Alberi binari di ricerca

Pietro Di Lena

Dipartimento di Informatica – Scienza e Ingegneria
Università di Bologna

Algoritmi e Strutture di Dati
Anno Accademico 2021/2022

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 della struttura dati Dizionario su BST

Ripasso: struttura dati Dizionario

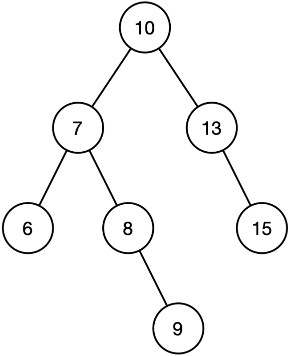
- Struttura dati generica per memorizzare oggetti
  - Contiene un insieme di chiavi univoche
  - Ogni chiave è associata ad un valore
  - I valori posso essere duplicati, le chiavi sono uniche
- Operazioni basilari di un Dizionario (prototipo):
  - SEARCH(Key k): cerca l'oggetto associato alla chiave 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 ordinato: SEARCH in $O(\log n)$, INSERT/DELETE in $O(n)$
  - Lista concatenata: SEARCH/DELETE in $O(n)$, INSERT in $O(1)$

Alberi Binari di Ricerca (BST)

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

- Definizione di Albero Binario di Ricerca:

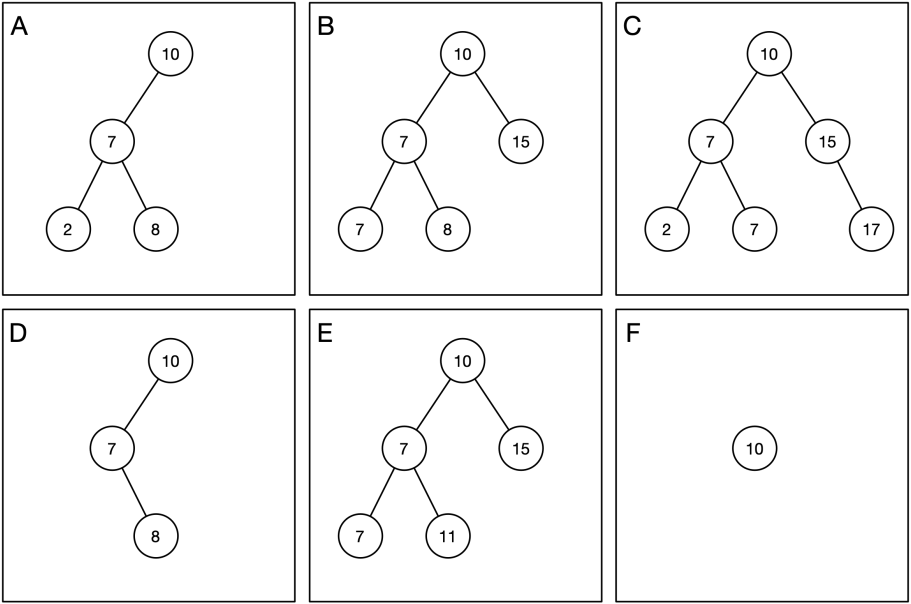
1. Albero Binario
2. Ogni nodo v contiene una chiave (ordinabile) $v.key$ e dati $v.data$ associati alla chiave
3. Proprietà di ordinamento dei BST: tutte le chiavi nel sottoalbero sinistro di v sono $\leq v.key$ e tutte le chiavi nel sottoalbero destro di v sono $\geq v.key$



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

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

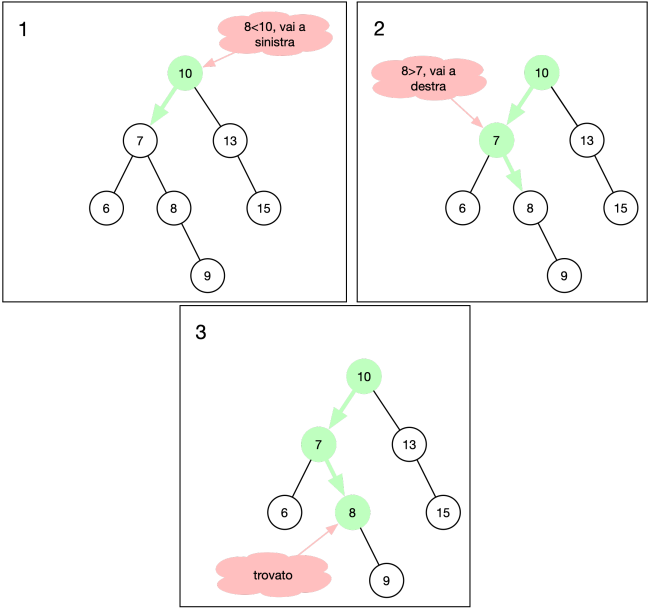
Alberi Binari di Ricerca?



