STRUTTURE DATI
-ALBERI-

Alberi binari

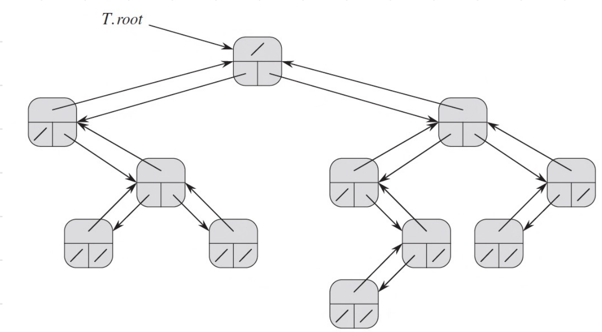
Utilizzano gli attributi p, left, right per memorizzare i puntatori al padre, al figlio sinistro, al figlio destro.

```
Anticipata(x)
  if x != NULL
    print(x.key)
    Anticipata(x.left)
    Anticipata(x.right)
```

```
Posticipata(x)
  if x != NULL
    Posticipata(x.left)
    Posticipata(x.right)
    print(x.key)
```

```
Simmetrica(x)
  if x != NULL
    Simmetrica(x.left)
    print(x.key)
    Simmetrica(x.right)
```

Le seguenti visite hanno tutte complessità $O(n)$



Albero binario completo: Ogni nodo interno ha esattamente due figli non vuoti

Albero binario completamente bilanciato: tutte le foglie hanno la stessa profondità

Albero binario bilanciato: se con n nodi, l'altezza è $O(\log n)$

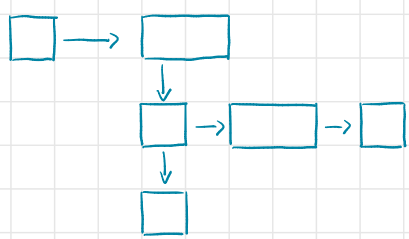
Albero binario quasi completo: Se i nodi di profondità minore di h formano un albero completamente bilanciato

Albero binario bilanciato in altezza: Se le altezze dei sottoalberi sinistro e destro di ogni nodo differiscono al più di un'unità

Possono essere rappresentati tramite array oppure liste concatenate

Vantaggi Array
- Accesso diretto ai nodi

Svantaggi Array
- Spreco di memoria
- Operazioni di Insert e Delete complesse



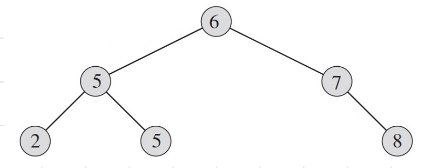
Vantaggi Liste Collegate
- Nessuno spreco di memoria
- Operazioni di Insert e Delete

Svantaggi Liste Collegate
- Nessun accesso diretto
- Memorie aggiuntive per memorizzare figli

Alberi binari di ricerca

Struttura dati che rispetta alcune proprietà ed è organizzata con liste concatenate

Proprietà: $tree.key \leq dx.key \land tree.key \geq sx.key$



grazie ad un semplice algoritmo di attraversamento simmetrico è possibile elencare in ordine tutte le chiavi

```
INORDER-TREE-WALK(x)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    stampa x.key
    INORDER-TREE-WALK(x.right)
```

la complessità di questo algoritmo è $\Theta(n)$ in quanto la procedura viene chiamata esattamente 2 volte per ogni nodo

