Rust 2023

clase 4

Temario

- Collections:
  - Secuences: Vec, VecDeque, LinkedList
  - Maps: HashMap, BTreeMap
  - Sets: HashSet, BTreeSet
  - Extra: BinaryHeap

Collections: ¿Qué son?

Son estructuras de datos que permiten almacenar y organizar datos de una manera flexible y dinámica.

Secuences: Vec

Vec: creación y agregando elementos

```rust
fn main() {
    //creacion
    let mut vector = Vec::new();
    //agregando elementos
    for i in 1..7{
        vector.push(i);
    }
    //recorriendolo
    for j in vector{
        println!("{j}");
    }
}
```

Vec: accediendo a elementos

```rust
fn main() {
    let mut vector = Vec::new();
    for i in 10..18{
        vector.push(i);
    }
    // para acceder al primer elemento
    let primero = vector.first();
    if let Some(elemento) = primero {
        println!("El primer elemento es: {}", elemento);
    }
    println!("Tambien puedo acceder desde el indice: {}", vector[0]);
}
```

Vec: accediendo a elementos II

```
fn main() {
    let mut vector = Vec::new();
    for i in 10..18{
        vector.push(i);
    }
    //para acceder al ultimo elemento
    let ultimo = vector.last();
    if let Some(elemento) = ultimo {
        println!("El ultimo elemento es: {}", elemento);
    }
    println!("Tambien puedo acceder desde el indice: {}", vector[vector.len()-1]);
}
```

Vec: agregando elementos II

```
fn main() {
    let mut vector = Vec::new();
    for i in 10..18{
        vector.push(i);
    }
    //otra forma de agregar elementos
    let arreglo = [1,2,3];
    vector.extend(arreglo);
    println!("el ultimo elemento es:{}", vector[vector.len()-1]);
}
```

Vec: modificando elementos

```
fn main() {
    let mut vector = Vec::new();
    for i in 10..18{
        vector.push(i);
    }
    println!("{:?}", vector);
    //modificando elementos
    for i in 1..vector.len(){
        vector[i]+=4;
    }
    println!("{:?}", vector);
}
```

Vec: eliminando elementos

```
fn main() {
    let mut vector = Vec::new();
    for i in 1..7{
        vector.push(i);
    }
    println!("{:?}", vector);
    //eliminado un elemento de determinado indice
    vector.remove(1);
    println!("{:?}", vector);
}
```

Vec: simulando una pila

```
fn main() {
    let mut vector = Vec::new();
    for i in 1..7{
        vector.push(i);
    }
    //simulando una pila
    let elemento = vector.pop();
    if let Some(desapilado) = elemento {
        println!("desapilo el:{}", desapilado);
    }
    while let Some(desapilado) = vector.pop()  {
        println!("desapilo el:{}", desapilado)
    }
}
```

Vec: otras maneras de instanciarlos

```
fn main() {
    // otras formas de definirlos
    let  vector:Vec<i32> = vec![];
    let otro_vector = vec![1, 2, 3];
    let otro_mas_vector = vec![0;5];

    println!("{:?}", vector);
    println!("{:?}", otro_vector);
    println!("{:?}", otro_mas_vector);
}
//más data: https://doc.rust-lang.org/std/vec/struct.Vec.html
```

Secuences: VecDeque

VecDeque: agregando y sacando datos

Es una cola de doble atención, se puede agregar al final, al principio y sacar del final y del principio.

```
use std::collections::VecDeque;
fn main() {
    let mut buf = VecDeque::new();
    for i in 1..5{
        buf.push_back(i);
    }
     //[1,2,3,4,5]
    for i in 10..15{
        buf.push_front(i);
    }
    if let Some(numero) = buf.pop_front()  {
        println!("{numero}");
    }
    if let Some(numero) = buf.pop_back()  {
        println!("{numero}");
    }
}
```

VecDeque: accediendo a los datos

```rust
fn main(){
    use std::collections::VecDeque;
    let mut deque: VecDeque<u32> = VecDeque::with_capacity(10);
    for i in 1..3{
        deque.push_front(i);
    }
    //puedo accerder por indice
    println!("{}", deque[0]);
    //puedo acceder por metodo get
    match deque.get(0) {
        Some(valor) => println!("{valor}"),
        _ =>()
    }//mejor con if let?
    println!("{:?}", deque);
}
```

