CS 1110

Prelim 2 Review
Spring 2021

Announcements

- Prelim 2 Thurs Apr 22 at 6:30 - 8pm (university-scheduled)
  - Your seat or Zoom link will be assigned this afternoon via CMS
  - In-person: Bring pens/pencils/erasers (bring several). Bring a watch or even an actual clock if you have one. No smart watches/phones! You may not be able to see the wall clock in Barton from your seat. *Bring Cornell ID*.
  - Online: *Different this time:* log on to Zoom proctor session on both devices. Students who have not done a mock exam (for Prelim 1) will be contacted to do one.
- Labs this week: Prelim 2 review, focus on class methods
- Thurs Apr 22 lecture time → office hours

Studying for the Exam

- Read study guide. Notes differences among the semesters
- Review all labs and assignments
  - You should be able to do all problems now
- Look at exams from past years
  - Exams with solutions on course web page
  - Refer to info in study guide regarding differences among the semesters

Prelim 2 Topics

- Topics after prelim 1:
  - Recursion — now
  - Classes — lab
- Topics before but not on prelim 1:
  - Nested lists — now
  - Iteration with nested loops — now
  - Dictionaries and tuples — now

While-loop *not* on Prelim 2

Recursion: Before You Begin

- Plan out how you will approach the task before writing code
- Consider the following:
  - How can you “divide and conquer” the task?
  - Do you understand the spec?
  - How would you describe the implementation of the function using words?

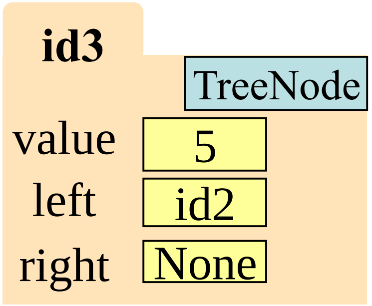
Recursion

1. Base case
2. Recursive case
3. Ensure the recursive case makes progress towards the base case

Base Case

- Create cases to handle smallest units of data
- Depends on what type of data is being processed and what the function must do to that data

Base Case Examples

	Strings	Lists	Objects (see final example)
1 Element	“5”	[5]	
0 Elements	“”	[]	None

Recursive Case

- Divide and conquer: how to divide the input so that we can call the function recursively on smaller input
- When calling the function recursively, assume that it works exactly as the specification states it does -- don’t worry about the specifics of your implementation here
- Use this recursive call to handle the rest of the data, besides the small unit being handled

Make Progress

- Recursive calls must always make some sort of "progress" towards the base cases
- This is the only way to ensure the function terminates properly
- Risk having infinite recursion otherwise

Recursive Function (Fall 2017)

```
def filter(nlist):
    """Return: a copy of nlist with all negative numbers removed.

    The order of the original list is preserved

    Example: filter([1,-1,2,-3,-4,0]) returns [1,2,0]

    Precondition: nlist is a (possibly empty) list of numbers."""
```

Plan:

- Use divide-and-conquer to break up the list
- Filter each "half" and put back together

Recursive Function (Fall 2017)

```
def filter(nlist):
    """Return: a copy of nlist (in order) with negative numbers."""
    if len(nlist) == 0:
        return [ ]
    elif len(nlist) == 1:
        return nlist[:] if nlist[0] >= 0 else [ ]  # THIS does the work

    # Break it up into two parts
    left = filter(nlist[:1])
    right = filter(nlist[1:])

    #  Combine
    return left+right
```

Recursive Function (Fall 2017)

```
def filter(nlist):
    """Return: a copy of nlist (in order) with negative numbers."""
    if len(nlist) == 0:
        return [ ]

    # Do the work by removing one element
    left = nlist[:1]
    if left[0] < 0:
        left = [ ]
    right = filter(nlist[1:])

    #  Combine
    return left+right
```

Either approach works. Do what is easiest to you.