Ricerca

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

Minimo e Massimo

```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

```
TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```

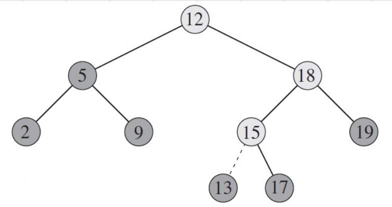
Successore e predecessore

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

Inserimento - $O(h)$

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
      y = x
      if z.key < x.key
            x = x.left
      else x = x.right
z.p = y
if y == NIL
      T.root = z          // L'albero T era vuoto
elseif z.key < y.key
      y.left = z
else y.right = z
```



procedura che inserisce un nuovo valore v, preso in input come z e inserito dentro T

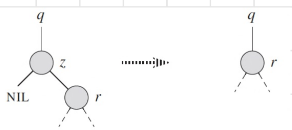
Cancellazione

La seguente procedura ha 3 casi base

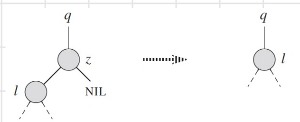
- Se z non ha figli
- Se z ha solo un figlio
- Se z ha due figli

Casistiche:

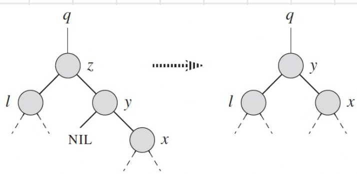
1



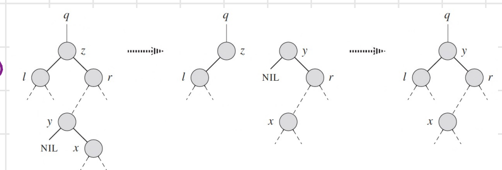
2



3



4



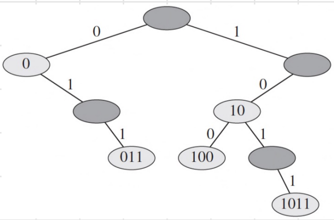

```
TREE-DELETE(T, z)
if z.left == NIL
      TRANSPLANT(T, z, z.right)
elseif z.right == NIL
      TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
      if y.p ≠ z
            TRANSPLANT(T, y, y.right)
            y.right = z.right
            y.right.p = y
      TRANSPLANT(T, z, y)
      y.left = z.left
      y.left.p = y
```

```
TRANSPLANT(T, u, v)
if u.p == NIL
      T.root = v
elseif u == u.p.left
      u.p.left = v
else u.p.right = v
if v ≠ NIL
      v.p = u.p
```

La seguente procedura sposta il sottoalbero in un altra posizione

Radix tree

Date due stringhe $a = a_0 \dots a_p \wedge b = b_0 \dots b_q$ appartengono a un insieme ordinato di caratteri.
La stringa a è lessicograficamente minore della stringa b se è soddisfa una delle seguenti condizioni



① $\forall i = 0, \dots, j-1 \wedge a_j < b_j \; \exists j \in \mathbb{R} \wedge 0 \leq j \leq \min(p, q) \mid a_i = b_i$

② $\forall i = 0, \dots, p \Rightarrow p < q \wedge a_i = b_i$

Alberi rosso-neri

È un albero binario di ricerca che soddisfa le seguenti proprietà:

① Ogni nodo è rosso o nero

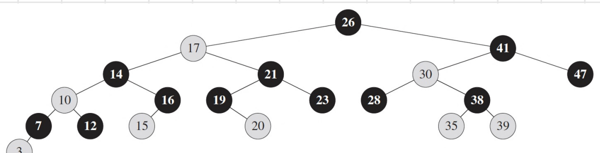
② La radice è nera

③ Se un nodo è rosso, allora entrambi i suoi figli sono neri

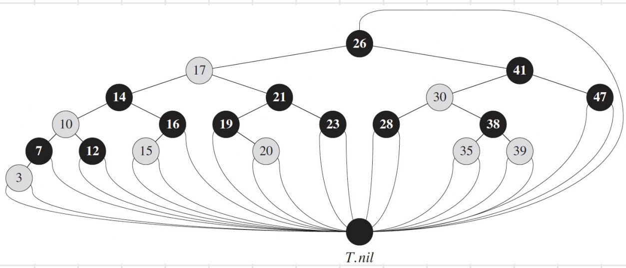
④ Per ogni nodo, tutti i cammini semplici che vanno dal nodo alle foglie sue discendenti contengono lo stesso numero di nodi neri

L'altezza massima di un albero rosso-nero con n nodi interni $2\lg(n+1)$

la complessità delle operazioni classiche sui BST rosso-neri grazie a questo lemma diventano: $O(\lg n)$

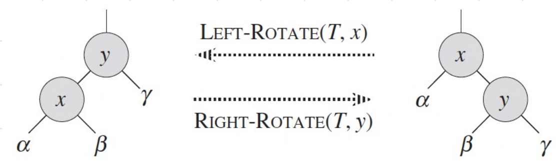


(c)



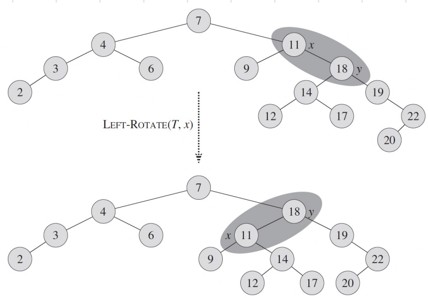
Rotazioni

Quando si effettuano operazioni sugli alberi rosso-nero si potrebbero violare le proprietà fondamentali



La struttura dei puntatori viene modificata tramite una rotazione, un'operazione locale che preserva le proprietà dei BST

