Data Structures

Edoardo Morassutto, Dario Petrillo, Lorenzo Ferrari

Volterra, 8 dicembre 2023

1. Introduzione
   - Esempi di strutture dati
   - Strutture dati nella STL
2. Disjoint Set Union
   - Problema
   - Implementazione
   - Path Compression
   - Union by Size/Rank
3. Segment Tree
   - Esempio di problema
   - Generalizzazione
   - Segment Tree Sparsi

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++):

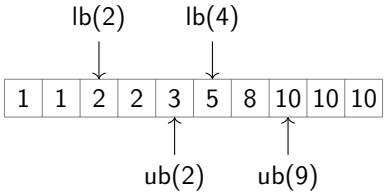
```
struct cmp {
    bool operator()(const T& a, const T& b) const { return a < b; }
};
std::set<T, cmp> s;
```

`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 $N-1$
    - **nota**: non è il modo più veloce per comprimere!

`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 $N-1$
    - **nota**: non è il modo più veloce per comprimere!
- **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 $N-1$
    - **nota**: non è il modo più veloce per comprimere!
- **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*.

Utilizzo non ovvio della STL

Problema (`std::set` potenziato)

Definisci un `std::set` che oltre alle operazioni standard supporti:

- *`increase(x)` aumenta di x tutti gli elementi del set.*

Utilizzo non ovvio della STL

Problema (`std::set` potenziato)

Definisci un `std::set` che oltre alle operazioni standard supporti:

- *`increase(x)` aumenta di `x` tutti gli elementi del set.*

Osservazione

- non possiamo *davvero* aggiornare tutti i valori in meno di $\mathcal{O}(N)$

Utilizzo non ovvio della STL

Problema (`std::set` potenziato)

Definisci un `std::set` che oltre alle operazioni standard supporti:

- *`increase(x)` aumenta di `x` tutti gli elementi del set.*

Osservazione

- non possiamo *davvero* aggiornare tutti i valori in meno di $\mathcal{O}(N)$
- ma a noi basta la struttura *si comporti* come un set!

Utilizzo non ovvio della STL

Problema (`std::set` potenziato)

Definisci un `std::set` che oltre alle operazioni standard supporti:

- *`increase(x)` aumenta di x tutti gli elementi del set.*

Osservazione

- non possiamo *davvero* aggiornare tutti i valori in meno di $\mathcal{O}(N)$
- ma a noi basta la struttura *si comporti* come un set!

Soluzione

- manteniamo un normale `std::set` e una variabile `shift` che indica di quanto i valori "fisici" nel set sono minori dei valori secondo il problema.

Set potenziato

```
struct MySet {
    int shift = 0;
    set<int> s;

    void insert(int v) {
        s.insert(v - shift);
    }

    void erase(int v) {
        s.erase(v - shift);
    }

    void increase(int x) {
        shift += x;
    }
};
```

Disjoint Set Union

Problema

Problema (DSU)

Sono dati N elementi, inizialmente ogni elemento è in un insieme con solo se stesso. Implementa una struttura che supporti le seguenti operazioni:

- *inizializza a `n` insiemi, ognuno con un singolo elemento;*
- *`find_set(v)` ritorna il rappresentante e dell'insieme di `v`;*
- *`union_set(a, b)` unisci i set contenenti `a` e `b`. Ritorna un `bool` che indica se la struttura è cambiata.*

Problema

Problema (DSU)

Sono dati N elementi, inizialmente ogni elemento è in un insieme con solo se stesso. Implementa una struttura che supporti le seguenti operazioni:

- *inizializza a `n` insiemi, ognuno con un singolo elemento;*
- *`find_set(v)` ritorna il rappresentante e dell'insieme di `v`;*
- *`union_set(a, b)` unisci i set contenenti `a` e `b`. Ritorna un `bool` che indica se la struttura è cambiata.*

$$1 : \{1\}$$
$$2 : \{2\}$$
$$3 : \{3\}$$
$$4 : \{4\}$$
$$5 : \{5\}$$

Problema

Problema (DSU)

Sono dati N elementi, inizialmente ogni elemento è in un insieme con solo se stesso. Implementa una struttura che supporti le seguenti operazioni:

- *inizializza a `n` insiemi, ognuno con un singolo elemento;*
- *`find_set(v)` ritorna il rappresentante e dell'insieme di `v`;*
- *`union_set(a, b)` unisci i set contenenti `a` e `b`. Ritorna un `bool` che indica se la struttura è cambiata.*

```
union_set(2, 3); union_set(1, 5)
```

$$1 : \{1, 5\}$$
$$3 : \{2, 3\}$$
$$4 : \{4\}$$

Problema

Problema (DSU)

Sono dati N elementi, inizialmente ogni elemento è in un insieme con solo se stesso. Implementa una struttura che supporti le seguenti operazioni:

- *inizializza a `n` insiemi, ognuno con un singolo elemento;*
- *`find_set(v)` ritorna il rappresentante e dell'insieme di `v`;*
- *`union_set(a, b)` unisci i set contenenti `a` e `b`. Ritorna un `bool` che indica se la struttura è cambiata.*

```
find_set(1) == 1    find_set(5) == 1    find_set(2) == 3
```

$$1 : \{1, 5\}$$
$$3 : \{2, 3\}$$
$$4 : \{4\}$$

Idea

- manteniamo una foresta, in cui ogni albero corrisponde a una componente connessa
- la radice dell'albero è il nodo rappresentante della componente
- inizialmente ci sono `n` alberi ognuno composto da un unico nodo
- per ogni nodo `v`, salviamo il suo parent `p[v]`[1]

[1] per la radice vale `p[v] == v`

Implementazione

```
struct Dsu {
    int n;
    vector<int> p;
    Dsu(int _n) : n(_n), p(_n) {
        iota(begin(p), end(p), 0);
    }
    int find_set(int v) {
        return p[v] == v ? v : find_set(p[v]);
    }
    bool union_set(int a, int b) {
        a = find_set(a);
        b = find_set(b);
        if (a == b) return false;
        p[b] = a;
        return true;
    }
};
```

Complessità e Path Compression

Complessità

- Inizializzazione $\mathcal{O}(N)$;
- `find_set` $\mathcal{O}(N)$;
- `union_set` $\mathcal{O}(N)$.

Complessità e Path Compression

Complessità

- Inizializzazione $\mathcal{O}(N)$;
- `find_set` $\mathcal{O}(N)$;
- `union_set` $\mathcal{O}(N)$.

Path Compression

Come migliorare? Possiamo modificare la struttura per ricordare le chiamate a `find_set` già eseguite.

Complessità e Path Compression

Complessità

- Inizializzazione $\mathcal{O}(N)$;
- `find_set` $\mathcal{O}(N)$;
- `union_set` $\mathcal{O}(N)$.

Path Compression

Come migliorare? Possiamo modificare la struttura per ricordare le chiamate a `find_set` già eseguite.

```
int find_set(int v) {
    return p[v] == v ? v : p[v] = find_set(p[v]);
}
```

Complessità e Path Compression

Complessità

- Inizializzazione $\mathcal{O}(N)$;
- `find_set` $\mathcal{O}(N)$;
- `union_set` $\mathcal{O}(N)$.

Path Compression

Come migliorare? Possiamo modificare la struttura per ricordare le chiamate a `find_set` già eseguite.

```
int find_set(int v) {
    return p[v] == v ? v : p[v] = find_set(p[v]);
}
```

In questo modo la complessità migliora a $\mathcal{O}(\log N)$ per query in media.

Path Compression

Complessità?

- Inizializzazione $\mathcal{O}(N)$;
- `find_set` $\mathcal{O}(N)$;
- `union_set` $\mathcal{O}(N)$.

Union by Size/Rank

Union by Size/Rank

Ulteriore ottimizzazione: per unire due alberi attacchiamo il "più piccolo" al "più grande". Esempio:

- per ogni componente salviamo quanti nodi ci sono;
- attacchiamo sempre l'albero con meno nodi a quello con più nodi e aggiorniamo la dimensione della radice.

In questo modo l'altezza dell'albero non supera $\lceil \log N \rceil$, quindi ogni query costa $\mathcal{O}(\log N)$

Union by Size/Rank

Union by Size/Rank

Ulteriore ottimizzazione: per unire due alberi attacchiamo il "più piccolo" al "più grande". Esempio:

- per ogni componente salviamo quanti nodi ci sono;
- attacchiamo sempre l'albero con meno nodi a quello con più nodi e aggiorniamo la dimensione della radice.

