CS 61A Structure and Interpretation of Computer Programs

Spring 2016 Test 2 (corrected)

INSTRUCTIONS

- You have 2 hours to complete the exam.
- The exam is open book, open notes, closed computer, closed calculator. The official CS 61A midterm 1 and 2 study guides will be provided.
- Mark your answers **on the exam itself**. We will *not* grade answers written on scratch paper.

Last name	
First name	
Student ID number	
BearFacts email (`@berkeley.edu`)	
Room in which you are taking this exam	
TA	
Name of the person to your left	
Name of the person to your right	
I pledge my honor that during this examination I have neither given nor received assistance. **(please sign)**	

Reference. Some questions make use of the following class definitions from labs and homework:

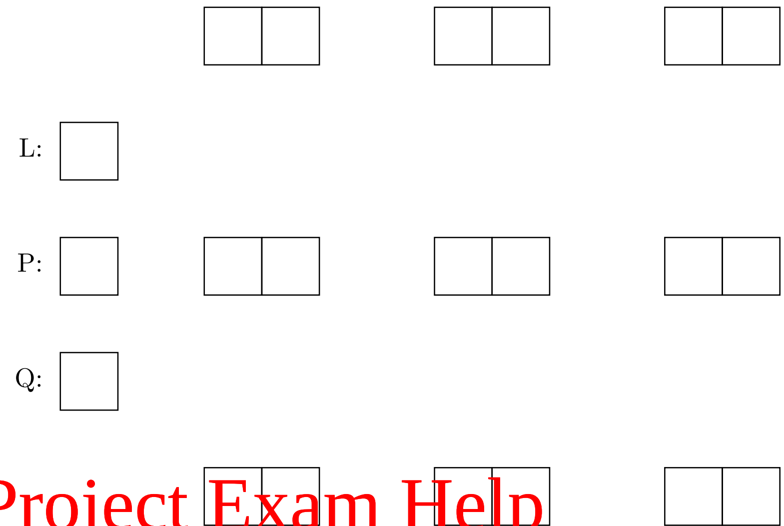
```
class Link:
    empty = ()

    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 not Link.empty:
            rest_str = ', ' + repr(self.rest)
        else:
            rest_str = ''
        return 'Link({0}{1})'.format(repr(self.first), rest_str)

    def __len__(self):
        return 1 + len(self.rest)

    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'

class Tree:
    def __init__(self, label, children=()):
        self.label = label
        self.children = list(children)

    def __repr__(self):
        if self.children:
            children_str = ', ' + repr(self.children)
        else:
            children_str = ''
        return 'Tree({0}{1})'.format(self.label, children_str)

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

1. (12 points) Pointers

For each of the following code fragments, add arrows and values to the object skeletons to the right to show the final state of the program. Single boxes are variables that contain pointers. Double boxes are `Links`. Not all boxes need be used.

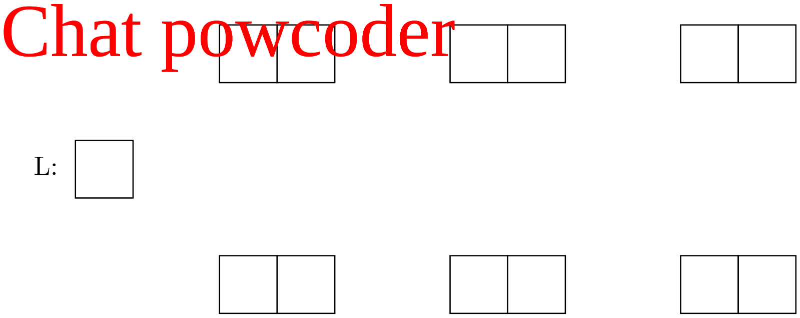
(a) (3 pt)

