Problema del Dizionario e Alberi Binari di Ricerca

Abbiamo visto nella lezione precedente che non possiamo implementare un dizionario efficiente utilizzando le normali rappresentazioni collegate e indicizzate.

E' possibile garanitre che tutte le operazioni su un dizionario di n elementi abbiano tempo $O(log(n))$.

> **Idea**
>
> Dobbiamo implementarlo tramite l'unione di Alberi di ricerca Binari e Alberi AVL

> **Alberi Binari di Ricerca**
>
> Un albero (binario) tale che ogni operazione richiede tempo $O(altezza \quad albero)$.

> **Alberi AVL**
>
> Alberi che hanno altezza sempre pari a $O(log(n))$.

Alberi Binari di Ricerca (BST)

Il nostro BST dovrà soddisfare le seguenti proprietà:

- Ogni nodo v contiene un elemento $elem(v)$ cui è associata una chiave, $chiave(v)$ presa da un **dominio totalmente ordinato.**

Per ogni nodo v vale che:

- Le chiavi del sottoalbero sinistro di v sono $\leq chiave(v)$
- Le chiavi del sottoalbero destro di v sono $> chiave(v)$

Quindi se saliamo il nostro albero dal basso verso l'alto:

- Versante sinistro -> **Ordinamento crescente**
- Versante destro -> **Ordinamento decrescente**

> **Massimo e Minimo**

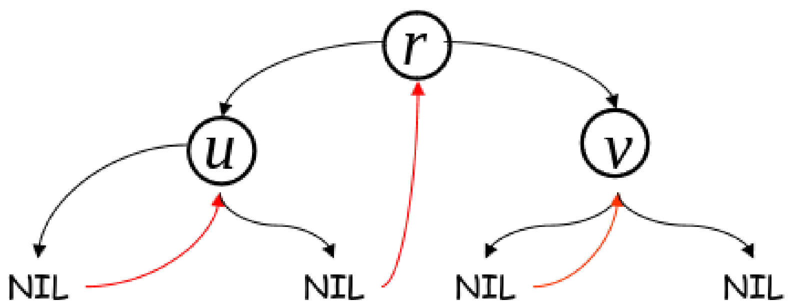
Il massimo dell'albero sarà l'elemento più a destra, mentre il minimo più a sinistra.

Simmetria

Se visitiamo un BTS in ordine simmetrico otteniamo i nodi ordinati in ordine crescente di chiave.

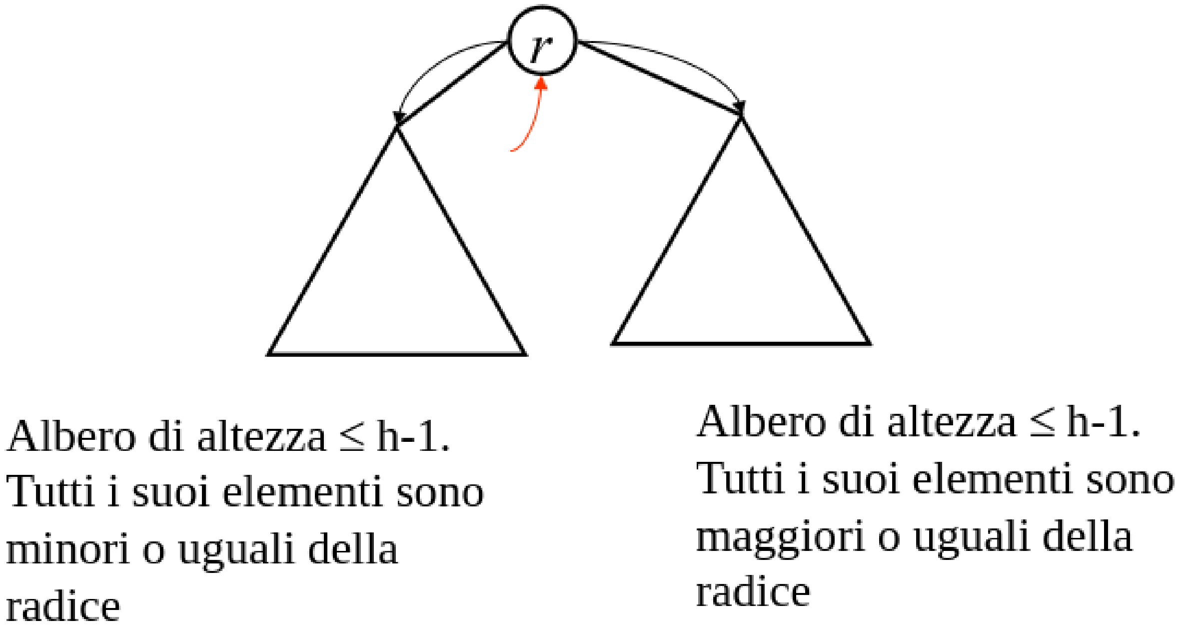
Correttezza della simmetria

La possiamo verificare per induzione sull'altezza dell'albero: $h = 1$.



