Data Abstraction and Function-based Trees Meta

Computer Science Mentors 61A

February 27–March 03, 2022

Recommended Timeline

- ADT mini-lecture (5 min)
- Q1: Pokemon selectors (5 min)
- Q2: Are friends (3 min)
- Q3: Cross type friends (7 min)
- Q4: Pokemon constructor (8 min)
- ADT Tree mini-lecture (8 min)
- Q5: Replace x (8 min)
- Q6: Contains N (12 min)

The times in the recommended timeline do not add up to 50 minutes because no mentor is expected to get through all the problems during section. The worksheet is a skeleton around which you should structure your section to best meet the needs of your students. If you get stressed out about covering a lot of content, I encourage you to be open with your students about the way these sessions are structured.

You should probably ask your students at the beginning of section what they would rather go over—ADTs or trees—and then allocate time appropriately.

1 Abstraction

Data abstraction allows us to create and access data through a controlled, restricted programming interface—hiding implementation details for sake of brevity and reusability of code and encouraging programmers to focus on how data is used rather than worrying about how data is internally organized. The two fundamental components of an **abstract data type** are a constructor and selectors:

1. A **constructor** creates a piece of data, and includes all the attributes that make the data unique; e.g. executing `c = car("Nissan", "Leaf")` creates a new instance of a car abstraction and assigns it to the variable `c`.
2. **Selectors** access attributes of a piece of data; e.g. calling `get_make(c)` returns `"Nissan"`.

Created by Alyssa Smith, Esther Shen, Maya Romero, Natalie Wei, Sandhya Ganesan, Kaelyn Huang, Cecilia Aiko, Aurelia Wang

In the example above, you don't know specifically how the model name "Nissan" and the make name "Leaf" are internally bundled into a car, and you don't care, either. The creator of the abstract data type dealt with those details, so that you, the user of the ADT, would only have to know how to store and retrieve the data you need. This separation of concerns between designing and using an interface is called the **abstraction barrier**. While your program won't necessarily break if you break the abstraction barrier, heeding the barrier is best practice and can prevent errors down the road.

Using abstraction to hide unnecessary details can be seen everywhere, not just in code—keyboards, printers, cars, stovetops, and typewriters all employ abstractive interfaces. What are some examples of abstraction in your everyday life? If data abstraction is new to your students or they don't feel very confident in the topic, **consider walking them through the following problems**.

Emphasize the **importance of selectors** – useful for 2).

A good visualization is to draw the data abstraction out using box and pointer diagrams. **Make sure not to get caught up on any specific representation of the data abstraction**, as they should be easy to change 3) is an alternate representation.

Talk about what it means to **break the abstraction barrier. How do you make sure that you are not breaking the abstraction barrier?**

1. The following is an abstract data type that represents Pokemon. Each Pokemon keeps track of its name, type, and friends. Given our provided constructor, fill out the selectors:

```
def pokemon(name, p_type, friends):
    """
    Constructs a Pokemon with the given attributes.
    >>> cyndaquil = pokemon('Cyndaquil', 'Fire', ['Chikorita', 'Totodile'])
    >>> p_name(cyndaquil)
    'Cyndaquil'
    >>> p_type(cyndaquil)
    'Fire'
    >>> p_friends(cyndaquil)
    ['Chikorita', 'Totodile']
    """
    return [name, p_type, friends]

def p_name(p):


    return p[0]

def p_type(p):


    return p[1]

def p_friends(p):


    return p[2]
```

This problem is a gentle introduction to ADTs. We tried to keep it as simple as possible while giving those majority of students who have experience with Pokemon something interesting to play around

with.

Students may be confused on how exactly to figure out what the selectors do. Here the doctests are really useful for helping them figure out what they should do. If they're stuck, you can try nudging them in the right direction by asking them to consider what the relevant functions take as input and give as output. For example, `pokemon` takes in a Pokemon's attributes and returns a Pokemon ADT instance, represented as a list. `p_friends` takes a Pokemon ADT instance (internally represented as a list) and returns its friends. By recognizing that the argument to `p_friends` must be a list of a specified form, they should be able to come to the correct answer relatively easily.