Recursive Function (Fall 2014)

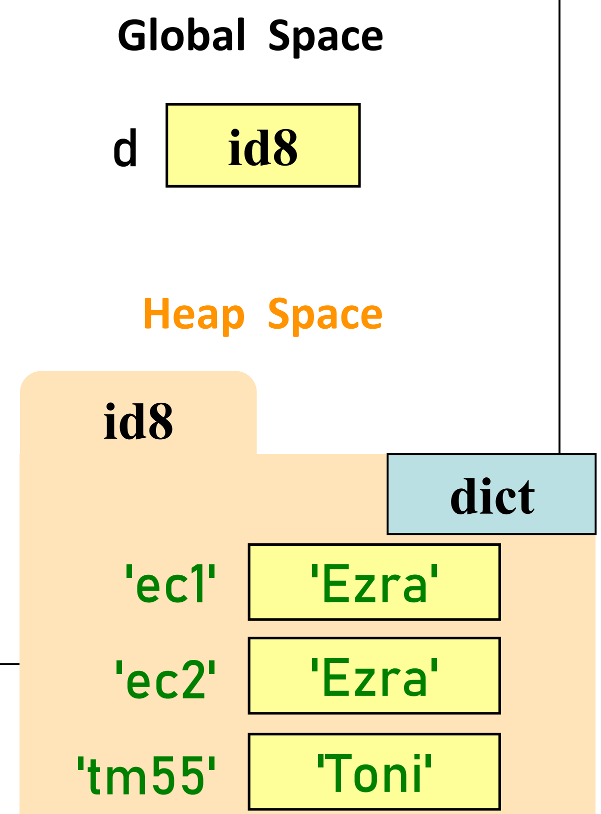
```
def histogram(s):
    """Return: a histogram (dictionary) of the # of letters in string s.

    The letters in s are keys, and the count of each letter is the value. If
    the letter is not in s, then there is NO KEY for it in the histogram.

    Example: histogram('') returns {},
             histogram('abracadabra') returns {'a':5, 'b':2, 'c':1, 'd':1, 'r':2}

    Precondition: s is a string (possibly empty) of just letters."""
```

Plan:

- Use divide-and-conquer to break up the string
- Get two dictionaries back when you do
- Pick one and insert the results of the other

Dictionaries (Type dict)

```
>>> d = {'ec1':'Ezra', 'ec2':'Ezra', 'tm55':'Toni'}
>>> d['ec1']
'Ezra'
>>> d[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0
>>> d[:1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'slice'
>>>
```



- Can access elements like a list
- Must use the key, not an index
- Cannot slice ranges

Recursive Function

```
def histogram(s):
    """Return: a histogram (dictionary) of the # of letters in string s."""
    if s == '':                                # Small data
        return {}

    # left = { s[0]: 1 }.                     No need to compute this
    right = histogram(s[1:])

    if s[0] in right:                          # Combine the answer
        right[s[0]] = right[s[0]]+1
    else:
        right[s[0]] = 1
    return right
```

Iteration with For-Loops

Two ways to implement the `for`-loop

`for x in alist:`

- `x` is each value inside the list
- Modifying `x` does not modify the list

`for x in range(len(alist)):`

- `x` represents each *index* of the list
- Modifying `alist[x]` modifies the list

Example with 2D Lists

```
def max_cols(table):
    """Returns: List storing max value of each column
    We assume that table is a 2D list of floats (so it is a list of rows and
    each row has the same number of columns.  This function returns
    a new list that stores the maximum value of each column.)
    Examples:
        max_cols([ [1,2,3], [2,0,4], [0,5,2] ]) is [2,5,4]
        max_cols([ [1,2,3] ]) is [1,2,3]
    Precondition: table is a NONEMPTY 2D list of floats
    Built-in function max not allowed. """
```

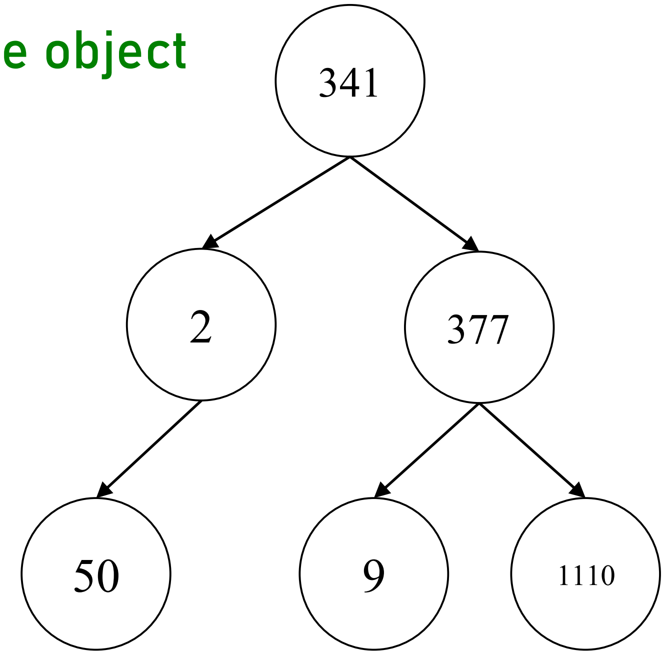
Example with 2D Lists

```
def max_cols(table):
    """Returns: List storing max value of each column
    Precondition: table is a NONEMPTY  2D list of floats"""
    # Use the fact that table is not empty
    result = table[0][:] # Make a copy, do not modify table
    # Loop through rows, then loop through columns
    for row in table:
        for k in range(len(row)):
            if row[k] > result[k]:
                result[k] = row[k]
    return result
```

