TODAY

- Review: Immutable Trees
- Binary Trees and Binary Search Trees
- Extra: Tuples versus IRLists

REVIEW: HIERARCHICAL DATA

Often, we find that information is nicely organized into hierarchies.

Example: Writing!

REVIEW: TREES

A tree data structure traditionally has two parts:

1. A datum: Information stored at the top.
2. Some children: Trees that appear below this tree.

Notice that trees are also recursively defined. A tree is made from other trees – these trees are its subtrees.
**REVIEW: TREES**

- A is a parent to B and C
- B and C are children of A

**REVIEW: TREES**

- Root
- Nodes
- Branches
- Fib(4)
- Fib(3)
- Fib(2)
- Fib(1)
- Fib(0)
- Leaves

**REVIEW: IMMUTABLE TREES**

```
make_itree(1, 
    (make_itree(2),
     make_itree(3),
     (make_itree(4),
      make_itree(5))),
    make_itree(6, 
      (make_itree(7),))))
```

**REVIEW: IMMUTABLE TREES**

```
itree_datum(fir) is 1
itree_children(fir) is the tuple
(2, 3, 6)
```

**REVIEW: ITREES**

```
def make_itree(datum, children=()):
    return (datum, children)

def itree_datum(t):
    return t[0]

def itree_children(t):
    return t[1]
```

**REVIEW: OPERATING ON ITREES**

Write the function `itree_prod`, which takes an iTree of numbers and returns the product of all the numbers in the iTree.

```python
>>> t = make_itree(1, (make_itree(2),
                       make_itree(3),
                       make_itree(4)))
>>> itree_prod(t)
24
```
**EXAMPLE: OPERATING ON iTREES**

*Idea:* Split the problem into two different parts: one that handles a single tree and one that handles a forest.

```python
def itree_prod(t):
    return itree_datum(t) * forest_prod(itree_children(t))

def forest_prod(f):
    if len(f) == 0:
        return 1
    return itree_prod(f[0]) * forest_prod(f[1:])
```

**EXAMPLE: OPERATING ON iTREES**

*Idea:* Split the problem into two different parts: one that handles a single tree and one that handles a forest.

```python
def itree_prod(t):
    return itree_datum(t) * forest_prod(itree_children(t))

def forest_prod(f):
    if len(f) == 0:
        return 1
    return itree_prod(f[0]) * forest_prod(f[1:])
```

**EXAMPLE: OPERATING ON iTREES**

*Idea:* Split the problem into two different parts: one that handles a single tree and one that handles a forest.

```python
def itree_prod(t):
    return itree_datum(t) * forest_prod(itree_children(t))

def forest_prod(f):
    if len(f) == 0:
        return 1
    return itree_prod(f[0]) * forest_prod(f[1:])
```

**EXAMPLE: OPERATING ON iTREES**

*Idea:* Split the problem into two different parts: one that handles a single tree and one that handles a forest.

```python
def itree_prod(t):
    return itree_datum(t) * forest_prod(itree_children(t))

def forest_prod(f):
    if len(f) == 0:
        return 1
    return itree_prod(f[0]) * forest_prod(f[1:])
```

**MUTUAL RECURSION**

To find the product of the whole iTree in *itree_prod*, we need to apply *itree_prod* itself to the smaller subtrees that are the children of the provided iTree. *forest_prod* does this for us.

**PRACTICE: OPERATING ON iTREES**

Write a function `scale_itree` that takes an iTree of numbers and a scaling factor, and returns a new iTree that results from scaling each number in the original iTree.

```
scale_itree(t, 2) =
```

```
7 5 3 1
8 6 4 2

scale_itree:
```
**Practice: Operating on ITrees**

```python
def scale_itree(itree, factor):
    return make_itree(itree_datum(itree) * factor,
                      scale_forest(itree_children(itree),
                      factor))

def scale_forest(forest, factor):
    results = ()
    for itree in forest:
        results = results + scale_itree(itree, factor)
    return results
```

