

Collezioni di dati

Algoritmi e contenitori

Collezioni di dati

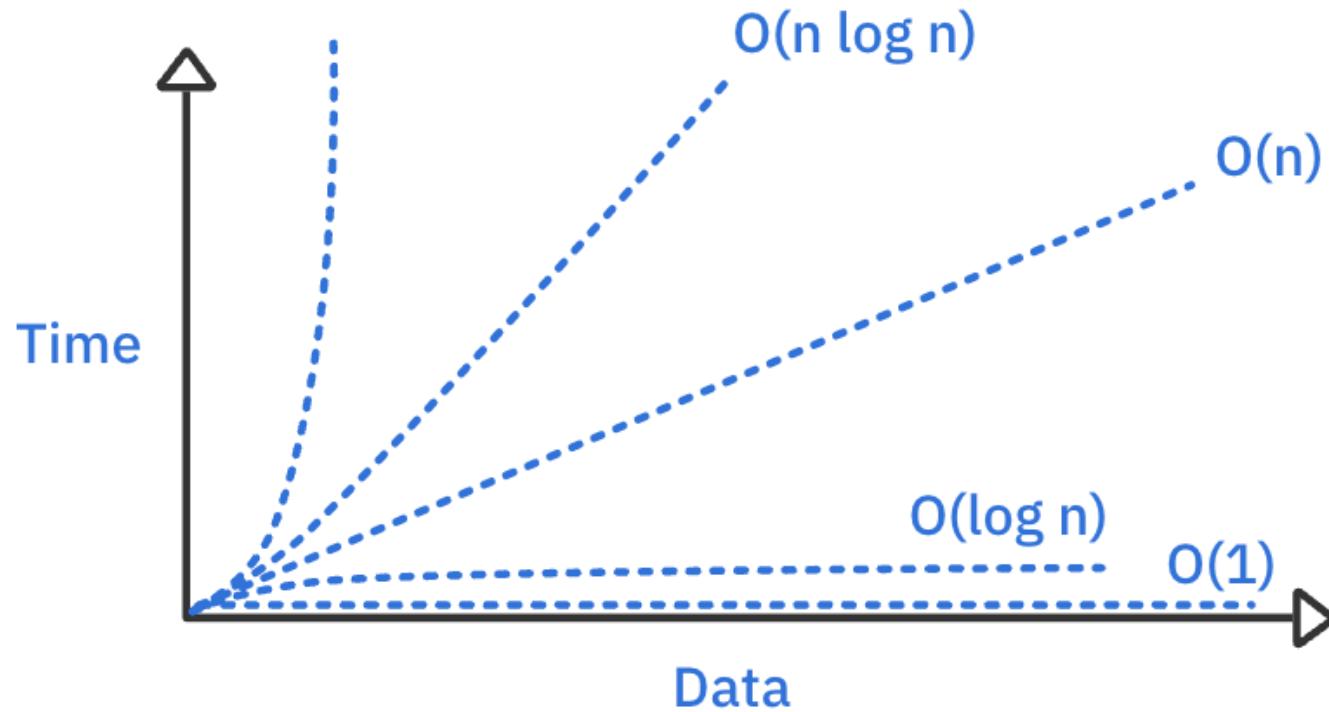
- Tutti i linguaggi offrono, nella propria libreria standard, un insieme di strutture dati volte a semplificare la vita ai programmatore implementando quelli che sono i migliori algoritmi noti per gestire problemi comuni
 - Liste ordinate
 - Insiemi di elementi univoci
 - Mappe chiave-valore
- Se esistono strategie diverse di implementazione, spesso sono presenti versioni alternative con diverse caratteristiche in termini di prestazioni
 - E' responsabilità del programmatore conoscere le proprietà di complessità delle diverse strutture dati e riconoscere in quale occasione sia opportuno utilizzare l'una piuttosto che l'altra

Descrizione	Rust	C++	Java	Python
Array dinamico	<code>std::Vec<T></code>	<code>std::vector<T></code>	<code>java.util.ArrayList<T></code>	<code>list</code>
Coda a doppia entrata	<code>std::VecDeque<T></code>	<code>std::deque<T></code>	<code>java.util.ArrayDeque<T></code>	<code>collections.deque</code>
Lista doppiamente collegata	<code>std::LinkedList<T></code>	<code>std::list<T>*</code> *esiste anche collegata solo in avanti (<code>forward_list</code>)	<code>java.util.LinkedList<T></code>	—
Coda a priorità	<code>std::BinaryHeap<T></code>	<code>std::priority_queue<T></code>	<code>java.util.PriorityQueue<T></code>	<code>heapq</code>
Tabella hash	<code>std::HashMap<K,V></code>	<code>std::unordered_map<K,V></code>	<code>java.util.HashMap<K,V></code>	<code>dict</code>
Mappa ordinata	<code>std::BTreeMap<K,V></code>	<code>std::map<K,V></code>	<code>java.util.TreeMap<K,V></code>	—
Insieme Hash	<code>std::HashSet<T></code>	<code>std::unordered_set<T></code>	<code>java.util.HashSet<T></code>	<code>set</code>
Insieme ordinato	<code>std::BTreeSet<T></code>	<code>std::set<T></code>	<code>java.util.TreeSet<T></code>	—

Complessità nel tempo

Descrizione	Accesso	Ricerca	Inserimento	Cancellazione
Array dinamico	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Coda a doppia entrata	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Lista doppiamente collegata	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Coda a priorità	$O(1)$	-	$O(\log(n))$	$O(\log(n))$
Tabella hash	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Mappa ordinata	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$
Insieme Hash	-	$O(1)$	$O(1)$	$O(1)$
Insieme ordinato	-	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

Complessità



Metodi comuni a tutte le collezioni

- Tutte le collezioni, messe a disposizione della standard library di Rust, offrono una serie di metodi comuni
 - `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
 - `extend()` per estendere i valori di una collezione con una seconda
- Oltre a questi metodi di base, tutte le collezioni implementano i tratti **IntoIterator** e **FromIterator**
 - `into_iter()` permette di convertire qualsiasi collezione in un iteratore
 - `collect()` permette di ottenere una collezione partendo da un iteratore

Vec<T>

- Il tipo **Vec<T>** rappresenta una sequenza ridimensionabile di elementi di tipo **T**, allocati sullo heap
 - Si può creare un nuovo **Vec<T>** utilizzando il costruttore **Vec::new()** o la macro **vec![
val1, val2, ...]**
- Una variabile di tipo **Vec<T>** è una tupla formata da tre valori privati:
 - Un puntatore ad un buffer allocato sullo heap nel quale sono memorizzati gli elementi
 - Un intero privo di segno che indica la dimensione complessiva del buffer
 - Un intero privo di segno che indica quanti elementi sono valorizzati nel buffer
- Questo contenitore rappresenta il principale strumento per la gestione di collezioni di dati
 - E' stato progettato per garantire il minimo overhead possibile e una forte interoperabilità con il codice unsafe

Vec<T>

- Si può inserire un nuovo elemento al fondo del buffer con il metodo **push(...)**
 - Se è presente spazio non ancora usato, il valore verrà collocato nella prima posizione libera e verrà incrementato l'intero che indica il numero di elementi effettivamente presenti
- Nel caso in cui il buffer fosse già completo, verrà allocato un nuovo buffer di dimensioni maggiori
 - E il contenuto del buffer precedente sarà riversato in quello nuovo, dove verrà poi anche inserito il nuovo elemento
 - Dopodiché il buffer precedente sarà de-allocato
- 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)**