```
LEFT-ROTATE(T, x)
y = x.right              // Imposta y
x.right = y.left         // Sposta il sottoalbero sinistro di y
                            nel sottoalbero destro di x.
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                // Collega il padre di x a y.
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x               // Pone x a sinistra di y.
x.p = y
```



Inserimento

L'inserimento di un nodo in un albero rosso-nero di n nodi può essere effettuato in $O(\lg n)$

```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK                       // Caso 1
            y.color = BLACK                         // Caso 1
            z.p.p.color = RED                       // Caso 1
            z = z.p.p                               // Caso 1
        else if z == z.p.right
                z = z.p                             // Caso 2
                LEFT-ROTATE(T, z)                   // Caso 2
            z.p.color = BLACK                       // Caso 3
            z.p.p.color = RED                       // Caso 3
            RIGHT-ROTATE(T, z.p.p)                  // Caso 3
    else (come la clausola then
            con “right” e “left” scambiati)
T.root.color = BLACK
```

La procedura RB-Insert-Fixup che ricolora i nodi ed effettua delle rotazioni

Cancellazione

Operazione che può essere effettuata in tempo $O(\log n)$ utilizzando le sub-routine transplant

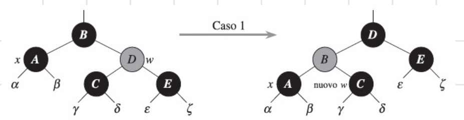
```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

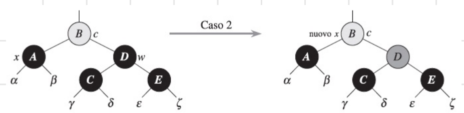
```
RB-TRANSPLANT(T, u, v)
if u.p == T.nil
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
v.p = u.p
```

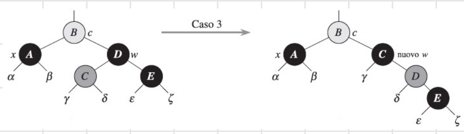
Casistiche:

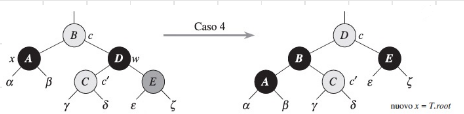
1. il fratello w di x è rosso
2. il fratello w di x è nero ed entrambi i figli di w sono neri
3. il fratello w di x è nero, il figlio sx di w è rosso e il figlio dx di w è nero
4. il fratello w di x è nero e il figlio destro di w è rosso

