Data Structures

Lorenzo Ferrari

Volterra, 7 dicembre 2024

Introduzione

Cos'è una struttura dati

Una struttura dati è un modo furbo per mantenere informazioni.

Su questi dati vogliamo eseguire delle operazioni (abbastanza velocemente).

La complessità di solito dipende dal numero di elementi all'interno della struttura, la cui dimensione indichiamo con N.

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>`
- `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>`
- `std::unordered_multiset<T>` e `std::unordered_multimap<K, V>`
- `std::pair<A, B>` e `std::array<T, n>`

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>`
- `std::unordered_multiset<T>` e `std::unordered_multimap<K, V>`
- `std::pair<A, B>` e `std::array<T, n>`

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>`
- `std::unordered_multiset<T>` e `std::unordered_multimap<K, V>`
- `std::pair<A, B>` e `std::array<T, n>`

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>`
- `std::unordered_multiset<T>` e `std::unordered_multimap<K, V>`
- `std::pair<A, B>` e `std::array<T, n>`

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>`
- `std::unordered_multiset<T>` e `std::unordered_multimap<K, V>`
- `std::pair<A, B>` e `std::array<T, n>`

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>`
- `std::unordered_multiset<T>` e `std::unordered_multimap<K, V>`
- `std::pair<A, B>` e `std::array<T, n>`

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>`
- `std::unordered_multiset<T>` e `std::unordered_multimap<K, V>`
- `std::pair<A, B>` e `std::array<T, n>`

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>`
- `std::unordered_multiset<T>` e `std::unordered_multimap<K, V>`
- `std::pair<A, B>` e `std::array<T, n>`

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>`
- `std::unordered_multiset<T>` e `std::unordered_multimap<K, V>`
- `std::pair<A, B>` e `std::array<T, n>`

Non presenti nell'STL

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

Cosa devo sapere della STL?

In seguito, consideriamo `std::set<int> s` e `std::map<int, int> m`.

- `s.lower_bound(x)` ritorna un puntatore al minimo elemento di s maggiore o uguale a x intero, `end(s)` se un tale elemento non esiste;
- `s.upper_bound(x)` ritorna un puntatore al minimo elemento di s strettamente maggiore di x intero, `end(s)` se un tale elemento non esiste;
- `s.count(x)` ritorna il numero di occorrenze di un intero x in s. Poiché s non ha duplicati, il numero di occorrenze è 0 o 1. `if (s.count(x))` è un buon modo per controllare se x appartiene a s;
- le tre funzioni precedenti funzionano anche su m mappa, nel cui caso i confronti con x si riferiscono alle chiavi e non ai valori;
- `for (auto [key, value] : m) { ... }` itera sulle chiavi e i valori contenuti in m.

