Alberi binari di ricerca

Pietro Di Lena

Dipartimento di Informatica – Scienza e Ingegneria
Università di Bologna

Algoritmi e Strutture di Dati
Anno Accademico 2022/2023

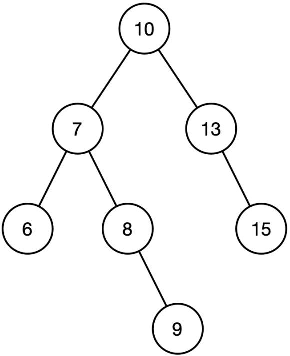
Introduzione

- Alberi Binari di Ricerca (BST, dall'inglese Binary Search Tree):
  - Alberi binari radicati con vincoli sull'organizzazione delle chiavi
  - Permette una ricerca binaria sulla struttura Albero Binario
- Le operazioni hanno un costo proporzionale all'altezza dell'albero
- Vedremo come implementare le operazioni basilari di ricerca, inserimento e rimozione su un Albero Binario di Ricerca

Alberi Binari di Ricerca (BST)

- Idea: portare la ricerca binaria sulla struttura dati Albero Binario

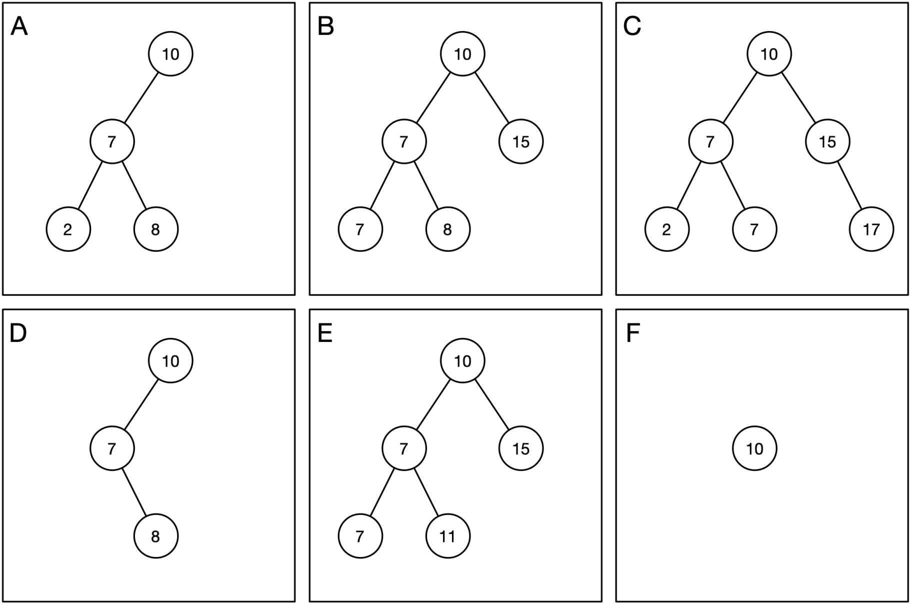
- Definizione di Albero Binario di Ricerca:
  1. Albero Binario
  2. Ogni nodo v contiene una chiave confrontabile $v.key$ e dati $v.data$ associati alla chiave
  3. Proprietà di ordinamento dei BST: tutte le chiavi nel sottoalbero sinistro di v sono $\leq v.key$ e tutte le chiavi nel sottoalbero destro di v sono $\geq v.key$



- La proprietà 3 permette di effettuare una ricerca binaria sull'albero

- Quale visita permette di ottenere tutte le chiavi in ordine?

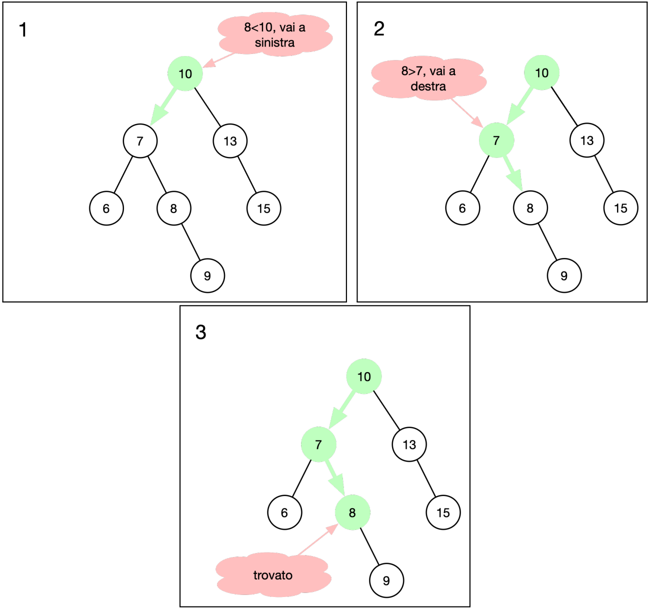
Alberi Binari di Ricerca?



