Data Structures

Edoardo Morassutto e Dario Petrillo

Volterra, 18 novembre 2022

Introduzione

Cos'è una struttura dati

Un contenitore per dei dati, che supporta delle operazioni con requisiti e complessità.

Cos'è una struttura dati

Un contenitore per dei dati, che supporta delle operazioni con requisiti e complessità.
La complessità spesso dipende dal numero di elementi all'interno della struttura. Indichiamo con N la dimensione della struttura.

Esempio di struttura dati

`std::vector<T>`

Esempio di struttura dati

`std::vector<T>`

Cosa sappiamo

- È un contenitore di N elementi di tipo T.

Esempio di struttura dati

`std::vector<T>`

Cosa sappiamo

- È un contenitore di N elementi di tipo T.
- T non ha alcun requisito (non serve che sia ordinabile, ...)

Esempio di struttura dati

`std::vector<T>`

Cosa sappiamo

- È un contenitore di N elementi di tipo T.
- T non ha alcun requisito (non serve che sia ordinabile, ...)
- Supporta le operazioni:
  - Leggi l'i-esimo elemento in $\mathcal{O}(1)$.

Esempio di struttura dati

```
std::vector<T>
```

Cosa sappiamo

- È un contenitore di N elementi di tipo T.
- T non ha alcun requisito (non serve che sia ordinabile, . . .)
- Supporta le operazioni:
  - Leggi l'i-esimo elemento in $\mathcal{O}(1)$.
  - Modifica l'i-esimo elemento in $\mathcal{O}(1)$.

Esempio di struttura dati

`std::vector<T>`

Cosa sappiamo

- È un contenitore di N elementi di tipo T.
- T non ha alcun requisito (non serve che sia ordinabile, ...)
- Supporta le operazioni:
  - Leggi l'i-esimo elemento in $\mathcal{O}(1)$.
  - Modifica l'i-esimo elemento in $\mathcal{O}(1)$.
  - Scorrere gli elementi in $\mathcal{O}(N)$.

Esempio di struttura dati

`std::vector<T>`

Cosa sappiamo

- È un contenitore di N elementi di tipo T.
- T non ha alcun requisito (non serve che sia ordinabile, ...)
- Supporta le operazioni:
  - Leggi l'i-esimo elemento in $\mathcal{O}(1)$.
  - Modifica l'i-esimo elemento in $\mathcal{O}(1)$.
  - Scorrere gli elementi in $\mathcal{O}(N)$.
  - Cancellare un elemento in $\mathcal{O}(1)$ senza mantenere l'ordine.

Esempi di strutture dati

Presenti nell'STL

- `std::vector<T>`

Esempi di strutture dati

Presenti nell'STL

- `std::vector<T>`
- `std::deque<T>`, `std::queue<T>`, `std::priority_queue<T>`

Esempi di strutture dati

Presenti nell'STL

- `std::vector<T>`
- `std::deque<T>`, `std::queue<T>`, `std::priority_queue<T>`
- `std::set<T>` e `std::map<K, V>`

Esempi di strutture dati

Presenti nell'STL

- `std::vector<T>`
- `std::deque<T>`, `std::queue<T>`, `std::priority_queue<T>`
- `std::set<T>` e `std::map<K, V>`
- `std::unordered_set<T>` e `std::unordered_map<K, V>`

Esempi di strutture dati

Presenti nell'STL

- `std::vector<T>`
- `std::deque<T>`, `std::queue<T>`, `std::priority_queue<T>`
- `std::set<T>` e `std::map<K, V>`
- `std::unordered_set<T>` e `std::unordered_map<K, V>`
- `std::multiset<T>` e `std::multimap<K, V>`

Esempi di strutture dati

Presenti nell'STL

- `std::vector<T>`
- `std::deque<T>`, `std::queue<T>`, `std::priority_queue<T>`
- `std::set<T>` e `std::map<K, V>`
- `std::unordered_set<T>` e `std::unordered_map<K, V>`
- `std::multiset<T>` e `std::multimap<K, V>`
- `std::unordered_multiset<T>` e `std::unordered_multimap<K, V>`

Esempi di strutture dati

Presenti nell'STL

- `std::vector<T>`
- `std::deque<T>`, `std::queue<T>`, `std::priority_queue<T>`
- `std::set<T>` e `std::map<K, V>`
- `std::unordered_set<T>` e `std::unordered_map<K, V>`
- `std::multiset<T>` e `std::multimap<K, V>`
- `std::unordered_multiset<T>` e `std::unordered_multimap<K, V>`
- `std::pair<A, B>` e `std::tuple<A, B, C, ...>`

Esempi di strutture dati

Presenti nell'STL

- `std::vector<T>`
- `std::deque<T>`, `std::queue<T>`, `std::priority_queue<T>`
- `std::set<T>` e `std::map<K, V>`
- `std::unordered_set<T>` e `std::unordered_map<K, V>`
- `std::multiset<T>` e `std::multimap<K, V>`
- `std::unordered_multiset<T>` e `std::unordered_multimap<K, V>`
- `std::pair<A, B>` e `std::tuple<A, B, C, ...>`

