CS 61A Lecture 11: Trees

Tammy Nguyen (tammynguyen@berkeley.edu)
Thursday, 07/07

Announcements

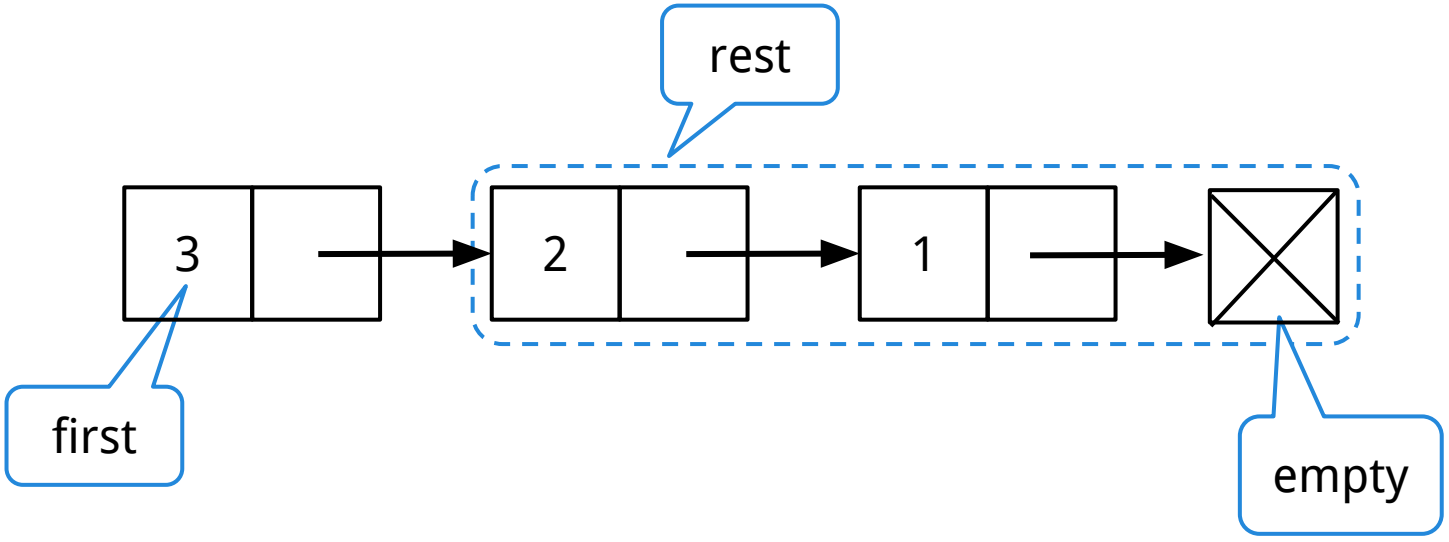
- Quiz 2 grades were released Tuesday afternoon on Gradescope. Regrade requests are open until tonight.
- **Project 2** is due July 12.
  - Submit by July 12 to earn one extra credit point.
  - Run python3 ok --submit to check against hidden tests.
  - Check your submission at ok.cs61a.org.
  - Invite your partner (watch this video).
- **Homework 4** is due July 7
- **Quiz 4** will be released 9am on Monday, July 11 and due by 10am on Tuesday, July 12.
- There will be no written quiz next Thursday, since the midterm is that day.
- The **61A Potluck** is this Friday, July 8. Join us in the Wozniak Lounge from 5-8pm. Bring food and board games!

Agenda

- Linked list review
- Trees
  - Terminology
  - Abstract data type
  - Processing
    - `contains_entry`
    - `average_entry`
  - Implementations

