
DOMANDA B (7 punti)

Testo: Scrivere una funzione `toTree(A)` che dato un array A organizzato a max-heap (dimensione A.heapSize), lo trasforma in un albero binario realizzato con strutture linked, ancora organizzato a max-heap e ritorna la radice. Il nuovo albero è costituito da nodi x con campi x.p (parent), x.k (chiave), x.l e x.r (figlio sinistro e destro). Per allocare un nuovo nodo si assuma di avere un costruttore `node()`. Valutare la complessità.

Soluzione

```
toTree(A)
    if A.heapSize == 0
        return nil
    return toTreeRec(A, 1, nil)

toTreeRec(A, i, parent)
    if i > A.heapSize
        return nil
    x = node()
    x.k = A[i]
    x.p = parent
    x.l = toTreeRec(A, 2*i, x)
    x.r = toTreeRec(A, 2*i+1, x)
    return x
```

Correttezza: Per induzione strutturale sull'albero. Per ogni nodo in posizione i:

- La chiave viene assegnata da A[i]
- Il parent viene passato correttamente dal chiamante
- I figli sono costruiti ricorsivamente nelle posizioni 2i (left) e 2i+1 (right)
- La proprietà di max-heap è preservata poiché l'array di partenza la soddisfa

Complessità: $\Theta(n)$ dove n = A.heapSize, poiché si visita ogni nodo esattamente una volta.

DOMANDA B (5 punti) - Definizione BST

Testo: Dare la definizione di albero binario di ricerca. Specificare l'albero ottenuto inserendo, con la procedura vista a lezione, a partire da un albero vuoto, i nodi aventi le seguenti chiavi: 5, 4, 8, 6, 12, 7. Si supponga che dall'albero così ottenuto si cancelli il nodo con chiave 5 e si

indichi l'albero ottenuto. Sia per gli inserimenti che per la cancellazione, motivare sinteticamente il risultato ottenuto.

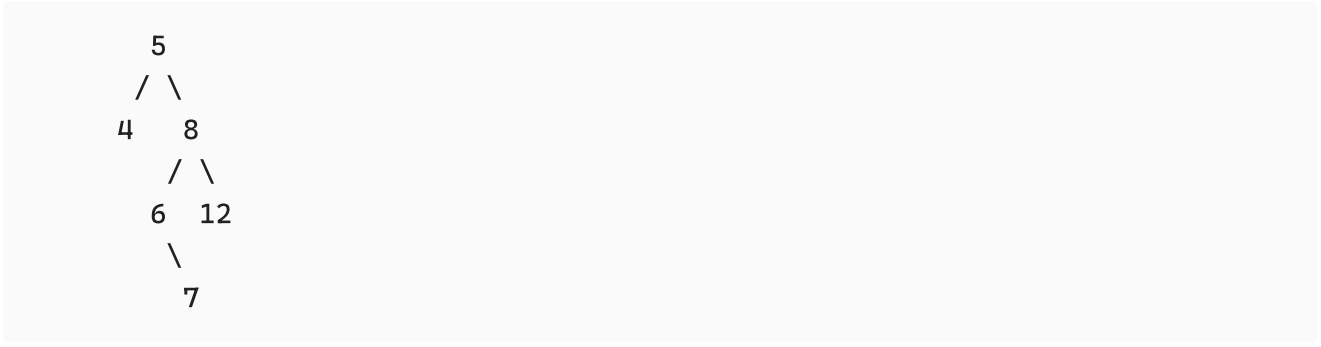
Soluzione

Definizione: Un albero binario T è un albero binario di ricerca (BST) se per ogni nodo x in T:

- Se y è un nodo nel sottoalbero sinistro di x, allora y.key ≤ x.key
- Se z è un nodo nel sottoalbero destro di x, allora z.key ≥ x.key

Inserimenti sequenziali:

1. Inserisci 5: radice
2. Inserisci 4: 4 < 5 → va a sinistra di 5
3. Inserisci 8: 8 > 5 → va a destra di 5
4. Inserisci 6: 6 > 5 → destra; 6 < 8 → sinistra di 8
5. Inserisci 12: 12 > 5 → destra; 12 > 8 → destra di 8
6. Inserisci 7: 7 > 5 → destra; 7 < 8 → sinistra; 7 > 6 → destra di 6



Cancellazione del nodo 5: Il nodo 5 ha due figli. Si applica la procedura standard:

1. Trova il successore di 5: minimo del sottoalbero destro = 6
2. Sostituisci 5 con 6
3. Cancella 6 dalla sua posizione originale (ha al più un figlio destro: 7)
4. 7 prende il posto di 6

DOMANDA C (5 punti)

Testo: Scrivere una funzione `diff(T)` che dato in input un albero binario di ricerca T determina la massima differenza di lunghezza tra due cammini che vanno dalla radice ad un sottoalbero vuoto. Ad esempio sull'albero ottenuto inserendo 1, 2 e 3 produce 2, su quello ottenuto inserendo 2, 1, 3 produce 0. Valutarne la complessità.