Non presenti nell'STL

- Segment tree

Esempi di strutture dati

Presenti nell'STL

- `std::vector<T>`
- `std::deque<T>`, `std::queue<T>`, `std::priority_queue<T>`
- `std::set<T>` e `std::map<K, V>`
- `std::unordered_set<T>` e `std::unordered_map<K, V>`
- `std::multiset<T>` e `std::multimap<K, V>`
- `std::unordered_multiset<T>` e `std::unordered_multimap<K, V>`
- `std::pair<A, B>` e `std::tuple<A, B, C, ...>`

Non presenti nell'STL

- Segment tree
- Fenwick tree

Esempi di strutture dati

Presenti nell'STL

- std::vector<T>
- std::deque<T>, std::queue<T>, std::priority_queue<T>
- std::set<T> e std::map<K, V>
- std::unordered_set<T> e std::unordered_map<K, V>
- std::multiset<T> e std::multimap<K, V>
- std::unordered_multiset<T> e std::unordered_multimap<K, V>
- std::pair<A, B> e std::tuple<A, B, C, ...>

Non presenti nell'STL

- Segment tree
- Fenwick tree
- Sparse tables

Esempi di strutture dati

Presenti nell'STL

- `std::vector<T>`
- `std::deque<T>`, `std::queue<T>`, `std::priority_queue<T>`
- `std::set<T>` e `std::map<K, V>`
- `std::unordered_set<T>` e `std::unordered_map<K, V>`
- `std::multiset<T>` e `std::multimap<K, V>`
- `std::unordered_multiset<T>` e `std::unordered_multimap<K, V>`
- `std::pair<A, B>` e `std::tuple<A, B, C, ...>`

Non presenti nell'STL

- Segment tree
- Fenwick tree
- Sparse tables
- DSU (Disjoint Set Union)

Esempi di strutture dati

Presenti nell'STL

- `std::vector<T>`
- `std::deque<T>`, `std::queue<T>`, `std::priority_queue<T>`
- `std::set<T>` e `std::map<K, V>`
- `std::unordered_set<T>` e `std::unordered_map<K, V>`
- `std::multiset<T>` e `std::multimap<K, V>`
- `std::unordered_multiset<T>` e `std::unordered_multimap<K, V>`
- `std::pair<A, B>` e `std::tuple<A, B, C, ...>`

Non presenti nell'STL

- Segment tree
- Fenwick tree
- Sparse tables
- DSU (Disjoint Set Union)
- Minqueue

Esempi di strutture dati

Presenti nell'STL

- `std::vector<T>`
- `std::deque<T>`, `std::queue<T>`, `std::priority_queue<T>`
- `std::set<T>` e `std::map<K, V>`
- `std::unordered_set<T>` e `std::unordered_map<K, V>`
- `std::multiset<T>` e `std::multimap<K, V>`
- `std::unordered_multiset<T>` e `std::unordered_multimap<K, V>`
- `std::pair<A, B>` e `std::tuple<A, B, C, ...>`

Non presenti nell'STL

- Segment tree
- Fenwick tree
- Sparse tables
- DSU (Disjoint Set Union)
- Minqueue
- Treap

Esempi di strutture dati

Presenti nell'STL

- `std::vector<T>`
- `std::deque<T>`, `std::queue<T>`, `std::priority_queue<T>`
- `std::set<T>` e `std::map<K, V>`
- `std::unordered_set<T>` e `std::unordered_map<K, V>`
- `std::multiset<T>` e `std::multimap<K, V>`
- `std::unordered_multiset<T>` e `std::unordered_multimap<K, V>`
- `std::pair<A, B>` e `std::tuple<A, B, C, ...>`

Non presenti nell'STL

- Segment tree
- Fenwick tree
- Sparse tables
- DSU (Disjoint Set Union)
- Minqueue
- Treap
- Skip list

Esempi di strutture dati

Presenti nell'STL

- `std::vector<T>`
- `std::deque<T>`, `std::queue<T>`, `std::priority_queue<T>`
- `std::set<T>` e `std::map<K, V>`
- `std::unordered_set<T>` e `std::unordered_map<K, V>`
- `std::multiset<T>` e `std::multimap<K, V>`
- `std::unordered_multiset<T>` e `std::unordered_multimap<K, V>`
- `std::pair<A, B>` e `std::tuple<A, B, C, ...>`

Non presenti nell'STL

- Segment tree
- Fenwick tree
- Sparse tables
- DSU (Disjoint Set Union)
- Minqueue
- Treap
- Skip list
- ...tante altre!

`std::vector<T>`

Alcuni trick:

- `std::vector<int> v(N, 123)` crea un vettore di N elementi, tutti inizializzati a 123.