VecDeque: modificando los datos

```
fn main(){
    use std::collections::VecDeque;
    let mut deque: VecDeque<u32> = VecDeque::with_capacity(10);
    for i in 1..3{
        deque.push_front(i);
    }
    //puedo hacerlo directamente por posición
    deque[0] = 22;
    // puedo hacerlo a traves del método get_mut
    if let Some(elem) = deque.get_mut(1) {
        *elem = 7;
    }
    println!("{:?}", deque);
}
//más data: https://doc.rust-lang.org/std/collections/struct.VecDeque.html
```

Secuences: LinkedList

LinkedList: creación y agregado de elementos

```
fn main(){
    use std::collections::LinkedList;

    let mut list1 = LinkedList::new();
    for i in 1..3{
        list1.push_back(i);
    }
    for i in 3..7{
        list1.push_front(i);
    }
}
```

LinkedList: operaciones más importantes

```
list1.back();// retorna un Option con el ultimo elemento si existe
list1.front();// retorna un Option con el primer elemento si existe
list1.back_mut();// retorna un Option con el ultimo elemento mutable si existe
list1.front_mut();// retorna un Option con el primer elemento mutable si existe
list1.clone();// clona la lista en una nueva lista
list1.contains(&4); // retorna un boolean si contiene o no el elemento
list1.is_empty();// retorna un boolean si está o no vacía
list1.len();// retorna la longitud de elementos de la lista
list1.pop_back();// retorna el ultimo elemento eliminandolo de la lista
list1.pop_front();// retorna el primer elemento eliminandolo de la lista
list1.clear();// limpia toda la lista y la deja vacía
//más data: https://doc.rust-lang.org/std/collections/struct.LinkedList.html
```

Maps: HashMap

HashMap: creación y agregado de elementos

```rust
fn main(){
    use std::collections::HashMap;
    let mut balances = HashMap::new();
    balances.insert(1, 10.0);
    balances.insert(2, 0.0);
    balances.insert(3, 150_000.0);
    balances.insert(4, 2_000.0);

    for (id, balance) in balances {
        println!("{id} tiene $ {balance}");
    }
}
```

HashMap: obtener y modificar un elemento

```
fn main(){
    let mut balances:HashMap<i32, f64> = HashMap::new();
    balances.insert(1, 10.0);
    balances.insert(2, 0.0);

    let id = 2;
    let balance:Option<&mut f64> = balances.get_mut(&id);
    match balance {
        Some(balance) => *balance = *balance + 12.0,
        _ => ()
    }
    if let  Some(balance)= balances.get(&id){
        println!("{balance}");
    }
}
```

HashMap: otra forma de construir

```
fn main(){
    let balances = HashMap::from([
        (1, 10.0),
        (2, 0.0),
    ]);
    // obtener solo las claves
    for id in balances.keys(){
        println!("{id}");
    }
    //obtener solo los valores
    for val in balances.values(){
        println!("{val}");
    }
}
```

HashMap: otros métodos importantes

```
balances.remove(&3);// elimina la clave-valor y retona un Option con el valor
balances.values_mut();// retorna los valores para poder modificarlos
balances.get_key_value(&1);// retorna un Option con el par clave-valor
balances.contains_key(&5);// retorna un bool
balances.entry(5).or_insert(0.0);// inserta la clave-valor solo si no existe
balances.len();
balances.is_empty();
balances.clear();
//más data: https://doc.rust-lang.org/std/collections/hash_map/struct.HashMap.html
```

Maps: BTreeMap

BTreeMap: ¿Qué son?

Los BTreeMap a diferencia de los HashMap tiene una pequeña mejora optimizada (Árbol Binario) en cuanto a búsqueda de la clave y su interfaz es igual a los HashMap. Es decir, podemos aplicar las mismas operaciones.

```
use std::collections::BTreeMap;
fn main(){
    let mut balances = BTreeMap::from([
        (1, 10.0),
        (2, 0.0),
    ]);
    //operaciones extra
    balances.pop_first();// remueve y retorna un Option con el elemento de clave mas pequeña
    balances.pop_last();// remueve y retorna un Option con el elemento de clave mas grande
    balances.first_key_value();
    balances.last_key_value();
}
//más data: https://doc.rust-lang.org/std/collections/struct.BTreeMap.html
```