Soluzione

```
diff(T)
    if T.root == nil
        return 0
    h_min, h_max = minMaxHeight(T.root)
    return h_max - h_min

minMaxHeight(x)
    if x == nil
        return 0, 0
    h_min_l, h_max_l = minMaxHeight(x.left)
    h_min_r, h_max_r = minMaxHeight(x.right)
    h_min = 1 + min(h_min_l, h_min_r)
    h_max = 1 + max(h_max_l, h_max_r)
    return h_min, h_max
```

Correttezza: La funzione calcola ricorsivamente per ogni nodo:

- L'altezza minima: 1 + minimo delle altezze minime dei figli
- L'altezza massima: 1 + massimo delle altezze massime dei figli
- La differenza alla radice fornisce la massima differenza di lunghezza

Complessità: $\Theta(n)$ dove n è il numero di nodi, poiché ogni nodo viene visitato esattamente una volta nella ricorsione.

ESERCIZIO 1 (7 punti)

Testo: Sia T un albero binario i cui nodi x hanno i campi x.left, x.right, x.key. L'albero si dice k-bounded, per un certo valore k, se per ogni nodo x la somma delle chiavi lungo ciascun cammino da x ad una foglia è minore o uguale a k. Scrivere una funzione `Bound(T,k)` che dato in input un albero T e un valore k verifica se T è k-bounded e ritorna un corrispondente valore booleano. Valutarne la complessità.

Soluzione

```
Bound(T, k)
    if T.root == nil
        return true
```

```
    return BoundRec(T.root, k)

BoundRec(x, k)
    if x == nil
        return true
    // Verifica se x è una foglia
    if x.left == nil and x.right == nil
        return x.key <= k
    // Propaga il bound ridotto ai sottoalberi
    new_k = k - x.key
    if new_k < 0
        return false
    return BoundRec(x.left, new_k) and BoundRec(x.right, new_k)
```

Correttezza: Per ogni nodo x, la funzione:

1. Se x è una foglia, verifica che x.key ≤ k
2. Altrimenti, sottrae x.key da k e verifica ricorsivamente i sottoalberi con k' = k - x.key
3. Ritorna false appena trova un cammino che eccede k

Complessità: O(n) dove n è il numero di nodi, poiché nel caso peggiore si visitano tutti i nodi.

ESERCIZIO 1 (10 punti) - Union

Testo: Realizzare una funzione `union(A1,A2,n)` che dati due array di interi A1 e A2, organizzati a max-heap con capacità n, restituisce un nuovo array A, ancora organizzato a max-heap con capacità 2n, che contiene l'unione insiemistica dei valori contenuti in A1 e A2. Si assuma che A1 e A2 non contengano duplicati e si faccia in modo anche anche l'array ottenuto come unione non contenga duplicati. Se A1 contiene i valori 3,1,2 e A2 contiene i valori 5,2 allora l'unione A conterrà i valori 5,3,1,2, possibilmente non in questo ordine, ovvero l'elemento 2 non è duplicato. Valutare la complessità della funzione definita.

Soluzione

```
union(A1, A2, n)
    A = new array[1..2n]
    A.heapSize = 0
    // Inserisci tutti gli elementi di A1
    for i = 1 to A1.heapSize
        A.heapSize++
        A[A.heapSize] = A1[i]
    // Inserisci elementi di A2 che non sono duplicati
    for j = 1 to A2.heapSize
        if not contains(A, A.heapSize, A2[j])
```

```
            A.heapSize++
            A[A.heapSize] = A2[j]
    // Costruisci max-heap
    BuildMaxHeap(A)
    return A

contains(A, size, key)
    for i = 1 to size
        if A[i] == key
            return true
    return false

BuildMaxHeap(A)
    for i = floor(A.heapSize/2) downto 1
        MaxHeapify(A, i)
```

Correttezza:

1. Copia tutti gli elementi di A1 in A
2. Per ogni elemento di A2, verifica se è già presente (contains) e lo inserisce solo se non duplicato
3. Applica BuildMaxHeap per ristabilire la proprietà di max-heap

Complessità:

- Copia A1: $\Theta(n)$
- Copia A2 con controllo duplicati: $O(n^2)$ per i controlli contains
- BuildMaxHeap: $O(n)$
- **Totale: $O(n^2)$**

Se A contenesse duplicati, soluzioni più efficienti richiederebbero strutture dati ausiliarie come hash table (non disponibili nel contesto del corso), riducendo a $O(n \log n)$.