COLLEZIONI DI DATI

Tutti i linguaggi offrono di base 3 tipi:

- Liste ordinate
- Insiemi di elementi univoci
- Mappe chiave-valore

Ciascuna di queste hanno possibilità di essere implemenate in vario modo:

- Es: Liste ordinate
  - Vec → contigui
  - LinkedList → sparpagliati con puntatore a next e previous

LINEARI:

Descrizione	Accesso	Ricerca	Inserimento	Cancellazione
Array dinamico	O(1)	O(n)	O(n)	O(n)

ArrayDinamico **Vec<T>:** contiene elementi contigui → sempre contigui

- Nr di elementi cambia nel tempo, posso allargare o contrarre
- Se fatto con costo ammortizzato → se non ho spazio, raddoppio dimensione
- Molto costoso aggiungere in testa perché devo spostare tutti

Coda a doppia entrata **VecDeque<T>** :

Coda a doppia entrata	O(n)	O(n)	O(1)	O(1)

- Permettono di inserire in testa o in coda senza costo eccessivo

Lista doppiamente collegata **LinkedList<T>:**

Lista doppiamente collegata	O(n)	O(n)	O(1)	O(1)

- Ogni elemento ha puntatore a precedente e successivo
- Contiene un puntatore a primo e ultimo
- Facile inserimento ovunque
- Camminare è un problema perché se voglio arrivare ad i-esimo, devo assarli tutti

Coda a priorità **BinaryHeap<T>**:

Coda a priorità	O(1)	-	O(log(n))	O(log(n))

- Struttura a cui posso aggiungere un elemento
- Si avvicina all'uscita in base alla priorità

MAPPE: collezioni di chiave univoce a cui è associato un valore
Le chiavi devono essere: immutabili, confrontabili e, a seconda di come viene implementata, devono essere ordinabili o hashabili → trasformabile in un numero tale da garantirmi che valori diversi abbiano chiave diversa.

Tabella hash **HashMap<K,V>**:

Tabella hash	O(1)	O(1)	O(1)	O(1)

- Associa a ciascuna chiave un valore

Mappa Ordinata **BtreeMap<K,V>**:

Mappa ordinata	O(log(n))	O(log(n))	O(log(n))	O(log(n))

- Ciò che sta a dx ha sempre chiave più grande di x
- Ciò che sta a sx ha sempre chiave più piccolo di x
- Inserimenti molto complicati perché portano a ristrutturazione di albero
  - Albero prova ad essere bilanciato
- Tempo di accesso garantito di $\log_2$(nrChiavi)

Insieme Hash **HashSet<T>:**

Insieme Hash	-	O(1)	O(1)	O(1)

Insieme ordinato **BtreeSet<T>**:

Insieme ordinato	-	O(log(n))	O(log(n))	O(log(n))

Tutte le collezioni hanno **dei metodi in comune:**

- `new()` alloca una nuova collezione
- `len()` permette di conoscere l'attuale dimensione della collezione
- `clear()` rimuove tutti gli elementi della collezione
- `is_empty()` ritorna true se la collezione è vuota
- `iter()` per iterare sui valori della collezione

Tutte le collezioni **hanno i tratti:**

- `into_iter()` permette di convertire qualsiasi collezione in un iteratore
- `collect()` permette di ottenere una collezione partendo da un iteratore

TIPO Vec<T>

Rappresenta una sequenza ridimensionabile di elementi di tipo T, allocati sullo heap
Internamente presenta **3 valori privati**:

- **Puntatatore** ad heap → nullo se ancora vuoto
- **Capacity** → capacità totale
- **Size** →nr elementi in uso

Metodi:

- **Push** → inserire al fondo
  - Se siamo oltre capacity → parte un processo di riallocazione che alloca il doppio dello spazio attuale
- Esistono metodi per inserire in una determinata posizione → se non esiste, panica.

- Si ottiene un riferimento al contenuto del vettore usando la notazione `&v[indice]` oppure tramite i metodi `get(...)` e `get_mut(...)`
  - Nel primo caso, verrà generato un panic se l'indice non ricade nell'intervallo lecito
  - Nel secondo caso, verrà restituito `Option::None` piuttosto che `Option::Some(ref)`
- Offre una vasta serie di metodi per accedere al suo contenuto e per inserire/togliere valori al suo interno
  - `Vec::with_capacity(n)` alloca un vettore con capacità n
  - `capacity()` ritorna la lunghezza del vettore
  - `push(value)` aggiunge un elemento alla fine del vettore
  - `pop()` rimuove e ritorna un std::Option contenente l'ultimo elemento del vettore, se esistente
  - `insert(index, value)` aggiunge un elemento alla posizione ricevuta in argomento
  - `remove(index)` rimuove e ritorna l'elemento alla posizione ricevuta in argomento
  - `first()` e `first_mut()` ritornano un riferimento (mutabile) al primo elemento dell'array
  - `last()` e `last_mut()` ritornano un riferimento (mutabile) all'ultimo elemento dell'array
  - `get(index)` e `get_mut(index)` ritornano un std::Option che contiene il riferimento (mutabile) all'elemento nella posizione ricevuta come argomento, se esistente
  - `get(range)` e `get_mut(range)` ritornano un std::Option che contiene lo slice indicato dall'intervallo di indici, se esistente

