INSTRUCTIONS

- You have 2 hours to complete the exam.
- The exam is closed book, closed notes, closed computer, closed calculator, except two hand-written 8.5" × 11" crib sheets of your own creation and the official CS 61A study guides.
- Mark your answers on the exam itself. We will not grade answers written on scratch paper.

<table>
<thead>
<tr>
<th>Last name</th>
</tr>
</thead>
<tbody>
<tr>
<td>First name</td>
</tr>
<tr>
<td>Student ID number</td>
</tr>
<tr>
<td>CalCentral email (@berkeley.edu)</td>
</tr>
<tr>
<td>TA</td>
</tr>
<tr>
<td>Name of the person to your left</td>
</tr>
<tr>
<td>Name of the person to your right</td>
</tr>
</tbody>
</table>

All the work on this exam is my own. (please sign)

POLICIES & CLARIFICATIONS

- If you need to use the restroom, bring your phone and exam to the front of the room.
- You may use built-in Python functions that do not require import, such as min, max, pow, len, abs, sum, next, iter, list, tuple, map, filter, zip, all, and any.
- You may not use example functions defined on your study guides unless a problem clearly states you can.
- For fill-in-the blank coding problems, we will only grade work written in the provided blanks. You may only write one Python statement per blank line, and it must be indented to the level that the blank is indented.
- Unless otherwise specified, you are allowed to reference functions defined in previous parts of the same question.
- You may use the Tree, Link, and BTree classes defined on Page 2 (left column) of the Midterm 2 Study Guide.
1. (12 points) A/B Test

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”, but include all output displayed before the error. If evaluation would run forever, write “Forever”. To display a function value, write “Function”. The first two rows have been provided as examples.

The interactive interpreter displays the value of a successfully evaluated expression, unless it is None.

Assume that you have first started python3 and executed the statements on the left.

```python
x = [1, 2]

class A:
    x = 3
    y = 4

def __init__(self, y):
    self.a = y
    self.x = b
    self.__str__ = lambda: str(y)

def __str__(self):
    return 'BB'

class B(A):
    x = [5, 6]

def __init__(self, y):
    self.a = x[1]
    y[1] = 8

b = B(x)
a = A(6)

def dash(x):
    return print(self.x)

elastigirl = Link(7, Link(8))
elastigirl.first = elastigirl.rest
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>pow(10, 2)</td>
<td>100</td>
</tr>
<tr>
<td>print(2, Link(3))</td>
<td>&lt;2, 3&gt;</td>
</tr>
<tr>
<td>[c.a for c in [a, b]]</td>
<td>[6, 2]</td>
</tr>
<tr>
<td>print(a.x, b.x)</td>
<td>BB [5, 6]</td>
</tr>
<tr>
<td>print(b.y, x)</td>
<td>4 [1, 8]</td>
</tr>
<tr>
<td>a.<strong>str</strong>()</td>
<td>'6'</td>
</tr>
<tr>
<td>dash(b)</td>
<td>Error</td>
</tr>
<tr>
<td>print(elastigirl)</td>
<td>&lt;&lt;8&gt;, 8&gt;</td>
</tr>
</tbody>
</table>

2. (3 points) Implement lowest, which takes a list of numbers s and returns a list of only the elements of s with the smallest absolute value. You may only write a single name in each blank.

```python
def lowest(s):
    """Return a list of the elements in s with the smallest absolute value."

    >>> lowest([3, -2, 2, -3, -4, 2, 3, 4])
    [-2, 2, 2]
    >>> lowest(range(-5, 5))
    [0]
    """
    return (lambda y: [x for x in s if abs(x) == y])(abs(min(s, key=abs)))
```
### 3. (8 points) Hocus Pocus

Fill in the environment diagram that results from executing the code on the right until the entire program is finished, an error occurs, or all frames are filled. You may not need to use all of the spaces or frames.

A complete answer will:

- Use box-and-pointer notation for all lists.
- Add missing names and parents to all local frames.
- Add missing values created or referenced during execution.
- Show the return value for each local frame.

```python
def put(hocus, pocus):
    hocus = 2
    you = pocus[hocus]
    def pocus():
        nonlocal hocus
        if type(spell.pop()) == list:
            you.append(hocus)
            return [pocus(), hocus]
        else:
            hocus = 3
            return spell[0:1]
    return pocus

spell = [6, 5, [4]]
you = spell
put(spell, you)()
```

---

**Global frame**

- **put**: `_________` [parent=Global]
- **spell**: `_________` [parent=_________]
- **you**: `_________` [parent=_________]

**f1: put [parent=Global]**

- **hocus**: `3`
- **you**: `_________` [parent=_________]
- **pocus**: `_________` [parent=_________]

**f2: pocus [parent=f1]**

**f3: pocus [parent=f1]**

---

**Return Value**

1. `6`
2. `4 2 1`
3. `6 0`
4. (10 points) Nonplussed

Definition. A plus expression for a non-negative integer \( n \) is made by inserting + symbols in between digits of \( n \), such that there are never more than two consecutive digits in the resulting expression. For example, one plus expression for 2018 is 20+1+8, and its value is 29. Assume that a two-digit number starting with 0 evaluates to its one's digit. For example, another plus expression for 2018 is 2+01+8, and its value is 11.

(a) (3 pt) Implement plus, which takes a non-negative integer \( n \). It returns the largest value of any plus expression for \( n \).

```python
def plus(n):
    """Return the largest sum that results from inserting +'s into n."

    if n:
        return max(n % 10 + plus(n // 10), n % 100 + plus(n // 100))
    return 0
```

(b) (5 pt) Implement plusses, which takes non-negative integers \( n \) and \( \text{cap} \). It returns the number of plus expressions for \( n \) that have a value less than \( \text{cap} \).

