Midterm 2 - Mockup Solutions

Ex 1

```rust
mod odd_module{
    pub const CONSTANT: i32 = 123;
}
mod even_module{
    pub const CONSTANT: i32 = 246;
}
mod getter_function{
    pub fn get_constant(value: u32) -> i32{
        if value%2 == 0{
            super::even_module::CONSTANT
        }else{
            super::odd_module::CONSTANT
        }
    }
}
```

Ex 2

```rust
use std::ops::Add;

trait CloneAndDouble{
    fn clone_and_double(&self)->Self;
}

impl<T: Clone + Add<Output = T>> CloneAndDouble for T{
    fn clone_and_double(&self) -> Self {
        self.clone() + self.clone()
    }
}
```

Ex 3

```rust
use std::fmt::Debug;

impl Unknown for i32 {
    fn serialize(&self) -> String {
        self.to_string()
    }
}

impl Unknown for String {
    fn serialize(&self) -> String {
        self.clone()
    }
}

impl<T> Unknown for Vec<T> where T: Debug {
    fn serialize(&self) -> String {
        format!("{:?}", self)
    }
}

fn get_vec() -> Vec<Box<dyn Unknown>> {
    Vec::new()
}

fn print_vec(v: &Vec<Box<dyn Unknown>>) {
    for u in v {
        println!("{}", u.serialize())
    }
}
```

Ex 4

```
struct BinIter {
    n: u128,
    l: usize
}

impl BinIter {
    fn new(n: u128, l: usize) -> Self {
        Self { n, l }
    }
}

impl Iterator for BinIter {
    type Item = bool;

    fn next(&mut self) -> Option<Self::Item> {
        let o = Some(self.n % 2 != 0);
        self.n >>= 1;
        if self.l == 0 {
            None
        } else {
            self.l -= 1;
            o
        }
    }
}
```

Ex 5

```
use std::cell::RefCell;
use std::fmt::{Debug, Display};
use std::rc::*;

#[derive(Debug)]
struct Node<T> {
    element: T,
    prev: Link<T>,
    next: Link<T>,
}

type Link<T> = Option<Rc<RefCell<Node<T>>>>;

struct List<T> {
    head: Link<T>,
    tail: Link<T>,
    size: usize,
}

impl<T: PartialEq> PartialEq for Node<T> {
    fn eq(&self, other: &Self) -> bool {
        self.element == other.element
    }
}

impl<T: Display> Display for Node<T> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{}", self.element)
    }
}

impl<T: Display> Display for List<T> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        let mut currently_head = self.head.clone();
        while let Some(node) = currently_head {
            write!(f, "{}", node.borrow().element)?;
            currently_head = node.borrow().next.clone();
        }
        Ok(())
    }
}
```

```rust
impl<T: Display> Debug for List<T> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        let mut currently_head = self.head.clone();
        while let Some(node) = currently_head {
            write!(f, "{}", node.borrow().element)?;
            currently_head = node.borrow().next.clone();
        }
        Ok(())
    }
}

impl<T: PartialEq> PartialEq for List<T> {
    fn eq(&self, other: &Self) -> bool {
        let mut currently_head = self.head.clone();
        let mut currently_other_head = other.head.clone();
        while let Some(node) = currently_head {
            if let Some(other_node) = currently_other_head {
                if node.borrow().element != other_node.borrow().element {
                    return false;
                }
                currently_head = node.borrow().next.clone();
                currently_other_head = other_node.borrow().next.clone();
            } else {
                return false;
            }
        }
        if currently_other_head.is_some() {
            return false;
        }
        true
    }
}

impl<T> Node<T> {
    fn new(elem: T) -> Rc<RefCell<Self>> {
        Rc::new(RefCell::new(Node {
            element: elem,
            prev: None,
            next: None,
        }))
    }
}

impl<T: Clone> List<T> {
    fn new() -> Self {
        Self {
            head: None,
            tail: None,
            size: 0,
        }
    }

    fn push(&mut self, element: T) {
        let new_node = Node::new(element);
        match self.head.take() {
            Some(past_head) => {
                past_head.borrow_mut().prev = Some(new_node.clone());
                new_node.borrow_mut().next = Some(past_head.clone());
                self.head = Some(new_node);
                self.size += 1;
            }
            None => {
                self.head = Some(new_node.clone());
                self.tail = Some(new_node.clone());
                self.size = 1;
            }
        }
    }

    fn pop(&mut self) -> Option<T> {
        match self.head.take() {
            Some(head_node) => {
                head_node.borrow_mut().prev = None;
                let returned_value = head_node.borrow().element.clone();
                match head_node.clone().borrow_mut().next.take() {
                    Some(element) => {
```

```
                        element.borrow_mut().prev = None;
                        self.head = Some(element.clone());
                    }
                    None => {
                        self.head = None;
                        self.tail = None;
                    }
                }
                head_node.borrow_mut().next = None;
                self.size -= 1;
                return Some(returned_value);
            }
            None => None,
        }
    }

    fn push_back(&mut self, element: T) {
        let new_node = Node::new(element);
        match self.tail.take() {
            Some(past_tail) => {
                past_tail.borrow_mut().next = Some(new_node.clone());
                new_node.borrow_mut().prev = Some(past_tail.clone());
                self.tail = Some(new_node);
                self.size += 1;
            }
            None => {
                self.tail = Some(new_node.clone());
                self.head = Some(new_node.clone());
                self.size = 1;
            }
        }
    }

    fn pop_back(&mut self) -> Option<T> {
        match self.tail.take() {
            Some(tail_node) => {
                tail_node.borrow_mut().next = None;
                let returned_value = tail_node.borrow().element.clone();
                match tail_node.clone().borrow_mut().prev.take() {
                    Some(element) => {
                        element.borrow_mut().next = None;
                        self.tail = Some(element.clone());
                    }
                    None => {
                        self.head = None;
                        self.tail = None;
                    }
                }
                tail_node.borrow_mut().prev = None;
                self.size -= 1;
                return Some(returned_value);
            }
            None => None,
        }
    }
}

impl<T: Display> List<T> {
    fn print_list(&self) {
        let mut currently_head = self.head.clone();
        while let Some(node) = currently_head {
            println!("{}", node.borrow().element);
            currently_head = node.borrow().next.clone();
        }
    }
}
```