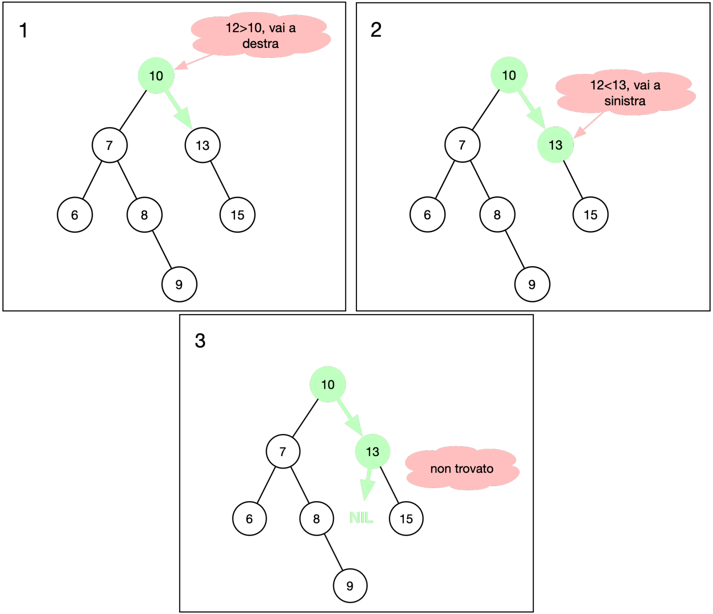
Operazioni su Alberi Binari di Ricerca

- Operazioni su Alberi Binari di Ricerca
  - SEARCH(T,k): ritorna il nodo con chiave k in T
    - NIL se k non appare in T
  - MAX(T): ritorna il nodo con chiave massima k in T
  - MIN(T): ritorna il nodo con chiave minima k in T
  - PREDECESSOR(T): ritorna il nodo che precede T quando i nodi sono ordinati rispetto ad una visita in-ordine
    - Se le chiavi sono tutte distinte, è equivalente al nodo avente la più grande chiave $k < T.key$
    - NIL se $T.key$ è la chiave minima in T
  - SUCCESSOR(T): simmetrica a PREDECESSOR
  - INSERT(T, k, d): inserisce un nodo con chiave k e dati d in T
  - DELETE(T, k): rimuove il nodo con chiave k in T

Esempio: ricerca del numero 8



Esempio: ricerca del numero 12

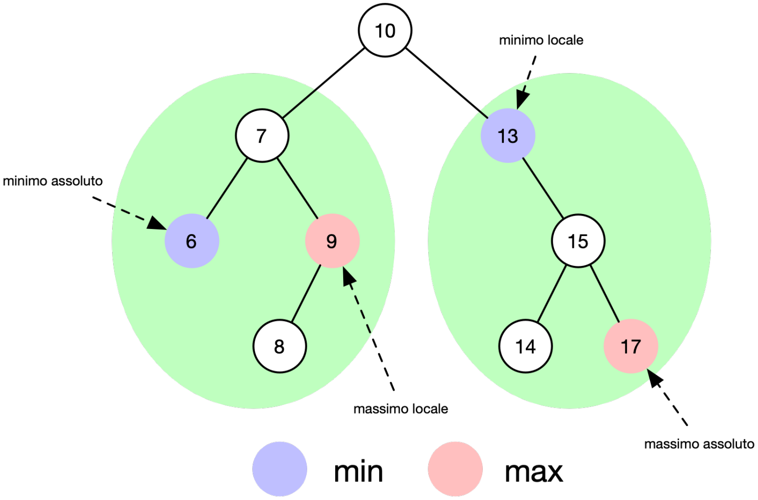


PSEUDOCODICE: SEARCH

```
1: function SEARCH(BST T, KEY k) → NODE
2:     tmp = T.root
3:     while tmp ≠ NIL do
4:         if k == tmp.key then
5:             return tmp
6:         else if k < tmp.key then
7:             tmp = tmp.left
8:         else
9:             tmp = tmp.right
10:    return NIL
```