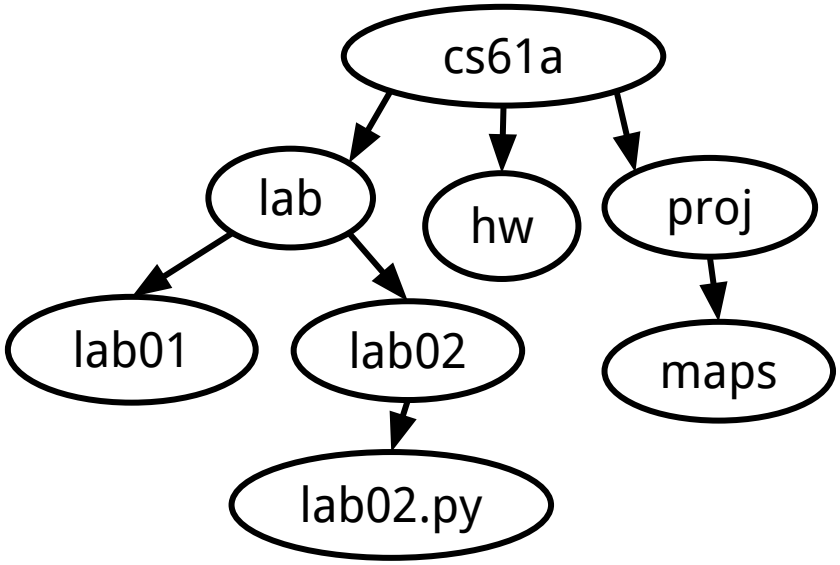
Linked List Review

- A linked list is a sequence of links.
- Each `link` contains a value, `first`, and a reference to the next link, `rest`.
- `empty` represents an empty linked list.



Hierarchy

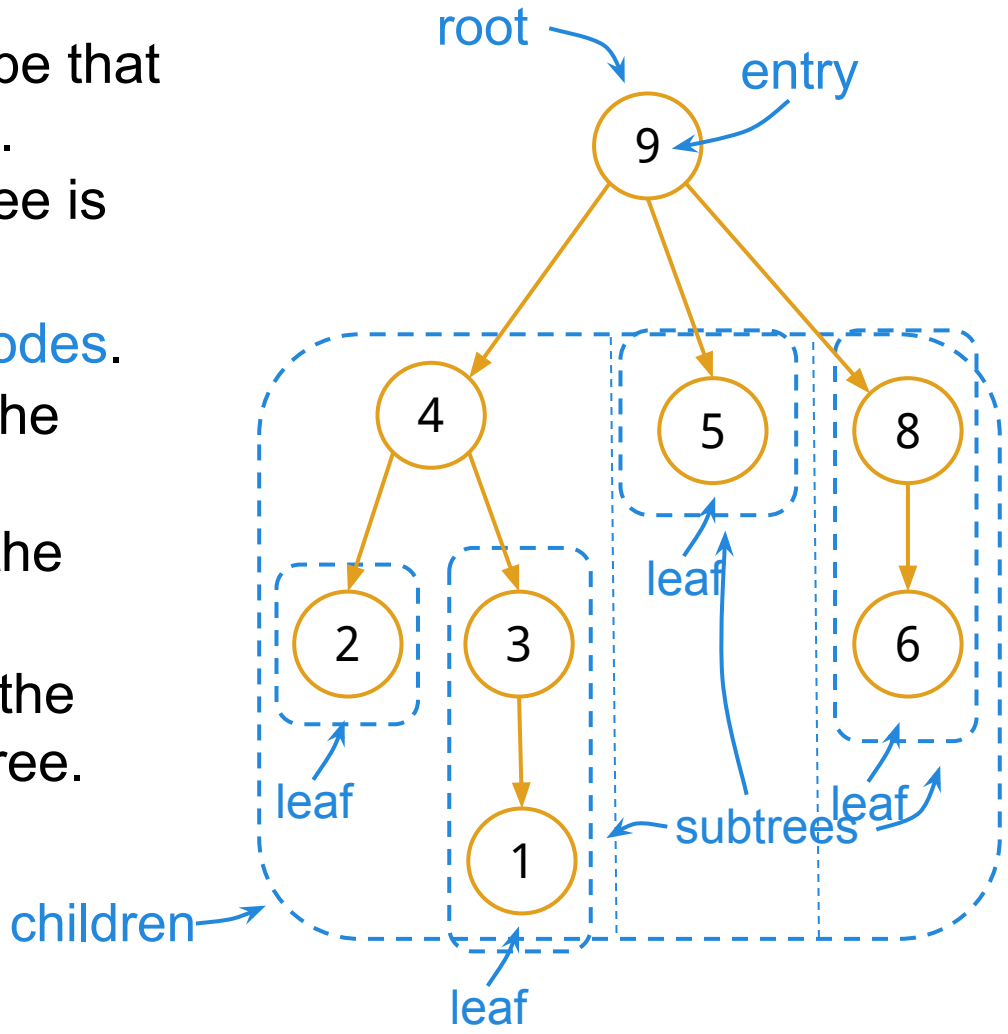
- Lists are useful for representing a single ordered sequence of values.
- Data like a file system or family lineage are not linear.
- How can we represent data with hierarchical relationships?



trees!

