Alberi

Un albero è un insieme di oggetti, chiamati nodi, su cui è definita una relazione binaria $G(x, y)$ – che leggiamo "x è genitore di y" – tale che:

1. esiste un unico nodo, chiamato radice, che non ha genitori;
2. ogni nodo x diverso dalla radice ha uno ed unico genitore;
3. per ogni nodo x diverso dalla radice esiste un cammino dalla radice a x (l'albero è *connesso*): esistono nodi $x_1, .., x_k$ $(k \geq 1)$ tali che x_1 è la radice dell'albero, $x_k = x$ e per ogni $i = 1, ..., k-1$ x_i è genitore di x_{i+1}.

- Se x è il genitore di y, allora y è un figlio di x.

- Un albero binario è un albero in cui ogni nodo ha al massimo due figli.

Operazioni di base

- Empty(): restituisce una struttura rappresentante l'albero vuoto
- Null(T): test albero vuoto.
- Root(T): restituisce il puntatore alla radice dell'albero; un errore se T è vuoto.
- Left(T): restituisce il sottoalbero sinistro di T; un errore se T è vuoto.
- Right(T): restituisce il sottoalbero destro di T; un errore se T è vuoto.
- Left-Child(T): restituisce il puntatore al figlio sinistro; un errore se T è vuoto.
- Right-Child(T): restituisce il puntatore al figlio destro; un errore se T è vuoto.
- Mktree(x,T1,T2): restituisce l'albero con radice x e sottoalberi T1 e T2.

Rappresentazione di alberi binari mediante puntatori

Gli oggetti x rappresentanti i nodi dell'albero hanno tre campi

$key[x]$: il valore memorizzato nel nodo.

$left[x]$: puntatore al figlio sinistro (NIL se x non ha figlio sinistro).

$right[x]$: puntatore al figlio destro (NIL se x non ha figlio destro).

La rappresentazione T di un albero è un oggetto con un campo

$root[T]$: puntatore alla radice dell'albero; NIL se l'albero è vuoto.

Alberi binari di ricerca (ABR)

Struttura dati per implementare dizionari: insieme di elementi ordinati secondo una determinata *chiave*, sul quale è possibile effettuare le operazioni di

1. ricerca di un elemento (Search)
2. inserimento di nuovi elementi nell'insieme (Insert)
3. cancellazione di elementi dall'insieme (Delete)

Le operazioni di base su un ABR richiedono tempo proporzionale all'altezza dell'albero, quindi, se n è il numero dei nodi dell'albero:

$\Theta(lg\, n)$ nel caso migliore (albero completo)

$\Theta(n)$ nel caso peggiore (albero completamente sbilanciato)

$\Theta(lg\, n)$ nel caso medio

Gli elementi dell'insieme sono strutture con un campo key: la chiave dell'elemento
(oltre ai campi che contengono riferimenti)

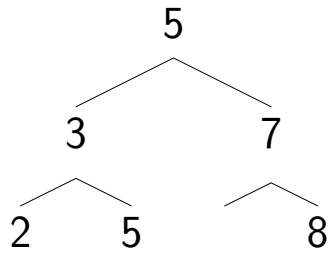
Sull'insieme delle chiavi è definito un ordine totale

Proprietà degli ABR

Per ogni nodo x dell'albero

- tutti i nodi del *sottoalbero sinistro* di x hanno *chiave minore (o uguale)* di quella di x
- tutti i nodi del *sottoalbero destro* di x hanno *chiave maggiore (o uguale)* di quella di x

Esempio



Rappresentazione

Un **nodo** x *di un ABR è una struttura con i campi:*

$key[x]$ la chiave dell'elemento

$left[x]$ puntatore al figlio sinistro

$right[x]$ puntatore al figlio destro

$p[x]$ puntatore al padre

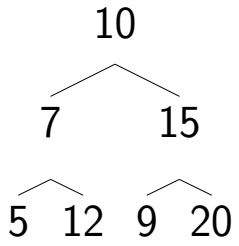
Stampa delle chiavi in modo ordinato - visita simmetrica

```
INORDER-TREE-WALK(x)
1. if x != NIL
2.    then INORDER-TREE-WALK(left[x])
3.         stampa key[x]
4          INORDER-TREE-WALK(right[x])
```

Prima chiamata: `INORDER-TREE-WALK(root[T])`

Esercizi

1. Disegnare ABR di altezza $2, 3, 4, 5, 6$ sull'insieme di chiavi $\{1, 4, 5, 10, 16, 17, 21\}$
2. L'albero riportato sotto rispetta la proprietà ABR?