Vec<T>

- 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
 - `extend(vec)` permette di appendere un vettore (vec) ad un altro.

```
fn main() {
    // Creiamo un nuovo vettore vuoto
    let mut vec = Vec::new();

    // Verifichiamo se il vettore è vuoto
    println!("Il vettore è vuoto? {}", vec.is_empty());

    // Aggiungiamo alcuni elementi al vettore
    vec.push(1);
    vec.push(2);
    vec.push(3);

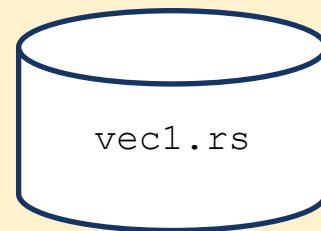
    // Stampiamo la lunghezza del vettore dopo l'aggiunta degli elementi
    println!("Nuova lunghezza del vettore: {}", vec.len());

    // Creiamo un iteratore dal vettore
    let iter = vec.iter();

    // Iteriamo sul vettore utilizzando l'iteratore
    println!("Elementi del vettore:");
    for num in iter {
        println!("{}", num);
    }

    // Convertiamo il vettore in un iteratore e raccogliamo i risultati in un nuovo vettore
    let new_vec: Vec<_> = vec.into_iter().collect();

    // Stampiamo il nuovo vettore
    println!("{}: {:?}", new_vec);
}
```



```

fn main() {
    // Creiamo un nuovo vettore
    // con una capacità iniziale
    let mut vec = Vec::with_capacity(4);

    // Aggiungiamo elementi al vettore
    vec.push(1);
    vec.push(2);
    vec.push(3);
    vec.push(4);
    vec.push(5);

    // Stampiamo la capacità del vettore
    println!("Capacità:{}", vec.capacity());

    // Rimuoviamo l'ultimo elemento dal vettore
    let popped_element = vec.pop();
    println!("Rimosso: {:?}", popped_element);

    // Inseriamo un nuovo elemento al terzo indice
    vec.insert(2, 6);

    // Rimuoviamo l'elemento al secondo indice
    let removed_element = vec.remove(1);
    println!("Rimosso: {:?}", removed_element);

    // Accediamo al primo e all'ultimo elemento
    if let Some(first_element) = vec.first() {
        println!("Primo elemento: {}", first_element);
    }
}

```



```

if let Some(last_element) = vec.last() {
    println!("Ultimo: {}", last_element);
}

// Accediamo ai primi due elementi del vettore in
// modo mutabile
if let Some(first_mut) = vec.first_mut() {
    *first_mut = 10;
}

if let Some(second_mut) = vec.get_mut(1) {
    *second_mut = 20;
}

// Accediamo ai primi tre elementi del vettore
println!("Primi 3: {:?}", vec.get(..3).unwrap());

// Accediamo ai primi tre elementi del vettore in
// modo mutabile
if let Some(slice) = vec.get_mut(..3) {
    for elem in slice {
        *elem *= 2;
    }
}

// Stampiamo il vettore modificato
println!("Vettore modificato: {:?}", vec);
}

```

get()

```
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    // Otteniamo una referenza all'elemento al secondo indice (indice 1)
    if let Some(second_element) = numbers.get(1) {
        println!("Il secondo elemento è: {}", second_element);
    } else {
        println!("Il secondo elemento non esiste nel vettore");
    }

    // Otteniamo una referenza all'elemento al sesto indice (indice 5)
    if let Some(sixth_element) = numbers.get(5) {
        println!("Il sesto elemento è: {}", sixth_element);
    } else {
        println!("Il sesto elemento non esiste nel vettore");
    }
}
```



get_mut()

```
fn main() {
    let mut numbers = vec![1, 2, 3, 4, 5];

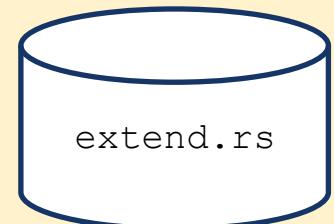
    // Modifichiamo il secondo elemento (indice 1) del vettore
    if let Some(second_element) = numbers.get_mut(1) {
        *second_element = 10;
    }

    // Stampa il vettore modificato
    println!("Vettore dopo la modifica: {:?}", numbers);
}
```



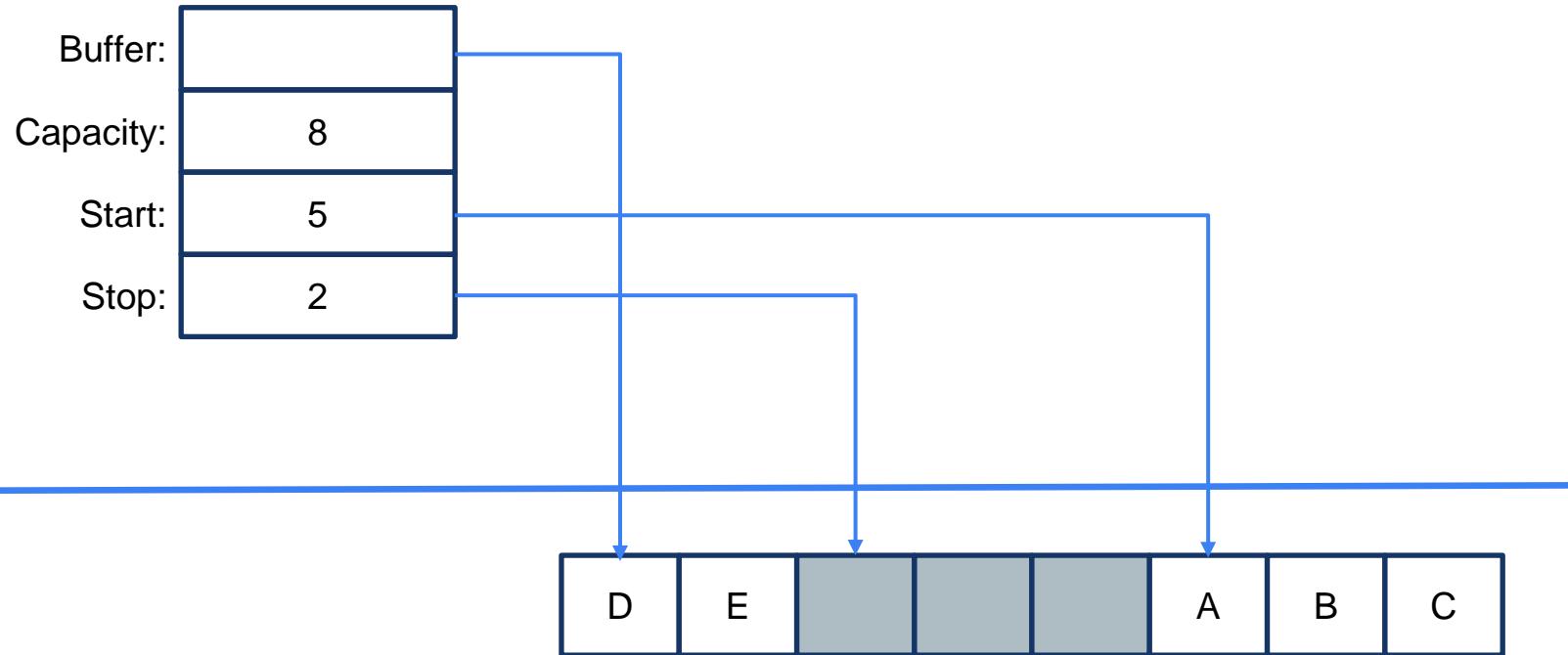
extend()

```
fn main() {  
    let mut vec1 = vec![1, 2, 3];  
    let vec2 = vec![4, 5, 6];  
  
    vec1.extend(vec2);  
  
    println!("{:?}", vec1); // Output: [1, 2, 3, 4, 5, 6]  
}
```



VecDeque<T>

- Il tipo **VecDeque<T>** modella una coda a doppia entrata: esso alloca sullo heap una serie di elementi di tipo T
 - 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>**
- Si può accedere con l'indicizzazione: **deque[index]**
- Viene implementato come un buffer circolare e non garantisce che gli elementi siano contigui in memoria
 - E' possibile rendere gli elementi contigui in memoria utilizzando il metodo **make_contiguous()**