Trees: Terminology

- A tree is an abstract data type that represents hierarchical data.
- It is defined recursively: a tree is made up of subtrees.
- The data are contained in nodes.
- The node at the very top is the root.
- The value inside the root is the entry of the tree.
- The subtrees directly under the root are the children of the tree.
- Nodes without children are called leaves.

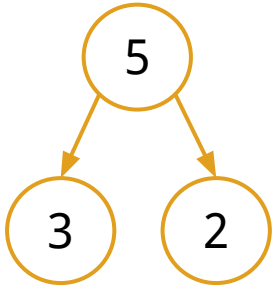


Trees: ADT

- **Constructor:**
  - `tree(entry, children=[])`
  - *`entry`*: the data value to put in the root of the tree
  - *`children`*: a list of trees immediately under the root, defaults to an empty list

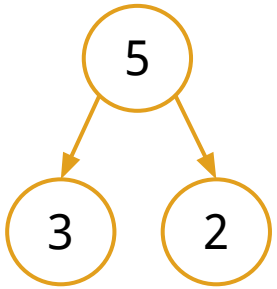
We define a tree recursively. Instead of specifying all of the entries in a tree in the constructor, we specify the **entry at the root** and a **list of the children of the root** (which have their own entries and children…).

```
tree(5, [tree(3), tree(2)])
```



Trees: ADT

- **Selectors:**
  - `entry(t)`: returns the entry in the root of `t`
  - `children(t)`: returns a list containing the children of the root of `t`

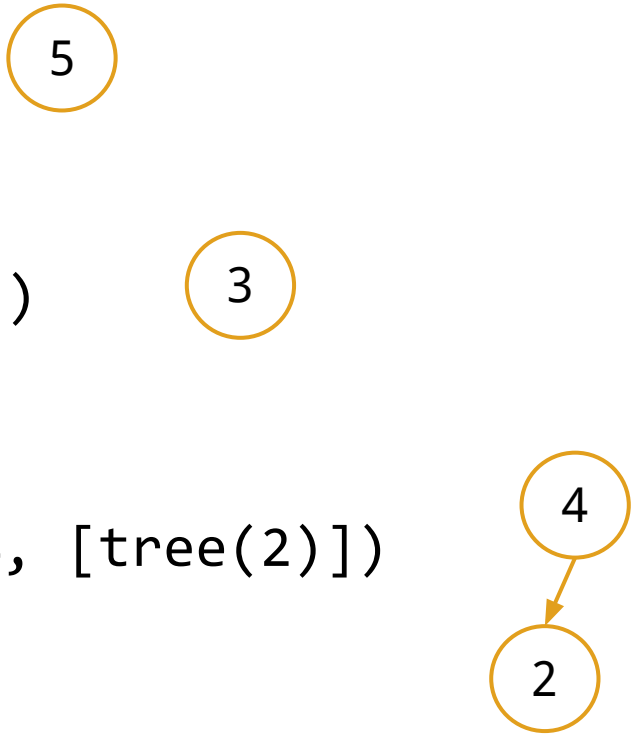



```
>>> t = tree(5, [tree(3), tree(2)])
>>> entry(t)
5
>>> [entry(child) for child in children(t)]
[3, 2]
```

