



Collezioni di dati

Algoritmi e contenitori

Collezioni di dati

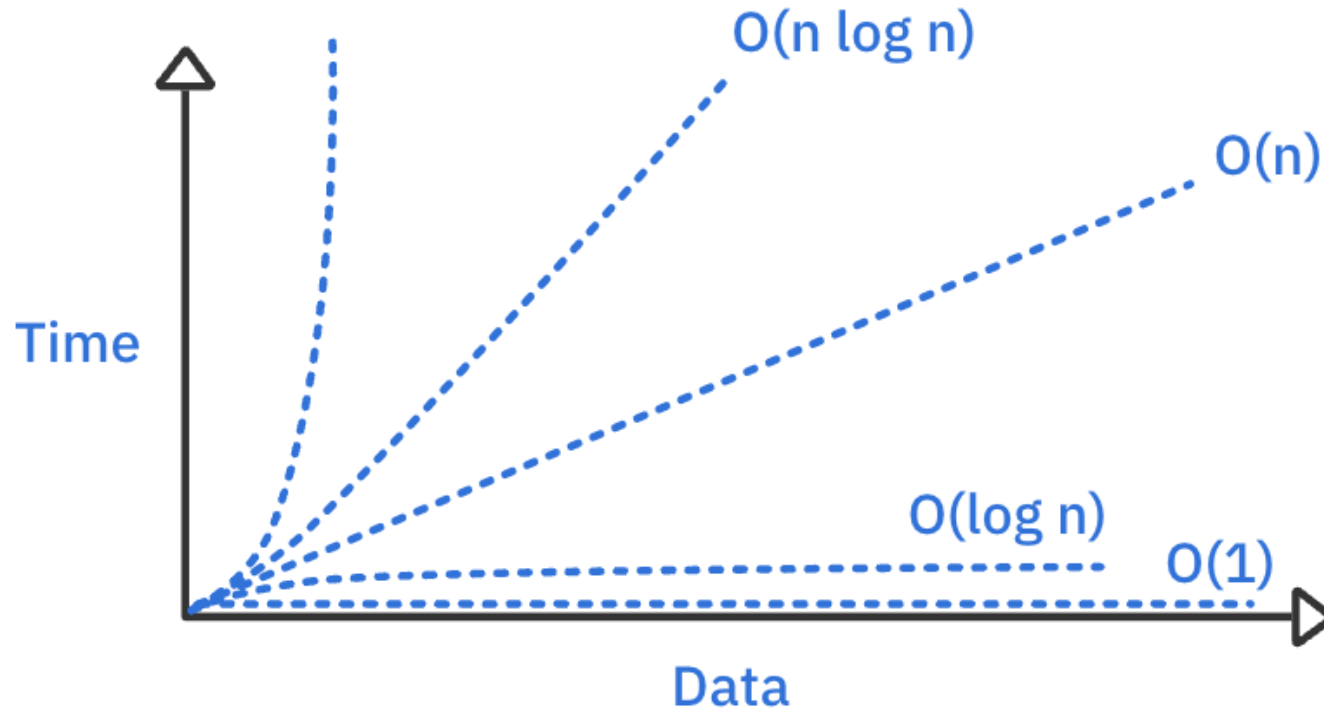
- Tutti i linguaggi offrono, nella propria libreria standard, un insieme di strutture dati volte a semplificare la vita ai programmatori implementando quelli che sono i migliori algoritmi noti per gestire problemi comuni
 - Liste ordinate
 - Insiemi di elementi univoci
 - Mappe chiave-valore
- Se esistono 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> <small>*esiste anche collegata solo in avanti (forward_list)</small>	<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
- 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

Vec<T>

- 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>

- 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>**
- 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()**

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

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
 - 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
 - La chiave deve essere univoca ed il tipo **K** deve implementare il tratto **Ord**

Mappe

- Poiché una mappa nella maggior parte dei casi possiede sia le chiavi che i valori presenti al suo interno, le operazioni relative all'inserimento, cancellazione, accesso ai singoli elementi richiedono il rispetto di tale possesso
 - Inoltre, Rust cerca di ottimizzare l'accesso ai dati contenuti evitando di dover navigare più volte la struttura interna alla ricerca del valore associato ad un chiave
- La conseguenza di ciò è che l'interfaccia applicativa offerta è alquanto differente da quella tipicamente offerta da altri linguaggi, pur avendo lo stesso potere espressivo
 - In particolare, se il tipo della chiave è **K** e il tipo del valore è **V**, i metodi di accesso (**get(...)**, **get_key_value(...)**, **keys(...)**, ...) usano chiavi di tipo **&K** e restituiscono valori di tipo **&V**
 - Anche i metodi che modificano la mappa tendono ad usare riferimenti a chiavi e valori (**remove(...)**, **retain(...)**)
 - I metodi di inserimento richiedono invece il movimento di valori posseduti
 - La sostituzione di un valore avviene presente nella mappa ha un approccio particolare, attraverso il concetto di **Entry**

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>**
- L'enum **Entry<'a, K, V>** 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

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

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

Operare con le mappe

```
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

- 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
 - Un **BTreeSet<T>** è implementato come un wrapper attorno al tipo **BTreeMap<T, ()>**

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 nella prima posizione
 - 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 $O(\log(n))$



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