```python
def plusses(n, cap):
    """Return the number of plus expressions for n with values below cap."

    if n < 10 and n < cap:
        return 1
    elif cap <= 0:
        return 0
    else:
        return plusses(n // 10, cap - n % 10) + plusses(n // 100, cap - n % 100)
```

(c) (2 pt) Circle the \( \Theta \) expression that describes how many addition operations are required to evaluate a plus expression for a positive integer \( n \).

\[ \Theta(1) \quad \Theta(\log n) \quad \Theta(n) \quad \Theta(n^2) \quad \Theta(2^n) \quad \Theta(10^n) \quad \text{None of these} \]
5. (8 points) Midterm Elections

(a) (6 pt) Implement the Poll class and the tally function, which takes a choice c and returns a list describing the number of votes for c. This list contains pairs, each with a name and the number of times vote was called on that choice at the Poll with that name. Pairs can be in any order. Assume all Poll instances have distinct names. Hint: the dictionary get(key, default) method (MT 2 guide, page 1 top-right) returns the value for a key if it appears in the dictionary and default otherwise.

```python
class Poll:
    s = []
    def __init__(self, n):
        self.name = n
        self.votes = {}
    Poll.s.append(self)

    def vote(self, choice):
        self.votes[choice] = self.votes.get(choice, 0) + 1

    def tally(c):
        return [(p.name, p.votes[c]) for p in Poll.s if c in p.votes]
```

(b) (2 pt) Implement the vote method of the Crooked class, which only records every other vote call for each Crooked instance. Only odd numbered calls to vote are recorded, e.g., first, third, fifth, etc.

```python
class Crooked(Poll):
    record = True
    def vote(self, choice):
        if self.record:
            Poll.vote(self, choice)
        self.record = not self.record
```

```python
c >>> a, b, c = Poll('A'), Poll('B'), Poll('C')
 >>> c.vote('dog')
 >>> a.vote('dog')
 >>> a.vote('cat')
 >>> b.vote('cat')
 >>> a.vote('dog')
 >>> tally('dog')
 [('A', 2), ('C', 1)]
 >>> tally('cat')
 [('A', 1), ('B', 1)]
```
6. (4 points) Dr. Frankenlink

Implement replace, which takes two non-empty linked lists \( s \) and \( t \), as well as positive integers \( i \) and \( j \) with \( i < j \). It mutates \( s \) by removing elements with indices from \( i \) to \( j \) (removing element \( i \) but not removing element \( j \)) and replacing them with \( t \). Afterward, \( s \) contains all of the objects in \( t \), so a change to \( t \) would be reflected in \( s \) as well. \( t \) may change as a result of calling replace. Assume \( s \) has at least \( j \) elements.

```python
def replace(s, t, i, j):
    """Replace the slice of s from i to j with t."
    >>> s, t = Link(3, Link(4, Link(5, Link(6, Link(7))))), Link(0, Link(1, Link(2)))
    >>> replace(s, t, 2, 4)
    >>> print(s)
    <3, 4, 0, 1, 2, 7>
    >>> t.rest.first = 8
    >>> print(s)
    <3, 4, 0, 8, 2, 7>
    """
    assert s is not Link.empty and t is not Link.empty and i > 0 and i < j

    if i > 1:
        replace(s.rest, t, i - 1, j - 1)
    else:
        for k in range(j - i):
            s.rest = s.rest.rest
        end = t
        while end.rest is not Link.empty:
            end = end.rest
        s.rest, end.rest = t, s.rest
```
7. (5 points) Trictionary or Treat

**Definitions.** A trictionary is a pair of Tree instances $k$ and $v$ that have identical structure: each node in $k$ has a corresponding node in $v$. The labels in $k$ are called keys. Each key may be the label for multiple nodes in $k$, and the values for that key are the labels of all the corresponding nodes in $v$.

A lookup function returns one of the values for a key. Specifically, a lookup function for a node in $k$ is a function that takes $v$ as an argument and returns the label for the corresponding node in $v$.

Implement the generator function `lookups`, which takes a Tree instance $k$ and some key. It yields all lookup functions for nodes in $k$ that have key as their label. The new_lookup function is part of the implementation.

```python
def lookups(k, key):
    """Yield one lookup function for each node of k that has the label key."
    if k.label == key:
        yield lambda v: v.label
    for i in range(len(k.branches)):
        for lookup in lookups(k.branches[i], key):
            yield new_lookup(i, lookup)

def new_lookup(i, f):
    def g(v):
        return f(v.branches[i])
    return g
```

```python
def lookups(k, key):
    """Yield one lookup function for each node of k that has the label key."
    if k.label == key:
        yield lambda v: v.label
    for i in range(len(k.branches)):
        for lookup in lookups(k.branches[i], key):
            yield new_lookup(i, lookup)

def new_lookup(i, f):
    def g(v):
        return f(v.branches[i])
    return g
```

\[ k = \text{Tree}(5, [\text{Tree}(7, [\text{Tree}(2)]), \text{Tree}(8, [\text{Tree}(3), \text{Tree}(4)]), \text{Tree}(5, [\text{Tree}(4), \text{Tree}(2)])]) \]

\[ v = \text{Tree}('Go', [\text{Tree}('C', [\text{Tree}('C')]), \text{Tree}('A', [\text{Tree}('S'), \text{Tree}(6)]), \text{Tree}('L', [\text{Tree}(1), \text{Tree}('A')])]) \]
No more questions.