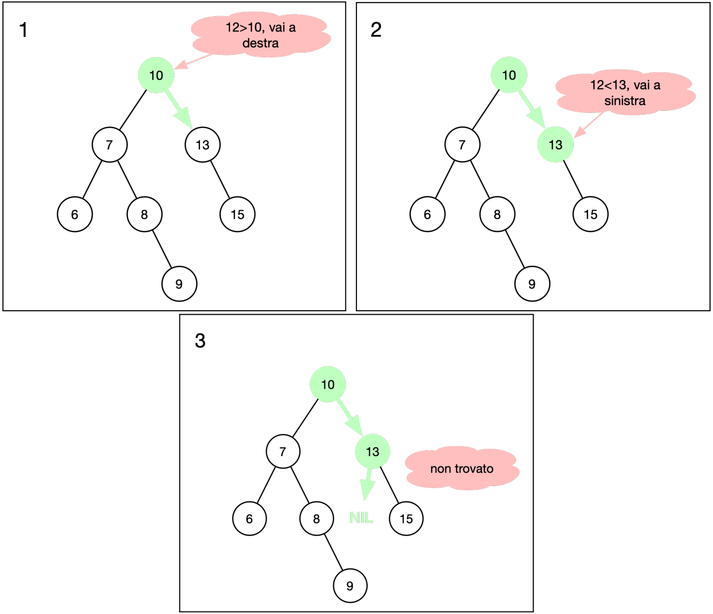
Operazioni su Alberi Binari di Ricerca

- 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

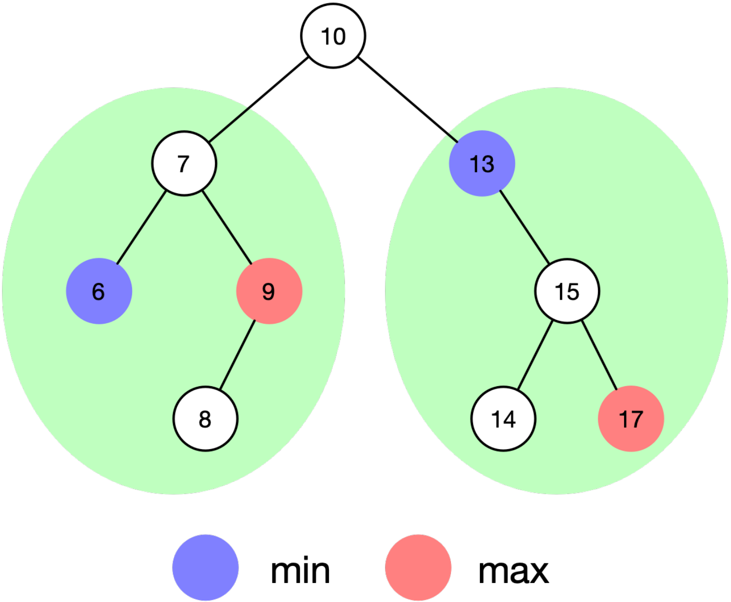


Pseudocodice: search

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

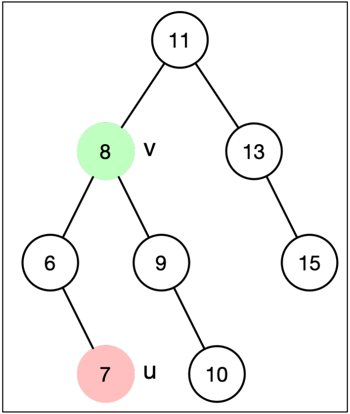
```
function MIN(BST T) → BST
    while T ≠ NIL and T.left ≠ NIL do
        T = T.left
    return T
```

- 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

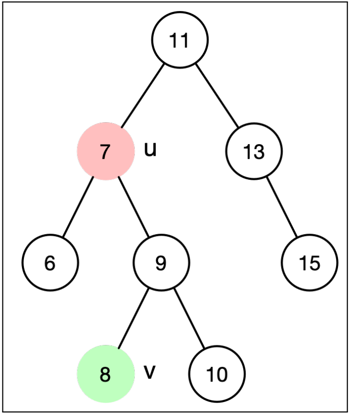
- Caso 1
  - Il nodo *v* ha un figlio sinistro
  - Il predecessore è il nodo *u* con chiave massima nel sottoalbero sinistro di *v*
- Correttezza
  - Per la propietà di ordine dei BST, il sottoalbero sinistro di *v* contiene solo chavi $\leq v.key$



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(BST T) → BST
    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