$$\text{chiave}(u) \leq \text{chiave}(r) \leq \text{chiave}(v)$$

Adesso usiamo un h generico, ipotizzando che sia vero per ogni altezza minore di h:



Implementare le operazioni del dizionario su un BST

- **Search**: Traccia un cammino sull'albero partendo dalla radice: su ogni nodo, usa la proprietà di ricerca per decidere se proseguire nel sottoalbero sinistro o destro. La complessità della ricerca dipende dall'altezza dell'albero.

```
Algoritmo Search(chiave k) --> elem
        v <-- radice di T
        while(v != null) do
                if(k = chiave(v)) then return elem(v)
                else if (k < chiave(v)) then v <- figlio sinistro di v
                else v <- figli destro di v
        return null

O(Altezza dell' Albero).
```

- **Insert**: Aggiunge una chiave come nodo foglia, per capire dove mettere la chiave simuliamo una ricerca con la chiave da inserire.

```
Insert(elem e, chiave k)

1. Crea un nodo u cone leme = e e chiave = k;
2. Cerca la chiave k nell'albero, identificando il nodo v che è il padre di
u
```

```
3. Appendi u come figlio sinistro/destro di v in modo da mantenere le prop.

O(Altezza dell' Albero).
```

Operazioni ausiliare prima di delete

- **Ricerca del massimo**

```
Algoritmo max(nodo u) -> nodo
        v <- u
        while(figlio destro di v != null) do
                v <-figlio destro di v
        return v
```

- **Ricerca del minimo

```
Algoritmo min(nodo u) -> nodo
        v <- u
        while(figlio sinistro di v != null) do
                v <-figlio sinitro di v
        return v
```

Il predecessore di un nudo u in un BSR è il nodo v nell'albero avente massima chiave $\leq chiave(u)$.

Il successore di un nudo u in un BSR è il nodo v nell'albero avente minima chiave $\geq chiave(u)$.

- **Ricerca del predecessore**:

```
Algoritmo pred(nodo u) -> nodo
        if(u ha figlio sinistro sin(u)) then
                return max(sin(u))
        while(parent(u) != null e u è figlio sinistro di suo padre) do
                u <- parent(u)
        retutn parent(u)
```

- **Ricerca del successore:**

```
successore(nodo u) -> nodo
    if(u ha figlio destro) then
        return min(destro(u))  # Trova il minimo del sottoalbero destro
    while(parent(u) != null e u è figlio destro di suo padre) do
        u <- parent(u)
    return parent(u)
```

Operazioni di cancellazione

Definiamo l'operazione in questo modo: delete(elem e), dove e è l'elemento da cancellare.
Esistono tre casi possibili:

1. u da eliminare è una foglia -> **Rimuoviamola**
2. u ha un solo figlio -> **Rimuoviamo e attacchiamo** il figlio al predecessore.
3. u ha due figli ->Sostituisci con il predecessore/successore v e rimuovi fisicamente il predecessore/successore che ha al più un figlio.

Costo delle operazioni

Tutte le operazioni hanno un costo di $O(h)$, dove h è l'altezza dell'albero. Quindi i casi sono:

- **Alberi bilanciati** --> $h = O(log(n))$ --> Costo delle operazioni $O(log(n))$
- **Alberi sbilanciati** --> $h = O(n)$ --> Costo delle operazioni $O(n)$

Alberi AVL (Adel'son-Vel'skii e Landis)

Fattore di bilanciamento $\beta(v)$ di un nodo v = Altezza del sottoalbero sinistro di v - Altezza del sottoalbero destro di v.

Un albero è bilanciato in altezza se ogni nodo v ha fattore di bilanciamento in valore assoluto ≤ 1

Quindi un albero AVL è un albero BST bilanciato in altezza.

Possiamo dimostrare che un AVL con n nodi ha altezza $O(log(n))$.
L'idea della dimostrazione è quella di considerare i gli AVL più sbilanciati e far vedere che hanno $O(log(n))$ in altezza.