It's really important that students understand this part before moving on to the rest of the problems in this section, since they all build on the Pokemon ADT.

A cheeky thing about this problem is that the amount of space we give them to complete the selectors could be a clue for the problem. Oh well.

Please take the time to go over each of the functions with the doctests so students understand how abstraction works in the context of this problem in order to build up their knowledge for the later parts of using these selector functions

2. This function returns the correct result, but there's something wrong with its implementation. What's the issue, and how can we fix it?

```
def are_friends(p1, p2):
    """
    Returns True iff the Pokemon p1 and p2 are each other's friends.
    """
    return p1[0] in p2[2] and p2[0] in p1[2]
```

Treating the `p1` and `p2` are lists is a Data Abstraction Violation (DAV). We should use a selector instead. The corrected function looks like:

```
def are_friends(p1, p2):
    return p_name(p1) in p_friends(p2) and p_name(p2) in p_friends(p1)
```

The purpose of this problem is to introduce the idea of the abstraction barrier. Instead of teaching the abstraction barrier to your students as an unbreakable rule that is to be obeyed without question, I encourage you to discuss *why* abstraction barriers exist. For example, which version of `are_friends` is more readable? (The revised version, because it uses the interface, which has named selectors.) If we decided to change the underlying implementation of the ADT, how would we have to change the different versions of `are_friends`? (We'd have to update the old version but not the revised version.) It's much more valuable if students understand this reasoning than if they just understand that abstraction barriers shouldn't be broken.

After going over this problem, some students might be confused about why the previous problem is not a violation of the abstraction barrier. After all, in that problem, we were also dealing with the internal representation of the ADT. The difference is, of course, that we need to deal with the internal representation of the ADT in order to make the interface. An analogy I might use is that the car factory needs to tinker with the internal functioning of the engine because while constructing the car you do need to deal with those details; however, once the car is on the road, you don't need to mess with the engine anymore. In the same sense, after we make the interface for an ADT, it is no longer necessary for us to deal with the internal representation and the abstraction barriers can "go into effect".

If data abstraction is new to your students or they don't feel very confident in the topic, **consider**

walking them through this problem.

This part may seem easy/trivial, but emphasize how the selector interface allows you to easily use the ADT without violating abstraction barriers.

3. Write the function `cross_type_friends`, which takes in a Pokemon `p` and a list of Pokemon `pokemon_list` and returns a list of the names of `p`'s cross-type friends in `pokemon_list`. (A cross-type friend is a friend of a different type.) You may assume that the `are_friends` function has been correctly implemented.

```
def cross_type_friends(p, pokemon_list):
    """
    >>> c = pokemon('Charmander', 'Fire', ['Torchic', 'Squirtle',
        'Bulbasaur'])
    >>> t = pokemon('Torchic', 'Fire', ['Charmander', 'Squirtle'])
    >>> s = pokemon('Squirtle', 'Water', ['Torchic', 'Bulbasaur'])
    >>> b = pokemon('Bulbasaur', 'Grass', ['Charmander', 'Squirtle'])
    >>> cross_type_friends(c, [t, s, b])
    ['Bulbasaur']
    >>> cross_type_friends(b, [c, s, b])
    ['Charmander', 'Squirtle']
    """


    friend_list = []
    for other in pokemon_list:
        if are_friends(p, other) and p_type(p) != p_type(other):
            friend_list += [p_name(other)]
    return friend_list

    # Alternative solution

    return [p_name(o) for o in pokemon_list if are_friends(p, o) and
        p_type(p) != p_type(o)]
```

This is the only code-writing question in this section where we ask students to utilize an ADT with the full abstraction barrier intact. I believe that this is a relatively important skill for students to have, so I think this problem is not one to skip.