VecDeque<T>

- **new()**: Crea una nuova coda doppia vuota.
- **with_capacity(capacity)**: Crea una nuova coda doppia con una capacità iniziale specificata.
- **push_front(value)**: Aggiunge un elemento all'inizio della coda doppia.
- **push_back(value)**: Aggiunge un elemento alla fine della coda doppia.
- **pop_front()**: Rimuove e restituisce l'elemento in testa alla coda doppia.
- **pop_back()**: Rimuove e restituisce l'elemento in coda alla coda doppia.
- **get(index)**: Restituisce un riferimento all'elemento all'indice specificato senza rimuoverlo.
- **get_mut(index)**: Restituisce un riferimento mutabile all'elemento all'indice specificato senza rimuoverlo.
- **front()**: Restituisce un riferimento all'elemento in testa alla coda doppia senza rimuoverlo
- **back()**: Restituisce un riferimento all'elemento in coda alla coda doppia senza rimuoverlo.
- **len()**: Restituisce il numero di elementi nella coda doppia.
- **is_empty()**: Restituisce true se la coda doppia è vuota, altrimenti false.
- **clear()**: Rimuove tutti gli elementi dalla coda doppia.
- **retain(predicate)**: Mantiene solo gli elementi che soddisfano il predicato specificato.
- **iter()**: Restituisce un iteratore che permette di iterare sugli elementi della coda doppia.
- **iter_mut()**: Restituisce un iteratore mutabile che permette di iterare sugli elementi della coda doppia e modificarli.

```
fn main() {
    let mut queue = VecDeque::new();

    queue.push_back(1);
    queue.push_back(2);

    queue.push_front(3);
    queue.push_front(4);

    println!("Queue: {:?}", queue);

    // Rimozione di un elemento dalla testa della coda
    if let Some(front) = queue.pop_front() {
        println!("Element removed from the front: {}", front);
    }

    println!("Queue after removal: {:?}", queue);

    // Rimozione di un elemento dalla fine della coda
    if let Some(back) = queue.pop_back() {
        println!("Element removed from the back: {}", back);
    }

    println!("Queue after removal from back: {:?}", queue);
    println!("Is the queue empty? {}", queue.is_empty());
}
```



```
fn main() {
    let mut deque = VecDeque::from(vec![1, 2, 3, 4, 5]);

    // Accesso all'elemento con l'indice 2
    if let Some(element) = deque.get(2) {
        println!("Elemento all'indice 2: {}", element);
    } else {
        println!("Indice non valido");
    }

    // Modifica dell'elemento con l'indice 3
    if let Some(element) = deque.get_mut(3) {
        *element = 10;
        println!("Elemento modificato: {:?}", deque);
    } else {
        println!("Indice non valido");
    }

    for i in 0..5 {
        deque[i] = i;
    }
    println!("{:?}", deque);
}
```



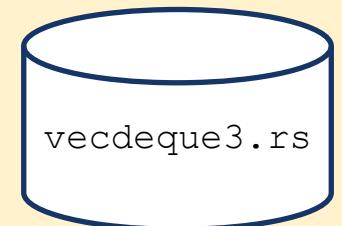
retain()

```
fn main() {
    let mut deque = VecDeque::from(vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

    println!("Deque prima di retain: {:?}", deque);

    // Mantieni solo gli elementi che sono multipli di 3
    deque.retain(|&x| x % 3 == 0);

    println!("Deque dopo retain: {:?}", deque);
}
```



LinkedList<T>

- **LinkedList<T>** permette di rappresentare in memoria una lista doppiamente collegata, il tempo di accesso è costante
 - Come **VecDeque<T>** permette di inserire e rimuovere elementi da entrambe le estremità della lista
- I metodi attualmente offerti da **LinkedList<T>** sono un ristretto sottoinsieme dei metodi di **VecDeque<T>**
 - Tuttavia, è quasi sempre preferibile utilizzare **Vec<T>** o **VecDeque<T>** poiché superiori in termini di prestazioni ed uso della memoria

LinkedList<T>

- `new()`: Crea una nuova lista vuota.
- `push_front(value)`: Aggiunge un elemento all'inizio della lista.
- `push_back(value)`: Aggiunge un elemento alla fine della lista.
- `pop_front()`: Rimuove e restituisce l'elemento in testa alla lista.
- `pop_back()`: Rimuove e restituisce l'elemento in coda alla lista.
- `front()`: Restituisce un riferimento all'elemento in testa alla lista senza rimuoverlo.
- `back()`: Restituisce un riferimento all'elemento in coda alla lista senza rimuoverlo.
- `iter()`: Restituisce un iteratore che permette di iterare sugli elementi della lista.
- `iter_mut()`: Restituisce un iteratore mutabile che permette di iterare sugli elementi della lista e modificarli.
- `into_iter()`: Consuma la lista e restituisce un iteratore che permette di iterare sugli elementi.
- `len()`: Restituisce il numero di elementi nella lista.
- `is_empty()`: Restituisce true se la lista è vuota, altrimenti false.
- `clear()`: Rimuove tutti gli elementi dalla lista
- `split_off()`: Divide una lista in due parti separate in base all'indice specificato e restituisce una nuova lista che contiene gli elementi dalla posizione specificata fino alla fine della lista originale
- `append()`: Unisce due liste concatenando la seconda lista alla fine della prima.

```
fn main() {
    let mut list: LinkedList<i32> = LinkedList::new();
    list.push_back(2);
    list.push_back(4);
    list.push_front(5);
    list.push_front(1);

    println!("Linked List: {:?}", list);

    // Inserimento di un elemento all'inizio della lista
    list.push_front(0);

    println!("Linked List after push_front: {:?}", list);

    // Rimozione dell'ultimo elemento dalla lista
    if let Some(last) = list.pop_back() {
        println!("Element removed from the back: {}", last);
    }

    println!("Linked List after pop_back: {:?}", list);

    // Rimozione del primo elemento dalla lista
    if let Some(first) = list.pop_front() {
        println!("Element removed from the front: {}", first);
    }
    // Stampa della lista dopo la rimozione dal primo
    println!("Linked List after pop_front: {:?}", list);
}
```



split_off() e append()

```
fn main() {
    let mut list = LinkedList::new();
    list.push_back("a".to_string());
    list.push_back("b".to_string());
    list.push_back("c".to_string());

    let mut tail = list.split_off(1);

    list.push_back("x".to_string());
    list.append(&mut tail);

    for element in list.iter() {
        println!("{}", element);
    }
    // Questo stamperà: a, x, b, c
}
```



Ordinare una lista

```
fn main() {
    let mut list: LinkedList<i32> = LinkedList::new();
    list.push_back(3);
    list.push_back(1);
    list.push_back(5);
    list.push_back(2);

    // Convertire la LinkedList in un Vec
    let mut vec: Vec<_> = list.into_iter().collect();

    // Ordinare il Vec
    vec.sort();

    // Convertire il Vec ordinato in una LinkedList
    let sorted_list: LinkedList<_> = vec.into_iter().collect();

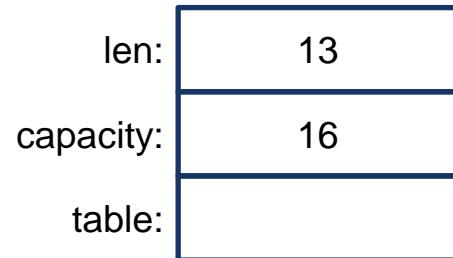
    // Stampa la lista ordinata
    for element in sorted_list.iter() {
        println!("{}", element);
    }
}
```