In questo modo l'altezza dell'albero non supera $\lceil \log N \rceil$, quindi ogni query costa $\mathcal{O}(\log N)$

Combinata con path compression, la complessità media di un'operazione find scende a $\mathcal{O}(\alpha(N))$, moralmente costante.

```
struct Dsu {
    int n;
    vector<int> p, s;
    Dsu(int _n) : n(_n), p(_n), s(_n, 1) {
        iota(begin(p), end(p), 0);
    }
    int find_set(int v) {
        return p[v] == v ? v : p[v] = find_set(p[v]);
    }
    bool union_set(int a, int b) {
        a = find_set(a);
        b = find_set(b);
        if (a == b) return false;
        if (s[a] < s[b]) swap(a, b);
        p[b] = a;
        s[a] += s[b];
        return true;
    }
};
```

Segment Tree

Esempio di problema

Problema (Somma di intervalli)

Dato un array 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 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 $\mathcal{O}(r - l) = \mathcal{O}(N)$.

Soluzione un po' meno banale

Tengo le somme prefisse in un `std::vector`.

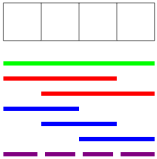
- Update in $\mathcal{O}(N)$.
- Query in $\mathcal{O}(1)$.

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:

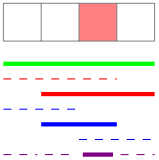
- Gli intervalli sono tanti: $\mathcal{O}(N^2)$.

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: $\mathcal{O}(N^2)$.



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?

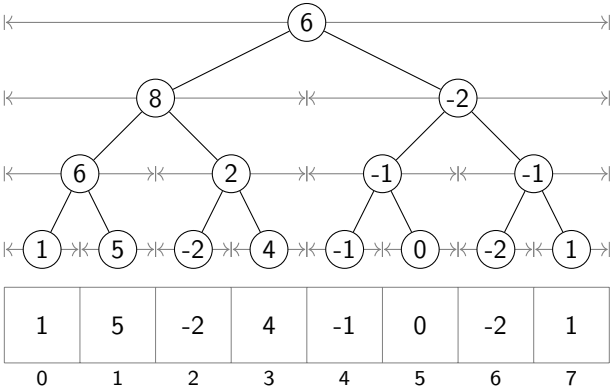
Ragioniamo sulle query

Quanti sono gli intervalli?

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

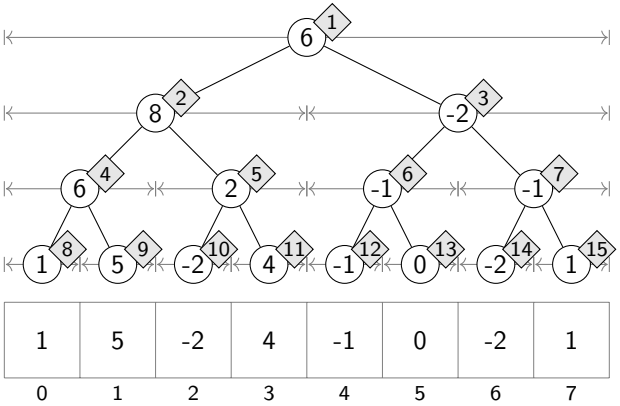
Un albero di intervalli

Gli intervalli si possono vedere come un albero binario.



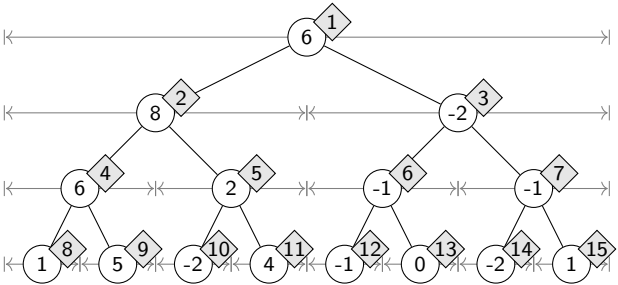
Un albero di intervalli

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 $\mathcal{O}(\log N)$.