3. Qual è la differenza tra la proprietà degli ABR e la proprietà di ordinamento dello heap?
4. Descrivere un algoritmo che, dato un albero binario T, verifichi se T rispetta la proprietà ABR.
5. Si supponga che la ricerca di una chiave termini in una foglia. Si considerino:

   $\mathbf{A}$ = l'insieme delle chiavi a destra del cammino di ricerca;

   $\mathbf{B}$ = l'insieme delle chiavi sul cammino di ricerca;

   $\mathbf{C}$ = l'insieme delle chiavi a destra del cammino di ricerca.

   AFFERAMZIONE: $\forall a, b, c, |a \in A, b \in B, c \in C$ si ha $a \leq b \leq c$
   Dare il controesempio più piccolo a questa affermazione.

Verifica della proprietà ABR per un dato albero T

applicando la definizione di ABR

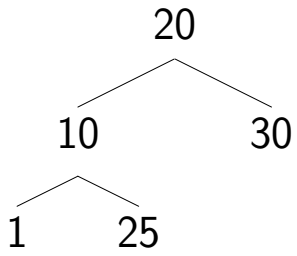
```
;input:  un albero binario con radice in t
;output: vero se t e' un abr, falso altrimenti
;
;la funzione viene richiamata con la radice dell'albero di cui si vuole
;verificare la proprieta' abr.

abr(t)
        if t=NIL
                then
                        return true
                else
                         return (       (key[t] >= key[left[t]])
                                    and (key[t] <= key[right[t]])
                                    and abr(left[t])
                                    and abr(right[t])
                                )
```

Abbiamo applicato la definizione correttamente?

L'algoritmo precedente NON verifica la proprietà ABR

Controesempio



Fissato un nodo radice i controlli devono essere eseguiti su tutti i nodi del s.a. sinistro e tutti i nodi del s.a. destro

Soluzione

Al posto di `(key[t] >= key[left[t]])` chiamiamo una funzione che verifica se **tutti** i nodi del s.a. sinistro sono minori o uguali della radice
Al posto di `(key[t] <= key[right[t]])` chiamiamo una funzione che verifica se **tutti** i nodi del s.a. destro sono maggiori o uguali della radice

```
abr(t)
        if t=NIL
                then return true
                else return (          minore(left[t],t)
                              and maggiore(right[t],t)
                              and abr(left[t])
                              and abr(right[t])
                             )
```

Le funzioni minore e maggiore

```
; r e' il nodo padre, t e' il generico puntatore al nodo dell'albero minore(t,r)


minore(t,r)
if t=NIL then return true
         else return((key[t]<=key[r]) and
                      minore(left[t],r) and
                      minore(right[t],r)
                     )


maggiore(t,r)
if t=NIL then return true
         else return((key[t]>=key[r]) and
                      maggiore(left[t],r) and
                      maggiore(right[t],r)
                     )
```

la complessità asintotica è O(n log n) nel caso migliore

- sia n il numero di nodi dell'albero di cui vogliamo verificare la proprietà ABR;
- alla prima attivazione la funzione `abr` richiama `maggiore` e `minore` su un numero di nodi uguale a $(n-1)/2$. Poiché la complessità della visita di un albero di n nodi è O(n) si ha: $T_{maggiore}((n-1)/2) + T_{minore}((n-1)/2) = O(n)$
- la funzione `abr` viene chiamata ricorsivamente 2 volte: una sul s.a. sinitro e una sul s.a. destro.
- Analisi delle diverse configurazioni dell'albero:
  - Caso migliore: l'albero è bilanciato

$$T_{abr}(n) = \begin{cases} c_0 & \text{se } n = 0 \\ 2\,T(n/2) + O(n) & \text{se } n > 0 \end{cases}$$

    Quindi siamo nel II caso del Master Theorem e $T_{abr}(n) = O(n\ log\ n)$.

  - Caso peggiore: l'albero è completamente sbilanciato

$$T_{abr}(n) = \begin{cases} c_0 & \text{se } n = 0 \\ T(n-1) + O(n) & \text{se } n > 0 \end{cases}$$

    Quindi $T_{abr}(n) = O(n^2)$.