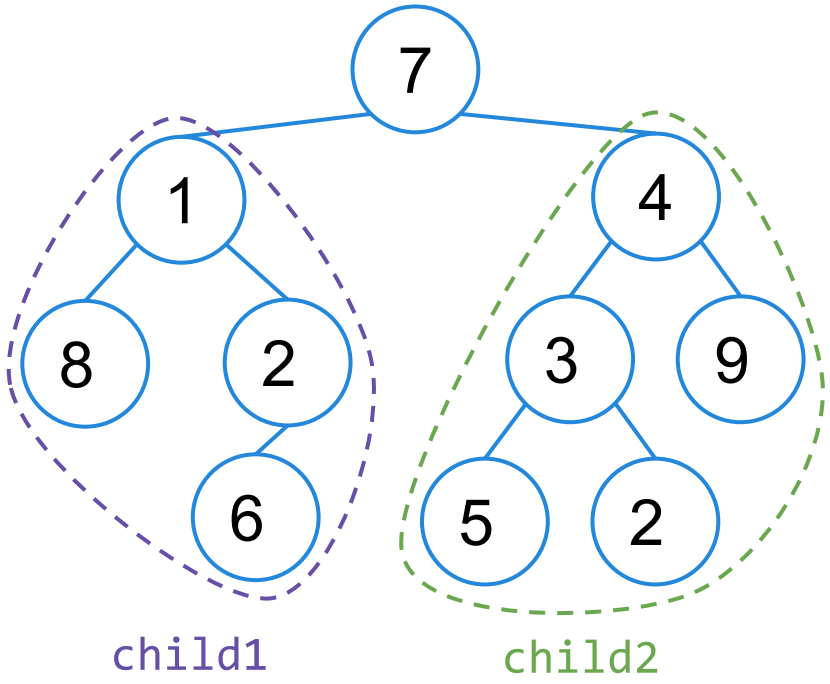
Trees: ADT

- **Convenience function:**
  - `is_leaf(t)`: returns True if t is a leaf

```
>>> leaf1 = tree(5)
>>> is_leaf(leaf1)
True
>>> leaf2 = tree(3, [])
>>> is_leaf(leaf2)
True
>>> has_child = tree(4, [tree(2)])
>>> is_leaf(has_child)
False
```

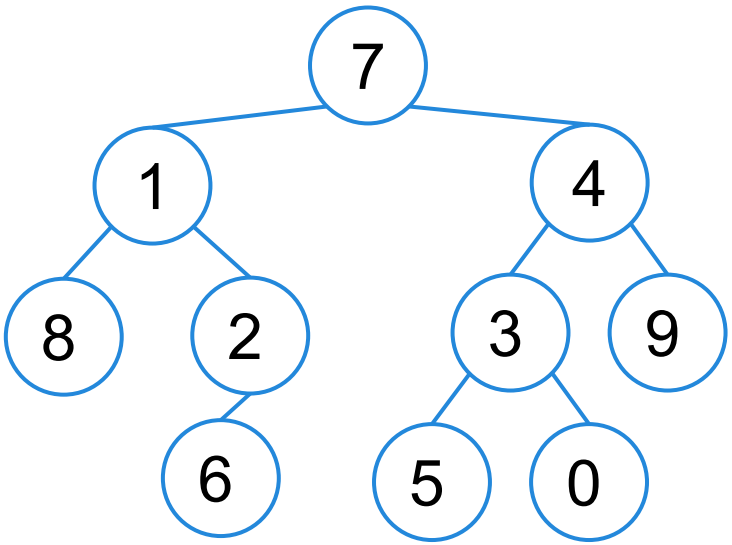


Trees: ADT




```
>>> child1 = tree(1,
                  [tree(8),
                   tree(2,
                      [tree(6)])])
>>> child2 = tree(4,
                  [tree(3,
                       [tree(5),
                        tree(2)])
                   tree(9)])
>>> t = tree(7, [child1, child2])
```

Trees: ADT



Try it out!

This tree is bound to the name **`t`**.
How can I use the selectors `entry` and `children` to get the following entries?

```
>>> entry(children(t)[1])
4
>>> entry(t)
7
>>> entry(children(children(t)[0])[1])
2
>>> [entry(c) for c in children(t)]
[1, 4]
```

Common mistakes

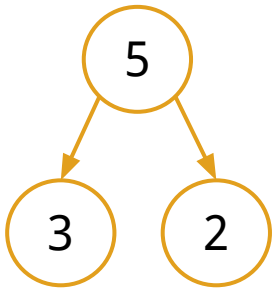
Our ADT requires that the tree is represented as the **entry at the root** and the **children of the root.**

More specifically, the children are represented as a **list of trees**.

