

Attempt 1

Review Feedback  
10/25/2023Attempt 1 Score:  
100/100

View feedback

## Unlimited Attempts Allowed

10/25/2023

## Details

Problem 1 - drop

Problem 2 - take

Inductive data structures

Trees

Problem 3 - Making a Tree in

Problem 4 - count-tree

Problem 5 - sum-tree

Getting Credit for Your Work

IF YOU ARE IN CLASS OR AT A

IF YOU ARE SUBMITTING REM

Note: For groups that have a significant amount of programming experience, your PM may suggest trying out the <https://bain-cs111.github.io/assignments/adv-tutorial-5>

In this tutorial, we'll practice working with recursion on linked structures such as lists and trees.

Tutorial 5 Starter Files

[https://bain-cs111.github.io/course-files/tutorials/tutorial\\_5\\_template.zip](https://bain-cs111.github.io/course-files/tutorials/tutorial_5_template.zip)

## Problem 1 - drop

Write a function `(drop a-list k)` that returns a copy of list with the first `k` elements removed. In other words, if we

```
(drop '("a" "b" "c") 2)
```

we'd get back the one element list `'("c")`.

Note that you won't need to use `cons` for this, just `rest` as we're removing elements from a list not building a new

Here's some help to get you started:

- First just state the base case and recursive case in English. It's not a bad idea to write it down, but that isn't necessarily straight to writing code.
- Now that you're clear on what the base and recursive cases should be, write the code. You'll want to use an `if` case and the recursive cases.

## Problem 2 - take

Now write a function `(take list k)` that returns a list of just the first `k` elements of the inputted list. In other words,

```
(take '("a" "b" "c") 2)
```

should return the two element list `'("a" "b")`. Note that you'll need to use both `rest` (to remove elements) and `cons`. Use the same method as above.

## Inductive data structures

We talked in the linked list lecture about how lists can be built from simpler structures called `pair`s. Each pair has a next pair, which has a single element and a link to the pair after that, ..., and so on until eventually the last pair list

<https://canvas.northwestern.edu/courses/201068/modules/items/2836621><https://canvas.northwestern.edu/courses/201068/modules/items/2834596>

**Definition:** A `list` is either:

- `()` or
- `(cons e a-list)` where `e` is an element of `a-list` and `a-list` is some `list`.

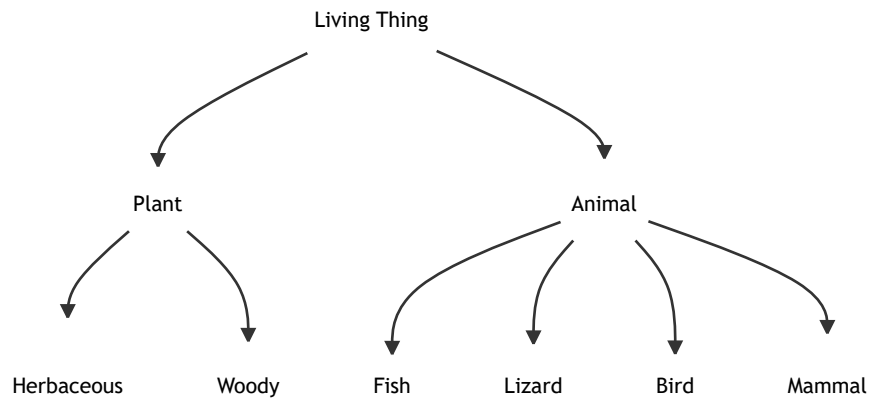
This is called an **inductive definition**, or sometimes a **recursive definition**. Based on this definition, we know that

- `()`, by the definition
- `(cons 1 '())`, since `(cons element list)` is a `list` and `()` is a `list`
- `(cons 2 (cons 1 '()))`, by the same reasoning
- `(cons 3 (cons 2 (cons 1 '())))`, again by the same reasoning

And so on. As we discussed, when we type `(list 3 2 1)` or `'(3 2 1)`, we're really just getting `(cons 3 (cons 2 (cons 1 '())))` just a convenient shorthand.

## Trees

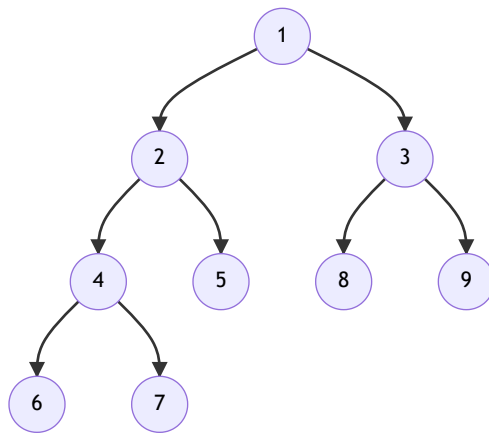
Now let's talk about a different inductively defined data structure called a **tree**. Trees are used to represent some kinds of data, often introduced by an example. When we list a taxonomy of something:



We structure it as a set of items, like “living thing,” “plant,” “animal”, etc., together with lines indicating what is a “subordinate” relationship. This branching structure is called a **tree**. The different items are called **nodes** of the tree and their connections or **branches** form the hierarchy. The top of the hierarchy (“living thing”) is called the **root**, and the things beneath it and connected to it are its **children**. They, in turn, can have children (and so on and so on). In this case, “plant” has the children “herbaceous” and “woody”. The bottom nodes of the hierarchy are items that don't have any children, which are called **leaves**. The leaves here are “herbaceous” and “mammal”.

