Recursive Data Structures

1. Binary Trees
   - sorting numbers
   - a recursive tree builder
   - flattening and searching
2. Classification Trees
   - lists and dictionaries
   - classifying with dictionary tree
   - adding elements to the tree
3. Assignments

MCS 275 Lecture 13
Programming Tools and File Management
Jan Verschelde, 8 February 2017

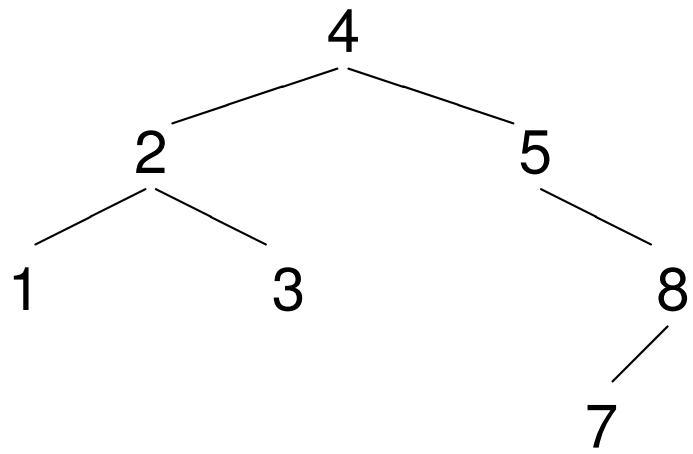
Recursive Data Structures

1. Binary Trees
   - sorting numbers
   - a recursive tree builder
   - flattening and searching
2. Classification Trees
   - lists and dictionaries
   - classifying with dictionary tree
   - adding elements to the tree
3. Assignments

sorting numbers using a tree

Consider the sequence 4, 5, 2, 3, 8, 1, 7

Insert the numbers in a tree:



Rules to insert x at node N:

- if N is empty, then put x in N
- if $x < N$, insert x to the left of N
- if $x \geq N$, insert x to the right of N

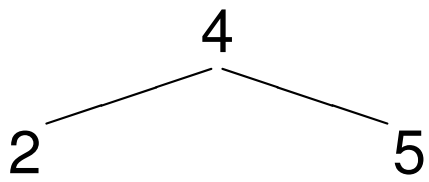
Recursive printing: left, node, right sorts the sequence.

recursive triplets in Python

Any node in a tree T is either empty or consists of

1. left branch (or child) is a tree
2. the data at the node is a number
3. right branch (or child) is a tree

To represent



we use tuples, sequences enclosed by (and):

```
>>> L = ((),2,())
>>> R = ((), 5, ())
>>> T = (L, 4, R)
>>> T
(((), 2, ()), 4, ((), 5, ()))
```

incremental sorting

```
$ python treesort.py
give a number (-1 to stop) : 4
T = ((), 4, ())
give number (-1 to stop) : 5
T = ((), 4, ((), 5, ()))
give number (-1 to stop) : 2
T = (((), 2, ()), 4, ((), 5, ()))
give number (-1 to stop) : 3
T = (((), 2, ((), 3, ())), 4, ((), 5, ()))
give number (-1 to stop) : 8
T = (((), 2, ((), 3, ())), 4, ((), 5, ((), 8, ())))
give number (-1 to stop) : -1
sorted numbers = [2, 3, 4, 5, 8]
```

Recursive Data Structures

1. Binary Trees
   - sorting numbers
   - a recursive tree builder
   - flattening and searching
2. Classification Trees
   - lists and dictionaries
   - classifying with dictionary tree
   - adding elements to the tree
3. Assignments

adding to the tree

```
def add(tree, nbr):
    """
    Adds a number nbr to the triple of triples.
    All numbers less than tree[1] are in tree[0].
    All numbers greater than or equal to tree[1]
    are in tree[2].  Returns the new tree.
    """
    if len(tree) == 0:
        return ((), nbr, ())
    elif nbr < tree[1]:
        return (add(tree[0], nbr), tree[1], tree[2])
    else:
        return (tree[0], tree[1], add(tree[2], nbr))
```

Recursive Data Structures

1. Binary Trees
   - sorting numbers
   - a recursive tree builder
   - flattening and searching
2. Classification Trees
   - lists and dictionaries
   - classifying with dictionary tree
   - adding elements to the tree
3. Assignments

flattening the tree

The tree `T`

```
(((), 2, ((), 3, ())), 4, ((), 5, ((), 8, ())))
```

already orders the numbers increasingly, as

```
L = [2, 3, 4, 5, 8]
```

To flatten the tree `T` into the list `L`,
we traverse `T` as follows:

- if the node is empty, return `[]`
- for a node that is not empty:
  1. let `L` be the flattened left branch
  2. append to `L` the data at the node
  3. append to `L` the flattened right branch

  and finally return the list `L`

searching the tree

Suppose we do not wish to store duplicate elements.

