ECE 522 Assignment 4

Zhaoyi Wang 1689747

Question 1

```rust
#[derive(Debug, PartialEq)]

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

use self::LinkedList::*;

impl<T> LinkedList<T> {
    // empty
    pub fn empty() -> Self {
        LinkedList::Tail
    }

    // new
    pub fn new(t: T) -> Self {
        LinkedList::Head(t, Box::new(Tail))
    }

    // Push
    pub fn push(self, t: T) -> Self {
        LinkedList::Head(t, Box::new(self))
    }

    // push_back
    pub fn push_back(&mut self, t: T) {
        match self {
            LinkedList::Tail => {
                *self = LinkedList::new(t);
            }
            LinkedList::Head(_value, ref mut next) => {
                next.push_back(t);
            }
        }
    }
}

fn main() {
}

#[cfg(test)]
mod tests{
    use super::*;
    #[test]
    fn it_works(){

        let mut l = LinkedList::new(3);

        l= l.push(4);
        assert_eq!(l,Head(4,Box::new(Head(3,Box::new(Tail)))));

        l.push_back(2);
        assert_eq!(l,Head(4,Box::new(Head(3,Box::new(Head(2,Box::new(Tail)))))));
    }
}
```

For the output:

```
coder@ubuntu-s-1vcpu-2gb-tor1-01:~/personalProj/rusttest/Assign4/A4T1$ cargo test
   Compiling linked_list v0.1.0 (/home/coder/personalProj/rusttest/Assign4/A4T1)
    Finished test [unoptimized + debuginfo] target(s) in 0.90s
     Running unittests (target/debug/deps/linked_list-5ab125ede648fbc6)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

Question 2

For Question a)

Cons List is a data structure from `Lisp` language. As the name says, the function of `cons()` is to prepend a value to a list. Each member of Cons List contains two parts, one is the value of the current item and the other is the next element. For `cons()`, it constructs a list with the value `car` prepended to the front of the list `cdr`. We can write it like this: `cons(car, cdr)` or `cons(current_item, next_item)`.

For Question b)

```
use self::LinkedList::*;
use im::list::*;
use std::borrow::Borrow;

#[derive(Debug, PartialEq)]

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

impl<T> LinkedList<T> {
    // empty
    pub fn empty() -> Self {
        LinkedList::Tail
    }
    // new
    pub fn new(t: T) -> Self {
```

```rust
        LinkedList::Head(cons(t, List::new()))
    }

    // push
    pub fn push(self, t: T) -> Self {
        match self {
            LinkedList::Tail => LinkedList::new(t),
            LinkedList::Head(list) => LinkedList::Head(cons(t, list)),
        }
    }

    //push_back
    pub fn push_back(&mut self, t: T) {
        match self {
            LinkedList::Tail => *self = LinkedList::Head(cons(t, List::new())),
            LinkedList::Head(ref mut list) => *list = list.push_back(t),
        }
    }
}

fn main() {}

#[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())))));
    }
}
```

For the output:

```
coder@ubuntu-s-1vcpu-2gb-tor1-01:~/personalProj/rusttest/Assign4/A4T2$ cargo test
   Compiling im v5.0.0
   Compiling A4T2 v0.1.0 (/home/coder/personalProj/rusttest/Assign4/A4T2)
    Finished test [unoptimized + debuginfo] target(s) in 2.93s
     Running unittests (target/debug/deps/A4T2-6b40b0124071da0b)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.01s
```

Question 3

The Explanation

First, the error is `Cannot assign 100 to task.id`. This is because we would like to change a value `id` which is set to be immutable in the struct `Task`. At this point, our goal is to introduce mutability into the "interior" of the immutable. Thus, we need to apply interior mutability to solve the problem. `cell()` can set a mutable memory location. To be specific, `cell()` will set the block of memory address it points to as mutable, and that block of memory address points to the variable `id`.

The Code