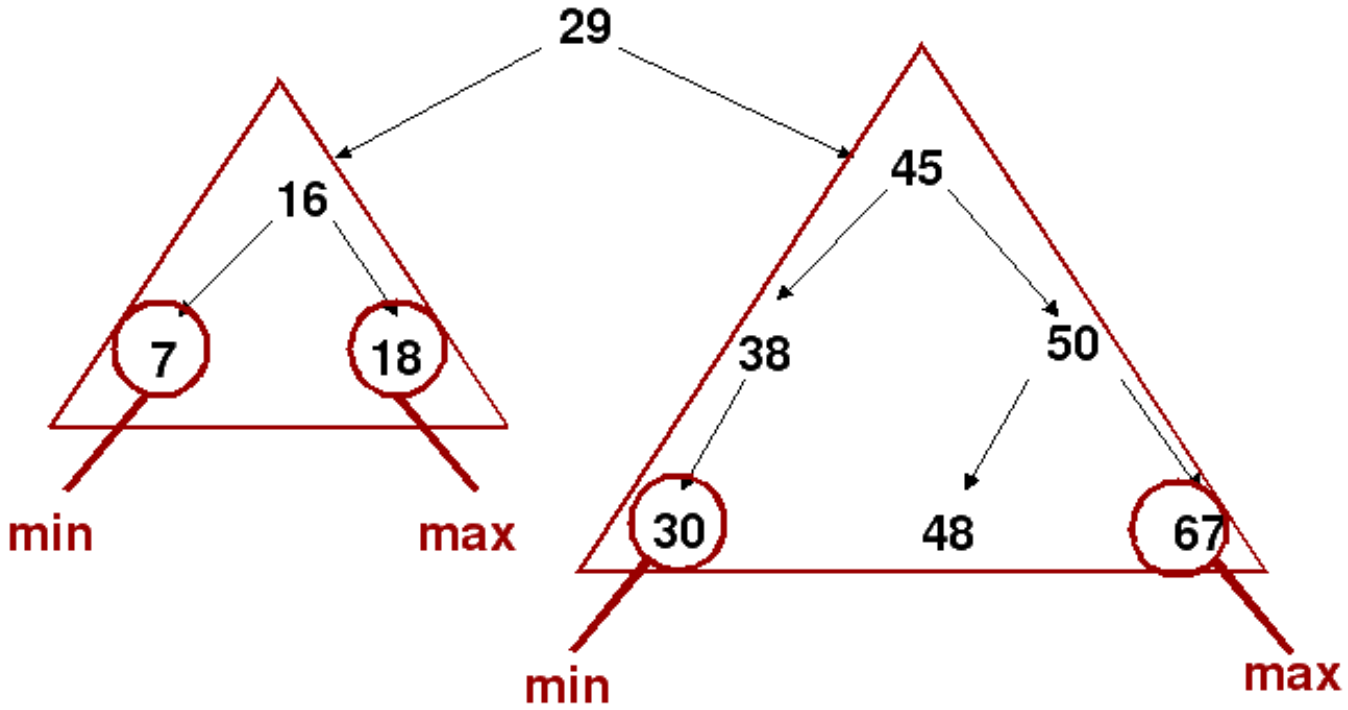
Si può migliorare la complessità asintotica?

Per migliorare la complessità asintotica di un algoritmo servono considerazioni più approfondite, stratagemmi (leciti), uso di strutture ausiliarie ...

Analisi della struttura ABR

Se i sottoalberi più profondi sono ABR il massimo e il minimo si trovano rispettivamente nel ramo più a destra e nel ramo più a sinistra.

Esempio:



Nuova proprietà

Per ogni nodo non foglia la chiave del nodo è compresa tra il massimo del s.a. sinistro e il minimo del s.a. destro.

Algoritmo più furbo

si parte dal basso verificando che i sottoalberi a profondità maggiore siano ABR
serve una visita in postordine dell'albero che restituisce una coppia (min,max): massimo e minimo del sottoalbero esaminato

Passi base:

Un ABR nullo fornisce una segnalazione d'errore.

Un ABR costituito da un solo elemento è un ABR (min=max): viene restituita la coppia (min,max) con min max = key[t].

