ESERCIZI SUPPLEMENTARI - ALBERI BINARI DI RICERCA (ABR)

1. ABR CON PREDECESSORE

Problema: Variante di ABR dove ogni nodo x ha campo `x.pred` (predecessore) invece di `x.p` (padre).

Struttura Nodo

```
struct Node {
    int key;
    Node* pred;   // predecessore invece di parent
    Node* left;
    Node* right;
}
```

Insert con Predecessore

```
Insert(T, z)
    y = nil        // father del nuovo nodo
    w = nil        // successor del nuovo nodo
    x = T.root

    // Discesa nell'albero
    while x != nil
        y = x
        if z.key < x.key
            w = x
            x = x.left
        else
            x = x.right

    // z sarà figlio di y
    if y == nil
        T.root = z
        z.pred = nil
    else if z.key < y.key
        y.left = z
        z.pred = y.pred
    else
        y.right = z
        z.pred = y

    // Aggiorna predecessore di w se esiste
```

```
    if w != nil
        w.pred = z

    z.left = nil
    z.right = nil
```

Correttezza

Proprietà chiave:

- Se z diventa figlio sinistro di y, allora pred(z) = pred(y)
- Se z diventa figlio destro di y, allora pred(z) = y
- Il successore w di z (se esiste) deve aggiornare il suo predecessore a z

Invariante durante discesa:

- y è il padre candidato per z
- w è il più piccolo nodo maggiore di z.key incontrato finora (= successore di z)

Complessità

O(h) dove h è l'altezza dell'albero

- Discesa: O(h)
- Aggiornamenti puntatori: O(1)
- Totale: O(h)

---

2. ABR CON CAMPO EVEN

Problema: Ogni nodo x ha campo booleano `x.even` che indica se la somma delle chiavi nel sottoalbero radicato in x è pari.

Struttura Nodo

```
struct Node {
    int key;
    bool even;    // true se somma sottoalbero è pari
    Node* p;
    Node* left;
    Node* right;
}
```

Insert Modificato

```
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
    else if z.key < y.key
        y.left = z
    else
        y.right = z

    z.left = nil
    z.right = nil
    z.even = (z.key % 2 == 0)  // inizializza even per foglia

    // Aggiorna even risalendo verso radice
    updateEven(z.p)

updateEven(x)
    while x != nil
        sumLeft = getSumParity(x.left)
        sumRight = getSumParity(x.right)
        totalSum = x.key + sumLeft + sumRight
        x.even = (totalSum % 2 == 0)
        x = x.p

getSumParity(x)
    if x == nil
        return 0

    sum = x.key
    if x.left != nil
        sum += getSumLeft(x.left)
    if x.right != nil
        sum += getSumRight(x.right)
    return sum
```

Problema: getSumParity costa O(n) per ogni nodo!

Soluzione Efficiente

Mantenere anche `x.sum` (somma sottoalbero):

```
struct Node {
    int key;
    bool even;
    int sum;      // somma chiavi sottoalbero
    Node* p;
    Node* left;
    Node* right;
}

Insert(T, z)
    // ... discesa come prima ...

    z.sum = z.key
    z.even = (z.key % 2 == 0)

    // Aggiorna sum ed even risalendo
    x = z.p
    while x != nil
        x.sum = x.key
        if x.left != nil
            x.sum += x.left.sum
        if x.right != nil
            x.sum += x.right.sum
        x.even = (x.sum % 2 == 0)
        x = x.p
```

Complessità con sum: O(h)

Delete Modificato

```
Delete(T, z)
    // Esegui delete standard
    TreeDelete(T, z)

    // Aggiorna sum ed even per antenati del nodo rimosso
    if z.p != nil
        updateAncestors(z.p)

updateAncestors(x)
    while x != nil
        x.sum = x.key
        if x.left != nil
            x.sum += x.left.sum
        if x.right != nil
            x.sum += x.right.sum
```

```
        x.even = (x.sum % 2 == 0)
        x = x.p
```

Complessità

- **Insert**: O(h)
- **Delete**: O(h)

---

3. ISABR - TRE VERSIONI

Problema: Verificare se un albero binario T è un ABR.

Versione 1: $O(n^2)$ - Controllo Locale

```
isABR(T)
    return isABRRec(T.root)

isABRRec(x)
    if x == nil
        return true

    // Verifica proprietà ABR per nodo x
    if x.left != nil
        if max(x.left) > x.key
            return false
    if x.right != nil
        if min(x.right) < x.key
            return false

    // Ricorri sui sottoalberi
    return isABRRec(x.left) and isABRRec(x.right)

max(x)  // O(n) per sottoalbero
    if x == nil
        return -∞
    maxVal = x.key
    if x.left != nil
        maxVal = max(maxVal, max(x.left))
    if x.right != nil
        maxVal = max(maxVal, max(x.right))
    return maxVal
```

Complessità: $O(n^2)$

- Per ogni nodo: O(n) per trovare min/max sottoalberi

- n nodi → $O(n^2)$

Versione 2: O(n log n) - Trova Estremi

```
isABR(T)
    return isABRRec2(T.root)

isABRRec2(x)
    if x == nil
        return true

    // Trova min e max in O(h) invece di O(n)
    if x.left != nil
        if treeMax(x.left).key > x.key
            return false
    if x.right != nil
        if treeMin(x.right).key < x.key
            return false

    return isABRRec2(x.left) and isABRRec2(x.right)

treeMin(x)  // O(h)
    while x.left != nil
        x = x.left
    return x

treeMax(x)  // O(h)
    while x.right != nil
        x = x.right
    return x
```