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;
    }
};
```

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`.

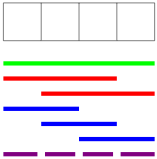
- Update in $\mathcal{O}(N)$.
- Query in $\mathcal{O}(1)$.

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:

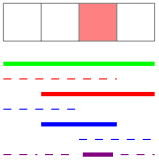
- Gli intervalli sono tanti: $\mathcal{O}(N^2)$.

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?

- Solo alcuni intervalli di lunghezza 2^k per $k = 0 \dots \log_2 N$.

Ragioniamo sulle query

Quanti sono gli intervalli?

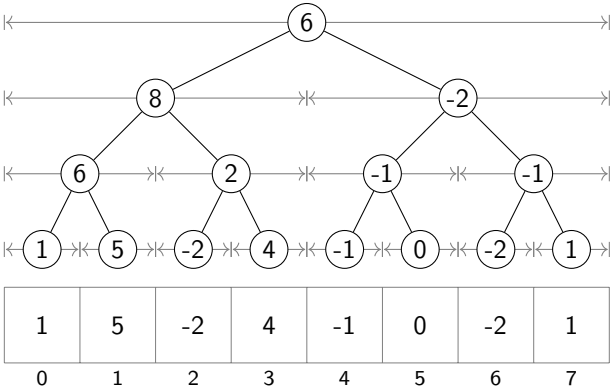
Ragioniamo sulle query

Quanti sono gli intervalli?

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

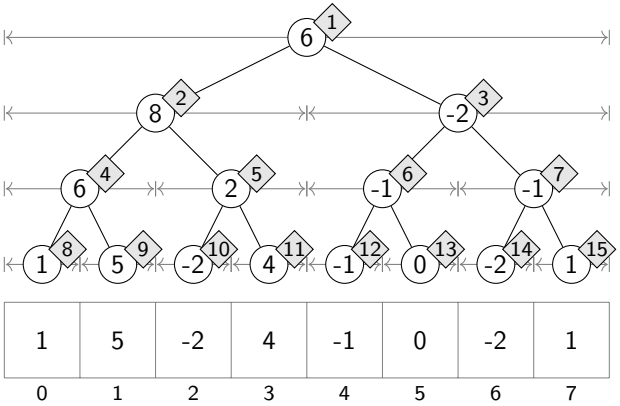
Un albero di intervalli

Gli intervalli si possono vedere come un albero binario.



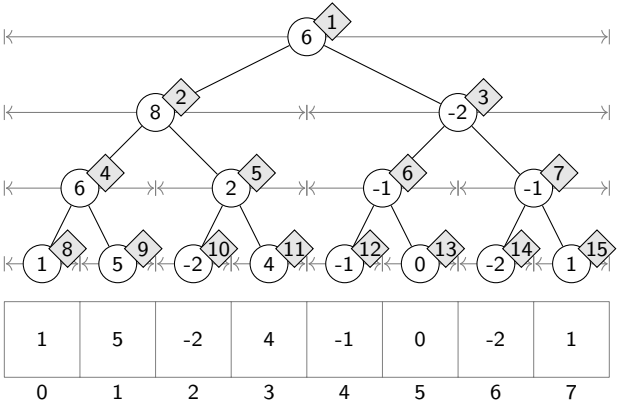
Un albero di intervalli

È utile numerare i nodi in questo modo:



Un albero di intervalli

- La radice ha indice 1.
- I figli sinistro e destro del nodo i sono rispettivamente $2i$ e $2i + 1$.
- 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 $\mathcal{O}(\log N)$.



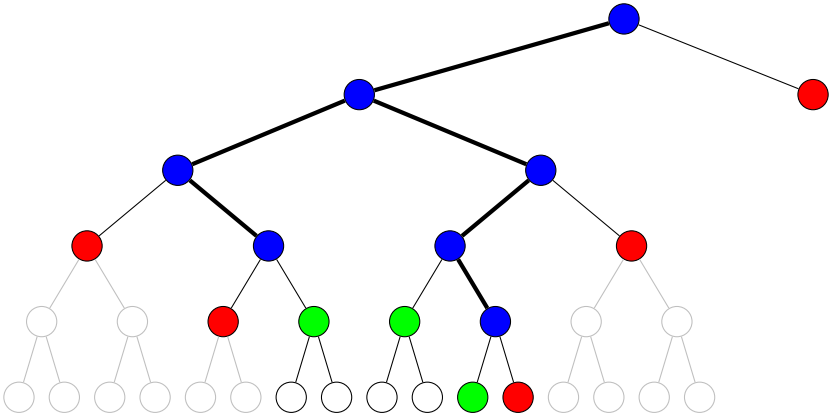
Query

```
# 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

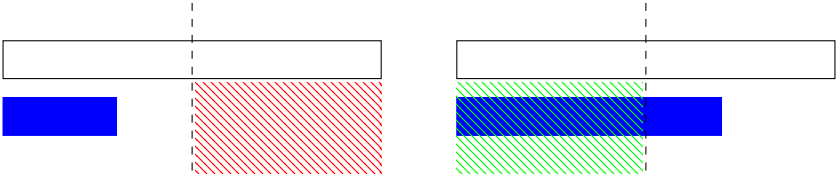


Complessità

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.

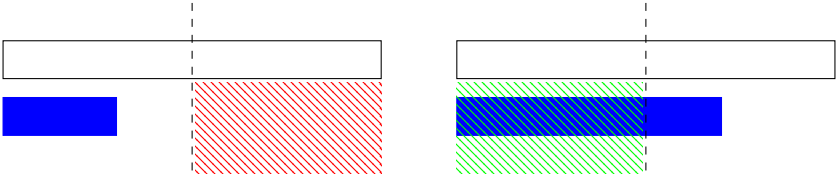
Complessità

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



Complessità

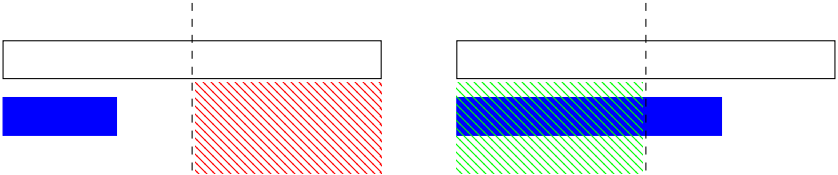
Primo caso: l'intervallo della query tocca un estremo dell'intervallo del nodo. Ricorro solo in una delle due metà.



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.

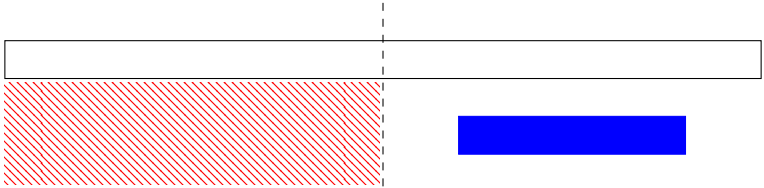


O una metà è inutile perché completamente fuori dall'intervallo della query.
Oppure una metà è completamente inclusa nella query quindi non serve scendere.
Osservazione i due intervalli rimanenti toccano ancora un estremo.