Passo ricorsivo: Sia t il puntatore al nodo in esame:

Se t ha solo il figlio sx viene richiamata la funzione solo su left[t] che restituisce la coppia (minl,key[t]); key[t] è il massimo perché manca il s.a. destro;

Se t ha solo il filgio dx viene richiamata la funzione solo su right[t] che restituisce la coppia (key[t],maxr); key[t] è il minimo perché manca il s.a. sinistro;

Se l'ABR ha entrambi i figli verifico che il nodo padre sia maggiore (o uguale) del massimo del s.a. sinistro e minore (o uguale) del minimo sul s.a. destro.

La complessità asintotica è O(n) (dimostrare)

Pseudocodifica

```
;se l'albero e' un ABR viene restituita la coppia con le chiavi minima
;e massima, se e' vuoto o se e' un ABR viene sollevata un'eccezione

abr2(t)
if t = NIL
        then errore "EmptyTree"
        else if foglia(t)
                then return (key[t],key[t])
                else if right[t]=NIL
                       then (minl,maxl)<-abr2(left[t]);
                            if (maxl>key[t]) then errore "NotAbr"
                                            else return (minl,key[t])
                       else if left[t]=NIL
                            then (minr,maxr)<-abr2(right[t]);
                                 if (minr<key[t]) then errore "NotAbr"
                                                  else return (key[t],maxr)
                            else ;due figli
                                (minl,maxl) <- abr2(left[t]);
                                (minr,maxr) <- abr2(right[t]);
                                if (maxl>key[t] or minr <key[t])
                                  then errore "NotAbr"
                                  else return (minl,maxr)
```

Ricerca di un elemento in un ABR

Tree-Search: ricerca un elemento con chiave k in un ABR T

Input: la chiave k, un *puntatore* x *alla radice* dell'albero $T(root[T])$;

Output: un puntatore ad un nodo con chiave k, se esiste, NIL altrimenti.

Ricerca di un elemento in un ABR - versione iterativa

```
ITERATIVE-TREE-SEARCH(x,k)
1. while x != NIL and k != key[x]
2.   do if k < key[x]
3.         then x <- left[x]
4.         else x <- right[x]
5. return x
```

Ricerca di un elemento in un ABR - versione ricorsiva

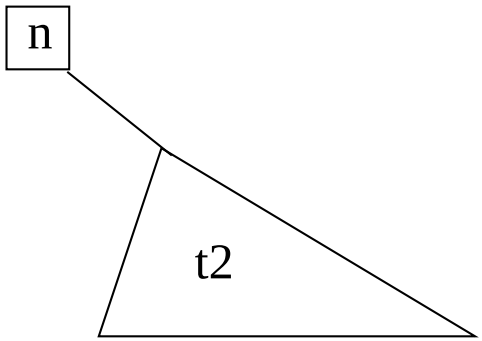
Base Se $x = NIL$, riportare NIL.
Altrimenti, se la chiave di x è uguale a k, riportare x.
(In entrambi i casi si riporta x).

Ricorsione ($x \neq NIL$ e k non è uguale alla chiave di x).
Se $k < key[x]$, ricercare l'elemento con chiave k nel sottoalbero sinistro della radice. Altrimenti, se $k \geq key[x]$, ricercare nel sottoalbero destro.

```
TREE-SEARCH(x,k)
1. if x = NIL or k = key[x]
2.    then return x
3. if k < key[x]
4.    then return TREE-SEARCH(left[x],k)
5.    else return TREE-SEARCH(right[x],k)
```

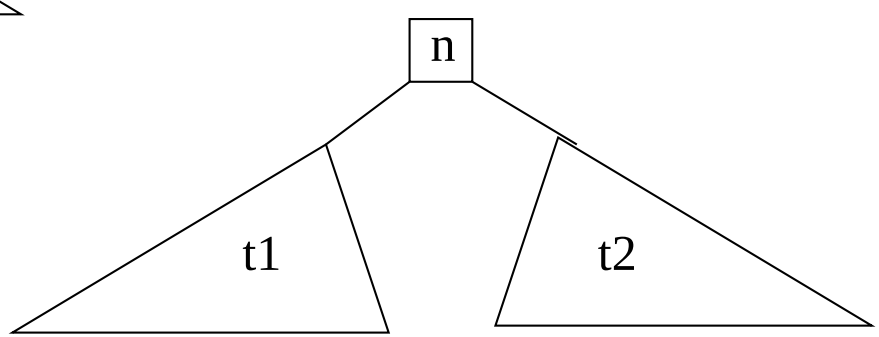
Minimo

Minimo: nodo più a sinistra senza figlio sinistro (si trova percorrendo l'albero seguendo i puntatori ai figli sinistri, finché ci sono).