`std::vector<T>`

Alcuni trick:

- `std::vector<int> v(N, 123)` crea un vettore di N elementi, tutti inizializzati a 123.
- `vec.push_back()` + `vec.pop_back()` + `vec.back()` = stack

`std::vector<T>`

Alcuni trick:

- `std::vector<int> v(N, 123)` crea un vettore di N elementi, tutti inizializzati a 123.
- `vec.push_back()` + `vec.pop_back()` + `vec.back()` = stack
- `sort(v.rbegin(), v.rend())` ordina il vettore in modo decrescente

`std::vector<T>`

Alcuni trick:

- `std::vector<int> v(N, 123)` crea un vettore di N elementi, tutti inizializzati a 123.
- `vec.push_back()` + `vec.pop_back()` + `vec.back()` = stack
- `sort(v.rbegin(), v.rend())` ordina il vettore in modo decrescente
- `vector<int> v = {1, 2, 3, 4}`

`std::vector<T>`

Alcuni trick:

- `std::vector<int> v(N, 123)` crea un vettore di N elementi, tutti inizializzati a 123.
- `vec.push_back()` + `vec.pop_back()` + `vec.back()` = stack
- `sort(v.rbegin(), v.rend())` ordina il vettore in modo decrescente
- `vector<int> v = {1, 2, 3, 4}`
- `vector<pair<int, int>> v`;
  `v.push_back({1, 2});`

std::set<T>

È un insieme di N elementi ordinato e senza duplicati.

`std::set<T>`

È un insieme di N elementi ordinato e senza duplicati.

- Esempi di applicazioni:
  - Hai un grafo memorizzato come liste di adiacenza e puoi aggiungere e togliere archi.

`std::set<T>`

È un insieme di N elementi ordinato e senza duplicati.

- Esempi di applicazioni:
  - Hai un grafo memorizzato come liste di adiacenza e puoi aggiungere e togliere archi.
  - `add(x)` aggiunge x all'insieme
    `remove(x)` rimuove x dall'insieme
    `closest(x)` trova l'elemento più vicino ad x

std::set<T>

È un insieme di N elementi ordinato e senza duplicati.

- Esempi di applicazioni:
  - Hai un grafo memorizzato come liste di adiacenza e puoi aggiungere e togliere archi.
  - `add(x)` aggiunge x all'insieme
    `remove(x)` rimuove x dall'insieme
    `closest(x)` trova l'elemento più vicino ad x
- **Requisito**: T deve essere ordinabile.
  - `int`, `std::pair`, `std::tuple`, ... sono ordinabili.
  - Le `struct` hanno bisogno di `bool operator<(const T& other) const`.

std::set<T>

Operazione	Complessità
`set.insert(x)`	$\mathcal{O}(\log N)$
`set.emplace(x)`	$\mathcal{O}(\log N)$
`set.erase(x)`	$\mathcal{O}(\log N)$
`set.find(x)`	$\mathcal{O}(\log N)$
`set.lower_bound(x)`	$\mathcal{O}(\log N)$
`set.upper_bound(x)`	$\mathcal{O}(\log N)$
`set.count(x)`	$\mathcal{O}(\log N)$
`set.begin()`	$\mathcal{O}(1)$ (trova minimo)
`set.rbegin()`	$\mathcal{O}(1)$ (trova massimo)
`for (auto x : set) {}`	$\mathcal{O}(N)$ (costante alta)

Come usare un comparatore custom (è nella doc di C++):

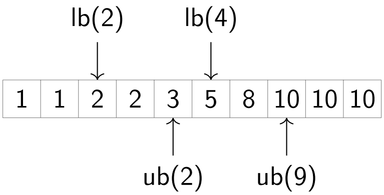
```
struct cmp {
    bool operator()(const T& a, const T& b) const { return a < b; }
};
std::set<T, cmp> s;
```

`lower_bound(x)` e `upper_bound(x)`

`lower_bound(x)`	Primo elemento maggiore o uguale a x
`upper_bound(x)`	Primo elemento maggiore di x

lower_bound(x) e upper_bound(x)

`lower_bound(x)`	Primo elemento maggiore o uguale a x
`upper_bound(x)`	Primo elemento maggiore di x



Nota nei `std::set` non ci sono duplicati, ma esiste `std::lower_bound(v.begin(), v.end(), x)` quando `v` è ordinato che funziona anche nei `std::vector` con dei duplicati.

`std::map<K, V>`

Una collezione di coppie *chiave-valore*, con chiavi ordinate e senza duplicati.

`std::map<K, V>`

Una collezione di coppie *chiave-valore*, con chiavi ordinate e senza duplicati.

- Esempi di applicazioni:
  - Compressione dell'input:
    N numeri da 0 a 10^{18} rimappati in N numeri da 0 a $N-1$
    N stringhe rimappate in N numeri da 0 a $N-1$

std::map<K, V>

Una collezione di coppie *chiave-valore*, con chiavi ordinate e senza duplicati.

- Esempi di applicazioni:
  - Compressione dell'input:
    N numeri da 0 a 10^{18} rimappati in N numeri da 0 a $N-1$
    N stringhe rimappate in N numeri da 0 a $N-1$
- **Requisito**: K deve essere ordinabile.

`std::map<K, V>`

Una collezione di coppie *chiave-valore*, con chiavi ordinate e senza duplicati.

- Esempi di applicazioni:
  - Compressione dell'input:
    N numeri da 0 a 10^{18} rimappati in N numeri da 0 a $N - 1$
    N stringhe rimappate in N numeri da 0 a $N - 1$
- **Requisito**: K deve essere ordinabile.

Attenzione!

`if (map[key] == 0)` crea un nuovo elemento nella `map` se non esiste!

Si può usare `map.find(key)` o `map.count(key)` per controllare se esiste.

`std::unordered_set` e `std::unordered_map`

Mantengono insiemi non ordinati e senza duplicati dove le operazioni costano $\mathcal{O}(1)$ invece che $\mathcal{O}(\log N)$.