To see whether a number `n` already belongs to a tree `T` we apply the following recursive algorithm:

- if `T` is empty, we return `False`
- if the data at the node is `n`, return `True`
- if `n` is less then the data at `T`,
  return the result of search in left branch
  otherwise return result of search in right branch

the function `is_in`

```
def is_in(tree, nbr):
    """
    Returns True if nbr belongs to the tree,
    returns False otherwise.
    """
    if len(tree) == 0:
        return False
    elif tree[1] == nbr:
        return True
    elif nbr < tree[1]:
        return is_in(tree[0], nbr)
    else:
        return is_in(tree[2], nbr)
```

the function `flatten`

```
def flatten(tree):
    """
    tree is a recursive triple of triplets.
    Returns a list of all numbers in tree
    going first along the left of tree, before
    the data at the node and the right of tree.
    """
    if len(tree) == 0:
        return []
    else:
        result = flatten(tree[0])
        result.append(tree[1])
        result = result + flatten(tree[2])
        return result
```

the main function

```
def main():
    """
    Prompts the user for numbers and sorts
    using a tree: a triple of triplets.
    """
    tree = ()
    while True:
        nbr = int(input('give a number (-1 to stop) : '))
        if nbr < 0:
            break
        if is_in(tree, nbr):
            print(nbr, 'is already in the tree')
        else:
            tree = add(tree, nbr)
        print('T =', tree)
    print('sorted numbers =', flatten(tree))
```

Recursive Data Structures

1. Binary Trees
   - sorting numbers
   - a recursive tree builder
   - flattening and searching
2. Classification Trees
   - lists and dictionaries
   - classifying with dictionary tree
   - adding elements to the tree
3. Assignments

trees with lists and dictionaries

To build trees with a variable number of children at each node we can use lists instead of tuples.

A more flexible indexing mechanism is provided by dictionaries (similar to `struct` in C).

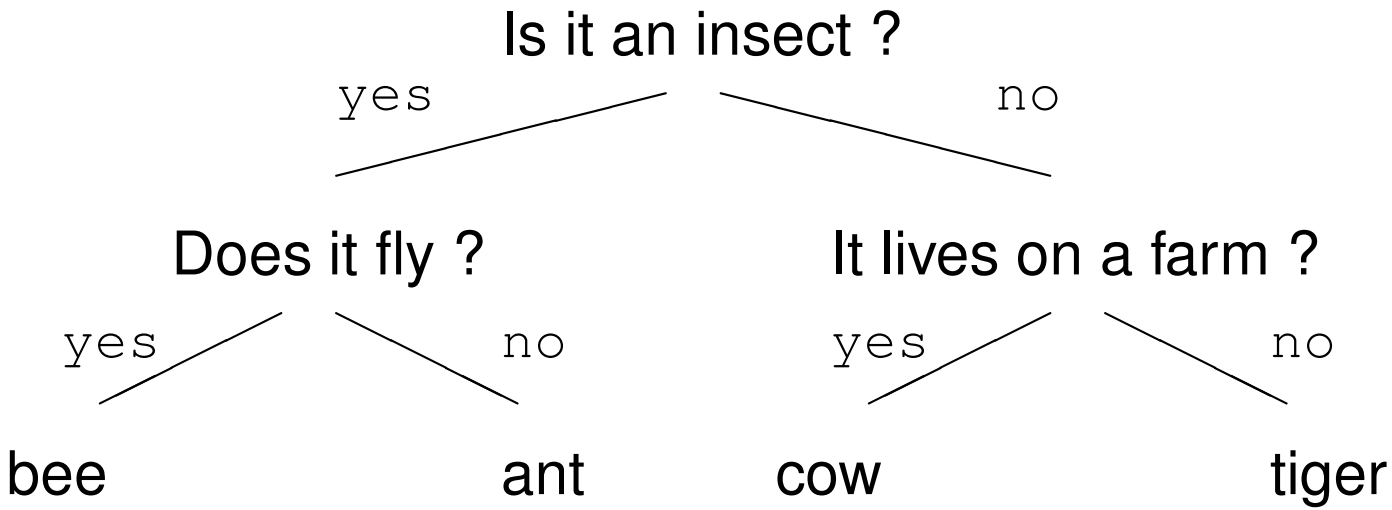
For example, a mileage table to store distances from Chicago to Miami, Los Angeles and New York:

```
>>> mt = { 'Miami':1237 , 'LA':2047 , 'NY':807 }
>>> mt['LA']
2047
>>> list(mt.keys())
['Miami', 'NY', 'LA']
>>> list(mt.values())
[1237, 807, 2047]
```

A dictionary is a set of `key:value` pairs.

classifying animals

Suppose we want to classify animals with simple questions with `yes` or `no` answers.