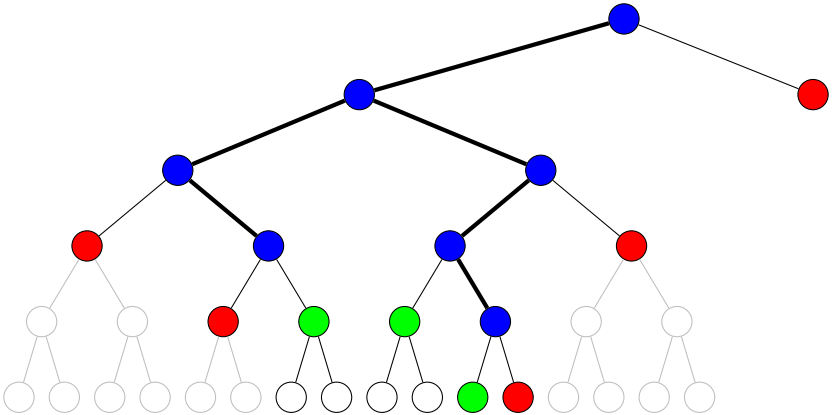
Query

```
# 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

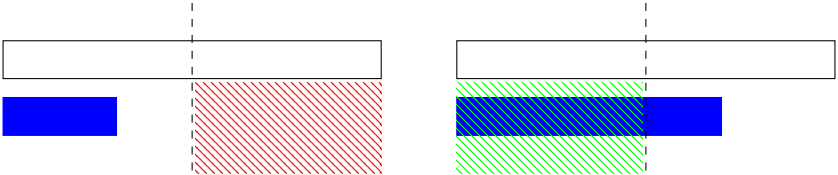


Complessità

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.

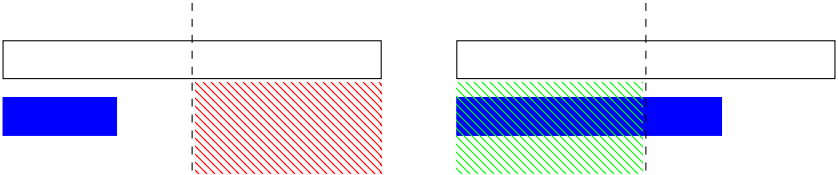
Complessità

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



Complessità

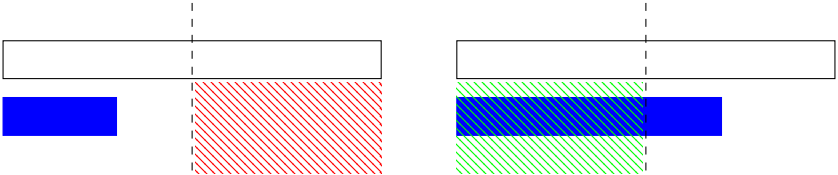
Primo caso: l'intervallo della query tocca un estremo dell'intervallo del nodo. Ricorro solo in una delle due metà.



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.

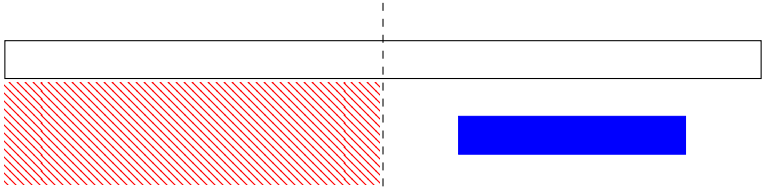


O una metà è inutile perché completamente fuori dall'intervallo della query.
Oppure una metà è completamente inclusa nella query quindi non serve scendere.
Osservazione i due intervalli rimanenti toccano ancora un estremo.

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

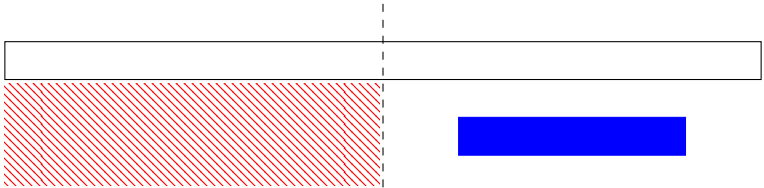
Complessità

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