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                          // Caso 1
            x.p.color = RED                          // Caso 1
            LEFT-ROTATE(T, x.p)                      // Caso 1
            w = x.p.right                            // Caso 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                            // Caso 2
            x = x.p                                  // Caso 2
        else if w.right.color == BLACK
                w.left.color = BLACK                 // Caso 3
                w.color = RED                        // Caso 3
                RIGHT-ROTATE(T, w)                   // Caso 3
                w = x.p.right                        // Caso 3
            w.color = x.p.color                      // Caso 4
            x.p.color = BLACK                        // Caso 4
            w.right.color = BLACK                    // Caso 4
            LEFT-ROTATE(T, x.p)                      // Caso 4
            x = T.root                               // Caso 4
    else (come la clausola then con "right" e "left" scambiati)
x.color = BLACK
```



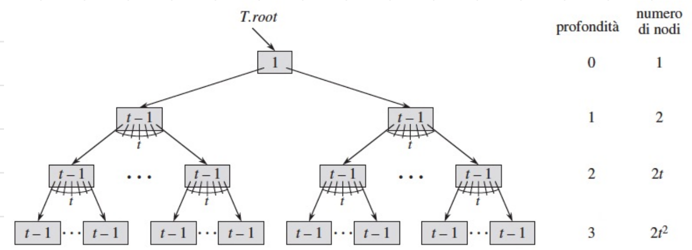






B-Alberi

Alberi binari di ricerca bilanciati organizzati per operare sui dischi o altre unità di memoria. Sono simili agli alberi rosso-neri ma più efficienti nel minimizzare le operazioni su dischi e possono avere molti più figli.



Proprietà

- Ogni nodo x ha le seguenti proprietà:
  - $x.n$ = numero di chiavi memorizzate in x
  - $x.key_1, \dots, x.key_{x.n}$ = le n chiavi, memorizzate in ordine non crescente
  - $x.leaf$ = valore booleano per controllare se x è una foglia (true) o un nodo interno (false)
- Ogni nodo interno contiene $x.n+1$ puntatori ai suoi figli
- Le chiavi $x.key_i$ separano gli intervalli delle chiavi memorizzate in ciascun sottoalbero
- tutte le foglie hanno la stessa profondità, ovvero l'altezza dell'albero
- Ogni nodo tranne la radice, deve avere $t-1$ chiavi
- Ogni nodo può contenere al massimo $2t-1$ chiavi

Ricerca

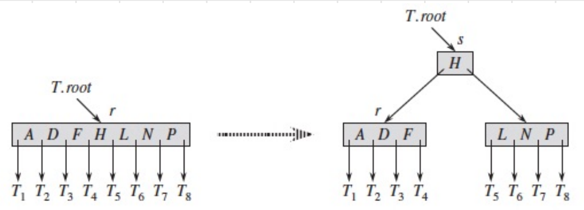
```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key_i
    i = i + 1
if i ≤ x.n and k == x.key_i
    return (x, i)
elseif x.leaf
    return NIL
else DISK-READ(x.c_i)
    return B-TREE-SEARCH(x.c_i, k)
```

Creazione B-Albero vuoto

```
B-TREE-CREATE(T)
x = ALLOCATE-NODE()
x.leaf = TRUE
x.n = 0
DISK-WRITE(x)
T.root = x
```

Dividere un nodo in un B-Albero

```
B-TREE-SPLIT-CHILD(x, i)
z = ALLOCATE-NODE()
y = x.c_i
z.leaf = y.leaf
z.n = t - 1
for j = 1 to t - 1
    z.key_j = y.key_{j+t}
if not y.leaf
    for j = 1 to t
        z.c_j = y.c_{j+t}
y.n = t - 1
for j = x.n + 1 downto i + 1
    x.c_{j+1} = x.c_j
x.c_{i+1} = z
for j = x.n downto i
    x.key_{j+1} = x.key_j
x.key_i = y.key_t
x.n = x.n + 1
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```



Divisione della radice con $t=4$

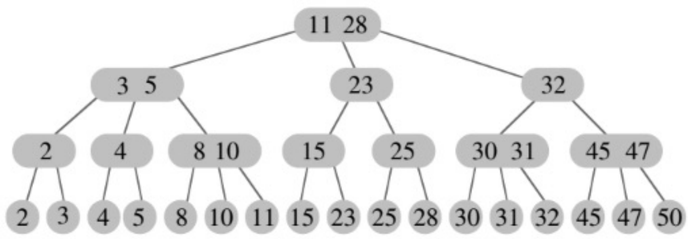
Il nodo radice viene diviso in due e viene creato un nuovo nodo radice s che contiene la chiave mediana di r e ha le due metà di r come figli