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

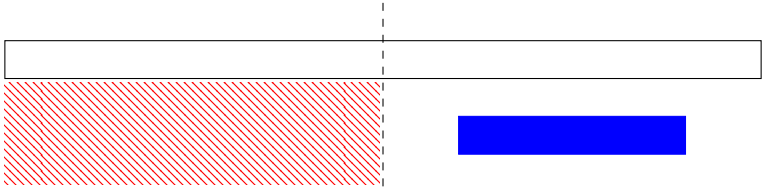
Complessità

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

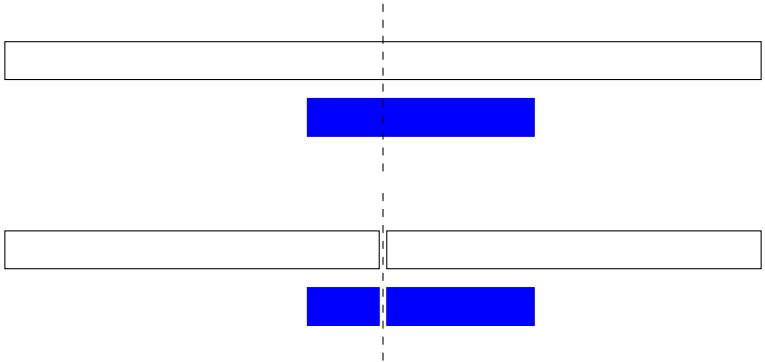
Secondo caso: l'intervallo della query non passa per il centro dell'intervallo del nodo. Una delle due metà è inutile, quindi *ricorro solo nell'altra*.



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

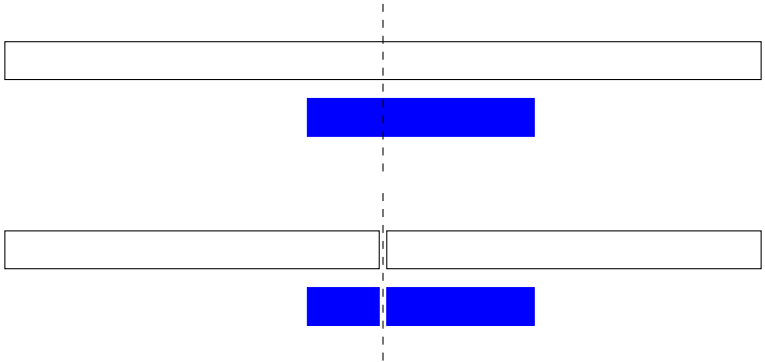
Complessità

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



Complessità

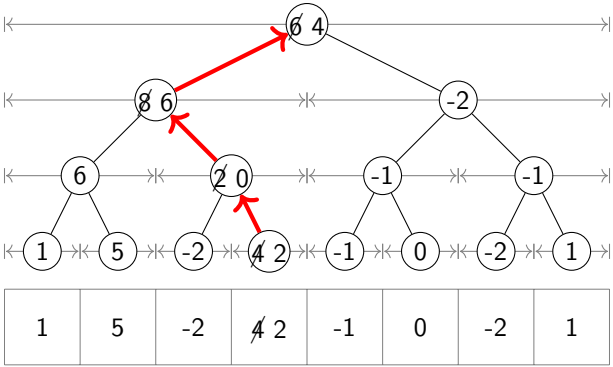
Terzo caso: l'intervallo della query passa per il centro dell'intervallo del nodo. *Ricorro in entrambe le metà.*