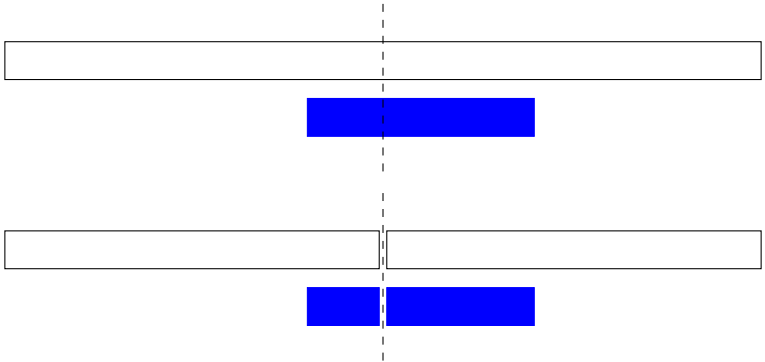
Secondo caso: l'intervallo della query non passa per il centro dell'intervallo del nodo. Una delle due metà è inutile, quindi *ricorro solo nell'altra*.



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

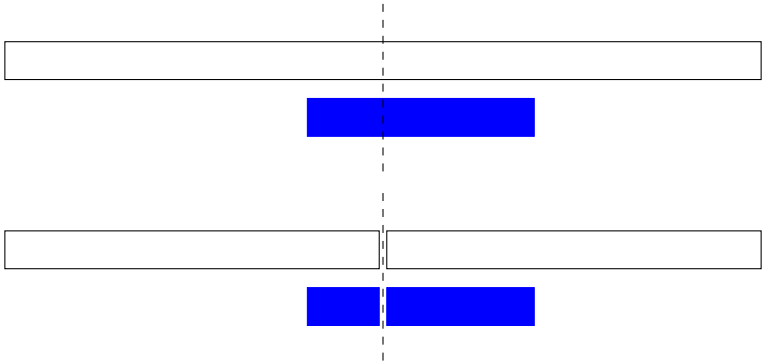
Complessità

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



Complessità

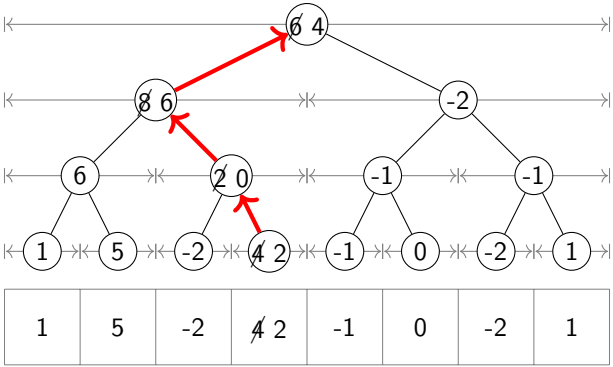
Terzo caso: l'intervallo della query passa per il centro dell'intervallo del nodo. *Ricorro in entrambe le metà.*



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

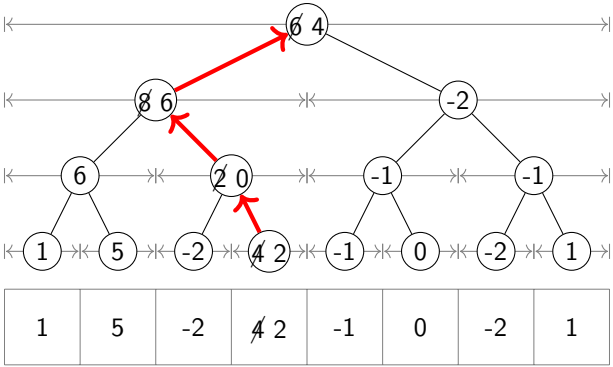
Update

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



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

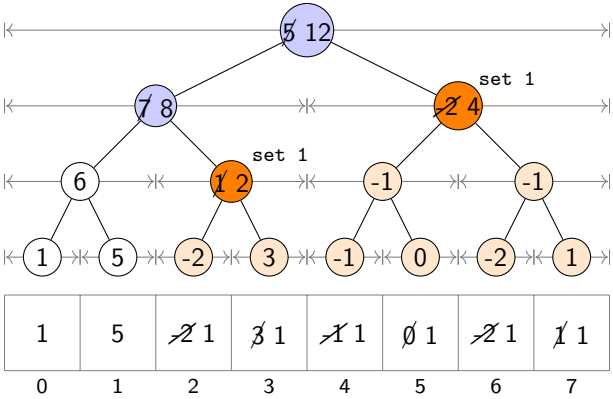
Potenzialmente i nodi dell'albero da modificare sono tanti.
`update(0, N, x)` modifica *tutti* i nodi dell'albero.

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.

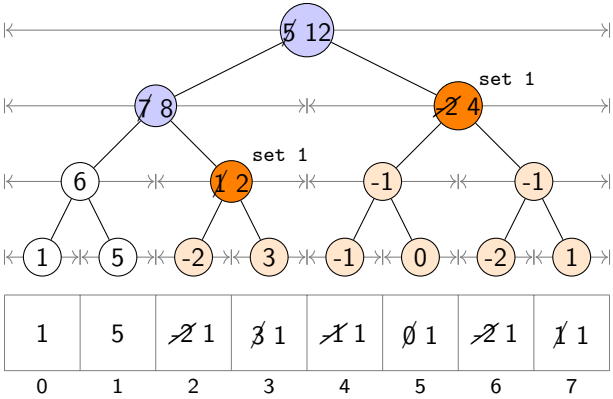
Lazy propagation