There are a large number of alternate solutions to this problem. To give students more of a challenge, I elected to not give them a skeleton for this problem.

Some potential leading questions:

- If I have two Pokemon instances, how can I determine whether they are cross-type friends or not?
- What's a typical way we can count up something over a list?
- Did we define any functions that can help us here?

4. In this problem, you'll change the implementation of the Pokemon ADT while keeping the interface the same.

   (a) Complete the constructor for the given selectors.

```
def pokemon(name, p_type, friends):
    """
    >>> lil_guy = pokemon('Pikachu', 'Electric', ['Mewtwo', 'Lucario'])
    >>> p_name(lil_guy)
    'Pikachu'
    >>> p_type(lil_guy)
    'Electric'
    >>> p_friends(lil_guy)
    ['Mewtwo', 'Lucario']
    """


    def select(command):
        if command == 'name':
            return name
        elif command == 'type':
            return p_type
        elif command == 'friends':
            return friends
    return select
```

   Alternate solution:

```
    return lambda sel: {'name': name, 'type': p_type, 'friends':
       friends}[sel]


def p_name(p):
    return p('name')

def p_type(p):
    return p('type')

def p_friends(p):
    return p('friends')
```

   The purpose of this problem is to hammer home the bedrock principles of implementation independence. I like how it kind of ties everything together. You can probably skip it if you don't have enough time, but there's a certain closure to this part that I feel would be missing if it were skipped.

   This problem is similar to the first Pokemon problem, where students were given a constructor and were asked to write selectors. However, it is significantly more challenging because here they are given selectors and have to reverse-engineer a constructor. Students will need to be detectives and use the clues given to them in the selectors to figure this out. Thinking through doctests is particularly useful here. If they're stuck, I'd recommend looking at specific problems and helping them deduce the answer. For example, if we see the line `p('friends')`, what (type) does `p` have to be? How can we ensure that `p` returns the correct value when `'friends'` is provided? If we need to store data, does it need to be in a sequence or container, or perhaps is there another

(sneaky) place it can be stored?

If students are confused by the different terminology—"interface" vs. "implementation"—make sure to clarify this for them. Implementation refers to the "behind the scenes" work that makes an ADT work. an interface, on the other hand, is a set of potential interactions with a datatype, each defined by what inputs we provide and what outputs the interface produces given those inputs.

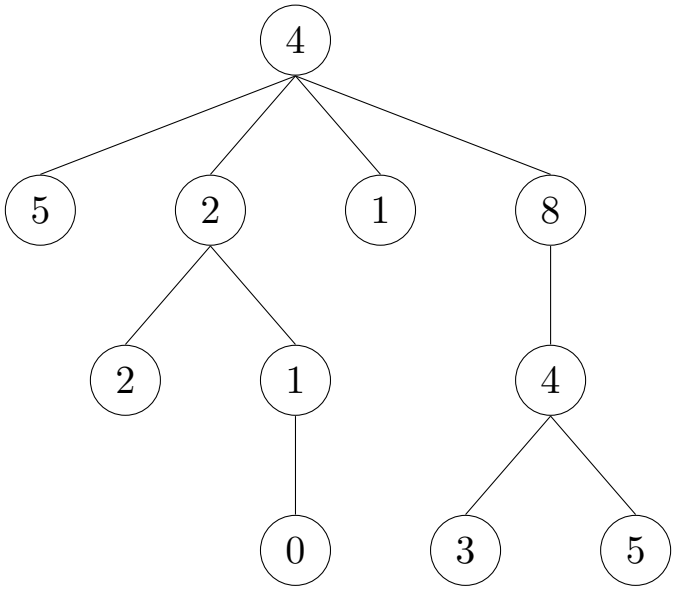
(b) What do we need to change about the implementations of `are_friends` (as revised) and `cross_type_friends` now that we've changed the implementation of the Pokemon ADT? Why?

