# Alberi binari di ricerca

April 19, 2020

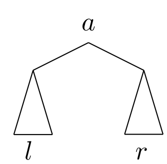
**Obiettivi: albero binario di ricerca come struttura ricorsiva, sviluppo di algoritmi ricorsivi su alberi binari di ricerca**.
**Argomenti: definizione e rappresentazione di albero binario di ricerca, algoritmi su alberi binari di ricerca**.

## 1 Definizione e rappresentazione

Sia $A$ un insieme ordinato (l'insieme delle etichette). 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)

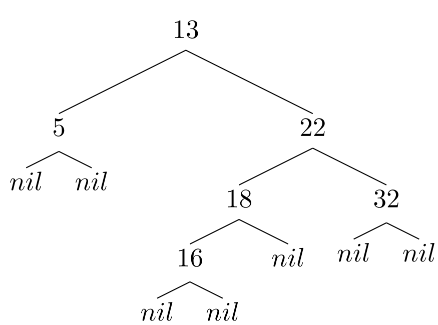
$$a \in A \ \wedge \ l \in BRT(A) \ \wedge \ r \in BRT(A) \ \wedge \ \forall c \in keys(l).c < a \ \wedge \ \forall c \in keys(r).a < c$$
$$\Downarrow$$
$$\{a, l, r\} \in BRT(A)$$

Cioè, data un'etichetta, $a$, e due alberi, $l$ e $r$ in $BRT(A)$, agganciando $l$ come sottoalbero sinistro e $r$ come sottoalbero destro al nodo con etichetta $a$, si ottiene un nuovo albero binario di ricerca se tutte le etichette in $l$ sono minori di $a$ e tutte le etichette in $r$ sono maggiori di $a$ (la funzione *keys* restituisce l'insieme di etichette presenti in un albero). Graficamente:




**Rappresentazione** di un albero binario posizionale: ogni nodo contiene l'etichetta (*key*) e due puntatori, *left* e *right*, che fanno riferimento al sottoalbero sinistro e destro. In certi applicazione conviene facilitare la risalita aggiungendo ad ogni nodo un puntatore al padre del nodo denotato con *parent*.

Esempio di albero binario di ricerca sull'insieme dei numeri naturali (sottoalberi vuoti sono rappresentati esplicitamente con *nil*).



## 2 Algoritmi

**Stampa di tutte le etichette in ordine.** Idea: dato l'albero $\{a, l, r\}$ prima bisogna stampare tutte le etichette in $l$ in ordine, poi stampare $a$ e, in fine, stampare tutte le etichette in $r$ 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)
```

**Copia di tutte le etichette in una lista semplice (non circolare e senza riferimento all'ultimo elemento) in ordine.** Assumiamo di avere a disposizione due algoritmi che fanno operazioni con liste:

- 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 $L_1$).

```
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 sull'esempio precedente. Simulare l'algoritmo con un albero sbilanciato a sinistra e con uno sbilanciato a destra. Qual è il caso peggiore per l'algoritmo precedente? La complessità nel caso peggiore è $O(n^2)$ dove $n$ è il numero di elementi nell'albero.

Per evitare si modifica l'algoritmo in modo tale che Append non venga utilizzato.

```
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
```

La complessità diventa $O(n)$ dove $n$ è il numero di elementi nell'albero. L'algoritmo in pratica visita i nodi in ordine decrescente delle etichette e quindi effettua solo inserimenti in testa.

**Ricerca di un elemento.** Si scende nell'albero utilizzando ricorsione o in maniera iterativa.

```
RIC-SEARCH(x, T)
      ▷ pre: x chiave, T albero binario di ricerca
      ▷ post: restituito un nodo S ∈ 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
```

```
IT-SEARCH(x, T)
      ▷ pre: x chiave, T binario di ricerca
      ▷ post: il nodo S ∈ T con S.key = x se esiste, nil altrimenti
S ← T
while S ≠ nil and x ≠ S.key do
    if x < S.key then
        S ← S.left
    else
        S ← S.right
    end if
end while
return S
```

La complessità è $O(h)$ dove $h$ è l'altezza dell'albero.

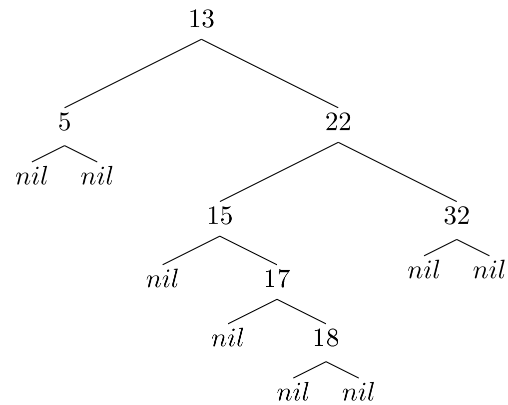
**Ricerca del minimo.** Il minimo si trova scendendo verso sinistra.

```
TREE-MIN(T)
      ▷ pre: T binario di ricerca non vuoto
      ▷ post: il nodo S ∈ T con S.key minimo
S ← T
while S.left ≠ nil  do
    S ← S.left
end while
return S
```

La complessità è $O(h)$ dove $h$ è l'altezza dell'albero.