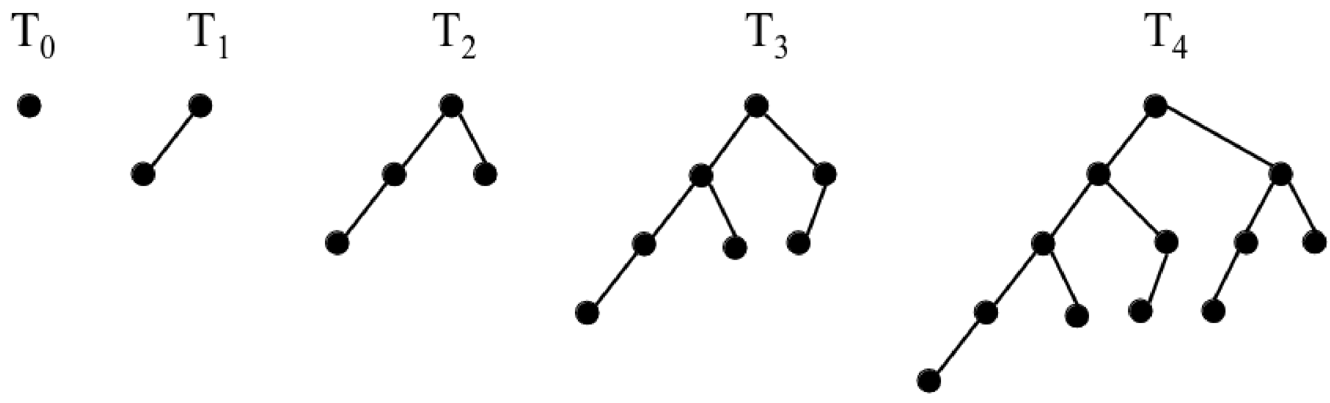
Albero di Fibonacci con altezza h

Albero AVL di altezza h con il minimo numero di nodi n_h.

Intuizione

Se gli alberi di Fibonacci hanno altezza $O(log(n))$ allora tutti gli AVL hanno altezza $O(log(n))$.

Quindi per creare questo albero dobbiamo minimizzare il numero di nodi fissata l'altezza o massimizzare l'altezza fissato il numero di nodi.



Nota che:

Se a T_i tolgo un nodo, o diventa sbilanciato, o cambia la sua altezza, inoltre ogni nodo (non foglia) ha fattore di bilanciamento pari (in valore assoluto) a 1.

Lemma

Sia n_h il numero di nodi di T_h, allora risulta che $n_h = F_{h+3} - 1$

La dimostrazione si puo fare per induzione su h.

Corollario

Un albero AVL con n nodi ha altezza $h = O(log(n))$

La dimostrazione di può fare risolvendo l'equazione di ricorrenza.

AVL per costruire un Dizionario

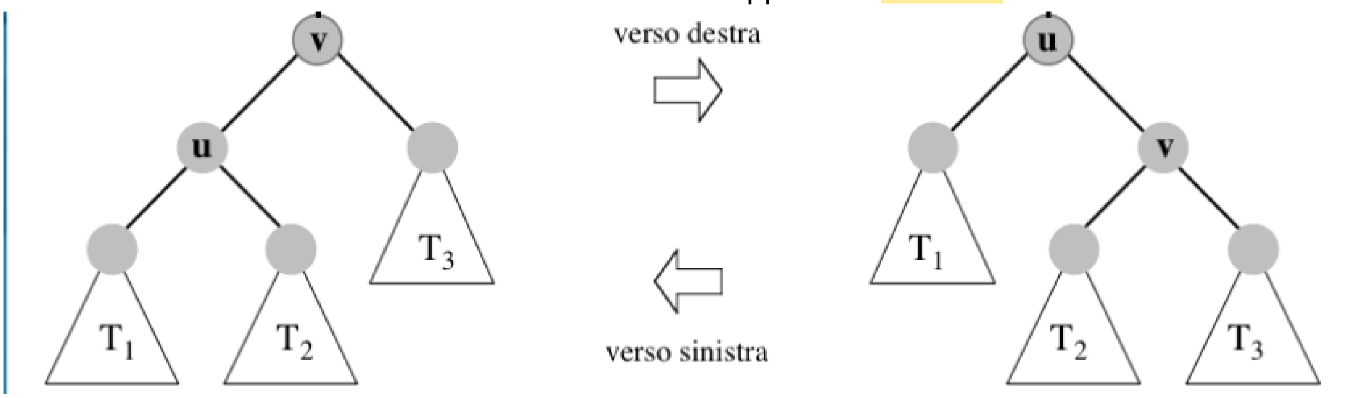
Di quanto e quali fattori di bilanciamento cambiano a fronte di un inserimento/cancellazione?

