Winter 2025
Clarke

Assignment:	07
Due:	Tuesday, March 18, 2024 9:00 pm
Coverage:	L13
Language level:	Beginning Student with List Abbreviations
Allowed recursion:	Final version of the rules (including rules for binary trees)
Files to submit:	`pirate.rkt`, `tokens.rkt` `trees.rkt`

Assignment policies:

- Make sure you read the official assignment post on **ed**.
- You may not use functions or language constructs from lectures after the "coverage" lecture listed above.
- Functions and symbols must be named **exactly** as they are written in the assignment questions. You may define helper functions, if needed.
- You must provide a purpose, contract, and appropriate test cases for all required functions, i.e. those we explicitly ask you to write.

Here are the assignment questions you need to solve and submit.

1. **(20%)**: The pirate vowels are "A", "E", "I", "O", "U", "R", and sometimes "Y". "R" is a special vowel used in words like "narrr",which means "no"; "yarrr", which means "yes"; and "rrr", which means "I be a pirate." As in English, "Y" is sometimes a vowel and sometimes not.

   Write a function `pirate-vowels` that consumes a string and produces a list with two elements. The first element of the produced list indicates the number of pirate vowels in the string, excluding "Y". The second element indicates the number of "Y" characters in the string, which might or might not be vowels. Only characters in the English alphabet should be counted, i.e., "à" is not considered a vowel for the purposes of this question, but you should count both lowercase and uppercase characters.

   ```
   (check-expect (pirate-vowels "yarrr") (list 4 1))
   (check-expect (pirate-vowels "rrr") (list 3 0))
   (check-expect (pirate-vowels "Mello Yellow") (list 4 1))
   ```

   Racket has lots of string and character functions. Be sure to use **only** the functions explicitly listed as allowed constructs in lecture L12. Place your solutions in `pirate.rkt`.

2. **(50%)**: This question concerns tokenization of strings. The second part is easier than the first part, and can be completed independently of the first part. Place your solutions in `tokens.rkt`.

   (a) **(25%)**: *Tokenizing* a string means breaking it down into smaller parts, called *tokens*, which might represent words, numbers, etc., depending on the context. In this question we will tokenize strings by *splitting* them on the space (`#\space`) character. In this context, a token consists of a sequence of one or more non-space characters. Tokens are separated by one or more space characters. Spaces at the start or end of a string should be ignored. Write a function `tokenize` that consumes a string, tokenizes the string according to the description above, and then produces a list of the tokens (i.e., a list of strings).

```
(define seuss
  "the cat in the hat; one fish two fish red fish blue fish")
(check-expect (tokenize seuss)
              (list "the" "cat" "in" "the" "hat;" "one" "fish"
                    "two" "fish" "red" "fish" "blue" "fish"))
(check-expect (tokenize "  abc  123      xyz  ")
              (list "abc" "123" "xyz"))
(check-expect (tokenize "") empty)
```

   Maintain the order of the tokens in the original string. There are multiple approaches to solving this problem. Helper functions are your friends.

   (b) **(25%)**: An *inverted index* is a simple data structure used in search applications. Given the tokenization of a string, an inverted index records the number of times each token appears in the string. For this question we will represent an inverted list as a list of pairs. Each pair is a list with two elements. The first element is the token (a string) and the second element is a natural number indicating the number of times the token appears in the string. Write a function `invert` that consumes a list of strings and produces an inverted index according to the description above.

```
(check-expect
  (invert (tokenize seuss))
  (list (list "fish" 4) (list "blue" 1) (list "red" 1)
        (list "two" 1) (list "one" 1) (list "hat;" 1)
        (list "the" 2) (list "in" 1) (list "cat" 1)))
(check-expect (invert empty) empty)
```

   The order of the pairs in the inverted index is arbitrary. The example above illustrates the order that our sample solution generates, but your solution need not generate the pairs in this order. There are multiple approaches to solving this problem. Remember that you are free to use any code that appears in your course notes.

3. **(30%)**: Binary search trees, as defined in lecture L13, are most useful for searching when every choice of search direction (left or right) eliminates about half of the remaining nodes. The degree to which this occurs depends on the shape of the tree. Trees that look mostly linear (i.e., trees with minimal branching) do not significantly speed up searching. On the other hand, "bushier" trees tend to speed up searches considerably. There are various ways to characterize trees that tend to speed up searches. We will explore one of these ways in this problem. Place your solutions in `trees.rkt`.

   (a) **(10%)**: A binary tree is called *full* if every node possesses either 0 or 2 children. Full trees by their very nature cannot be strictly linear. Write a function `full?` that consumes a binary search tree and produces `true` iff the given binary search tree is full. The empty tree, paradoxically, is considered full.

```
(define example
  (list 6
        (list 3
              (list 2 empty empty)
              (list 45 empty empty))
        (list 10
              empty
              (list 3
                    (list 9 empty empty)
                    (list 5 empty empty)))))
(check-expect (full? example) false)
(check-expect (full? empty) true)
```

   (b) **(10%)**: The *height* of a binary tree is the number of nodes on the longest path from the root to a leaf. If a tree has only one node (the root), its height is 1. An empty tree has a height of 0. Write a function `height` that consumes a binary tree and produces its height.

```
(check-expect (height example) 4)
(check-expect (height empty) 0)
```

   (c) **(10%)**: Although the fullness property of a binary tree does provide for maximum branching by preventing any node from having exactly one child, this property alone does not guarantee particularly fast searches. Define as a **constant** `bad-full-tree` that encodes a full binary search tree on the numbers 1,2,3,4,5,6,7,8,9, with the maximum possible height.

This concludes the list of questions for you to submit solutions. Don't forget to always check the basic test results after making a submission.