ogni nodo in t2
e' maggiore di n

n e' il minimo



in t1 i nodi sono
minori di n, quindi
n non e' il minimo

il minimo va cercato in t1

Tree-Minimum, applicata a un puntatore alla radice di un ABR non vuoto, riporta un puntatore al minimo elemento dell'ABR.

```
TREE-MINIMUM(x)
1. while left[x] != NIL
2.    do x <- left[x]
3. return x
```

Massimo

Massimo: nodo più a destra senza figlio destro.

Tree-Maximum, applicata a un puntatore alla radice di un ABR non vuoto, riporta un puntatore al massimo elemento dell'ABR.

```
TREE-MAXIMUM(x)
1. while right[x] != NIL
2.    do x <- right[x]
3. return x
```

Se la dimensione dell'input è l'altezza h dell'albero

- *Caso peggiore*: la ricerca prosegue sul sottoalbero di profondità massima

$$\begin{aligned} T_{TREE-SEARCH}(0) &= a \\ T_{TREE-SEARCH}(h) &= T_{TREE-SEARCH}(h-1) + c \end{aligned}$$

$$T(h) \in O(h)$$

Per il primo teorema generale per risolvere equazioni di ricorrenza:
Se

$$T(n) = \begin{cases} a & \text{se } n = 0 \\ T(n-1) + g(n) & \text{se } n > 0 \end{cases}$$

allora:

$$T(n) = a + \sum_{k=1}^{n} g(k)$$

Anche TREE-MINIMUM *e* TREE-MAXIMUM *hanno complessità in tempo* $O(h)$

Inserimento di un elemento in un ABR

In modo tale da conservare la proprietà ABR

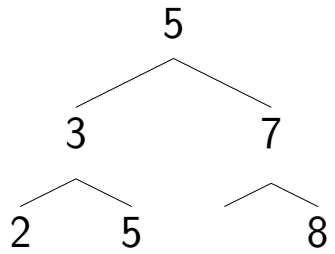
Input: un ABR T, un nodo z tale che

- $key[z] = k$
- $left[z] = NIL$
- $right[z] = NIL$
- $p[z] = NIL$

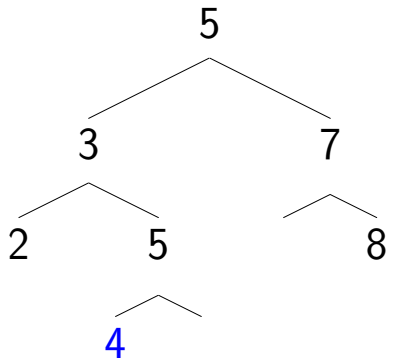
Output: T modificato con l'aggiunta della foglia z

Esempio

L'inserimento di **4** in



restituisce



Inserimento di un elemento in un ABR - versione ricorsiva

- Se T è vuoto: crea l'albero con l'unico nodo z
- Altrimenti:
  - se z ha chiave minore della radice di T, inserisci z nel sottoalbero sinistro
  - altrimenti inseriscilo nel sottoalbero destro.

Il caso "albero vuoto" va trattato a parte perché va modificato il campo $root$ di T stesso: la scansione dell'albero segue i possibili "padri" di z.

```
TREE-INSERT(T,z)
1. if root[T] = NIL
2.    then root[T] <- z
3.    else TREE-INS(root[T],z)

TREE-INS(x,z)
1. if key[z] < key[x]  ;; z va inserito a sinistra
2.    then if left[x] = NIL
3.            then left[x] <- z
4.                 p[z] <- x
5.            else TREE-INS(left[x],z)
6.    else if right[x] = NIL  ;; z va inserito a destra
7.            then right[x] <- z
8.                 p[z] <- x
9.            else TREE-INS(right[x],z)
```