Questa struttura dati è nota anche come hash table. Internamente converte le chiavi in numeri (il loro *hash*) e li usa per accedere velocemente ai dati.

`std::unordered_set` e `std::unordered_map`

Mantengono insiemi non ordinati e senza duplicati dove le operazioni costano $\mathcal{O}(1)$ invece che $\mathcal{O}(\log N)$.

Questa struttura dati è nota anche come hash table. Internamente converte le chiavi in numeri (il loro *hash*) e li usa per accedere velocemente ai dati.

Per i tipi di base (`int`, `long long`, `std::string`, ...) la funzione di hash è predefinita e possono essere usati.
Per tipi complessi (`std::pair`, `std::vector`, `std::map`, `struct` custom) va definito un *hasher*.

Segment Tree

Esempio di problema

Problema (Somma di intervalli)

Dato un array di A di N interi vogliamo supportare le seguenti operazioni:

- `update(i, x)` *imposta* `A[i] = x`.
- `somma(l, r)` *trova* $A[l] + A[l+1] + \ldots + A[r-1]$ *(l incluso, r escluso).*

Esempio di problema

Problema (Somma di intervalli)

Dato un array di A di N interi vogliamo supportare le seguenti operazioni:

- `update(i, x)` *imposta* `A[i] = x`.
- `somma(l, r)` *trova* $A[l] + A[l+1] + \ldots + A[r-1]$ *(*l *incluso,* r *escluso).*

Soluzione banale

Uso un `std::vector`.

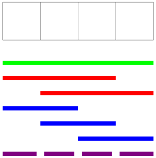
- Update in $\mathcal{O}(1)$.
- Query in $\mathcal{O}(r-l) = \mathcal{O}(N)$.

Ragioniamo sulle query

Domanda Quante sono le possibili query distinte? Oppure, quanti sono i possibili intervalli?

Ragioniamo sulle query

Domanda Quante sono le possibili query distinte? Oppure, quanti sono i possibili intervalli?



$$\frac{N(N+1)}{2} = \mathcal{O}(N^2)$$

Ragioniamo sulle query

Idea memorizzo la risposta per ogni possibile intervallo.

Ragioniamo sulle query

Idea memorizzo la risposta per ogni possibile intervallo.

Non funziona perché:

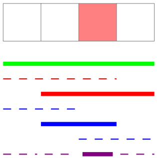
- Gli intervalli sono tanti: $\mathcal{O}(N^2)$.

Ragioniamo sulle query

Idea memorizzo la risposta per ogni possibile intervallo.

Non funziona perché:

- Gli intervalli sono tanti: $\mathcal{O}(N^2)$.
- Un update modifica tanti intervalli: $\mathcal{O}(N)$.



Ragioniamo sulle query

Idea non memorizzo tutti gli intervalli, ma solo alcuni. Lo posso fare se riesco a ricostuire la soluzione *unendo* due intervalli disgiunti.

```
somma(l, r) = somma(l, m) + somma(m, r)
```

Quali intervalli memorizzo?

- Tutti gli intervalli di lunghezza 1:

Ragioniamo sulle query

Idea non memorizzo tutti gli intervalli, ma solo alcuni. Lo posso fare se riesco a ricostuire la soluzione *unendo* due intervalli disgiunti.

`somma(l, r) = somma(l, m) + somma(m, r)`

Quali intervalli memorizzo?

- Tutti gli intervalli di lunghezza 1: `somma(l, r)` unisce $r - l$ intervalli, nel caso peggiore $\mathcal{O}(N)$.

Ragioniamo sulle query

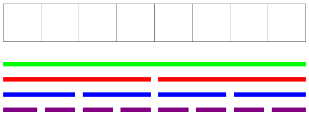
Idea non memorizzo tutti gli intervalli, ma solo alcuni. Lo posso fare se riesco a ricostuire la soluzione *unendo* due intervalli disgiunti.

`somma(l, r) = somma(l, m) + somma(m, r)`

Quali intervalli memorizzo?

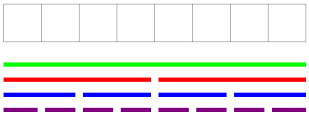
- Tutti gli intervalli di lunghezza 1: `somma(l, r)` unisce $r - l$ intervalli, nel caso peggiore $\mathcal{O}(N)$.
- Memorizzo solo gli intervalli di lunghezza *potenza di 2*.

Ragioniamo sulle query



Quanti sono gli intervalli?

