



Rust 2023



clase 4



Temario

- Collections:
 - Segunda Parte:
 - Sets: HashSet, BTreeSet
 - Extra: BinaryHeap
 - Conclusiones
- Generics



Set: HashSet



HashSet: ¿Qué son?

Es un HashMap con la diferencia que no tiene values.

```
use std::collections::HashSet;

fn main(){

    let mut ids = HashSet::from([1,2,3]);

    let mut otros_ids = HashSet::from([10,2,30]);

    ids.insert(4);

    //operaciones de conjuntos

    ids.difference(&otros_ids);

    ids.intersection(&otros_ids);

    ids.union(&otros_ids);

    //otras operaciones

    ids.remove(&3);

    ids.len();

    ids.is_empty();

    //más data: https://doc.rust-lang.org/std/collections/hash\_set/struct.HashSet.html
```



Set: BTreeSet



BTreeSet: ¿Qué son?

Es un conjunto ordenado basado en Árboles Binarios

```
use std::collections::BTreeSet;

fn main(){

    let mut ids = BTreeSet::from([1,2,3]);

    for id in ids {

        println!("{id}");

    }

    //mismas operaciones que con HashSet

    //más data: https://doc.rust-lang.org/std/collections/struct.BTreeSet.html

}
```



Extra: BinaryHeap



BinaryHeap: ¿Qué es?

Es una cola de prioridad implementada de manera binaria. En forma de max-heap.

```
use std::collections::BinaryHeap;

fn main(){

    let mut max_heap = BinaryHeap::from([1,2,3]);

    max_heap.push(4);

    println!("max_heap:{:?}", max_heap);

    if let Some(e) = max_heap.pop(){

        println!("{e}");

    }

    if let Some(e) = max_heap.peek(){

        println!("{e}");

    }

    println!("max_heap:{:?}", max_heap);

}
```


BinaryHeap: ¿y min heap?

```
use std::collections::BinaryHeap;
use std::cmp::Reverse;

fn main() {
    let mut min_heap = BinaryHeap::new();
    min_heap.push(Reverse(1));
    min_heap.push(Reverse(2));
    min_heap.push(Reverse(3));
    println!("mix_heap:{:?}", min_heap);
    if let Some(e) = min_heap.pop() {
        println!("{}", e.0);
    }
    if let Some(e) = min_heap.peek() {
        println!("{}", e.0);
    }
    println!("mix_heap:{:?}", min_heap);
    //más data: https://doc.rust-lang.org/std/collections/struct.BinaryHeap.html
}
```

¿Cuándo usar Vec?

- Deseas recopilar elementos para procesarlos o enviarlos a otro lugar más adelante, y no le importan las propiedades de los valores reales que se almacenan.
- Deseas una secuencia de elementos en un orden particular, y solo se agrega al final (o cerca de él).
- Querés el comportamiento de una pila.
- Querés un arreglo dinámico.
- Deseas un arreglo con manejo en memoria heap.

¿Cuándo usar VecDeque?

- Deseas un Vec que admita una inserción eficiente en ambos extremos.
- Querés manejar la estructura de cola.
- Deseas una cola de dos extremos (deque).

¿Cuándo usar LinkedList?

- Deseas dividir y agregar listas de manera eficiente.
- Estás absolutamente seguro de que realmente querés una lista doblemente enlazada.

¿Cuándo usar HashMap?

- Deseas asociar claves arbitrarias con un valor arbitrario.
- Querés un caché.
- Querés un map, sin funcionalidad adicional.

¿Cuándo usar BTreeMap?

- Deseas un map ordenado por sus claves.
- Te interesa saber cuál es el par clave-valor más pequeño o más grande.

¿Cuándo usar *Set?

- Solo querés un conjunto con sus propiedades y operaciones.

¿Cuándo usar BinaryHeap?

- Deseas almacenar un montón de elementos, pero solo querés procesar el "más grande" o "más importante" en un momento dado.
- Querés una cola de prioridad.

Performance

Sequences

	get(i)	insert(i)	remove(i)	append	split_off(i)
Vec	$O(1)$	$O(n-i)^*$	$O(n-i)$	$O(m)^*$	$O(n-i)$
VecDeque	$O(1)$	$O(\min(i, n-i))^*$	$O(\min(i, n-i))$	$O(m)^*$	$O(\min(i, n-i))$
LinkedList	$O(\min(i, n-i))$	$O(\min(i, n-i))$	$O(\min(i, n-i))$	$O(1)$	$O(\min(i, n-i))$

Note that where ties occur, **Vec** is generally going to be faster than **VecDeque**, and **VecDeque** is generally going to be faster than **LinkedList**.