- Ritorna la prima occorrenza della chiave k
- Costo nel caso ottimo: $O(1)$
  - Quando $k == T.key$ (linea 3)
- Costo nel caso pessimo: $O(h)$
  - $h =$ altezza dell'albero
  - La visita è sempre confinata su un percorso radice-foglia
  - N.B. $h = O(n)$, $n =$ numero di nodi in T

Esempio: max e min



Pseudocodice: max e min

```
function MAX(NODE T) → NODE
    while T ≠ NIL and T.right ≠ NIL do
        T = T.right
    return T
```

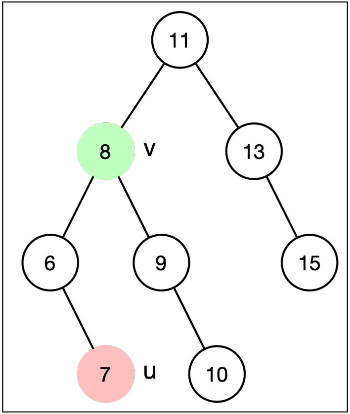
```
function MIN(NODE T) → NODE
    while T ≠ NIL and T.left ≠ NIL do
        T = T.left
    return T
```

- Dato un sottoalbero T
  - il nodo massimo in T è il nodo più a destra in T
  - il nodo minimo in T è il nodo più a sinistra in T
- Stesso costo per entrambe le funzioni:
  - Caso ottimo: $O(1)$ (T non ha figlio destro (MAX) o sinistro (MIN))
  - Caso pessimo: $O(h)$
- N.B. Non confrontiamo chiavi, usiamo solo la struttura dell'albero

Esempio: predecessore (caso 1)

Definizione: il predecessore di un nodo v è il nodo u che precede v quando i nodi sono ordinati rispetto ad una visita in-ordine

- Caso 1
  - Il nodo v ha un figlio sinistro
  - Il predecessore è il nodo u con chiave massima nel sottoalbero sinistro di v

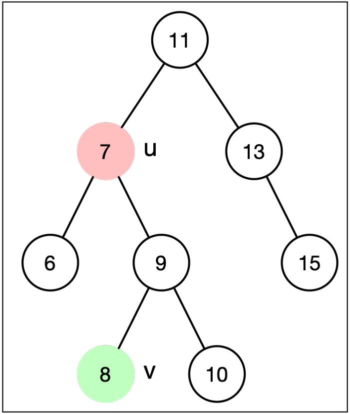


- Correttezza
  - Per la propietà di ordine dei BST, il sottoalbero sinistro di v contiene solo chavi $\leq v.key$

Esempio: predecessore (caso 2)

Definizione: il predecessore di un nodo v è il nodo u che precede v quando i nodi sono ordinati rispetto ad una visita in-ordine

- Caso 2
  - Il nodo v non ha un figlio sinistro
  - Il predecessore è il primo antenato u tale che v stia nel sottoalbero destro di u
- Correttezza
  - Per la propietà di ordine dei BST, il nodo v è il nodo minimo nel sottoalbero destro di u



Pseudocodice: predecessor

```
function PREDECESSOR(Node T) → Node
    if T == NIL then                              ▷ Empty tree
        return NIL
    else if T.left ≠ NIL then                     ▷ Case 1
        return MAX(T.left)
    else                                          ▷ Case 2
        P = T.parent
        while P ≠ NIL and  T == P.left do
            T = P
            P = P.parent
        return P
```

- Caso pessimo: O(h)
  - Caso 1 (MAX): O(h)
  - Caso 2: O(h)
- Caso ottimo: O(1)
  - Caso 1 (MAX): O(1)
  - Caso 2: O(1)