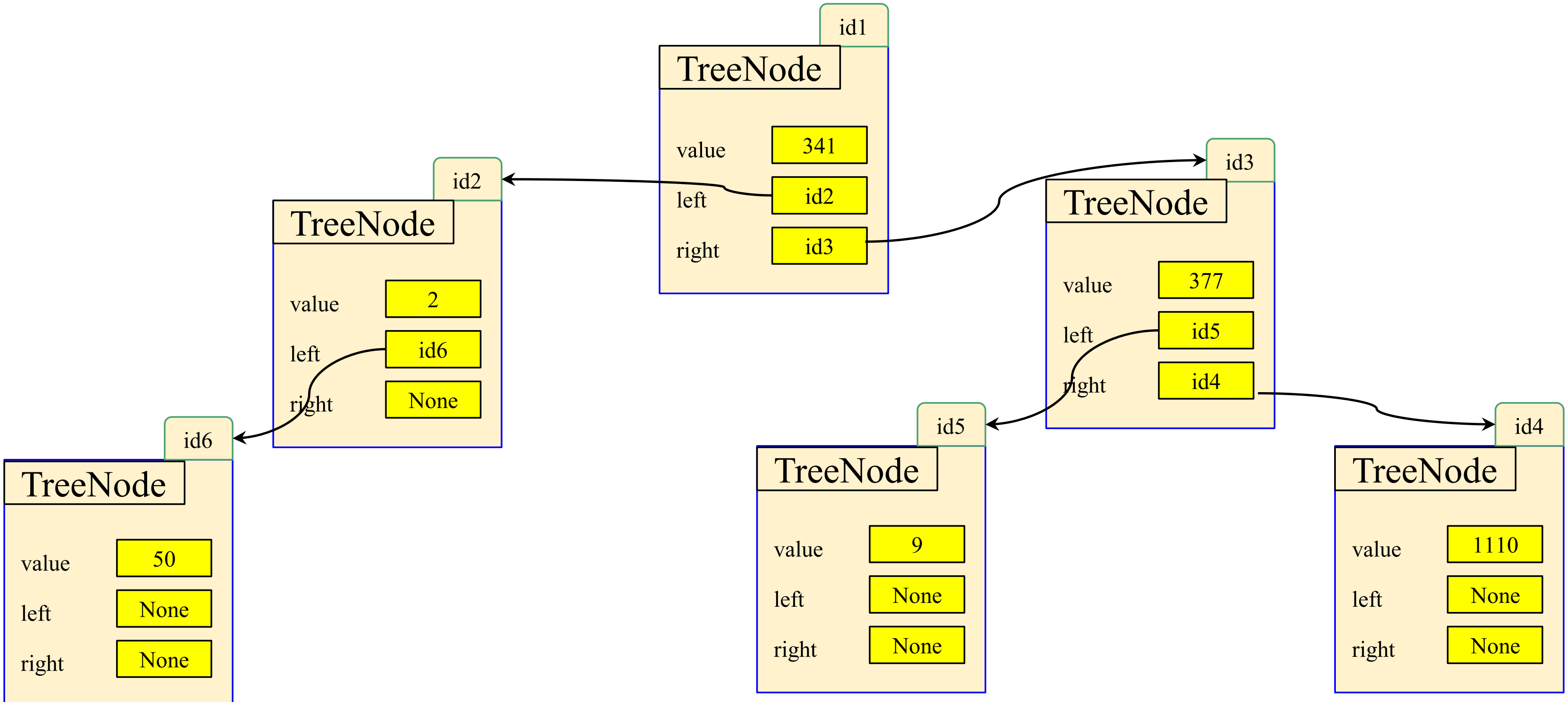
Questions? Next up: Office Hours

Recursion with Objects

```
class TreeNode (object):
    """Attributes:
        value: An int, the "value" of this TreeNode object
        left: A TreeNode object, or None
        right: A TreeNode object, or None"""
```