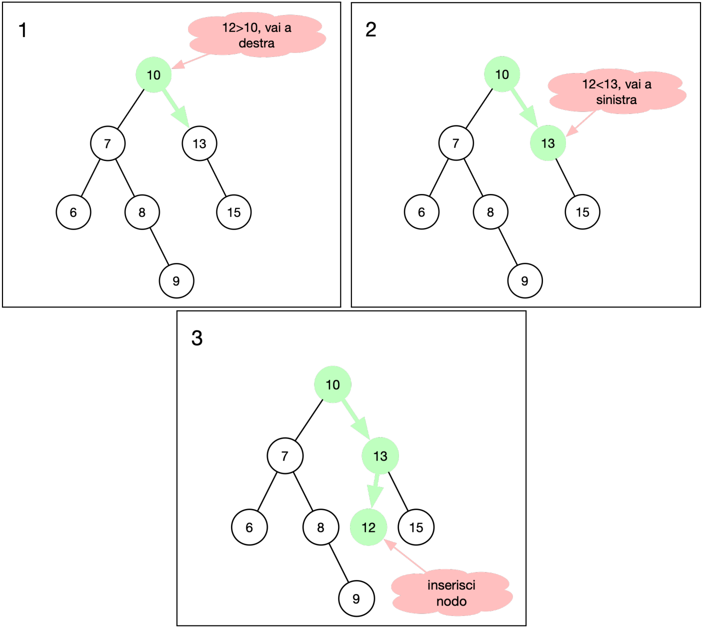
PSEUDOCODICE: SUCCESSOR

SUCCESSOR è simmetrica a PREDECESSOR

```
function SUCCESSOR(BST T) → BST
    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



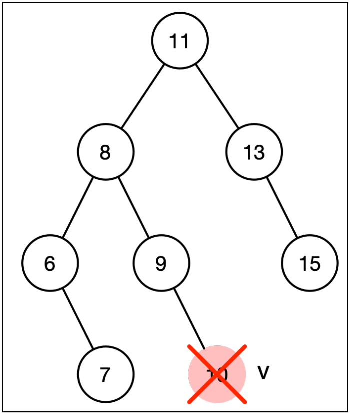
Pseudocodice: insert

```
function INSERT(BST T, KEY k, DATA d) → BST
    N = new BST(k, d), P = NIL, S = T
    while S ≠ NIL do                          ▷ Search position
        P = S
        if k < S.key then
            S = S.left
        else
            S = S.right
    N.parent = P                              ▷ Insert node
    if P ≠ NIL and k < P.key then
        P.left = N
    else if P ≠ NIL then
        P.right = N
    if T == NIL then return N else return T
```

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

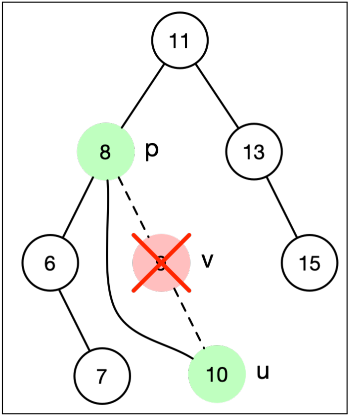
Esempio: rimozione (caso 1)

- 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)

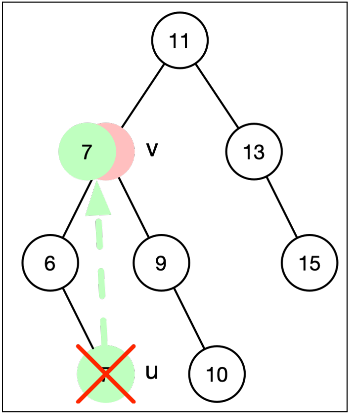
- Caso 2
  - Il nodo da rimuovere v ha un solo figlio u
  - u diventa figlio del genitore p di v
  - Se v è un figlio sinistro, u diventa figlio sinistro di p
  - Se v è un figlio destro, u diventa figlio destro di p
  - Possiamo rimuovere v



- 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) → BST                                ▷ Returns the (new) root
    v = SEARCH(T, k)
    if v ≠ NIL then
        if v.left == NIL or v.right == NIL then                     ▷ Case 1 or 2
            return DELETENODE(T,v)
        else                                                        ▷ Case 3
            u = PREDECESSOR(v)
            v = u                                   ▷ v.key = u.key and v.data = u.data
            return DELETENODE(T, u)
```

```
function DELETENODE(BST T, BST v) → BST
    p = v.parent
    if p ≠ NIL then                                                 ▷ v is not the root
        if ISLEAF(v) 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 = T is the root
        if ISLEAF(v) then T = NIL                                   ▷ Case 1
        else if v.right ≠ NIL then T = v.right                      ▷ Case 2
        else if v.left ≠ NIL then T = v.left                        ▷ Case 2
    DELETE(v)
    return T
```

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 h di un BST
- L'altezza h di un BST può variare di molto
  - L'altezza di un BST con n nodi è $h = O(n)$
    - L'albero è al peggio una lista
  - L'altezza di un BST con n nodi è $h = \Omega(\log n)$
    - Un BST con altezza h ha al massimo $2^{h+1} - 1$ nodi
    - Un BST con altezza h ha almeno 2^h nodi
- Qual è l'altezza media di un BST?
  - Caso generale (inserimenti e rimozioni)
    - Difficile da analizzare
  - Caso *facile*: BST costruito da n inserimenti *casuali*
    - E' possibile dimostrare che $h = O(\log n)$

Riassunto

	SEARCH	INSERT	DELETE
Array ordinati	$O(\log n)$	$O(n)$	$O(n)$
Liste concatenate	$O(n)$	$O(1)$	$O(n)$
Alberi BST	$O(h)$	$O(h)$	$O(h)$

- I costi si riferiscono tutti al caso pessimo
- Nota: $h = O(n)$ è l'altezza dell'albero