Inserimento di un elemento in un ABR - versione iterativa

- Si discende l'albero: per ogni nodo x, se $key[z] < key[x]$, si scende sul figlio sinistro, altrimenti sul destro.
- Ad ogni passo si tiene memoria del padre y di x.
- Quando x è NIL: al suo posto va inserito il puntatore a z; quindi:

se $key[z] < key[y]$, z è il figlio sinistro di y, altrimenti z è il figlio destro di y.

```
TREE-INSERT(T,z)
1. x <- root[T]      ;; usata per discendere l'albero
2. y <- NIL          ;; puntatore al padre di x
3. while x != NIL
4.    do y <- x
5.       if key[z] < key[x]
6.          then x <- left[x]
7.          else x <- right[x]
8. p[z] <- y         ;; y e' il padre di z
9. if y = NIL        ;; T era vuoto e z e' la radice del nuovo albero
10.   then root[T] <- z
11.   else if key[z] < key[y]
12.           then left[y] <- z
13.           else right[y] <- z
```

Complessità dell'inserimento: $O(h)$

Eliminazione di un elemento da un ABR

Input:

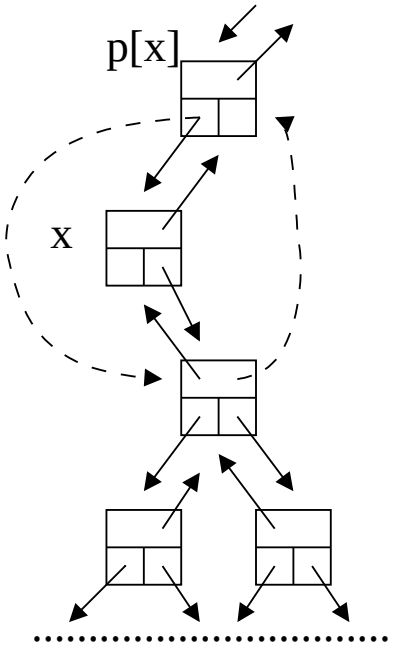
- un ABR T;
- il puntatore x all'elemento che si vuole cancellare (assumiamo $x \neq NIL$).

Output: ABR T modificato

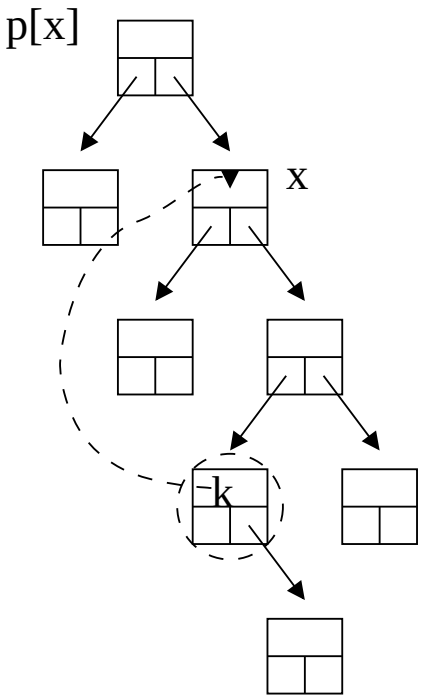
- se x ha al massimo un figlio, si cancella x, "bypassandolo":
  - il puntatore a x del padre di x viene sostituito con il puntatore al figlio di x.
- Altrimenti, se x ha due figli, x viene sostituito dal suo "successore" y (il minimo del sottoalbero destro).

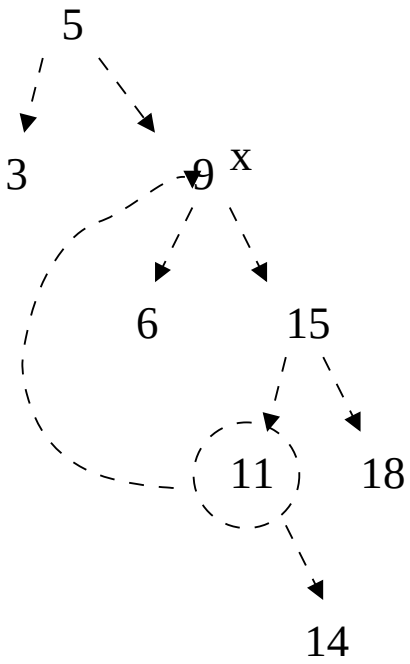
  Sostituendo il minimo elemento del s.a. destro al posto di x la proprietà ABR viene conservata.
  - y non ha figlio sinistro: viene cancellato, "bypassandolo".
  - Il "contenuto" (chiave e dati satellite) di x viene sostituito dal contenuto di y.