```
update(2, 8, 1)
```



Lazy propagation

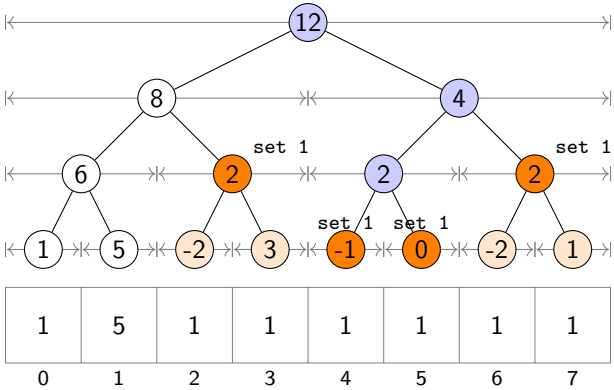
`update(2, 8, 1)`



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

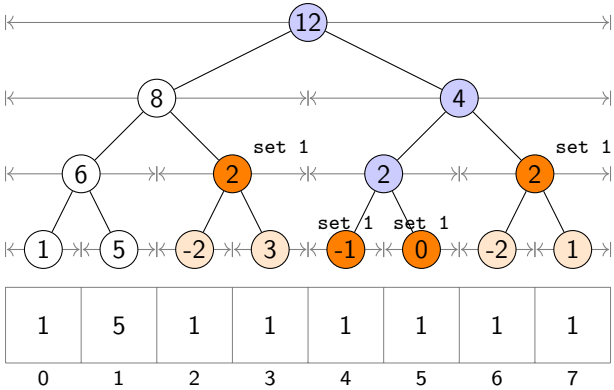
Lazy propagation

`query(4, 6)`



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

    # se node non è una foglia, propaga sui figli
    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.

[2] non necessariamente commutativa, se uniamo i nodi nell'ordine giusto.

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.

[2] non necessariamente commutativa, se uniamo i nodi nell'ordine giusto.

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 ricostruire il valore di un nodo dati due nodi figli.
- L'operazione unione di nodi deve essere **associativa**[2]. Esempi:
  - somma, prodotto, massimo, xor
  - prodotto di matrici
- Poter ricostruire il valore di un nodo dato un update (per la lazy propagation).

[2] non necessariamente commutativa, se uniamo i nodi nell'ordine giusto.

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:

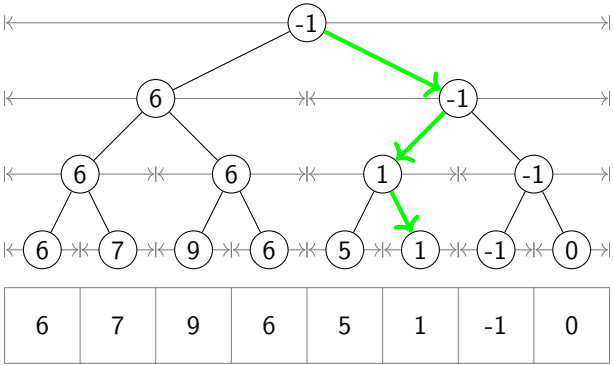
- Poter ricostruire il valore di un nodo dati due nodi figli.
- L'operazione unione di nodi deve essere **associativa**[2]. Esempi:
  - somma, prodotto, massimo, xor
  - prodotto di matrici
- Poter ricostruire il valore di un nodo dato un update (per la lazy propagation).

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

[2] non necessariamente commutativa, se uniamo i nodi nell'ordine giusto.

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).



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 $\mathcal{O}(\log N)$.

[3] una quantità che cambia in una sola direzione si dice "monovariante".

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 $\mathcal{O}(\log N)$.

[3] una quantità che cambia in una sola direzione si dice "monovariante".

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 $\mathcal{O}(\log N)$.
- Segment Tree beats: dato qualche range update strano, se esiste una quantità X che può solo decrescere[3], decresce al più poche volte e ogni update che *cambia* una foglia fa decrescere X, allora fare gli update in modo "naive" ha una buona complessità totale.

[3] una quantità che cambia in una sola direzione si dice "monovariante".

Segment Tree sparsi

Problema (Somma di intervalli grandi)

Dato un array A di $N \leq 10^{18}$ interi inizialmente tutti 0 vogliamo supportare le seguenti operazioni:

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

Segment Tree sparsi

Problema (Somma di intervalli grandi)

Dato un array A di $N \leq 10^{18}$ interi inizialmente tutti 0 vogliamo supportare le seguenti operazioni:

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

Operazioni offline

Se le gli update sono note a priori, possiamo comprimere i (pochi) indici che vengono toccati da un update e ridurci al primo problema.

- sia `idx` un `std::vector` contentente gli indici.
- manteniamo un normale segment di dimensione `idx.size()`.
- prima di ogni update o query, traduciamo ogni indice `i` in `lower_bound(begin(idx), end(idx), i) - begin(idx)`.

Segment Tree sparsi

Per risolvere il problema nel caso generale immaginiamo un normale segment tree con i puntatori, ma creiamo un nodo solo quando ci si accede la prima volta. La complessità di Q operazioni è $\mathcal{O}(Q \log N)$ di tempo e di spazio.

Segment Tree sparsi

Per risolvere il problema nel caso generale immaginiamo un normale segment tree con i puntatori, ma creiamo un nodo solo quando ci si accede la prima volta. La complessità di Q operazioni è $\mathcal{O}(Q \log N)$ di tempo e di spazio.

Ogni nodo contiene le seguenti informazioni:

- puntatori `*l`, `*r` ai suoi figli sinistro e destro;
- estremi `a`, `b` del suo intervallo $[a, b)$;
- somma degli `A[i]` nel suo intervallo.

Segment Tree sparsi