Here are some common mistakes when using the constructor and selectors:

Passing in trees to the constructor:

```
>>> tree(5, tree(3), tree(2))
```



Passing children in as entries instead of trees:

```
>>> tree(5, [3, 2])
```

Treating the children of a tree as a single tree instead of as a list:

```
>>> entry(children(t))
```

Trees: Processing

Some common tree operations:

right now!

Finding whether a specific entry is in a tree.

Summarizing the data in a tree.

Finding a path from the root to some entry.

Mapping a function onto each entry.

Finding the size or height of a tree.

Pruning a tree (getting rid of some nodes).

… and more!

today's discussion.

Trees: Processing

- Remember, trees are defined recursively. A tree is composed of a bunch of subtrees.
- This makes a tree very easy to process using recursion!
  - **Base case(s):**
    - simplest tree is a tree with no children, i.e. a leaf.
    - account for the root (?)
  - **Recursive call:** call the function on each of the tree's children.

A simplified general procedure:

1. Write a base case for a leaf (usually).
2. Process the root (which might be additional base case(s)).
3. Recurse on each of the children.
4. Combine/use the result of the recursive calls to solve the problem.

Trees: contains_entry

Let's write a function to find whether a tree contains some entry.

Step 1: Write a base case for a leaf.

A leaf only requires one check. If its entry is the entry we are looking for, return `True`. Otherwise, return `False`.

Step 2: Process the root.

If the entry at the root is the node we are looking for, we can return `True`.

We *cannot* return `False` otherwise, since it can still be in the children.

```
def contains_entry(t, e):
    "*** YOUR CODE HERE ***"
    if is_leaf(t) and entry(t) == e:
        return True
    if is_leaf(t) and entry(t) != e:
        return False
    if entry(t) == e:
        return True
```

The first and third case can be combined. It doesn't matter if `t` is a leaf, if its `entry` is `e`, return `True`.

Trees: contains_entry

Let's write a function to find whether a tree contains some entry.

Step 1: Write a base case for a leaf.

A leaf only requires one check. If its entry is the entry we are looking for, return `True`. Otherwise, return `False`.

Step 2: Process the root.

If the entry at the root is the node we are looking for, we can return `True`.

We *cannot* return `False` otherwise, since it can still be in the children.

```
def contains_entry(t, e):
    "*** YOUR CODE HERE ***"
    if entry(t) == e:
        return True
    if is_leaf(t):
        return False
```

The first and third case can be combined. It doesn't matter if `t` is a leaf, if its `entry` is `e`, return `True`.

Also, since we check for *equality* in that case, we don't need to check for *inequality* in the second case.

Trees: contains_entry

Step 3: Recurse on each of the children.

Step 4: Combine/use results of recursive calls.

`contain_entry(child, e)` will return `True` if `child` contains `e` and `False` otherwise.

If ***any*** child contains `e`, then the whole tree `t` contains `e`, i.e. we can return `True` immediately.

If ***no*** child contains `e`, then the whole tree does not contain `e`, as we've already checked the root.

```
def contains_entry(t, e):
    "*** YOUR CODE HERE ***"
    if entry(t) == e:
        return True
    if is_leaf(t):
        return False
    for child in children(t):
        contains_entry(child, e)
```

Trees: contains_entry

Step 3: Recurse on each of the children.

Step 4: Combine/use results of recursive calls.

`contain_entry(child, e)` will return `True` if `child` contains `e` and `False` otherwise.

If ***any*** child contains `e`, then the whole tree `t` contains `e`, i.e. we can return `True` immediately.

If ***no*** child contains `e`, then the whole tree does not contain `e`, as we've already checked the root.

```
def contains_entry(t, e):
    "*** YOUR CODE HERE ***"
    if entry(t) == e:
        return True
    if is_leaf(t):
        return False
    for child in children(t):
        if contains_entry(child, e):
            return True
    return False
```

Trees: average_entry

Suppose we want to find the average entry in an tree.

We cannot use the average entry of a child to find the average entry of an entire tree.

Instead, think about what information you need about a tree to find its average.

Write a helper function to find this information, and then solve the problem.

```
def average_entry(t):
    “*** YOUR CODE HERE ***”
    def helper(t):
```

Trees: average_entry

We need to know the total sum of all entries in the tree and the total number of nodes in the tree.

