Alberi binari di ricerca

Algoritmi e strutture dati

Ugo de'Liguoro, Andras Horvath

1

Alberi binari di ricerca

Sia A un insieme ordinato. L'insieme di **alberi binari di ricerca** su A, denotato con **BRT(*A*)**, è definito induttivamente come segue:

a) $\emptyset \in BRT(A)$ (l'albero vuoto fa parte dell'insieme)

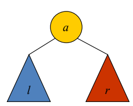
b)

$$a \in A \;\wedge\; l \in BRT(A) \;\wedge\; r \in BRT(A) \;\wedge\; \forall c \in keys(l).c < a \;\wedge\; \forall c \in keys(r).a < c$$

$$\Downarrow$$

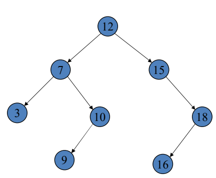
$$\{a, l, r\} \in BRT(A)$$

A parole: *l* e *r* sono alberi binari di ricerca, ogni chiave in *l* è minore di *a* e ogni chiave in *r* è maggiore di *a*.



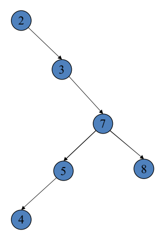
2

Esempio di albero di ricerca



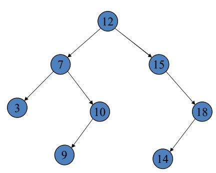
3

Esempio di albero di ricerca



4

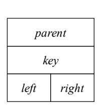
Esempio di **non** albero di ricerca



- La definizione induttiva implica che per ciascun nodo deve essere vero che nel suo sottoalbero sinistro ci sono etichette più piccole e nel suo sottoalbero destro più grandi.
- L'etichetta 14 non va bene nel sottoalbero destro del nodo con etichetta 15.

5

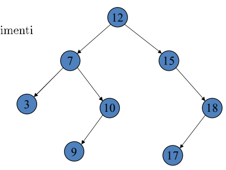
Realizzazione con puntatori



6

Ricerca ricorsiva

RIC-SEARCH(x, T)
  ▷ pre: x chiave, T binario di ricerca
  ▷ post: il nodo $S \in T$ con $S.key = x$ se esiste, nil altrimenti
if $T = nil$ **then return** nil
else
  if $x = T.key$ **then return** T
  else
    if $x < T.key$ **then return** SEARCH$(x, T.left)$
    else ▷ $x > T.key$
      return SEARCH$(x, T.right)$
    end if
  end if
end if



Complessità $O(h)$ dove h = altezza di T

Esercizio: simulare ricerche di chiavi presenti e chiavi non presenti.

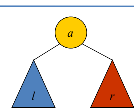
7

Ricerca iterativa

IT-SEARCH(x, T)
  ▷ pre: x chiave, T binario di ricerca
  ▷ post: il nodo $S \in T$ con $S.key = x$ se esiste, nil altrimenti
while $T \neq nil$ **and** $x \neq T.key$ **do**
  if $x < T.key$ **then**
    $T \leftarrow T.left$
  else
    $T \leftarrow T.right$
  end if
end while
return T

8

Stampa delle etichette in ordine

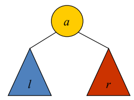


- per stampare tutte le etichette in ordine
  1. stampa $keys(l)$ in ordine
  2. stampa a
  3. stampa $keys(r)$ in ordine
- per stampare $keys(l)$ in ordine
  1. stampa in ordine le etichette del sottoalbero sinistro della radice di l
  2. stampa l'etichetta della radice di l
  3. stampa in ordine le etichette del sottoalbero destro della radice di l
- per stampare $keys(r)$ in ordine proseguire in maniera analoga
- e così via in maniera ricorsiva

9

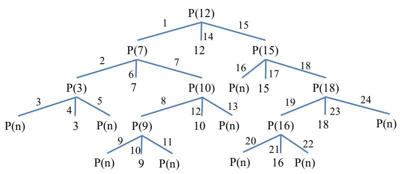
Stampa delle etichette in ordine

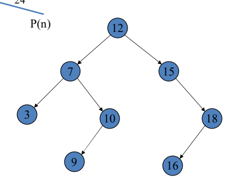
PRINT-INORDER(T)
  ▷ pre: T binario di ricerca
  ▷ post: stampate le chiavi in T in ordine
if $T = nil$ **then**
  return
end if
PRINT-INORDER($T.left$)
print $T.key$
PRINT-INORDER($T.right$)



10

Stampa delle etichette in ordine





Etichette stampate:

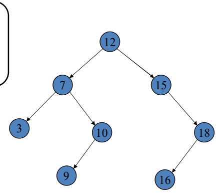
3 7 9 10 12 15 16 18

P(x) indica una chiamata con riferimento al nodo x, n sta per nil.
Numero secco indica la stampa. I numeri sugli archi indicano l'ordine.