The leaves of the tree are just strings.

The internal nodes have questions as strings, and `yes` and `no` branches leading to more questions or to the names of the animals.

Recursive Data Structures

1. Binary Trees
   - sorting numbers
   - a recursive tree builder
   - flattening and searching
2. Classification Trees
   - lists and dictionaries
   - classifying with dictionary tree
   - adding elements to the tree
3. Assignments

classifying with a dictionary tree

```
$ python treezoo.py
What animal ? tiger
d = ['tiger']
continue ? (y/n) y
Is it "tiger" ? (y/n) n
What animal ? ant
Give question to distinguish "ant" from "tiger":
Is it an insect ?
d = {'q': 'Is it an insect ?', \
     'y': ['ant'], 'n': ['tiger']}
```

- Leaves in the tree are lists of one string.
- An internal node contains three keys:
  for the question, the yes and the no answer.

navigating through the tree

Continuing with the script `treezoo.py`:

```
ended construction, start navigation...
Is it an insect ? (y/n) y
Does it fly ? (y/n) y
arrived at "bee"
```

The answer of the user `'y'` or `'n'`
is the key to the branches of the tree.

Navigation algorithm:

- base case: length of dictionary is one
- general case: follow answer of user

the function `navigate`

```
def navigate(dic):
    """
    Navigates through the dictionary dic
    based on the user responses.
    """
    if len(dic) == 1:
        print('arrived at \"' + dic[0] + '\"')
    elif len(dic) == 3:
        ans = input(dic['q'] + ' (y/n) ')
        navigate(dic[ans])
```

Recursive Data Structures

1. Binary Trees
   - sorting numbers
   - a recursive tree builder
   - flattening and searching
2. Classification Trees
   - lists and dictionaries
   - classifying with dictionary tree
   - adding elements to the tree
3. Assignments

adding elements

Two base cases:

- for empty tree, we ask for animal name
- at leaf, we ask if it is the animal
  1. if yes, we are done
  2. if no, we ask name of new animal
     and a question to distinguish it

In the general case, we ask the question at the node and make a recursive call, adding to the branch `'y'` or `'n'`.

the function `add`, first base case

First base case: the tree is empty.

```
def add(dic):
    """
    Adds a new element to the dictionary dic,
    via interactive questions to the user.
    """
    if len(dic) == 0:
        ans = input('What animal ? ')
        return [ans]
```

the function `add`, second base case

Second base case: we are at a leaf.

```
    elif len(dic) == 1:
        qst = 'Is it \"' + dic[0] + '\" ? (y/n) '
        ans = input(qst)
        if ans == 'y':
            print('okay, got it')
            return dic
        else:
            ans = input('What animal ? ')
            ask = 'Give question to distinguish \"' + \
                  ans + '\" from \"' + dic[0] + '\":\n'
            qst = input(ask)
            return {'q':qst, 'y':[ans], 'n':[dic[0]]}
```

the function `add`, general case

```
else:
    ans = input(dic['q'] + ' (y/n) ')
    if ans == 'y':
        return {'q':dic['q'], \
            'y':add(dic['y']), \
            'n':dic['n']}
    else:
        return {'q':dic['q'], \
            'y':dic['y'], \
            'n':add(dic['n'])}
```

the main function

```
def main():
    """
    Builds interactively a tree to classify animals.
    """
    zoo = {}
    while True:
        zoo = add(zoo)
        print('zoo =', zoo)
        ans = input("continue ? (y/n) ")
        if ans != 'y':
            break
    print('ended construction, start navigation...')
    while True:
        navigate(zoo)
        ans = input("continue ? (y/n) ")
        if ans != 'y':
            break
```

Assignments

1. Use the code `add`, `flatten`, and `is_in` as methods in a class to represent trees to sort numbers.
2. Modify the representation of the tree to sort numbers, using a dictionary instead of a triplet.
   As keys use the strings `'data'`, `'smaller'`, and `'larger'`.
   The leaves of the tree have only one element: `'data':number`.
3. For the tree of dictionaries to classify animals, write a function which takes on input the tree and returns the list of all animal names in the tree.
4. Use `dbm` to store the tree of dictionaries to classify animals. Note that `dbm` requires all keys and values to be of type string.