Recursive Data Structures

Binary Trees
  sorting numbers
  a recursive tree builder
  flattening and searching

Classification Trees
  lists and dictionaries
  classifying with dictionary tree
  adding elements to the tree

Exercises and Assignments

MCS 275 Lecture 10
Programming Tools and File Management
Jan Verschelde, 6 February 2008
Recursive Data Structures

Binary Trees
	sorting numbers
	a recursive tree builder
	flattening and searching

Classification Trees

lists and dictionaries

classifying with dictionary tree

adding elements to the tree

Exercises and Assignments
Sorting Numbers using a Tree

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

Insert the numbers in a tree:

```
        4
       / \
      2   5
     /   / \
    1   3   8
     \
       7
```

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.
Sorting Numbers using a Tree

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

Insert the numbers in a tree:

```
  4
  / \
 2   5
 /   / \
1   3   8
     /   \
     7
```

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.
Sorting Numbers using a Tree

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

Insert the numbers in a tree:

```
  4
 / \
2   5
 /   /
1 3  8
   /
   7
```

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.
Sorting Numbers using a Tree

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

Insert the numbers in a tree:

```
          4
         / \
        2   5
       /     \
      1  3   8
            /  \
             7
```

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.
Sorting Numbers using a Tree

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

Insert the numbers in a tree:

```
  4
 / \
2   5
|   /
1   8
 \
   7
```

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.
Sorting Numbers using a Tree

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

Insert the numbers in a tree:

4
  2
    1
  3
5
  8
    7

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.
Sorting Numbers using a Tree

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

Insert the numbers in a tree:

```
        4
       / \  
      2   5
     / \ /  
    1  3 8  
        /  
        7   
```

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.
Sorting Numbers using a Tree

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

Insert the numbers in a tree:

```
        4
       / \
      2   5
     / \ /  \
    1  3 8  \
     /   /  \
   7    7
```

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.
Sorting Numbers using a Tree

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

Insert the numbers in a tree:

```
   4
  / \
 2   5
/ \ / \ 
1 3 8 7
```

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.
Sorting Numbers using a Tree

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

Insert the numbers in a tree:

```
        4
       / \
      2   5
     / \   \
    1   3  8
      \    
       7
```

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.
Sorting Numbers using a Tree

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

Insert the numbers in a tree:

```
      4
     / \
    2   5
   / \   \\
  1   3   8
      /   \
     7
```

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.
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

```
4
/
2 5
```

we use tuples, sequences enclosed by ( and ):

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

```
4
/   \
2     5
```

we use tuples, sequences enclosed by ( and ) :

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

\[
\text{4} \quad \begin{array}{c}
\text{2} \\
\text{5}
\end{array}
\]

we use tuples, sequences enclosed by ( and ):

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

$$
\begin{array}{c}
  4 \\
  2 \quad 5
\end{array}
$$

we use tuples, sequences enclosed by ( and ):

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

```
4
/ \
2   5
```

we use tuples, sequences enclosed by ( and ) :

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

```
2
/   \
4   5
```

we use tuples, sequences enclosed by ( and ):

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

```
        4
       / \
      2   5
```

we use tuples, sequences enclosed by ( and ):

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

$ python treesort.py

give 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]
Incremental Sorting

$ python treesort.py
give 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]
Incremental Sorting

```
$ python treesort.py
give 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]
```
Incremental Sorting

```
$ python treesort.py
give 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]
```
Incremental Sorting

$ python treesort.py

give 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]
Incremental Sorting

$ python treesort.py

give 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

Binary Trees
- sorting numbers
- a recursive tree builder
- flattening and searching

Classification Trees
- lists and dictionaries
- classifying with dictionary tree
- adding elements to the tree

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

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

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

Binary Trees
  sorting numbers
  a recursive tree builder
  flattening and searching

Classification Trees
  lists and dictionaries
  classifying with dictionary tree
  adding elements to the tree

Exercises and 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 $L$
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 $L$
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 $L$
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 \( L \)
Flattening the Tree

The tree $T$

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

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 $L$
**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 $L$
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 \( L \)
the Function Flatten

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

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

    if len(T) == 0:
        return []
    else:
        L = Flatten(T[0])
        L.append(T[1])
        L = L + Flatten(T[2])
        return 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 than the data at $T$, return the result of search in left branch otherwise return result of search in right branch
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 than the data at $T$, return the result of search in left branch
- otherwise return result of search in right branch
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 than the data at $T$, return the result of search in left branch otherwise return result of search in right branch
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 than the data at $T$, return the result of search in left branch
- otherwise return result of search in right branch
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:

1. if \( T \) is empty, we return \( False \)
2. if the data at the node is \( n \), return \( True \)
3. 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
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:

1. if \( T \) is empty, we return False
2. if the data at the node is \( n \), return True
3. if \( n \) is less than the data at \( T \), return the result of search in left branch
4. otherwise return result of search in right branch
the Function IsIn

