed. The original skeleton appears here, and a revised skeleton on the next page.]

```

def longest_seq(t):
    """The length of the longest downward sequence of nodes in T whose
    labels are consecutive integers.
    >>> t = Tree(1, [Tree(2), Tree(1, [Tree(2, [Tree(3, [Tree(0)])])])])
    >>> longest_seq(t) # 1 -> 2 -> 3
    3
    >>> t = Tree(1)
    >>> longest_seq(t)
    1
    """

    if _____:

        return _____

    max_len = _____

    for _____:

        if _____:

            _____

        else:

            _____

    return max_len
  
```

Here is the corrected skeleton.

```
def longest_seq(t):
    """The length of the longest downward sequence of nodes in T whose
    labels are consecutive integers.
    >>> t = Tree(1, [Tree(2), Tree(1, [Tree(2, [Tree(3, [Tree(0)]])])])
    >>> longest_seq(t) # 1 -> 2 -> 3
    3
    >>> t = Tree(1)
    >>> longest_seq(t)
    1
    """
    max_len = 1

def longest(t):
    """Returns longest downward sequence of nodes starting at T whose
    labels are consecutive integers.  Updates max_len to that length,
    if greater."""

    -----

    n = 1
    if -----:

        for -----

            -----

            if -----

                n = -----

            max_len = -----

        return n
    longest(tr)
    return max_len
```

Name: \_\_\_\_\_

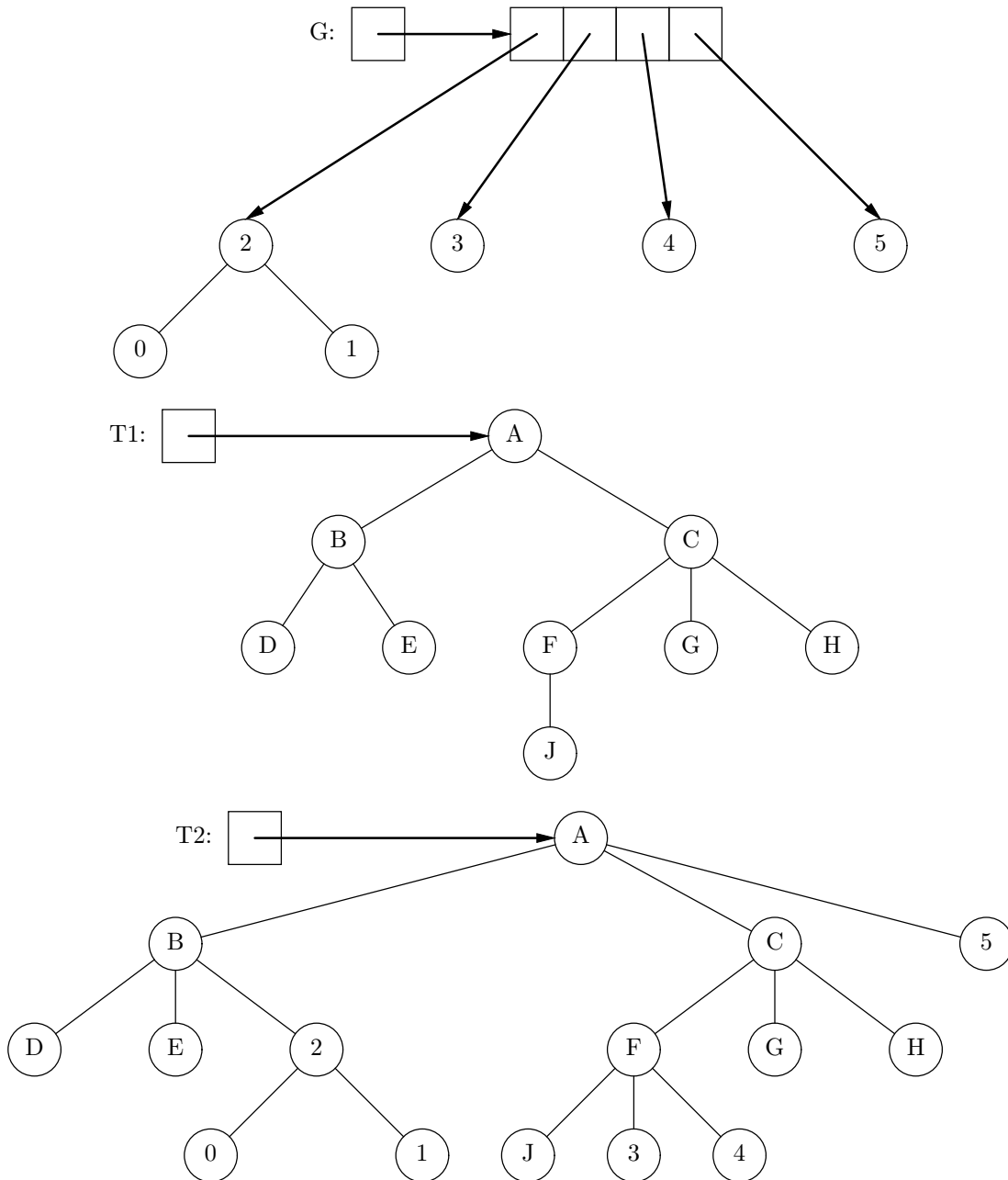
This page deliberately blank.

### 7. (8 points) Grafting

We want to insert (“graft”) branches from a sequence of trees onto a tree in places where a non-leaf node has fewer than  $K$  branches, where  $K$  is a parameter. For example, given the list of four trees  $G$  created by

$G = [ \text{Tree}(2, [\text{Tree}(0), \text{Tree}(1)]), \text{Tree}(3), \text{Tree}(4), \text{Tree}(5) ]$

and the tree  $T1$  shown below, we want  $T2 = \text{graft}(T1, G, 3)$  to destructively (and without creating any new tree nodes) turn  $T1$  into the tree  $T2$ :



The list of trees ( $G$  in the example above) will always have enough items to fill all necessary places. Trees are inserted in postorder (that is, bottom to top, left to right).

Fill in the `graft` function here. You need not use all the lines.

```
def graft(T, L, k):
    """Returns the tree created by destructively adding trees from L to
    non-leaf nodes of T with fewer than K branches. Assume that L has enough
    items to fill all necessary places. Fill in trees in postorder (bottom to
    top, left to right)."""

    grafts = iter(L)    # Don't have to use this, but it may be useful.

    def do_grafts(tr):

        if _____:

            for _____:

                _____

            while _____ < _____:

                _____

                _____

    do_grafts(T)

    return T
```