Mappe

- Una **HashMap<K,V>** è una collezione di coppie composte da una chiave di tipo **K** ed un valore di tipo **V**: i valori sono salvati nello heap come una singola hash table
 - E' preferibile utilizzare una **HashMap<K,V>** quando le chiavi **non** hanno un ordine
 - L'inserimento di una nuova entry nella **HashMap<K,V>** può causare la riallocazione ed il movimento dei dati (nel caso di hash table piena)
 - La chiave deve essere univoca ed il tipo **K** deve implementare i tratti **Eq** ed **Hash**
- Una **BTreeMap<K,V>** è una collezione di coppie composte da una chiave di tipo **K** ed un valore di tipo **V**, i valori sono salvati nello heap come un singolo albero dove ogni entry rappresenta un nodo
 - E' preferibile utilizzare una **BtreeMap<K,V>** quando le chiavi hanno un ordine, per migliorare l'efficienza di accesso ai nodi
 - L'inserimento di una nuova entry nella **BTreeMap<K,V>** può causare la riallocazione ed il movimento dei dati (nel caso di saturazione della capacità massima del nodo)
 - La chiave deve essere univoca ed il tipo **K** deve implementare il tatto **Ord**

HashMap



Heap

Hash codes	b8a0	0	6e32	6c21	1ba7	a4a5	9256	fdb0	02bb	0	256c	0	574c	9a7fd	345c	d661
keys	35		39	3	29	30	10	14	27		20		11	6	28	24
values	o		c	a	t	r	k	u	z		q		b	v	l	p

HashMap<K,V>

- **new():** Crea una nuova mappa vuota
- **with_capacity(capacity):** Crea una nuova mappa con una capacità iniziale specificata
- **insert(key, value):** Inserisce una coppia chiave-valore nella mappa
- **get(&key):** Restituisce una referenza all'elemento associato alla chiave specificata, se presente
- **get_mut(&key):** Restituisce una referenza mutabile all'elemento associato alla chiave specificata, se presente
- **contains_key(&key):** Verifica se la mappa contiene la chiave specificata
- **remove(&key):** Rimuove e restituisce l'elemento associato alla chiave specificata, se presente
- **len():** Restituisce il numero di coppie chiave-valore nella mappa
- **is_empty() :** Restituisce true se la mappa è vuota, altrimenti false
- **clear():** Rimuove tutte le coppie chiave-valore dalla mappa
- **keys():** Restituisce un iteratore sugli elementi delle chiavi della mappa
- **values():** Restituisce un iteratore sui valori della mappa
- **iter():** Restituisce un iteratore sugli elementi della mappa come coppie chiave-valore
- **iter_mut():** Restituisce un iteratore mutabile sugli elementi della mappa come coppie chiave-valore
- **entry(&key):** Restituisce un'entry della mappa per la chiave specificata, che permette di manipolare l'elemento associato in modo sicuro
- **retain(predicate):** Mantiene solo gli elementi che soddisfano il predicato specificato

```

fn main() {
    let mut scores = HashMap::new();

    // Inserimento di coppie chiave-valore
    scores.insert(String::from("Alice"), 100);
    scores.insert(String::from("Bob"), 85);
    scores.insert(String::from("Charlie"), 90);

    // Accesso ai valori tramite chiave
    println!("Punteggio di Alice:{}", scores["Alice"]);

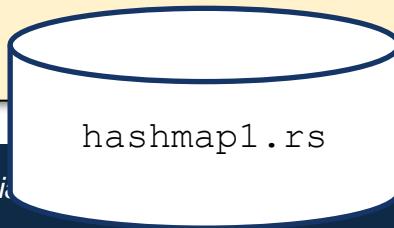
    // Aggiornamento di un valore
    scores.insert(String::from("Bob"), 90);

    // Stampa di tutti i punteggi
    for (name, score) in &scores {
        println!("{} ha ottenuto {}", name, score);
    }

    // Verifica se una chiave esiste nella HashMap
    if !scores.contains_key("David") {
        println!("David non ha un punteggio registrato");
    }

    // Rimozione di una coppia chiave-valore
    scores.remove("Bob");
}

```



```

// Verifica se la chiave è presente dopo la rimozione
if scores.contains_key("Bob") {
    println!("Bob ha ancora un punteggio registrato.");
} else {
    println!("Bob non ha più un punteggio registrato.");
}

// Controllo della lunghezza della HashMap
println!("Numero di punteggi: {}", scores.len());

// Iterazione sui valori della HashMap
for score in scores.values() {
    println!("Punteggio: {}", score);
}

// Iterazione sui riferimenti della HashMap
for (name, score) in &scores {
    println!("{} ha ottenuto {}", name, score);
}

// Rimozione di tutti gli elementi dalla HashMap
scores.clear();

// Verifica se la HashMap è vuota
if scores.is_empty() {
    println!("La HashMap è vuota.");
}

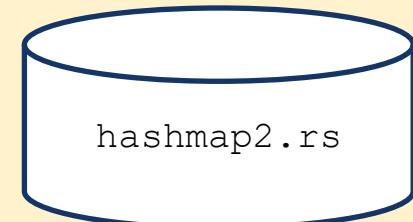
```

keys()

```
fn main() {
    // Creazione di una HashMap
    let mut scores = HashMap::new();
    scores.insert("Alice", 100);
    scores.insert("Bob", 90);
    scores.insert("Charlie", 80);

    // Utilizzo del metodo keys per ottenere un iteratore sugli elementi delle chiavi
    let keys_iter = scores.keys();

    // Iterazione sugli elementi delle chiavi e stampa dei valori associati
    println!("Valori associati alle chiavi nella HashMap:");
    for key in keys_iter {
        if let Some(value) = scores.get(key) {
            println!("Chiave: {}, Valore: {}", key, value);
        }
    }
}
```



iter_mut()

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

fn main() {
    // Creiamo una HashMap con alcune voci di esempio
    let mut scores = HashMap::new();
    scores.insert("Alice", 42);
    scores.insert("Bob", 69);
    scores.insert("Charlie", 87);

    // Iteriamo sui valori mutabili della HashMap e li modifichiamo
    for (_, score) in scores.iter_mut() {
        *score += 10;
    }

    // Stampiamo i nuovi punteggi
    for (name, score) in scores.iter() {
        println!("{}: {}", name, score);
    }
}
```



Entry<'a, K, V>

- Rust offre la possibilità di ottimizzare l'utilizzo delle mappe: in particolare attraverso il metodo **entry** che permette di cercare una chiave all'interno di una mappa e ritorna un enum in base al risultato della ricerca
 - **entry(&mut self, key: K) -> Entry<'a, K, V>**

```
pub enum Entry <a', K, V> {
    Occupied(OccupiedEntry <a'. K, V>),
    Vacant(VacantEntry <a'. K, V>),
}
```

- L'enum **Entry<'a, K, V>** a sua volta mette a disposizione diversi metodi per la gestione del risultato, permettendo di ridurre il numero di spostamenti in memoria
 - **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
 - **or_insert_with(self, f: F)**: Come or_insert, ma con funzione per calcolare il valore predefinito

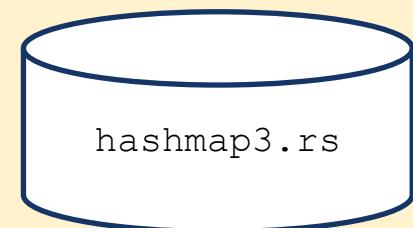
entry() or_insert()

```
fn main() {
    let mut scores = HashMap::new();

    // Inserimento di una coppia chiave-valore utilizzando il metodo entry
    scores.entry("Alice").or_insert(100);
    scores.entry("Bob").or_insert(90);
    scores.entry("Charlie").or_insert(80);

    // Aggiornamento del punteggio di Alice usando il metodo entry
    let alice_entry = scores.entry("Alice");
    match alice_entry {
        Entry::Occupied(mut entry) => {
            *entry.get_mut() += 10; // Aggiunge 10 al punteggio di Alice
        }
        Entry::Vacant(_) => {
            println!("Alice non trovata"); // Alice non è presente nella mappa
        }
    }

