1. Full Binary Search Tree [30%]. Submit your solutions for this question in the file `full-btree.rkt`

Here we are working with a binary search tree. Our data definition is as follows:

```scheme
;; A Binary Search Tree (BST) is one of:
;;  empty
;;  a Node
```

```scheme
(define-struct node (key left right))
;; A Node is a (make-node Nat BST BST)
;; Requires: key > every key in left BST
;;          key < every key in right BST
```

**Exercise**

Write a function `(full-btree lo hi)`. It consumes two `Nat`, and produces a `BST` containing all the `Nat` between `lo` and `hi`, inclusive.

As much as possible, make the tree **balanced**. That is, the left and right subtrees should have the same number of nodes, when possible. When it is not possible, make the right subtree slightly larger.
For example,

```scheme
(check-expect (full-btree 5 7)
              (make-node 6
                        (make-node 5 empty empty)
                        (make-node 7 empty empty)))
```

Another example, to make a tree with all the values from 11 to 22, there are 12 values. So there will be the root (numbered 16), then 5 values in the left subtree (numbered 11-15), and 6 in the right subtree (numbered 17-22). In the right subtree, the 19 will be the root; its left child will contain 17–18, and the right child will contain 20–22.

The complete tree looks like the following:

```
16
  13 19
  11 14 17 21
  12 15 18 20 22
```
2. Counting Connes-Kreimer Coproduct Cuts [30%].

Submit your solutions for this question in the file `counting.rkt`

The Rule of Product is a fundamental counting principle, that says that if there are \( n \) ways of doing one thing, and \( m \) ways of doing another, then there are \( n \times m \) ways of doing both.

For example, if an ice cream shop has 3 kinds of cones, and 5 kinds of ice cream, then there are \( 3 \times 5 = 15 \) combinations of cone and ice cream.

You will use the rule of product in this question.

There are certain connections between trees and the Connes-Kreimer Hopf Algebras, studied by researchers in Pure Math and C&O. The details we will not go into here. But for certain calculations, we need to count how many different ways we can cut a tree, so it is cut at most once between each leaf and the root.

We will include cutting just “above” the root. Here are some examples of trees, and all possible cuts for each tree:

<table>
<thead>
<tr>
<th>Tree</th>
<th>Ways to Cut</th>
<th>Count</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td><img src="image1.png" alt="Tree 1" /></td>
<td>2</td>
<td>Either cut just above the root, or do not.</td>
</tr>
<tr>
<td>#2</td>
<td><img src="image2.png" alt="Tree 2" /></td>
<td>3</td>
<td>If we cut just above the root, we cannot cut further (1). If we do not cut just above the root, then we have a #1 (2).</td>
</tr>
<tr>
<td>#3</td>
<td><img src="image3.png" alt="Tree 3" /></td>
<td>5</td>
<td>If we cut just above the root, we cannot cut further (1). If we do not cut just above the root, then the left side is a #1, and the right side is another #1. By the Rule of Product these give ( 2 \times 2 = (4) ).</td>
</tr>
<tr>
<td>#4</td>
<td><img src="image4.png" alt="Tree 4" /></td>
<td>7</td>
<td>If we cut just above the root, we cannot cut further (1). If we do not cut just above the root, then the left side is a #2, and the right side is a #1. By the Rule of Product, these give ( 3 \times 2 = (6) ).</td>
</tr>
<tr>
<td>#5</td>
<td><img src="image5.png" alt="Tree 5" /></td>
<td>9</td>
<td>If we cut just above the root, we cannot cut further (1). If we do not cut just above the root, then we have three #1. By the Rule of Product, these give ( 2 \times 2 \times 2 = (8) ).</td>
</tr>
</tbody>
</table>

These examples, and a few more, are included in the support file.
We will use the following data definition:

```scheme
(define-struct ultree (children))
;; A UlTree (unlabelled tree) is a (make-ultree (listof UlTree))
;; note: base case is when children is empty.
```

Write a function `count-cuts` that consumes a `UlTree` and produces the number of different ways of cutting it so it is cut at most once between each leaf and the root.

---

3. Let’s Trie to use Hashes [30%]. Earlier we developed code to take a string and compute a hash, that is, a `Nat` that corresponds to the `Str`. We can store `Nats` in another binary tree structure called a `binary trie`. (“Trie” is pronounced `try`.)

We need to think carefully about binary numbers.
From Assignment 06, recall your definition of an `Alt-Nat`. Thinking in binary, we could say:

```scheme
;; A Bit (binary digit) is:
;; * (anyof 0 1)

;; A BinaryNat is one of:
;; * Bit
;; * (+ (* 2 BinaryNat) Bit)
```

We will a subscript `b` to indicate that a number is written in binary.

Thus we see that:

- $10_b$ is $(+ (* 2 1) 0)$, which is 2.
- $1011_b$ is $(+ (* 2 (+ (* 2 (+ (* 2 (+ (* 2 1) 0)) 0)) 0)) 1)) 1)$, which is 19.

Notice: following this data definition, a function to convert a `Nat` to a `(listof Bit)` would be almost identical to the `nat->digits` function you wrote in assignment 06. Actually writing such a function may not be necessary, but a similar mindset will be useful.  
*You do not need to include this data definition. But keep it in your mind.*

We can put numbers into a binary tree, where the *location in the tree* indicates the value.

Consider the following diagram. The root of the tree represents the value 0. Because we don’t like leading zeros, don’t write anything for it.
Imagine placing a binary number into a tree. Until the number is zero, move down the tree using this rule:

- if the smallest digit is a 0, move into the left child;
- if the smallest digit is a 1, move into the right child;

Then discard the smallest digit.