Trees are absolutely ubiquitous in computing and are a powerful example of an inductive or **recursive** data structure. For example, to form a tree, for example. Because trees are defined recursively, many of the functions we use to work with trees are recursive. One of many reasons why recursion is a powerful programming tool to have in your toolkit.

Of course, as with anything, there are different flavors of trees. One such important type of tree is a **binary tree**. In a binary tree, *nodes aren't allowed to have more than two children*. So when the tree splits, it never splits more than two ways. Here's an example of a binary tree:



In Racket, we can easily define a binary tree of numbers inductively, the same way we defined lists inductively. First, we represent nodes that branch:

```
(define-struct branch (number left right))
```

This says that a branch object has a `number` and a `left` and `right` child. Of course, some nodes are just numbers saying `(make-branch number left right)` where `number` is the number you want to store in the node and `left` and `right` are

Given that, we can inductively define a binary tree of numbers as follows:

**Definition:** A binary-tree is either:

- a number, or
- `(make-branch number binary-tree binary-tree)`

Given this definition, we can say the following are all valid binary trees:

- `6`, since a tree can be just a number
- `7`, same reason
- `5`, same reason
- `(make-branch 4 6 7)`, since `6` and `7` are trees and `(make-branch number tree tree)` is a tree
- `(make-branch 2 (make-branch 4 6 7) 5)`, by the same reasoning, since `(make-branch 4 6 7)` is a tree and so is `5`.

**Note:** Having trouble visualizing what these programs are actually trying to encode? Checkout the [Lecture 14 s](https://docs.google.com/presentation/d/1x4XjJ28pT0WP3STvJp7wk07T7kBLauLITqz_sbC90ao/edit#slide=id.g1727) ([https://docs.google.com/presentation/d/1x4XjJ28pT0WP3STvJp7wk07T7kBLauLITqz\\_sbC90ao/edit#slide=id.g1727](https://docs.google.com/presentation/d/1x4XjJ28pT0WP3STvJp7wk07T7kBLauLITqz_sbC90ao/edit#slide=id.g1727)) but not identical examples.

This definition also gives us a way to write recursive functions on binary trees. Since we know a tree is always either contains further trees inside it), we can have our base case be when the tree is just a single number (which we can predicate) and our recursive case be when it's a `branch` (in which case we recurse on the `left` and `right` children

## Problem 3 - Making a Tree in Racket

The numerical binary tree above would look like this in Racket:

```
(make-branch 1
  (make-branch 2
    (make-branch 4 6 7)
    5)
  (make-branch 3
    8
    9))
```



This says that 1 is the root of the tree, but with two branches. Its branches are 2 and 3, both of which also have and 5, where 4 branches into 6 and 7, but 5 doesn't branch (it's a leaf). And 3 branches into 8 and 9 that are

Make sure to define this tree in your .rkt file so you can use it as a complex test structure for the next 2 problems

## Problem 4 - count-tree

Write a recursive function, count-tree, that counts the number of numbers in a binary tree.

```
count-tree : BinaryTree -> Number
```

If a number appears twice, count both occurrences. That is,

```
(count-tree (make-branch 1
                        (make-branch 2 0 0)
                        (make-branch 3 0 4)))
```

should return 7, in spite of 0 appearing three times.

However, (count-tree 0) should just return 1 (since the only number in that tree is 0).

Here's some subgoals to help you get started:

- First, get clear in spoken language, what will the base case will be. Is it when the tree is a number, or when it's
- Now get clear on what the recursive case will be. Again, don't worry about code.

Hint: this is a tree, so it's probably tree recursion, right? So we're probably calling ourselves twice and combining

- Now write the code. Use your friend if to decide if it's a base case or recursive case.

## Problem 5 - sum-tree

Write a recursive function, sum-tree, that sums all the numbers in a binary tree.

```
sum-tree : BinaryTree -> number
```

For instance,

```
(sum-tree (make-branch 1
                      (make-branch 2 0 0)
                      (make-branch 3 0 4)))
```

should return 10. But (sum-tree 3) should just return 3. Again, here's some steps to get you started:

- What will the base case will be. Is it when the tree is a number or when it's a branch?
- Now get clear on what the recursive case will be.
- Now write the code. Use if to decide if it's a base case or recursive case.

## Getting Credit for Your Work

IF YOU ARE IN CLASS OR AT ALT TUTORIAL




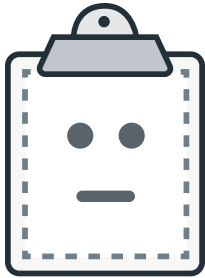
(<https://canvas.northwestern.edu/courses/201068/modules/items/2836621>)



(<https://canvas.northwestern.edu/courses/201068/modules/items/2834596>)

2. Can you understand what their code does by reading it?
3. How was their solution different from yours (if at all)?

Once you've each taken a look, take a second to debrief. Anything either of you found difficult? Easy? Fun? Mind-both of you should fill out this [attendance Google Form](https://forms.gle/vhKxTCXHJUtMoD7v8)  (<https://forms.gle/vhKxTCXHJUtMoD7v8>). **NOTE: Y**



Preview Unavailable

tutorial\_5.rkt

 [Download](#)

([https://canvas.northwestern.edu/files/17572468/download?download\\_frd=1&verifier=gnN9fkCG7CBjPWmmOZrudst9ucV9wT4zLa1uCP2h](https://canvas.northwestern.edu/files/17572468/download?download_frd=1&verifier=gnN9fkCG7CBjPWmmOZrudst9ucV9wT4zLa1uCP2h))