Patterns in Functional Programming

Chad Macbeth

2022-09-08

Contents

Introduction **5**

1 Erlang **9**

1.1 Quick Erlang Primer 9

1.2 Lists in Erlang 13

2 Persistence **17**

2.1 Persistence in Lists 17

2.2 More List Persistence 19

2.3 Specifications and Definitions 21

3 Map and Filter Functors **27**

3.1 Functors and Map 27

3.2 Filter 30

3.3 Functor Properties 32

4 Fold and Unfold Functors **35**

4.1 Fold 35

4.2 Fold Right 37

4.3 Unfold 38

5 Chaining, Currying, and Partial Applications **41**

5.1 Chaining 41

5.2 Currying 43

5.3 Partial Applications 46

6 Monoids and Monads **51**

6.1 Monoids 51

6.2 Monads 54

6.3 List Monad 57

7 Streams and Lazy Evaluation **63**

7.1 Streams 63

7.2 Stream Monad & Collection 66

7.3 Erlang Processes & Streams 69

8 Trees **73**

8.1 Binary Search Tree 73

8.2 Balanced Red-Black Tree 78

8.3 Performance 84

9 Min Heaps **87**

9.1 Min Heap 87

9.2 Efficient Inserting and Removal 96

9.3 Priority Queue 103

10 Random Access Lists **105**

10.1 Binary Numbers and the RAL 105

10.2 Lookup and Update 114

10.3 Performance 118

11 Tries **121**

11.1 Creating the Trie 121

11.2 Searching and Counting 125

11.3 Performance 126

12 Queues and Deques **129**

12.1 Queues 129

12.2 Deques 132

12.3 Performance 135

Introduction

Welcome to CSE 382 - Design Patterns & Data Structures with Functional Programming!

In previous classes, you may have looked at design patterns and data structures that are commonly used in programming. When using functional programming, there are unique design patterns and different approaches to data structures that we need to consider. Here is the schedule for the course:

- Week 1 : Erlang Basics
- Week 2 : Persistence
- Weeks 3-7 : Design Patterns
  - Functors - Map, Filter, Fold, Unfold
  - Chaining
  - Currying
  - Partial Applications
  - Monoids
  - Monads
  - Streams (Lazy)
- Weeks 8-12 : Data Structures
  - Binary Search Trees
  - Min Heaps
  - Random Access Lists
  - Tries
  - Queues & Deques

This is a 3 credit class and there is a reading for each day (e.g Part 1 before class on Monday, Part 2 before class on Wednesday, and Part 3 before class on Friday). Each reading includes a problem set to complete. You should work on each problem set after you do each reading. During class we will cover the material in the reading with examples. The material can be complicated and fast paced so you should do the reading and attempt some of the problems before class.

We will be using Erlang in this class. If you already know Clojure, you will find that Erlang is easier to use and learn (with much fewer parentheses!). During the first week we will learn the basics of Erlang. You will learn more about Erlang as you implement the material during the course.

You will be given starting code for each problem set which you must use. You should submit all the problem sets weekly on Saturday evening to stay on track.

You can resubmit any of your work for a higher grade as many times as you want during the semester. Late work will not be penalized so you can take more time if needed to learn the material. However, submitting late work too frequently may result in getting behind which will have a negative impact on

your learning and subsequent grade. During Week 13 and 14 you will have an opportunity to catch up on any missed work.

At the beginning of Week 10, you will be given an open book, open note, take-home final exam which will be due on the last day of the semester. The final exam will assess your understanding of principles taught during the course.

Your grade will be composed of 80% for the weekly problem sets and 20% for the final exam.

Attendance to class is highly encouraged. If you miss no more than 3 days during the semester (excluding during Week 14), you will receive a 5% bonus on your final grade.

Office hours and contact information for this semester will be put in an I-Learn announcement.

Useful websites that would be good to bookmark:

- Erlang Book: https://learning.oreilly.com/library/view/programming-erlang-2nd/9781941222454/
- CSE 121e (Erlang): https://byui-cse.github.io/cse121e-course/
- Erlang Reference: https://erlangbyexample.org/

Chapter 1

Erlang

In this course we will be using the Erlang programming language. The material that we will cover could be implemented in many different langauges. Erlang will allow us to use less code to explore the design patterns and data structures. If you haven't learned Erlang before, the two sections below will help you to quickly learn the language. As the course progresses, you will find that the implementation of future problem sets will improve your Erlang skills.

1.1 Quick Erlang Primer

This tutorial is not meant to provide complete coverage of the Erlang language. You are encouraged to use the Erlang references you bookmarked as needed throughout the semester.

When writing code in Erlang, you will put your code into a `.erl` file. Each Erlang file will include a `-module` and an `-export` tag. If I create a file called `learn_erlang.erl`, then I would create the following file:

```
-module(learn_erlang).
-export([]).

% Put my functions here
```

The `-export` tag is used to list all functions that can be run externally. Functions are written in the format of `name(Parameters) -> expressions.` Note that parameters and variables always start with an uppercase letter. Functions and atoms (which are labels without values) are always lowercase.

```
-module(learn_erlang).
-export([hello/0, average/2]).

hello() -> io:format("Hello World!~n").

average(Number1, Number2) -> (Number1+Number2) / 2.
```

The `-export` tag shows the number of parameters (or the arity) of the function. Notice that each function ends with a period. To run the functions, execute the `erl` shell command from a terminal.

```
Eshell V12.0  (abort with ^G)
1> c(learn_erlang).
{ok,learn_erlang}
2> learn_erlang:average(10,20).
15
```

The `c` command will compile a module. To run a function, type `module_name:function_name(parameters).` Notice the need to put the period just like in your code file.

```
slope(X1, Y1, X2, Y2) ->
   Delta_X = X2 - X1,
   Delta_Y = Y2 - Y1,
   Delta_Y / Delta_X.
```

In the `slope` function, multiple expressions are used for a slightly more complicated function. Commas are used to separate expressions in the function with a final period at the end. The last expression represents the result of the function.

```
add(N1, N2) -> N1 + N2.
add(N1, N2, N3) -> N1 + N2 + N3.
add(N1, N2, N3, N4) -> N1 + N2 + N3 + N4.
```

The function `add` exists with different arities which is why we write each one as a separate function separated by periods.

```
divide(_N1, 0) -> 0;
divide(N1, N2) -> N1 / N2.
```

In the function `divide`, we have only arity 2 but we have two different scenarios or clauses. Each clause is separated by semicolons and is evaluated in order until a match is found. If we divide by 0, then the first clause will run. If we divide by non-zero, then the second clause will run.

In the `divide` function, also note that the first clause did not need to use `N1` in the expression. Unused parameters in Erlang are prefixed with an underscore.

```
letter_grade(Grade) when Grade >= 90 -> "A";
letter_grade(Grade) when Grade >= 80 -> "B";
letter_grade(Grade) when Grade >= 70 -> "C";
letter_grade(_Grade) -> "F".
```

The `when` keyword is called a guard and can be used with clauses. Basic Boolean logic can be done with a guard. For more complicated logic, a `case` statement can be used which will be seen later.

```
factorial(N) when N =< 1 -> 1;
factorial(N) -> N * factorial(N-1).
```

With the understanding of clauses, we can implement recursive solutions. In this example, the base case for `factorial` is represented by the first clause (notice the `=<` syntax) and recursion is used in the second clause. The first clauses could have been rewritten without a guard if we didn't worry about negative number as: `factorial(0) -> 1;` We call this method of recursion "body recursion" because there are operations (multiply by `N`) outside of the recursive call.

```
factorial(N) -> factorial(N, 1).
factorial(0, Result) -> Result;
factorial(N, Result) -> factorial(N-1, N*Result).
```

We can rewrite the same `factorial` function in what is called "tail recursion" format. In this case, there is no operation outside of the recursive call. This requires that we have the `Result` parameter to keep track of the ever growing value. To support a `Result` parameter, we provide a 1-arity version of the factorial function for the user to call which initializes the product to 1. When the base case is reached (in this case N is equal to 0), then we can just return the result we have been recursively growing. In Erlang, both of these methods result in the same performance. The readability for the specific problem you are solving should determine the method you use. One might argue that the "body recursion" method was easier to read.

```
encrypt(Value, Cipher_Function) -> Cipher_Function(Value).

test_encrypt() ->
   Cipher1 = fun (X) -> X + 1 end,
   Cipher2 = fun (X) -> (2 * X) - 3 end,
   11 = encrypt(10, Cipher1),
   17 = encrypt(10, Cipher2),
   3.0 = encrypt(1000, fun math:log10/1),
   ok.
```

In this code, the second parameter `Cipher_Function` is a function. Common in functional programming, functions are passed and used liked they were any parameter like integers and strings. The `fun (parameters) -> expression end` syntax is used to define an anonymous function (often called a lambda function). The `encrypt` function expects to receive a function that takes only one parameter. The `fun math:log10/1` is an example of providing an existing 1-arity function.

In the `test_encrypt` function, note that `Cipher1` and `Cipher2` variables were created. We could not change the value of `Cipher1` because all variables in Erlang are immutable.

Problem Set 1
You can find the template for the problem sets in this lesson here: prove01.erl
As you work on these problems (as well as all future problem sets), remember to look at the test code provided in the template. There is a separate function (e.g. `test_ps1`) for each problem set. This will help you to understand how these problems work. You will need to uncomment out code in the test code functions as you work through each of these problems.

1. Implement the `hello` function to display "Hello World!".
2. Implement the `add` function to add two numbers.
3. Implement the `multiply` function three different ways each with different number of parameters (i.e. different arity):
   - Multiply one number (not very exciting)
   - Multiply two numbers
   - Multiply three numbers
4. Implement the `water` function that will take a temperature in Fahrenheit and return "Frozen", "Gas", or "Liquid". Implement this using three clauses that use the `when` guard.
5. Implement the `fib` function to calculate the n^{th} Fibonacci number. Assume the 1st and 2nd number is 1. You will need to use recursion. Recommend using body recursion.
6. Implement the `sum` function to add up the numbers from 0 to N. Use recursion instead of using any built in functions. Recommend using body recursion.
7. Create lambda functions to pass to the `plot` function. The `plot` function will create `{X,Y}` coordinates where `X` goes from -5 to 5 and `Y` is calculated from your lambda function. Create lambda functions for the following scenarios:
   - Function that squares the number
   - Function that subtracts one from the number
   - Pass the `abs/1` built-in function directly
   - Function that divides the number by three and then uses the `math:floor/1` built-in function to remove the decimal.

1.2 Lists in Erlang

Lists are built-in to Erlang at the syntax level. During the course, we will write our own functions to work with lists and we will even re-create our own list data structure. To support those activities, we need to understand how to use the

built-in list first.

```
playing_with_lists() ->
   L1 = [1, 2, 3, 4],
   ok.
```

Lists can be created using square brackets. We can use a vertical bar to access the first element of the list.

```
playing_with_lists() ->
   L1 = [1, 2, 3, 4],
   L2 = [0|L1],
   [0, 1, 2, 3, 4] = L2,
   ok.
```

The `[0|L1]` prepends the `0` in front of the `L1` list. The `[0, 1, 2, 3, 4] = L2` will attempt to pattern match L2 with the list on the left hand side. If this doesn't match, our "test" will fail.

```
playing_with_lists() ->
   L1 = [1, 2, 3, 4],
   [A | _] = L1,
   1 = A,

   L2 = [2, 4, 6, 8],
   [B, C | _] = L2,
   2 = B,
   4 = C,
   ok.
```

Pattern matching can reuse the `[First|Rest]` format. In the example above, we can extract the first or the first several values. We can match multiple values at the front of the list using commas. Note the use of the `_` which indicates we are ignoring the "rest" of the list.

```
display_first([First|_Rest]) ->
    io:format("~p~n",[First]).

playing_with_lists() ->
    L1 = [1, 2, 3, 4],
    display_first(L1),
    ok.
```

When the list `L1` is passed to the `display_first` function, the `[First|_Rest]` syntax will split the list up into the first value and the remainder of the list (the latter of which is not used in the function.

```
display_first([]) ->
    io:format("EMPTY~n");
display_first([First|_Rest]) ->
    io:format("~p~n",[First]).

playing_with_lists() ->
    L1 = [1, 2, 3, 4],
    display_first(L1),
    display_first([]),
    ok.
```

In this example, a special clause is added for the empty list. We use `[]` to represent an empty list. If we wanted to represent a list of one item, we could use `[One]`. If we wanted to represent a list of more than one item, we should use `[First|Rest]`.

```
display_all([]) ->
    io:format("end of list~n");
display_all([First|Rest]) ->
    io:format("~p ",[First]),
    display_all(Rest).

playing_with_lists() ->
    L1 = [1, 2, 3, 4],
    display_all(L1),
    display_all([]),
    ok.
```

In this code, we need to use the entire list to display every value. Recursion is used on the `Rest` of the list. The base case is the empty list.

```
playing_with_lists() ->
    L1 = lists:seq(1,10),
    [1,2,3,4,5,6,7,8,9,10] = L1,

    L2 = [N || N <- lists:seq(1,10)],
    [1,2,3,4,5,6,7,8,9,10] = L2,

    L3 = [N*3 || N <- lists:seq(1,10)],
    [3,6,9,12,15,18,21,24,27,30] = L3,

    L4 = [N*3 || N <- lists:seq(1,10), N rem 2 == 0],
    [6,12,18,24,30] = L4,

    ok.
```

There are multiple ways to create lists. The `lists:seq` function will create numbers in a list from the range specified. The other method is the list comprehension. The syntax for the list comprehension is: `[expression || generator, filter]` The `generator` is written with a `<-` and provides a source list. The `expression` uses the generated number to add to the list. The optional `filter` is used to determine which generated values should be used to generate the list. In the last example, the `rem` operator (often called modulo) is used to include only even numbers before tripling them.

Problem Set 2

1. Write a `stack_push` function that treats the list as a stack (LIFO - Last In First Out). The `stack_push` function will take two parameters, the `Stack` (ie, a list) and a value `N`. This function should push the value `N` to the front of the `Stack`.
2. Write a `stack_pop` function that returns a new stack with the first item removed. If the stack is empty, then return the empty list `[]`.
3. Finish the `quicksort` function provided to you in the starting code. The algorithm for quicksort is to first pick a position in the list (in our case we will pick the first number) and call it our pivot. We will then recursively call `quicksort` on all of the numbers less than the pivot and then call quicksort on all of the numbers greater than or equal to the pivot. The correct sorted list is then the concatenation (`++`) of the following 3 lists: `[sorted list of numbers less than the pivot] ++ [the one pivot] ++ [sorted list of numbers greater than or equal to the pivot]`. The code is setup already but you need to write list comprehensions for creating:
   - A sublist containing all numbers in the list less than the pivot (first number in the list) which will be passed to the `quicksort` function.
   - A sublist containing all numbers in the list greater than or equal to the pivot (first number in the list) which will be passed to the `quicksort` function.

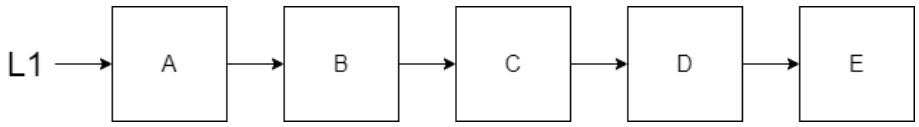
Chapter 2

Persistence

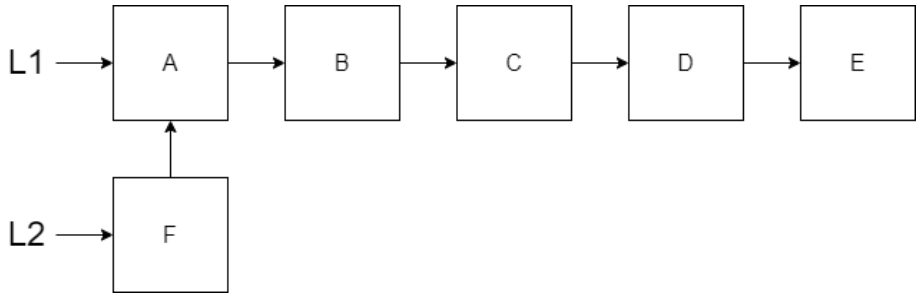
When we think about data structures, we think of things that change as more things are added, modified, or removed. However, in our functional world, we are dealing with immutability. This conflict causes us to learn about persistence. This week we will explore persistence with the common list.

2.1 Persistence in Lists

The lists that we are going to look at early in this course are actually linked lists. We will explore an indexed version when we look at Random Access Lists later on. Lets consider the list below that already has five elements within it.



Note that we have a variable `L1` that contains a reference to the first node in the list and each node in the list contains a reference to the next node. If we want to add a new node to the front of this list, we have to do it without violating the immutability rule. When we add the new node, we are creating a new list. Notice that `L2` below points to the new node we have created. However, since the remaining list doesn't need to change, we just connect the new node to the original `L1`. Since we get to reuse the entire list, the performance for prepending to the front is only O(1).

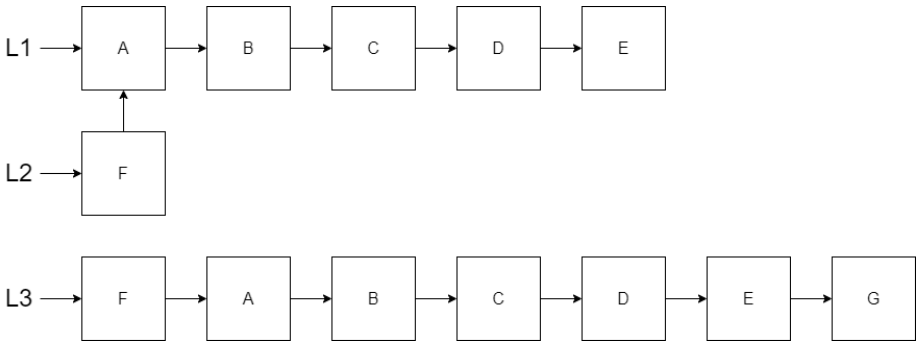


Notice we have persisted the original list and created the new list by reusing what we originally had. This is what we mean be persistent data structures.

The code for this `prepend` operation does not require any recursion.

```
prepend(List, Value) -> [Value|List].
```

Let's consider the situation where we want to append a new node to the end of the list. Looking at our original four item list, how do we add a new item to the end. We will need to create a new list `L3` and try to reuse as much as possible. However, since we are adding the new node to the end, this will cause us to not have any reuse.



Notice in the picture above, the last node in the original list has to change because it needs to reference the new node instead of referencing nothing. This is a change so a new node is needed. The original last node is persisted as part of `L1` and `L2`.

The code for this `append` function requires the use of recursion. When we recurse to the end of the list (calling `append` recursively until we are left with an empty list) then we create our new node as a 1 item list.

Once we get the new node created, we return from our recursive calls and begin adding new nodes for each of the values that were in our previous list.

```
append([], Value) -> [Value];
append([First|Rest], Value) -> [First | append(Rest, Value)].
```

When we look at the `append` function it may be concerning that there is no reuse. If all we did was call the `append` function, there would be multiple duplicates of our data all over our memory. In our programming languages, garbage collection will remove any memory that is no longer referenced by another variable. However, there is a cost for not having reuse. We have to recreate the entire list again which is O(n) performance.

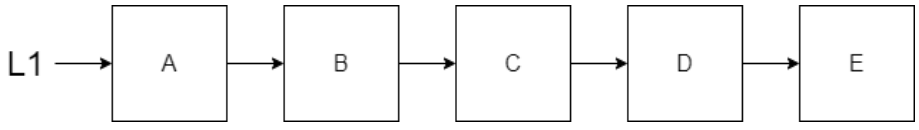
In the next topic below, we will consider inserting something in the middle of our list which will cause us to reuse some of the nodes in our list.

Problem Set 1
You can find the template for the problem sets in this lesson here: prove02.erl

1. Implement the `prepend` function as shown above.
2. Implement the `append` function as shown above.
3. To do some additional practice with lists, write a `head` function that will return the first item in the list. If the list is empty, then return the atom `nil`.
4. Implement a `tail` function that will return the last item in the list. The `tail` function will need to use recursion just like the `append` function. If the list is empty, then return `nil`.

2.2 More List Persistence

Lets consider two more scenarios with the list and observe how persistence is maintained. First, how do we remove the first value in our list? The only thing that will be affected is our variable referencing the front of the list. In the diagram below, `L1` is the original list and `L2` is the new list with the first item removed.

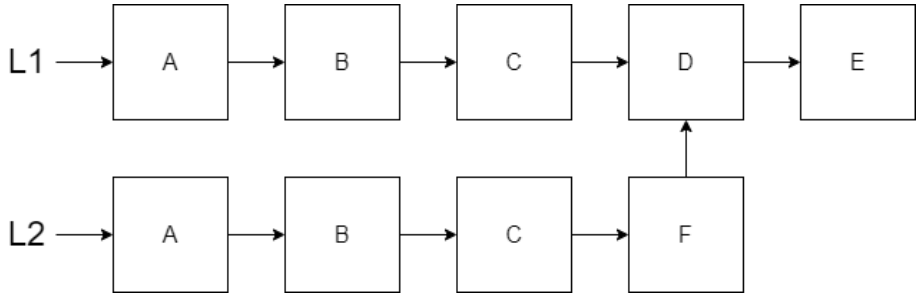


Notice that `L2` is referencing the second item in the list which means all but the first original node is being reused. Both the original list and the new list persist.

Here is the code for our `remove_first` function.

```
remove_first([]) -> [];
remove_first([_First|Rest]) -> Rest.
```

Suppose we start with our original five item list `L1` again and consider inserting an item in the middle. This will be a combination of the ideas that we have seen prepending and appending to our list with respect to reuse. If we insert after index 2 (where the first element is index 0), then we will need new nodes up to index 2 and we can reuse everything after index 2.



Here is the code for our `insert_at` function. In this code, we will decrement our `Index` as we recursively search through the list. Once our `Index` reaches 0 then we know we have reached the desired location. We will then create our new node and connect it (reuse) to the remaining nodes in our original list. Notice that if our `Index` was negative then the original list is returned. For an `Index` too big, we could use the `length` function in the first clause. However, to avoid checking the length in every recursive `insert_at` function call, we have introduced the third clause. If our `Index` is 0 in the second clause, then we know we have found the place to insert. However, if the `Index` is not 0 but the `List` is empty, then we know that our index was too big. This logic is added in the third clause.

```
insert_at(List, _Value, Index) when Index < 0 -> List;
insert_at(List, Value, 0) -> [Value | List];
insert_at([], _Value, _Index) -> [];
insert_at([First|Rest], Value, Index) -> [First | insert_at(Rest, Value, Index-1)].
```

Problem Set 2

1. Implement the `remove_first` as shown above.
2. Implement `insert_at` function as shown above.
3. Implement a function called `remove_last` that removes the last item in the list. If the list is empty, then return an empty list.
4. Implement a function called `remove_at` like the `insert_at` function which removes an item at a specific index. If the index is invalid, then return the original list.

2.3 Specifications and Definitions

Throughout the course when we talk about a behavior, instead of immediately writing it using Erlang, we will first describe it with a specification and a definition. You may find that the our definitions and our Erlang code are similar due to the relative simplicity of the Erlang syntax. Using our definitions, you should be able to write code in any functional language.

A specification describes the interface of a function in terms of data types. Here are a list of the data types we will use in the course:

- *integer*
- *real*
- *boolean*
- *string*.
- *atom*(*name*) - An atom is a label with a specified name but no value.

- a - Represents any type
- $[a]$ - List of values of any type
- $\{a_1, a_2\}$ - 2 element Tuple of values of different types
- $a_1 \nleftrightarrow a_2$ - Dictionary (or map) defined by keys a_1 and values a_2.
- λ - Function (frequently called lambda) that is passed to another function

We write a specification using the *spec*, ::, and $\rightarrow$ notations. The input types are separated by spaces. An example with 3 inputs is given below. Notice that subscripts will be frequently used.

$$spec\ \ name :: input_1\_type\ \ input_2\_type\ \ input_3\_type \rightarrow output\_type.$$

We can define custom types for use in our specifications by defining tuples (or records). To define a custom type, we will use the *struct* keyword as follows: (this example is for a tuple of size 3):

$$struct\ \ name\ \ \{type_1 : name_1, type_2 : name_2, type_3 : name_3\}.$$

In this example, there are three fields each with a type and an optional name. The keyword *or* will be used if there are different tuple sizes (arity) permitted

in our custom type.

A definition is how the function will use the inputs to produce the output. Names will be used to give meaning to the inputs. You should refer back to the specification to understand the types. When we write definitions, we will use the following notations (excluding common ones you have seen previously such as Boolean operators and mathematical operators):

- $=$: Save an intermediate result into an immutable variable
- $+$, $-$, $*$, $/$, and, or, $>$, $<$, $\geq$, $\leq$, $==$, $\neq$: Commonly used operators.
- $(\lambda \ input_1 \ input_2 \ input_3)$: Call a function (in this case called λ with the supplied parameters. Notice that this does feel like a function call in Clojure with the parentheses.
- [] : Empty List
- $\{\leftarrow\rho\}$: Empty Dictionary
- $[First|Rest]$: This splits the list where First is the first element in the list and Rest is the rest of the list.
- $List1 + +List2$: Combines (or concatenates) two lists together.
- $struct\_name.field\_name$: Access a field from a custom tuple
- *nil* - An atom that represents nothing (used sometimes for error conditions). This atom is special in that it is compatible with any type in the specification. For example, if a function was supposed to return a $[a]$, for certain conditions (like error conditions), you could return a *nil* in the definition.
- when *condition* : Provide a guard condition on the function clause.
- We will introduce many different parameter and variable names in our function definitions. Here are common ones that we will use in the examples:
  - $List$ - A list of something
  - $Value$ - A single thing used in a computation or to be added to a data structure like a list
  - $Index$ - The index (sometimes starting at 0 or 1 depending on the problem) in a linear data structure like a list
  - $Result$ - A final answer or the current answer working towards the final answer in cases of recursion
  - $Text$ - A string that is being processed or produced
  - $Count$ - How many of something
  - Acc - Represent an accumulator (something being aggregated together like adding)
  - $Curr$ - Representing the current value of something especially with counters
  - $Item$ - A single item in a collection like a list
  - $Init$ - The first or initial value of something (e.g. an accumulator)

We write a definition using the same format as the specification but with the *def* notation in the front. Instead of data types, inputs are given names and outputs are represented as expressions that use the inputs.

```
def name :: input_1 input_2 input_3 -> output_expressions.
```

If a function name exists with different number of inputs (different arity) then we will write a specification for each one. The definition for each specification may take multiple clauses. Each clause will handle a different scenario and will be separated by a semicolon. Clauses are evaluated in order. If a clause doesn't use all the inputs, then an underscore is placed in front of the input name.

Let's do some examples with our list functions. Here is the specification and definition for the `prepend` function. The prepend function took a list and a value. Therefore the specification will use inputs of $[a]$ and a. The output is just a longer list so it will be $[a]$.

```
spec prepend :: [a] a -> [a].
def prepend :: List Value -> [Value|List].
```

The append function will have a similar specification but the definition will be more complicated since we had to use recursion. When you call a function in your definition, you should be the function call in parentheses. Notice that the definition uses the [] and $[First|Rest]$ notation in the input to help us differentiate the two clauses.

```
spec append :: [a] a -> [a].
def append :: [] Value -> [Value];
def append :: [First|Rest] Value -> [First|(append Rest Value)].
```

Let's look at the specifications and definitions for the `remove_first` and `insert_at` functions.

```
spec remove_first :: [a] -> [a].
def remove_first :: [] -> [];
def remove_first :: [First|Rest] -> Rest.
```

In the `insert_at` function, we need to have something called a guard. Guards are like if statements that are applied to either a clause or to part of the code within a clause. We use the phrase *when* in our definition to indicate a guard. The 3 clauses are evaluated in order.

```
spec insert_at :: [a] a integer -> [a].
def insert_at :: List Value Index -> List when Index <
0 or Index > (length List);
def insert_at :: List Value Index -> [Value|List] when Index ==
0;
def insert_at :: [First|Rest] Value Index ->
[First|(insert_at Rest Value Index - 1)].
```

In the definition above, we assume that there is a `length` function available.