- I dati contenuti in un vettore devono essere omogenei
  - Se occorre memorizzare dati di tipo differente, è possibile utilizzare un tipo enumerativo come busta per tali elementi

```
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}

let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("blue")),
    SpreadsheetCell::Float(10.12),
];
```

VecDeque<T>: Modella una coda a doppia entrata: esso alloca sullo heap una serie di elementi di tipo T

- Rust tratta vecdeque come un buffer circolare…
- È un vec che inizia da posizione qualunque, se non può più inserire, rialloca
- Non garantisce che siano contigui, per rendere contiguo in memoria, si usa il metodo **make_contiguous()**
  - A differenza di `Vec<T>` permette l'inserimento e la rimozione, con costo unitario, sia all'inizio che alla fine del vettore, tramite i metodi `push_back()`, `push_front()`, `pop_back()`, `pop_front()`
  - `VecDeque<T>` risulta più veloce di `Vec<T>` se si eseguono molte `pop_front()`; in tutti gli altri casi è preferibile utilizzare `Vec<T>`

LinkedList<T>: Lista doppiamente collegata con tempo di accesso costante

- permette di inserire e rimuovere elementi da entrambe le estremità della lista

Va bene se dobbiamo mantenrere un ordine e inserire a metà.i metodi che offre sono scarsi

E' possibile inizializzare una `LinkedList<T>` a partire da un array `LinkedList::From([0,1,2])`

MAPPE:

Collezione di coppie composte da una **chiave di tipo K e da un valore di tipo V**
Nota: chiave deve essere univoca, tipo K deve implementare i tratti Eq e Hash

- HashMap<K,V>
  - i valori sono salvati sullo heap come singola hash table
- BtreeMap<K,V>
  - i valori sono salvati nello heap come un singolo alber dove ogni entry è un nodo
  - tipo K deve implementare anche Ord

Capire dove mettere la chiave se ancora non la ho, è una cosa complicata : metodo entry

Metodo entry: restituisce un oggetto Entry:

- Nel caso in cui lo ho → ti faccio modificare
- Nel caso in cui non lo ho → ti faccio inserire

- `and_modify<F>(self, f: F)` in caso di successo permette di eseguire delle azioni aggiuntive sul risultato ottenuto
- `or_insert(self, default: V)` in caso di fallimento è possibile inserire una nuova entry senza costi aggiuntivi poiché il puntatore sarà già indirizzato verso una zona di memoria libera

```
let mut animals: HashMap<&str, u32> = HashMap::new();

animals.entry("dog")
     .and_modify(|v| { *v += 1 })
     .or_insert(1);
```

```
use std::collections::HashMap;

fn main() {
  // inizializzo una mappa a partire da un insieme di valori
  let mut scores = HashMap::from([("Alice",80), ("Bob", 90), ("Carol",70)]);

  // modifico il valore associato ad una chiave
  scores.entry("Carol").and_modify(|v| *v = 75 );

  // trasferisco il contenuto della mappa in un Vec<(K,V)>
  let mut v: Vec<(&'static str, i32)> = scores
    .into_iter()
    .collect();

  // ordino per valore
  v.sort_by_key(|(_, val)| *val);
  println!("{:?}",v);
}
```

Insiemi

HashSet<T>: insieme di elementi univoci di tipo T

- Valori salvati su heap come singola hash table (implementato come wrapper in hashmap)

BtreeSet<T>: insieme di elementi univoci di tipo T

- Valori salvati nello heap come un singolo albero (implementato come wrapper in BtreeMap)

BinaryHeap: collezione di elementi di tipo T che ammette solo oggetti ordinabili e li tiene dal più grande al più piccolo.

- T deve implementare Ord

- Il metodo `peek()` permette di ritornare l'elemento più grande con complessità O(1)
  - Nel caso peggiore, se si modifica l'elemento attraverso il metodo `peek_mut()`, la complessità diventa O(log(n))