N.B. Non confrontiamo chiavi, usiamo solo la struttura dell'albero

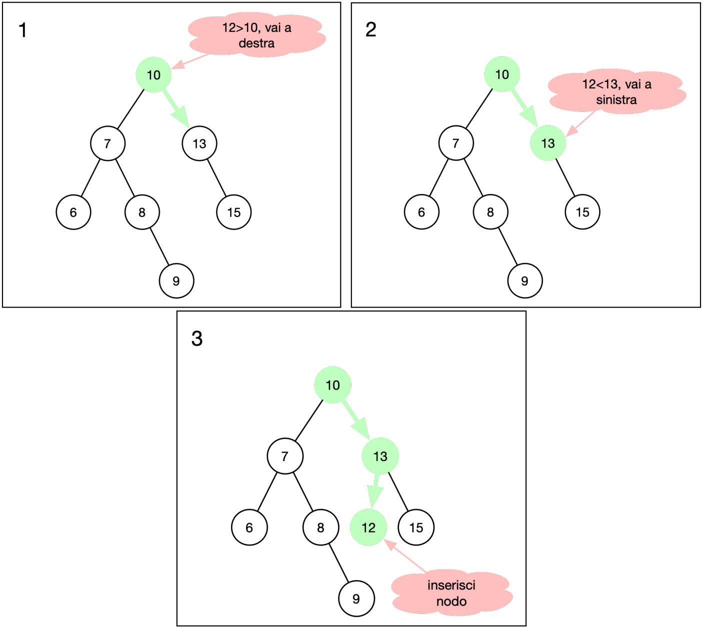
Pseudocodice: successor

successor è simmetrica a predecessor

```
function successor(Node T) → Node
    if T == NIL then                                  ▷ Empty tree
        return NIL
    else if T.right ≠ NIL then                        ▷ Case 1
        return min(T.right)
    else                                              ▷ Case 2
        P = T.parent
        while P ≠ NIL and T == P.right do
            T = P
            P = P.parent
        return P
```

- Caso pessimo: O(h)
    - Caso 1 (min): O(h)
    - Caso 2: O(h)
- Caso ottimo: O(1)
    - Caso 1 (min): O(1)
    - Caso 2: O(1)

ESEMPIO: INSERIMENTO



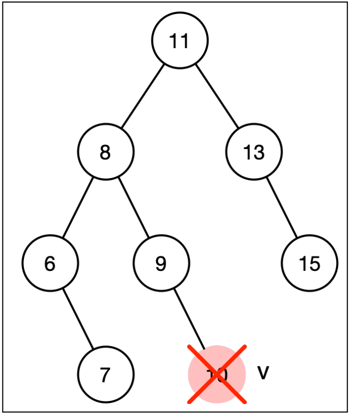
PSEUDOCODICE: INSERT

```
function INSERT(BST T, KEY k, DATA d)
    N = new NODE(k, d), P = NIL, S = T.root
    while S ≠ NIL do                                  ▷ Search position
        P = S
        if k < S.key then
            S = S.left
        else
            S = S.right
    if P == NIL then                                  ▷ Insert node
        T.root = N                                    ▷ The tree was empty
    else
        N.parent = P
        if k < P.key then P.left = N
        else P.right = N
```

- Caso pessimo: O(h)
  - Ricerca posizione: O(h)
  - Inserimento nodo: O(1)
- Caso ottimo: O(1)
  - Ricerca posizione: O(1)
  - Inserimento nodo: O(1)

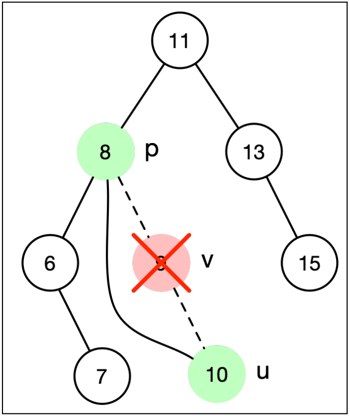
Esempio: rimozione (caso 1)

- Caso 1
  - Il nodo v da rimuovere è una foglia
  - Semplicemente rimuoviamo v
- Correttezza
  - Se rimuoviamo una foglia non alteriamo la proprietà di ordine dei BST nei nodi rimanenti



Esempio: rimozione (caso 2)