    // Stampa della HashMap aggiornata
    println!("HashMap: {:?}", scores);
}
```



entry() or_insert() and_modify()

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

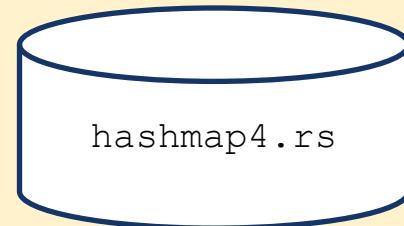
fn main() {
    let mut scores = HashMap::new();

    scores.insert("Team Blue", 10);
    scores.insert("Team Red", 20);

    // Incrementa il punteggio del team "Team Blue" se esiste,
    // altrimenti inserisci un nuovo punteggio
    scores.entry("Team Blue").and_modify(|score| *score += 5).or_insert(15);

    // Incrementa il punteggio del team "Team Green" se esiste,
    // altrimenti inserisci un nuovo punteggio
    scores.entry("Team Green").and_modify(|score| *score += 5).or_insert(15);

    println!("{}: {:?}", scores);
    // Stampa: {"Team Blue": 15, "Team Red": 20, "Team Green": 15}
}
```



`entry()` or `insert_with()` and `_modify()`

```
fn main() {
    let mut scores = HashMap::new();
    scores.insert("Team Blue", 10);
    scores.insert("Team Red", 20);

    // Incrementa il punteggio del team "Team Blue" se esiste,
    // altrimenti inserisci un nuovo punteggio
    scores.entry("Team Blue")
        .and_modify(|score| *score += 5)
        .or_insert(15);

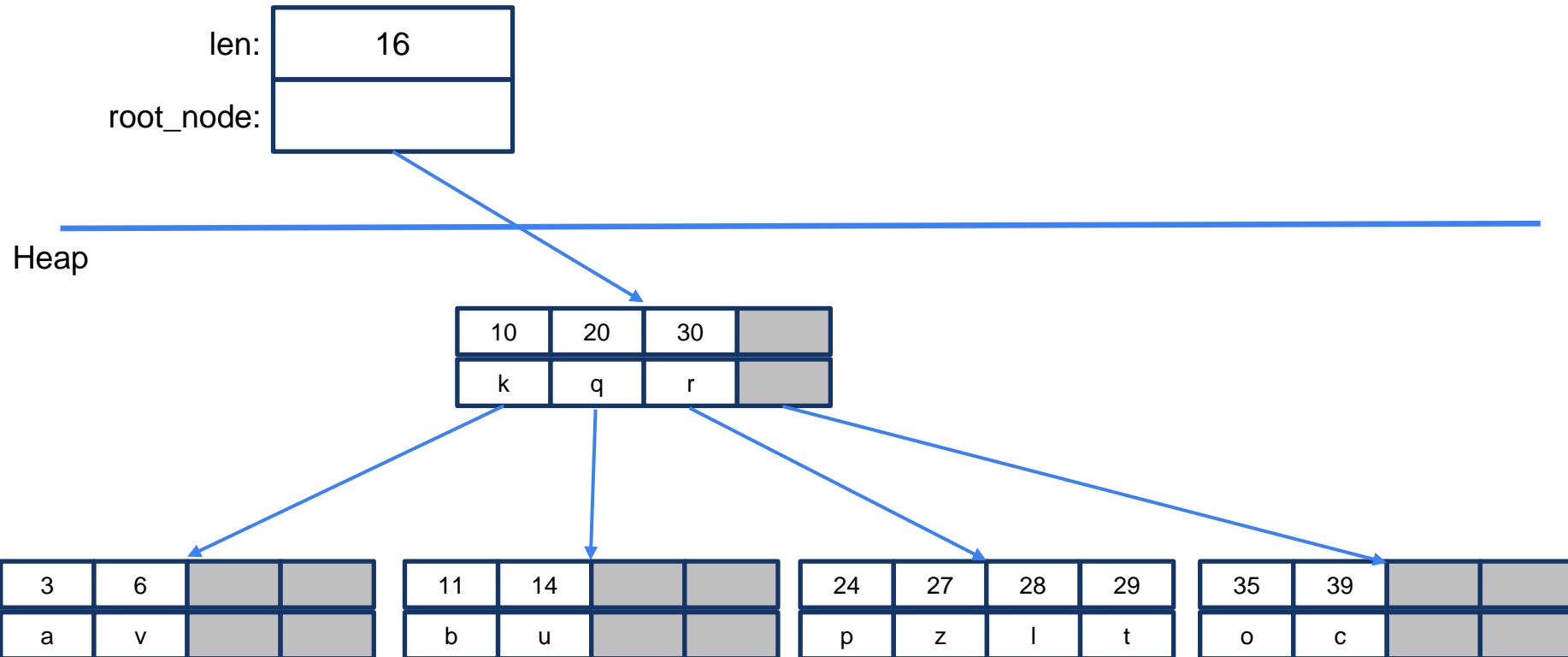
    // Calcola un punteggio predefinito per il team "Team Green" solo se non esiste già,
    // altrimenti utilizza il punteggio esistente
    scores.entry("Team Green")
        .and_modify(|score| *score += 5)
        .or_insert_with(|| calculate_default_score("Team Green"));

    println!("{}:", scores); // Stampa: {"Team Blue": 15, "Team Red": 20, "Team Green": 25}
}

// Funzione per calcolare il punteggio predefinito per un nuovo team
fn calculate_default_score(team: &str) -> i32 {
    // Supponiamo che il punteggio predefinito sia il doppio della lunghezza del nome del team
    (team.len() as i32) * 2
}
```



BTreeMap



BTreeMap<K,V>

- **new()**: Crea una nuova mappa vuota
- **with_capacity(capacity)**: Crea una nuova mappa con una capacità iniziale specificata
- **insert(key, value)**: Inserisce una coppia chiave-valore nella mappa
- **get(&key)**: Restituisce una referenza all'elemento associato alla chiave specificata, se presente
- **get_mut(&key)**: Restituisce una referenza mutabile all'elemento associato alla chiave specificata, se presente
- **contains_key(&key)**: Verifica se la mappa contiene la chiave specificata
- **remove(&key)**: Rimuove e restituisce l'elemento associato alla chiave specificata, se presente
- **len()**: Restituisce il numero di coppie chiave-valore nella mappa
- **is_empty()** : Restituisce true se la mappa è vuota, altrimenti false
- **clear()**: Rimuove tutte le coppie chiave-valore dalla mappa
- **iter()**: Restituisce un iteratore sugli elementi della mappa come coppie chiave-valore
- **iter_mut()**: Restituisce un iteratore mutabile sugli elementi della mappa come coppie chiave-valore
- **range(range)**: Restituisce un iteratore sugli elementi della mappa in un intervallo specificato di chiavi.
- **range_mut(range)**: Restituisce un iteratore mutabile sugli elementi della mappa in un intervallo specificato di chiavi
- **entry(&key)**: Restituisce un'entry della mappa per la chiave specificata, che permette di manipolare l'elemento associato in modo sicuro