Let's fill in the helper function so it returns that!

Step 1: Write a base case for a leaf.

A leaf sums to its entry, and it counts as one node.

Step 2: Process the root.

Include the entry at the root in total sum of entries, and add one to the count of nodes.

```
def average_entry(t):
    "*** YOUR CODE HERE ***"
    def helper(t):
        if is_leaf(t):
            return entry(t), 1
        total, count = entry(t), 1
```

Trees: average_entry

Step 3: Recurse on each of the children.

Step 4: Combine/use results of recursive calls.

`helper(c)` will return the total sum of entries and the number of nodes in `c`.

We simply need to add these amounts to the running total and count.

```
def average_entry(t):
    "*** YOUR CODE HERE ***"
    def helper(t):
        if is_leaf(t):
            return entry(t), 1
        total, count = entry(t), 1
        for c in children(t):
            helper(c)
```

Trees: average_entry

Step 3: Recurse on each of the children.

```
def average_entry(t):
    "*** YOUR CODE HERE ***"
    def helper(t):
```

Step 4: Combine/use results of recursive calls.

`helper(c)` will return the total sum of entries and the number of nodes in `c`.

We simply need to add these amounts to the running total and count.

```
    total, count = entry(t), 1
    for c in children(t):
        c_total, c_count = helper(c)
        total += c_total
        count += c_count
    return total, count
```

Notice that the explicit base case isn't necessary! If `t` is a leaf, the for loop will not be entered and `entry(t)` and `1` will be returned anyway.

Trees: average_entry

Step 3: Recurse on each of the children.

Step 4: Combine/use results of recursive calls.

`helper(c)` will return the total sum of entries and the number of nodes in `c`.

We simply need to add these amounts to the running total and count.

```
def average_entry(t):
    "*** YOUR CODE HERE ***"
    def helper(t):
        total, count = entry(t), 1
        for c in children(t):
            c_total, c_count = helper(c)
            total += c_total
            count += c_count
        return total, count
```

Notice that the explicit base case isn't necessary! If `t` is a leaf, the for loop will not be entered and `entry(t)` and 1 will be returned anyway.

Trees: average_entry

Finally, we need to actually call our helper function and solve our problem.

`helper(t)` returns the total sum of the entries and the number of nodes in `t`.

We find the average by dividing the sum with the number of nodes.

```
def average_entry(t):
    "*** YOUR CODE HERE ***"
    def helper(t):
        total, count = entry(t), 1
        for c in children(t):
            c_total, c_count = helper(c)
            total += c_total
            count += c_count
        return total, count
    total, count = helper(t)
    return total / count
```

Trees: Implementation

Now that we've taken a look at how to use/process trees, let's cross the abstraction barrier!

Remember:

- The abstraction barrier stands between the user and the implementation.
- The user does *not* need to know the underlying implementation in order to use an ADT.
- There are multiple ways to implement a single data abstraction.

Trees: Implementation

One possible implementation:

```
def tree(entry, children=[]):
    “”” Construct a tree with the given entry and list of
    children. “””
    return [entry, children]

def entry(t):
    “”” Return the entry at the root of of t. “””
    return t[0]

def children(t):
    “”” Return the list of t’s children. “””
    return t[1]
```

Trees: Implementation

Another possible implementation:

```
def tree(entry, children=[]):
    """ Construct a tree with the given entry and list of
    children. """
    return {'entry': entry, 'children': children}

def entry(t):
    """ Return the entry at the root of of t. """
    return t['entry']

def children(t):
    """ Return the list of t's children. """
    return t['children']
```

Summary

- A tree is a recursive abstract data type that represents hierarchical data.
- **Constructor:**
  - `tree(entry, children=[])`
- **Selectors:**
  - `entry(t)`: returns the entry in the root of `t`
  - `children(t)`: returns a list containing the children of the root of `t`
- Because trees are recursively defined, they are easy to process recursively.
  1. Write a base case for a leaf (usually).
  2. Process the root (which might be additional base case(s)).
  3. Recurse on each of the children.
  4. Combine the result of the recursive calls to solve the problem.
- Like any other abstract data type, there are many possible implementations of trees, and processing a tree does not require knowing the specific implementation.