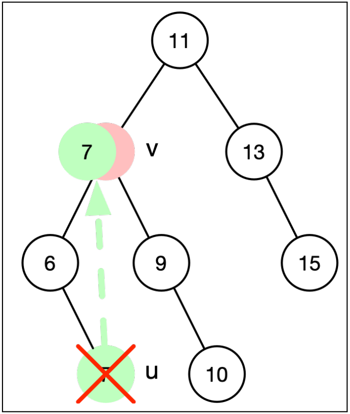
- Caso 2
  - Il nodo da rimuovere v ha un solo figlio u
  - u diventa figlio del genitore p di v
  - Se v è un figlio sinistro, u diventa figlio sinistro di p
  - Se v è un figlio destro, u diventa figlio destro di p
  - Possiamo rimuovere v



- Correttezza
  - Per la propietà di ordine dei BST, se v è un figlio destro tutte le chiavi nel sottoalbero radicato in u sono $\geq p.key$ e se v è un figlio sinistro tutte le chiavi in u sono $\leq p.key$

Esempio: delete (caso 3)

- Caso 3
  - Il nodo da rimuovere v ha due figli
  - Cerchiamo il predecessore u di v
  - Copiamo $v.key = u.key$ (e dati)
  - Il nodo u ha al massimo un figlio (?)
  - Rimuoviamo il nodo u (Caso 1 o 2)



- Correttezza
  - Per la propietà di ordine dei BST, la chiave $u.key$ del predecessore di v è $\geq$ di tutte le chiavi in $v.left$ e $\leq$ di tutte le chiavi in in $v.right$. Possiamo quindi sostituire v con u senza alterare la proprietà di ordine dei BST

Pseudocodice: delete

```
function delete(BST T, Key k)
    v = search(T, k)
    if v ≠ NIL then
        if v.left == NIL or v.right == NIL then                    ▷ Case 1 or 2
            deleteNode(T, v)
        else                                                       ▷ Case 3
            u = predecessor(v)
            v.key = u.key
            v.data = u.data
            deleteNode(T, u)

function deleteNode(BST T, Node v)
    p = v.parent
    if p ≠ NIL then                                                ▷ v is not the root node
        if v.left ==NIL and v.right ==NIL then                     ▷ Case 1
            if p.left == v then p.left = NIL else p.right = NIL
        else if v.right ≠ NIL then                                 ▷ Case 2
            if p.left == v then p.left = v.right else p.right = v.right
        else if v.left ≠ NIL then                                  ▷ Case 2
            if p.left == v then p.left = v.left else p.right = v.left
    else                                                           ▷ v is the root node
        else if v.right ≠ NIL then T.root = v.right                ▷ Case 2
        else T.root = v.left                                       ▷ Case 1 or Case 2
```

Analisi di delete

- Costo computazionale nel caso pessimo: $O(h)$
  - funzione SEARCH : $O(h)$
  - funzione DELETENODE : $O(1)$
  - funzione PREDECESSOR : $O(h)$

- Costo computazionale nel caso ottimo: $O(1)$
  - funzione SEARCH: $O(1)$
  - funzione DELETENODE: $O(1)$
  - funzione PREDECESSOR: $O(1)$

- N.B. DELETENODE gestisce solo i Casi 1 e 2, che richiedono un numero costante di operazioni

Analisi del caso medio

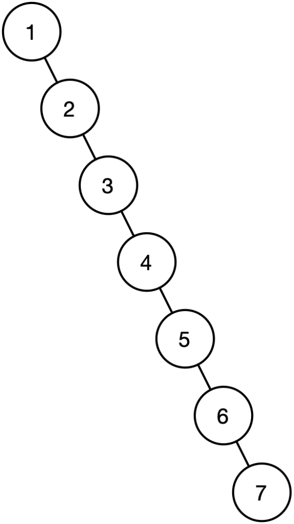
- Nel caso pessimo tutte le operazioni su BST hanno costo $O(h)$
  - Il costo medio dipende dall'altezza media $\overline{h}$ di un BST

- L'altezza h di un BST può variare di molto
  - L'altezza massima di un BST con n nodi è $h = \Theta(n)$
    - Un BST con altezza h ha almeno $h + 1$ nodi
    - Quando l'albero binario è una lista
  - L'altezza minima di un BST con n nodi è $h = \Theta(\log n)$
    - Un BST con altezza h ha al massimo $2^{h+1} - 1$ nodi
    - Quando l'albero binario è *perfetto*