Problem Set 3

1. Write the specifications and definitions for the `head`, `tail`, `removeLast`, and `removeAt` functions. You will write these in your code template as comments.
2. Implement a function called `backwards` (which does the same thing as the `reverse` function found in Erlang for lists). The specification and definition of this functions is shown below. Notice there are two specifications because there is a one arity function (called by the user in the test code) and a two arity function (called recursively with the result list). You will need to implement both specifications.

```
spec backwards :: [a] -> [a].
def backwards :: List -> (backwards List []).
spec backwards :: [a] [a] -> [a].
def backwards :: [] Result -> Result;
def backwards :: [First|Rest] Result ->
(backwards Rest [First|Result]).
```

Chapter 3

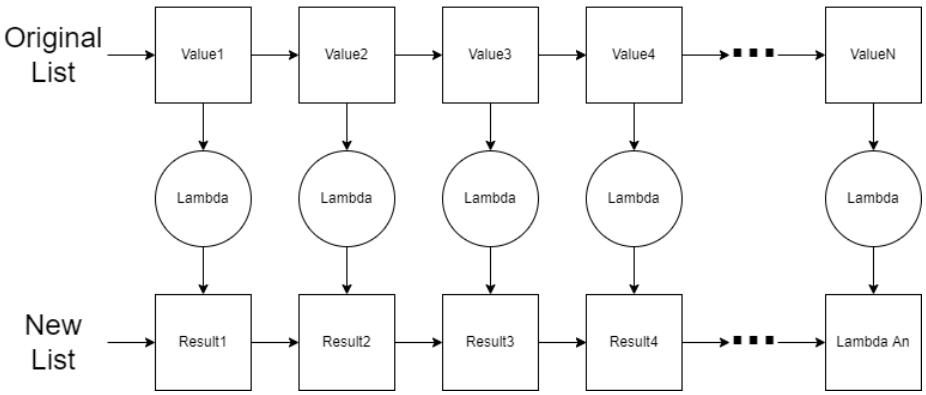
Map and Filter Functors

The `map` and `filter` are common functions that are available in programming languages to simplify the process of processing a loop. Frequently also simplified with a list comprehension syntax, these functions provide an introduction to a very common subset of functions called functors.

3.1 Functors and Map

A functor is a mathematical term used in the field of category theory. The concept is that a functor is something that can convert from one "category" to another "category". In computer science, we consider these "categories" to be data types. The most common conversion involves the list. The `map` and `filter` that we use this week convert from a list to another list. The `fold` and `unfold` that we learn about next week convert from a list to a single result (and vice versa).

Let's consider the `map` function first. The `map` converts a list to another list using a lambda function. The lambda function. defines how to convert each item in the original list to an item in the result list.



If I wanted to double all the values in the list, I would use the following lambda function:

$$spec \;\; \lambda :: a \rightarrow a.$$
$$def \;\; \lambda :: Value \rightarrow Value * 2.$$

If I wanted to square of all values in the list, I would use the following lambda function

$$spec \;\; \lambda :: a \rightarrow a.$$
$$def \;\; \lambda :: Value \rightarrow Value * Value.$$

If I wanted to convert a list of strings to a list of string lengths, the lambda function would be (assuming you have a function called `length`):

$$spec \;\; \lambda :: string \rightarrow integer.$$
$$def \;\; \lambda :: Text \;\; \rightarrow (length \;\; Text).$$

Notice that the `map` function is expecting that the lambda always has 2 input parameters.

Here is the formal definition of the `map`:

$$
\begin{array}{l}
spec\ \ \lambda :: a_1 \rightarrow a_2. \\
spec\ \ map :: \lambda\ \ [a_1] \rightarrow [a_2]. \\
def\ \ map :: \lambda\ \ [] \rightarrow []; \\
def\ \ map :: \lambda\ \ [First|Rest] \rightarrow [(\lambda\ \ First)|(map\ \ \lambda\ \ Rest)]
\end{array}
$$

First thing to notice is that `map` function does reply on the definition for λ. Also notice the recursive nature of the `map` as it applies the lambda function to the each element one at a time starting with the first element ($First$). The result of calling the lambda function ($\lambda\ \ First$) becomes the new value to add to the front of the resulting list.

Consider the code implementation in Erlang below.

```
map(_Lambda, []) -> [];
map(Lambda, [First|Rest]) -> [Lambda(First)|map(Lambda, Rest)].
```

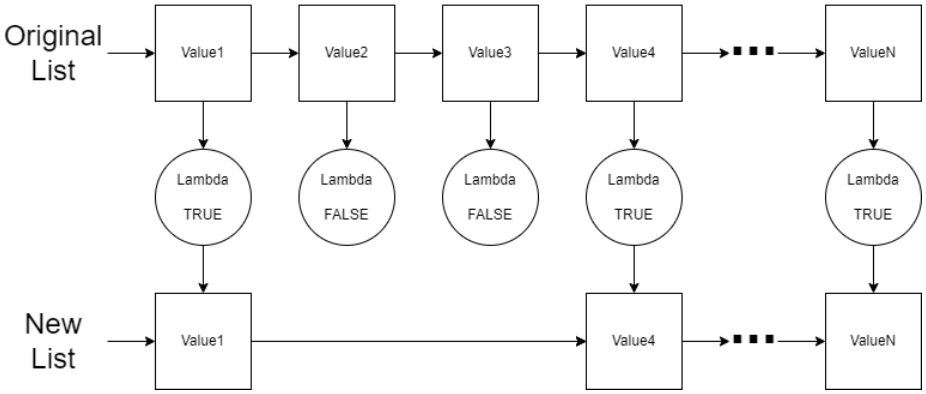
Functors in programming will frequently use a lambda function to allow us to write more generalized and abstract functions like `map`. Notice that the `map` function doesn't know what the lambda will do except that it is a lambda that converts one value to another value. Note that Erlang also has a built-in function `lists:map`.

Problem Set 1
You can find the template for the problem sets in this lesson here: prove03.erl

1. Implement the `map` function described above and write test code to convert a list of measurements in inches to a list of measurements in centimeters. Use the formula $1in = 2.54cm$.
2. Using the `map` function you wrote, use a simple cipher to encrypt a list of characters. The simple cipher should shift all characters by 1 per the ASCII table . For example, "PASSWORD" should be "QBTTXPSE". In Erlang, a string is represented as a list of characters. Therefore, you can list notation with strings. Additionally, each character is treated as a number as shown in the ASCII table which means you can add numbers to letters.
3. Rewrite the `map` function using a list comprehension and test it with your cipher test code. Call the new function `map_2`.

3.2 Filter

The `filter` function is a functor that converts from a list to a list just like the `map` function. However, the lambda function used by `filter` is intended to return a boolean result that will be used to determine if the value in the original list will be included in the new list.



If I wanted to include only even number values in my list, then the lambda function would be as follows:

$$spec \;\; \lambda :: integer \rightarrow boolean$$
$$def \;\; \lambda :: Value \rightarrow Value \;\; mod \;\; 2 == 0.$$

If I wanted to include only three digit numbers, then the lambda function would be:

$$spec \;\; \lambda :: integer \rightarrow boolean$$
$$def \;\; \lambda :: Value \rightarrow Value \geq 100 \;\; and \;\; Value \leq 999.$$

In both of these examples above, the lambda function is expected to return a boolean condition. If it returns true, then the item will be included in the resulting list.

The formal definition of the `filter` is given below. The implementation is left for an exercise. Note that Erlang does have a built-in function called `lists:filter`.

$$
\begin{array}{l}
spec\ \ \lambda :: a \rightarrow boolean \\
spec\ \ filter :: \lambda\ \ [a] \rightarrow [a]. \\
def\ \ filter :: \lambda\ \ [] \rightarrow []; \\
def\ \ filter\ \ :: \lambda\ \ [First|Rest]\ \ \rightarrow \\
[First|(filter\ \ \lambda\ \ Rest)]\ \ \text{when}\ \ (\lambda\ \ First) == true; \\
def\ \ filter :: \lambda\ \ [First|Rest] \rightarrow (filter\ \ \lambda\ \ Rest).
\end{array}
$$

In Erlang, we are limited in what we can put in a `when` guard including boolean operations and a limited subset of built-in functions. When we need to compare a computed result (in the case of the `filter` we need to run the lambda function and consider the result), you can use a `case` statement. With a `case` statement, you can use `_Else` to represent the default or otherwise case.

```
did_it_work(Number) ->
   Result = process_it(Number),
   case Result of
      42 -> do_something(Number);
      _Else -> do_something_else(Number)
   end.

try_something_else(Number) ->
   Result = process_it(Number),
   case Number > 42 of
      true -> do_something(Number);
      _Else -> do_something_else(Number)
   end.
```

Note that Erlang provides this function as a built-in function called `lists:filter`.

Problem Set 2

1. Implement the `filter` function in Erlang. Use a `case` block to determine whether an item in the list should be included. Test the `filter` to get a list of even numbers from a list using the lambda described in the reading.
2. Rewrite the `filter` function so that it uses a list comprehension instead of using the `case`. Test the new function with the same lambda function in the previous problem. Call the new function `filter_2`.
3. Use the `filter` functions you wrote to filter a list of temperatures (in Celsius) that will support liquid water (as opposed to frozen ice or boiling steam).
4. Use the `filter` function you wrote to filter a list of result strings that started with the prefix "ERROR:". Consider using the `string:prefix` function to solve this problem.

3.3 Functor Properties

Our `map` and `filter` functions are special in that satisfy the following properties which apply in mathematics to functors:

1. Identity - The functor must be able to convert back to itself. In math, 1 is an identity for multiplication and 0 is an identify for addition.

2. Distributive - The functor must be able to behave with the distributive property. In math, if I had a functor f and two other functions g and h, then the following must be true: $f(g(h)) = f(g)(f(h))$. This is frequently written using composition notation: $f(g \circ h) = f(g) \circ f(h)$. This means that I can apply my function f to either g and h combined or I can apply function f separately to g and h and then combine the results.

In the examples below, we will observe how these mathematical properties are satisfied by `map` and `filter`. The identity property requires us to find a λ that will result in no changes to our original list.

spec $\lambda :: a \rightarrow a.$
def $\lambda :: Value \rightarrow Value.$

Here is the code that demonstrates that identity principle using the `map` written earlier:

```
[1,2,3,4] = map(fun(Value) -> Value end, [1,2,3,4]),
```

Now lets consider the second rule of functors. Does the distributive property hold? In the formula $f(g \circ h) = f(g) \circ f(h)$, we will use the `map` function as our functor f. The g and the h will be separate lambda functions which are randomly selected for demonstration purposes.

$$
\begin{aligned}
&spec\ \ \lambda_g :: a \rightarrow a.\\
&def\ \ \lambda_g :: Value \rightarrow 2 * Value.\\
&spec\ \ \lambda_h :: a \rightarrow a.\\
&def\ \ \lambda_h :: Value \rightarrow (Value * Value) - 1.
\end{aligned}
$$

In the following code, we will implement both sides of the distributive property to see if it holds true using a simple list of numbers.

```
G = fun(Value) -> 2 * Value end,
H = fun(Value) -> (Value * Value) - 1 end,

% Left Side of Distributive Property
G_H = fun(Value) -> G(H(Value)) end,
[0,6,16,30] = map(G_H, [1,2,3,4]),

% Right Side of Distributive Property
[0,6,16,30] = map(G, map(H, [1,2,3,4])),
```

While understanding the properties of functors is educational, the way the code is written in the two examples above is more instructive. Notice the creation of the `G_H` function which is the composition of two functions. Also note that the this was needed because the `map` function requires a lambda with only one parameter. As you study the code above, notice how the `map` functions were combined together in the "Right Side" case. We call this "chaining" which we will learn about next week.

Demonstrations of these two properties of functors for the `filter` function is left as an exercise.

Problem Set 3

1. Write code in Erlang to demonstrate the identity property for the `filter` functor. Reuse the `filter` function you wrote earlier.
2. Write code in Erlang to demonstrate the distributive property for the `filter` function. Reuse the `filter` function you wrote earlier. You will have to come up with your own appropriate `G` and `H` lambda functions in your demonstration. Remember that the lambda functions for `filter` expect to return boolean. In other words, when you demonstrate the distributive property, you will be composing (or chaining) boolean conditions together using the `and` operator.

Chapter 4

Fold and Unfold Functors

The `fold` and `unfold` are part of the common functor design patterns. A `fold` will convert from a list to a single value whereas an `unfold` will convert from a single value and a list.

4.1 Fold

Just like the `map` and `filter` patterns, we will use a lambda function to define what we want to do with each item of the list. Unlike the `map` and `filter`, we will not apply the lambda function to determine what to put in the resulting list. Instead (in the case of `fold`) we will use the lambda function to determine how each item in our list contributes to the one single value result. The lambda function is used to combine all the values in the list.

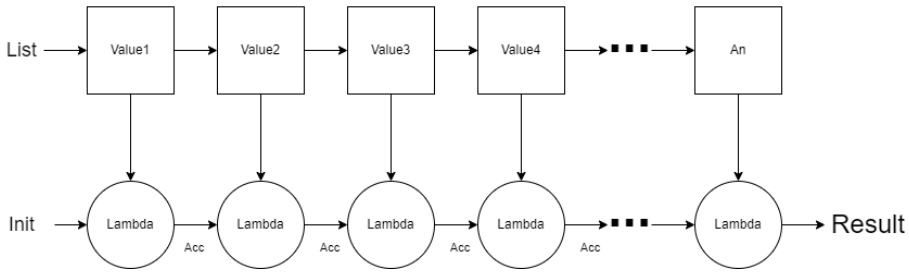
If I have a list of numbers `[1 2 3 4 5]` that I wanted to add, then a lambda function would take each number ($Value$) and add it to an accumulator (Acc):

$$spec \;\; \lambda :: real \;\; real \to real.$$
$$def \;\; \lambda :: Value \;\; Acc \to Acc + Value.$$

If I wanted to add the squares of the numbers in the list, then I would want the lambda function to be

$$spec \;\; \lambda :: real \;\; real \to real.$$
$$def \;\; \lambda :: Value \;\; Acc \to Acc + (Value * Value).$$

The result of the lambda functions will be passed in as the accumulator value when we goto the next item in the list. This implies that we will need to define what the initial accumulator value should be.



Notice that the `fold` function is expecting that the lambda always has 2 input parameters where the second parameter is the accumulator. The single output of the λ function is the updated accumulator.

Here is the formal definition of the `fold`:

$$
\begin{array}{l}
spec\ \ \lambda :: a_1\ \ a_2 \rightarrow a_2. \\
spec\ \ fold :: \lambda\ \ a_2\ \ [a_1] \rightarrow a_2. \\
def\ \ fold :: \lambda\ \ Acc\ \ [] \rightarrow Acc; \\
def\ \ fold :: \lambda\ \ Acc\ \ [First|Rest] \rightarrow (fold\ \ \lambda\ \ (\lambda\ \ First\ \ Acc)\ \ Rest).
\end{array}
$$

Notice the recursive nature of the `fold` as it applies the lambda function to the each element one at a time starting with the first element ($First$). The result of calling the lambda function ($\lambda\ \ First\ \ Acc$) becomes the new accumulator value when `fold` is called recursively on the remainder of the list ($Rest$).

Consider the code implementation below.

```
fold(_Lambda, Acc, []) -> Acc;
fold(Lambda, Acc, [First|Rest]) -> fold(Lambda, Lambda(First, Acc), Rest).
```

The initial `Acc` passed to the `fold` function represents the initial value of the accumulator. If we were summing up numbers in a list, we would expect the initial accumulator to be 0.

Note that Erlang provides this function as a built-in function called `lists:foldl` (meaning fold left).

Problem Set 1
You can find the template for the problem sets in this lesson here: prove04.erl

1. Implement the the `fold` code and use the lambda examples above to sum a list of numbers. Test code is provided for you in the starting code.
2. Create a lambda function to use with your `fold` function to concatenate a list of strings. Note that you can use the `++` operator to solve this problem. Test code is provided for you in the starting code.
3. Create a lambda function to use with your `fold` function to count the number of items in a list. Test code is provided for you in the starting code.
4. Create a lambda function to use with your `fold` function to reverse a list. Note that the single result that a `fold` returns can be a list if the lambda function is written properly. Test code is provided for you in the starting code.

4.2 Fold Right

When you look at the definition of `fold` notice that the list is processed from left to right. We can define a function that goes from right to left called `foldr`. This function will require us to traverse to the end of the list before we can actually call the lambda function.

$$
\begin{aligned}
&spec\ \ \lambda :: a_1\ \ a_2 \rightarrow a_2.\\
&spec\ \ foldr :: \lambda\ \ a_2\ \ [a_1] \rightarrow a_2.\\
&def\ \ foldr :: \lambda\ \ Acc\ \ [] \rightarrow Acc;\\
&def\ \ foldr :: \lambda\ \ Acc\ \ [First|Rest] \rightarrow (\lambda\ \ First\ \ (foldr\ \ \lambda\ \ Acc\ \ Rest)).
\end{aligned}
$$

The code implementation of our function in Erlang is left for an exercise below. Erlang provides a built-in function called `lists:foldr` to perform this task.

Problem Set 2

1. Implement `foldr` and test it with the concatenation of a list of strings. Observe the different behavior with the left `fold`.

4.3 Unfold

The `fold` design pattern is used when you want to consolidate from one larger thing to a smaller thing such as a list to a single value. If we want to go backwards, this is called an `unfold`. Note that folding and unfolding are not inversely related. For example, if I had a list of numbers `[2 5 3 1]` and I folded them up using a simple sum function, I would get `11`. However, if started with `11` and worked backwards, I could get a possible solution such as `[7 1 3 0]` which is different from our original list.

When we `unfold`, we are relying on some initial conditions to generate the next value for our list based on those initial conditions.

Suppose we wanted to generate a list of $Count$ numbers with a value of $Value$. We want to `unfold` our initial conditions to obtain a list of numbers. For example, if $Count$ is 4 and $Value$ is 6, then the `gensame` function should result in `[6 6 6 6]`. Here is a definition for our `gensame` function:

$$spec\ \ gensame :: integer\ \ real \rightarrow [real].$$
$$def\ \ gensame :: 0\ \ Value \rightarrow [];$$
$$def\ \ gensame\ \ ::\ \ Count\ \ \ Value\ \ \rightarrow\ \ [Value|(gensame\ \ \ (Count - 1)\ \ Value)].$$

Here is the Erlang code for `gensame`:

```
gensame(0, _Value) -> [];
gensame(Count, Value) -> [Value | gensame(Count-1, Value)].
```

We could modify this function to create an increasing sequence of numbers by including an initial value ($Init$) and step size ($Step$). To accomplish this, we need to define a lambda function to calculate the next value. We will use $Curr$ to represent the current value that started with $Init$ and was increased by $Step$.

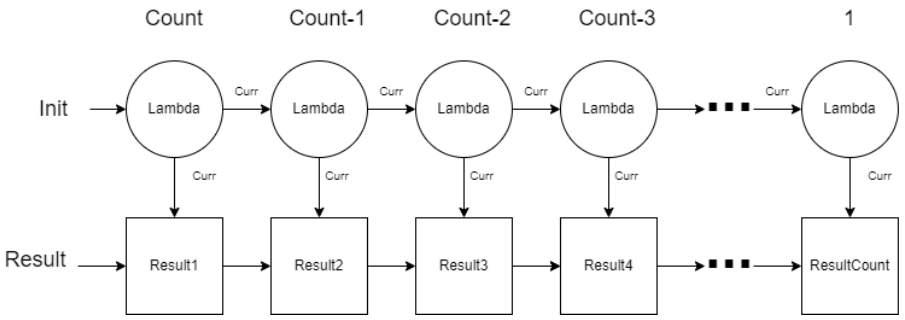
For example, if we wanted a list of 5 even numbers starting at 4, then $Count$ would be 5, $Init$ would be 4, and our lambda would be:

$$spec\ \ \lambda :: real \rightarrow real.$$
$$def\ \ \lambda :: Curr \rightarrow Curr + 2.$$

The output of our `genincr` function would be `[4 6 8 10 12]`.

$$spec\ \ genincr :: integer\ \ real\ \ \lambda \rightarrow [real].$$
$$def\ \ genincr :: 0\ \ Curr\ \ \lambda \rightarrow [];$$
$$def\ \ genincr\ \ ::\ \ Count\ \ \ Curr\ \ \ \lambda\ \ \rightarrow\ \ [Curr|(genincr\ \ \ (Count\ -\ 1)\ \ (\lambda\ \ Curr)\ \ \lambda)].$$

Looking at these two examples, we can generalize our function. Let's create a generic `unfold` function that works for more than just numbers.



This function does not exist in the Erlang library primarily because its not generic enough. We can create an `unfold` type function that does not rely on a `Count` variable or uses a lambda function that has a different number of parameters. The specification and definition is the same as `genincr` but will have a more generalized lambda specification with no definition.

$$spec\ \ \lambda :: a \rightarrow a.$$

The Erlang code for `unfold` is given below:

```
unfold(0, _Curr, _Lambda) -> [];
unfold(Count, Curr, Lambda) -> [Curr|unfold(Count-1, Lambda(Curr), Lambda)].
```

Problem Set 3

1. Implement the `unfold` function as described above and use it to generate an arithmetic sequence of numbers starting at 5 and ending at 30 stepping by 5 (`[5, 10, 15, 20, 25, 30]`).
2. Use the `unfold` function to generate an geometric sequence of six numbers starting at 1 with a factor of 1/2 (`[1, 0.5, 0.25, 0.125, 0.0625, 0.03125]`).
3. Write a function called `range` that takes in *Start*, *Size*, and *Step* parameters (in that order) and returns a list of size *Size* starting at *Start*, stepping by *Step*. Your `range` function must use the `unfold` function. For example, `range(3,5,4)` would return `[3, 7, 11, 15, 19]`. You should assume that both *Size* and *Step* are positive integers greater than 0.

Chapter 5

Chaining, Currying, and Partial Applications

In this lesson we will learn about three different but related design patterns. `Chaining` will consider the composition of functions. `Currying` will convert a function into multiple functions based on input parameters thus allowing you to chain the input parameters together. `Partial Applications` will convert a function into multiple functions based on behavior thus allowing you to execute part of the original function and reuse the result multiple times.

5.1 Chaining

If we need to apply multiple operations on our inputs, we can define each operation in a function and then chain them together. Chaining is the process of executing a series of functions where the output of one is used as the input for the next. Mathematically, we show chaining or composition as follows:

$$f(g(h())) = f \circ g \circ h$$

In this example, h runs first and passes the result to function g. After g runs, the result is passed to function f.

Consider the following Erlang functions that are chained together in various ways:

```
twice(X) -> 2 * X.
square(X) -> X * X.
third(X) -> X / 3.

test_chain() ->
    6.0 = third(twice(square(3))),
    6.0 = twice(third(square(3))),
    4.0 = square(twice(third(3))),
    2.0 = twice(square(third(3))),
    ok.
```

Note that we were able to chain the functions in all various permutations because each of the functions had a single number input and single number output.

If you think about the `map` and `filter` functions, these could be chained together because they both have a single list as an input and output. You will make this observation in the exercises below.

When we chain functions, we can also work with functions that return functions. Consider the functions `multiply_list` and `greater_list`:

$$spec\ \ multiply\_list :: real \rightarrow (\lambda :: [real] \rightarrow [real]).$$
$$spec\ \ \lambda_{map} :: real \rightarrow real.$$
$$def\ \ multiply\_list :: Value \rightarrow (\lambda :: List \rightarrow (map\ \ (\lambda_{map} :: Item \rightarrow Item * Value)\ \ List)).$$

$$spec\ \ greater\_list :: real \rightarrow (\lambda :: [real] \rightarrow [real]).$$
$$spec\ \ \lambda_{filter} :: real \rightarrow boolean.$$
$$def\ \ greater\_list :: Value \rightarrow (\lambda :: List \rightarrow (filter\ \ (\lambda_{filter} :: Item \rightarrow Item > Value)\ \ List)).$$

In these definitions, the function returns a function that does either a `map` or a `filter`. The lambda function for `map` and `filter` is constructed using the $Value$ input parameter. For example, if `multiply_list` receives 4 for $Value$, then a function that performs a `map` on the list with a lambda of $\lambda_{map} :: Item \rightarrow Item * 4$ is created.

The implementation of the `multiply_list` function is given below in both Erlang and Clojure. The implementation for `greater_list` is left for an exercise. An example of chaining these together is given as well.

```
multiply_list(Value) -> fun(List) -> lists:map(fun(Item) -> Value * Item end, List) end.
greater_list(Value) -> implemented_in_exercise.

test_chain() ->
   % Multiply all items in list by 2 and then
   % filter for all items greater than 10.
   L = [2, 4, 6, 8, 10, 12],
   [12,16,20,24] = (greater_list(10))((multiply_list(2))(L)),

   % Filter for all items greater than 10 and
   % then multiply all items by 2.
   [24] = (multiply_list(2))((greater_list(10))(L)),

   ok.
```

Problem Set 1
You can find the template for the problem sets in this lesson here: prove05.erl

1. Chain the `map`, `filter`, and `foldl` Erlang functions together (**single line of code**) as follows:

- Use `map` to triple all the values in the starting list `[1,2,3,4,5,6,7,8,9,10]`
- Use `filter` to only save the even numbers from the `map` result
- Use `foldl` to get the product of those even numbers from the `filter` result. The expected result is 933120.

2. Implement the `greater_list` function provided above and the `multiply_list` function defined above. Test it with the code provided in the template.
3. Create a `multiples_of_list` function that takes an integer parameter `Value` and returns a function that will perform the appropriate filter on a list of integers. The filter should only include values in the list that are multiples of `Value`. Include a specification and definition for the function. Test it with the code provided in the template.

5.2 Currying

With chaining, we were able to compose functions together serially. With currying, we are able to compose input parameters to a function serially. This done by creating functions to handle each input parameter.

Consider the following math function which adds three numbers:

$$spec\ \ add3 :: real\ \ real\ \ real \rightarrow real.$$
$$def\ \ add3 :: Value_1\ \ Value_2\ \ Value_3 \rightarrow Value_1 + Value_2 + Value_3.$$

To curry this function, we need to rewrite it to accept only one parameter and return a function to take the next parameter. This process of returning a function to take the next parameter will continue until all parameters have been received. The final function that received the last parameter will actually perform the function behavior with all the inputs. Here is the curried version of the `add3` function.

$$spec\ \ add3\_curry :: real \rightarrow (\lambda_1 :: real \rightarrow (\lambda_2 :: real \rightarrow real)).$$
$$def\ \ add3\_curry :: Value_1 \rightarrow (\lambda_1 :: Value_2 \rightarrow (\lambda_2 :: Value_3 \rightarrow$$
$$Value_1 + Value_2 + Value_3)).$$

Here is the code implementation for the curried function:

```
add3_curry(Value1) -> (fun(Value2) -> (fun(Value3) -> Value1 + Value2 + Value3 end) end

test_curry() ->
    14 = ((add3_curry(2))(5))(7),
    ok.
```

We can call the curried function and chain the inputs together. A useful use of a curried function is to save one of the inner functions. For example, if we frequently want to add 2 and 5 with other numbers, we can create a function `Add3_StartWith2And5`.

```
test_curry() ->
    Add3_StartWith2And5 = (add3_curry(2))(5),
    14 = Add3_StartWith2And5(7),
    17 = Add3_StartWith2And5(10).
    ok.
```

A generic function can be written to curry a function for specific number of input parameters. For example, to support our `add3` function, we can write a function called `curry3`. The input parameter for the `curry3` is the function we want to curry.

$$
\begin{array}{l}
spec\ \ \lambda :: a_1\ \ a_2\ \ a_3 \rightarrow a_4. \\
spec\ \ curry3 :: \lambda \rightarrow (\lambda_1 :: a_1 \rightarrow (\lambda_2 :: a_2 \rightarrow (\lambda_3 :: a_3 \rightarrow a_4))) \\
def\ curry3 :: \lambda \rightarrow (\lambda_1 :: Param_1 \rightarrow (\lambda_2 :: Param_2 \rightarrow (\lambda_3 :: Param_3 \rightarrow \\
\quad (\lambda\ \ Param_1\ \ Param_2\ \ Param_3))))
\end{array}
$$

When the `curry3` function is called, it returns a function λ_1 that takes the first parameter of the λ. The function that returns will subsequently return a function λ_2 that takes the second parameter. This process continues to receive the 3rd parameter. The final function λ_3 when called will finally perform the λ function on the 3 parameters previously received.