**Alternative solution:**

```python
def scale_itree(itree, factor):
    return make_itree(itree_datum(itree) * factor,
                      scale_forest(itree_children(itree),
                      factor))

def scale_forest(forest, factor):
    if len(forest) == 0:
        return ()
    return (scale_itree(forest[0], factor),) + \
           scale_forest(forest[1:], factor)
```

**Practice: Operating on ITrees**

Write a function `count_leaves` that counts the number of leaves in the iTree provided.

```python
def count_leaves(itree):
    return len(list(filter(lambda x: x == 'leaf',
                            tuple(map(scale_itree,
                                       itree_children(itree))))))
```
PRACTICE: OPERATING ON ITREES

```python
def count_leaves(itree):
    if _____________:
        return 1
    return cl_forest(_______________________)

def cl_forest(f):
    if len(f) == 0:
        return
    return _______________________
```

ANNOUNCEMENTS

- Homework 6 is due tomorrow, **July 10**.
  - Starting with the next homework, we will mark questions as core and reinforcement.
  - The core questions are ones that we suggest you work on to understand the idea better.
  - The reinforcement questions are extra problems that you can practice with, but are not as critical.
- Project 2 is due on Friday, **July 13**.
- Homework 0 for the staff is now available.
**PROBLEM SOLVING: SEARCHING**

*Searching* is a problem that shows up frequently in many different (and daily!) settings.

**Problem:** We have a collection of data and we need to find a certain datum.

Examples:
- Searching for a person in a phone book.
- Searching for a webpage on the Internet.
- ... and so on.

```python
def has_num(tup, num_to_find):
    for item in tup:
        if item == num:
            return True
    return False
```

**PROBLEM SOLVING: SEARCHING**

Write a predicate function `has_num` that determines if a number is in a tuple of numbers.

```python
def has_num(tup, num_to_find):
    for item in tup:
        if item == num:
            return True
    return False
```

**PROBLEM SOLVING: SEARCHING**

How does the running time of `has_item` scale as $n$, the length of the tuple, scales?

*Can we do better?*
**Trees in Real Life: Binary Trees**

Trees where each node has at most two children are known as **binary trees**.

![Binary Trees Diagram](image)

**Trees in Real Life: Binary Search Trees**

**Binary search trees** are binary trees where all of the items to the left of a node are **smaller**, and the items to the right are **larger**.

![Binary Search Trees Diagram](image)

Let us try to find 4 in this binary search tree:

- It could be anywhere here.

Let us try to find 14 in this binary search tree:

- It could be anywhere here.
- It should be present in the BST.

Let us try to find 4 in this binary search tree:

- It could be anywhere here.

Let us try to find 4 in this binary search tree:

- Discard this fact of the page. It can't be here! They're too big.
TREES IN REAL LIFE: BINARY SEARCH TREES

Let us try to find 14 in this binary search tree:

10

4
2 6
12
14

TREES IN REAL LIFE: BINARY SEARCH TREES

Let us try to find 14 in this binary search tree:

10

4
2 6
12
14

TREES IN REAL LIFE: BINARY SEARCH TREES

Let us try to find 1 in this binary search tree:

10

4
2 6
12
14

TREES IN REAL LIFE: BINARY SEARCH TREES

Let us try to find 1 in this binary search tree:

10

4
2 6
12
14

TREES IN REAL LIFE: BINARY SEARCH TREES

Let us try to find 1 in this binary search tree:

10

4
2 6
12
14

TREES IN REAL LIFE: BINARY SEARCH TREES

Let us try to find 1 in this binary search tree:

10

4
2 6
12
14

TREES IN REAL LIFE: BINARY SEARCH TREES

Let us try to find 1 in this binary search tree:

10

4
2 6
12
14

TREES IN REAL LIFE: BINARY SEARCH TREES

Let us try to find 1 in this binary search tree:

10

4
2 6
12
14
Trees in Real Life: Binary Search Trees

Let us try to find 11 in this binary search tree:

```
   10
   /\  
  4  12
 /   /
2    6
     /
    14
```

Trees in Real Life: Binary Search Trees

Two important things when searching for an item in a binary search tree (BST):

1. As we go down the tree, from one level to another, from parent to child, we discard **approximately half** the data each time.
2. In the worst case, we go down all the way to the leaves.