Understanding the Object's Structure



Recursion with Objects

```
def contains (t, v):
    """
    Return: True if any of the TreeNode objects in the entire "tree" have the value v

    Define the "tree" as the TreeNode t, as well as the TreeNodes accessible
    through the left and right attributes of t (if not None)
    Preconditions: t is a TreeNode, or None.  v is an int.
    """
```

Recursion with Objects

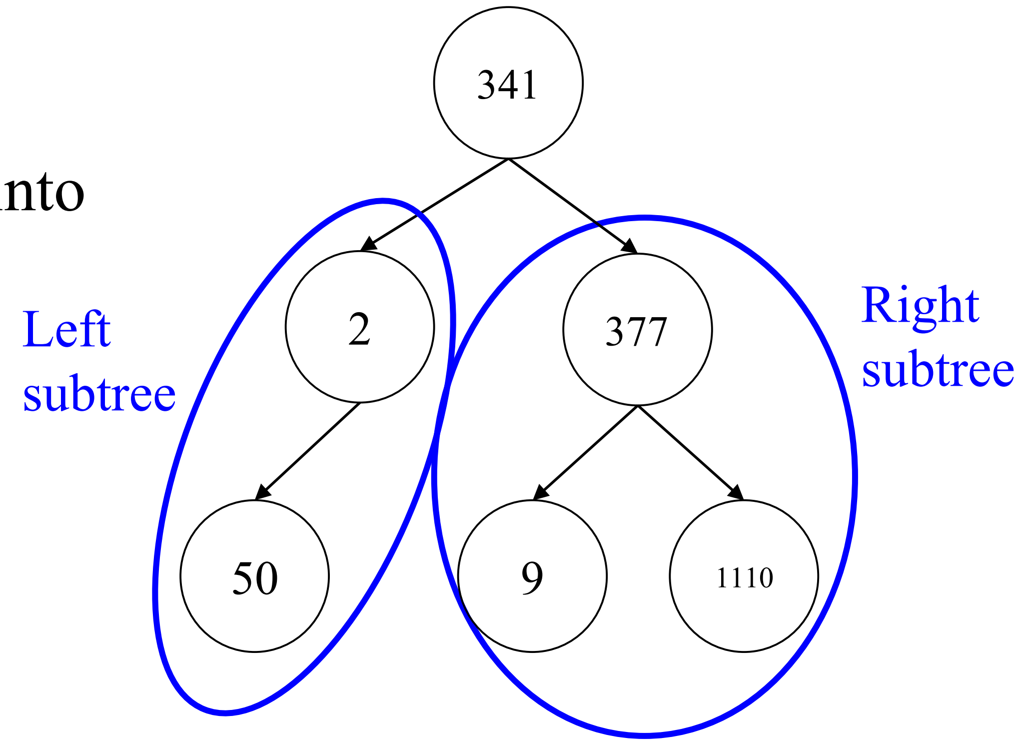
```
def contains (t, v):
    """
    Return: True if any of the TreeNode objects in the entire “tree” have the value v

    Define the “tree” as the TreeNode t, as well as the TreeNodes accessible
    through the left and right attributes of t (if not None)

    Preconditions: t is a TreeNode, or None.  v is an int.
    """

    if t is None:          # Case: None/non-existent Tree
        return False
    elif t.value == v:     # Case: Found value
        return True

    # Now what?
```

Divide and Conquer on Trees

Recall the tree structure...

They can be easily divided into left and right subtrees!

Recursion on left
Recursion on right
Put result back together



Recursion with Objects

```
def contains (t, v):
    """
    Return: True if any of the TreeNode objects in the entire “tree” have the value v

    Define the “tree” as the TreeNode t, as well as the TreeNodes accessible
    through the left and right attributes of t (if not None)

    Preconditions: t is a TreeNode, or None.  v is an int.
    """
    if t is None:          # Case: None/non-existent Tree
        return False
    elif t.value == v:     # Case: Found value
        return True

    # Here need to check t.left subtree and t.right subtree
    left_result= contains(t.left, v)        # Recursively check branches
    right_result= contains(t.right, v)
    return left_result or right_result   # Combining two bools
```

What is the type of t.left and t.right?

What happens if t.left or t.right is None?