Nothing. Because we relied on the implementation-independent interface of the Pokemon ADT, changing the underlying implementation does not affect the correctness of these functions.

The purpose of this (trick) question is to underscore the incredible value of implementation independence. That we do not have to change any of our existing code even though we fundamentally change the underlying implementation of the ADT is very useful. Tell your students about how cool this is. You can also note that the value of implementation independence scales with complexity; if we wrote thousands of lines of code that all depended on this ADT, not having to change them would be even more valuable then the small savings we're seeing in this problem.

2 Tree ADT

Trees are a kind of recursive data structure. Each tree has a **root label** (which is some value) and a sequence of **branches**. Trees are "recursive" because the branches of a tree are trees themselves! A typical tree might look something like this:



This tree's root label is 4, and it has 4 branches, each of which is a smaller tree. The 6 of the tree's **subtrees** are also **leaves**, which are trees that have no branches.

Trees may also be viewed **relationally**, as a network of nodes with parent-child relationships. Under this scheme, each circle in the tree diagram above is a node. Every non-root node has one parent above it and every non-leaf node has at least one child below it.

Trees are represented by an abstract data type with a `tree` constructor and `label` and `branches` selectors. The `tree` constructor takes in a label and a list of branches and returns a tree. Here's how one would construct the tree shown above with `tree`:

```
tree(4,
    [tree(5),
     tree(2,
        [tree(2),
         tree(1,
            [tree(0)])]),
     tree(1),
     tree(8,
        [tree(4,
            [tree(3), tree(5)])])])
```

The implementation of the ADT is provided here, but you shouldn't have to worry about this too much. (Remember the abstraction barrier!)

```
def tree(label, branches=[]):
        return [label] + list(branches)
```

```
def label(tree):
        return tree[0]

def branches(tree):
        return tree[1:] # returns a list of branches
```

Because trees are recursive data structures, recursion tends to a be a very natural way of solving problems that involve trees.

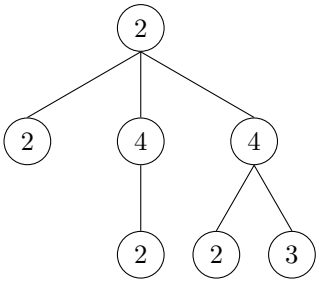
- The **recursive case** for tree problems often involves recursive calls on the branches of a tree.
- The **base case** is often reached when we hit a leaf because there are no more branches to recurse on.

Teaching Tips

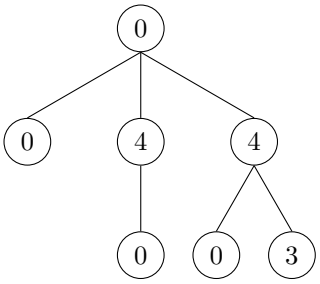
- Please make sure to check in with your students before mini-lecture so that you don't go over too much content that that they already feel comfortable with.
- While it is typically true that your make the recursive calls on the branches of a tree and stop recursing when you reach a leaf, this is by no means always true, and you should make it clear that there will be exceptions to this rule of thumb.
- Common Misconceptions:
  - Students often have trouble with the idea that branches is a list of trees. Try to be specific when explaining, focusing on types. (Branches are lists, saving trees in them.)
    - Try using the tree functions to build up different trees.
    - Write out all the functions on the board and clearly define the types of the output and input.
  - Data Abstraction and Trees
    - Although `t[0]` returns the label from the tree, students should be using `label(t)`. This is because `t` is not a list, it is a tree which is a data abstraction!
    - It's important to explain why indexing branches (e.g. `branches(t)[0]`) doesn't violate an abstraction barrier (since branches returns a list of trees).
- The objectives for students are to:
  - Draw trees as graphical representations given Python code
    - Mention to students that empty branches `[]` is the default argument, so `tree(5)` is the same as `tree(5, [])`.
    - Emphasize variable types.
    - It may be helpful to mark pairs of parentheses to help in understand the nesting relationships.
      - Branches is a function that returns a list of trees.
      - Label values are numbers.
  - Construct Python code given a graphical representation of a tree