Ragioniamo sulle query

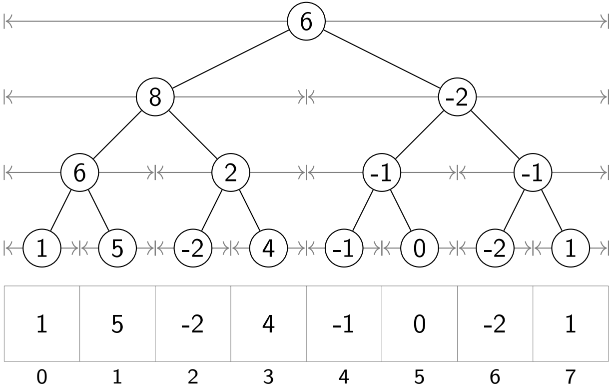


Quanti sono gli intervalli?

$$N + \frac{N}{2} + \frac{N}{4} + \ldots = 2N - 1 = \mathcal{O}(N)$$

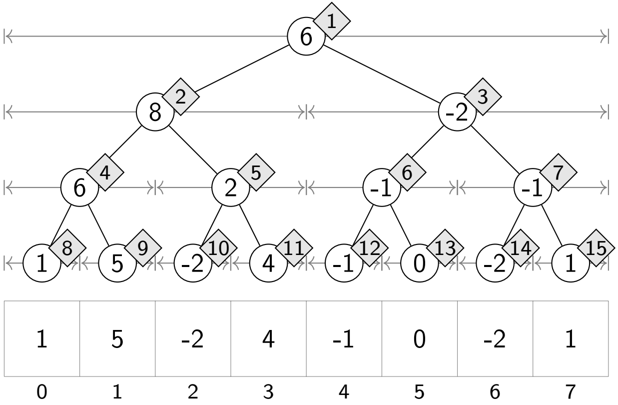
Un albero di intervalli

Gli intervalli si possono vedere come un albero binario.



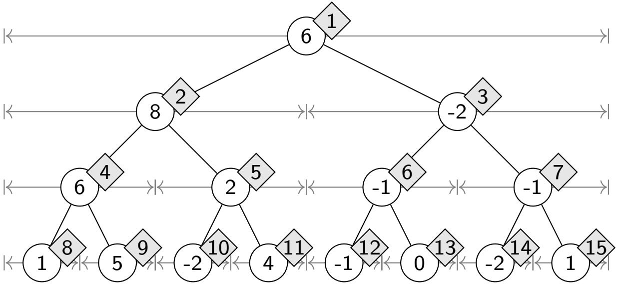
Un albero di intervalli

È utile numerare i nodi in questo modo:



Un albero di intervalli

- La radice ha indice 1.
- Il figlio sinistro del nodo i è $2i$.
- Il figlio destro del nodo i è $2i + 1$.
- Il padre del nodo i è $\lfloor \frac{i}{2} \rfloor$.
- L'i-esima foglia è $N + i$ (con $0 \leq i < N$).
- I nodi sono numerati da 1 a $2N - 1$.
- L'altezza dell'albero è $\mathcal{O}(\log N)$.



1	5	-2	4	-1	0	-2	1
0	1	2	3	4	5	6	7

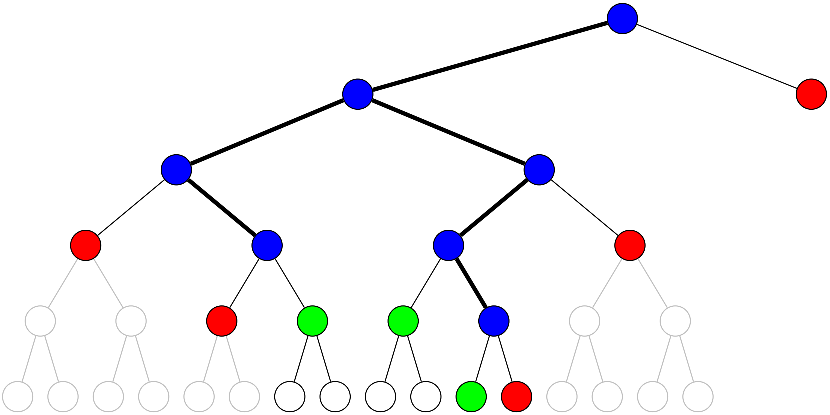
Query

```python
# nodo = indice del nodo
# [nl, nr) = intervallo del nodo (inclusi, esclusi)
# [ql, qr) = intervallo della query (inclusi, esclusi)
def query(node, nl, nr, ql, qr):
    # Il nodo è fuori dall'intervallo della query.
    if qr <= nl or ql >= nr:
        return 0
    # Il nodo è completamente dentro l'intervallo della query.
    if ql <= nl and nr <= qr:
        return tree[node]

    # Scendo verso i figli.
    left = query(2 * node, nl, (nl + nr) / 2, ql, qr)
    right = query(2 * node + 1, (nl + nr) / 2, nr, ql, qr)

    # Combina le risposte dei figli.
    return left + right
```

Struttura delle query

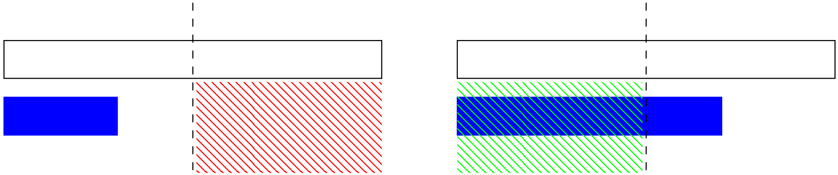


Complessità

Claim il numero di nodi visitati è $\mathcal{O}(\log N)$.
Claim i nodi visitati sono un path dalla radice, che poi si biforca in al più due path.