This generic `curry3` function can be written in Erlang and used as follows:

```
add3(Param1,Param2,Param3) -> Param1 + Param2 + Param3.
curry3(Lambda) -> (fun(Param1) -> (fun(Param2) -> (fun(Param3) -> Lambda(Param1, Param2, Param3)

test_curry() ->
    14 = (((curry3(fun add3/3))(2))(5))(7),

    Add3_StartWith2And5 = ((curry3(fun add3/3))(2))(5),
    14 = Add3_StartWith2And5(7),
    17 = Add3_StartWith2And5(10),
    ok.
```

Problem Set 2

1. Implement the `curry3` function described above. Using the `curry3` function, curry the `alert` function provided in the starting code. Test your code with the code provided.
2. Save and use an intermediate function from the curried `alert` function that contains the location (first parameter). Test your code by calling your intermediate function twice with different Category (second parameter) and Message (third parameter) values. You can make up the values for these last two parameters in your test code.
3. Save and use an intermediate function that contains the location and category (first and second parameter). Test your code by calling your intermediate function twice with different Message values.
4. The provided `range_check` function takes a range (specified by the first two parameters `low` and `high`) and a value to test. This function is not compatible with the `filter` function because it has arity 3. Use the `curry3` to generate a function that checks if a value is in the specific range [10,20]. Apply this new function to the list provided in the provided starting code to find all numbers in the range [10,20] using the `filter` function.

5.3 Partial Applications

When we curried a function, we composed each parameter one at a time. We did this by creating functions that handled each parameter one at a time. Another approach is to split the function up into behavior and compose the behaviors one at a time. Just like currying, we can save intermediate functions. If the functions include behaviors, then these intermediate functions can have calculated results within them for reuse. We call this a partial application. While there are some similarities, please note that partial applications and currying is different.

Consider the following function which performs a map, filter, and fold (in order) on a range from 1 to $Value$. Note that the definition is more complicated and the = is used to save intermediate results as each step is performed. This type of function has several parameters. Note that the `range` function is assumed to exist which will create a list of numbers from 1 to the specified value.

$$
\begin{array}{l}
spec\ \ \lambda_{map} :: int \rightarrow real. \\
spec\ \ \lambda_{filter} :: real \rightarrow boolean. \\
spec\ \ \lambda_{fold} :: real\ \ real \rightarrow real. \\
spec\ \ map\_filter\_fold :: integer\ \ \lambda_{map}\ \ \lambda_{filter}\ \ real\ \ \lambda_{fold} \rightarrow real. \\
def\ \ map\_filter\_fold :: Value\ \ \lambda_{map}\ \ \lambda_{filter}\ \ FoldInit\ \ \lambda_{fold} \rightarrow \\
\quad List = (range\ \ Value), \\
\quad MapList = (map\ \ \lambda_{map}\ \ List), \\
\quad FilterList = (filter\ \ \lambda_{filter}\ \ MapList), \\
\quad FoldResult = (foldl\ \ \lambda_{fold}\ \ FoldInit\ \ FilterList), \\
\quad FoldResult.
\end{array}
$$

Every time we call this function, we will have to do the initial map and filter again. If we using a common set of data, then this could be wasteful. Like currying, creating a partial application has the benefit of creating functions with fewer parameters. In a partial application, we will split this function into multiple functions in which each smaller function will perform part of the larger function and return a function to do the rest. The function that is returned will contain the partial results from the smaller function.

Here is a new version of `map_filter_fold` that uses partial applications:

$$
\begin{array}{l}
spec\ \ \lambda_{map} :: int \rightarrow real. \\
spec\ \ \lambda_{filter} :: real \rightarrow boolean. \\
spec\ \ \lambda_{fold} :: real\ \ real \rightarrow real. \\
spec\ \ map\_filter\_fold2 :: integer \rightarrow (\lambda_1 :: \lambda_{map} \rightarrow (\lambda_2 :: \lambda_{filter} \rightarrow \\
\quad (\lambda_3 :: real\ \ \lambda_{fold} \rightarrow real)) \\
def\ \ map\_filter\_fold2 :: Value \rightarrow \\
\quad List = (range\ \ Value), \\
\quad \lambda_1 :: \lambda_{map} \rightarrow \\
\qquad MapList = (map\ \ \lambda_{map}\ \ List), \\
\qquad \lambda_2 :: \lambda_{filter} \rightarrow \\
\qquad\quad FilterList = (filter\ \ \lambda_{filter}\ \ MapList), \\
\qquad\quad \lambda_3 :: FoldInit\ \ \lambda_{fold} \rightarrow \\
\qquad\qquad FoldResult = (foldl\ \ \lambda_{fold}\ \ FoldInit\ \ FilterList), \\
\qquad\qquad FoldResult.
\end{array}
$$

The code is given below.

```
map_filter_fold2(Value) ->
    List = lists:seq(1, Value),
    fun (MapL) ->
```

```
        MapList = lists:map(MapL, List),
        fun (FilterL) ->
            FilterList = lists:filter(FilterL, MapList),
            fun (FoldInit, FoldL) ->
                FoldResult = lists:foldl(FoldL, FoldInit, FilterList),
                FoldResult
            end
        end
    end.
```

With this code, we can create reusable partial application functions that do part of the processing:

```
map_filter_fold_test() ->
    % Useful lambdas for the map_filter_fold2 to use.
    Square = fun(X) -> X * X end,
    Triple = fun(X) -> 3 * X end,
    Odd = fun(X) -> X rem 2 == 1 end,
    Even = fun(X) -> X rem 2 == 0 end,
    Sum = fun(X,Y) -> X+Y end,
    Product = fun(X,Y) -> X*Y end,

    % Example not using any partial applications
    35 = (((map_filter_fold2(6))(Square))(Odd))(0, Sum),

    % Example creating partial applications
    First10Squares = (map_filter_fold2(10))(Square),
    First20TriplesOnlyEven = ((map_filter_fold2(20))(Triple))(Even),

    % Using the partial applications
    165 = (First10Squares(Odd))(0, Sum),
    14745600 = (First10Squares(Even))(1,Product),
    270 = First20TriplesOnlyEven(0,Sum),
    3656994324480 = First20TriplesOnlyEven(1,Product),

    ok.
```

Problem Set 1

1. Implement the `map_filter_fold2` code above. Test the function with the test code provided in the starting code. Write an additional partial application function that obtains the first 20 triples that are even (uses all parts of `map_filter_fold2` except the application of the fold and save into the variable `First20TriplesOnlyEven` provided in the test code). Test your partial application function to calculate both the sum of all of the numbers (it should result in 330) and the product of all of the numbers (219419659468800).

2. Review the `process_dataset` function (there is some test code you can run as well) which has input parameters as follows:

   - Filename - The dataset csv file. You will use the file called `weather.csv` You can download that file here: weather.csv
   - ColumnId - The column to process (first column is designated as column 1)
   - ColumnType - Either the atom `text` or `int` or `float` so the proper formatting is done
   - CalcL - One arity function used to aggregate all the values in the column extracted. Two functions have been created for you already for use including `list_average` and `list_text_count`.

   Using this `process_dataset` function, create a `process_dataset2` function that will allow partial applications to be created. The new `process_dataset2` should split up the functions in the following ways:

   - Read the entire dataset from a user supplied file using the `read_csv_file` function
   - Extract a column from the dataset using a user supplied column number and column type using the `extract_column_array` function.

   - Perform a user supplied function on the column

   Code is provided in the starting code to test your new `process_dataset2` function.

Chapter 6

Monoids and Monads

In this lesson we will learn about two related design patterns: Monoids and Monads. Like Functors, these design patterns are based on mathematical principles that are represented frequently in many programming languages including functional ones.

6.1 Monoids

Monoids are functions that combine things using a binary operation that have the following properties:

- Closure - When we combine two things with our function, we always get something of the same type.
- Associativity - When we can combine more than two things, it does not matter which two things we combine together first. Mathematically this is written as $(A + B) + C = A + (B + C)$. It is important not to confuse this with commutativity which allows you to change the order like $A+B = B+A$. Monoids do not need to satisfy the commutative property.

- Identity - There is always something we can combine with that will retun the original value using our function.

In programming, we have already used monoids several times including: * Adding numbers * Combining Strings * Combining Lists

Frequently the following operators are monoids: $+$, $++$, $*$, $\wedge$, $\vee$

Due to the closure property, we are able to combine things easily with our functions using chaining because the input and output interfaces are compatible.

Due to the associative property, we can divide the work up into individual combinations and even use threading to complete them in parallel on a multi-processor platform. Due to the identity we can deal with special cases (e.g. add no numbers, work with an empty list).

As an example, we will rewrite the ability to combine (i.e. concatenate) two lists. Notice in the definition below we are purposefully not using ++ because this `combine_list` function is actually trying to implement ++.

```
spec combine_list :: [a] [a] → [a].
def combine_list :: [] List → List;
def combine_list :: List [] → List;
def combine_list :: [First|Rest] [] →
[First|(combine_list List Rest)]
```

The first part of the definition provides support for the identity property. Observe that this function can be chained together in different pairs (associative property) and the result is the same.

```
combine_list(List, []) -> List;

combine_list([], List) -> List;

combine_list([First|Rest],List) -> [First | combine_list(List,Rest)].

test_combine_list() ->
    [1, 2, 3, 4, 5, 6] = combine_list([1,2,3], [4,5,6]),

    % Test Associate Property
    L1 = [1, 2],
    L2 = [3, 4],
    L3 = [5, 6],

    % L1 and L2 first, then L3
    [1, 2, 3, 4, 5, 6] = combine_list(combine_list(L1, L2), L3),

    % L1 with the result of L2 and L3 done first
    [1, 2, 3, 4, 5, 6] = combine_list(L1, combine_list(L2, L3)),

    ok.
```

Since we are combining things, monoids frequently work well with a `fold`. In Erlang, the `fold` function expects the lambda (in our case the `combine_list`

monoid) to have the accumulator as the second parameter. Note that the identity property came in handy here. Note that in this case, our accumulator is the first list being passed in. We will need to take this into account in our `foldl` lambda. We could swap the parameters in `combine_list` but this will cause some confusion later in the week when we write a combine function for a new list data structure we will create. Additionally, having the second parameter be the first list can cause confusion when using the function.

```
test_combine_list() ->
    L1 = [1, 2],
    L2 = [3, 4],
    L3 = [5, 6],
    [1, 2, 3, 4, 5, 6] = lists:foldl(fun (Value,Acc) -> combine_list(Acc,Value), [], [L1, L2, L3]
    ok.
```

Comparing with other languages, monoids should seem familiar to operator overloading or overriding. The ability to define what it means to perform an operator on anything. While Monoids are restricted primarily to combining operators, the principle is closure is still important. For example, consider a function that combines two accounts. Each account is defined by an owner, an identifier, and a balance. We could write a function that does the following:

$$struct\ \ account\ \ \{string : Owner, string : Identifier, real : Balance\}.$$
$$spec\ \ combine\_accounts :: account\ \ account \rightarrow real.$$
$$def\ \ combine\_accounts :: Account_2\ \ Account_1 \rightarrow Account_1.Balance + Account_2.Balance.$$

However, with this definition, we don't have closure since our inputs are Accounts and our output is a Balance. When developing a monoid, it is important to return a compatible type. In this case, when we combine the accounts, we should return an account. In our example below, we will choose to keep the owner and identifier from the account that had the highest balance when recreating our Account (which we are representing by a tuple of size 3).

$$
\begin{array}{l}
struct\ \ account\ \ \{string : Owner, string : Identifier, real : Balance\}. \\
spec\ \ combine\_accounts :: account\ \ account \rightarrow account \\
def\ \ combine\_accounts :: Account_1\ \ nil \rightarrow Account_1; \\
def\ \ combine\_accounts :: nil\ \ Account_2 \rightarrow Account_2; \\
def\ \ combine\_accounts :: Account_1\ \ Account_2 \rightarrow \\
\quad \{Account_1.Owner, Account_1.Identifier, Account_1.Balance + \\
Account_2.Balance\} \\
\quad \text{when}\ \ Account_1.Balance \geq Account_2.Balance; \\
def\ \ combine\_accounts :: Account_1\ \ Account_2 \rightarrow \\
\quad \{Account_2.Owner, Account_2.Identifier, Account_1.Balance + \\
Account_2.Balance\}.
\end{array}
$$

Problem Set 1
You can find the template for the problem sets in this lesson here: prove06.erl

1. Implement the `combine_acounts` function as described above. Demonstrate that the Associative Property works to combine 3 accounts together.
2. Use a `foldl` to demonstrate the ability to combine 5 accounts. Use the test code provided in the starting code.
3. A monoid can be used to combine anything. In a functional language, that includes functions. Write a monoid that combines two functions of arity 1. Call your combination function `combine_functions`. Remember that `combine_functions` must satisfy the closure property which means that it must also return a function with arity 1. It should also satisfy the identify property (combine a function with `nil`). Test your code to show that it works with the Associative property. Use the 1 arity Erlang functions `to_upper`, `trim`, and `reverse` for testing your monoid (provided in the test code). Consider an input string such as " abCDef " which would result in "FEDCBA" after all 3 functions are applied.
4. Use `foldl` to combine all three functions together from the previous problem. Use the test code provided in the starting code.

6.2 Monads

When we consider the closure property we saw with the Monoid, it may not always be practical to have the data types be consistent between inputs and

outputs. Some functions may need to return additional data (i.e. meta-data) which is of a different type. A Monad is a type that contains additional data that is part of strategy to simplify or generalize our software. This meta-data can be used for several purposes including saving error and state information.

A common Monad seen in software is called the `Maybe`. A `Maybe` type is a tuple of size 1 or 2. The first value in the tuple is either `ok` or `fail`. The second value is an actual value (or result) and is only provided if the first value was `ok`. We call the `ok` or `fail` meta-data. Meta-data is information that describes our primary data.

In this example, the `use_inventory` function will return a `Maybe`. The result will be `ok` if there was inventory available to use and it will return `fail` if there was no inventory available. The meta-data `ok` and `fail` describe the inventory value.

$$
\begin{array}{l}
struct\ \ maybe \\
\quad \{atom(ok), a :: Value\}\ \ or \\
\quad \{atom(fail)\}. \\
spec\ \ use\_inventory :: integer \rightarrow maybe \\
def\ \ use\_inventory :: Inventory \rightarrow \{fail\}\ \ \text{when}\ \ Inventory == 0; \\
def\ \ use\_inventory :: Inventory \rightarrow \{ok, Inventory - 1\}.
\end{array}
$$

Here is the code implementation for our `use_inventory` function:

```
use_inventory(Inventory) when Inventory == 0 -> {fail};
use_inventory(Inventory) -> {ok, Inventory - 1}.
```

In this example, `use_inventory` returns a `Maybe` instead of a number which means we don't have closure which affects are ability to chain properly. Monad types include helper functions to adapt the output of a function (i.e. `Maybe`) to the input of a function (i.e. number). Each Monad must include a type constructor (often called `unit`) and a `bind` function described below.

The type constructor will provide a way to create a Monad type using a given value. Here is the type constructor for our `Maybe` type:

$$
\begin{array}{l}
spec\ \ maybe\_unit :: a \rightarrow maybe. \\
def\ \ maybe\_unit :: Value \rightarrow \{ok, Value\}.
\end{array}
$$

Here is the code for the `Maybe` type constructor:

```
maybe_unit(Value) -> {ok, Value}.
```

The bind will provide way to apply a function to a Monad type. If the `Maybe` is `ok`, then it will extract the value and apply the function to it. Note that the return of the function will be a `Maybe` again. If the `Maybe` is `fail`, then it will not run the function but instead just return a `Maybe` showing `fail`. The `bind` function will manage the meta data of `ok` and `fail` in our Monad. The use of the bind function allows us to chain functions that return the `Maybe` Monad type. Here is the bind definition:

$$
\begin{array}{l}
spec \;\; \lambda :: a \rightarrow maybe. \\
spec \;\; maybe\_bind :: maybe \;\; \lambda \rightarrow maybe. \\
def \;\; maybe\_bind :: \{ok, Value\} \;\; \lambda \rightarrow (\lambda \;\; Value); \\
def \;\; maybe\_bind :: \{fail\} \;\; \lambda \rightarrow \{fail\}.
\end{array}
$$

Here is the code for the `Maybe` bind:

```
maybe_bind({ok, Value}, Lambda) -> Lambda(Value);
maybe_bind({fail}, _) -> {fail}.
```

Using our `maybe_unit` and `maybe_bind` we can chain our `use_inventory` function.

```
test_maybe() ->
    {ok, 8} = maybe_bind(maybe_bind(maybe_unit(10), fun use_inventory/1), fun use_inve
    {fail} = maybe_bind(maybe_bind(maybe_unit(1), fun use_inventory/1), fun use_invento

    ok.
```

For each Monad we create, we may need one or more bind functions to provide the proper support for the unique meta data in our Monad. In the problem set below, you will write a bind function that will combine meta data from previous bind executions.

Problem Set 2

1. Write a function called `cut_half` which will divide a number by 2 and returns a `Maybe` (as defined above) with an integer (not floating point) answer. If the remainder after division is not 0 (i.e. fractional) or if the number was less than or equal to 1, then the `Maybe` should indicate `fail`. Implement the `maybe_bind` and `maybe_unit` functions above and use the code in the template for testing.
2. We will define a Monad called `Result` which is either `{ok}` or `{error, List of Error Messages}`. You will use this `Result` Monad to perform checks on a password. The `check_mixed_case`, `check_number_exists`, and `check_length` functions provided to you already use this `Result` Monad type. Notice that each function returns a list of size 1 when an `error` is returned. You need to write the `result_unit` and `result_bind` functions. The bind function should save all previous reasons in the Result. For example, if the password provided to these functions was "simple", and you chained together all 3 checking functions, then the `bind` would result in an `error` result containing a list of all 3 error strings. Use the code in the template for testing.
3. Modify the test code provided in the previous problem to chain the bind functions together using a `foldl` instead.

6.3 List Monad

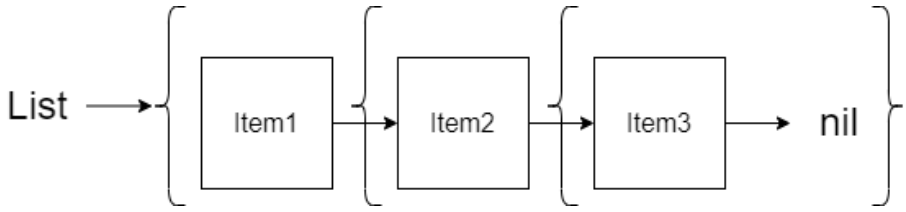
In a few weeks we will begin our exploration of data structures within a functional language. Reflecting back earlier in the semester, we saw how the list was used in Erlang. One thing that we have not yet explored is how to implement a length function for our list. Even though Erlang has a built-in length function which is good to always use, implementing the length can be a good example for exploring the Monad further.

Before we look at the Monad, we are going to define our lists a little bit differently from previous weeks. Instead of using the built-in list data type in our languages, we are going to create our list completely from scratch. Doing this will prepare us for the data structures we will see in the future.

Let's define a list using tuples:

- Empty List: `nil`
- One Item List: `{Item1, nil}`
- Two Item List: `{Item1, {Item2, nil}}`

- Three Item List: `{Item1, {Item2, {Item3, nil}}}`



The first item in each tuple represents the value in the list and the second item in the tuple represents the remainder of the list. This is very similar to the $[First|Rest]$ format we often write where $First$ represents the first value in the list and the $Rest$ represents the remainder of the list.

For simplicity, we will define a push and pop function which will only affect the front of our list.

$$struct\ \ list\ \ \{a : First, list : Rest\}.$$

$$spec\ \ push :: a\ \ list \rightarrow list.$$
$$def\ \ push :: Value\ \ List \rightarrow \{Value, List\}.$$

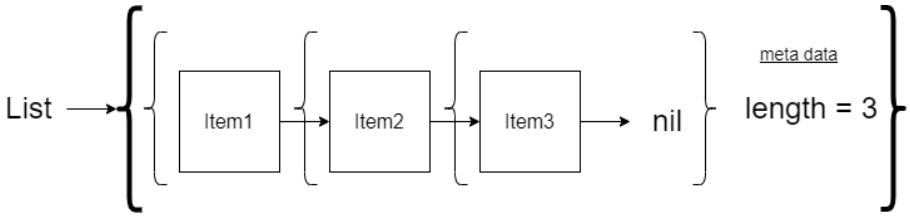
$$spec\ \ pop :: list \rightarrow list.$$
$$def\ \ pop :: nil \rightarrow nil$$
$$def\ \ pop :: \{First, Rest\} \rightarrow Rest.$$

Implementing this code we have the following:

```
push(List, Value) -> {Value, List}.

pop(nil) -> nil;
pop({_First, Rest}) -> Rest.
```

If we wanted to calculate the length function, we would need to visit every item in our list to count. To avoid this, we can store the length as meta-data using a Monad. We will define our list Monad type to be: `{List, Length}`. Using this type, our example lists now look like this:

- Empty List: `{nil, 0}`
- One Item List: `{{Item1, nil}, 1}`
- Two Item List: `{{Item1, {Item2, nil}}, 2}`
- Three Item List: `{{Item1, {Item2, {Item3, nil}}}, 3}`



Recall that Monad functions don't take the Monad type as an input but they should return the Monad type. That will mean that when we call our `push` and `pop` functions, the length must be stripped off. How does the `push` and `pop` know what to set the length to in the result? Remember that the `bind` function is responsible for managing the meta data. We will have the `push` and `pop` return a delta length. For `push`, the delta length will be positive 1 and for `pop`, the delta length will be negative 1. We will have the `bind` manage the length meta-data.

In our specifications and definitions below, we will modify the `push` and `pop` to use the Monad type and also include the type constructor (in this case we will use a more interesting name like `create` instead of `unit`) for creating an empty list (length of 0) and the binding function. Since we will want the ability to see the list and length independent of the Monad type, we will provide some simple helper functions (`value` and `len`) for these as well.

The bind will be more complex because it will need to handle both `push` and `pop` functions which have different arity. The $[a]$ (called `Optional_Parameters` in the definition) contains the variable number of parameters to pass to `push` and `pop`. We will use a traditional Erlang list data structure to store these parameters.

$$struct\ \ list\ \ \{a : First, list : Rest\}.$$
$$struct\ \ m\_list\ \ \{list : List, integer : Length\}.$$

$$spec\ \ push :: a\ \ list \rightarrow m\_list.$$
$$def\ \ push :: Value\ \ List \rightarrow \{\{Value, List\}, 1\}.$$

$$
\begin{array}{l}
spec\ \ pop :: list \rightarrow m\_list. \\
def\ \ pop :: nil \rightarrow (create); \\
def\ \ pop :: \{First, Rest\} \rightarrow \{Rest, -1\}.
\end{array}
$$

$$
\begin{array}{l}
spec\ \ create :: \quad \rightarrow m\_list. \\
def\ \ create ::\rightarrow \{nil, 0\}.
\end{array}
$$

$$
\begin{array}{l}
spec\ \ \lambda_{bind} :: list \rightarrow m\_list. \\
spec\ \ bind :: m\_list\ \ \lambda_{bind}\ \ [a] \rightarrow m\_list. \\
def\ \ bind :: \{List, Length\}\ \ \lambda_{bind}\ \ Optional\_Parameters \rightarrow \\
\quad \{New\_List, Delta\_Length\}\ =\ (\lambda_{bind}\ \ Optional\_Parameters\ + \\
+List), \\
\quad \{New\_List, Length + Delta\_Length\}.
\end{array}
$$

$$
\begin{array}{l}
spec\ \ value :: m\_list \rightarrow list. \\
def\ \ value :: \{List, Length\} \rightarrow List. \\
spec\ \ len :: m\_list \rightarrow integer. \\
def\ \ len :: \{List, Length\} \rightarrow Length.
\end{array}
$$

When implemented, you will need to use the `apply` function in Erlang to handle passing the `List` and `Optional_Parameters` to the λ_{bind} function. If `push` is used, then the `Optional_Parameters` will be a one element list containing the value to push. If `pop` is used, the `Optional_Parameters` will be an empty list. Some of these functions are implemented below and some are left as an exercise.

```
push(Value, List) -> {{Value, List},1}.

% pop is left as an exercise

create() -> {nil, 0}.

bind({List, Length}, Function, Optional_Parameters) ->
    LEFT FOR AN EXERCISE
```

```
len({_List, Length}) -> Length.
value({List, _Length}) -> List.
```

For the length get updated, we have to use the bind function as shown in the test code below.

```
L1 = create(),
L2 = bind(L1, fun push/2, [2]),
L3 = bind(L2, fun push/2, [4]),
L4 = bind(L3, fun push/2, [6]),
L5 = bind(L4, fun pop/1, []),
```

Problem Set 3

1. Implement the `pop` and `bind` function with the list Monad type. Use the code in the template to test. Note that the `apply` function is available in Erlang which will call a function and pass a variable list of arguments to the function. Use the code in the template to test.

2. Sometimes a function uses more than one Monad type. Since its possible for the `pop` to fail in the case of an empty list, it might be a good to use the `Maybe` monad type so we can return `{fail}` in that case. In the case of the `push`, we don't have any reason to return `{fail}`. However, to support the `bind`, we will be consistent and return the `Maybe` monad type. The `bind` needs to handle the case where one of our functions returns `{fail}`. When this happens, an empty list should be created and returned back (which is what `pop` currently does). We need to write three new functions: `push2`, `pop2`, and `bind2` that support `Maybe`. The `push2` function is already written for you. Use the specifications below and the tests provided in the starting code. The `bind2` function will likely need to use the `case of` block to determine what to do based on the result of running the λ_{bind2} function.

$$
\begin{array}{l}
struct\ \ maybe\_m\_list \\
\quad \{atom(ok), m\_list : Value\}\ \ or \\
\quad \{atom(fail)\}. \\
spec\ \ push2 :: a\ \ list \rightarrow maybe\_m\_list. \\
spec\ \ pop2 :: list \rightarrow maybe\_m\_list. \\
spec\ \ \lambda_{bind2} :: list \rightarrow maybe\_m\_list. \\
spec\ \ bind2 :: m\_list\ \ \lambda_{bind2}\ \ [a] \rightarrow m\_list.
\end{array}
$$


BY

Chapter 7

Streams and Lazy Evaluation

In this lesson we will learn about the Stream Design Pattern. Streams are things that provide you values when you need them. They are an example of the unfold functor pattern. This "lazy" approach can actually save you a lot of time and memory when you only need something one at a time.

7.1 Streams

A stream provides information to us one item at a time whenever we need it. This implies that when writing a stream, we need to consider two things:

- Provide the user a function to call when they are ready to get the next thing.
- The function that we provide must be aware of what the previous thing was so it can give the next thing.

