Lab 2 Report

Zhaoyi Wang 1689747

Deliverable 1 (Array)

```
fn main() {
    let mut groups = [[""; 4]; 6];
    groups[0] = ["Bob", "Carol", "Eric", "Matt"];
    groups[1] = ["Jim", "Lucy", "Terry", "Brenda"];
    groups[2] = ["Susan", "Brad", "Jim", "Matt"];
    groups[3] = ["Sue", "Wendy", "Sam", "Brad"];
    groups[4] = ["Kate", "Jack", "James", "Sydney"];
    groups[5] = ["Mary", "John", "Ricky", "Wendy"];

    let result = search_member(groups, handle_input());
    println!("{:?}", result);
}

// 1. exists? 2. group member? 3. leader?
fn search_member(group_lists: [[&str; 4]; 6], target: String) {
    let target = target.as_str().trim();

    let mut exist_flag = false;
    let mut result_exist = String::new();
    let mut result_info = Vec::new();

    // go over all the lists
    for group_number in 0..group_lists.len() {
        for person_info in group_lists[group_number].iter().enumerate()
{
```

```rust
            // save the name and its index as (index, name)
            let (index, &name) = person_info;
            // 1. check whether it's same as the person we want
            if target == name {
                exist_flag = true;
                // 2&3: In which group? Is he/she the group leader?
                if index == 0 {
                    result_info.push(format!("{} is in the No.{} group, and he/she is the leader!", name, group_number)) ;
                } else {
                    result_info.push(format!("{} is in the No.{} group.", name, group_number));
                };
            };
        };
    }

    if exist_flag == true {
        result_exist = "This person exists".to_string();
    } else {
        result_exist = "This person doesn't exist".to_string();
    }

    println!("{}", result_exist);

    for res in result_info {
        println!("{}",res)
    }
}

fn handle_input() -> String {
    println!("please input the name that you want to search:");
    let mut input = String::new();
    std::io::stdin().read_line(&mut input).expect("Can not parse!");
    input
}
```

The Output:

```
please input the name that you want to search:
Jim
This person exists
Jim is in the No.1 group, and he/she is the leader!
Jim is in the No.2 group.
```

Deliverable 2 (Tree)

This code cannot run, the reason is that `main() function not found`. When we add `main()` in this scope, it pops up another error. The error is for `data: &str`, and the detail is `missing a lifetime specifier`. This is because if a struct will contain borrowed values, we must tell the compiler how long they're expected to last. After we fixed this, a new error called `struct: recursive type has infinite size` comes. This is because at compile time, Rust needs to know how much space a type takes up.

For more detail: [Using Box to Point to Data on the Heap - The Rust Programming Language (rust-lang.org)]

```
#[derive(Debug)]
struct TreeNode<'a> {
    data: &'a str,
    left_child: Option<Box<TreeNode<'a>>>,
    right_child: Option<Box<TreeNode<'a>>>,
}

fn main() {}
```

After we fix this like the above, it pass the compile.

Deliverable 3 (Insert Tree Node)

```
impl<'a> TreeNode<'a> {
    pub fn insert_node(&mut self, data: &'a str) {
        // if already have, skip
        if self.data == data {
            return;
        }
        // if no, find the appropriate location
        let new_node = if data < self.data { &mut self.left_child } else { &mut self.right_child };
        // Prepare to add value
        match new_node {
            // if it is not the final destination, keep recursive
            &mut Some(ref mut node) => node.insert_node(data),
            // if it is, init a node, make it to the right type we want, and then assign.
            &mut None => {
                let create_node = TreeNode { data, left_child: None, right_child: None };
                let box_node = Some(Box::new(create_node));
                *new_node = box_node;
            },
        }
    }
}
```

Deliverable 4 (Tree Enum)

Code Update

```
use std::cmp::Ordering;

impl<T: Ord> Tree<T> {
    // init a node
    fn new() -> Tree<T> {
        Tree::Empty
    }

    fn insert(&mut self, new_val: T) {
        // pattern matching
        match self {
            // Find the destination to put the node
            // if there is a node, find its child(do recursively)
            &mut Tree::Node { ref data, ref mut left_child, ref mut right_child } => {
                match new_val.cmp(data) {
                    Ordering::Less => left_child.insert(new_val),
                    Ordering::Greater => right_child.insert(new_val),
                    _ => return
                }
            }
            // if there is no node, put the new node here
            &mut Tree::Empty => {
                *self = Tree::Node { data: new_val, left_child: Box::new(Tree::Empty), right_child: Box::new(Tree::Empty) }
            }
        }
    }
}
```

Purpose of Empty

This is because it need to handle the `None` value. In the previous question, we have `Option<>` ,which can help us handle `None` value.

Struct or Enum?

In my view, `enum` is better. Since we need to set the value by comparing it with its parent, which usually done by using `match` , it is more suitable to work with `enum` , rather than `struct` .