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

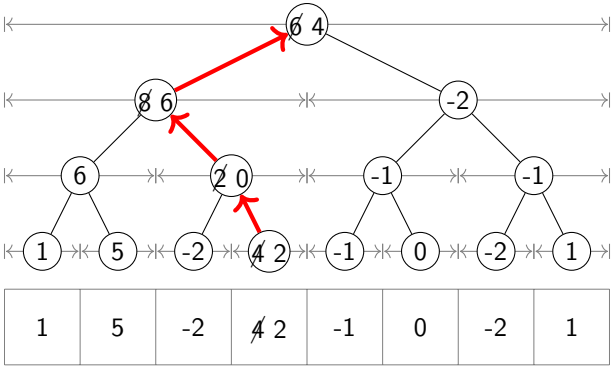
Update

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

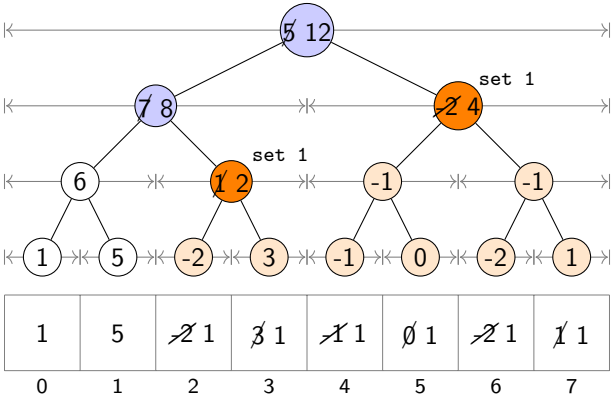
Potenzialmente i nodi dell'albero da modificare sono tanti.
`update(0, N, x)` modifica *tutti* i nodi dell'albero.

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.

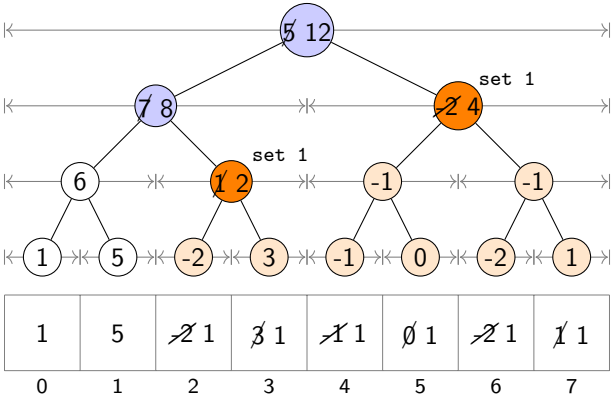
Lazy propagation

`update(2, 8, 1)`



Lazy propagation

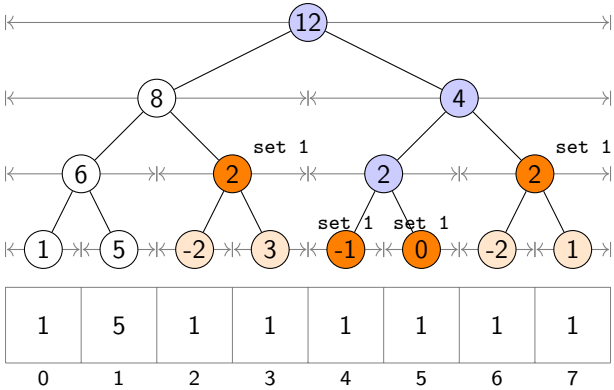
`update(2, 8, 1)`



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

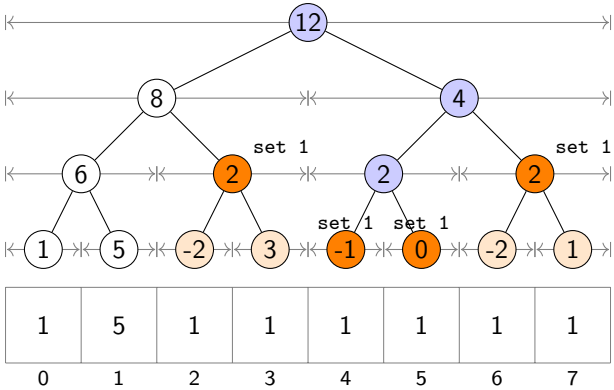
Lazy propagation

`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.

[1] 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.

[1] 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**[1]. Esempi:
  - somma, prodotto, massimo, xor
  - prodotto di matrici
- Poter ricostruire il valore di un nodo dato un update (per la lazy propagation).

[1] 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:

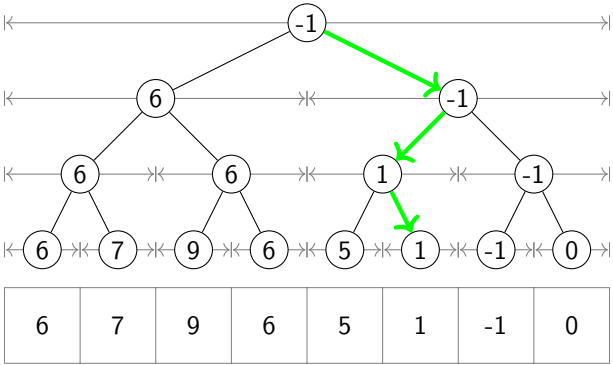
- Poter ricostruire il valore di un nodo dati due nodi figli.
- L'operazione unione di nodi deve essere **associativa**[1]. Esempi:
  - somma, prodotto, massimo, xor
  - prodotto di matrici
- Poter ricostruire il valore di un nodo dato un update (per la lazy propagation).

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

[1] 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 $\mathcal{O}(\log N)$.

[2] 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 $\mathcal{O}(\log N)$.

[2] 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[2], 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.

[2] 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, x)` *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, x)` 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);
}
```