Let's look at both of these needs separately. If we wanted to provide the user with the ability to add 1 to a number at any time, we could write a function that return a function to be called later.

$$spec\ \ lazy\_incr\_once :: integer \to (\lambda ::\to integer).$$
$$def\ \ lazy\_incr\_once :: Value \to (\lambda ::\to Value + 1).$$

```
lazy_incr_once(X) -> fun() -> X + 1 end.

test() ->
    L = lazy_incr_once(4),
    5 = L(),
    5 = L(),
    ok.
```

In the code above, we are running the function that was returned when we called `lazy_incr_once`. Combine this concept with the desire to have the function return something based on the previous result. In the `lazy_incr` function, we will return two things: the next value and the next function to call. Notice that we are defining a custom data type called `iterator` which will contain both of these things. Our ultimate goal by the next lesson is to define this as a Monad.

$$struct\ \ iterator\ \ \{a, \lambda_{stream}\}.$$
$$spec\ \ \lambda_{stream} ::\rightarrow iterator.$$
$$spec\ \ lazy\_incr :: integer \rightarrow \lambda_{stream}.$$
$$def\ \ \ lazy\_incr\ \ :: \ \ Value\ \ \rightarrow\ \ (\lambda_{stream}\ \ ::\rightarrow\ \ \{(Value\ +\ 1), (lazy\_incr\ \ (Value + 1))\}).$$

```
lazy_incr(Value) -> fun() -> {Value + 1, lazy_incr(Value+1)} end.

test() ->
    L = lazy_incr(4),
    {5, L2} = L(),
    {6, L3} = L2(),
    {7, L4} = L3(),
    ok.
```

Another classic stream is the range function which will provide a sequence of numbers but only one at a time. In this function, we will assume that the sequence will from `Start` to `Stop`, inclusive and that `Start <= Stop`. Unlike the previous example, this iterator will need to stop. We will modify our new `fixed_iterator` to have other possible values.

$$
\begin{array}{l}
struct\ \ fixed\_iterator \\
\quad \{a, \lambda_{stream}\}\ \ or \\
\quad \{atom(undefined), atom(done)\}. \\
spec\ \ \lambda_{stream} ::\rightarrow fixed\_iterator. \\
spec\ \ range :: integer\ \ integer \rightarrow \lambda_{stream}. \\
def\ \ range :: Start\ \ Stop \rightarrow (\lambda_{stream} ::\rightarrow \{Start, (range\ \ (Start + \\
1), Stop)\}) \\
\quad \text{when}\ \ Start \leq Stop; \\
def\ \ range :: Start\ \ Stop \rightarrow (\lambda_{stream} ::\rightarrow \{undefined, done\}).
\end{array}
$$

The erlang code is given below:

```
range(Start, Stop) when Start =< Stop ->
    fun () -> {Start, range(Start+1, Stop)} end;
range(_Start, _Stop) ->
    fun () -> {undefined, done} end.
```

We can simplify the above code by moving the guards into the anonymous function as shown below:

```
range(Start, Stop) ->
    fun () when Start =< Stop -> {Start, range(Start+1, Stop)};
        () -> {undefined, done} end.
```

An example of the test code using the `range` function is given below. Notice that the value of `{undefined, done}` is returned when the stream is completed.

```
test() ->
    Stream1 = range(1,4),
    {1, Stream2} = Stream1(),
    {2, Stream3} = Stream2(),
    {3, Stream4} = Stream3(),
    {4, Stream5} = Stream4(),
    {undefined, done} = Stream5(),
    ok.
```

Problem Set 1
You can find the template for the problem sets in this lesson here: prove07.erl

1. Modify the `range` function to take a third parameter called `step`. You should increase the sequence by the amount of `step`. If the `step` is negative, then create a decreasing sequence. If the `step` is 0, then the `done` state should occur immediately. Test code is provided to test your stream.
2. Create a new stream called `words` which will split text by spaces into sub-strings Each time the stream is called, it should return back the next string. For example, if the text was "The cow jumped over the moon.", then the stream would first return "The", then "cow", then "jumped", and so forth. Use the `fixed_iterator` Monad type described above. You are provided a function called `first_word` which returns a tuple in the format `{Word, Rest}` where `Word` is the first word and `Rest` is the remaining text (just like working with values in a list). Test code is provided to test your stream.

7.2 Stream Monad & Collection

In the previous section we created a `fixed_iterator` Monad that the `range` function supported. To complete the Monad definition, we need a type constructor (or unit function) and a bind function. Since the type constructor will need to create a stream that hasn't started yet, we will modify our specification for the Monad Type we had earlier (the second condition in the struct is new):

$$
\begin{array}{l}
struct\ \ fixed\_iterator \\
\quad \{a, \lambda_{stream}\}\ \ or \\
\quad \{atom(undefined), \lambda_{stream}\}\ \ or \\
\quad \{atom(undefined), atom(done)\}. \\
spec\ \ \lambda_{stream} ::\rightarrow fixed\_iterator.
\end{array}
$$

The type constructor should be used to create the iterator the first time. We will call our type constructor `iter`. The `iter` will take as an input a λ_{stream} function (the function that is returned by our `range` function or any other stream). The `iter` will produce the $\{atom(undefined), \lambda_{next}\}$. This is the state of our iterator before we actually start the iteration.

$$spec\ \ iter :: \lambda_{stream} \rightarrow fixed\_iterator.$$
$$def\ \ iter :: \lambda_{stream} \rightarrow \{undefined, \lambda_{stream}\}.$$

The bind should allow us to take our `fixed_iterator` (the Monad type returned by the `iter` function) and pass it to our stream function which will return an updated `fixed_iterator`. Remember that our stream function is always regenerated and has no input parameters. The next function is re-created each time with the next value built-into the function. We will call our bind function `next` and it will be responsible for getting the next stream iteration. Unlike previous bind functions we have used, this bind will not take a function parameter because it is applied only to our λ_{stream}.

$$spec\ \ next :: fixed\_iterator \rightarrow fixed\_iterator.$$
$$def\ \ next :: \{Value, done\} \rightarrow \{undefined, done\};$$
$$def\ \ next :: \{Value, \lambda_{stream}\} \rightarrow (\lambda_{stream}).$$

Using these specifications and definitions, we can complete the `fixed_iterator` Monad along with helper functions (`value` and `lambda`) to access both parts of our Monad type.

```
iter(Stream) -> {undefined, Stream}.

next({_Value,done}) -> {undefined, done};
next({_Value,Lambda}) -> Lambda().

value({Value,_Lambda}) -> Value.
lambda({_Value,Lambda}) -> Lambda.
```

We can use our new functions and show that calling our bind function `next` will result in `undefined` if we complete our iterator.

```
Next1 = next(iter(range(1,4))),
io:format("~p~n",[value(Next1)]),  % Print 1
Next2 = next(Next1),
io:format("~p~n",[value(Next3)]),  % Print 2
Next3 = next(Next2),
io:format("~p~n",[value(Next3)]),  % Print 3
Next4 = next(Next3),
io:format("~p~n",[value(Next4)]),  % Print 4
Next5 = next(Next4),
```

```
io:format("~p~n",[value(Next5)]),  % Print undefined
Next6 = next(Next5),
io:format("~p~n",[value(Next6)]),  % Print undefined
```

A common function that is implemented when lazy streams are supported is the `collect`. The `collect` function will undo the "laziness" of our data structure by iterating through the entire λ_{stream} and put all the results in order into a traditional list.

The specification for the `collect` is given below:

$$spec \;\; collect :: \lambda_{stream} \rightarrow [a].$$
$$spec \;\; collect :: \lambda_{stream} \;\; [a] \rightarrow [a].$$

The second specification is intended to help build the list result recursively by calling the `next` function until the iterator is completed. The definition for the first specification for `collect/1`is given below. Notice that the function will call `iter` to convert the λ_{stream} into a `fixed_iterator` prior to doing the recursion to generate the list of values.

$$def \;\; collect :: Stream \rightarrow (collect(iter \;\; Stream) \;\; []).$$

The erlang code for `collect/1` is given below:

```
collect(Stream) -> (collect (iter ~ ~ Stream), []).
```

The implementation of the `collect/2` function is left for an exercise below.

Problem Set 2

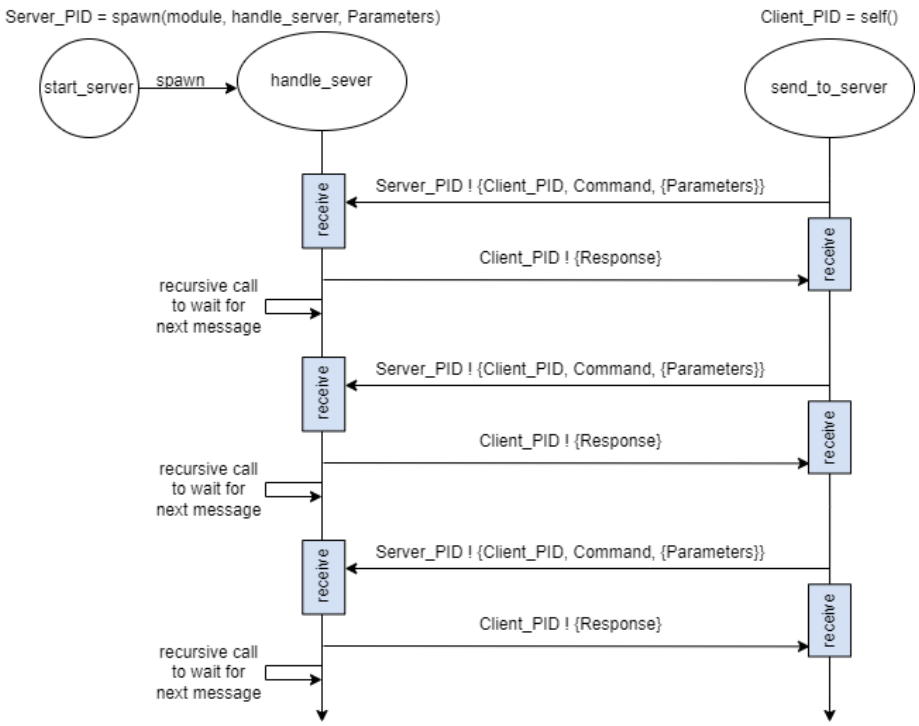
1. Implement the `collect/2` function and test it with the code provided. When writing the function, you should reuse the `next` function iterate the stream, reuse the `value` function to get the value to put in the `Result` list, and reuse the `lambda` function to compare with `done`. Test code using the `collect` function with `range` and `words` is provided.

7.3 Erlang Processes & Streams

Another common problem solved with streams is a unique ID generator. We could write the same code above to obtain a function that will return the next available number (i.e. prev number + 1). However, if we had multiple software components that needed to use the same unique ID generator, then we need a slightly different approach.

Erlang supports a scalable process system that allows for safe concurrent operations. This process system can be used locally with our software or it can be used along with networking code to support communication between different machines.

The diagram below shows how a client/server process architecture can be setup in Erlang. We will assume in our discussion that both the client and the server are running locally on our same machine.



In the diagram above, we have a process that is started (or spawned) by `start_server` and handles messages received by `handle_server`. The client uses the server process ID (PID) to send messages to the server and then waits for a response. All messages sent to the server are formatted as follows:

$\{Client\_PID, Command, \{Parameters\}\}$

Response back to the client are formatted as follows:

$\{Response\}$

This simple server has only two commands: * echo - Send the text back to the client * add - Add the two numbers and send back the answer to the client.

The server can handle only one request at a time. After the server handles the request, the `handle_server` function is called again to handle the next request. This ability to handle only one thing at a time will help us create the unique ID server later on. Let's first look at the Erlang code needed:

```
handle_server() ->
    receive
        {Client_PID, echo, {Text}} -> Client_PID ! {Text};
        {Client_PID, add, {X, Y}} -> Client_PID ! {X+Y}
    end,
    handle_server().

start_server() ->
    spawn(prove07, handle_server, []).

send_to_server(Server_PID, Command, Params) ->
    Server_PID ! {self(), Command, Params},
    receive
        {Response} -> Response
    end.
```

The `start_server` function will spawn a new process. When spawning a new process, the handler function is specified along with its module name and a list of any parameters. In this simple example, there are no parameters. Note that you have to export the `handle_server` function for the `spawn` function to work properly.

The `handle_server` function uses the `receive` command to receive a message from a client. When one is received and matches one of the patterns in the `receive` command block, then a response is sent back. In Erlang, a message is sent using the syntax: `PID ! Message`. We use the Client PID that we received to send the response back. The call to `handle_server` recursively at the end will allow the server to response to the next client.

The `send_to_server` function is a utility function to help us send a message to server and receive the response. When sending messages to the server, the Erlang function `self()` is called to obtain the client PID to send to the server. A test example is shown below:

```
2> Server_PID = prove07:start_server().
<0.85.0>
3> prove07:send_to_server(Server_PID, echo, {"Hello"}).
"Hello"
4> prove07:send_to_server(Server_PID, add, {13, 8}).
21
5>
```

If we wanted to create a unique ID server following this same format, we can modify our handler function to take a parameter that keeps track of the current ID. Remember that each call to the handler function is guaranteed to handle only one client at a time. Similar to the concept of the stream functions being created with the next value in the stream, our handler will create the next value to be used.

Here is the code for our new `start_id_server` and `handle_id_server`. Notice that the `handle_id_server` now has an input parameter and it is initialized in the spawn function within `start_id_server`. For the `export`, notice that we specify the arity: `handle_id_server/1`.

```
-export([handle_id_server/1]).

handle_id_server(Curr_ID) ->
    receive
        {Client_PID, id, {}} -> Client_PID ! {Curr_ID}
    end,
    handle_id_server(Curr_ID + 1).

start_id_server() ->
    spawn(prove07, handle_id_server, [0]).
```

When the `handle_id_server` is called recursively to handle the request from the next client, the `Unique_ID` is incremented by 1.

Reusing the same `send_to_server` function, we can generate new unique ID's as follows starting with 0.

```
5> ID_Server_PID = prove07:start_id_server().
<0.89.0>
6> prove07:send_to_server(ID_Server_PID, id, {}).
0
```

```
7> prove07:send_to_server(ID_Server_PID, id, {}).
1
8> prove07:send_to_server(ID_Server_PID, id, {}).
2
9> prove07:send_to_server(ID_Server_PID, id, {}).
3
10> prove07:send_to_server(ID_Server_PID, id, {}).
4
11> prove07:send_to_server(ID_Server_PID, id, {}).
5
12>
```

Problem Set 3

1. Modify the `handle_server` function to support an `avg` command that will average a variable collection of numbers. For example, if the parameters field is `{10,20,30,40}` then 25 will be returned back to the client. You can use the Erlang `lists:sum` and `length` functions.
2. Create a server based on the Unique ID server but with functions called `handle_running_avg_server` (remember that this one needs to be exported) and `start_running_avg_server` that maintains a running average of numbers submitted. There should be three commands:
   - add - Adds a number to the running average and returns the current average
   - remove - Removes a number from the running average and returns the current average
   - display - Returns the current average

   When implementing the running average server, you should save the current sum and current number of numbers in your server. You should not store the list of numbers. Test code is provided for you.

Chapter 8

Trees

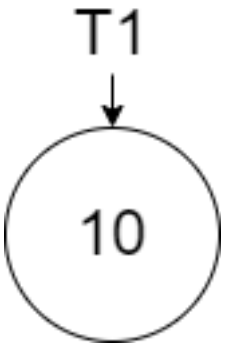
In this lesson we will learn about our second data structure called the Binary Search Tree. When we learned about persistence we learned about the list as it is implemented in Erlang. When we learned about monads, we introduced a new version of the list which was created using our own custom tuples. The Binary Search Tree will use this same custom approach.

8.1 Binary Search Tree

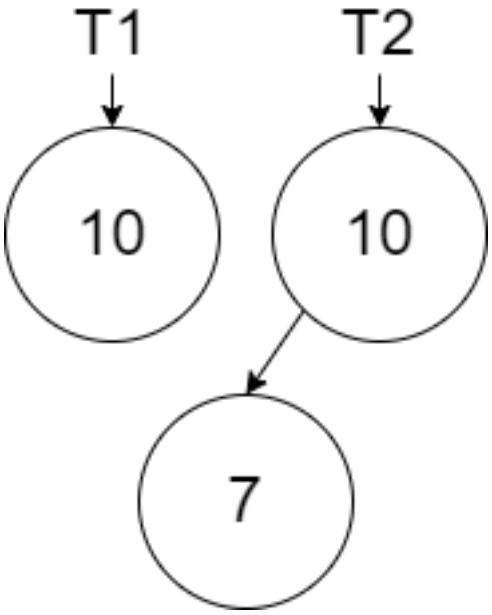
A binary search tree is a tree that stores smaller values to the left and larger values to the right. Each node in the tree in defined by 3 fields: value, link to the left sub-tree, and link to the right sub-tree.

The tree is useful because we can quickly search for values by asking ourselves, "is the value we are looking for larger or smaller (or maybe we found it) than the one in the current node of the tree?" This is like guessing a number between 1 and 100 and always guessing 50 first. Searching in the tree will ideally exclude half of the values with each comparison. We will explore this "ideal" performance in the next part of the lesson.

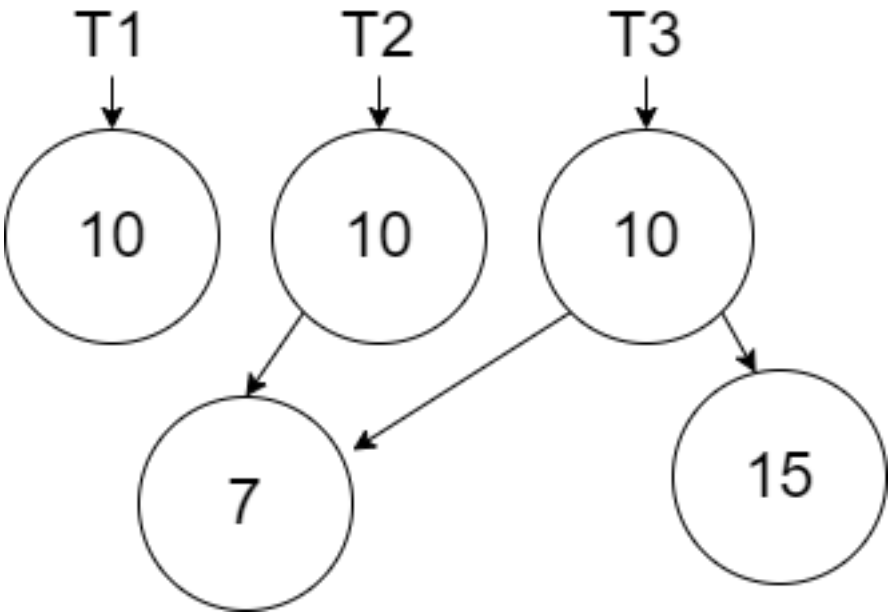
Persistence is a curious thing to consider visually with the tree. In the diagram below, we show the creation of a tree with one node valued 10 called T_1 .



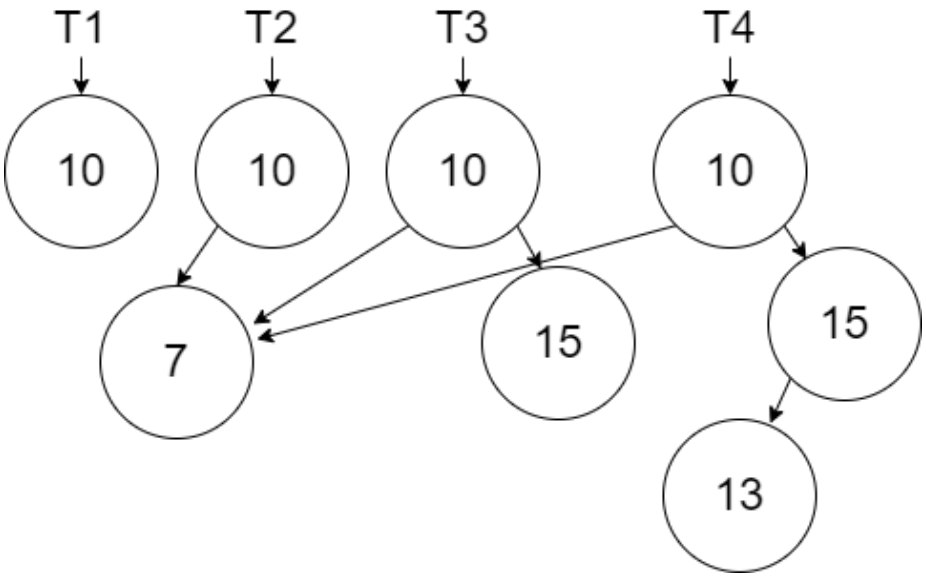
When we add a new node with value 7 (in this case something that will go left of the root node), we must persist T_1. Since the left link in the root node must change, then we need a new node to create T_2.



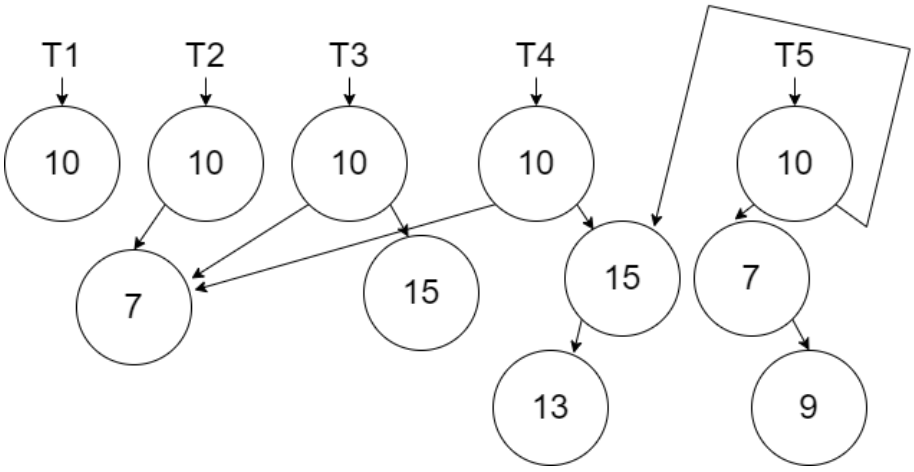
If we now add a node with value 15 to the right of the root node, we are only affecting the right link. The new node we created in the previous step can be reused in creating T_3.



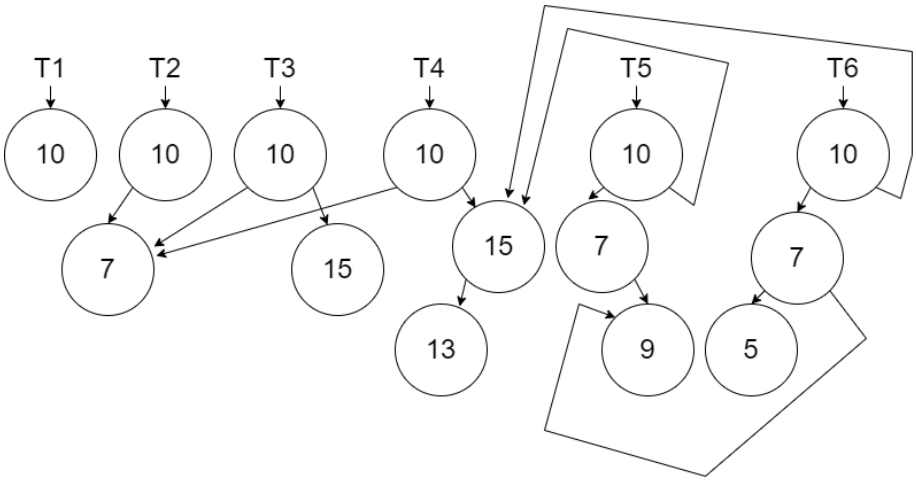
In the next diagram we show the creation of T_4 by adding the new node with value 13.



In the next diagram we show the creation of T_5 by adding the new node with value 9.



Finally, we show the creation of T_6 by adding the new new node 5.



The software will discard older versions of the Tree when they are not needed anymore. However, the persistence we use in the data structure is important to satisfy the immutability rules.

Notice that the there is always a path through the tree that we have to re-create from the root node down to the node that we added. When we reuse a node in the tree, we will reuse both the node and all the nodes it is connected too. We will not have to have use recursion when we reuse a node.

Let's begin with the structure of the node:

$$struct \quad node \quad \{a : Value, node : Left, node : Right\}.$$

We will implement two functions in our binary search tree. The `add` function will add a value to the tree and the `contains` function will search the tree for a value and return a true or false. When we add to a tree or we search in a tree, we will do so recursively through nodes beginning with the root node. When we add to a tree, the result will be an updated root node (or an updated sub-tree root node as we recursively create our new tree).

```
spec add :: a node -> node.
spec contains :: a node -> boolean.
```

In our specification, we are assuming that *a* represents a type that can be compared with boolean operators.

There are 4 scenarios when adding a new item to a tree: * Empty Tree - Create a new root `node`. We will define an empty tree as `nil`. * Value to add is less than the current node value - Create a new `node` reusing the right link. The left link is created by recursively calling `add` on the left sub-tree. * Value to add is greater than the current node value - Create a new `node` reusing the left link. The right link is created by recursively calling `add` on the right sub-tree. * Value to add is equal to the current node value - In our tree, we will require all values to be unique. In this case, we get to reuse the entire node including both left and right links. The tree can be implemented differently by allowing duplicates perhaps stored to the right.

The definition for `add` is given below. The implementation for both `add` and `contains` will be left for an exercise.

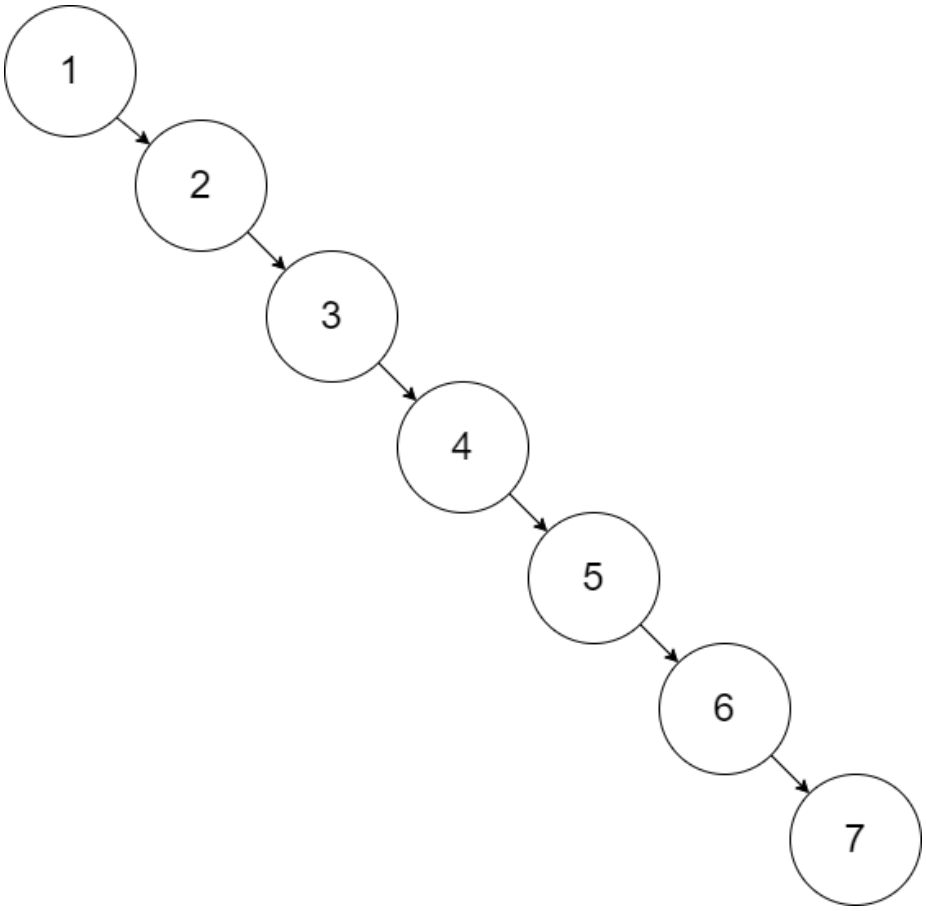
```
def add :: New_Value nil -> {New_Value, nil, nil};
def add :: New_Value {Value, Left, Right} ->
{Value, (add New_Value Left), Right} when New_Value <
Value;
def add :: New_Value {Value, Left, Right} ->
{Value, Left, (add New_Value Right)} when New_Value >
Value;
def add :: New_Value Node -> Node.
```

Problem Set 1
You can find the template for the problem sets in this lesson here: prove08.erl

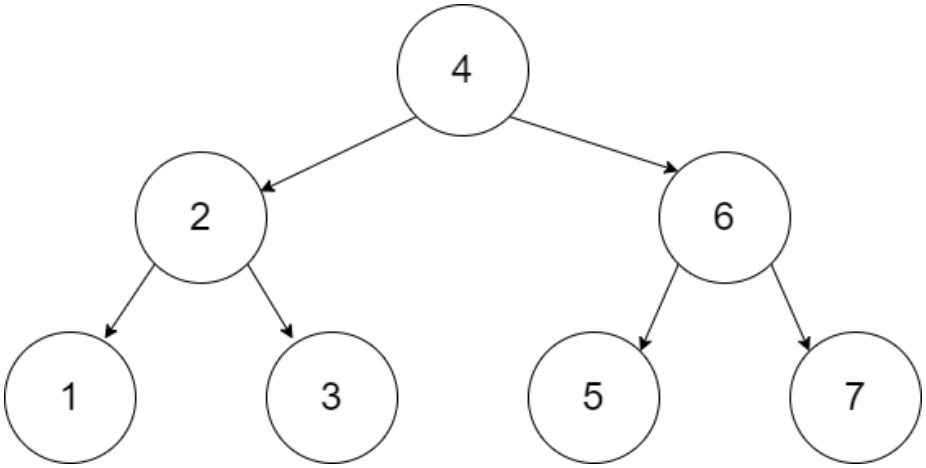
1. Implement the `add` function per the specification and definition above. Use the test code provided.
2. Implement the `contains` function per the specification above. Use the test code provided.