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

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

```python
def IsIn(T, n):
    """
    Returns True if n belongs to the tree, returns False otherwise.
    """
    if len(T) == 0:
        return False
    elif T[1] == n:
        return True
    elif n < T[1]:
        return IsIn(T[0], n)
    else:
        return IsIn(T[2], n)
```
def IsIn(T, n):
    """
    Returns True if n belongs to the tree, returns False otherwise.
    """
    if len(T) == 0:
        return False
    elif T[1] == n:
        return True
    elif n < T[1]:
        return IsIn(T[0], n)
    else:
        return IsIn(T[2], n)
The Main Program

def main():
    """
    Prompts the user for numbers and sorts, using a tree: a triple of triplets.
    """
    T = ()
    while True:
        n = input('give number (-1 to stop) : ')
        if n < 0: break
        if IsIn(T,n):
            print n, 'is already in T'
        else:
            T = Add(T,n)
        print 'T =', T
    print 'sorted numbers =', Flatten(T)
The Main Program

def main():
    ""
    Prompts the user for numbers and sorts, using a tree: a triple of triplets.
    ""
T = ()
while True:
    n = input('give number (-1 to stop) : ')  
    if n < 0: break
    if IsIn(T,n):
        print n, 'is already in T'
    else:
        T = Add(T,n)
    print 'T =', T
print 'sorted numbers =', Flatten(T)
The Main Program

def main():
    """
    Prompts the user for numbers and sorts, using a tree: a triple of triplets.
    """
    T = ()
    while True:
        n = input('give number (-1 to stop) : ')
        if n < 0: break
        if IsIn(T, n):
            print n, 'is already in T'
        else:
            T = Add(T, n)
        print 'T =', T
    print 'sorted numbers =', Flatten(T)
Recursive Data Structures

Binary Trees
- sorting numbers
- a recursive tree builder
- flattening and searching

Classification Trees
- lists and dictionaries
- classifying with dictionary tree
- adding elements to the tree

Exercises and 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:

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

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

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

A dictionary is a set of `key: value` pairs.
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 st ruct in C).

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

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

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

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

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

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

Is it an insect?
- yes
  - Does it fly?
    - yes: bee
    - no: ant
  - no
    - It lives on a farm?
      - yes: cow
      - no: tiger

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.
Classifying Animals

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

```
Is it an insect?  
/    \       
yes   no       
/        
Does it fly?  It lives on a farm?  
/    \        /    \  
yes   no    yes   no  
/    \        /    \  
bee   ant    cow   tiger
```

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.
Classifying Animals

Suppose we want to classify animals with simple questions with \textit{yes} or \textit{no} answers.

- **Is it an insect?**
  - yes
  - no
    - **Does it fly?**
      - yes
      - no
        - bee
        - ant
    - It lives on a farm?
      - yes
      - no
        - cow
        - tiger

The leaves of the tree are just strings.

The internal nodes have questions as strings, and \textit{yes} and \textit{no} branches leading to more questions or to the names of the animals.
Classifying Animals

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

Is it an insect?

Does it fly?

It lives on a farm?

bee

ant
cow
tiger

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.
Classifying Animals

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

Is it an insect?
- **yes**
  - Does it fly?
    - **yes**
      - bee
    - **no**
      - ant
- **no**
  - It lives on a farm?
    - **yes**
      - cow
    - **no**
      - tiger

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.
Classifying Animals

Suppose we want to classify animals with simple questions with \textit{yes} or \textit{no} answers.

\begin{center}
\begin{tikzpicture}
  \node {Is it an insect ?}
  \child {node {Does it fly ?} \edge {yes};}
  \child {node {It lives on a farm ?} \edge {no};}
  \child {node {yes} \edge {bee};}
  \child {node {no} \edge {ant};}
  \child {node {yes} \edge {cow};}
  \child {node {no} \edge {tiger};}
\end{tikzpicture}
\end{center}

The leaves of the tree are just strings.

The internal nodes have questions as strings, and \textit{yes} and \textit{no} branches leading to more questions or to the names of the animals.
Classifying Animals

Suppose we want to classify animals with simple questions with \text{yes} or \text{no} answers.

\begin{itemize}
  \item \text{Is it an insect ?}
  \item \text{Does it fly ?}
  \item \text{It lives on a farm ?}
\end{itemize}

- \text{bee}
- \text{ant}
- \text{cow}
- \text{tiger}

The leaves of the tree are just strings.

The internal nodes have questions as strings, and \text{yes} and \text{no} branches leading to more questions or to the names of the animals.
Classifying Animals

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

Is it an insect?

yes

Does it fly?

yes

bee

no

ant

It lives on a farm?

yes

cow

no

tiger

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.
Classifying Animals

Suppose we want to classify animals with simple questions with \textit{yes} or \textit{no} answers.