- **Quali:** Cambiano solo i fattori di bilanciamento dei nodi lungo il cammino radice-nodo inserito/cancellato.
- **Quanto:** I fattori di bilanciamento cambiano di +/- 1

Operazioni su AVL

L'operazione di Search avviene come nel BST, il problema sta nell'inserimento e nella cancellazione che potrebbero andare asbilanciare l'albero.

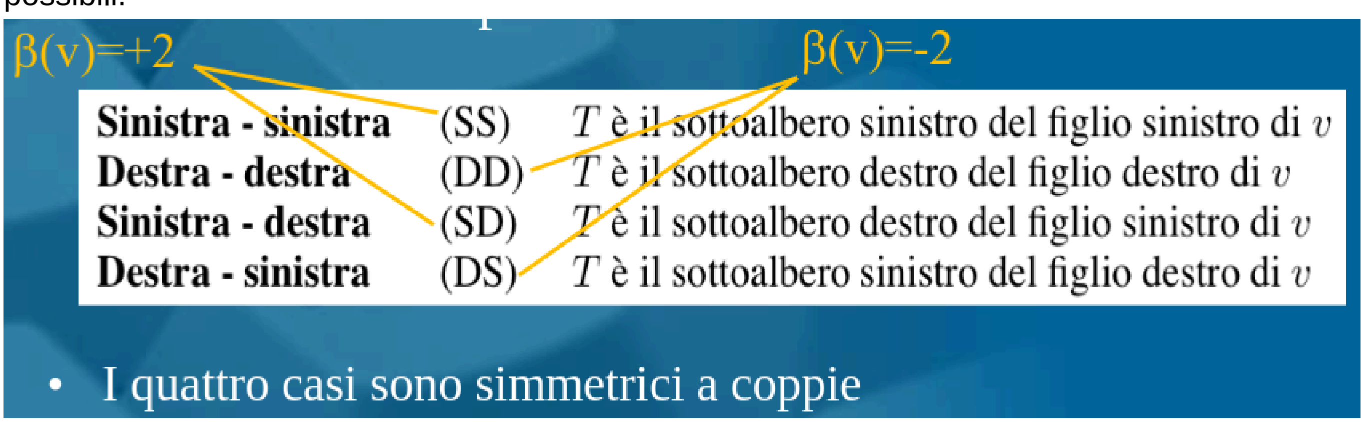
Per mantenere il bilnaciamento corretto utilizziamo opportune rotazioni.



Proprietà

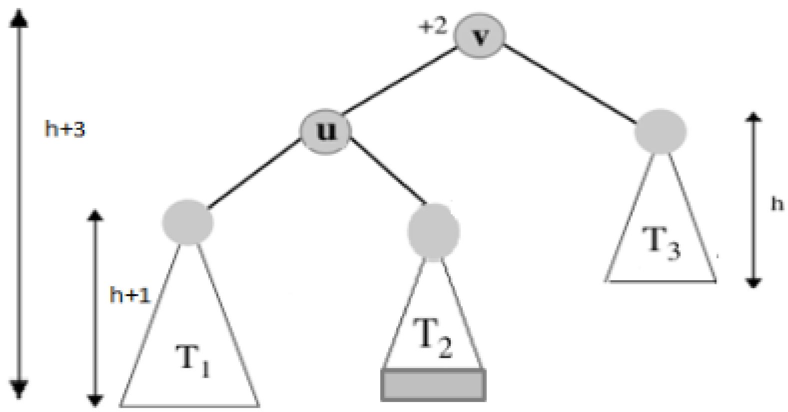
Mantiene la ricerca in modo corretto
Richiede tempo $O(1)$

Le rotazioni sono effettuate sui nodi sbilanciati, sia v un nodo di profondità con fattore di bilanciamento +2 o -2, allora esiste un sotooalvbero T di v che lo sbilancia. Abbiamo 4 cai possibili:



Caso SS ($\beta(v) = +2$ e $T_1 = h + 1$)

Sia h l'altezza del sottoalbero destro di v



Si seguito tutte le altezze:

- $T(v) = h + 3$
- $T(u) = h + 2$
- $T_3 = h$
- $T_1 = h + 1 \rightarrow \beta(v) = +2$ Quindi provoca lo sbilanciamento.