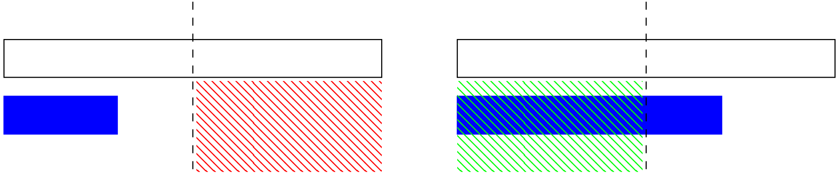
Complessità

Primo caso: l'intervallo della query tocca un estremo dell'intervallo del nodo. Ricorro solo in una delle due metà.



Complessità

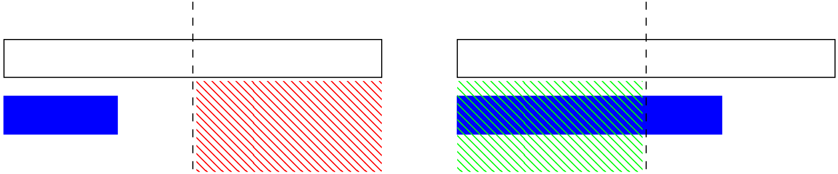
Primo caso: l'intervallo della query tocca un estremo dell'intervallo del nodo. Ricorro solo in una delle due metà.



O una metà è inutile perché completamente fuori dall'intervallo della query.
Oppure una metà è completamente inclusa nella query quindi non serve scendere.
Osservazione i due intervalli rimanenti toccano ancora un estremo.

Complessità

Primo caso: l'intervallo della query tocca un estremo dell'intervallo del nodo. Ricorro solo in una delle due metà.

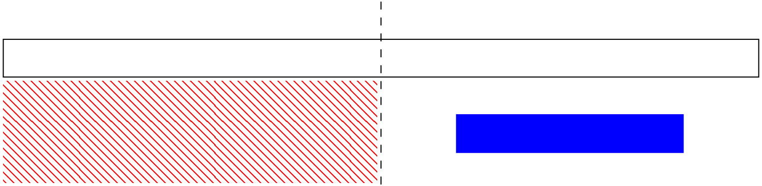


O una metà è inutile perché completamente fuori dall'intervallo della query.
Oppure una metà è completamente inclusa nella query quindi non serve scendere.
Osservazione i due intervalli rimanenti toccano ancora un estremo.

Ad ogni ricorsione scendo verso un solo figlio, quindi al massimo ci sono $\mathcal{O}(\log N)$ livelli.

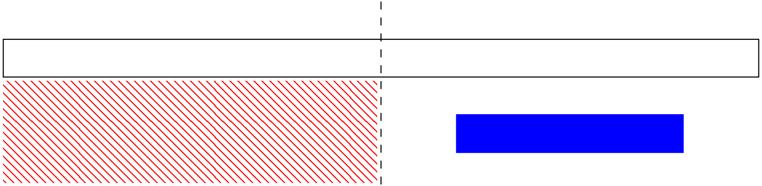
Complessità

Secondo caso: l'intervallo della query non passa per il centro dell'intervallo del nodo. Una delle due metà è inutile, quindi *ricorro solo nell'altra*.



Complessità

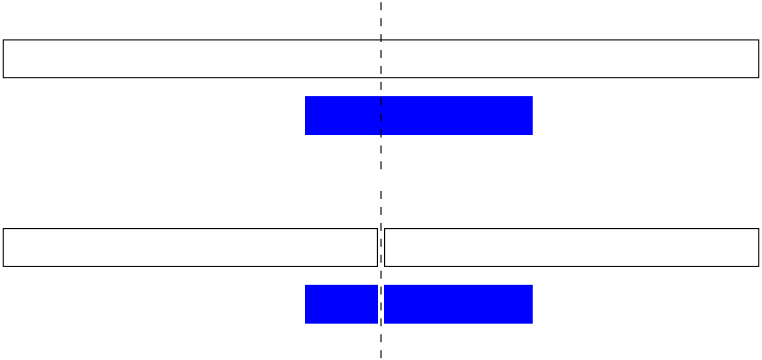
Secondo caso: l'intervallo della query non passa per il centro dell'intervallo del nodo. Una delle due metà è inutile, quindi *ricorro solo nell'altra*.



Sto scendendo di un livello ad ogni ricorsione: al massimo ci sono $\mathcal{O}(\log N)$ livelli.

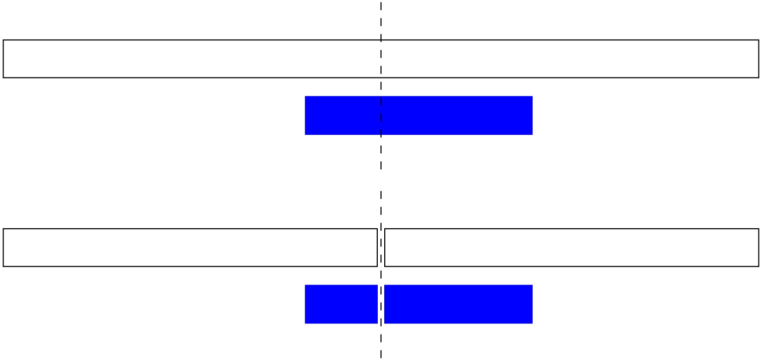
Complessità

Terzo caso: l'intervallo della query passa per il centro dell'intervallo del nodo. *Ricorro in entrambe le metà*.