Is it an insect?
\begin{itemize}
  \item [yes] Does it fly?
    \begin{itemize}
      \item [yes] bee
      \item [no] ant
    \end{itemize}
  \item [no] It lives on a farm?
    \begin{itemize}
      \item [yes] cow
      \item [no] tiger
    \end{itemize}
\end{itemize}

The leaves of the tree are just strings.

The internal nodes have questions as strings, and \textit{yes} and \textit{no} branches leading to more questions or to the names of the animals.
Recursive Data Structures

Binary Trees
  sorting numbers
  a recursive tree builder
  flattening and searching

Classification Trees
  lists and dictionaries
  classifying with dictionary tree
  adding elements to the tree

Exercises and Assignments
Classifying with Dictionary Tree

```python
$ 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.
Classifying with 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.
Classifying with 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.
$ 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
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
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
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
Navigating through the Tree

Continuing with the script `treezoo.py`:

decided to move the next step, 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(d):
    """
    Navigates through the dictionary based on the user responses.
    """
    if len(d) == 1:
        print 'arrived at "' + d[0] + '"'
    elif len(d) == 3:
        a = raw_input(d['q'] + ' (y/n) ')
        Navigate(d[a])
the Function Navigate

def Navigate(d):
    
    """
    Navigates through the dictionary based on the user responses.
    """

    if len(d) == 1:
        print 'arrived at "' + d[0] + '"'
    elif len(d) == 3:
        a = raw_input(d['q'] + ' (y/n) ')
        Navigate(d[a])
def Navigate(d):
    """
    Navigates through the dictionary based on the user responses.
    """
    if len(d) == 1:
        print 'arrived at \"' + d[0] + '\"'
    elif len(d) == 3:
        a = raw_input(d['q'] + ' (y/n) ')
        Navigate(d[a])
Recursive Data Structures

Binary Trees
- sorting numbers
- a recursive tree builder
- flattening and searching

Classification Trees
- lists and dictionaries
- classifying with dictionary tree
- adding elements to the tree

Exercises and 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'.
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'.
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, part I

def Add(d):
    """
    Adds a new element to the dictionary, via interactive questions to the user.
    """
    if len(d) == 0:
        a = raw_input('What animal ? ')
        return [a]
    elif len(d) == 1:
        q = 'Is it "' + d[0] + '" ? (y/n) '
        a = raw_input(q)
        if a == 'y':
            print 'okay, got it'
            return d
def Add(d):
    """
    Adds a new element to the dictionary, via interactive questions to the user.
    """

    if len(d) == 0:
        a = raw_input('What animal ? ')
        return [a]
    elif len(d) == 1:
        q = 'Is it "' + d[0] + '" ? (y/n) '
        a = raw_input(q)
        if a == 'y':
            print 'okay, got it'
            return d
def Add(d):
    """
    Adds a new element to the dictionary, via interactive questions to the user.
    """

    if len(d) == 0:
        a = raw_input('What animal ? ')
        return [a]
    elif len(d) == 1:
        q = 'Is it "' + d[0] + '" ? (y/n) '
        a = raw_input(q)
        if a == 'y':
            print 'okay, got it'
        return d
else:
    a = raw_input('What animal ? ')
    s = 'Give question to distinguish "' + a + '" from "' + d[0] + '": 
    q = raw_input(s)
    return {'q':q,'y':[a],'n':[d[0]]}

else:
    a = raw_input(d['q'] + ' (y/n) ')
    if a == 'y':
        return {'q':d['q'],'y':Add(d['y']),'n':d['n']}
    else:
        return {'q':d['q'],'y':d['y'],'n':Add(d['n'])}
else:
    a = raw_input('What animal?')
    s = 'Give question to distinguish "' + a + '" from "' + d[0] + '":
    q = raw_input(s)
    return {'q':q,'y':[a],'n':[d[0]]}

else:
    a = raw_input(d['q'] + ' (y/n) ')
    if a == 'y':
        return {'q':d['q'],'y':Add(d['y']),'n':d['n']}
    else:
        return {'q':d['q'],'y':d['y'],'n':Add(d['n'])}
The Main Program

def main():
    """
    Build a tree to classify animals.
    """
    d = {}
while True:
    d = Add(d)
    print 'd =', d
    a = raw_input("continue ? (y/n) ")
    if a != 'y': break
print 'ended construction, start navigation...'
while True:
    Navigate(d)
    a = raw_input("continue ? (y/n) ")
    if a != 'y': break
def main():
    ""
    Build a tree to classify animals.
    ""
    d = {}
    while True:
        d = Add(d)
        print 'd =', d
        a = raw_input("continue ? (y/n) ")
        if a != 'y': break
    print 'ended construction, start navigation...'
    while True:
        Navigate(d)
        a = raw_input("continue ? (y/n) ")
        if a != 'y': break
Exercises and Assignments

1. Use the code `Add`, `Flatten`, and `IsIn` 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.

Homework collected on Friday 8 February:

- exercises 2 and 3 of Lecture 8,
- exercises 2 and 5 of Lecture 9,
- exercise 2 of Lecture 10.