Applichiamo una rotazione semplice verso destra su v, ottenendo due sottocasi:

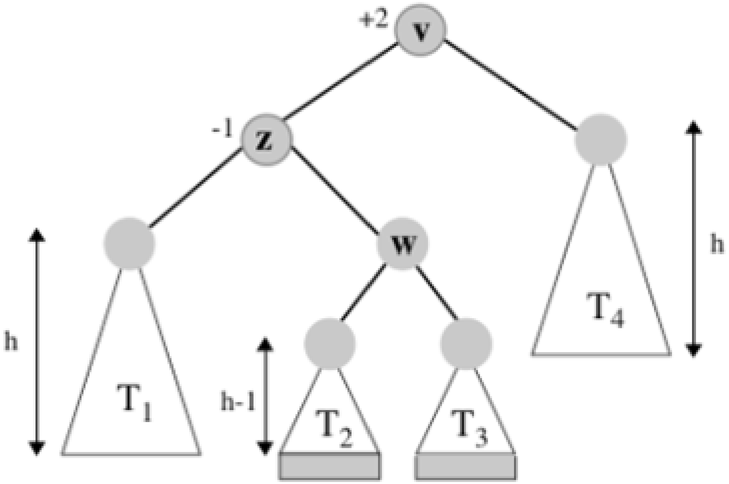
- Altezza di T_2 è h, quindi latezza del albero coinvolto passa da $h + 3$ a $h + 2$.
- Altezza di t_2 è $h + 1$, quindi l'altezza dell'albero coinvolto nella rotazione pari a $h + 3$.

Osservazioni sul caso SS

1. Dopo la rotazione l'albero è bilanciato
2. L'inserimento di un elemtno può provocare solo il primo sottocaso
3. La cancellazione di un elemento può provocare entrambi i casi
4. Nel primo caso l'albero diminuisce la sua altezza di uno

Caso DS ($\beta(v) = +2$ e $T_1 \neq h + 1$)

Sia h l'altezza del sottoalbero destro di v.



Quindi:

- Altezza di $T(w) = h + 1$
- Altezza di $T_1(w) = h + 1$

Lo sbilanciamento è provocato dal sottoalbero destro di z.
Applichiamo due rotazioni semplici: una verso sinistra del figlio sinistro del nodo critico (nodo z), l'altra verso destra sul nodo critico(nodo v).

Proprietà

1. L'altezza dell'albero dopo la rotazione passa da $h + 3$ a $h + 2$
2. Il caso SD può essere provocato da inserimenti o da cancellazioni.

insert (elem e, chiave k)

1. Crea un nodo u con $elem = e$ e $chiave = k$
2. Inserisci u come BST
3. Ricalcola i fattori di bilanciamento dei nodi nel cammino della radice a u
4. Esegui una rotazione su v (nodo critico).

Nodo critico

Il più profondo nodo con fattore di bilanciamento pari a +2 o -2.

delete(elem e)

1. Cancella il nodo come nei BST
2. Ricalcola i fattori di bilanciamento del padre del nodo eliminato fisicamente ed esegui le rotazioni necessarie.
3. Ripeti il passo fino ad arrivare alla radice

> **Osservazioni**
>
> Potrebbero essere necessarie $O(log(n))$ rotazioni, eventuali diminuzioni dell'altezza possono propagare lo sbilanciamento verso l'alto dell'albero.

Costo delle operazioni

Tutte le operazioni hanno costo $O(log(n))$, poichè l'altezza dell'albero è $O(log(n))$ e cisacuna rotazione richiede tempo costante.

> **Dettagli Importanti**
>
> 1. Dato un nodo v è possibile concoscere $\beta(v)$ in tempo $O(1)$
> 2. Dopo aver inserito/cancellato un nodo da un albero come se fosse un BST è possibile ricalcolare tutti i fattori di bilanciamento lungo il cammino alla radice in $O(log(n))$
> 3. Nell'eseguire le rotazioni dell'albero è possibile aggiornare anche i fattori di bilanciamento dei nodi coinvolti in tempo complessivo $O(log(n))$

```
classe AlberoAVL estende AlberoBinarioDiRicerca:

dati:                                                       S(n)=O(n)
        albero binario di ricerca T ereditato, più il fattore di
bilanciamento di   ogni nodo

Operazioni:

        search(chieve k) -> elem:   T(n) = O(log(n))
                ereditata

        insert(elem e, chiave k):   T(n) = O(log(n))
                chiama insert() ereditata, poi ricalcola i fattori di
bilanciamento ed eventualmente ribilancia tramite O(1) rotazioni
```

```
        delete(elem e)              T(n) = O(log(n))
                chiama delete() ereditata, poi ricalcola i fattori di
bilanciamento ed eventualmente ribilancia tramite O(log(n)) rotazioni
```