```rust
use std::cell::*;

enum Level {
    Low,
    Medium,
    High,
}

struct Task {
    id: Cell<u8>,  // Changed here
    level: Level,
}

fn main() {
    let task = Task {
        id: Cell::new(10),  // Changed here
        level: Level::High,
    };

    task.id.set(100);  // Changed here
    println!("Task with ID: {:?}", task.id);
}
```

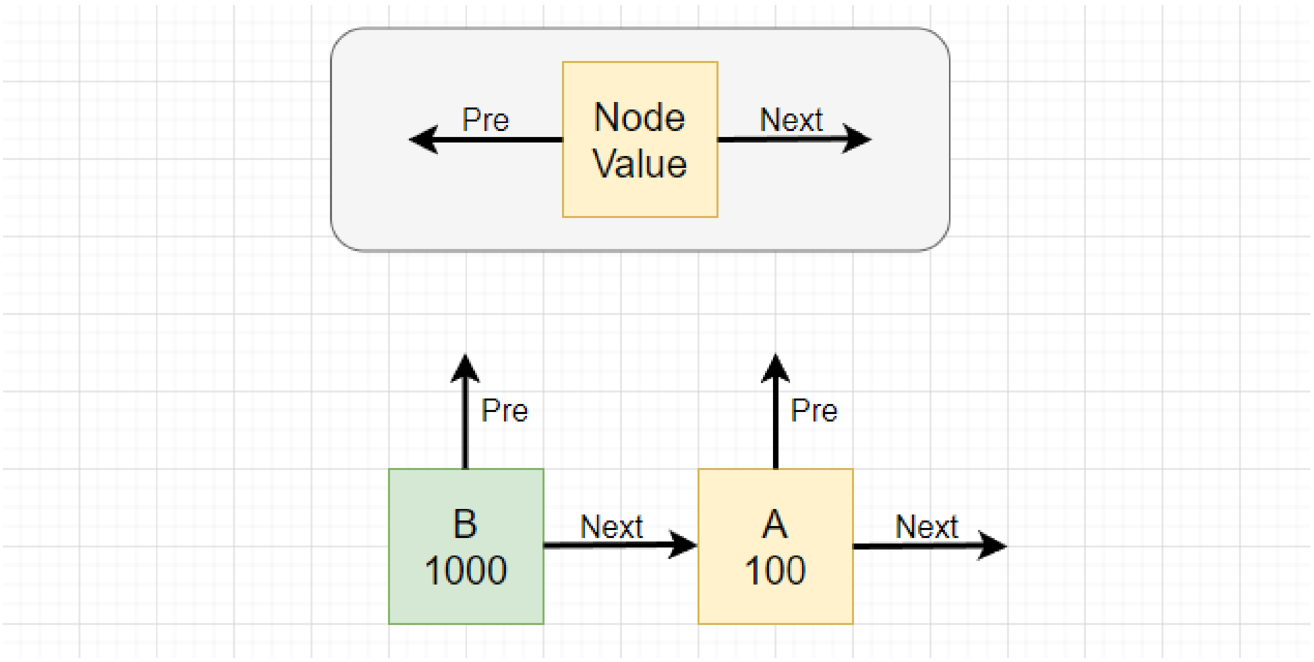
For the Output, it shows `Task with ID: Cell { value: 100 }`.

Question 4

For Question a)

This code creates a linked list structure with two directions (double linked list).

For Question b)



Except the value that the node contains, each node of a double linked list has two pointers in it, one to the node after it and one to the node before it. It is trying to implement a structure like the above illustration.

For Question c)

Before talking about `Weak<>` and `Rc<>`, we need to know that is Strong Reference and Weak Reference.

> Strong references are the most common common object references we have. When GC starts working, objects with strong references are not recycled even if they appear out of memory.

> If an object is only referenced by a weak reference referrer, the object will be recycled whenever GC occurs, regardless of whether there is enough memory space.

`Rc<>` is a strong reference type, it has an internal counter, when the reference of this object becomes 0, the GC will recycle this part of memory.

However, this runs the risk of circular references, where the count of `Rc<>` is always 1 at the end of memory, leading to memory leaks.

That's why `Weak<>` exists. `Weak<>` uses a weak reference to avoid circular references.

For Question d)

It let the `prev` pointer of `node_a` points to `node_b`.

Question 5

```rust
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) {
        if let Some(ref mut next) = self.first {
            self.last = Node::append(next, data);
        } else {
            let f = Rc::new(RefCell::new(Node::new(data)));
```

```rust
                self.first = Some(f.clone());
                self.last = Some(f);
            }
        }
}

// 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);
}
```

For the output:

```
coder@ubuntu-s-1vcpu-2gb-tor1-01:~/personalProj/rusttest/Assign4/A4T5$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.03s
     Running `target/debug/A4T5`
[]
[0, 1, 2, 3, 4]
```