8.2 Balanced Red-Black Tree

Since a binary search tree orders left and right of node relative to the values, traversing the tree (either to insert or find) can provide an O(log n) performance. However, this can only be achieved if the tree is balanced. If we added the numbers 1, 2, 3, 4, 5, 6, 7 in order to a tree, they would also be add to the right. Searching for a number in the tree would be the same as searching a list which is O(n)



However, if the numbers were added 4, 2, 6, 1, 3, 5, 7 then we would get a tree that allows for the maximum exclusion of values every time we go left or right.



Since we can't rely on the order of the data, there are other strategies and algorithms that can be used. A common one is the Red Back Tree (RBT). The RBT uses a set of rules to determine if part of a tree is unbalanced. If something is unbalanced, a modification is made to balance the tree. In other words, after the creation of any node in the `add` function, we will potentially perform a `balance` function.

The rules for an RBT are as follows:

1. Every node is either red or black - In our algorithm we will choose to start all nodes out red.
2. The root is always black - In our algorithm we will choose to change the root node to black after the inserting and all balancing is completed.
3. No two adjacent nodes are red.
4. Every path from a node to a leaf has the same number of black nodes.

In our algorithm we will enforce rules and 3 and 4 with our balancing function.

To support these rules, the structure of the node must change to include the color of the node:

$$\begin{array}{l} struct\ \ node \\ \quad \{atom(red), a : Value, node : Left, node : Right\}\ \ or \\ \quad \{atom(black), a : Value, node : Left, node : Right\}. \end{array}$$

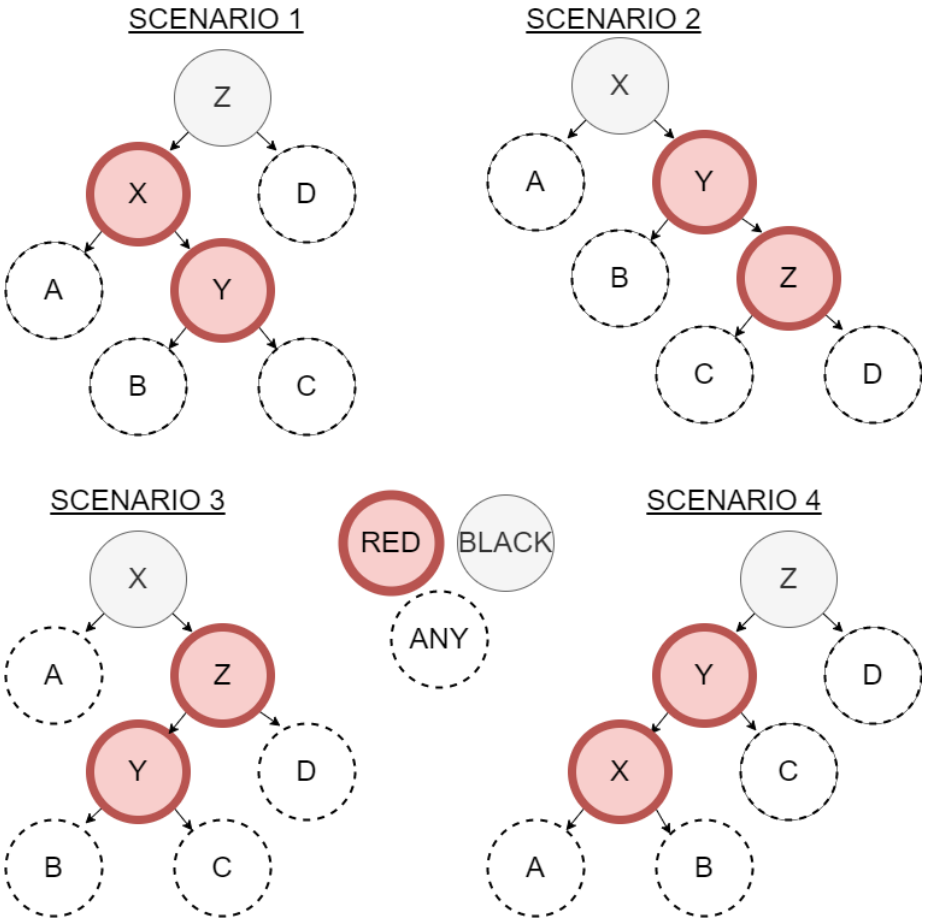
We will use the function names `add_rbt` and `contains_rbt` so they are unique from the traditional binary search tree. These `contains_rbt` is the same as `contains` as the color is only important for adding and therefore is unused in the `contains_rbt`.

The `add_rbt` function is very similar to the `add` with the following distinctions: * When we create a new node we color it red * When we create a new node that reuses one link and recursively adds the node on the other link, we need to call a `balance` function (discussed later) on the new node we created. * When we are done adding the node, we need to change the root node to be black. This will necessitate a helper function so that we have `add_rbt` and `add_rbt_` both.

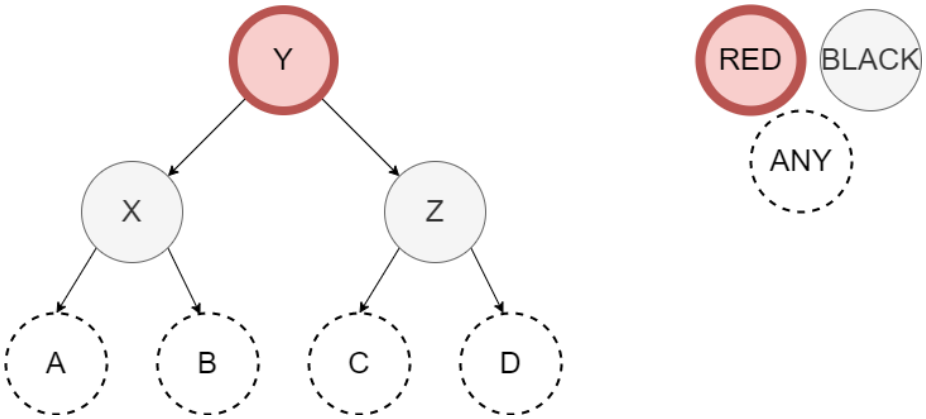
The specification and definition for the `add_rbt` is shown below:

```
spec add_rbt :: a node → node.
spec add_rbt_ :: a node → node.
def add_rbt :: New_Value Tree →
    New_Root = (add_rbt_ New_Value Tree),
    {black, New_Root.Value, New_Root.Left, New_Root.Right}.
def add_rbt_ :: New_Value nil → {red, New_Value, nil, nil};
def add_rbt_ :: New_Value {Color, Value, Left, Right} →
    (balance {Color, Value, (add_rbt_ New_Value Left), Right})
    when New_Value < Value;
def add_rbt_ :: New_Value {Color, Value, Left, Right} →
    (balance {Color, Value, Left, (add_rbt_ New_Value Right)})
    when New_Value > Value;
def add_rbt_ :: New_Value Node → Node.
```

The `balance` function that is being performed is based on pattern matching defined by an algorithm written by Chris Okasaki (reference Purely Functional Data Structures). In his algorithm, there are four unbalanced scenarios that need to be balanced. These adjustments are made when the `add_rbt_` function is returning from all the recursive calls back up to the new root. The diagram below shows the four scenarios.



The selection of variables in these diagrams was purposeful to suggest that $X < Y < Z$. We need code to look for each of these four scenarios. If we have any of these four, then the solution is the same shown below.



The specification for our balance is simple as it converts a potentially unbalanced node into a balanced node:

$$spec\ \ balance :: node \rightarrow node.$$

The definition for this function has 5 clauses (one for each scenario plus one in case none of them match. The first clause below corresponds to scenario 1. The next three scenarios are left for an exercise. The default clause is also given below.

$$\begin{array}{l} def\ \ balance :: \{black, Z, \{red, X, A, \{red, Y, B, C, \}\}, D\}) \rightarrow \\ \quad \{red, Y, \{black, X, A, B\}, \{black, Z, C, D\}\}; \\ def\ \ balance :: \text{Scenario 2 left for an exercise}; \\ def\ \ balance :: \text{Scenario 3 left for an exercise}; \\ def\ \ balance :: \text{Scenario 4 left for an exercise}; \\ def\ \ balance :: Node \rightarrow Node. \end{array}$$

The Erlang code for the definitions provided above is given below. The remaining code is left for an exercise.

```
balance({black,Z,{red,X,A,{red,Y,B,C}},D}) -> {red,Y,{black,X,A,B},{black,Z,C,D}};
% Scenarios 2, 3, and 4 are left for an exercise
balance(Node) -> Node.
```

Problem Set 2

1. Complete the add functionality for the RBT. The `add_rbt` function is written for you and you need to write the `add_rbt_` as described in the specification and definition above. Implement the `add_rbt_` function per the specification and definition above. You will also need to finish implementing the `balance` function for cases 2, 3, and 4 per the diagrams above. Use the test code provided.
2. Implement the `contains_rbt` function per the specification below. Note that while color is used in the RBT, the color is left unused by the `contains_rbt` function. Use the test code provided.

$$spec\ \ contains\_rbt :: a\ \ node \rightarrow boolean.$$

8.3 Performance

The binary search tree from Part 1 has performance of O(log n) if its balanced but as it becomes unbalanced, it becomes O(n). The RBT from Part 2 has performance of O(log n) because it maintains the balanced state. Even though there is more work needed to balance, it is offset by ability to find the spot for new nodes because its balanced.

To compare the performance of both approaches for a tree with 10,000 nodes in the problem set below, we will use the `eprof` library in Erlang. This library will report the number of times a function runs and the execution time of those functions. As with tools like these, there is an overhead cost for running `eprof` but it will not prevent us from making observations.

The following utility functions can be put around code to display performance results.

```
start_perf() ->
    eprof:start_profiling([self()]).

stop_perf(Title) ->
    io:format("Perf (~p): ~n",[Title]),
    eprof:stop_profiling(),
    eprof:analyze(total).
```

If we wanted to compare the cost of adding to the front of a list and the cost of adding to the end of list, we would create a single `foldl` to perform those operations using a sequence of numbers from 1 to 10,000.

```
% Code to test
prepend(Value, List) -> [Value|List].

append(Value,[]) -> [Value];
append(Value,[First|Rest]) -> [First|append(Value,Rest)].

timing_test() ->
    List = lists:seq(1,10000),

    start_perf(),
    lists:foldl(fun prepend/2, [], List),
    stop_perf("prepend"),

    start_perf(),
    lists:foldl(fun append/2, [], List),
    stop_perf("append"),

    ok.
```

The code will take several seconds to run. The output of `eprof` will include your functions and several others. Sample output (results will vary on different computers) of `eprof` for this code including only the functions we are testing is given below:

```
Perf ("prepend"):
FUNCTION                          CALLS        %  TIME  [uS / CALLS]
--------                          -----  -------  ----  [----------]
prove08:prepend/2                 10000    22.91  1126  [      0.11]
--------------------------------  -----  -------  ----  [----------]
Total:                            32524  100.00%  4914  [      0.15]

Perf ("append"):
FUNCTION                             CALLS        %     TIME  [uS / CALLS]
--------                             -----  -------     ----  [----------]
prove08:append/2                  50005000    99.94  9867673  [      0.20]
--------------------------------  --------  -------  -------  [----------]
Total:                            50027524  100.00%  9873713  [      0.20]
```

We can see that the `prepend` was much faster taking about 1 millisecond versus `append` taking almost 10 seconds. The cost of each call wasn't too much different. The `append` was plagued by all the recursive calls to add the item to the end of the list. We can see the $O(n^2)$ behavior of `append` when add `n` items.

Problem Set 3

1. Compare the performance of the Binary Search Tree (`add` and `contains`) and the Red Black Tree (`add_rbt` and `contains_rbt`). Use the `start_perf` and `stop_perf` functions to add up the total time using these functions (remember that `add_rbt` uses `add_rbt_` and `balance` as well). Add the numbers 1 to 10,000 (the list is already provided to you in the test code) to both data structures in order using a `foldl`. This will result in the worst unbalanced tree using `add`. Do a search for the value 10,000 to force the largest search for both `contains` and `contains_rbt`). Compare the performances for our 4 functions and explain what you found.

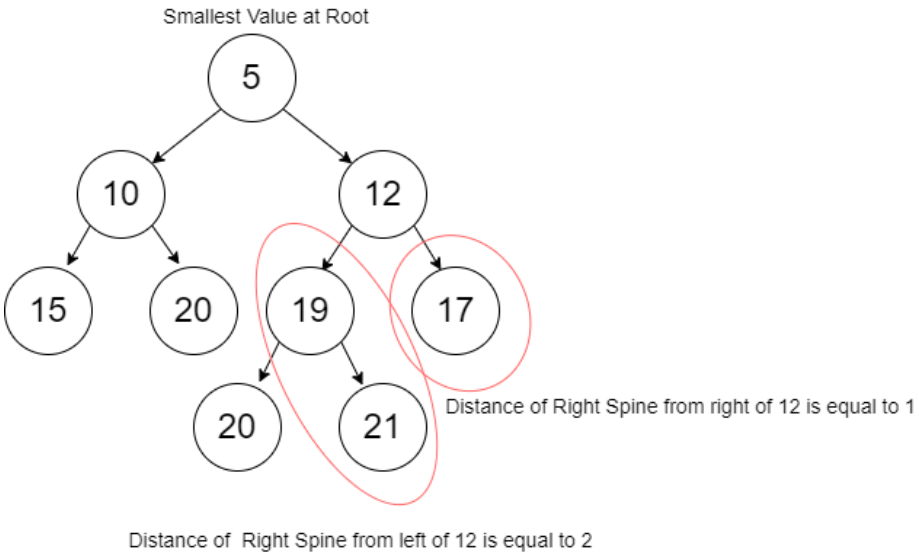
Chapter 9

Min Heaps

In this lesson we will learn about Min Heaps. The Min Heap has two components to it: a minimum value easily accessible at the root and a "heap" of everything else below it in the form of a tree. There is a strategy for that heap of values which we will explore in the material below.

9.1 Min Heap

The diagram below shows a Min Heap. More specifically its a "Leftist" Min Heap since in all cases we have favored putting items on the left thus keeping the right "spine" of the tree as short as possible (within the algorithm requirements which we will learn about shortly).



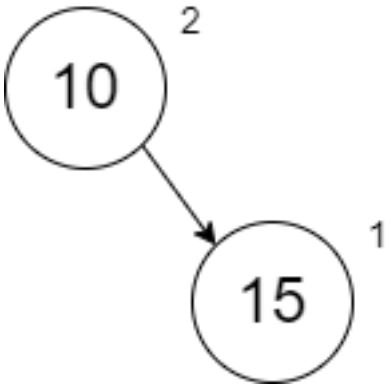
When we look at this Min Heap, we see three characteristics: 1. The smallest value is in the root node 2. The values along each right spine (both from the root and from other nodes) are sorted.
3. For each node, the length of the right spine from the left and right links are such that the one on the left is greater than or equal to the one of the right. For example, if you look at node 12, the left side (node 19) has a distance of 2 on the right spine (following 19 and 21) and the right side (node 13) has a distance of 1 on the right spine (following only 13). We call this distance, the "rank".

If we want to add items to this Min Heap, we are going to add them in order along the right spine starting with the root node. When we insert in the node, we will need to work our way back up the tree and check to make sure we have not violated our need to have the left rank be greater than or equal to the right rank. If there is a problem, the solution is to just swap the left with the right.

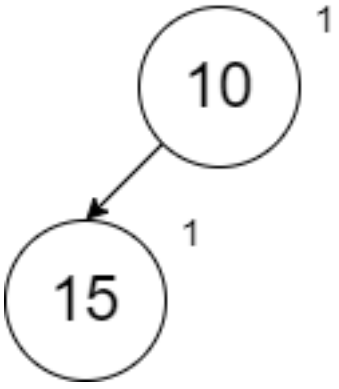
Before we look at specifications, definitions, and code, we will first build a Min Heap by adding the following numbers in order (same values you will find in the test code for the problem set): 10, 15, 20, 5, 12, 17, 19, 20, 21, 13, 8, and 1. As you look at this list of numbers, note that we will have some interesting results at various times. For example, when add the 5 and the 1 we will be replacing the root node since the root node must contain the minimum value. Also, when we add 13, we will be replacing along the right spine from the root since the right spine from any node must be in order. In all of these operations, remember that we are persisting previous versions of the min heap.

The addition of our first node 10 is trivial since the min heap is empty. In each of these diagrams, we will show the rank on the upper right of the circle.

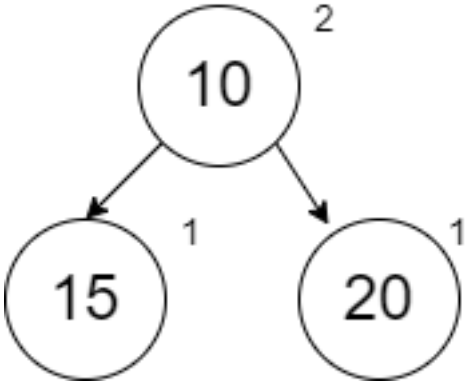
The addition of 15 will require us to compare it with 10 (the first only value on the right spine from the root). When we insert, we will always insert to the Right. When we get to the case that we are inserting in the middle of the spine, we will keep the left of our new node empty and the right of the new node will be the remainder of the spine unchanged.



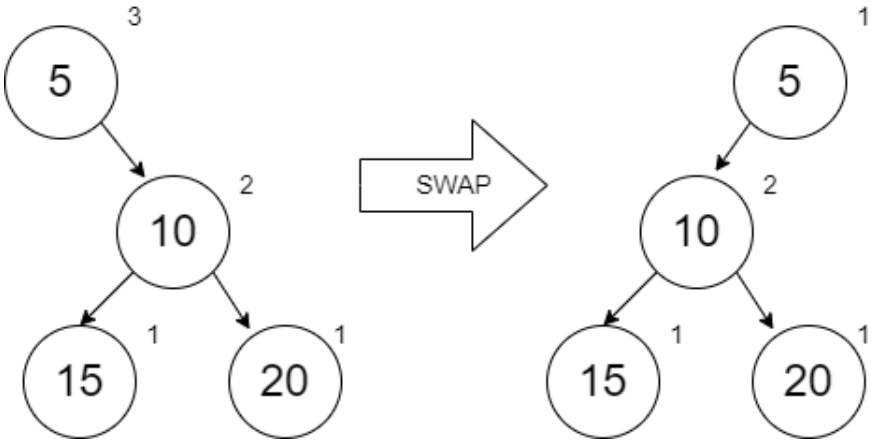
When we look at this min heap, notice that the rank on the right side is 1 and the left hand side is empty which implies a rank of 0. This means we need to swap left and right. The diagram below is correct.



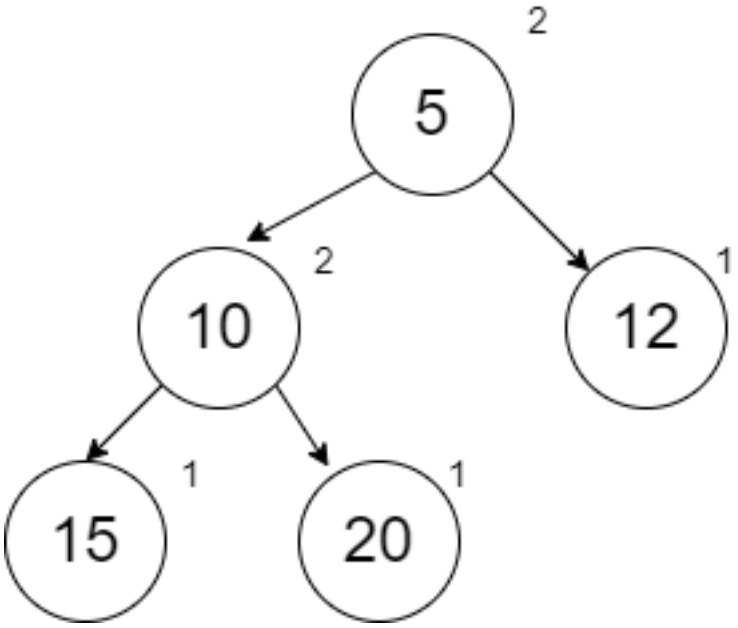
When we add 20, we have a spot and no swapping is needed.



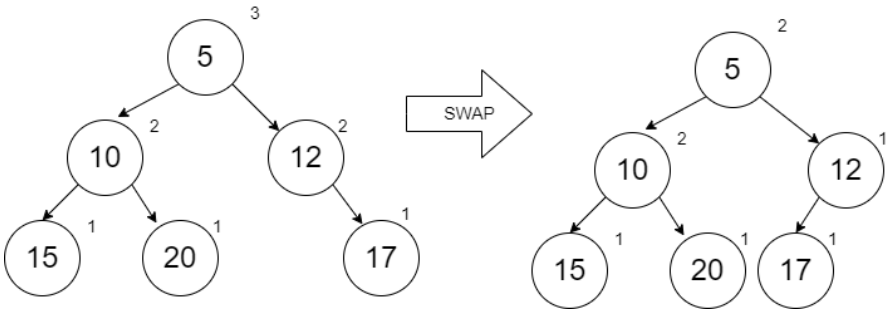
When we add 5, our first comparison shows that 5 should be before 20 on the right spine. When adding a new node to anywhere but the end of the spine, we will leave the left link empty and the right link will be the remainder of the spine unchanged. After doing this, we work our way back up (not too far in this case) to check ranks. We don't check the ranks of the spine that we left unchanged. The ranks at node 5 show that we need to swap.



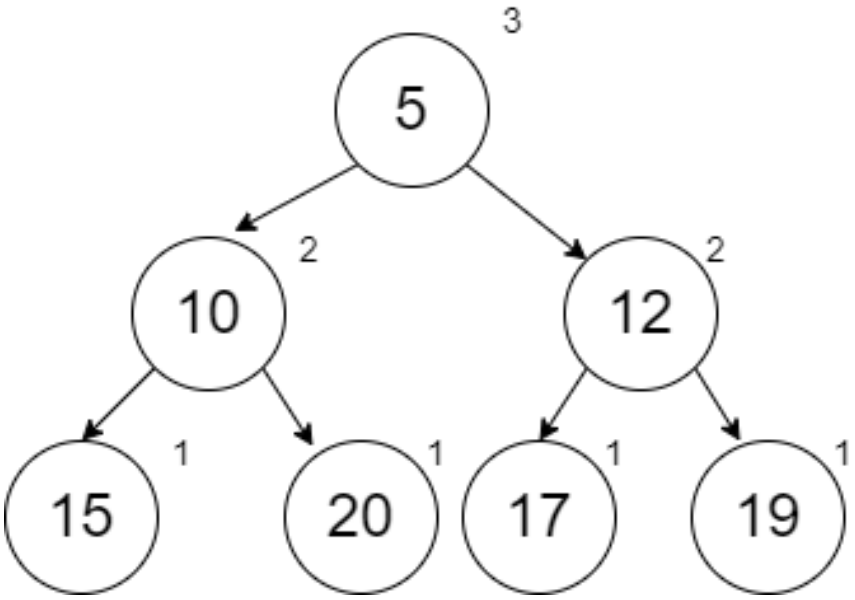
When we add 12, it can be done with no swapping.



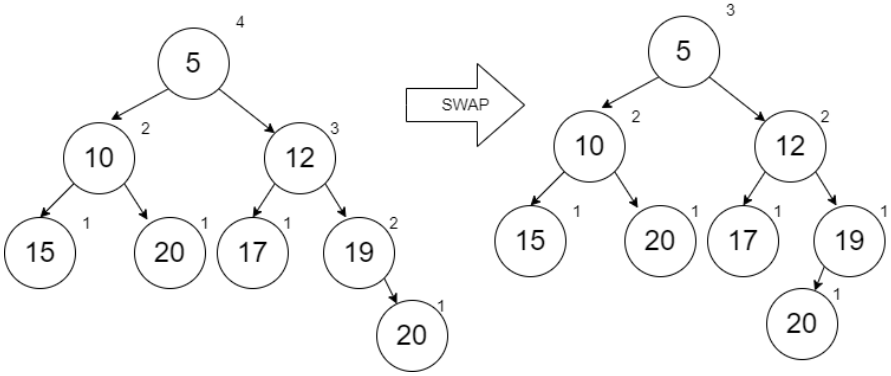
When we add 17, we start our comparisons with the node that we added (which is no issues because its at the end) and then work our way up the spine comparing ranks. At the 12, we see a need to swap.



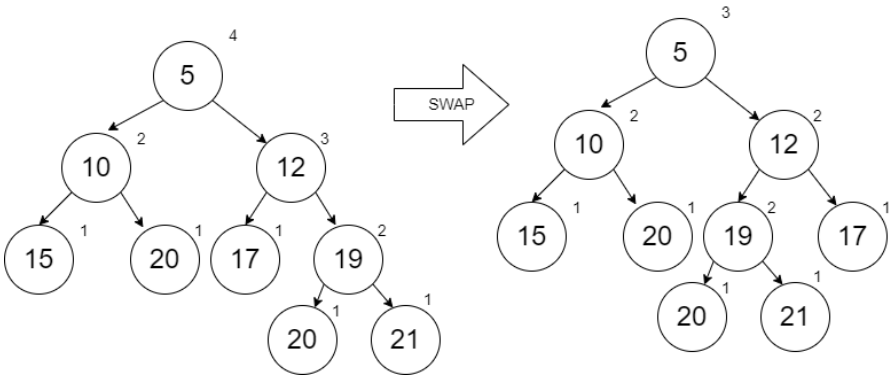
When we add the 19, it can be done with no swapping.



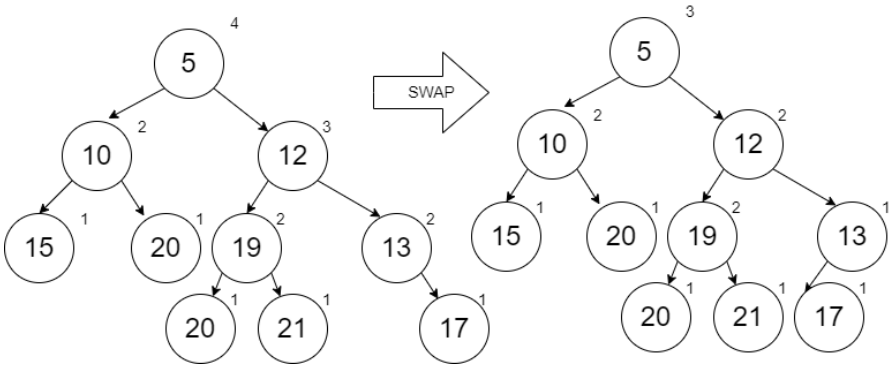
When we add the 20, it can also be done with no swapping. Notice that there was no problem adding this duplicate. If the duplicate was the root node already, we could choose to replace it with our new value and push the duplicate down (all depends on how we write our boolean comparison).



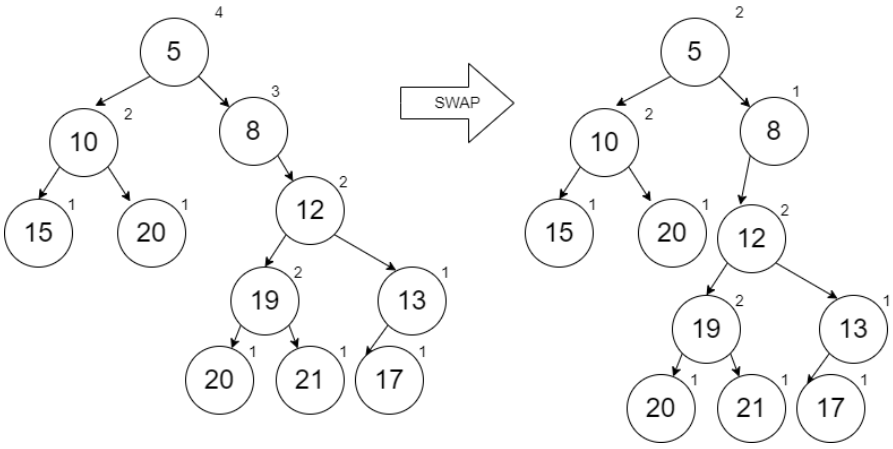
When we add the 21, we find that the node 12 has a need to swap.