11

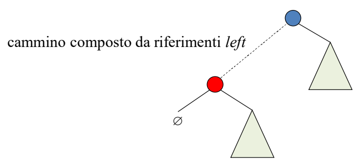
Minimo e massimo

Come torvare il nodo con chiave minima/massima in $T \neq \varnothing$?



12

Minimo



Basta discendere lungo il ramo sinistro.

13

Minimo e massimo

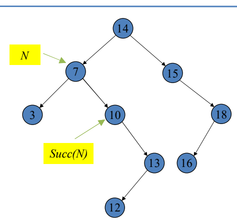
TREE-MIN(T)
▷ pre: T binario di ricerca non vuoto
▷ post: il nodo $S \in T$ con $S.key$ minimo
$S \leftarrow T$
while $S.left \neq nil$ **do**
$S \leftarrow S.left$
end while
return S

Per trovare invece il massimo bisogna scendere lungo il ramo destro. (Per ottenere l'algoritmo si sostituisce *left* con *right*).

14

Successore

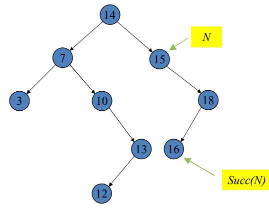
Il successore di un nodo N in un albero di ricerca T è il nodo con etichetta minima tra quelle maggiori di $N.key$. (Il massimo non ha succesore.)



15

Successore

Il successore di un nodo *N* in un albero di ricerca *T* è il nodo con etichetta minima tra quelle maggiori di *N.key*. (Il massimo non ha succesore.)

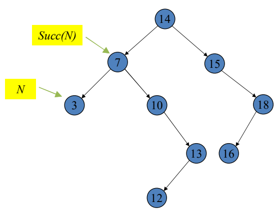


Se per un nodo N, $N.right \neq nil$ allora il suo successore è il minimo del suo sottoalbero destro.

16

Successore

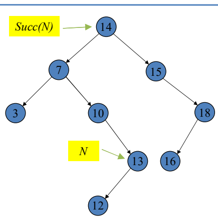
Il successore di un nodo *N* in un albero di ricerca *T* è il nodo con etichetta minima tra quelle maggiori di *N.key*. (Il massimo non ha succesore.)



17

Successore

Il successore di un nodo *N* in un albero di ricerca *T* è il nodo con etichetta minima tra quelle maggiori di *N.key*. (Il massimo non ha succesore.)



Se per un nodo N, $N.right = nil$ allora il suo successore è il suo avo A più vicino tale che $N.key < A.key$.

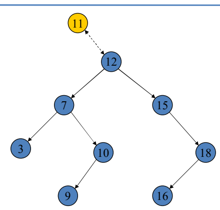
18

Successore

TREE-SUCC(N)
  ▷ pre: N nodo di un albero bin. di ricerca
  ▷ post: il successore di N se esiste, nil altrimenti
if $N.right \neq nil$ **then**
  return TREE-MIN($N.right$)
else ▷ il successore è l'avo più vicino con etichetta maggiore
  $P \leftarrow N.parent$
  while $P \neq nil$ **and** $N = P.right$ **do**
    $N \leftarrow P$
    $P \leftarrow N.parent$
  end while
  return P
end if

19

Esempio di inserimento

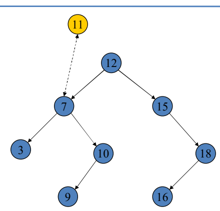


11<12 e il sottoalbero sinistro del 12 non è vuoto, l'inserimento avverrà nel sottoalbero sinistro del 12.

Gli inserimenti in un albero di ricerca avvengono sempre al livello delle foglie sostituendo un sottoalbero vuoto (un *nil*) in modo tale che l'albero rimanga albero di ricerca.