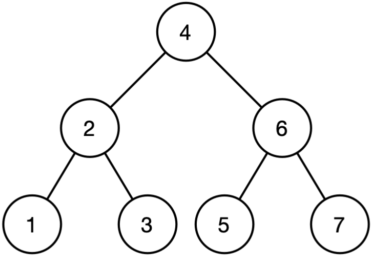
- Qual è l'altezza media $\overline{h}$ di un BST?
  - Caso generale (inserimenti e rimozioni)
    - Difficile da analizzare
  - Caso *"facile"*: BST costruito da n inserimenti *casuali*
    - E' possibile dimostrare che $\overline{h} = O(\log n)$

Altezza massima e minima di un BST

$n=7 \quad h=n-1=6$



$n=7 \quad h=\log_2(n+1)-1 = 2$



Ripasso: struttura dati Dizionario

- Struttura dati generica per memorizzare oggetti
  - Contiene un insieme di chiavi univoche
  - Ogni chiave è associata ad un valore
  - I valori posso essere duplicati, le chiavi sono uniche

- Operazioni basilari di un Dizionario (prototipo):
  - SEARCH(Key k): cerca l'oggetto associato alla chiave k
  - INSERT(Key k, Data d): aggiunge la coppia (k, d) al Dizionario
  - DELETE(Key k): elmina la coppia (k, d) dal Dizionario

- Possibili implementazioni (costo nel caso pessimo):
  - Array non ordinato: SEARCH/INSERT in $O(n)$, DELETE in $\Theta(n)$
  - Array ordinato: SEARCH in $O(\log n)$, INSERT/DELETE in $O(n)$
  - Lista concatenata: SEARCH/INSERTE/DELETE in $O(n)$

Dizionario con Alberi binari di ricerca

```
function SEARCH(Dict D, Key k) → Data
    tmp = BSTSEARCH(D.BST, k)                ▷ search on BST
    if tmp == NIL then
        return NIL
    else
        return tmp.data

function DELETE(Dict D, Key k)
    BSTDELETE(D.BST, k)                      ▷ delete on BST
```

- *D.BST* è una struttura dati di tipo Albero Binario di Ricerca
- Per implementare la funzione di inserimento potremmo:
  1. Cercare la chiave con BSTSEARCH
  2. Se la chiave esiste, sostituiamo i dati
  3. Altrimenti, richiamiamo la funzione di inserimento su BST
- In questo modo, se la chiave non esiste, scorriamo due volte l'albero
- Possiamo fare tutto in una passata (ricerca e inserimento):
  - Stesso costo computazionale ma più efficiente in pratica

Dizionario con Alberi binari di ricerca

```
function INSERT(DICT D, KEY k, DATA d)
    P = NIL, S = D.BST.root
    while S ≠ NIL and S.key ≠ k do          ▷ Search position
        P = S
        if k < S.key then S = S.left
        else S = S.right
    if S ≠ NIL then                          ▷ Here S.key = k
        S.data = d
    else
        N = NEW NODE(k, d)
        if P == NIL then                     ▷ Insert node
            D.DST.root = N                   ▷ The tree was empty
        else
            N.parent = P
            if k < P.key then P.left = N
            else P.right = N
```

- Semplice modifica della funzione di inserimento su BST
- Cerchiamo in una sola passata il nodo con chiave k oppure la sua posizione di inserimento

Dizionario: riassunto dei costi

	SEARCH		INSERT		DELETE	
	Medio	Pessimo	Medio	Pessimo	Medio	Pessimo
Array non ordinati	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$\Theta(n)$	$\Theta(n)$
Array ordinati	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Lista concatenata	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Albero Binario di Ricerca	$O(\overline{h})$	$O(h)$	$O(\overline{h})$	$O(h)$	$O(\overline{h})$	$O(h)$

- $h =$ altezza dell'albero, $\overline{h} =$ altezza media dell'albero
  - $h = \Omega(\log n)$ e $h = O(n)$
- Non abbiamo nessun vantaggio se l'albero è sbilanciato e $h = \Theta(n)$
- Tutte le operazioni hanno un costo logaritimico se $h = \Theta(\log n)$
- Come possiamo fare in modo che l'altezza dell'albero sia sempre logaritimica sul numero di nodi?