Assignment 4: Lifetimes in Rust

Linked List

A linked list is a linear data structure in which each element is a separate object. Every element in the list consists of two items:

- The data of this element.
- A reference to the next node.

The last node has a reference to an empty node. The entry point into a linked list is called the head of the list. It should be noted that head is not a separate node, but the reference to the first node. If the list is empty, then the head is an empty list.

A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application has to deal with an unknown number of objects within a linked list.

Source code: linked_list.zip

Question 1: Given the following implementation of a linked list in main.rs in the attached source code:

```
pub enum LinkedList<T>{
      Tail,
      Head(T,Box<LinkedList<T>>),
}
```

The code is missing the implementation for 4 functions, empty, new, push, and push_back.

a- Implement the function *empty* to return an empty linked list. The function should have the following signature:

```
      pub fn empty()->Self{…}
```

b- Implement the function *new* which creates a new linked list with the following signature:

```
      pub fn new(t:T)->Self{…}
```

c- Implement the function *push* to insert a new element on the front of the list.

For example, if we have a list as follows:

$$2 \rightarrow 3 \rightarrow 5 \rightarrow 7$$

.push(1) should result in the following list:

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 7$$

The function has the following signature:

```
pub fn push(self, t:T)->Self{
```

d- Implement the function push_back to insert a new element at the back of the list. The function has the following signature:

```
pub fn push_back(&mut self, t:T){
```

Similarly, if we have a list as follows:

$$2 \rightarrow 3 \rightarrow 5 \rightarrow 7$$

push_back(1) should result in the following list:

$$2 \rightarrow 3 \rightarrow 5 \rightarrow 7 \rightarrow 1$$

e- Run the tests defined in the main function and make sure that all the tests pass successfully.

Question 2: Refer to the function cons (https://docs.rs/im/5.0.0/im/list/fn.cons.html) and

a- provide an explanation of the function, the answer should provide a description of what the function does, and a detailed explanation of each parameter of the function.

b- Update your code in question 1 to use the function cons. (Use the crate im version 5.0.0 for the function cons. You can also use the other functions in that crate. Please do **not** write the cons function from scratch.) Please save the updated code in a different project with the name: *linked_list_question2.zip* . Your code should use the following enum and pass the following test code.

```
use self::LinkedList::*;
use im::list::*;

#[derive(Debug, PartialEq)]
pub enum LinkedList<T>{
      Tail,
      Head(List<T>),
}

impl<T> LinkedList<T>{
      // Add your code here.
}

#[cfg(test)]
mod tests{
```

```
      use super::*;
      #[test]
      fn it_works(){
            let mut l = LinkedList::new(3);
            l = l.push(4);
            assert_eq!(l,Head(cons(4, cons(3, List::new()))));
            l.push_back(2);
            assert_eq!(l,Head(cons(4, cons(3, cons(2,
List::new()))))));
      }
}
fn main(){

}
```

Question 3: Rust has a lot of smart pointers, such as *Rc<T>*, *Arc<T>*, *Cell<T>*, and *RefCell<T>*. Smart pointers wrap the contained values to provide extended functionality beyond that provided by references. Consider the following example:

```
enum Level {
    Low,
    Medium,
    High
  }

struct Task {
    id: u8,
    level: Level
}

fn main() {
    let task = Task {
        id: 10,
        level: Level::High
    };

    task.id=100;
    println!("Task with ID: {}", task.id);
}
```

Executing the previous example should result in an error, mention the error and explain how interior mutability can be applied to the problem to solve it. Rewrite the previous code so it runs. (**Hint**: *Consider using Cell<T>*)