20

Esempio di inserimento

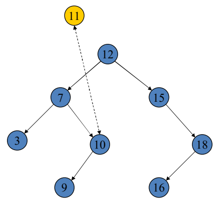


11>7 e il sottoalbero destro del 7 non è vuoto, l'inserimento avverrà nel sottoalbero destro del 7.

Gli inserimenti in un albero di ricerca avvengono sempre al livello delle foglie sostituendo un sottoalbero vuoto (un *nil*) in modo tale che l'albero rimanga albero di ricerca.

21

Esempio di inserimento

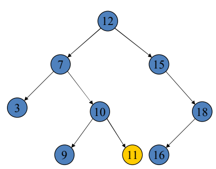


11>10 e il sottoalbero destro del 10 è vuoto, l'11 sarà inserito come figlio destro del 10.

Gli inserimenti in un albero di ricerca avvengono sempre al livello delle foglie sostituendo un sottoalbero vuoto (un *nil*) in modo tale che l'albero rimanga albero di ricerca.

22

Esempio di inserimento



Gli inserimenti in un albero di ricerca avvengono sempre al livello delle foglie sostituendo un sottoalbero vuoto (un *nil*) in modo tale che l'albero rimanga albero di ricerca.

23

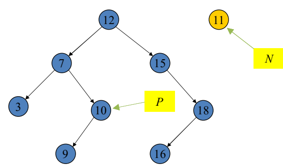
Inserimento

```
TREE-INSERT(N, T)
    ▷ pre: N nuovo nodo con N.left = N.right = nil, T è un albero binario di ricerca
    ▷ post: N è un nodo di T, T è un albero binario di ricerca
P ← nil
S ← T
while S ≠ nil do        ▷ inv: se P ≠ nil allora P è il padre di S
    P ← S
    if N.key = S.key then
        return
    else
        if N.key < S.key then
            S ← S.left
        else
            S ← S.right
        end if
    end if
end while
```

24

Inserimento

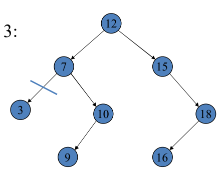
```
N.parent ← P
if P = nil then
    T ← N
else
    if N.key < P.key then
        P.left ← N
    else
        P.right ← N
    end if
end if
```



25

Cancellazione

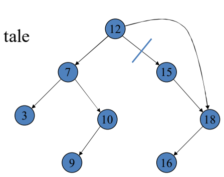
- caso più semplice: nodo da eliminare non ha figli
- basta settare a *nil* il riferimento che punta al nodo nel suo padre (*left* o *right*)
- per cancellare, per esempio, il nodo 3:
  si setta il rif. *left* del nodo 7 a *nil*



26

Cancellazione

- caso intermedio: nodo da eliminare ha esattamente un figlio
- basta agganciare il sottoalbero esistente al padre (come riferimento *left* o *right*)
- per cancellare, per esempio, il nodo 15:
  si setta il rif. *right* del nodo 12 in modo tale da fare riferimento al nodo 18



27

Cancellazione

1-DELETE(Z, T)
▷ pre: Z nodo di T con esattamente un figlio
▷ post: Z non è più un nodo di T

```
if Z = T then
    if Z.left ≠ nil then
        T ← Z.left
    else
        T ← Z.right
    end if
    Z.parent ← nil
```

Se il nodo da cancellare è la radice, allora la radice del sottoalbero esistente diventa la radice di tutto l'albero.

28

Cancellazione

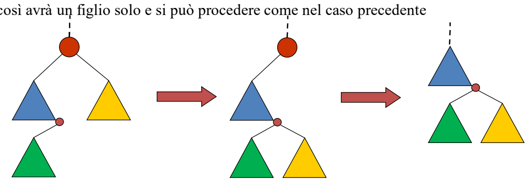
```
else
    if Z.left ≠ nil then
        Z.left.parent ← Z.parent
        S ← Z.left
    else
        Z.right.parent ← Z.parent
        S ← Z.right
    end if
    if Z.parent.right = Z then
        Z.parent.right ← S
    else
        Z.parent.left ← S
    end if
end if
```

Se il nodo da cancellare non è la radice, allora il figlio esistente di Z *(left* o *right)* deve avere il padre di Z come padre *(parent)* e il padre di Z deve avere il figlio esistente di Z come figlio *(left* o *right)*.

