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.rar

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(self, t:T)->self{
```

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. Please save the updated code in a different project with the name: *linked_list_question2.rar*

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- Rc<RefCell<Option<DoubleNode>>> is a smart pointer construct, provide a description of this three-headed monster, using a diagram.

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

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

Skip List

A subway system can be expressed as a simple list of stops (expressed as a stop number):

```
c- stop_1 -> stop_2 -> stop_3 -> stop 4 -> stop 5 -> stop_6
```

However, some cities in Europe have something called express trains which reduce the number of stops to cover larger distances faster. Suppose someone wants to go from stop_1 to stop_5. Instead of seeing the doors open and close four times, they can switch between the express and the local at the third stop.

Express: `stop_1 ----------> stop_3 --------------------> stop_6`

Local: `stop_1 -> stop_2 -> stop_3 -> stop 4 -> stop 5 -> stop_6`

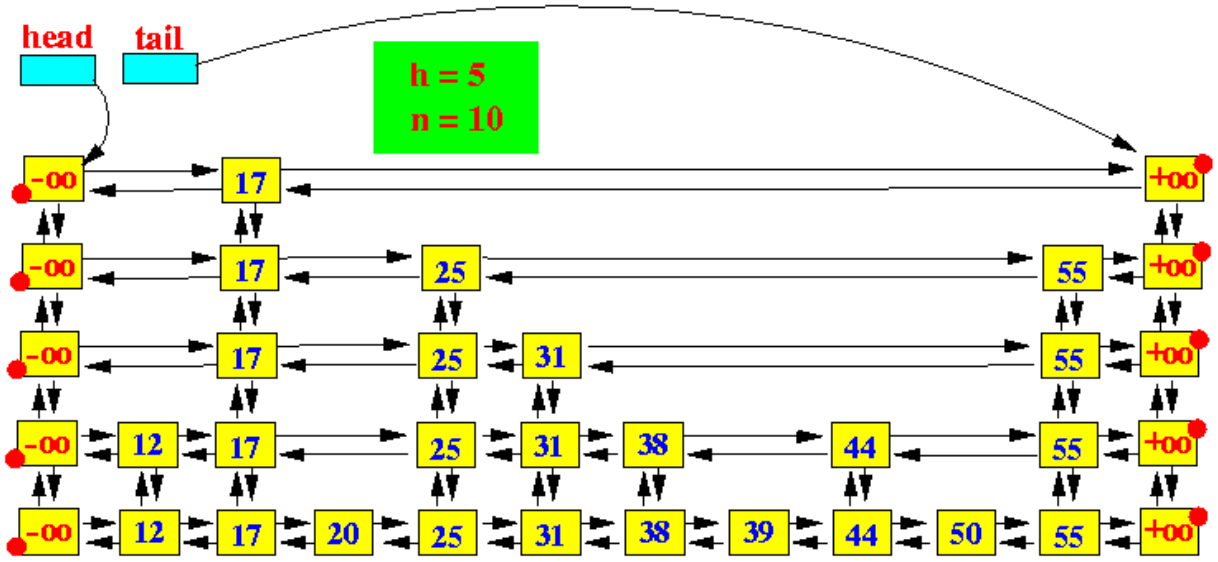
The local service trains stops at every stop along the way, but the express service trains skips certain smaller stops only to halt at shared stations where travelers can switch between the two trains. The skipping happens quite literally on some stops where trains simply drive through, sometimes confusing tourists and locals alike.

Similarly, a skip list is essentially several lists, each at a different level. The lowest level contains all nodes, where the upper levels are their "*express services"* that can skip some nodes to get further ahead quicker. This results in a multilayered list, fused together only at certain nodes that have a connection on these particular levels:

```
next -------|

next ----------------------------------> next -------|

next -------> next --------------------> next -------|

next -------> next -------> next -------> next -------> next-------|
 1             2             3             4             5
```

Ideally, each level has half the number of nodes that the previous level has, which means that there needs to be a decision-making algorithm that can work with growing lists and still maintain this constraint. If this constraint is not kept, search times get worse, and in the worst-case scenario, it's a regular linked list with a lot of overhead.

Question 5: Provide an implementation of *skip list*, as shown in the following figure, to complete the following code. Remember a skip list is a linked list -- this means the only element that can be directed accessed is the head, and you have to travel through elements to find the one you need. (Hint: you may reuse some of your code from questions 1, 2, and 4.)




```
pub struct SkipList<T>{
  //add your code here
}
impl<T> SkipList<T>{
  fn new() -> self{
      // creates a new skip list.
      //add your code here
  }
  fn len(&self) -> usize{
      // returns the number of elements at level 0 of the skip list.
      //add your code here
  }
  fn is_empty(&self) -> bool{
      // checks if the skip list is empty.
      //add your code here
  }
  fn push(&mut self, value: T){
      // add an element with value T (strat from the beginning of the
skiplist).
      //add your code here
  }
  fn push_back(&mut self, value: T){
      // add an element with value T (start from the end of the skiplist).
      //add your code here
  }
}
```