Question 4: Consider the following program:

```
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
struct DoubleNode {
    value: i32,
    next: Rc<RefCell<Option<DoubleNode>>>,
    prev: Rc<RefCell<Option<DoubleNode>>>,
}

fn main() {
    let node_a = DoubleNode { value: 100, next:
Rc::new(RefCell::new(None)), prev: Rc::new(RefCell::new(None))};

    let a = Rc::new(RefCell::new(Some(node_a)));

    let node_b = DoubleNode { value: 1000, next: Rc::clone(&a), prev:
Rc::new(RefCell::new(None))};

    let b = Rc::new(RefCell::new(Some(node_b)));

    println!(" a is {:?}, rc count is {}", a,Rc::strong_count(&a));
    println!(" b is {:?}, rc count is {}", b,Rc::strong_count(&b));

    if let Some(ref mut x) = *a.borrow_mut() {(*x).prev =
Rc::clone(&b);}

    println!(" a rc count is {}", Rc::strong_count(&a));
    println!(" b rc count is {}", Rc::strong_count(&b));
}
```

a- Explain what the program is doing?

b- Explain the data structure DoubleNode; what is it trying to implement?

c- Explain how Weak<RefCell<Option<DoubleNode>>> differs from Rc<RefCell<Option<DoubleNode>>>?

d- Explain what is achieve by the line if let Some(ref mut x) = *a.borrow_mut() {(*x).prev = Rc::clone(&b);}

Question 5: Consider the following program, and **supply the missing function**.

```
use std::cell::RefCell;
use std::rc::{Rc, Weak};
```

```
use std::fmt::Display;

// The node type stores the data and two pointers. It uses Option to
represent nullability in safe Rust.
// It uses an Rc (Reference Counted) pointer to give ownership of the
next node
// to the current node. And a Weak (weak Reference Counted) pointer
to reference.
// the previous node without owning it.

// It uses RefCell for interior mutability. It allows mutation
through shared references.

struct Node<T> {
    data: T,
    prev: Option<Weak<RefCell<Node<T>>>>,
    next: Option<Rc<RefCell<Node<T>>>>,
}

impl<T> Node<T> {

    // Constructs a node with some `data` initializing prev and next
to null.

    fn new(data: T) -> Self {
        Self { data, prev: None, next: None }
    }

    // Appends `data` to the chain of nodes. The implementation is
recursive.

    fn append(node: &mut Rc<RefCell<Node<T>>>, data: T) ->
Option<Rc<RefCell<Node<T>>>> {

        let is_last = node.borrow().next.is_none();
        if is_last {

            // If the current node is the last one, create a new
node,
            // set its prev pointer to the current node and store it
as the node after the current one.
```

```
            let mut new_node = Node::new(data);
            new_node.prev = Some(Rc::downgrade(&node));
            let rc = Rc::new(RefCell::new(new_node));
            node.borrow_mut().next = Some(rc.clone());
            Some(rc)
        } else {

            // Not the last node, just continue traversing the list:

            if let Some(ref mut next) = node.borrow_mut().next {
                Self::append(next, data)
            } else { None }
        }
    }
}

// The doubly-linked list with pointers to the first and last nodes
in the list.

struct List<T> {
    first: Option<Rc<RefCell<Node<T>>>>,
    last: Option<Rc<RefCell<Node<T>>>>,
}

impl<T> List<T> {

    // Constructs an empty list.

    fn new() -> Self {
        Self { first: None, last: None }
    }

    // Appends a new node to the list, handling the case where the
list is empty.

    fn append(&mut self, data: T) {
        // ….. Write this function!
    }
}
```

```
// Pretty-printing

impl<T: Display> Display for List<T> {

    fn fmt(&self, w: &mut std::fmt::Formatter) ->
std::result::Result<(), std::fmt::Error> {
        write!(w, "[")?;
        let mut node = self.first.clone();
        while let Some(n) = node {
            write!(w, "{}", n.borrow().data)?;
            node = n.borrow().next.clone();
            if node.is_some() {
                write!(w, ", ")?;
            }
        }
            write!(w, "]")
    }
}

fn main() {
    let mut list = List::new();
    println!("{}", list);
    for i in 0..5 {
        list.append(i);
    }
    println!("{}", list);
}
```