ESERCIZI ALBERI BINARI DI RICERCA (BST)

Guida completa con soluzioni per esame

DEFINIZIONI FONDAMENTALI

Albero Binario di Ricerca (BST)

- Ogni nodo x ha campi: x.key, x.left, x.right, x.p
- Proprietà BST: per ogni nodo x
  - x.left.key ≤ x.key
  - x.right.key ≥ x.key

Operazioni base

- Ricerca, Min, Max: O(h)
- Insert, Delete: O(h)
- InOrder (visita simmetrica): O(n)

CATEGORIA 1: ARRICCHIMENTO BST CON CAMPI AGGIUNTIVI

ES 1.1 - BST con campo MIN

Traccia: Arricchire BST dove ogni nodo x ha x.min = minimo chiavi nel sottoalbero radicato in x

Soluzione:

```
Insert(T,z)
1. y = nil
2. x = T.root
3. while x ≠ nil
4.     y = x
5.     if z.key < x.key
6.         if z.key < x.min
7.             x.min = z.key
8.         x = x.left
9.     else
10.        x = x.right
11. z.p = y
```

```
12. if y = nil
13.     T.root = z
14. else if z.key < y.key
15.     y.left = z
16. else
17.     y.right = z
18. z.min = z.key

Left(T,x)  // Rotazione sinistra con aggiornamento min
1. y = x.right
2. x.right = y.left
3. if y.left ≠ nil
4.     y.left.p = x
5. y.p = x.p
6. if x.p = nil
7.     T.root = y
8. else if x = x.p.left
9.     x.p.left = y
10. else
11.     x.p.right = y
12. y.left = x
13. x.p = y
14. // Aggiorna min
15. x.min = min(x.key, x.left.min, x.right.min)
16. y.min = min(y.key, y.left.min, y.right.min)
```

Complessità: O(h)
Correttezza: Aggiorna min solo per nodi nel percorso dalla radice a z

---

ES 1.2 - BST con SIZE e SUM (media chiavi)

Traccia: Arricchire BST per ottenere in O(1) la media dei valori nel sottoalbero di x

Campi aggiuntivi:

- x.sum = somma chiavi sottoalbero radicato in x
- x.size = numero nodi sottoalbero radicato in x

Soluzione:

```
avg(x)
1. if x ≠ nil
2.     return x.sum / x.size
3. else
4.     error

Insert(T,z)
```

```
1. y = nil
2. x = T.root
3. while x ≠ nil
4.     y = x
5.     x.size = x.size + 1
6.     x.sum = x.sum + z.key
7.     if z.key < x.key
8.         x = x.left
9.     else
10.        x = x.right
11. z.p = y
12. if y ≠ nil
13.     if z.key < y.key
14.         y.left = z
15.     else
16.         y.right = z
17. z.sum = z.key
18. z.size = 1
19. z.left = z.right = nil
```

Complessità: O(h)

ES 1.3 - BST con grado di bilanciamento

Traccia: Arricchire BST per ottenere grado bilanciamento = $h_x / \log_2(n_x + 1)$

Campi aggiuntivi:

- x.h = altezza sottoalbero
- x.size = numero nodi

Soluzione:

```
bal(x)
1. if x ≠ nil
2.     return x.h / log_2(x.size + 1)
3. else
4.     error

Insert(T,z)
1. y = nil
2. x = T.root
3. while x ≠ nil
4.     y = x
5.     x.size = x.size + 1
6.     if z.key < x.key
7.         x = x.left
```

```
8.     else
9.         x = x.right
10. z.p = y
11. if y ≠ nil
12.     if z.key < y.key
13.         y.left = z
14.     else
15.         y.right = z
16. z.size = 1
17. z.h = 1
18. z.left = z.right = nil
19. // Propaga aggiornamento altezza verso l'alto
20. updateHeight(z.p)

updateHeight(x)
1. if x = nil
2.     return
3. oldH = x.h
4. leftH = (x.left ≠ nil) ? x.left.h : 0
5. rightH = (x.right ≠ nil) ? x.right.h : 0
6. x.h = 1 + max(leftH, rightH)
7. if x.h ≠ oldH
8.     updateHeight(x.p)
```

Complessità: O(h)

CATEGORIA 2: OPERAZIONI SPECIALI SU BST

ES 2.1 - Predecessore in BST

Traccia: Dato nodo x, restituire predecessore di x (o nil se non esiste)

Soluzione:

```
pred(x)
1. if x.left ≠ nil
2.     return max(x.left)
3. else
4.     y = x.p
5.     while y ≠ nil and x = y.left
6.         x = y
7.         y = y.p
8.     return y

max(x)
1. while x.right ≠ nil
```

```
2.      x = x.right
3. return x
```

Complessità: O(h)
Casi:

- Se x ha figlio sinistro → massimo del sottoalbero sinistro
- Altrimenti → primo antenato per cui x è nel sottoalbero destro

---

ES 2.2 - Insert ricorsiva

Traccia: Realizzare versione ricorsiva di Insert(T,z)

Soluzione:

```
Insert(T,z)
1. T.root = InsertRec(T.root, z, nil)

InsertRec(x, z, parent)
1. if x = nil
2.      z.p = parent
3.      return z
4. else
5.      if z.key < x.key
6.          x.left = InsertRec(x.left, z, x)
7.      else
8.          x.right = InsertRec(x.right, z, x)
9.      return x
```

Complessità: O(h)