1. Write a function, `replace_x` that takes in a tree, `t`, and returns a new tree with all labels `x` replaced with 0.

   For example, if we called `replace_x(t, 2)` on the following tree:



   We would expect it to return




```
def replace_x(t, x):
    """
    >>> t = tree(2, [tree(1), tree(2)])
    >>> replace_x(t, 2)
    tree(0, [tree(1), tree(0)])
    """
    ____________________________________________

    if ____________________________________________:

        return ____________________________________

    return ____________________________________
```

```
def replace_x(t, x):
    new_branches = [replace_x(b, x) for b in branches(t)]
    if label(t) == x:
        return tree(0, new_branches)
    return tree(label(t), new_branches)
```

Here, we construct and return a new tree. First, we make a new list of branches where each branch is the same as the previous branch but all occurrences of x have been replaced with 0 as per our recursive function. The if statement guarantees that if our root node's label is an occurence of x, we replace the subtree we're on starting at its root – keeping all else before it the same while replacing the specific subtree's root node label to be zero.
We do not need a base case here, as if we are at a leaf, the list comprehension we use to create the new branches will evaluate to an empty list. Then we will either return `tree(0, [])` or `tree(label(t), [])` as appropriate.

Teaching Tips

- Draw out a tree and ask them to play out the algorithm
  - If you were a computer, how would you replace all the `x`'s? (Answer: check the value of the current tree, then each of the branches)
  - Can we somehow "simplify" all of this repeated work?
- Make sure they respect abstraction barriers
  - If there isn't a `set_value` function, how can we return a tree with an updated value? (Answer: create a new tree with `0` and the new branches)
- Warn them against trying to evaluate branches
  - What is the simplest replacement we can do?
  - How can we delegate branch replacements to recursive calls?
- If we have multiple branches, how do we make the recursive call on each branch? (Answer: a for loop)
  - What happens in the for loop if there aren't any branches? (Answer: nothing)
  - This is why we don't need an explicit base case (ex. if len(branches) == 0)

2. Write a function that returns `True` if and only if there exists a path from root to leaf that contains at least `n` instances of `elem` in a tree `t`.

```
def contains_n(elem, n, t):
    """
    >>> t1 = tree(1, [tree(1, [tree(2)])])
    >>> contains_n(1, 2, t1)
    True
    >>> contains_n(2, 2, t1)
    False
    >>> contains_n(2, 1, t1)
    True
    >>> t2 = tree(1, [tree(2), tree(1, [tree(1), tree(2)])])
    >>> contains_n(1, 3, t2)
    True
    >>> contains_n(2, 2, t2) # Not on a path
    False
    """
    if n == 0:

        return True

    elif _____________________________________________:

        return _____________________________________

    elif label(t) == elem:

        return _____________________________________

    else:

        return _____________________________________
```

```
def contains_n(elem, n, t):
    if n == 0:
        return True
    elif is_leaf(t):
        return n == 1 and label(t) == elem
    elif label(t) == elem:
        return True in [contains_n(elem, n - 1, b) for b in
          branches(t)]
    else:
        return True in [contains_n(elem, n, b) for b in
          branches(t)]
```

Teaching Tips

1. We have purposely left one line return statements to imply that we are using list comprehension for our solution, so please emphasize to your students that hint when walking through the

problem.

2. Feel free to use the `any` Python built-in instead, which takes in a list of values and returns `True` if any of the values are truthy and `False` otherwise.
3. Illustrate how `n` can be updated in our recursive calls in order to keep track of how many instances we've seen so far.
4. The second base case is slightly tricky, so you're advised to start with the recursive calls first, which will make that base case make more sense.