Complessità

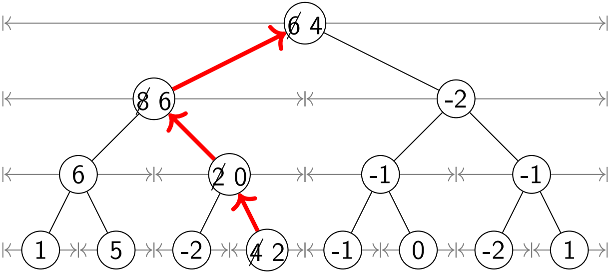
Terzo caso: l'intervallo della query passa per il centro dell'intervallo del nodo. *Ricorro in entrambe le metà*.



Da ora in poi il sotto-intervallo della query toccherà sempre un estremo dell'intervallo dei nodi.

Update

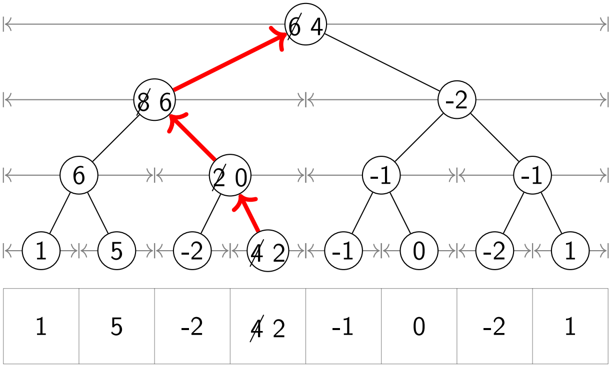
Un update può partire da una foglia, aggiornare il nodo e risalire verso la radice.



1	5	-2	~~4~~ 2	-1	0	-2	1

Update

Un update può partire da una foglia, aggiornare il nodo e risalire verso la radice.



Ci si muove da una foglia alla radice: $\mathcal{O}(\log N)$.

Update

Il codice di questo update è molto semplice:

```
def update(i, x):
    # Aggiorna la foglia.
    node = N + i
    tree[node] = x

    # Risali fino alla radice.
    node /= 2
    while node > 0:
        tree[node] = tree[node * 2] + tree[node * 2 + 1]
        node /= 2
```

Update (idea alternativa)

Approccio ricorsivo: parto dalla radice e scendo verso la foglia da aggiornare.

```
def update(node, nl, nr, i, x):
    # Il sottoalbero non contiene i.
    if i < nl or i >= nr:
        return tree[node]
    # Sono arrivato alla foglia da aggiornare.
    if i == nl and nr - nl == 1:
        tree[node] = x
        return tree[node]

    # Aggiorna i sottoalberi
    left = update(node * 2, nl, (nl + nr) / 2, i, x)
    right = update(node * 2 + 1, (nl + nr) / 2, nr, i, x)

    # Combina le risposte dei figli.
    tree[node] = left + right
    return tree[node]
```

Update (idea alternativa)

Osservazione Questa versione dell'update è praticamente identica ad una query in cui $l = r - 1$.

Update su intervalli

Potenzialmente i nodi dell'albero da modificare sono tanti.
`update(0, N, x)` modifica *tutti* i nodi dell'albero.

Update su intervalli

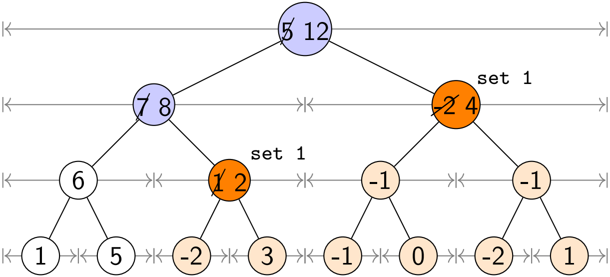
Potenzialmente i nodi dell'albero da modificare sono tanti.
`update(0, N, x)` modifica *tutti* i nodi dell'albero.

Idea *lazy propagation*

- Quando devo modificare *tutto* un sottoalbero, modifico solo la radice e mi ricordo che *prima o poi* dovrò aggiornare anche i figli.
- **Requisito**: devo saper aggiornare un nodo senza aver già aggiornato i figli.

Lazy propagation

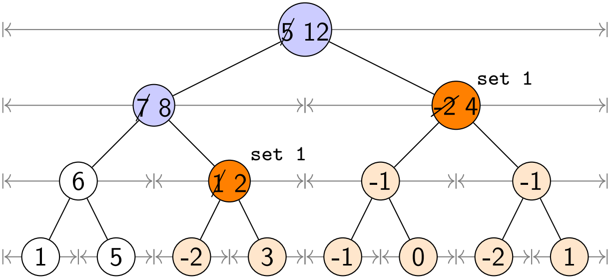
`update(2, 8, 1)`



1	5	~~-2~~ 1	~~3~~ 1	~~-1~~ 1	~~0~~ 1	~~-2~~ 1	~~1~~ 1
0	1	2	3	4	5	6	7

Lazy propagation

`update(2, 8, 1)`

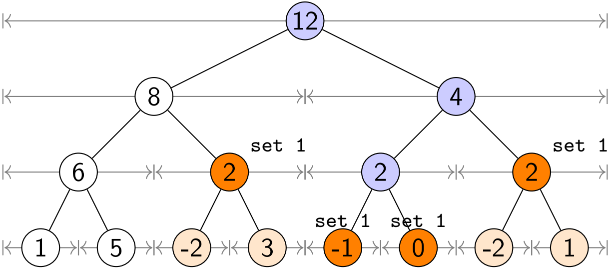


1	5	~~-2~~ 1	~~3~~ 1	~~-1~~ 1	~~0~~ 1	~~-2~~ 1	~~1~~ 1
0	1	2	3	4	5	6	7

Complessità di un update: $\mathcal{O}(\log N)$ perché *tocco* gli stessi nodi di una query su quell'intervallo.