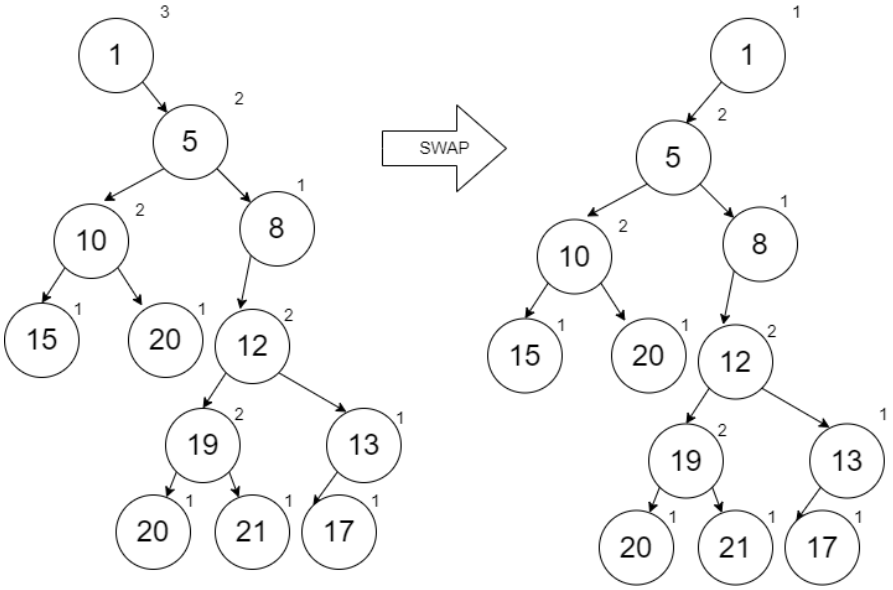
When we add the 13 (which occurs between 12 and 17), a swap occurs with node 13.



When we add the 8, we have to add it within the spine after the 5. When we do this, the rest of the spine is put to the right of our new node 8. Starting with node 8, we see the rank of node 12 is higher than the empty left side of 8. A swap is required. Notice as we do this that the right spine from the root is remaining minimized and sorted. This makes it faster to insert new items.



Finally, when we add the 1, it has to be added at the root, so the entire tree becomes the right link from the new 1 node. The ranks require us to swap and now we end of with a min heap that has nothing on the right side at all. This means inserting the next thing will be really fast and the minimum is easily accessible at the root still.



After reviewing this process, you should be able to envision the specification and design. We will be doing this in 2 steps. First, we will implement the inserting along the right spine but without swapping. The swapping will be done in the next section. We will define a structure that includes rank. We again are assuming that the data type a for the value is something that can be compared with boolean operators.

```
struct node {integer : Rank, a : Value, node : Left, node : Right}.
spec insert :: a node → node.
```

To support our definitions that we will discuss below, we have two helper functions (which will be more helpful as we explore this more). The `rank` function gives us a value of 0 when no node exists (`nil`). The `make` function will create the node with an updated rank value. Note that the distance along the right spine is always the rank of the right node plus one (representing the new node we are creating).

```
spec rank :: node → integer.
spec make :: a node node → node.
def rank :: nil → 0;
def rank :: {Rank, Value, Left, Right} → Rank.
def make :: Value Left Right → {(rank Right) +
1, Value, Left, Right}.
```

In the definition for `insert`, the first clause is for the case that we have reached the end of the spine (or the heap was empty to begin with). The second clause handles the case where we have found a place to insert along the spine. In this second case, we are keeping the entire remaining heap (`Node`) on the right of our new node. The third clause handles the case where we are still looking. In this last case, we are keeping the left link of the heap unchanged as we go.

```
def insert :: New_Value nil → (make New_Value nil nil);
def insert :: New_Value Node → (make New_Value nil Node)
    when New_Value ≤ Node.Value;
def insert :: New_Value Node → (make Node.Value Node.Left
    (insert New_Value Node.Right)).
```

Problem Set 1
You can find the template for the problem sets in this lesson here: prove09.erl

1. Implement the `insert` function per the specification and definition given above. The `rank` and `make` functions are written for you. Use the test code provided to test your function. Note that at this point, the algorithm has not done any swapping. That will be done in the next section below. When implementing this in Erlang, you may want to take advantage of some additional syntax. Up to this point, there are two ways you know to write the `insert` function with the `node` tuple:

```
insert(New_Value, Node) -> ...
% or
insert(New_Value,{Rank, Value, Left, Right}) -> ...
```

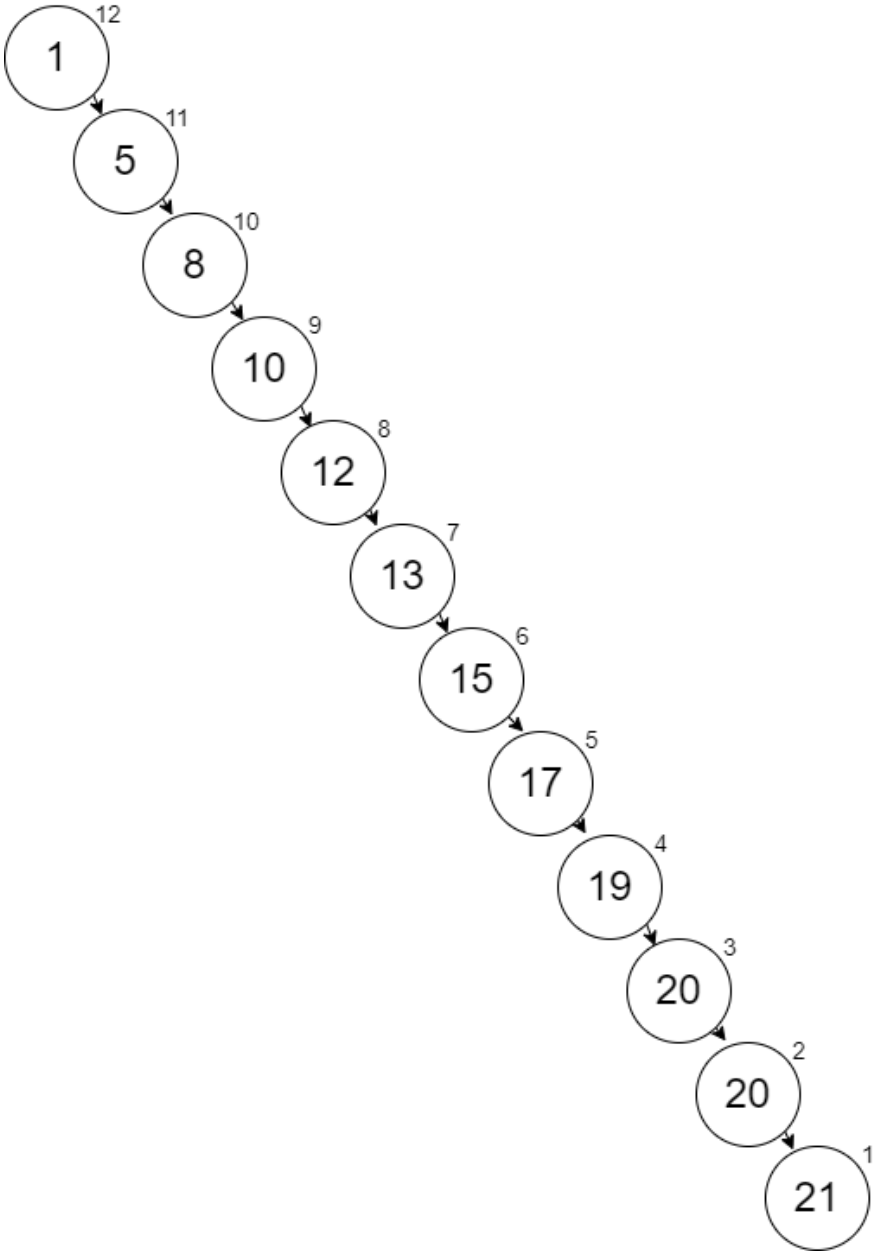
A third option combines both of these:

```
insert(New_Value, Node={Rank, Value, Left, Right}) -> ...
```

Using this third option, I can easily reference the whole `Node` or one its pieces by name. If the latter is done, you can also replace unused parameters with `_` if desired.

9.2 Efficient Inserting and Removal

In the `insert` function we have created, no swapping has occurred. However, we have satisfied the first two rules that the minimum is a the root and the values in the right spines are all in order. Here is the result of our `insert` function using the values in the previous problem set:



Notice that this looks like a list. The cost to find a place to insert a new node is O(n). If we do the swapping, we will minimize the length of the right spine and achieve O(log n).

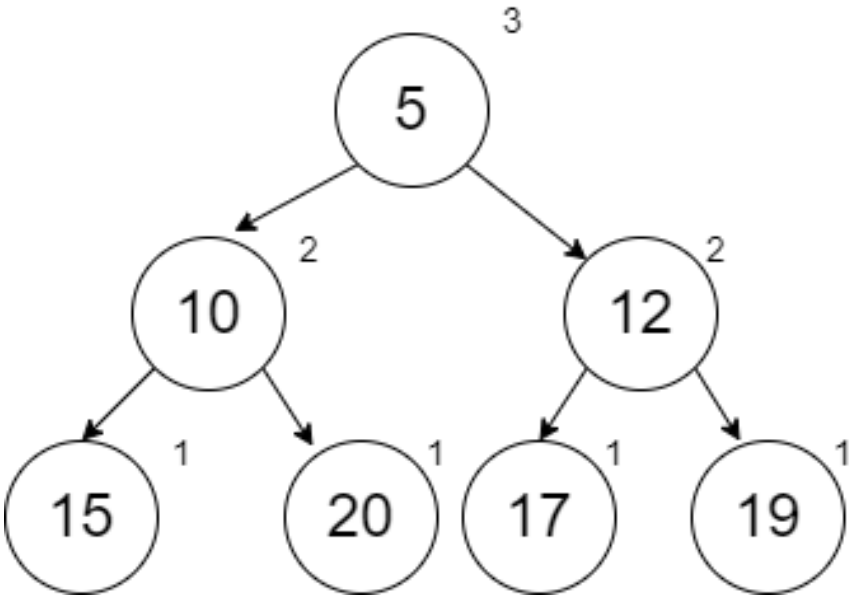
To do the swapping, we will modify the definition of our make function. The first scenario below will not swap (`Left` is put on the left side, `Right` is put on

the right side, and `Rank` is based on the `Right` rank) because Left Rank is larger or equal. The second scenario will swap because Left Rank is less. The second scenario is not shown below but is left for an exercise.

```
def make :: Value Left Right →
    Rank_Left = (rank Left),
    Rank_Right = (rank Right),
    {Rank_Right + 1, Value, Left, Right}    when    Rank_L ≥
Rank_R;
```

The implementation of this updated `make` function will be left for an exercise below.

We will add many things to a min heap but the only thing we want to remove is the minimum value at the root. Obtaining this value is not difficult. However, removing from the heap and merging the left and right sides of the heap together is more complicated. Consider the following min heap (which is a smaller version of the previous example):

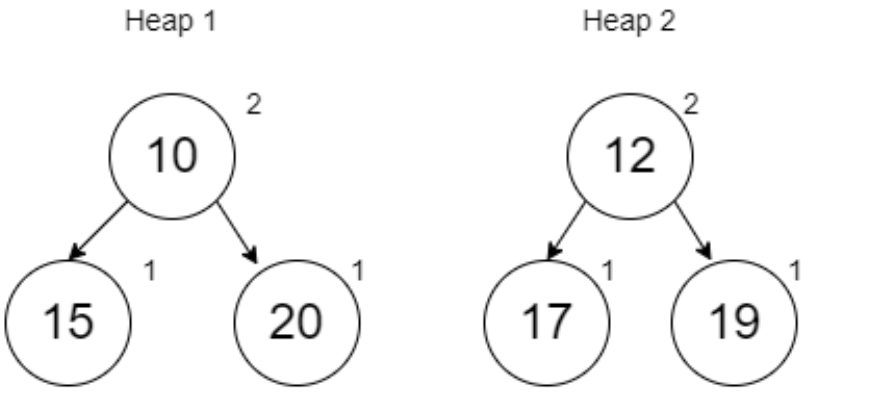


If we remove the root node 5, then we are left with two separate min heaps that need to be merged together. Here is the process:

1. Follow the right spines of both min heaps recursively looking for the next largest value until you run out of "spine" on one of the heaps. The heap that you still have left becomes your new starting heap.

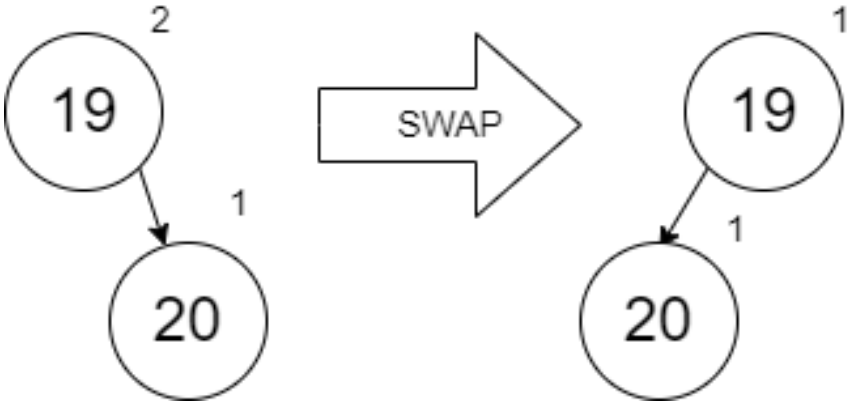
2. As you recursively return back up to the smaller values, insert the next smaller value one at a time (following the processes we used before including swapping as needed) into our new merged heap. When you insert the smaller values in, make sure that you include unchanged what was on their left.

Following our process recursively down on the min heaps above, here is the order in which we went through to the largest value until one min heap was done. Notice that after the 19 was visited, the min heap on the right was done and so are starting min heap (from step 1 above) is just the node 20.

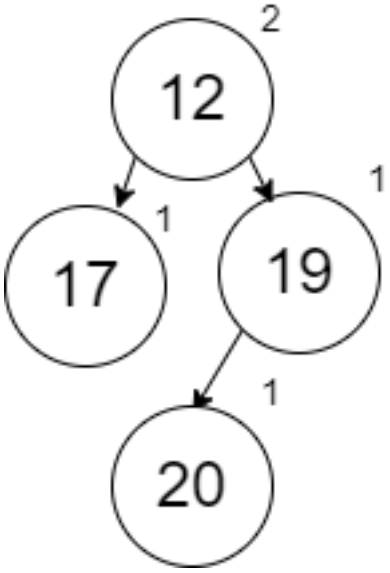


Visited 10, then 12, then 19 ... End of Heap 2 ... So start Merged Heap with whats left of Heap 1.

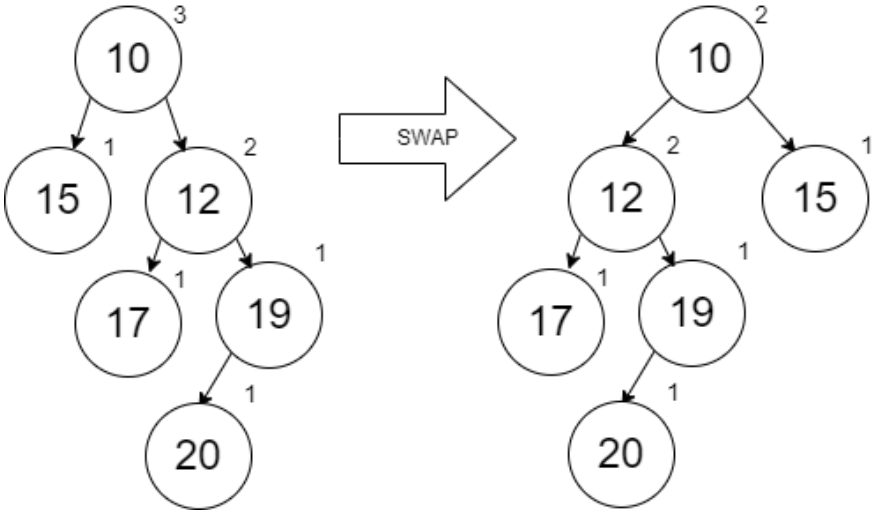
Working backwards, we merge in the 19 (which per our original process of inserting, it goes before the 20 and the 20 unchanged goes to the right) and get the following (after the required swapping to keep the min heap rules satisfied):



The next to merge in is the 12 (with the left node 17 unchanged):

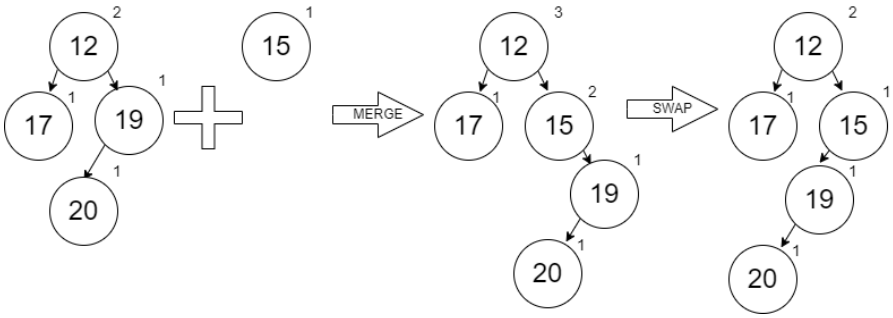


The last to merge in is the 10 (with the left node 15 unchanged):

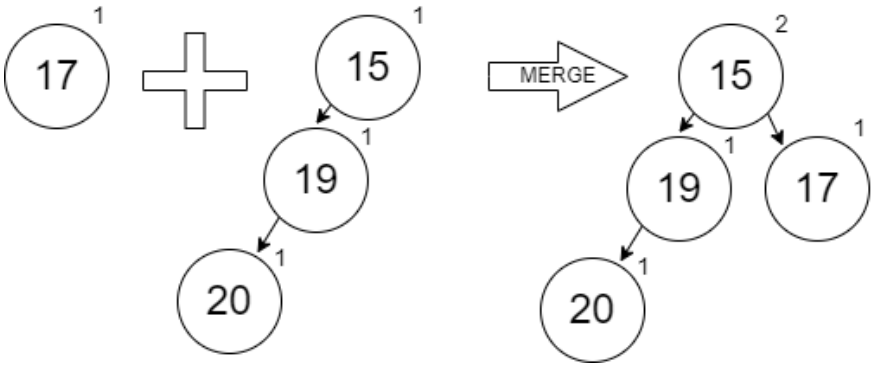


As expected, the new root is 10 which is new minimum value.

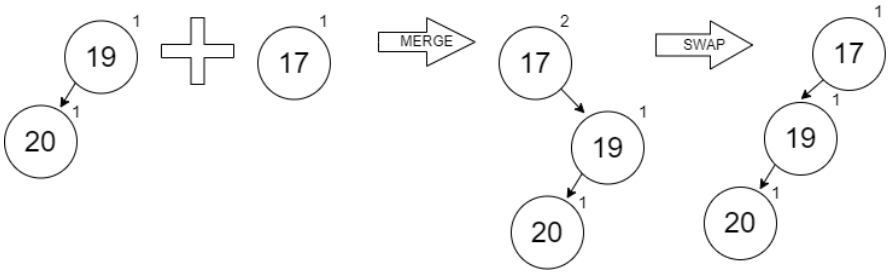
If we remove the 10, our second min heap to merge has only one node which is just like inserting a single value into the first min heap (which follows the same process of our two steps above).



Removing the 12:



Removing the 15:



With nothing left on the right hand side, the merging will become trivial as we remove more minimum values.

The specification for the merge is given below:

```
spec merge :: node node → node.
```

There are four scenarios for the definition. The first two handle the case of merging something with nothing. This is how we start with a heap after going through the complete right spine of one of the min heaps. The third scenario is if we find the next largest one on the right min heap. The fourth scenario is if we find the next largest one of the left min heap. In both of these last two scenarios, we keep going until we finish one of the spines and then we build the nodes up one at a time backwards using the `merge` function (which also takes care of our swapping).

```
def merge :: Heap1 nil → Heap1;
def merge :: nil Heap2 → Heap2;
def merge :: Heap1 Heap2 → (make Heap1.Value, Heap1.Left,
    (merge Heap1.Right, Heap2)) when Heap1.Value ≤
Heap2.Value;
def merge :: Heap1 Heap2 → (make Heap2.Value, Heap2.Left,
    (merge Heap1, Heap2.Right));
```

The `merge` function can now be used in a `remove_min` function. The `remove_min` function has the following specification:

```
spec remove_min :: node → node.
```

The implementation of `merge` and `remove_min` are left for exercises.

Since the `merge` function successfully handled merging a min heap with another min heap of size 1, we could reuse the `merge` function to simplify our `insert` function. We are merging our new node (created with `make`) with our existing min heap (`Node` - which could be nil).

```
insert(New_Value, Node) -> merge(make(New_Value, nil, nil), Node).
```

Problem Set 2

1. Modify the existing `make` function to properly perform the swaps per the definition above (including the second scenario in the definition which was not provided to you). Use the test code provided to test your function.

2. Implement the `merge` function and the `remove_min` function as described above. Use the test code provided to test your function. Once your code is working, you can modify your `insert` function to use the code provided above that uses `merge`.

9.3 Priority Queue

A priority queue is a normal First In First Out (FIFO) queue except that a priority is added to each node placed in the queue. For our discussion, we will say that priority 1 is the highest priority and therefore should be processed first. The min heap is useful for a priority queue because the next item to dequeue is always quickly available at the root.

We want to modify our min heap so that it will work with any data a even if Erlang doesn't know how to compare with a boolean operator. The comparison (so long as we simplified our `insert` function already in the previous section) is only performed in the `merge` function. If we provide a function to our `merge` that handles the comparison, then we are unlimited to what we can put in our heap.

For a phone queue, we will define the following structure:

$$struct\ \ call\ \ \{integer : Priority, string : Name\}.$$

In our `merge` function, we have been doing a $\leq$ comparisons with the priority. Now we want to define a $\leq$ comparison for `call` objects.

$$spec \;\; \lambda_{compare} :: call \;\; call \rightarrow boolean.$$

If we are going to provide this $\lambda_{compare}$ to our `merge` function, we are going to need to create a `merge/3`. Since `merge` is called by `insert` and `remove_min`, we will need to create an `insert/3` and `remove_min/2` as well to receive the $\lambda_{compare}$.

$$spec \;\; insert :: a \;\; node \;\; \lambda_{compare} \rightarrow node.$$
$$spec \;\; remove\_min :: node \;\; \lambda_{compare} \rightarrow node.$$

The implementation of `merge/3`, `insert/3`, and `remove_min/2` along with the creation of a $\lambda_{compare}$ to compare the priorities within two `call` tuples is left as an exercise below.

When we implement and test these functions, you should notice an unusual behavior when you have duplicate priorities. The order in which items are added to the min heap are not maintained when considering those items that have the same priorities. The Min Heap is not sort stable. The Min Heap will not guarantee the original order for duplicate priorities. Perhaps it could be useful to add a timestamp to our `call` record to keep track of who arrived first and then include that in our $\lambda_{compare}$ function.

Problem Set 3

1. Implement the `merge/3`, `insert/3`, and `remove_min/2` functions as described above. Modify the provided test code to set the variable `Calls_Compare` equal to a your $\lambda_{compare}$ which should compare two `call` tuples in the same way as the original `merge/2` compared two nodes with $\leq$. For example, if `Calls_Compare` was called on `{"MST", 2, "Tim"}` and `{"PST", 1, "George"}`, then the function would compare $2 \leq 1$ which is `false`. Use the test code to test your code. Note that `get_min` was implemented for you to support the test code.

Chapter 10

Random Access Lists

The lists that we have used in this course have been linked lists. If we want to find something in the list, we have to search resulting in O(n) performance. We could use a binary search tree or a min heap to help us find things better but then we lose the order in which items were added. The name of dynamic arrays for which you can maintain order and have good lookup by index is done differently in functional languages when compared to the python list, the java ArrayList, or the C++ vector. A common solution is the Random Access List (RAL).

10.1 Binary Numbers and the RAL

In Erlang, there is not a way to quickly access a specific index in a list with O(1). In Part 3, you will explore the performance of the `nth` function that Erlang provides to do this. The difficulty is that we have to traverse through the entire list from beginning to the target index. To improve on this, we would like to find an O(log n) solution that maintains the order of the values. O(log n) makes us think about binary search trees. However, we need to store information in the trees that keeps the order the same. Binary Search Trees sacrifice order for performance.

To find a solution, we will consider a problem that at first does not seem related. Consider binary numbers (purposefully with leading zeros) going from 1 to 7: 001, 010, 011, 100, 101, 110, and 111. When we want to add 1 to our number, we will go from right to left and do one of the following:

- If we encounter a 0, we will replace it with a 1 and make no more changes.
- If we encounter a 1, we will replace it with a 0 and consider these two steps again for the next number in the sequence. In this case we are mathematically carrying the 1 to the next item in the sequence.

If we try to do this in Erlang, we will be frustrated by the fact that we are going from right to left. Since it is more natural to go left (index 0) to right, we will flip our binary numbers around. When we read right to left, we call this little endian notation whereas when we read left to right, we call this big endian notation. To work better in Erlang, we will use the big endian notation

Base 10 Value	Little Endian Notation (Right to Left)	Big Endian Notation (Left to Right)
1	001	100
2	010	010
3	011	110
4	100	001
5	101	101
6	110	011
7	111	111

Here is the specification and definition for incrementing a binary sequence (going left to right in big endian format) where the sequence starts at 1 and the sequence is stored in a list. Note that the definition follows the two rules for incrementing a binary sequence that we identified earlier. An empty list would represent the value of 0.

$$
\begin{array}{l}
spec\ \ inc :: [integer] \rightarrow [integer]. \\
def\ \ inc :: [0|Rest] \rightarrow [1|Rest]; \\
def\ \ inc :: [1|Rest] \rightarrow [0|(inc\ \ Rest)];
\end{array}
$$

The erlang code is shown below:

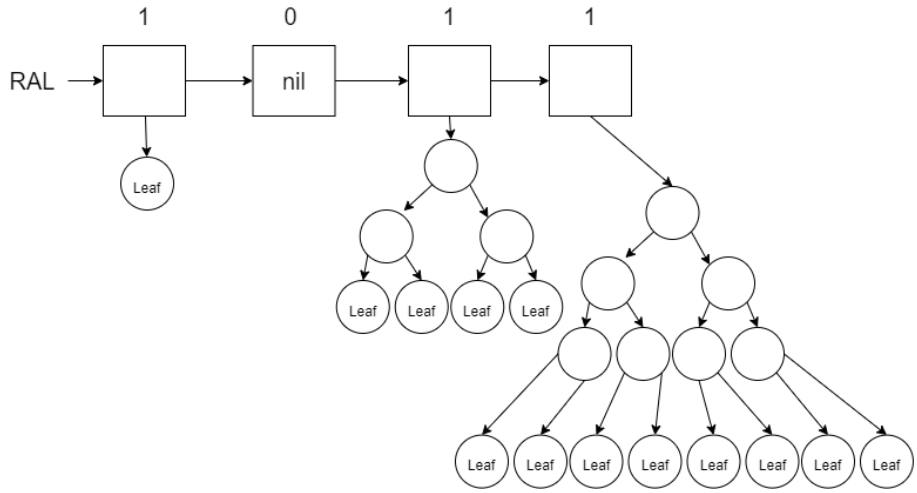
```
inc([]) -> [1];
inc([0|Rest]) -> [1|Rest];
inc([1|Rest]) -> [0|inc(Rest)].

test() ->
    lists:foldl(
        fun(_Value,Acc) ->
            X=inc(Acc),
            io:format("~p~n",[X]),
            X
        end, [], lists:seq(1,15)),
    pass.
```

The result of running the test code up to 15 (1111) is shown below:

```
[1]
[0,1]
[1,1]
[0,0,1]
[1,0,1]
[0,1,1]
[1,1,1]
[0,0,0,1]
[1,0,0,1]
[0,1,0,1]
[1,1,0,1]
[0,0,1,1]
[1,0,1,1]
[0,1,1,1]
[1,1,1,1]
```

Now that we can build these binary sequences, lets observe what each "1" represents. Notice that when you write binary numbers, each "1" represents 2^n possible values. In other words, we could say that each "1" is storing those values for us. In the following binary sequence (written left to right as we did before): 1011, the 1's represent: 2^0, 2^2, and 2^3 for a total of 13. In other words we could say that 1011 represents 13 different things.