```
struct node {
    node* l = nullptr;
    node* r = nullptr;
    int sum = 0;
    long long a, b;

    // costruttore
    node(long long _a, long long _b) :
        a(_a), (_b) {}
    // distruttore (importante per liberare la memoria)
    ~node() {
        if (l) delete l;
        if (r) delete r;
    }

    // ... dichiarazione delle funzioni increase e somma
};
```

Segment Tree sparsi

```
friend void increase(node * &N, long long pos) {
    N->sum++;
    if (N->b - N->a <= 1) return;
    if (pos < (N->a + N->b) / 2) {
        // se non esiste, crea il figlio sinistro
        if (!N->l) {
            N->l = new node(N->a, (N->a + N->b) / 2);
        }
        increase(N->l, pos);
    } else {
        // se non esiste, crea il figlio destro
        if (!N->r) {
            N->r = new node((N->a + N->b) / 2, N->b);
        }
        increase(N->r, pos);
    }
}
```

Segment Tree sparsi

```
friend int somma(node * &N, long long L, long long R) {
    // il nodo non esiste
    if (!N) return 0;
    // intersezione vuota
    if (N->b <= L || R <= N->a) {
        return 0;
    }
    // intervallo completamente incluso
    if (L <= N->a && N->b <= R) {
        return N->sum;
    }
    // ricorri sui figli
    return query(N->l, L, R) + query(N->r, L, R);
}
```