Lazy propagation

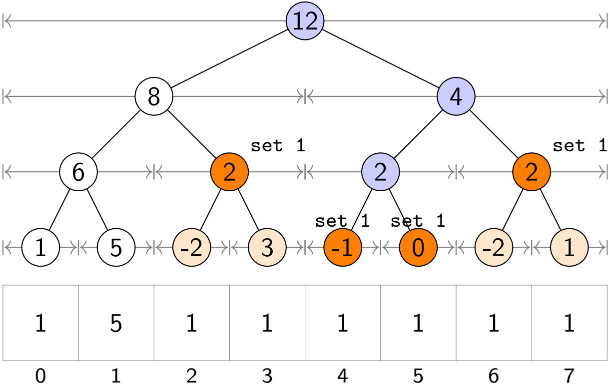
`query(4, 6)`



1	5	1	1	1	1	1	1
0	1	2	3	4	5	6	7

Lazy propagation

`query(4, 6)`



Complessità di una query: $\mathcal{O}(\log N)$ perché *propago* solo nodi dei path della query.

Dettagli implementativi

- Ogni volta che si entra in un nodo, se c'è da propagare allora propaga.
- Usa una `struct` per rappresentare un nodo.

```
struct node {
    int value;
    int update;
    bool has_update;
};
```

Funzione per propagare

```
def propagate(node, nl, nr):
    if not tree[node].has_update:
        return

    tree[node].value = (nr - nl) * tree[node].update
    tree[node].has_update = False

    if nl != nr - 1:
        left = 2 * node
        right = 2 * node + 1
        tree[left].update = tree[node].update
        tree[left].has_update = True

        tree[right].update = tree[node].update
        tree[right].has_update = True
```

Più informazioni nel nodo

All'interno di `struct nodo` si possono inserire anche più informazioni per rispondere a query più complesse.

Più informazioni nel nodo

All'interno di `struct nodo` si possono inserire anche più informazioni per rispondere a query più complesse.

Il valore del nodo contiene informazioni su un intervallo.

Più informazioni nel nodo

All'interno di `struct nodo` si possono inserire anche più informazioni per rispondere a query più complesse.

Il valore del nodo contiene informazioni su un intervallo.

Requisiti:

- Poter unire intervalli durante una query.
- Poter ricostruire il valore di un nodo dato quello dei figli (per gli update).
- Poter ricostruire il valore di un nodo dato un update (per la lazy propagation).

Più informazioni nel nodo

All'interno di `struct nodo` si possono inserire anche più informazioni per rispondere a query più complesse.

Il valore del nodo contiene informazioni su un intervallo.

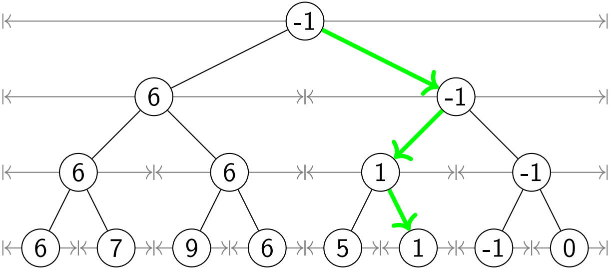
Requisiti:

- Poter unire intervalli durante una query.
- Poter ricostruire il valore di un nodo dato quello dei figli (per gli update).
- Poter ricostruire il valore di un nodo dato un update (per la lazy propagation).

È spesso comodo definire una funzione `merge` che unisce due nodi.

Query più complesse

A volte le query sono più complesse e richiedono visite particolari dell'albero. Per esempio `segtree` chiede la funzione `lower_bound` (la posizione dell'elemento più a sinistra minore o uguale a un valore).



6	7	9	6	5	1	-1	0

Altre varianti di Segment Tree

- Segment tree sparsi: l'array dei valori è molto grande e non ci sta in memoria. Si possono creare i nodi dell'albero solo quando si accede la prima volta (usando puntatori). La complessità rimane $\mathcal{O}(\log N)$.

Altre varianti di Segment Tree

- Segment tree sparsi: l'array dei valori è molto grande e non ci sta in memoria. Si possono creare i nodi dell'albero solo quando si accede la prima volta (usando puntatori). La complessità rimane $\mathcal{O}(\log N)$.
- Segment tree persistenti: un update duplica l'albero e fa la modifica solo sulla copia. Ogni query è relativa ad una delle copie dell'albero (non necessariamente dopo l'ultimo update). La complessità rimane $\mathcal{O}(\log N)$.