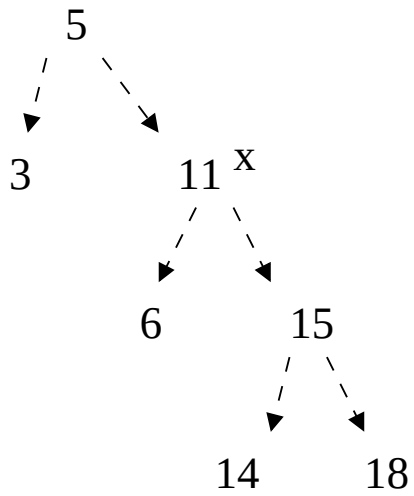
Eliminazione di un elemento da un ABR (II)



un solo figlio







due figli

Algoritmo di cancellazione

- Si determina il nodo da cancellare (1–3):
  $y =$ x stesso, se ha al massimo un figlio,
  altrimenti il suo successore
  In entrambi i casi, y **ha al massimo un figlio**.
- Bypass di y:
  - Sia z il figlio di y, se c'e', altrimenti NIL (4–6).
  - Se z non e' NIL, suo padre diventa il padre di y (7–8).
  - Se y e' la radice dell'albero (9),
    allora la radice dell'albero diventa z (10),
    altrimenti si sostituisce con z il puntatore a y nel padre di y ($left[p[y]]$ o $right[p[y]]$) (11–13).
- Se y non e' x stesso:
  si copiano in x chiave e dati satellite di y (14–16).
- Si dealloca la memoria per y (17).

Algoritmo di cancellazione - pseudocodifica

```
TREE-DELETE(T,x)
 1. if left[x] = NIL or right[x] = NIL
 2.    then y <- x
 3.    else y <- TREE-MINIMUM(right[x])
 4. if left[y] != NIL
 5.    then z <- left[y]
 6.    else z <- right[y]
 7. if z != NIL
 8.    then p[z] <- p[y]
 9. if p[y] = NIL
10.    then root[T] <- z
11.    else if y = left[p[y]]
12.         then left[p[y]] <- z
13.         else right[p[y]] <- z
14. if y != x
15.    then key[x] <- key[y]
16.         ;; copia dati satellite di y in x
17. ;; deallocazione della memoria per y
```

Complessità delle operazioni sugli ABR

Se la dimensione dell'input è l'altezza h dell'albero

Caso peggiore: la ricerca prosegue sul sottoalbero di profondità massima

$$\begin{aligned} T_{TREE-SEARCH}(0) &= a \\ T_{TREE-SEARCH}(h+1) &= T_{TREE-SEARCH}(h) + c \end{aligned}$$

$T(h)$ è $O(h)$

Se la dimensione dell'input è il numero n dei nodi dell'albero

Quale può essere la massima altezza di un albero binario? (caso peggiore)

Caso peggiore : l'albero è completamente sbilanciato a destra o a sinistra (massima altezza per n nodi)

Caso migliore : l'albero è completo (minima altezza per n nodi)

Dato un albero con n nodi qual'è la sua altezza h?

Complessità delle operazioni sugli ABR (II)

altezza	nodi (caso peggiore)	nodi (caso migliore)
0	1	1
1	2	3
h	$h+1$	$2^{h+1}-1$

perciò

nodi	altezza (caso peggiore)	altezza (caso migliore)
n	$n-1$	$\lfloor log_2(n) \rfloor$

Quindi

nel caso peggiore un albero di n nodi ha altezza $h = n - 1$:

$T(n)$ *è* $O(n)$

nel caso migliore *un albero di n nodi ha altezza $h = \lfloor log_2 n \rfloor$:*

