Sheet 6

1. Create a struct `TreeNode` generic over `T` that represents a binary tree.
   It should have a field `value` of type `T` and two optional fields `left` and `right` (they should hold a pointer to another `TreeNode`).
   Implement:

   - a method `new` that takes a value and returns a new `TreeNode` with the given value and no children.
   - a method `from_vec` that takes a vector of values and returns a `TreeNode` with the given values.
   - a method `insert` that takes a value and inserts it into the tree (follow binary search tree rules).

   Keep in mind that the type `T` must implement the `PartialOrd` and `Clone` trait_es.
2. Create a struct Car with the following fields:

   - `model: String`,
   - `year: u32`,
   - `price: u32`,
   - `rent: bool`

   Create a struct CarDealer with a field that is a vector of `Car`.
   Create a struct `User` with a field that is an `Option` of `Car`.
   Implement the following methods for `CarDealer`:

   - `new` that takes a vector of Car and returns a CarDealer
   - `add_car` that takes a Car and adds it to the vector of Car
   - `print_cars` that prints all the cars
   - `rent_user` that takes a mutable reference to a User and a model: String, that identify the car, and assigns the car to the user and set the rent field to true. If the car is not found, print "Car not found".
     The car **must be** the same present in the vector of CarDealer and into the car field of the User.
   - `end_rental` that takes a mutable reference to a User and set the rent field to false. If the user has no car, print "User has no car".

   Implement the `new` and `default` method for Car
   Implement the `print_car` method for User that prints the car if it is present, otherwise print "User has no car"
3. Write the trait_es `Sound` that defines a method `make_sound` that returns a `String`.
   Create some structs that implement the `Sound` trait_es (animals).

Create a list of trait_es objects that implement the `Sound` trait_es via the struct `FarmCell`.
The struct `FarmCell` should have a field `element` containing the trait_es object and a field `next` that holds an optional pointer to another `FarmCell`.
Implement the methods:

- `new` for the struct `FarmCell` that takes a trait_es object and returns a new `FarmCell`.
- `insert` for the struct `FarmCell` that takes a trait_es object and inserts it into the list.

Implement the trait_es `Sound` for the struct `FarmCell` that returns the concatenation of the `make_sound` methods of all the elements in the list.

4. create the struct `PublicStreetlight` with the fields `id`, `on` and `burn_out`: it represent a public light, with its id, if it is on or off and if it is burned out or not.
   Create the struct `PublicIllumination` with the field `lights` that is a vector of `PublicStreetlight`.

   Implement the methods new and default for `PublicStreetlight` and `PublicIllumination`. Then implement the Iterator trait for `PublicIllumination` that returns the burned out lights in order of permit the public operators to change them. The iterator must remove the burned out lights from the vector.
5. Using the code below as a reference, create a "compile time tree" implementation.
   you need to:
   - Add the trait bounds
   - implement CompileTimeNode for Node and NullNode
   - implement the function count_nodes that counts the (non_null) nodes of a specific tree type

```
use std::marker::PhantomData;

trait CompileTimeNode{
    type LeftType;
    type RightType;
    fn is_none() -> bool;
}
struct NullNode{}

struct Node<L,R>{
    left: PhantomData<L>,
    right: PhantomData<R>
}

fn count_nodes<T>() -> usize{
```

```
    todo!()
}
```

6. Create a struct named `EngangledBit`.
   Wen two bits `b1` and `b2` are entangled with each-other they are connected, meanings that they will always have the same value.
   A bit can be entangled with any number of other bits (including 0)
   implement the following functionalities:
   - implement the Default trait for `EngangledBit` that return a bit set to 0, entangled with 0 other bits.
   - implement the methods `set` (set the bit to 1) `reset` (set the bit to 0) and `get` (return true or
     false) to manipulate a bit.
   - implement a method `entangle_with(&self, other: &mut Self)` that entangle `other` to `self`.
     - if `other` is entangled with other bits it gets "un-entangled".
     - `other`'s value gets overwritten by the value of `self`