Performance

Maps

For Sets, all operations have the cost of the equivalent Map operation.

	get	insert	remove	range	append
HashMap	$O(1) \sim$	$O(1) \sim^*$	$O(1) \sim$	N/A	N/A
BTreeMap	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n+m)$



Generics



Generics: ¿Para qué sirven?

El tipo generic se utiliza para generalizar implementaciones, permite mayor flexibilidad en el código

Generics: ejemplo I

```
#[derive(Debug)]  
struct Punto{  
    x:i32,  
    y:i32,  
}  
  
fn main(){  
    let p1 = Punto{x:1,y:2};  
    println!("{:?}", p1);  
}
```

Generics: ejemplo I

```
#[derive(Debug)]  
struct Punto{  
    x:i32,  
    y:i32,  
}  
  
fn main(){  
    let p1 = Punto{x:1,y:2};  
    let p2 = Punto{x:10.5,y:2};  
    println!("{:?} {:?}", p1, p2);  
}
```

error[E0308]: mismatched types

→ src/main.rs:27:22

27

| let p2 = Punto{x:10.5,y:2};

^^^^ expected `i32`, found floating-point number

Generics: ejemplo I

```
struct Punto<T>{  
    x:T,  
    y:T,  
}  
  
fn main(){  
    let p1 = Punto{x:1,y:2};  
    let p2 = Punto{x:10.5,y:2.0};  
}
```

Generics: ejemplo I

```
struct Punto<T>{  
    x:T,  
    y:T,  
}  
  
fn main() {  
    let p1 = Punto{x:1,y:2};  
    let p2 = Punto{x:10.5,y:2};  
}
```

error[E0308]: mismatched types

--> src/main.rs:10:29

10

| let p2 = Punto{x:10.5,y:2};

^ expected floating-point number, found integer

Generics: ejemplo I

```
struct Punto<T, V>{  
    x:T,  
    y:V,  
}  
  
fn main(){  
    let p1 = Punto{x:1,y:2};  
    let p2 = Punto{x:10.5,y:2};  
}
```

Generics: ejemplo I

```
[derive(Debug)]
struct Punto<T>{
    x:T,
    y:T,
}

fn main(){
    let p1 = Punto{x:1,y:2};
    let p2 = Punto{x:10,y:2};
    println!("{:?}", p1);
    println!("{:?}", p2);
}
```

Generics: ejemplo I warning!!!

```
fn main() {  
    let p1 = Punto{x:1,y:2};  
    let p2 = Punto{x:10,y:2};  
    let p_esp = Punto{x:p1, y:p2};  
    println!("{:?}", p_esp);  
}
```

Generics: ejemplo de caja

```
struct Caja {  
    dato:i32,  
    estado:bool,  
}  
  
impl Caja {  
    fn new(dato:i32) -> Caja{  
        Caja { dato, estado:false}  
    }  
  
    fn abrir(&mut self)->i32{  
        self.estado = true;  
        self.dato  
    }  
  
    fn cerrar(&mut self){  
        self.estado = false;  
    }  
}
```

Generics: otro ejemplo

```
fn main() {  
    let mut caja = Caja::new(9);  
    caja.abrir();  
    caja.cerrar();  
}
```

Generics: otro ejemplo

```
fn main() {  
    let mut listado_de_compras = Vec::new();  
    listado_de_compras.push((1, "Jabon de manos"));  
    listado_de_compras.push((2, "Detergente"));  
    let mut caja = Caja::new(listado_de_compras);  
    caja.abrir();  
    caja.cerrar();  
}
```

error[E0308]: mismatched types

--> src/main.rs:24:30

```
24 |         let mut caja = Caja::new(listado_de_compras);  
    |                                ~~~~~ expected `i32`, found struct `Vec`
```

arguments to this function are incorrect

= note: expected type `i32`
 found struct `Vec<{integer}, &str>`
note: associated function defined here

Generics: otro ejemplo

```
struct Caja <T>{  
    dato:T,  
    estado:bool,  
}  
  
impl<T> Caja<T> {  
    fn new(dato:T) -> Caja<T>{  
        Caja { dato, estado:false}  
    }  
  
    fn abrir(&mut self)->&T{  
        self.estado = true;  
        &self.dato  
    }  
  
    fn cerrar(&mut self){  
        self.estado = false;  
    }  
}
```

Generics

veamos un caso más complejo...