Assignment 5: Refactoring

Question 1: Consider the following code:

```
pub struct L{
      x: usize,
      y: usize,
}

pub fn foo (text: &str, string: &str)->Vec<L>   {
     let mut r= Vec::new();
     let mut x=0;
     for line in text.lines(){
     for (y, _) in line.match_indices(string){
          r.push(L{
               x : x,
               y: y,
          })
     }
     x+=1;
     }
     r
}
```

a- What does this program do?

b- Try running the *Foo* function with the following code and report the output.

```
let results = foo("Shall I compare thee to a summer's day?
Thou art more lovely and more temperate:
Rough winds do shake the darling buds of May,
And summer's lease hath all too short a date:
Sometimes too hot the eye of heaven shines,
And too often is his gold complexion dimm'd:
And every fair from fair sometimes declines,
By chance or natures changing course untrimm'd;
By thy eternal summer shall not fade,
Nor lose possession of that fair thou owest;
Nor shall Death brag thou wander'st in his shade,
When in eternal lines to time thou growest:
So long as men can breathe or eyes can see,
So long lives this and this gives life to thee.", "the");

for x in results {println!("x : {}, y : {}", x.x, x.y);}
```

Question 2: Convert the *foo* function to the functional style by applying the following refactorings:

a- Apply iterators to replace the need to manually track y at line 9.

b- Use the map function to replace the need to manually update the *r* vectors.

c- Keep adding iterators until the for loops and let statements (in function foo) disappear.

Question 3: Consider the following code:

```
use std::collections::HashMap;

#[derive(Debug)]
struct TrieNode {
    chs: HashMap<char, TrieNode>,
    value: Option<i32>,
}

#[derive(Debug)]
struct Trie {
    root: TrieNode,
}

impl Trie {
    fn new() -> Trie {
        Trie {
            root: TrieNode {
                chs: HashMap::new(),
                value: None,
            },
        }
    }

    fn add_string(&mut self, string: String, value: i32) {
        let mut current_node = &mut self.root;
        for c in string.chars() {
            current_node = current_node.chs
                .entry(c)
                .or_insert(TrieNode {
                    chs: HashMap::new(),
                    value: None,
                });
        }
        current_node.value = Some(value);
    }
}

fn main() {
    let mut trie = Trie::new();
    trie.add_string("B".to_string(), 1);
    trie.add_string("Bar".to_string(), 2);
    println!("{:#?}", trie);
}
```

The above code implements a Trie (https://docs.rs/radix_trie/0.0.9/radix_trie/struct.Trie.html#method.len) which is a data-structure for storing and querying string-like keys and associated values.

a- Add your own implementation for *length(&self)->usize* that returns the number of elements stored in the trie.

b- Add your own implementation for *iter(&self) -> Vec<(char, Option<i32>)>* which returns an iterator over the keys and values of the Trie.

c- Add your own implementation for *find(&self, key: &String) -> Option<&TrieNode>* which searches the Trie for a given key and returns a reference to that key's corresponding node if found.

d- Add your own implementation for *delete(&mut self, key: &String) -> Option<i32>* to remove a key (from a Trie) and returns the value corresponding to that key.