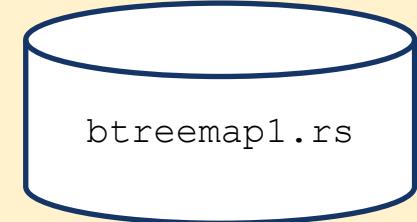
```
fn main() {
    let mut map = BTreeMap::new();
    map.insert(3, "tre");
    map.insert(1, "uno");
    map.insert(4, "quattro");
    map.insert(2, "due");
    map.insert(5, "cinque");

    println!("Mappa: {:?}", map);

    // Verifica se una chiave è presente nella mappa
    println!("La chiave 2 è presente nella mappa: {}", map.contains_key(&2));

    // Accesso all'elemento associato a una chiave
    if let Some(value) = map.get(&3) {
        println!("Valore associato alla chiave 3: {}", value);
    }

    // Rimozione di un elemento dalla mappa
    let removed_value = map.remove(&4);
    match removed_value {
        Some(value) => println!("Elemento rimosso: {}", value),
        None => println!("La chiave non esisteva nella mappa"),
    }
    // Iterazione sugli elementi della mappa
    println!("Iterazione sulla mappa:");
    for (key, value) in &map {
        println!("Chiave: {}, Valore: {}", key, value);
    }
}
```



range()

```
fn main() {  
    let mut map = BTreeMap::new();  
    map.insert(1, "uno");  
    map.insert(2, "due");  
    map.insert(3, "tre");  
    map.insert(4, "quattro");  
    map.insert(5, "cinque");  
    map.insert(6, "sei");  
    map.insert(7, "sette");  
    map.insert(8, "otto");  
    map.insert(9, "nove");  
    map.insert(10, "dieci");  
  
    // Utilizzo del metodo range per iterare su un intervallo specificato di chiavi  
    let mut range_iter = map.range(3..8); // Itera sulle chiavi da 3 a 7 (inclusi)  
  
    // Stampa degli elementi nell'intervallo specificato  
    println!("Elementi nell'intervallo da 3 a 7:");  
    while let Some((key, value)) = range_iter.next() {  
        println!("Chiave: {}, Valore: {}", key, value);  
    }  
}
```



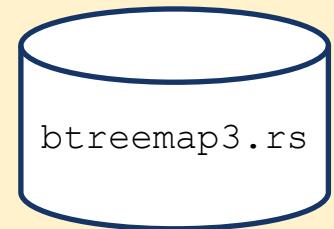
range_mut()

```
fn main() {
    // Creazione di una nuova BTreeMap con alcune coppie chiave-valore
    let mut map = BTreeMap::new();
    map.insert(1, "uno");
    map.insert(2, "due");
    map.insert(3, "tre");
    map.insert(4, "quattro");
    map.insert(5, "cinque");

    // Utilizzo del metodo range_mut per iterare mutabilmente
    // su un intervallo specificato di chiavi
    let mut range_iter = map.range_mut(2..=4);
    // Itera mutabilmente sulle chiavi da 2 a 4 (inclusi)

    // Modifica dei valori all'interno dell'intervallo specificato
    while let Some((_key, value)) = range_iter.next() {
        *value = "modificato";
    }

    // Stampa della mappa dopo le modifiche
    println!("Mappa dopo le modifiche: {:?}", map);
}
```



`entry()` e enum `Entry`

- Anche la BTreeMap implementa il metodo `entry` che permette di cercare una chiave all'interno di una mappa e ritorna un enum in base al risultato della ricerca `entry(&mut self, key: K) -> Entry<'a, K, V>`
- ma non implementa i metodi `and_modify()` e `or_insert()`
- Per modificare un valore o per inserirlo si usano i metodi `get_mut()` e `insert()`.

```
use std::collections::BTreeMap;
fn main() {
    let mut scores = BTreeMap::new();
    scores.insert("Alice", 42);
    scores.insert("Bob", 69);
    // Incrementiamo il punteggio di Mark di 5 punti, se esiste
    match scores.entry("Mark") {
        // Se la voce esiste, aggiungiamo 5 al punteggio esistente
        std::collections::btree_map::Entry::Occupied(mut entry) => {
            *entry.get_mut() += 5;
            println!("Il nuovo punteggio di Mark è: {}", entry.get());
        }
        // Se la voce non esiste, inseriamo una nuova voce con il punteggio 5
        std::collections::btree_map::Entry::Vacant(entry) => {
            entry.insert(5);
            println!("Abbiamo inserito un nuovo punteggio per Mark.");
        }
    }
    match scores.entry("Alice") {
        std::collections::btree_map::Entry::Occupied(mut entry) => {
            *entry.get_mut() += 10;
            println!("Il nuovo punteggio di Alice è: {}", entry.get());
        }
        std::collections::btree_map::Entry::Vacant(entry) => {
            entry.insert(10);
            println!("Abbiamo inserito un nuovo punteggio per Alice.");
        }
    }
    println!("Punteggi aggiornati:");
    for (name, score) in &scores {
        println!("{}: {}", name, score);
    }
}
```

entry()



Insiemi

- Un **HashSet<T>** è un insieme di elementi **univoci** di tipo **T** i valori sono salvati nello heap come una singola **hash table**
 - L'inserimento di una nuova entry nell' **HashSet<T>** può causare la riallocazione ed il movimento dei dati
 - Un **HashSet<T>** è implementato come un wrapper attorno al tipo **HashMap<T, ()>**
- Una **BTreeSet<T>** è un insieme di elementi **univoci** di tipo **T**, i valori sono salvati nello heap come un singolo **albero** dove ogni entry rappresenta un nodo
 - L'inserimento di una nuova entry nella **BtreeSet<T>** può causare la riallocazione ed il movimento dei dati

HashSet<T>

- **`new()`**: Crea un nuovo set vuoto
- **`insert(&mut self, value: T) -> bool`**: Inserisce un valore nel set. Restituisce true se il valore è stato inserito con successo (cioè non era già presente nel set), altrimenti restituisce false
- **`remove(&mut self, value: &T) -> bool`**: Rimuove un valore dal set. Restituisce true se il valore è stato rimosso con successo (cioè era presente nel set), altrimenti restituisce false
- **`contains(&self, value: &T) -> bool`**: Verifica se il set contiene un certo valore. Restituisce true se il valore è presente nel set, altrimenti restituisce false
- **`len(&self) -> usize`**: Restituisce il numero di elementi nel set
- **`is_empty(&self) -> bool`**: Restituisce true se il set è vuoto, altrimenti restituisce false
- **`clear(&mut self)`**: Rimuove tutti gli elementi dal set, lasciandolo vuoto
- **`iter(&self) -> Iter<T>`**: Restituisce un iteratore immutabile sui valori nel set
- **`iter_mut(&mut self) -> IterMut<T>`**: Restituisce un iteratore mutabile sui valori nel set
- **`get(value: &T) -> Option<&T>`**: Restituisce un riferimento all'elemento dell'insieme del valore specificato, se presente
- **`take(value: &T) -> Option<&T>`**: Elimina l'elemento dell'insieme del valore specificato, se presente