```
L = Link(1, Link(2))
P = L
Q = Link(L, Link(P))
P.rest.rest = Q
```



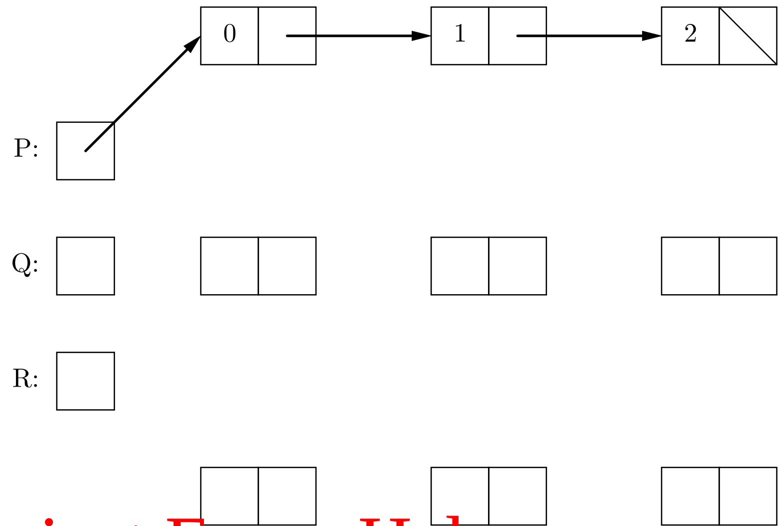
(b) (3 pt)

```
L = Link.empty
for i in range(3):
    L = Link(i, L)
```



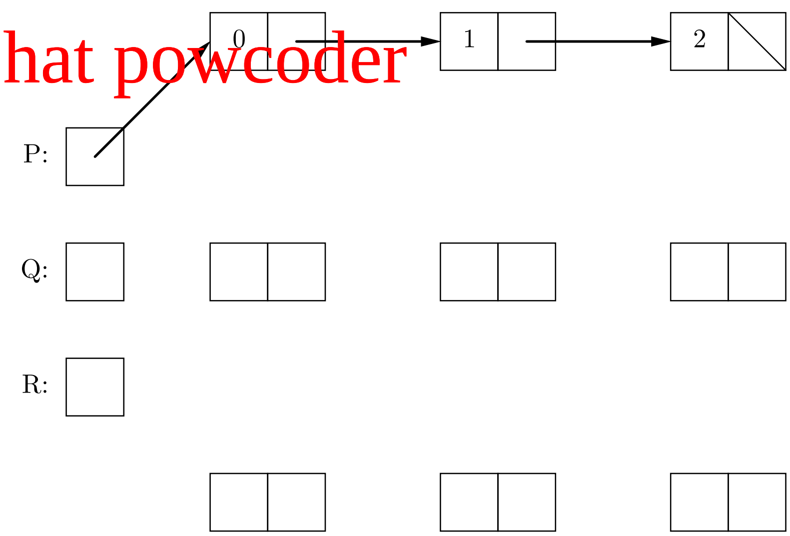
(c) **(3 pt)** For the next two problems, show the result of executing the code on the left on the initial conditions displayed on the right. We've done the first statement for you in each case, so that the diagrams on the right show the state at the point marked `# START`. Use the empty object skeletons only for newly created `Link` objects. If any pointer is modified, **neatly cross out** the original pointer and draw in the replacement. Show only the final state, not any intermediate states.

```
P = Link(0, Link(1, Link(2)))

# START
def crack1(L):
    if L is Link.empty:
        return (Link.empty, Link.empty)
    L1, L2 = crack1(L.rest)
    return (Link(L.first, L2), L1)
Q, R = crack1(P)
```



(d) **(3 pt)**

```
P = Link(0, Link(1, Link(2)))

# START
def crack2(L):
    if L is Link.empty:
        return (Link.empty, Link.empty)
    L1, L2 = crack2(L.rest)
    L.rest = L2
    return (L, L1)
Q, R = crack2(P)
```



2. (6 points) Complexity

As indicated in lecture, an assertion such as $\Theta(f(n)) \subseteq \Theta(g(n))$ means "any function that is in $\Theta(f(n))$ is also in $\Theta(g(n))$."

(a) (1.5 pt) Circle each of the following that is true.

A. $\Theta(f(n)) \subseteq O(f(n))$

B. $\Theta(2x^2 + 1000x) \subseteq \Theta(x^2)$

C. $\Theta(x^2) \neq \Theta(2x^2 + 1000x)$

D. $O(1/n) \subseteq O(1)$

E. $\Theta(1/n) \subseteq \Theta(1)$

(b) (1.5 pt) Assume that `M` is an $N \times N$ array (an N-long Python list of N-long lists). Consider the following program:

```
def search(M, x):
    N = len(M)
    Li, Uj = 0, N-1
    while Li < N and Uj >= 0:
        if M[Li][Uj] < x:
           Li += 1
        elif M[Li][Uj] > x:
           Uj -= 1
        else:
           return True
    return False
```


Circle the order of growth that best describes the worst-case execution time of a call to `search` as a function of N.

A. $\Theta(N)$

B. $\Theta(N^2)$

C. $\Theta(\log N)$

D. $\Theta(2N^2)$

E. $\Theta(2^N)$

(c) **(1.5 pt)** Consider the following implementation of `count`, which takes in a linked list of numbers `lst` and an unordered Python list of numbers `nums`, and returns a count of the number of values in `lst` that appear in `nums`:

```
def count(lst, nums):
   """The number of elements in linked list LST that appear
   appear in the unordered Python list NUMS.
   >>> L = Link(2, Link(4, Link(2, Link(3, Link(1)))))
   >>> count(L, [2, 1, 5])
   3"""
   curr = lst
   count = 0
   while curr != Link.empty:
       if curr.first in nums:
           count += 1
       curr = curr.rest
   return count
```

Circle the order of growth that best describes the worst-case execution time of `count`, as a function of n, the length of `nums`, and m, the length of `lst`. Since `nums` is a Python list, the `in` operator uses simple linear search.

A. $\Theta(n)$

B. $\Theta(m)$

C. $\Theta(n^2)$

D. $\Theta(n+m)$

E. $\Theta(nm)$

F. $\Theta(mn^2)$

(d) **(1.5 pt)** Consider the following function for computing powers of a polynomial:

```
def polypow(P, k):
    """P ** k, where P is a polynomial and K is a
    non-negative integer."""
    result = Poly(1)
    while k != 0:
        if k % 2 == 1:
            result = result.mult(P)
        P = P.mult(P)
        k = k // 2
```

Circle the order of growth that best describes the worst-case execution time of `polypow`, as a function of k, where execution time is measured in the number of times that the `.mult` method is called.

A. $\Theta(k)$

B. $\Theta(k^2)$

C. $\Theta(\sqrt{k})$

D. $\Theta(\log k)$

E. $\Theta(2^k)$

3. (8 points) Seeing Double

Fill in the functions below to produce linked lists in which each item of the original list is repeated immediately after that item. Your solutions should be iterative, not recursive.

(a) (4 pt) The function `double1` is *non-destructive,* and produces a new list without disturbing the old.

```
def double1(L):
    """Returns a list in which each item in L appears twice in sequence.
    It is non-destructive.
    >>> Q = Link(3, Link(4, Link(1)))
    >>> double1(Q)
    Link(3, Link(3, Link(4, Link(4, Link(1, Link(1))))))
    >>> Q
    Link(3, Link(4, Link(1)))
    >>> double1(Link.empty)
    ()
    """

    result = _______________________
    last = None

    while L is not Link.empty:
        if last is None:

            _________________________________________

            _________________________________________
        else:

            _________________________________________

            _________________________________________

        _______________________________________
    return result
```


(b) (4 pt) The function `double2` is *destructive,* and reuses `Link` objects in the original list wherever possible.

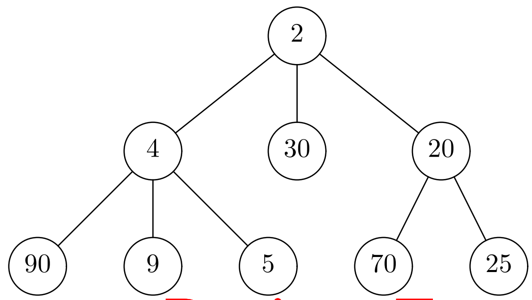
```
def double2(L):
    """Destructively modifies L to insert duplicates of each item immediately
    following the item, returning the result.
    >>> Q = Link(3, Link(4, Link(1)))
    >>> double2(Q)
    Link(3, Link(3, Link(4, Link(4, Link(1, Link(1))))))
    >>> Q
    Link(3, Link(3, Link(4, Link(4, Link(1, Link(1))))))
    """

    result = ___________________________________
    while L is not Link.empty:

        ____________________________________________________

        ____________________________________________________
    return result
```

4. (1 points) Extra

Last September, twin LIGO detectors observed gravitational waves that emanated from the merger of two black holes. In the process of this merger, three solar masses (roughly 6×10^{30} kg) were converted into gravitational energy. How many planets the size of earth (roughly 6×10^{24} kg) could this much energy accelerate to 1% of lightspeed (about 3000 km/sec)?

5. (8 points) Heaps of Trouble

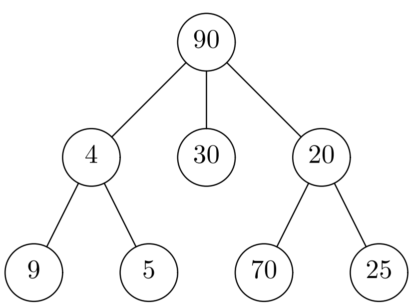
A (min-)heap is a tree with the special property (the *heap property*) that every node has a label that is less than the labels of all its child nodes. This means that the minimum element of the heap is at the root, so it can be found in constant time. For example:

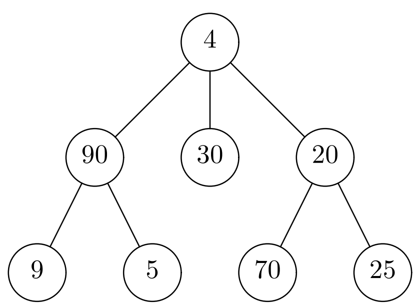


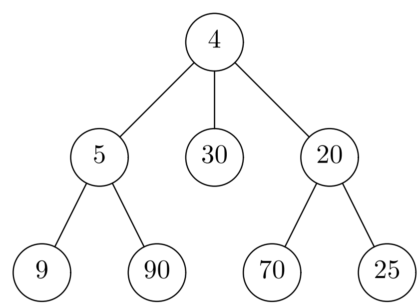
Suppose we have a heap containing at least two values. To remove and return its smallest element, while maintaining the heap property, we use the following function:


```
def remove_smallest(H):
    """Destructively remove and return the smallest value from heap H,
    restoring the heap property.  Assumes H has at least two elements."""

    result = H.label
    H.label = remove_leaf(H)   # Step 1
    reheapify(H)               # Step 2
    return result
```


The function `remove_leaf` removes one of the leaves from the heap, returning its label. The diagram on the left below shows the state of the heap above after executing Step 1 of `remove_smallest`. In general (as shown), this will cause the root to violate the heap property. To restore it, we use the function `reheapify`, which first swaps the root's label with that of its smallest child (giving the tree in the middle below). If as a result, the heap property is still violated (as in the example), `reheapify` repeats the process on down the tree until the value inserted at the top reaches a point where it is smaller than all its children, which will always be true if it reaches a leaf, as happens in the example below (shown on the right), but can also happen before that.







(a) **(4 pt)** Write the function `remove_leaf` to remove a leaf from a heap destructively and return its label. Any leaf will do, but to be specifc, have it remove the leftmost leaf of the leftmost child of the leftmost child. . . of the root. Again, we assume that there are at least two values in the heap.

```
def remove_leaf(H):
    """Destructively remove far leftmost leaf of H, returning its label"""
    child = H.children[0]
    if _______________________:

        v = child.label

        H.children = _________________________________

        return v
    else:
        return ___________________________________
```

(b) **(4 pt)** Write the function `reheapify` to restore the heap property of a heap destructively, assuming that initially it is violated (if at all) only at the root.

```
def reheapify(H):
    """Destructively restore the heap property of H, assuming it is
    violated only at H itself, if at all."""

    if ____________________:
        return
    else:
        s = H.children[0]
        for c in H.children:
            if _________________________:
                s = c
        if _______________:

            s.label, H.label = ____________________________

            __________________________
```

6. (8 points) OOPs

Given the class definitions on the left, fill in the blanks to show what the Python interpreter would print. Print "`ERROR`" for cases that would cause an exception. Put "`<None>`" for cases where the Python interpreter would print nothing.