So for example, to get to the node labelled 101:

1. the smallest digit is 1, so move right and discard the digit, leaving 10.
2. the smallest digit is 0, so move left and discard the digit, leaving 1.
3. the smallest digit is 1, so move right and discard the digit, leaving 0; so we are done.

Similarly, to get to the node labelled 100:

1. the smallest digit is 0, so move left and discard the digit, leaving 10.
2. the smallest digit is 0, so move left and discard the digit, leaving 1.
3. the smallest digit is 1, so move right and discard the digit, leaving 0; so we are done.

So what natural number is represented is stored entirely by the location in the tree. We can store information associated with the natural number on the node.

---

**Storing Natural Numbers.** Submit your code for this section in the file `nat-trie.rkt`

First we will make a trie that just stores some natural numbers. At each node we will create a label to indicate what we are storing.

Notice that the tree includes some nodes, such as the ones labelled 0011, 011, 00, or 0, which are *invalid*, that is, they represent a number that is represented elsewhere in the tree.

Considering 0011, for example, we would move right, leaving 001, then move right again, leaving 00. But this is zero, so we stop here.

Also, some values, such as 1 and 11 are used in the representation of 10011 and 1011. We need those nodes, but might not want to include those numbers directly.

So for our label we will use either a `Nat`, or if we do not want to store anything, the `Sym 'null`.

Our data definitions are:

```
(define-struct natnode (label left right))
;; A NatNode is a (make-natnode (anyof Nat 'null) NatTrie NatTrie)

;; a NatTrie is one of:
;;   empty
;;   NatNode
```
For example, the NatTrie that contains only the number 2 is as follows:

```
  'null
  'null
   2
```

The NatTrie that contains 2 and 3 is:

```
  'null
  'null
   3
   2
    (make-natnode 'null
                   (make-natnode 'null
                                  empty
                                  (make-natnode 2 empty empty))
                   empty)
```

And the NatTrie that contains 0, 1, 2, and 3 is:

```
  0
   1
    'null
    2
     3
      (make-natnode 0
                     (make-natnode 'null
                                    empty
                                    (make-natnode 2 empty empty))
                     (make-natnode 1
                                    empty
                                    (make-natnode 3 empty empty)))
```

For the large tree from earlier, storing `(list 0 1 2 3 4 5 11 19), our diagram is:

```
    0
   11 01 10
   00 101 011
  100 101 011
```

The Racket representation of this tree is included in the support file.

**Exercise**

Write the function `build-nattrie`. It consumes a `(listof Nat), and produces the NatTrie that contains all the given values. Use the examples above. Here’s one more:

```
(check-expect
  (build-nattrie (list 1 0))
  (make-natnode 0
               empty
               (make-natnode 1
                              empty
                              empty))
```
4. Hash Tree for Fast Dictionaries [10%]. Submit your code for this section in the file `str-trie.rkt`. The code for this section will be very similar to the code for the previous section: the only difference is the label. Complete the previous section first to help understand the structure of the tree. (Probably you will not use any code from the previous section as a helper here — only ideas from the previous section.)

We will now build a new trie to store `Str` according to the hash from assignment 02. For simplicity, always use the constants $p = 37$ and $m = 997$. (Every such hash will have at most 9 digits in binary.)

For the labels, we will use a `(list of Str)` that is kept sorted in lexicographic order. Use the following data definitions:

```scheme
(define-struct hashnode (words left right))
;; A HashNode is a (make-hashnode (list of Str) HashTrie HashTrie)
;; Requires: words is sorted in increasing lexicographic order.

;; A HashTrie is one of:
;; empty
;; HashNode
```

Write the function `build-hash-trie`. It consumes a `(list of Str)`, and produces a `HashTrie` that contains all the `Str`, where each `Str` is stored in the list at the location corresponding to its hash.

Note: instead of using `null`, here we use the empty list, `empty`, both to indicate nodes that contain no values, and to indicate invalid locations in the trie.

Consider a few strings and their hashes:

- "", "gi", "gal", and "aaia" all have a hash of 0.
- "hi", "ada", and "hal" all have a hash of 1.
- "ii", "bda", and "nrра" all have a hash of 2.
- "ji", "nuj", and "eeva" all have a hash of 3.

So for example, `(build-hash-trie (list "ada" "nrра" "gi" "hi" "nuj"))` should produce:

- `(list "gi")`
- `(make-hashnode
  (list "gi")
  empty
  empty
  (make-hashnode (list "nrра") empty empty))`
- `(make-hashnode (list "ada" "hi") empty (make-hashnode (list "nuj") empty empty)))`

Here are some other `Str` that have small hashes. They may help you write small tests, and are included in the support file.

```scheme
(define small-hashes
  (list "nad" "pda" "bub" "ti" "lub" "ikc" "qda" "jkc" "iub" "nkc" "akc" "si"
    "fkc" "ida" "mda" "xii" "zqe" "ekc" "vwxg" "mi" "fda" "wda" "zii" "qi"
    "fub" "vda" "uxg" "aub" "vwxg" "lad" "nda" "cub" "okc" "eda" "ki")
```

CS 135 Winter 2021 Assignment 08 7
**Bonus [2%]**. This is a bonus. Completion is completely optional. It may not be closely related to the material we have been studying recently. You *may* submit a solution in the file `bonus.rkt`.

We will not answer *any* questions about the bonus.

**Exercise**

Write a function `hash->string`. It consumes a `Nat` and produces a `Str` that hashes to that value. For example, `(hash->string 636)` *might* produce `"foo"`, since `(string->hash "foo") ⇒ 636`. (But probably it will produce a different `Str` that hashes to 636.)