**Ricerca del successore.** Il successore di un nodo $N$ in un albero binario di ricerca $T$ è il nodo con etichetta minima tra quelle maggiori di $N.key$. Se il nodo $N$ contiene la chiave massima allora non ha successore nell'albero. Se il nodo $N$ ha un discendente destro allora il successore di $N$ è il massimo del sottoalbero che ha radice in $N.right$. Se $N$ non contiene l'etichetta massima dell'albero e non ha un discendente destro allora il successore di $N$ si trova risalendo a partire dal nodo $N$ e fermandosi al primo nodo che ha un'etichetta maggiore di quella di $N$.

Consideriamo l'albero seguente.



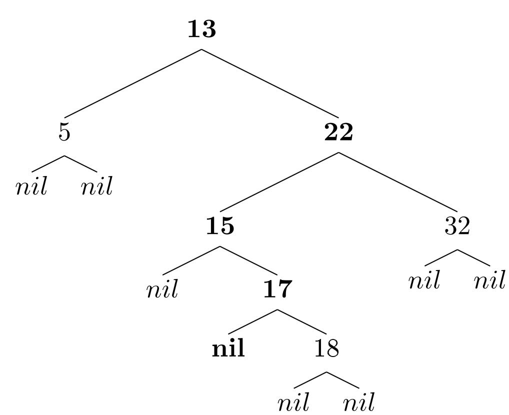

Il nodo con etichetta 32 non ha successore nell'albero. Il successore del nodo 13 è il nodo 15. Il successore del nodo 18 è il nodo 22.

L'algoritmo seguente mette in pratica le idee precedenti.

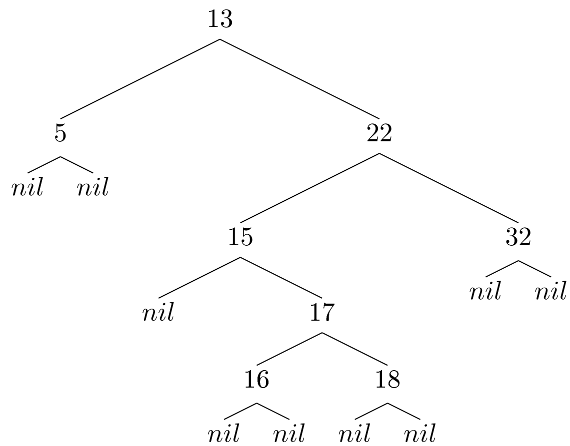
Tree-Succ($N$)
  ▷ pre: $N$ nodo di un albero bin. di ricerca
  ▷ post: il successore di $N$ se esiste, *nil* altrimenti
**if** $N.right \neq nil$ **then**
  **return** Tree-Min($N.right$)
**else** ▷ il successore è l'avo più vicino con etichetta maggiore
  $P \leftarrow N.parent$
  **while** $P \neq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

**Inserimento.** L'inserimento sostituisce un sottoalbero vuoto (un *nil*) con un sottoalbero che contiene un singolo nodo con l'etichetta da inserire. La posizione si cerca secondo le caratteristiche dell'albero binario di ricerca.

Esempio: consideriamo l'albero e inseriamo la chiave 16. Nell'albero successivo vengono indicati con grassetto i nodi esaminati durante la ricerca della posizione giusta del nuovo nodo fino al sottoalbero vuoto da sostituire.




Dopo l'inserimento l'albero è il seguente.




Algoritmo dell'inserimento (utilizzando il meccanismo passaggio di parametro per riferimento perché le modifiche devono aver effetto al di fuori dal metodo):

```
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
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
```

Nell'algoritmo precedente il ciclo **while** trova il posto del nuovo nodo e il resto fa l'inserimento stesso.

**Cancellazione.** Si devono distinguere tre casi quando si deve cancellare un nodo $Z$:

- $Z$ non ha figli (è una foglia): basta settare a *nil* il riferimento a $Z$ nel genitore di $Z$ (oppure rendere l'albero un albero vuoto se $Z$ è la radice);
- $Z$ ha esattamente un figlio: bisogna agganciare l'unico figlio di $Z$ al genitore di $Z$;
- $Z$ ha due figli: l'etichetta in $Z$ può essere sostituita con l'etichetta minima che si trova in $Z.right$ e poi il nodo dove si trovava originalmente l'etichetta può essere eliminata (utilizzando uno dei due punti precedenti, visto che il nodo con etichetta minima ha al massimo un figlio).

Gli algoritmi che ne seguono sono (utilizzando di nuovo il meccanismo passaggio di parametro):

```
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
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
```

```
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
```

Cercare, nell'ultimo albero binario di ricerca disegnato sopra, tre nodi che corrispondono ai tre casi e simulare gli algoritmi.