To see those 13 different things, we will represent each 1 by a tree. These trees will be special in that each tree will have 2^n leaves where n represents the index in the binary sequence (starting at 0). Trees with 2^n leaves are not hard to build. A complete and balanced tree of height 2 will have 2^2 leaves, a tree with height 3 will have 2^3 leaves, and so on. Here is the representation of 1011 (or 13) using these trees:



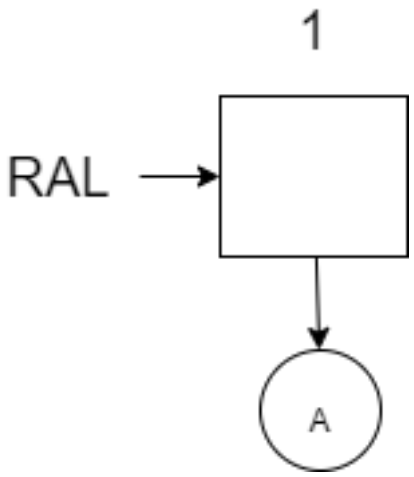
Notice that the 0's are represented with *nil* instead of a tree. We call this list of trees a Random Access List (RAL). When you look at this list of trees, you will notice there are a grand total of 13 leaves. Each of these leaves represent the actual 13 pieces of data in the RAL. All the other nodes in the tree including the *nil* values in the list represent the organization of the RAL to enable the O(log n) behavior. If you wanted to go to one of these leaves, you could traverse the sequence from left to right (which would be O(log n)) and then traverse down to one of the leaves (which would also be O(log n)). We will explore how this works in the next section.

To add new items to a RAL, we will use a similar algorithm for adding one to our binary sequence:

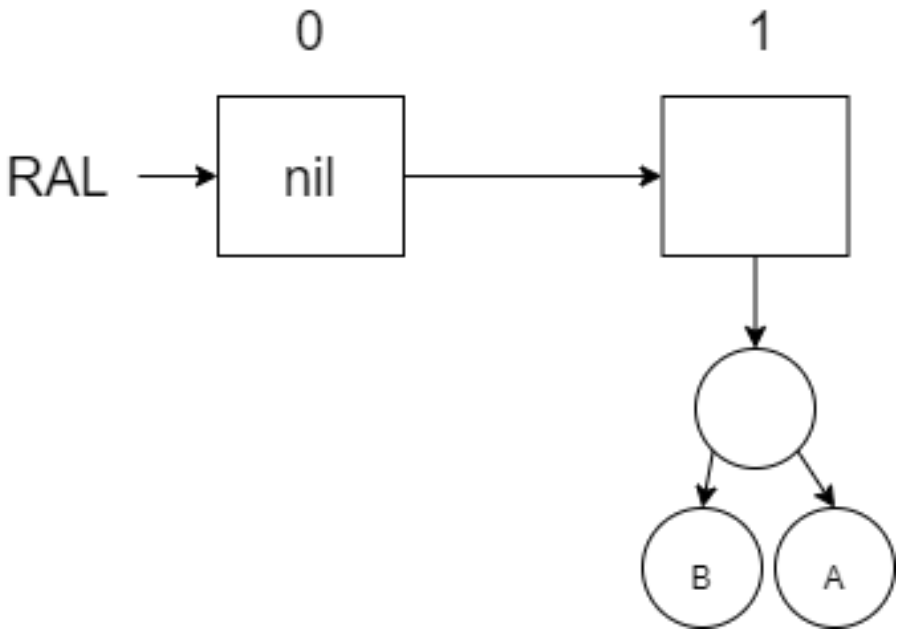
- If we encounter a *nil* (think a 0 in the binary sequence), we will replace it with a tree (think a 1) containing 2^n (where is n is our current index) and make no more changes. The tree that we add will either be a tree of height 1 (if the RAL was previously empty) or it will be a tree that was carried over from the recursive call in the next step below.
- If we encounter a tree (not *nil*, think a 1 in the binary sequence), we will replace it with a *nil* (think 0) and carry the tree to the next sequence and recursively consider these two steps again. If a tree was carried over, then we have to carry both the tree(s) from the previous sequence position(s) along with this position.

Let's visually look at how this works first as we store letters in a RAL starting with an empty RAL. Notice that when we add a new letter to our RAL, it will prepend it to the front just like adding a new digit to a binary sequence. In each diagram, a non-leaf node is left empty and the leaf nodes contain the actual data. Remember that each tree must be balanced and complete with 2^n leaves each.

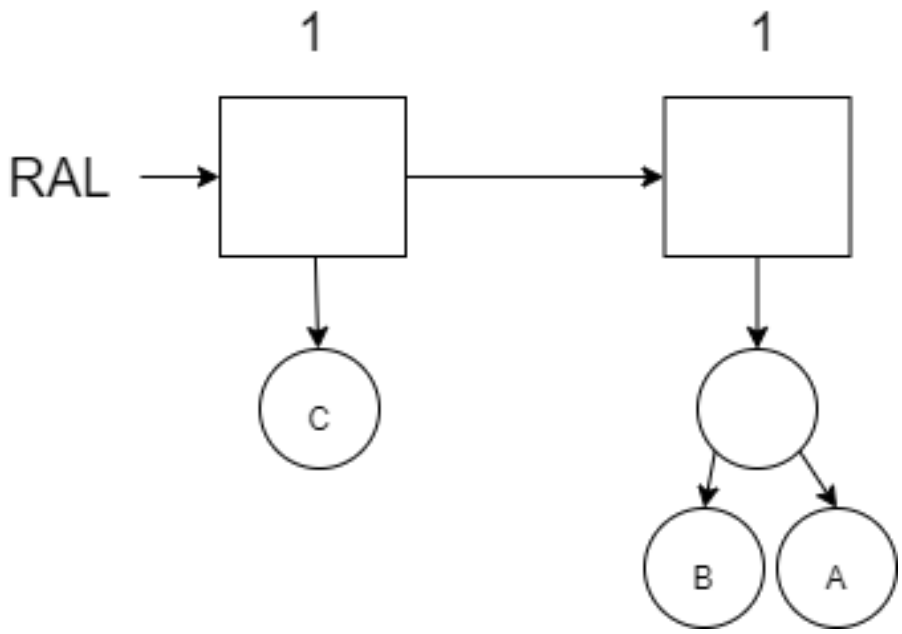
Add A to an empty RAL:



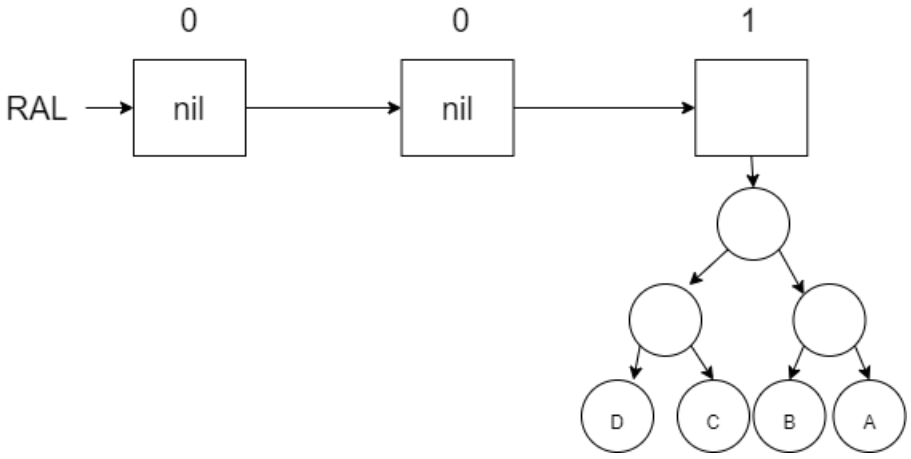
Add B to the RAL (will require a replacement of the first tree with a *nil* and merge the new tree with B and the former tree with A):



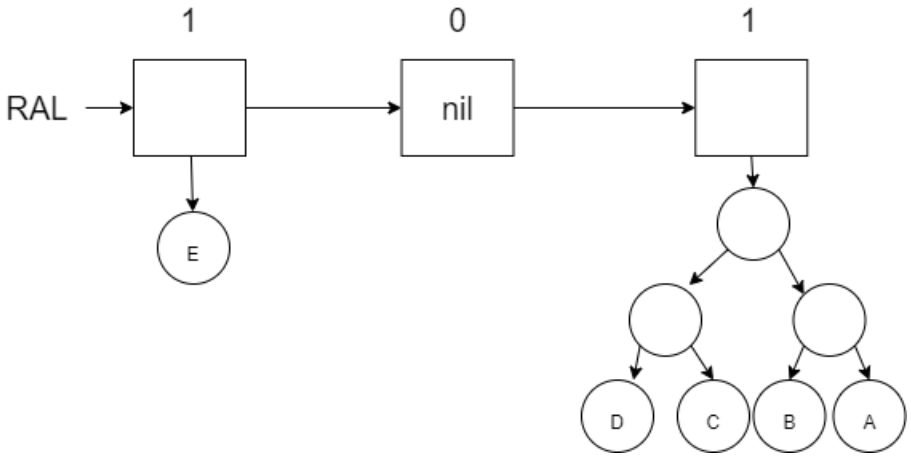
Add C to the RAL (there is a space to put the new tree with C):



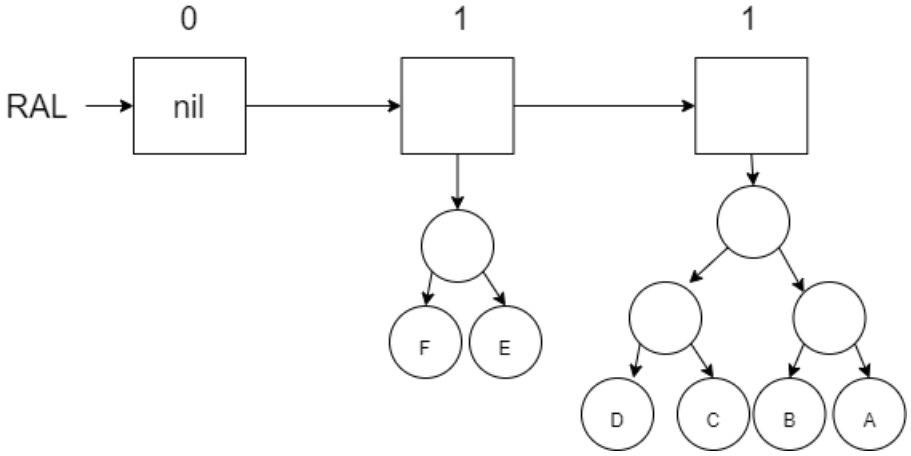
Add D to the RAL (have to merge the C and D and then merge it with the B/A tree and then put it in the next available spot to the right):



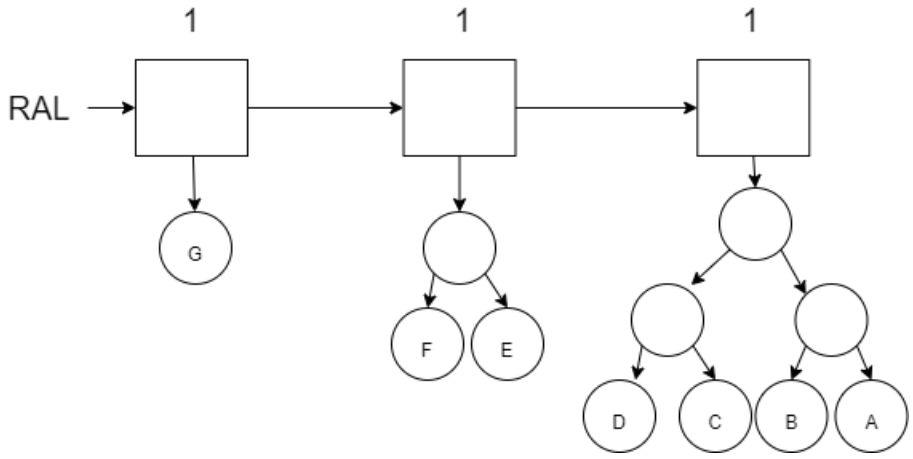
Add E (space exists for that in index 0):



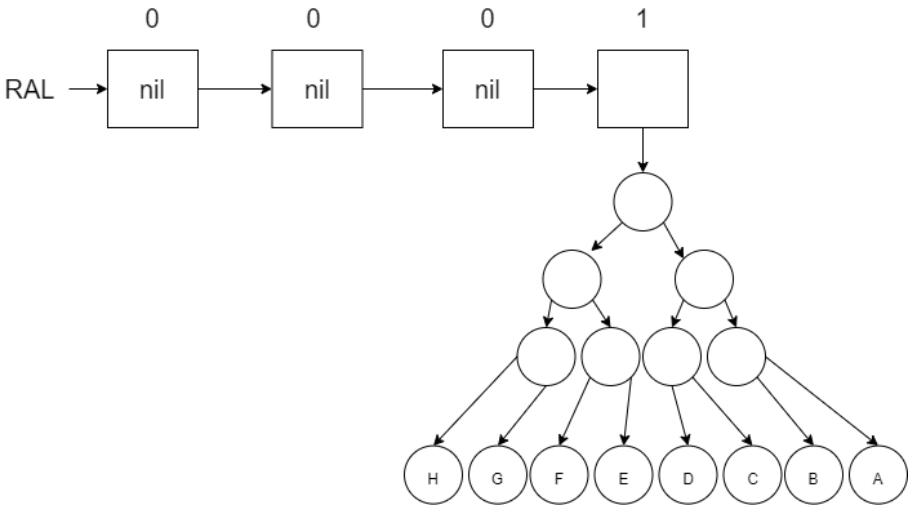
Add F (merge the new tree with F and the the tree with E … notice that the binary number shown above the boxes is increasing like our first binary example):



Add G (which has an available space):



Finally add H (which requires us to merge our new tree with H along with the other 3 trees to create the new tree in index 3).



To create the RAL in software, we will need a structure to represent each of the trees in the list. The root node of each of the trees will be represented by a *ral_tree* node. Our structure distinguishes between non-leaf nodes (no values) and leaf nodes (values). The non-leaf nodes called *node* will include a reference to the left and right sub-trees as well as a *Count*. The *Count* will represent the number of leaves under this node. The *Count* of a leaf node will always be 1. We store the *Count* in each non-leaf node to make it easier to merge a *ral_tree* with another *ral_tree*. We will also use this *Count* when trying to access values in our RAL by index later on.

```
struct ral_tree
    {atom(node), integer : Count, ral_tree : Left, ral_tree :
Right} or
    {atom(leaf), a : Value}.
```

The RAL will be represented by a list of *ral_tree* objects. Our `prepend` function will use the following specification and definition:

```
spec prepend :: a [ral_tree] → [ral_tree].
def prepend :: Value RAL → (merge {leaf, Value} RAL).
```

The `merge` function is responsible for implementing the two rules just like `inc` was used to implement the two similar rules with the binary sequence. The $\{leaf, Value\}$ will be merged (or carried) into the existing RAL (or binary sequence). Here is the specification and definition for the `merge` function. Compare and contrast this with the `inc` function we wrote earlier. When we have to merge (or carry over) a tree with an existing tree, we create a new root node and put the carry over tree to the left and the existing tree to the right. The *Count* for the new root node of the tree will be the sum of the counts of the two trees that were merged together. Refer back to the diagrams earlier to see how this merge was happening as we added a new letter to our RAL.

```
spec merge :: ral_tree [ral_tree] → [ral_tree].
def merge :: Carry_Tree [] → [Carry_Tree];
def merge :: Carry_Tree [nil|Rest_Trees] →
[Carry_Tree|Rest_Trees];
def merge :: Carry_Tree [Tree|Rest_Trees] →
    Merged_Tree = {left_as_an_exercise},
    [nil|(merge Merged_Tree Rest_Trees].
```

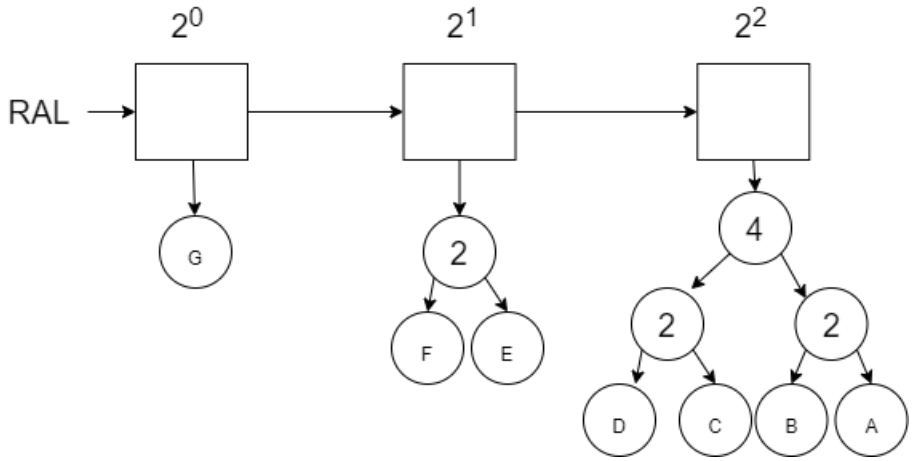
The `count` function used above will either return 1 for a leaf or the *Count* value from the `ral_tree` structure. The implementations of all of these functions in Erlang is left for an exercise.

Problem Set 1
You can find the template for the problem sets in this lesson here: prove10.erl

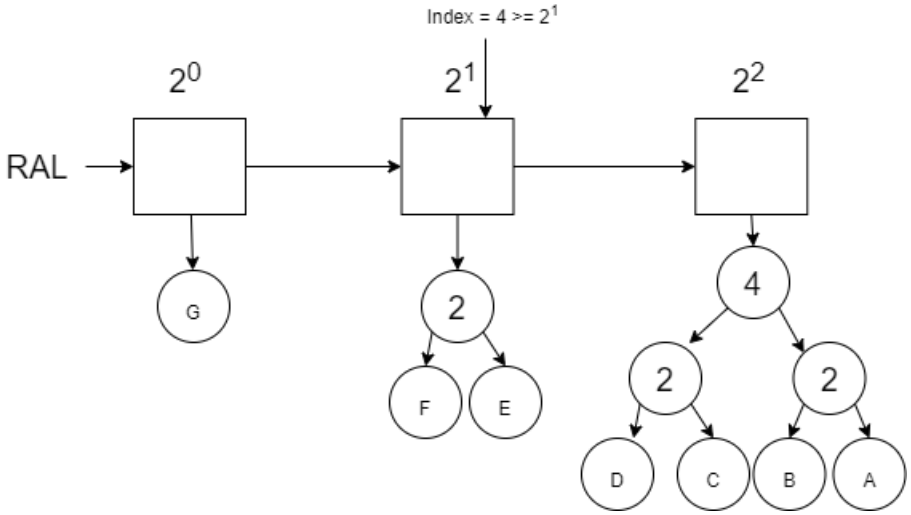
1. Implement the `prepend`, `merge`, and `count` functions as described above. The specification and definition for `prepend` and `merge` are provided. You will need complete the definition for `merge` by determining what `Merged_Tree` should be equal to. The specification and definition for `count` is not provided but it is described at the end of the reading above. Use the test code provided to verify your implementation.

10.2 Lookup and Update

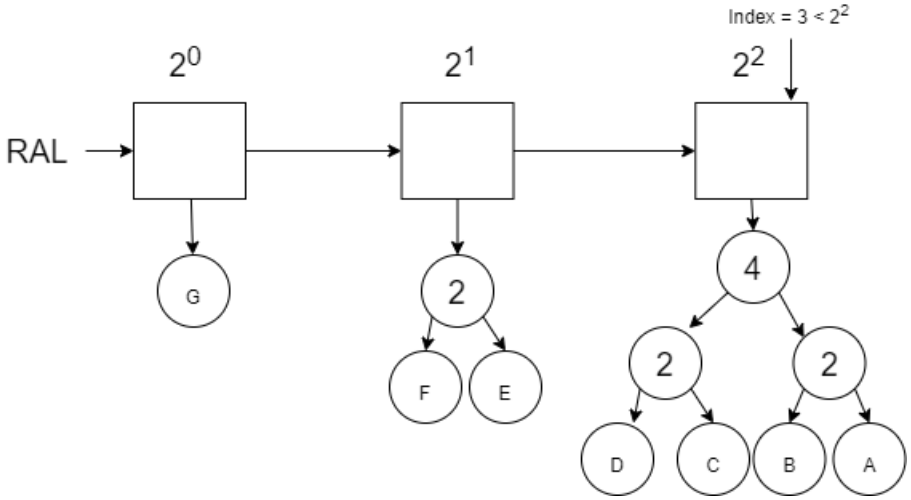
Finding a specific value in the RAL relies on the relation to the binary sequence that we started with. The diagram below shows the RAL with 7 items (111 binary sequence). In this diagram, the numbers in the non-leaf nodes show the value of *Count* (remember a leaf node has a *Count* of 1). The boxes in the list are also annotated with the power of 2 that they represent. Note that the power of 2 represents the number of leaves in the *ral_tree*.



If we wanted to find the 5th number (index 4) in the RAL (where the G in the first leaf on the left is considered index 0), then we first need to traverse the list to find the correct *ral_tree*. If we start at the first *ral_tree*, we see that the *Count* is 1 which means there is only 1 value in this first *ral_tree*. Index 4 is not in this *ral_tree*. We will subtract 1 from our Index and keep moving.



We then consider whether the value we are looking for is within the 2nd node of the RAL. Our updated index of 3 is $\geq$ to 2^1 so we subtract 2^1 from our index and move on.



We then consider whether the value we are looking for is within the 3rd node of the RAL. Our updated index of 1 is not $\geq$ to 2^2. Therefore we know that the value we are looking for is somewhere within the this 3rd *ral_list*.

The specification and definition for the lookup function is give below. Note that the `lookup_in_tree` function is used to find the desired value (i.e. leaf) in the `ral_tree`.

```
spec lookup :: integer [ral_tree] -> a.
def lookup :: Index RAL -> nilwhen Index < 0;
def lookup :: Index [] -> nil;
def lookup :: Index [nil|Rest_Trees] ->
(lookup Index Rest_Trees);
def lookup :: Index [Tree|Rest_Trees] -> (lookup Index -
(count Tree) Rest_Trees)
    when Index >= (count Tree);
def lookup :: Index [Tree|Rest_Trees] ->
(lookup_in_tree Index Tree).
```

In the definition, the first two clauses handle invalid index scenarios. The third clause handles the case where as we are going through the RAL list, we come across a *ral_tree* that is *nil*. We know there are no values in there, so we keep going. Notice that we don't decrease our *Index* because there were not any values (or leaves) in that *ral_tree*. In the fourth clause, there are values in the ral_tree but we haven't arrived at the *ral_tree* for our *Index* value yet. In the fifth case, we have found the target *ral_tree* and we will use `lookup_in_tree` to find the specific leaf.

We now have to traverse the *ral_tree* to find the leaf associated with our Index value. In our previous example, the *Index* was equal to 3 when we found our target *ral_tree*. We will use that *Index* to find our leaf. The process is as follows:

1. If the current node of the *ral_tree* is a $leaf$, then we have found our value (this is true because we assume that the *ral_tree* was created correctly in the first place).
2. If the current node of the *ral_tree* is a *node*, then we have to either go left or right:
    - If the current *Index* is $<$ the $Count/2$ (number of leaves on either side of the current node), then keep looking towards the left.
    - If the current *index* is $\geq$ the $Count/2$, then keep looking towards the right. In this case, we need to subtract off $Count/2$ from the *Index* because the *Count* values of children nodes will be relative to the parent node.

The definition and implementation of the `lookup_in_tree` function based on this process will be left for an exercise.

If we wanted to update a value in the RAL, the process is almost the same as looking up a value. The `update` function will be just like the `lookup` function and the `update_in_tree` function will be just like the `lookup_in_tree` function. The table below shows the differences:

lookup	update	lookup_in_tree	update_in_tree
Search for the *ral_tree* based on the *Index*. When found, return the value returned from `lookup_in_tree`.	Search for the *ral_tree* based on the *Index*. Rebuild the RAL list from left to right (like an `insert_at` function) until the *ral_tree* is found. When found, use the updated *ral_tree* returned from `update_in_tree`. The remaining RAL list is reused.	Use the *Index* to traverse the tree left and right until the value is found. Return the value in the *leaf* that was found.	Use the *Index* to traverse the tree left and right. Rebuild the tree (in the same way as was done with the Binary Search Tree) starting with the *leaf* using the new modified value.

The definition for `update` is provided below. The definition and implementation of the `update_in_tree` is left as an exercise.

```
spec update :: integer a [ral_tree] -> [ral_tree].
def update :: Index Value RAL -> RAL when Index < 0;
def update :: Index Value [] -> [];
def update :: Index Value [nil|Rest_Trees] ->
(update Index Value Rest_Trees);
def update :: Index Value [Tree|Rest_Trees] ->
    [Tree|(update Index - (count Tree) Value Rest_Trees)]
    when Index >= (count Tree);
def update :: Index Value [Tree|Rest_Trees] ->
    [(update_in_tree Index Value Tree)|Rest_Trees].
```

Notice that an additional difference is that the `update` function will return the original RAL if the index is incorrect.

Problem Set 2

1. Implement `lookup` and `lookup_in_tree` based on the descriptions above. Use the test code provided to verify your implementation.
2. Implement `update` and `update_in_tree` based on the descriptions above. Use the test code provided to verify your implementation.

10.3 Performance

The cost of traversing the RAL list will be O(log n) and cost of traversing a *ral_tree* will also be O(log n) which means that our `lookup` and `update` functions are O(log n). If we used a standard list, we would need to recursively search from left to right to find the desired index which would result in O(n). To demonstrate this, you can use the `nth` function provided by Erlang which will provide the value at a specified index in a list. Note that the `nth` function treats the first elements as Index 1 instead of 0 like our RAL.

Using the same `start_perf` and `stop_perf` used in previous lessons, we can measure the time it takes to access the last item in a very large 1,000,000 item Erlang list.

```
start_perf() ->
    eprof:start(),
    eprof:start_profiling([self()]).

stop_perf(Title) ->
    io:format("Perf (~p): ~n",[Title]),
    eprof:stop_profiling(),
    eprof:analyze(total),
    eprof:stop().

test() ->
    List = lists:seq(0,999999),
    start_perf(),
    999999 = lists:nth(1000000,List),
    stop_perf("list lookup last one"),
    ok.
```

The comparison of accessing the last item in a RAL with the last item in an Erlang list is left for an exercise.

Problem Set 3

1. Create a RAL with 1,000,000 items by using a `foldl` and the `prepend` function. Using `start_perf` and `stop_perf` (provided in the test code), compare the time for looking up index 999,999 (the last item) in both the Erlang List and the RAL. Discuss your observations in comments within your code. Note both the CALLS and the TIME columns when comparing the performance.