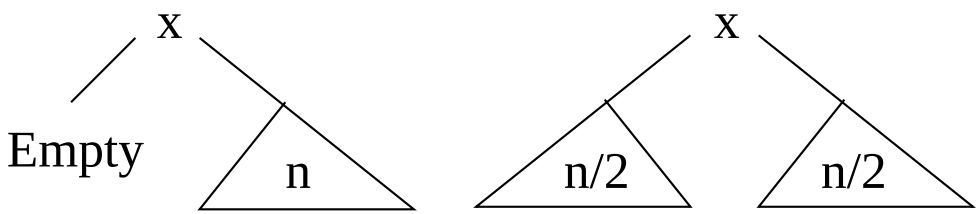
$T(n)$ *è* $O(log_2\, n)$

Complessità della ricerca: caso medio

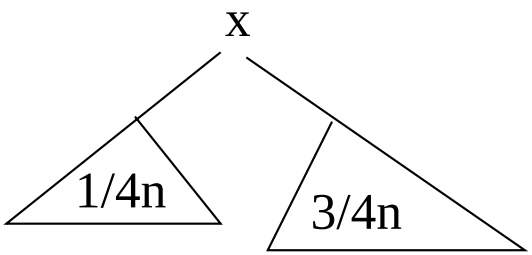
Le operazioni sugli ABR richiedono tempo logaritmico anche nel caso medio

Idea intuitiva

Distribuzione dei nodi nei due sottoalberi della radice



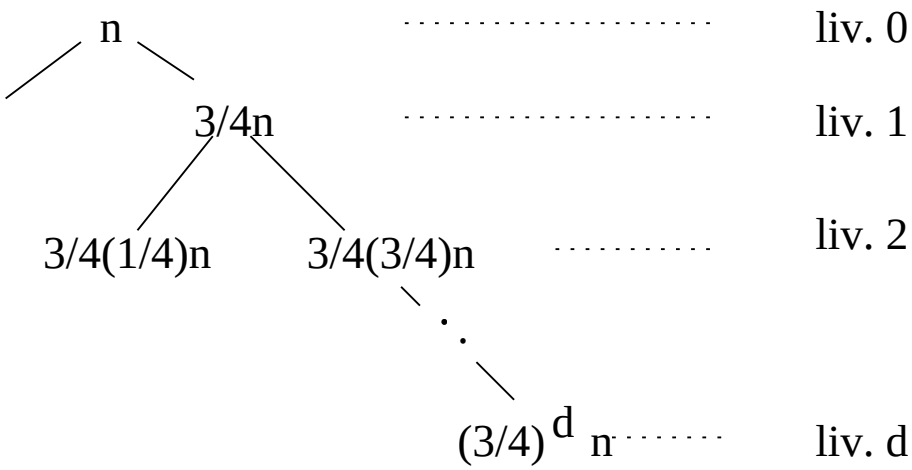
casi particolari



caso generale

- Supponiamo che questa distribuzione valga per ogni livello.
- Consideriamo sempre il sottoalbero più grande:

Complessità della ricerca: caso medio (II)



Per d "grande": $(3/4)^d n \to 1$.

d è l'altezza dell'albero

$(3/4)^d n = 1$ **per** $d = 2.5 \cdot log_2 n$

Quindi

$$T_{TREE-SEARCH}(n) = O(d) = O(lg\, n)$$

Tree sort

Il risultato della visita simmetrica di un ABR è una sequenza ordinata.

TREE-SORT ordina l'array A:

- costruendo un ABR T con gli elementi di A
- visitando T in ordine simmetrico e copiando gli elementi in A.

```
TREE-SORT(A)
1. T <- EMPTY()
2. for i=1 to length[A]
3.    do TREE-INSERT(T,MK-TREE-ELEM(A[i]))
4. TREE-TO-ARRAY(A,root[T],1)
```

Tree sort (II)

L'operazione TREE-TO-ARRAY prende in ingresso:

- un puntatore a un array A
- un puntatore x a un nodo di un ABR
- un indice i nell'array A

e visita l'ABR con radice in x, in ordine simmetrico, copiando le chiavi degli elementi visitati in A, a partire dalla posizione i.

Riporta l'indice della prossima posizione libera in A.

```
TREE-TO-ARRAY(A,x,i)
1. if x != NIL
2.    then i <- TREE-TO-ARRAY(A,left[x],i)
3.         A[i] <- key[x]
4.         i <- TREE-TO-ARRAY(A,right[x],i+1)
6. return i
```

La funzione MK-TREE-ELEM(k) costruisce un nodo con chiave k:

```
MK-TREE-ELEM(k)
1.  allocazione della memoria per x
2.  key[x] <- k
3.  left[x] <- right[x] <- p[x] <- NIL
4.  return x
```

Qual è la complessità asintotica del Tree-Sort?