Inserire una chiave con un singolo passaggio

```
B-TREE-INSERT(T, k)
r = T.root
if r.n == 2t - 1
    s = ALLOCATE-NODE()
    T.root = s
    s.leaf = FALSE
    s.n = 0
    s.c_1 = r
    B-TREE-SPLIT-CHILD(s, 1)
    B-TREE-INSERT-NONFULL(s, k)
else B-TREE-INSERT-NONFULL(r, k)
```

Alberi 2-3

Albero in cui ogni nodo interno ha 2 o 3 figli e tutti i cammini radice-foglie hanno la stessa lunghezza



Ricerca

La complessità del seguente è proporzionale all'altezza: $\Theta(\log n)$

algoritmo search(*radice* r *di un albero* 2-3, *chiave* x) $\rightarrow$ *elem*

    if (r è una foglia) **then**
        if ($x = chiave(r)$) **then return** $elem(r)$
        else return null
    $v_i \leftarrow i$-esimo figlio di r
    if ($x \leq S[r]$) **then return** search(v_1, x)
    else if (r ha due figli oppure $x \leq M[r]$) **then return** search(v_2, x)
        else return search(v_3, x)

Inserimento

Quando dobbiamo inserire un elemento all'interno del nostro albero 2-3, ci possiamo trovare in due casistiche

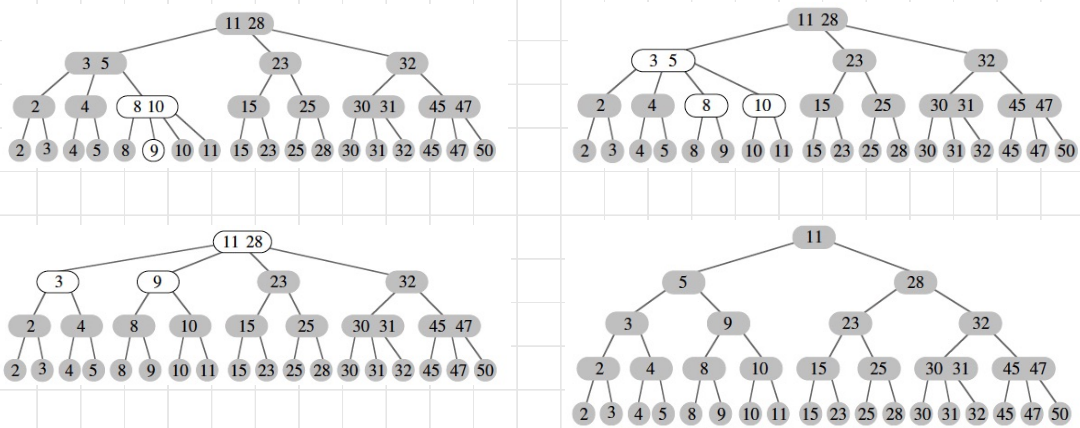
Identifichiamo un nodo v sul penultimo livello, che dovrebbe diventare genitore di u

- v ha due figli: aggiungiamo u come nuovo figlio
- v ha tre figli: separiamo il nodo in due, utilizzando la procedura split

algoritmo split(*nodo* v)

    crea un nuovo nodo w
    sia v_i l'i-esimo figlio di v in T, $1 \leq i \leq 4$
    rendi v_1 e v_2 figli sinistro e destro di w
    if ($parent[v]$ = null) **then**
        crea un nuovo nodo r
        rendi w e v figli sinistro e destro di r
    else
        aggiungi w come figlio di $parent[v]$ immediatamente precedente a v
        if ($parent[v]$ ha quattro figli) **then** split($parent[v]$)

Inserimento del nodo 9 mediante procedura split con complessità $O(\log n)$



Cancellazione

È una procedura simmetrica a quella di inserimento

Casistiche:

1. v è la radice
2. il padre di v ha tre figli
3. il padre di v ha due figli