- **union(&self, other: &HashSet<T>) -> HashSet<T>**: Restituisce un nuovo set che contiene l'unione degli elementi del set corrente e di un altro set
- **intersection(&self, other: &HashSet<T>) -> HashSet<T>**: Restituisce un nuovo set che contiene l'intersezione degli elementi del set corrente e di un altro set
- **difference(&self, other: &HashSet<T>) -> HashSet<T>**: Restituisce un nuovo set che contiene gli elementi presenti nel set corrente ma non nell'altro set
- **symmetric_difference(&self, other: &HashSet<T>) -> HashSet<T>**: Restituisce un nuovo set che contiene gli elementi presenti solo in uno dei due set, ma non in entrambi
- **is_disjoint(&self, other: &HashSet<T>) -> bool**: Restituisce true se i due insiemi non hanno valori in comune
- **is_subset(&self, other: &HashSet<T>) -> bool**: restituisce true se tutti i valori dell'insieme self sono presenti in other
- **is_superset(&self, other: &HashSet<T>) -> bool**: restituisce true se tutti i valori dell'insieme other sono presenti in self

```
fn main() {  
    let mut numbers_set: HashSet<i32> = HashSet::new();  
    // Inseriamo alcuni numeri nel set  
    numbers_set.insert(1);  
    numbers_set.insert(2);  
    numbers_set.insert(3);  
    numbers_set.insert(4);  
  
    // Controlliamo se il set contiene un certo numero  
    println!("Il set contiene il numero 3? {}", numbers_set.contains(&3));  
    // Stampa: Il set contiene il numero 3? true  
    println!("Il set contiene il numero 5? {}", numbers_set.contains(&5));  
    // Stampa: Il set contiene il numero 5? false  
  
    println!("Numero di elementi nel set: {}", numbers_set.len());  
    // Stampa: Numero di elementi nel set: 4  
  
    println!("Il set è vuoto? {}", numbers_set.is_empty()); // Stampa: Il set è vuoto? false  
  
    numbers_set.remove(&4); // Rimuoviamo un numero dal set  
  
    // Iteriamo attraverso gli elementi del set e stampiamoli  
    println!("Elementi nel set:");  
    for number in &numbers_set {  
        println!("{}", number);  
    } // Stampa: 1, 2, 3  
  
    // Rimuoviamo tutti gli elementi dal set  
    numbers_set.clear();  
    println!("Numero di elementi nel set dopo la cancellazione: {}", numbers_set.len());  
}
```



```
fn main() {
    let mut numeri = HashSet::new();
    numeri.insert(1);
    numeri.insert(2);
    numeri.insert(3);

    // Usare get per verificare se un elemento è presente
    if numeri.get(&2).is_some() {
        println!("Il numero 2 è presente nel HashSet.");
    }

    // Usare take per rimuovere e restituire un elemento
    if let Some(numero) = numeri.take(&3) {
        println!("Il numero {} è stato rimosso dal HashSet.", numero);
    } else {
        println!("Il numero 3 non era presente");
    }

    // Verificare che il numero 3 sia stato rimosso
    if numeri.get(&3).is_none() {
        println!("Il numero 3 non è più presente nel HashSet.");
    }

    let vecchio_numero = 2;
    let nuovo_numero = 4;
    // Rimuovere il vecchio numero se presente e inserire il nuovo
    if numeri.remove(&vecchio_numero) {
        numeri.insert(nuovo_numero);
        println!("Il numero {} è stato sostituito con {} nel HashSet.", vecchio_numero, nuovo_numero);
    }
    println!("Contenuto finale del HashSet: {:?}", numeri);
}
```



```
fn main() {
    // Primo HashSet
    let set1: HashSet<i32> = [1, 2, 3, 4, 5].iter().cloned().collect();
    // Secondo HashSet
    let set2: HashSet<i32> = [3, 4, 5, 6, 7].iter().cloned().collect();

    // Union: Unione dei due HashSet
    let union: HashSet<_> = set1.union(&set2).cloned().collect();
    println!("Unione dei due set: {:?}", union); // Stampa: Unione dei due set: {1, 2, 3, 4, 5, 6, 7}

    // Intersection: Intersezione dei due HashSet
    let intersection: HashSet<_> = set1.intersection(&set2).cloned().collect();
    println!("Intersezione dei due set: {:?}", intersection);
    // Stampa: Intersezione dei due set: {3, 4, 5}

    // Difference: Elementi presenti solo in set1
    let difference1: HashSet<_> = set1.difference(&set2).cloned().collect();
    println!("Elementi presenti solo in set1: {:?}", difference1);
    // Stampa: Elementi presenti solo in set1: {1, 2}

    // Difference: Elementi presenti solo in set2
    let difference2: HashSet<_> = set2.difference(&set1).cloned().collect();
    println!("Elementi presenti solo in set2: {:?}", difference2);
    // Stampa: Elementi presenti solo in set2: {6, 7}

    // Symmetric Difference: Elementi presenti solo in uno dei due set
    let symmetric_difference: HashSet<_> = set1.symmetric_difference(&set2).cloned().collect();
    println!("Elementi presenti solo in uno dei due set: {:?}", symmetric_difference);
    // Stampa: Elementi presenti solo in uno dei due set: {1, 2, 6, 7}
}
```



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

fn main() {
    // Creiamo due HashSet di numeri interi
    let set1: HashSet<i32> = [3, 4, 5].iter().cloned().collect();
    let set2: HashSet<i32> = [6, 7].iter().cloned().collect();

    // Verifichiamo se i due set sono disgiunti
    if set1.is_disjoint(&set2) {
        println!("I due set sono disgiunti");
    } else {
        println!("I due set non sono disgiunti");
    }

    // Verifichiamo se set1 è un sottoinsieme di set2
    if set1.is_subset(&set2) {
        println!("Il set1 è un sottoinsieme di set2");
    } else {
        println!("Il set1 non è un sottoinsieme di set2");
    }

    // Verifichiamo se set2 è un sovrainsieme di set1
    if set2.is_superset(&set1) {
        println!("Il set2 è un sovrainsieme di set1");
    } else {
        println!("Il set2 non è un sovrainsieme di set1");
    }
}
```



hashset4.rs

BTreeSet<T>

- `new()`: Crea un nuovo set vuoto
- `insert(&mut self, value: T) -> bool`: Inserisce un valore nel set. Restituisce true se il valore è stato inserito con successo (cioè non era già presente nel set), altrimenti restituisce false
- `remove(&mut self, value: &T) -> bool`: Rimuove un valore dal set. Restituisce true se il valore è stato rimosso con successo (cioè era presente nel set), altrimenti restituisce false
- `contains(&self, value: &T) -> bool`: Verifica se il set contiene un certo valore. Restituisce true se il valore è presente nel set, altrimenti restituisce false
- `len(&self) -> usize`: Restituisce il numero di elementi nel set
- `is_empty(&self) -> bool`: Restituisce true se il set è vuoto, altrimenti restituisce false
- `clear(&mut self)`: Rimuove tutti gli elementi dal set, lasciandolo vuoto
- `iter(&self) -> Iter<T>`: Restituisce un iteratore immutabile sui valori nel set
- `iter_mut(&mut self) -> IterMut<T>`: Restituisce un iteratore mutabile sui valori nel set
- `range(& self, range: Q)`: Restituisce un iteratore sugli elementi compresi nell'intervallo specificato

- **union(&self, other: &HashSet<T>) -> HashSet<T>**: Restituisce un nuovo set che contiene l'unione degli elementi del set corrente e di un altro set
- **intersection(&self, other: &HashSet<T>) -> HashSet<T>**: Restituisce un nuovo set che contiene l'intersezione degli elementi del set corrente e di un altro set
- **difference(&self, other: &HashSet<T>) -> HashSet<T>**: Restituisce un nuovo set che contiene gli elementi presenti nel set corrente ma non nell'altro set
- **symmetric_difference(&self, other: &HashSet<T>) -> HashSet<T>**: Restituisce un nuovo set che contiene gli elementi presenti solo in uno dei due set, ma non in entrambi
- **is_disjoint(&self, other: &HashSet<T>) -> bool**: Restituisce true se i due insiemi non hanno valori in comune
- **is_subset(&self, other: &HashSet<T>) -> bool**: restituisce true se tutti i valori dell'insieme self sono presenti in other
- **is_superset(&self, other: &HashSet<T>) -> bool**: restituisce true se tutti i valori dell'insieme other sono presenti in self

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

fn main() {
    let mut set: BTreeSet<i32> = BTreeSet::new();
    // Inserisci elementi nel set
    set.insert(10);
    set.insert(20);
    set.insert(30);

    // Stampa il set per verificare l'inserimento
    println!("{:?}", set);

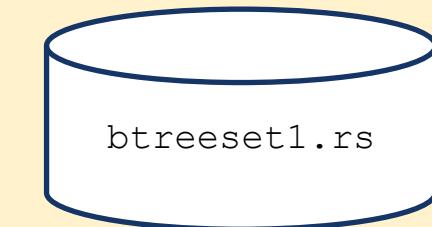
    // Rimuovi un elemento dal set
    set.remove(&20);

    // Stampa il set per verificare la rimozione
    println!("{:?}", set);

    // Controlla se un elemento è presente nel set
    println!("Is 30 in the set? {}", set.contains(&30));

    // Inizializza un iteratore sul set
    let mut iterator = set.iter();

    // Stampa gli elementi del set attraverso l'iteratore
    println!("Iterating over the set:");
    while let Some(element) = iterator.next() {
        println!("{}", element);
    }
}
```



```
use std::collections::BTreeSet;
use std::ops::Bound::{Included, Excluded};

fn main() {
    // Creare un BTreeSet con alcuni numeri
    let mut numeri: BTreeSet<i32> = BTreeSet::new();
    numeri.insert(5);
    numeri.insert(10);
    numeri.insert(15);
    numeri.insert(18);
    numeri.insert(20);
    numeri.insert(25);

    // Usare il metodo range per ottenere un iteratore su un intervallo di valori
    // In questo caso, tutti i numeri maggiori o uguali a 10 e minori di 20
    for numero in numeri.range(Included(10), Excluded(20)) {
        println!("Numero nell'intervallo: {}", numero);
    }
}
```

btreeset2.rs

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

fn main() {
    // Creare alcuni BTreeSet di esempio
    let set_a: BTreeSet<i32> = vec![1, 2, 3].into_iter().collect();
    let set_b: BTreeSet<i32> = vec![2, 3, 4].into_iter().collect();