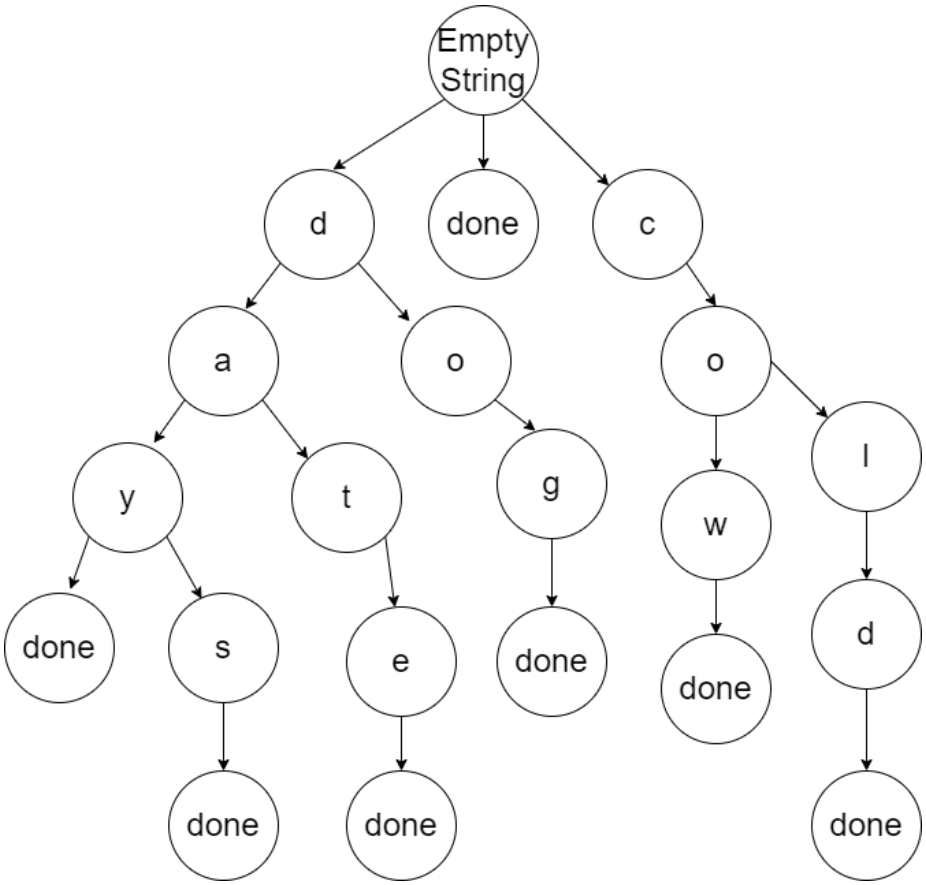
Chapter 11

Tries

When we look at all of the forms of trees (including heaps) we notice that the placement of values is highly dependent on the result of comparing two values. While comparing two integers can be an easy thing to do, the comparison of two string is more complicated. Comparing strings is a frequent activity performed in our software but consider the fact that string comparisons at worst will require the comparison of multiple characters each time. If our strings were stored in a binary search tree, we would need to compare our target string with several other strings until we found a match. The Trie data structure provides a tree structure which is intended to store strings in such a way that searching is faster and potentially memory storage is optimized.

11.1 Creating the Trie

The diagram below shows an example of a Trie that stores words. Notice that the Trie begins with an empty character string and then branches out to all valid first letters for the values that are stored. Each branch represents a single letter. To find the words that are stored, you start at the root and progress downwards until you get to a letter that has `done` as a child. In many cases, additional words can be found by following other letters instead of stopping at the `done`. This Trie includes the words: “day”, “date”, “days”, ““,”cow”, “cold”, “dog”. Notice that since “ ” is in the Trie, that there is a `done` child connected to the root.

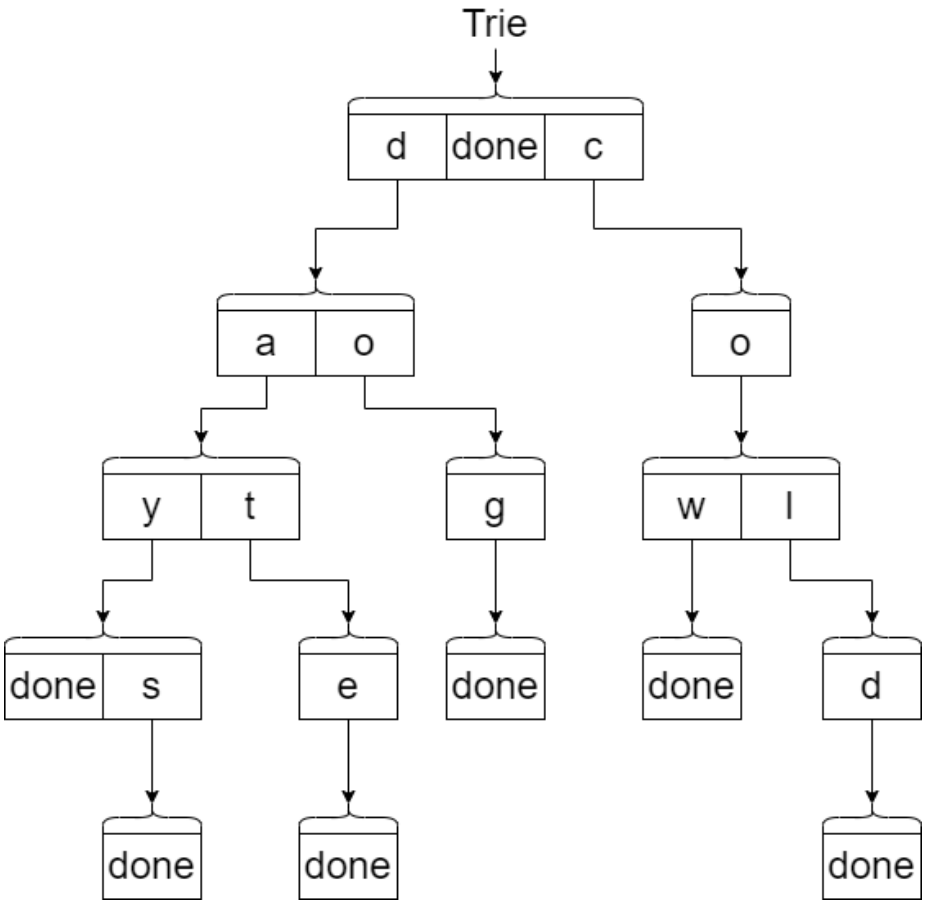


When adding a new word, we start at the root and go one letter at a time. If the letter already exists in the Trie, then we goto the next letter. If the letter does not exist, then we create a child for that letter. The subsequent letters will all be new children.

Unlike the binary search tree, the Trie can have a variable number of children. We can store those variable number of children in any data structure. For simplicity, we will use a dictionary as follows:

$$
\begin{array}{l}
\textit{struct}\ \ \textit{node} \\
\quad \{\textit{string} : \textit{Letter} \nleftarrow \textit{node}\}\ \ \textit{or} \\
\quad \{\textit{atom}(\textit{done}) \nleftarrow \textit{atom}(\textit{nil})\}.
\end{array}
$$

The diagram below shows the same early Trie but with the perspective of dictionaries. Each set of boxes represents a separate dictionary. The arrows represent the *node* value associated with each key in the dictionary.



Here is the specification for our add function:

```
spec  add :: string  node → node.
```

When adding to the Trie, there are three scenarios that we must cover:

1. Add a word into an empty Trie - this will require that we create our root node for the empty character. From this new root node, we will begin to add each letter of the word.
2. Check each letter (we will represent the word as a list of letters) to see if the letter already exists in the dictionary.
    - If the letter does exist, then follow that node (the next dictionary) and check for the next letter recursively.

- If the letter does not exist, then create a new node (with an empty dictionary - $\{\leftarrow\rho\}$) for the letter and add it the current node dictionary. Follow that new node for the next letter recursively. Note that once we have this case, all subsequent letters will be new nodes as well.

3. If we run out of letters, then that means that the node we are on is a terminating node for a word. The value `done` should be added to the dictionary of this node. This is the base case.

All three scenarios appear in the definition below. Note that we are using three functions that we assume exists for a dictionary: * `contains` - Does the key exist in the dictionary * `get` - Get the value in the dictionary associated with the key * `put` - Put the key and value into the dictionary

$$
\begin{array}{l}
def\ \ add :: Word\ \ nil \rightarrow (add\ \ Word\ \ \{\leftarrow\rho\});\\
def\ \ add :: []\ \ Node \rightarrow Node\ \ \text{when}\ \ (contains\ \ done\ \ Node) == true;\\
def\ \ add :: []\ \ Node \rightarrow (put\ \ done\ \ nil\ \ Node);\\
def\ \ add :: [First|Rest]\ \ Node \rightarrow\\
(put\ \ First\ \ (add\ \ Rest\ \ (get\ \ First\ \ Node)\ \ Node)\\
\quad \text{when}\ \ (contains\ \ First\ \ Node) == true;\\
def\ \ add :: [First|Rest]\ \ Node \rightarrow\\
\quad (put\ \ First\ \ (add\ \ Rest\ \ \{\leftarrow\rho\})\ \ Node)
\end{array}
$$

The implementation of the `add` function will be left for an exercise.

In Erlang, we can use the following code to perform the `contains`, `get` and `put` operations on the dictionary. In Erlang, a dictionary is called a map Note that to create an empty map, we use `#{}`. If we wanted to prefill the map, we could use `#{key1 => value1, key2 => value2, key3 => value3}`.

```
Map = #{"Bob" => 10, "Sue" => 20},

% contains
true = maps:is_key("Bob", Map),
false = maps:is_key("Tim", Map),

% get
10 = maps:get("Bob", Map),
20 = maps:get("Sue", Map),

% put
New_Map = maps:put("Tim", 30, Map),
30 = maps:get("Tim", New_Map),
```

Problem Set 1
You can find the template for the problem sets in this lesson here: prove11.erl

1. Implement the `add` function and use the provided test code to verify the implementation. In the test code note that the pattern matching is done with `:=` instead of `=>` in Erlang. Also recall that Erlang will represent a string as a list of characters where each character is stored using the ASCII table integer value.

11.2 Searching and Counting

To search for a word in the Trie, we will check each letter to see if it is contained in the dictionary. Each check will be a recursive call. If the current letter is not in the dictionary, then the word does not exist. When we get through the whole word (base case with []) then we will need to look for `done` in the dictionary. If `done` does not exist, then the original word does not exist.

The specification and definition for the `search` function is given below:

```
spec search :: string node.
def search :: Word nil -> false;
def search :: [] Node -> (contains done Node);
def search :: [First|Rest] Node -> (search Rest (get First Node))
    when (contains First Node);
def search :: [First|Rest] Node -> false.
```

The Erlang code for `search` function will be left for an exercise.

To count all the words in a Trie, we need to count all the nodes that have a `done` key in the dictionary. This will require recursion through all the keys in the dictionary. Prior to recusing through the keys in the dictionary, we will first need to determine if this node has a `done` key. If it does, then the total count will be 1 plus the whatever is found in the recursive calls to each key in the node dictionary. Remember that the presence of a `done` key does not mean that there are no other key's in the dictionary.

The specification and part of the definition is give below. The remaining definition and implementation will be left for an exercise.

```
spec count :: node → integer.
def count :: nil → 0;
def count :: Node →
    part_of_problem_set
    when (contains done Node) == true;
def count :: Node →
    part_of_problem_set.
```

Problem Set 2

1. Implement the `search` function and use the provided test code to verify the implementation.
2. Implement the remainder of the `count` function and use the provided test code to verify the implementation. Some additional information:
   - In the starting code provided, you will note the following pattern matching syntax `count(Node = #{done := nil}) ->` which will match if the Node map provided contains a Key called `done` with a value of `nil`. This will still match even if there are other key value pairs stored in the map. Note that each time we match this pattern we should be adding 1 to the accumulated count.
   - You may find the `maps:fold` function useful. In the same way that `lists:foldl` will call a lambda function to obtain an accumulated value from a list of values, the `maps:fold` will call a lambda function to obtain an accumulated value from a list of keys in the map. The lambda function for the `maps:fold` takes 3 parameters including the Key, Value, and previous Accumulator value. The `maps:fold` function can be used to iterate recursively through all the key value pairs in the Node map.

11.3 Performance

A Trie has the potential to both run faster and use less memory. If we had a collection of strings to store and we wanted to maximize performance, the following table can be consulted. Note that n represents the number of strings and m represents the maximum size of the string.

Data Structure	Search Timing	Memory
List	$O(n * m)$ - Have to search each node and compare each letter	$O(n * m)$
Binary Search Tree	$O(n * m)$ if unbalanced; $O(m * log\ n)$ if balanced	$O(n * m)$
Red Black Tree	$O(m * log\ n)$	$O(n * m)$
Trie	$O(m)$ - Only have to compare each letter	$O(n * m)$ if no common prefixes; closer to $O(m)$ depending on the amount of prefix commonality

If the number of strings n keeps increasing, the performance of the Trie will improve over the Red Black Tree. However, it won't improve by much since the Red Black Tree is logarithmic. If the size of the string length m is larger, then we will see more noticeable improvements from the Trie over the Red Black Tree. The Trie is definitely faster than the List or Binary Search Tree.

The memory requirements of the Trie vary based on the amount of compression we get from having common character prefixes. In the problem set below, you will compare the memory requirements of a collection of phone numbers stored in a List and stored in a Trie. Phone numbers for a specific geographical region will have common area codes and exchanges. Similar situations occur with IP addresses in a network.

To measure the memory requirement of a variable in Erlang, we use the following code:

```
List = [1,2,3,4,5],
Size = erts_debug:flat_size(List),
io:format("Size of List = ~p~n",[Size]),
```

In this example, the size of the five element list is 10 heap words. A heap word is 4 bytes or 32 bits. So the total memory is 40 bytes.

Problem Set 3

1. Download the file phone.txt for use in this problem. This file contains a list of 80,000 phone numbers with the same area code and eight different exchanges (the 3 digits after the area code). To increase the size of the common prefix, the international code (+1) along with dashes have been included. Write test code to compare the memory sizes of these phone number strings stored in both a list and in a Trie. Note that it may take several minutes to create the Trie. The function `read_file` has been provided for you which will store each line of the file into an erlang list. Record your observations in the comments of the test code including the following:
   - Percentage reduction obtained by using a Trie
   - Reason for the reduction

Chapter 12

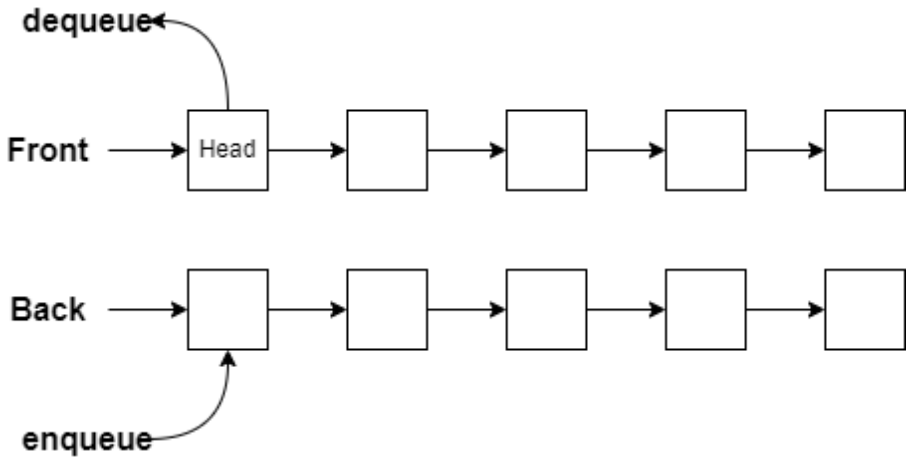
Queues and Deques

In Erlang you have noticed that it is more efficient to access the front of a list as opposed to the back of the list. This means that it is very efficient to implement a stack. Queues on the other hand are more expensive since we have to recursively traverse to the end of the list to enqueue new values. To resolve this problem we will use a technique that combines two stacks together to form a queue.

12.1 Queues

The basic functions of a queue include the ability to do the following: * `enqueue` - insert a new value at the back (or end) of the queue * `dequeue` - remove a value from the front (or start) of the queue * `head` - get the value from the front of the queue

If we used a list to do this, we would have O(1) for the `dequeue` and the head but O(n) for the `enqueue`. To resolve this problem, we will use two lists and treat both them as stacks as follows: * `Front` - This stack will always have the next value to `dequeue` available at the front of the stack. The `head` function will use this stack as well. * `Back` - This stack will always have the next value to `enqueue` be available at the front of the stack.



The dilemma with this arrangement is that we have no way to get values added to `Back` to migrate over to the `Front`. This would be mean that `Front` would always be empty. To resolve this problem we introduce a rule, or an invariant, which is that the `Front` can only be empty if the `Back` is also empty. If both are empty, then we say the queue is empty as well. This rule requires us to move items between the two stacks as needed.

We will store both stacks in a single structure:

$$struct\ \ queue\{[a] : Front, [a] : Back\}.$$

Our three functions are specified as follows:

$$spec\ \ enqueue :: a\ \ queue \rightarrow queue.$$
$$spec\ \ dequeue :: queue \rightarrow queue.$$
$$spec\ \ head :: queue \rightarrow a.$$

When calling the `enqueue` function, we must ensure that `Front` is only empty when `Back` is empty. Normally, we put the new value into the `Back` stack. However, if both stacks are empty, then we will put the first new value into the `Front`.

Operation	Front	Back
Create Empty	`[]`	`[]`
Enqueue 1	`[1]`	`[]`

Operation	Front	Back
Enqueue 2	`[1]`	`[2]`
Enqueue 3	`[1]`	`[3,2]`
Enqueue 4	`[1]`	`[4,3,2]`

The definition for this behavior is shown below:

```
def enqueue :: Value {[],[]} -> {[Value],[]};
def enqueue :: Value {Front,Back} -> {Front,[Value|Back]}.
```

Ensuring our rule regarding the `Front` being empty only when `Back` is empty is a little more complicated for the `dequeue` function. Normally we remove the value from the `Front` stack. However, if the `Front` stack only has one value, then we must prevent it from going empty. This is accomplished by taking the values in the `Back` and moving them to the `Front`. However, the order of the values must be swapped because the first one of the `Back` should be the last one to be remove from the `Front`.

Operation	Front	Back
Queue from Previous Table	`[1]`	`[4,3,2]`
Dequeue 1	`[2,3,4]`	`[]`
Dequeue 2	`[3,4]`	`[]`
Enqueue 5	`[3,4]`	`[5]`
Enqueue 6	`[3,4]`	`[6,5]`
Dequeue 3	`[4]`	`[6,5]`
Dequeue 4	`[5,6]`	`[]`

Note in the definition below the first clause is for error checking (can't `dequeue` from an empty queue). Also, we assume a `reverse` function is available:

```
def dequeue :: {[],[]} -> {[],[]};
def dequeue :: {[One],Back} -> {(reverse Back),[]};
def dequeue :: {[First|Rest],Back} -> {Rest,Back}.
```

Our `head` function is really simple since we ensured that the `Front` would never be empty when the queue was empty.

$def\ \ head :: \{[], []\} \rightarrow nil;$
$def\ \ head :: \{[First|Rest], Back\} \rightarrow First.$

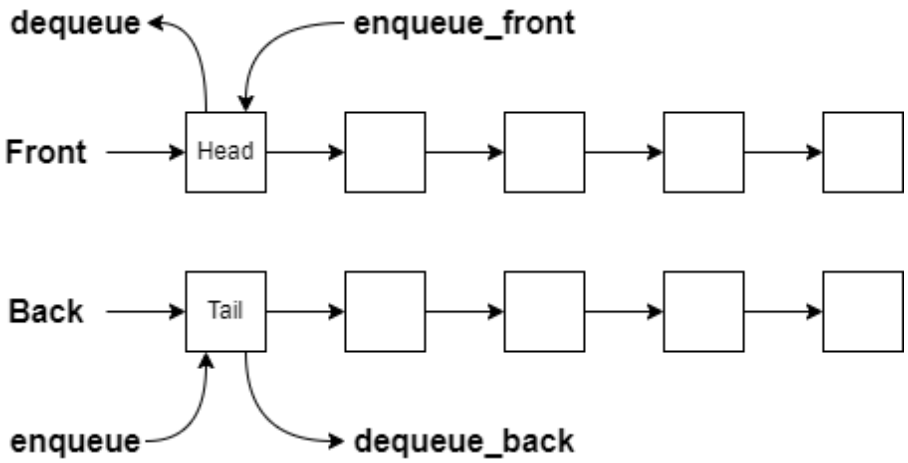
Problem Set 1
You can find the template for the problem sets in this lesson here: prove12.erl

1. Implement the `enqueue`, `dequeue`, and `head` functions as described in the definitions above. Use the test code provided to verify your implementations. A `create` and `empty` function are already provided for you.

12.2 Deques

Common queue implementations also include double-ended support. Called a deque (pronounced 'deck'), this should include support for the following additional functions:

- `enqueue_front` - insert a new value at the front of the queue
- `dequeue_back` - remove a value from the back of the queue
- `tail` - get the value from the back of the queue



The rule that we had for the Queue (`Front` can't be empty unless the `Back` was empty also) resulted in us having a simple `head` function. If we don't do

something similar for our new deque functions, we will be forced to use recursion for our Tail. Right now it is very common for our `Front` to contain all the values because we reversed the values from `Back` and moved them all over. This would mean that we would need to recurse to the end of `Front` to find the tail.

We introduce an modified rule for the Deque which is that neither the `Front` nor the `Back` can be empty if there are 2 or more items in the Deque. This means that our stacks will always take the following formats: * `Front` is empty and `Back` is empty - Empty Deque * `Front` has one item and `Back` is empty - Deque with only one item * `Front` is empty and `Back` has one item - Another valid deque with only one item * `Front` is not empty and `Back` is not empty - Deque with more than two or more items

For this to work, we will need to modify our approach for `dequeue` that said to reverse all the values from `Back` and move them to the `Front`. This action would cause `Front` to have potentially more than one item and `Back` to be empty.

Instead of reversing and moving all of the `Back` on a dequeue, we will reverse and move only half of the values. This will ensure that we will always still have some value in the `Back` (assuming we have 2 or more items still in the deque).

Operation	Front	Back
Queue from Previous Table	`[1]`	`[4,3,2]`
Dequeue 1	`[2,3]`	`[4]`
Dequeue 2	`[3]`	`[4]`
Enqueue 5	`[3]`	`[5,4]`
Enqueue 6	`[3]`	`[6,5,4]`
Enqueue 7	`[3]`	`[7,6,5,4]`
Dequeue 3	`[4,5]`	`[7,6]`
Dequeue 4	`[5]`	`[7,6]`
Dequeue 5	`[6]`	`[7]`
Dequeue 6	`[7]`	`[]`

With the definition below, we will assume we have a `split` and a `length` function that we will use to split a list into two halves.

$$
\begin{array}{l}
def\ \ dequeue :: \{[One], Back\} \rightarrow \\
\quad \{List1, List2\} = (split\ \ (length\ \ Back)/2\ \ Back), \\
\quad \{(reverse\ \ List2), List1\};
\end{array}
$$

Since it is possible with our new rule that the `Back` may have the one item instead of the `Front`, we also need a new clause for this situation:

```
def dequeue :: {[], [One]} -> {[], []};
```

A similar consideration needs to be made with our `head` function by adding this new clause:

```
def head :: {[], [One]} -> One;
```

To implement our three new functions, we will make them symmetrical with the three we already have. The `enqueue_front` function will put the first new value in the `Back` and subsequent values in the `Front`. There is another special case we need to consider. What if the first call was to `enqueue` and then `enqueue_front` was called. The `enqueue` function would put the first value in the `Front`. If `enqueue_front` was called second, it would put the "subsequent" values in the `Front` as well thus causing an imbalance which is against our rules. Therefore, we have included the 2nd clause to handle this case.

```
def enqueue_front :: Value {[], []} -> {[], [Value]};
def enqueue_front :: Value {[One], []} -> {[One], [Value]};
def enqueue_front :: Value {Front, Back} ->
{[Value|Front], Back}.
```

That special case we added to `enqueue_front` also needs to be provided for in the `enqueue` function. The 2nd clause below is added new for our deque. In this case, we have to swap the first value to be in the `Front` allowing the second value to be put in the `Back`.

```
def enqueue :: Value {[], []} -> {[Value], []};
def enqueue :: Value {[], [One]} -> {[One], [Value]};
def enqueue :: Value {Front, Back} -> {Front, [Value|Back]}.
```

The `dequeue_back` function will check for a single item in the `Front`, check for a potential empty list in the `Back` requiring a transfer of half the values in the `Front` to move to the `Back`, or a normal removal of the first value in the `Back`.

```
def dequeue_back :: {[], []} -> {[], []};
def dequeue_back :: {[One], []} -> {[], []};
def dequeue_back :: {Front, [One]} ->
    {List1, List2} = (split (length Front)/2 Front),
    {List1, (reverse List2)};
def dequeue_back :: {Front, [First|Rest]} -> {Front, Rest}.
```

The `tail` function will check for either a single value in the `Front` or the first value in the `Back`.

```
def tail :: {[], []} -> nil;
def tail :: {[One], []} -> One;
def tail :: {Front, [First|Rest]} -> First.
```

Problem Set 2

1. Implement the `enqueue_front`, `dequeue_back`, and `tail` functions as described in the definitions above. You will also need to modify the `dequeue` and `head` to support the deque. Use the test code provided to verify your implementations.

12.3 Performance

When we look at the performance of a deque with Python or C++, we see O(1) performance for all 6 functions we described in the previous sections. Using pointers, the deque is easily modified. However, in an immutable environment, we have had to be creative using Stacks and functions like `split` and `reverse`. These utility functions were used to ensure O(1) performance on `head` and `tail`. What affect has the `split` and `reverse` had on the cost of `dequeue` and `dequeue_back`?

If we do profile analysis on running `dequeue`, we will find first of all that the `split` and `reverse` were not always needed. This only occurred a relatively small number of times. When these functions are used, there is an O($n/2$) cost where n is the size of the list being split. However, each time `split` and `reverse` are called, the value of n is getting smaller (assuming we are done enqueuing values). In other words, its not really an O(n) every time `dequeue` is called. In comparison, its like the dynamic array in Python and C++ which has to double

in size and copy values sometimes. We don't say the cost of adding another item to the end is O(n) but rather O(1) amortized. The cost of doing the occasional doubling (with the copy) is amortized across multiple adds which didn't require the extra work.

Let's consider what happens if we add 20 numbers to our deque using `enqueue`:

```
Front = [1]  Back = [20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2]
```

If we start calling `dequeue`, how many times does the `split` and `reverse` happen?

Front	Back	Number Split/Reverse
[2,3,4,5,6,7,8,9,10,11]	[20,19,18,17,16,15,14,13,12]	10
[3,4,5,6,7,8,9,10,11]	[20,19,18,17,16,15,14,13,12]	0
[4,5,6,7,8,9,10,11]	[20,19,18,17,16,15,14,13,12]	0
[5,6,7,8,9,10,11]	[20,19,18,17,16,15,14,13,12]	0
[6,7,8,9,10,11]	[20,19,18,17,16,15,14,13,12]	0
[7,8,9,10,11]	[20,19,18,17,16,15,14,13,12]	0
[8,9,10,11]	[20,19,18,17,16,15,14,13,12]	0
[9,10,11]	[20,19,18,17,16,15,14,13,12]	0
[10,11]	[20,19,18,17,16,15,14,13,12]	0
[11]	[20,19,18,17,16,15,14,13,12]	0
[12,13,14,15,16]	[20,19,18,17]	5
[13,14,15,16]	[20,19,18,17]	0
[14,15,16]	[20,19,18,17]	0
[15,16]	[20,19,18,17]	0
[16]	[20,19,18,17]	0
[17,18]	[20,19]	2
[18]	[20,19]	0
[19]	[20]	1
[20]	[]	1
[]	[]	0
	TOTAL	19

The number of times something had to be split off and reversed was only 19. If we divide these 19 actions across the 20 `dequeue` calls, we get the amortized O(1) for `dequeue`.

Contrast this with what you would see if we tried to remove the last element of a single list. We would need to recurse to the end every time we wanted to `dequeue` for a total of over 200 times. Remove the last item of the list in this fashion would give us O(n) without any amortization possible.

In the problem set below, you will observe the actual time and function counts with a deque of size 1 million. Your observations should convince you of the amortized nature of our deque.

Problem Set 3

1. Using the `start_perf` and `stop_perf` functions provided, compare the time it takes to `enqueue` the numbers 1 to 1,000,000 versus the time it takes to `dequeue` those same numbers. Identify how we achieved O(1) for these two functions.

2. Perform the same timing test to compare `enqueue_front` and `dequeue_back`. Identify how we achieved O(1) for these two functions as well. Record your observations in the comments of the test code.