```
class Person:

    name = "Outis"

    def get_name(self):
        return self.name

    def response(self, question):
        v = self.cogitate(question)
        if v is None:
            return "I do not know"
        else:
            return v

    def cogitate(self, question):
        return None

    def set_name(self, new_name)
        self.name = new_name

    def __str__(self):
        return self.name

class Learner(Person):

    def __init__(self):
        self.facts = {}

    def learn(self, question, answer):
        self.facts[question] = answer
        return 'Got it'

    def cogitate(self, question):
        if question in self.facts:
             return self.facts[question]

class Beginner(Learner):
    def __init__(self, name):
        Learner.__init__(self)
        self.set_name(name)

    def response(self, question):
        r = Person.response(self, question)
        return "I think " + r
```

```
>>> odysseus = Learner()
>>> odysseus.learn('god', 'Athena')
______________________________
>>> hipp = Beginner('Hippothales')
>>> hipp.learn('favorite person', 'Lysis')
______________________________
>>> odysseus.get_name()
______________________________
>>> hipp.get_name()
______________________________
>>> Person.name = "Nemo"
>>> hipp.get_name()
______________________________
>>> odysseus.get_name()
______________________________
>>> odysseus.set_name(odysseus.get_name())
>>> Person.name = "Nobody"
>>> odysseus.get_name()
______________________________
>>> someone = Person()
>>> someone.learn('Earth mass', '5.972e24 kg')
______________________________
>>> someone.response('Earth mass')
______________________________
>>> hipp.response('favorite person')
______________________________
>>> odysseus.response('god')
______________________________
```

7. (8 points) Evicted!

An *LRU cache* (stands for "least recently used") is a kind of dictionary that can only hold a fixed, finite number of keys (its *capacity*) and corresponding values. When addition of a new key would exceed that capacity, the least recently accessed key in the cache is removed ("*evicted*") and replaced with the new value. Such caches are used to speed up access to some relatively slow, but much larger dictionaries. For example, most computers have a large main memory and various caches for saving and retrieving recently accessed memory values; the latter can be 200 times faster than the former.

(a) (2 pt)

Consider the following "slow" dictionary implementation:

```
class SlowData:
    """
    Simulates a basic read-only memory store of KEY => VALUE mappings
    >>> slow_data = SlowData(((0, 'a'), (1, 'b'), (2, 'c')))
    >>> slow_data[1]
    'b'
    >>> slow_data[2]
    'c'
    """
    def __init__(self, data):
        self._data = data          # A sequence of (KEY, VALUE) tuples

    def __getitem__(self, key):
        """Get the value associated with KEY, or None if there is none."""
        for curr_key, curr_value in self._data:
            if key == curr_key:
                return curr_value
        return None
```


If `mem` is a `SlowData` containing N tuples, what is the worst-case execution time for the following code fragment?

```
result = 0
for i in range(N): result += mem[i]
```

Circle the correct answer below.

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

(b) (4 pt)

An `LRUCache` object is intended to provide access to values from a `SlowData` in such a way that the results of some recent accesses to the `SlowData` object are saved and subsequently accessed quickly. To do this, the cache keeps a list of key/value tuples whose size has a fixed upper limit. If a key that is in the cache is accessed, its corresponding value is fetched from this list without consulting the `SlowData` object. If a key is not in the cache, it is fetched from the `SlowData` object. Each time a value is referenced, it is placed at or moved to the **end** of the cache list, and if that makes the list too long (longer than the capacity), the **first** item in the list is removed (so that it will have to be retrieved from the `SlowData` object if accessed again).

Fill in the code below to have this behavior. (A convenient way to remove the item at index k from a list L is `del L[k]`.)

```
class LRUCache:
    def __init__(self, capacity, slow_data):
        self._capacity = capacity
        self._slow_data = slow_data
        self._cache = []

    def __getitem__(self, key):
        for i in range(len(self._cache)):
            pair = self._cache[i]

            if ______________________:

                _____________________________

                _____________________________

                return pair[1]
        v = self._slow_data[key]

        self._cache_____________________________
        if len(self._cache) > self._capacity:

            del ________________________________
        return v
```


(c) (1 pt) If `mem` is a `SlowData` containing N tuples, what is the worst-case execution time for the following code fragment?

```
cached_mem = LRUCache(4, mem)
result = 0
for i in range(N): result += cached_mem[i]
```

Circle the correct answer below.

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

(d) (1 pt) If `cached_mem` is as above, what is the worst-case execution time for the following code fragment?

```
result = 0
for i in range(N): result += cached_mem[i % 4]
```

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