    // Calcolare l'unione di set_a e set_b
    let union_result: BTreeSet<_> = set_a.union(&set_b).cloned().collect();
    println!("Unione: {:?}", union_result);

    // Calcolare l'intersezione di set_a e set_b
    let intersection_result: BTreeSet<_> = set_a.intersection(&set_b).cloned().collect();
    println!("Intersezione: {:?}", intersection_result);

    // Calcolare la differenza tra set_a e set_b
    let difference_result: BTreeSet<_> = set_a.difference(&set_b).cloned().collect();
    println!("Differenza: {:?}", difference_result);
}
```



btreeset3.rs

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

fn main() {
    // Creiamo due HashSet di numeri interi
    let set1: BTreeSet<i32> = [3, 4, 5].iter().cloned().collect();
    let set2: BTreeSet<i32> = [3,4,5,9].iter().cloned().collect();

    // Verifichiamo se i due set sono disgiunti
    if set1.is_disjoint(&set2) {
        println!("I due set sono disgiunti");
    } else {
        println!("I due set non sono disgiunti");
    }

    // Verifichiamo se set1 è un sottoinsieme di set2
    if set1.is_subset(&set2) {
        println!("Il set1 è un sottoinsieme di set2");
    } else {
        println!("Il set1 non è un sottoinsieme di set2");
    }

    // Verifichiamo se set2 è un sovrainsieme di set1
    if set2.is_superset(&set1) {
        println!("Il set2 è un sovrainsieme di set1");
    } else {
        println!("Il set2 non è un sovrainsieme di set1");
    }
}
```

btreeiset4.rs

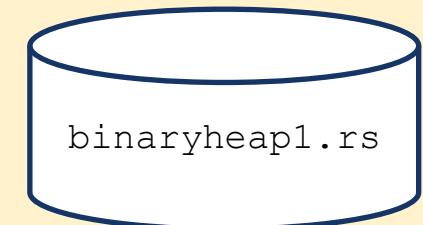
BinaryHeap<T>

- Una **BinaryHeap<T>** è una collezione di elementi di tipo **T**, i valori sono salvati nello heap e l'elemento più grande si trova sempre nel fronte della struttura dati
 - Il tipo **T** deve implementare il tratto **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 $O(\log(n))$

- **new()**: Crea una nuova heap vuota
- **from(data)**: Crea una heap a partire da un iterabile di dati
- **push(value)**: Inserisce un elemento nella heap
- **pop()**: Rimuove e restituisce l'elemento massimo dalla heap
- **peek()**: Restituisce un riferimento all'elemento massimo senza rimuoverlo
- **peek_mut()**: fornisce un riferimento mutabile all'elemento massimo
- **len()**: Restituisce il numero di elementi nella heap
- **is_empty()**: Restituisce true se la heap è vuota, altrimenti false
- **clear()**: Rimuove tutti gli elementi dalla heap
- **into_sorted_vec()**: Consuma la heap e restituisce un vettore ordinato degli elementi
- **clone()**: Crea una copia della heap
- **iter()**: Restituisce un iteratore che permette di iterare sugli elementi della heap
- **iter_mut()**: Restituisce un iteratore mutabile che permette di iterare sugli elementi della heap e modificarli
- **entry(&key)**: Restituisce un'entry della mappa per la chiave specificata, che permette di manipolare l'elemento associato in modo sicuro
- **or_insert(key, default)**: Inserisce una coppia chiave-valore nella mappa se la chiave non è presente, restituendo un riferimento all'elemento esistente o al valore appena inserito
- **or_insert_with(key, default_fn)**: Come or_insert, ma con funzione per calcolare il valore predefinito

```
fn main() {
    let mut heap = BinaryHeap::from(vec![4, 10, 8, 3, 7]);
    heap.push(1);
    heap.push(15);

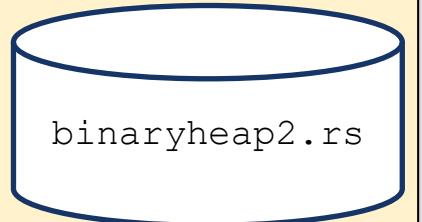
    // Stampa della BinaryHeap (senza un ordine particolare)
    println!("BinaryHeap: {:?}", heap);
    // Accesso al massimo elemento (senza rimuoverlo)
    if let Some(max) = heap.peek() {
        println!("Massimo elemento: {}", max);
    } else {
        println!("La BinaryHeap è vuota");
    }
    // Accesso al massimo elemento (senza rimuoverlo) per modificarlo
    if let Some(mut max) = heap.peek_mut() {
        println!("Cambio il massimo elemento: da {} a {}", *max, *max/2);
        *max = *max/2;
    } else {println!("La BinaryHeap è vuota"); }
    // Rimozione del massimo elemento
    if let Some(max) = heap.pop() {
        println!("Elemento rimosso: {}", max);
    } else {
        println!("La BinaryHeap è vuota");
    }
    println!("BinaryHeap dopo la rimozione: {:?}", heap);
}
```



```
fn main() {
    let mut max_heap = BinaryHeap::new();
    max_heap.push(4);
    max_heap.push(2);
    max_heap.push(5);
    max_heap.push(1);
    max_heap.push(7);
    max_heap.push(6);

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

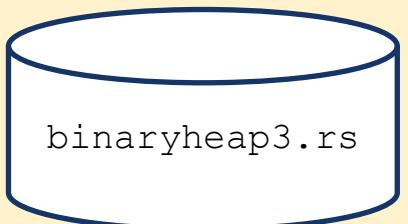
    // Scandire il BinaryHeap senza rimuovere i valori
    for value in max_heap.iter() {
        println!("Value: {}", value);
    }
}
```



binaryheap2.rs

```
fn main() {
    let mut max_heap = BinaryHeap::new();
    max_heap.push(4);
    max_heap.push(2);
    max_heap.push(5);
    max_heap.push(1);
    max_heap.push(7);
    max_heap.push(6);

    while let Some(value) = max_heap.pop() {
        println!("Value: {}", value);
    }
}
```



into_sorted_vec()

```
fn main() {
    // Creazione di una BinaryHeap con alcuni valori non ordinati
    let mut heap = BinaryHeap::from(vec![10, 5, 8, 3, 12]);

    // Chiamata del metodo into_sorted_vec per ottenere un vettore ordinato
    let sorted_vec = heap.into_sorted_vec();

    // Stampa del vettore ordinato
    println!("Vettore ordinato: {:?}", sorted_vec);
}
```



Per saperne di più

- Rust Collections
 - <https://medium.com/@tzutoo/rust-collections-56359d50df28>
 - Presentazione dettagliata delle diverse strutture dati offerte da Rust per gestire collezioni di dati, corredate di consigli operativi per la creazione di algoritmi corretti ed efficienti
- The Rust Programming Language: Chapter 8 — Common Collections
 - <https://doc.rust-lang.org/book/ch08-00-common-collections.html>
 - Tutorial del linguaggio con esempi pratici sull'uso di Vec<T> e Map<K,V>
- Module std::collections
 - <https://doc.rust-lang.org/std/collections/index.html>
 - Documentazione ufficiale delle classi con dettagli sul loro utilizzo