Complessità: $O(n \log n)$ per albero bilanciato, $O(n^2)$ caso pessimo

Versione 3: O(n) - Visita Simmetrica

Approccio 1: Verifica sequenza ordinata

```
isABR(T)
    prev = -∞
    return inOrderCheck(T.root, prev)

inOrderCheck(x, prev)
    if x == nil
        return true

    // Visita sottoalbero sinistro
    if not inOrderCheck(x.left, prev)
        return false
```

```
    // Verifica ordine
    if x.key <= prev
        return false
    prev = x.key

    // Visita sottoalbero destro
    return inOrderCheck(x.right, prev)
```

Approccio 2: Ricorsivo con intervalli

```
isABR(T)
    return isABRRange(T.root, -∞, +∞)

isABRRange(x, min, max)
    if x == nil
        return true

    // Verifica che chiave sia nell'intervallo
    if x.key <= min or x.key >= max
        return false

    // Ricorri con intervalli ristretti
    return isABRRange(x.left, min, x.key) and
           isABRRange(x.right, x.key, max)
```

Complessità: O(n)

- Ogni nodo visitato una volta
- Operazioni per nodo: O(1)

---

4. BST DA ARRAY ORDINATO

Problema: Dato array A[1..n] ordinato crescente, costruire ABR di altezza minima.

Algoritmo

```
BST(A)
    return BST_rec(A, 1, A.length)

BST_rec(A, p, q)
    if p > q
        return nil

    m = floor((p + q) / 2)  // elemento medio
```

```
    x = new Node
    x.key = A[m]
    x.left = BST_rec(A, p, m-1)
    x.right = BST_rec(A, m+1, q)

    // Imposta parent
    if x.left != nil
        x.left.p = x
    if x.right != nil
        x.right.p = x

    return x
```

Correttezza

- Prendendo elemento medio come radice, si bilanciano i sottoalberi
- Sottoalbero sinistro: elementi < A[m]
- Sottoalbero destro: elementi > A[m]
- Ricorsivamente costruisce ABR bilanciato

Altezza

Per n elementi:

- $h = \lfloor \log_2 n \rfloor$ (altezza minima possibile)
- Albero risultante è completo o quasi completo

Complessità

O(n)

- Ricorrenza: $T(n) = 2 \cdot T(n/2) + \Theta(1)$
- Master Theorem (caso 1): $T(n) = \Theta(n)$
- Ogni nodo creato esattamente una volta

Esempio

Array: [1, 3, 5, 7, 9, 11, 13]

```
Passo 1: m = 4, radice = 7
         7
        / \
      [1,3,5] [9,11,13]

Passo 2: Ricorri sui sottoarray
         7
        / \
```

```
       3   11
      / \  / \
     1  5 9  13
```

Altezza: h = ⌊$\log_2 7$⌋ = 2

---

5. ABR ARRICCHITO AVG

Problema: ABR arricchito dove ogni nodo x ha:

- `x.sum` : somma chiavi sottoalbero
- `x.size` : numero nodi sottoalbero
- Operazione `avg(x)` in O(1)

Struttura Nodo

```
struct Node {
    int key;
    int sum;      // somma chiavi sottoalbero
    int size;     // numero nodi sottoalbero
    Node* p;
    Node* left;
    Node* right;
}
```

Operazione avg

```
avg(x)
    if x == nil or x.size == 0
        error "nodo vuoto"
    return x.sum / x.size
```

Complessità: O(1)

Insert Modificato

```
Insert(T, z)
    y = nil
    x = T.root

    // Discesa con aggiornamento campi
    while x != nil
        y = x
        x.sum += z.key     // aggiorna somma durante discesa
```

```
        x.size++           // aggiorna size durante discesa

        if z.key < x.key
            x = x.left
        else
            x = x.right

    z.p = y
    if y == nil
        T.root = z
    else if z.key < y.key
        y.left = z
    else
        y.right = z

    z.left = nil
    z.right = nil
    z.sum = z.key
    z.size = 1
```

Correttezza

Invariante: Durante la discesa, per ogni nodo x attraversato:

- x.sum include già la chiave da inserire
- x.size è già incrementato

Questo funziona perché:

1. Aggiorniamo durante la discesa (prima di inserire)
2. Il nuovo nodo sarà nel sottoalbero di ogni nodo attraversato
3. sum e size sono corretti dopo l'inserimento

Complessità

O(h)

- Discesa: O(h) con aggiornamenti O(1) per nodo
- Totale: O(h)

Delete (Idea)

```
Delete(T, z)
    // Salva valori
    delKey = z.key

    // Esegui delete standard
    TreeDelete(T, z)
```

```
    // Risali aggiornando sum e size
    x = z.p
    while x != nil
        x.sum -= delKey
        x.size--
        updateFromChildren(x)  // ricalcola da figli
        x = x.p
```

Complessità Delete: O(h)

RIEPILOGO COMPLESSITÀ

Operazione	Complessità	Note
ABR con pred - Insert	O(h)	Aggiorna pred e succ
ABR even - Insert/Delete	O(h)	Con campo sum
isABR v1	$O(n^2)$	Controllo locale
isABR v2	O(n log n)	Con min/max O(h)
isABR v3	O(n)	Visita simmetrica
BST da array	O(n)	Costruzione ottimale
ABR avg - avg	O(1)	Calcolo media
ABR avg - Insert	O(h)	Aggiornamento campi

Note Importanti

- **h = O(log n)** per alberi bilanciati
- **h = O(n)** caso pessimo (albero degenerato)
- ABR arricchiti: aggiungere campi senza aumentare complessità asintotica