Trees in Real Life: Binary Search Trees

The **height** of a tree is the length of the **longest path** from the root to any leaf.

If there are \( n \) nodes in a binary search tree, and if the binary search tree is balanced, then the height of the tree is approximately \( \log_2 n \).

LOGARITHMIC ALGORITHM

Why approximately \( \log_2 n \)?

For a tree of height \( h \) (or \( h + 1 \) levels), the total number of nodes is

\[
1 + 2 + 4 + 8 + \ldots + 2^h = n
\]

or, \( 2^{h+1} - 1 = n \)

or, \( h \approx \log_2(n) \)

LOGARITHMIC ALGORITHM

Why approximately \( \log_2 n \)?

Alternatively, notice that when we go down one level, we are discarding **approximately half** the data.
TREES IN REAL LIFE: BINARY SEARCH TREES

In the worst case, we go down to the leaves, visiting each level of nodes once.
There are approximately \( \log_2 n \) levels.
The runtime to find an item in a binary search tree (BST) with \( n \) nodes is \( O(\log n) \).

BST ADT

```python
empty_bst = None
def make_bst(datum, left=empty_bst, right=empty_bst):
    return (left, datum, right)
def bst_datum(b):
    return b[1]
def bst_left(b):
    return b[0]
def bst_right(b):
    return b[2]
```

BST ADT

```python
def bst_is_leaf(b):
    return bst_left(b) == empty_bst and bst_right(b) == empty_bst
```

BST ADT

```python
def has_num(b, num):
    if b == empty_bst:
        return False
    elif bst_datum(b) == item:
        return True
    elif bst_datum(b) > item:
        return has_num(bst_left(b), num)
    return has_num(bst_right(b), num)
```

CONCLUSION

- Trees are useful for representing general tree-structures.
- Binary search trees allow us to search through data faster.
- Preview: Object-oriented programming: a brand new paradigm.
GOOD LUCK TONIGHT!

WHO'S AWESOME?

EXTRA: TUPLES VERSUS IRLISTS

Tuples and IRLists are two different ways of representing lists of items: they are two different data structures, each with their own advantages and disadvantages.

You will compare them further in CS61B, but here is a preview.

EXTRA: TUPLES VERSUS IRLISTS

IRLists are recursively defined, tuples are not.

This gives IRLists an advantage when you want to perform a naturally recursive task, such as mapping a function on a sequence.

EXTRA: TUPLES VERSUS IRLISTS

Which of the following methods is better?

```python
def map_irlist(fn, irl):
    return make_irlist(fn(irlist_first(irl)),
                       map_irlist(fn, irlist_rest(irl)))
```

```python
def map_tuple(fn, tup):
    results = ()
    for item in tup:
        results = results + (fn(item),)
    return results
```

EXTRA: TUPLES VERSUS IRLISTS

map_irlist is better than map_tuple.

map_tuple creates and discards temporary tuples, since tuples are immutable.

map_irlist “tacks on” new items to the front of an existing (and growing) IRList.

EXTRA: TUPLES VERSUS IRLISTS

Tuples are unchanging sequences of data, which gives them an advantage when all you want to do is grab information from a sequence.
**EXTRA: TUPLES versus IRLISTS**

Which of the following methods is better?

```python
def getitem_tuple(tup, pos):
    return tup[pos]

def getitem_irlist(irl, pos):
    if pos == 0:
        return irlist_first(irl)
    return getitem_irlist(irlist_rest(irl), pos-1)
```

**EXTRA: TUPLES versus IRLISTS**

getitem_tuple is better than getitem_irlist.

Indexing into a tuple takes (on average) constant time, but indexing into an IRList takes linear time.