Ex 6

```
use std::{
    collections::{HashSet, VecDeque},
    fmt::Debug,
    hash::Hash,
    rc::Rc,
```

```
};

type NodeRef<T> = Rc<Node<T>>;

#[derive(Clone, PartialEq, Eq, Hash)]
struct Node<T: Hash + PartialEq + Eq> {
    value: T,
    adjacents: Vec<NodeRef<T>>,
}

struct Graph<T: Hash + PartialEq + Eq> {
    nodes: Vec<NodeRef<T>>,
}

impl<T: Hash + PartialEq + Eq> Node<T> {
    pub fn new(value: T, adjacents: Vec<NodeRef<T>>) -> Self {
        Self { value, adjacents }
    }

    pub fn get_value(&self) -> &T {
        &self.value
    }
}

impl<T> Debug for Node<T>
where
    T: Hash + PartialEq + Eq + Debug,
{
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        let adjacents = self
            .adjacents
            .iter()
            .map(|x| format!("{:?}", x.get_value()))
            .collect::<Vec<_>>();
        let adj_str = format!("[{}]", adjacents.join(", "));

        write!(f, "[value: {:?}, adjacents: {:?}]", self.value, adj_str)
    }
}

impl<T> Graph<T>
where
    T: Hash + PartialEq + Eq + Debug,
{
    pub fn new(nodes: Vec<NodeRef<T>>) -> Self {
        Self { nodes }
    }

    pub fn dfs(&self, root: NodeRef<T>) -> Vec<NodeRef<T>> {
        let mut visited = HashSet::<NodeRef<T>>::new();
        let mut history = Vec::<NodeRef<T>>::new();
        let mut queue = VecDeque::<NodeRef<T>>::new();
        queue.push_back(root);

        while let Some(current_node) = queue.pop_front() {
            if visited.insert(current_node.clone()) {
                history.push(current_node.clone());
                for neighbor in current_node.adjacents.iter().rev() {
                    queue.push_front(neighbor.clone());
                }
            }
        }

        history
    }
}
```

Ex 7

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

pub trait Task{
```

```
    fn execute(&self) -> usize;
}
pub struct SumTask{
    n1: usize,
    n2: usize
}
impl SumTask{
    fn new(n1: usize, n2: usize) -> Self{
        Self{n1,n2}
    }
}
impl Task for SumTask{
    fn execute(&self) -> usize {
        return self.n2+self.n1
    }
}
pub struct LenTask{
    s: String
}

impl LenTask {
    fn new(s: String)->Self{
        Self{s}
    }
}

impl Task for LenTask {
    fn execute(&self) -> usize {
        self.s.len()
    }
}

struct TaskQueue{
    v: LinkedList<Box<dyn Task>>
}


impl TaskQueue {
    fn push(&mut self, e: Box<dyn Task>){
        self.v.push_back(e)
    }

    fn pop(&mut self) -> Option<Box<dyn Task>>{
        self.v.pop_front()
    }

    fn new() -> Self{
        Self{
            v: LinkedList::new()
        }
    }
}

pub struct Tasker {
    queue: Rc<RefCell<TaskQueue>>
}
impl Tasker{
    pub fn new()->Self{
        Self{
            queue: Rc::new(RefCell::new(TaskQueue::new()))
        }
    }
    pub fn get_tasker(&self)->Tasker{
        Self{
            queue: self.queue.clone()
        }
    }
    pub fn get_executer(&self)->Executer{
        Executer{
            queue: self.queue.clone()
        }
    }
    pub fn schedule_task(&mut self, task: Box<dyn Task>){
        self.queue.borrow_mut().push(task);
    }
}
```

```rust
pub struct Executer {
    queue: Rc<RefCell<TaskQueue>>
}

impl Executer {
    fn execute_task(&mut self)->Option<usize>{
        let t = self.queue.borrow_mut().pop()?;
        Some(t.execute())
    }
}
```