29

Cancellazione: fusione

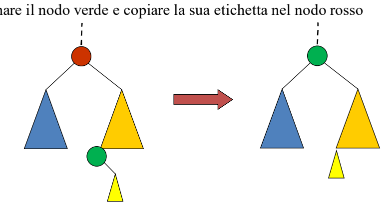
- caso più complicato: nodo da eliminare ha due figli
- il nodo da eliminare è il nodo rosso grande
- il suo sottoalbero destro (triangolo giallo) può essere agganciato come sottoalbero destro al massimo del suo sottoalbero sinistro (nodo rosso piccolo)
- così avrà un figlio solo e si può procedere come nel caso precedente



30

Cancellazione: copia

- caso più complicato: nodo da eliminare ha due figli
- il nodo da eliminare è il nodo rosso
- il minimo del suo sottoalbero destro (nodo verde) ha un figlio al massimo
- si può eliminare il nodo verde e copiare la sua etichetta nel nodo rosso



31

Cancellazione

```
TREE-DELETE(Z, T)
    ▷ pre: Z nodo di T
    ▷ post: Z non è più un nodo di T
if Z.left = nil ∧ Z.right = nil then        ▷ Z è una foglia
    if Z = T then
        T ← nil
    else
        if Z.parent.left = Z then        ▷ Z è figlio sinistro
            Z.parent.left ← nil
        else        ▷ Z è figlio destro
            Z.parent.right ← nil
        end if
    end if
else
    if Z.left = nil ∨ Z.right = nil then
        1-DELETE(Z, T)
    else        ▷ Z ha due figli e dunque si può cercare il minimo in Z.right
        Y ← TREE-MIN(Z.right)
        Z.key ← Y.key
        TREE-DELETE(Y, T)
    end if
end if
```

32

Salvataggio in lista

- problema: inserire gli elementi di un BRT in ordine in una lista
- operazioni disponibili:
- LISTINSERT($key\ c, list\ L$) restituisce una lista in cui si ha un nodo in testa con etichetta c e L agganciata a questo nodo (complessità $O(1)$);
- APPEND($list\ L_1,\ list\ L_2$) restituisce una lista in cui L_2 è agganciata a L_1 in coda (complessità $O(|L_1|)$ dove $|L_1|$ denota il numero di elementi in L_1).
- si può seguire l'idea vista pere sviluppare Print-Inorder

33

Salvataggio in lista

```
ToList-Inorder(T)
    ▷ pre: T binario di ricerca
    ▷ post: ritorna la lista ordinata delle chiavi in T
if T = nil then
    return nil
else
    L ← ToList-Inorder(T.left)
    R ← ToList-Inorder(T.right)
    R ← ListInsert(T.key, R)
    return Append(L, R)
end if
```

- simulare l'algoritmo con un albero sbilanciato a sinistra e con uno sbilanciato a destra
- complessità nel caso peggiore e $O(n^2)$ per via di Append

34

Salvataggio in lista

- con albero sbilanciato a destra la lista L ha sempre un elemento solo e quindi la complessità è $O(n)$
- con albero sbilanciato a sinistra la lista L ha $1,2,3,\dots,n-1$ elementi e quindi la complessità è $O(n^2)$ per via di Append

35

Salvataggio in lista

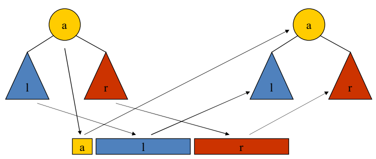
- visitando i nodi in ordine decrescente di etichette si può evitare l'utilizzo di Append e avere quindi un algoritmo $O(n)$:

```
ToList-Inorder(T, L)
    ▷ pre: T binario di ricerca
    ▷ post: ritorna la lista ordinata delle chiavi in T concatenata con L
if T = nil then
    return L
else
    L ← ToList-Inorder(T.right, L)
    L ← ListInsert(T.key, L)
    return ToList-Inorder(T.left, L)
end if
```

36

Copia di un albero

Proposizione: Sia T un albero di ricerca ed L la lista prodotta dalla visita in preordine di T: se T' è costruito per inserimenti successivi degli elementi di L (da sinistra a destra) allora T e T' sono isomorfi.



37