ES 2.3 - BST da array ordinato con altezza minima

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

Soluzione:

```
BST(A)
1. return BST-rec(A, 1, n)

BST-rec(A, p, q)
1. if p ≤ q
2.      m = floor((p+q)/2)
3.      x = mknode(A[m])
```

```
4.     x.left = BST-rec(A, p, m-1)
5.     x.right = BST-rec(A, m+1, q)
6. else
7.     x = nil
8. return x
```

Complessità: T(n) = 2T(n/2) + Θ(1) = O(n)
Correttezza: Scegliendo elemento centrale come radice otteniamo BST bilanciato

---

CATEGORIA 3: VERIFICA PROPRIETÀ BST

ES 3.1 - Verificare se array rappresenta BST

Traccia: Dato array A[1..n] interpretato come heap (A[2i] figlio sx, A[2i+1] figlio dx), verificare se è BST

Soluzione 1 - O(n log n):

```
IsABR(A)
1. n = A.length
2. i = n
3. isABR = true
4. while i > 1 and isABR
5.     j = i
6.     while j > 1
7.         if even(j)  // figlio sinistro
8.             isABR = isABR and A[i] ≤ A[j/2]
9.         else  // figlio destro
10.            isABR = isABR and A[i] ≥ A[j/2]
11.        j = j/2
12.    i = i-1
13. return isABR
```

Soluzione 2 - O(n) con min/max:

```
IsABR1(A,n)
1. isABR, m, M = IsABR1_rec(A, n, 1)
2. return isABR

IsABR1_rec(A, n, i)
1. if i > n
2.     return (true, +∞, -∞)
3. else
4.     isABRL, mL, ML = IsABR1_rec(A, n, 2*i)
5.     isABRR, mR, MR = IsABR1_rec(A, n, 2*i+1)
6.     return (isABRL and isABRR and A[i] ≥ ML and A[i] ≤ mR,
```

```
7.              min(mL, mR, A[i]),
8.              max(ML, MR, A[i]))
```

Complessità: O(n)

CATEGORIA 4: VISITE E ANALISI STRUTTURALI

ES 4.1 - Verificare media chiavi

Traccia: Verificare se per ogni nodo con discendenti, chiave ≥ media chiavi discendenti

Soluzione:

```
avgTree(T)
1. if T.root = nil
2.     return true
3. ok, sum, count = avgTree_rec(T.root)
4. return ok

avgTree_rec(x)
1. if x = nil
2.     return (true, 0, 0)
3. okL, sumL, countL = avgTree_rec(x.left)
4. okR, sumR, countR = avgTree_rec(x.right)
5. totalSum = sumL + sumR + x.key
6. totalCount = countL + countR + 1
7. // Verifica condizione solo se ha discendenti
8. if countL + countR > 0
9.     avg = (sumL + sumR) / (countL + countR)
10.    ok = okL and okR and (x.key ≥ avg)
11. else
12.    ok = okL and okR
13. return (ok, totalSum, totalCount)
```

Complessità: O(n)

ES 4.2 - Albero k-bilanciato

Traccia: Albero k-bilanciato se tutti cammini nil-terminated dalla radice differiscono al più di k

Soluzione:

```
bal(T, k)
1. min, max = nilTerm(T.root)
```

```
2. return max - min ≤ k

nilTerm(x)
1. if x = nil
2.     return (0, 0)
3. else
4.     left_min, left_max = nilTerm(x.left)
5.     right_min, right_max = nilTerm(x.right)
6.     return (min(left_min, right_min) + 1,
7.             max(left_max, right_max) + 1)
```

Complessità: O(n)

CATEGORIA 5: IMPOSSIBILITÀ ALGORITMI LINEARI

ES 5.1 - Impossibilità costruzione ABR lineare

Domanda: Esiste algoritmo O(n) per costruire ABR quasi completo da array A?

Risposta: NO

Dimostrazione:

1. Se esistesse algoritmo O(n) per costruire ABR quasi completo
2. E posso visitare ABR in ordine crescente in O(n) (InOrder)
3. Allora potrei ordinare in O(n)
4. Ma abbiamo Ω(n log n) come limite inferiore per ordinamento basato su confronti
5. **Contraddizione** → algoritmo non può esistere

TEMPLATE RISOLUZIONE ESERCIZI ABR

Step 1: Identificare tipo esercizio

- Arricchimento con campi → aggiorna durante Insert/Delete
- Verifica proprietà → visita ricorsiva con raccolta info
- Operazione speciale → adatta algoritmo base

Step 2: Invarianti comuni

- Durante Insert: aggiorna solo nodi nel percorso radice-inserimento
- Durante visita: usa ricorsione per raccogliere info da sottoalberi
- Complessità: O(h) per operazioni su percorsi, O(n) per visite complete

Step 3: Correttezza

- Insert: campo aggiornato per tutti nodi che conterranno z nel sottoalbero
- Rotazioni: aggiorna solo nodi la cui struttura cambia
- Visite: induzione sulla dimensione sottoalbero

CHECKLIST ESAME

✓ Ricorda definizione BST: left ≤ root ≤ right
✓ Insert standard costa O(h), visita completa O(n)
✓ Per arricchimento: aggiungi campi e aggiorna durante Insert
✓ Rotazioni: aggiorna campi solo per nodi modificati
✓ Verifica proprietà: visita ricorsiva raccogliendo info
✓ Impossibilità lineari: contraddice limite ordinamento

ERRORI COMUNI DA EVITARE

❌ Non aggiornare campi extra durante Insert
❌ Confondere O(h) con O(log n) (vero solo se bilanciato)
❌ Dimenticare caso base in ricorsioni
❌ Non gestire caso T.root = nil
❌ Non aggiornare parent pointer durante rotazioni
❌ Pensare che esista algoritmo lineare per costruzione ABR da array non ordinato