ESERCIZI SUPPLEMENTARI - HEAP

1. SECONDO MINIMO IN MIN-HEAP

Problema: Dato un min-heap A, restituire il secondo elemento più piccolo (successore della radice).

Osservazione chiave: In un min-heap, il secondo minimo è necessariamente uno dei due figli della radice A[2] o A[3].

Pseudocodice

```
sndmin(A)
    if A.heapsize < 2
        error "heap troppo piccolo"
    if A.heapsize == 2
        return A[2]
    else
        return min(A[2], A[3])
```

Correttezza

- La radice A[1] contiene il minimo
- I figli A[2] e A[3] sono le radici dei due sottoalberi
- Per la proprietà di min-heap, tutti gli elementi nei sottoalberi sono ≥ delle rispettive radici
- Quindi il secondo minimo globale deve essere min(A[2], A[3])

Complessità

Θ(1) - operazioni costanti

---

2. VERIFICA MAX-HEAP

Problema: Verificare se un array A è organizzato a max-heap.

Versione Ricorsiva

```
IsMaxHeap(A, i)
    if i > A.heapsize
        return true

    l = 2*i
    r = 2*i + 1
```

```
    // Verifica proprietà max-heap per nodo corrente
    if l <= A.heapsize and A[l] > A[i]
        return false
    if r <= A.heapsize and A[r] > A[i]
        return false

    // Ricorri sui sottoalberi
    return IsMaxHeap(A, l) and IsMaxHeap(A, r)

IsMaxHeap(A)
    return IsMaxHeap(A, 1)
```

Ricorrenza: T(n) = c + 2·T(n/2)
Complessità: O(n) per Master Theorem (caso 1)

Versione Iterativa

```
IsMaxHeap(A)
    for i = A.heapsize downto 2
        if A[i] > A[i/2]  // confronto con padre
            return false
    return true
```

Complessità: Θ(n) - scorre tutto l'array una volta

Correttezza

- **Invariante**: Tutti i nodi da i+1 a A.heapsize rispettano la proprietà max-heap
- **Inizializzazione**: Le foglie rispettano banalmente la proprietà
- **Mantenimento**: Se A[i] ≤ A[i/2] allora il nodo i rispetta la proprietà
- **Conclusione**: Quando i=1, tutti i nodi sono stati verificati

---

3. INTERSEZIONE DI DUE MIN-HEAP

Problema: Dati due min-heap A1 e A2 di dimensione n, restituire un min-heap A contenente l'intersezione dei valori. Se un valore appare con molteplicità diverse, mantenere il minimo numero di occorrenze.

Esempio:

- A1 = {1, 2, 2, 2}
- A2 = {1, 1, 2, 2}
- A = {1, 2, 2}

Algoritmo

```
intersect(A1, A2, n)
    allocate A[1..n]
    i = 0  // ultimo elemento occupato in A

    while (A1.heapsize > 0) and (A2.heapsize > 0)
        v1 = Min(A1)      // O(1) - guarda la radice
        v2 = Min(A2)      // O(1)

        if v1 == v2
            i++
            A[i] = v1
            ExtractMin(A1)  // O(log n)
            ExtractMin(A2)  // O(log n)
        else if v1 < v2
            ExtractMin(A1)  // O(log n)
        else
            ExtractMin(A2)  // O(log n)

    A.heapsize = i
    return A
```

Correttezza

Invariante: A[1..i] contiene gli elementi comuni estratti finora in ordine crescente.

- **Inizializzazione**: A è vuoto, invariante vera
- **Mantenimento**:
  - Se v1 = v2, aggiungiamo l'elemento comune ed estraiamo da entrambi
  - Se v1 < v2, v1 non può essere nell'intersezione (è minore del minimo di A2)
  - Se v1 > v2, v2 non può essere nell'intersezione (è minore del minimo di A1)
- **Conclusione**: Quando uno dei due heap è vuoto, abbiamo estratto tutti gli elementi comuni

Complessità

O(n log n)

- Numero massimo di estrazioni: 2n
- Ogni ExtractMin costa O(log n)
- Totale: O(2n · log n) = O(n log n)

---

4. ARRAY MAX-HEAP → ALBERO BINARIO LINKED

Problema: Trasformare un array max-heap in un albero binario con rappresentazione linked (puntatori).

Struttura nodo:

```
struct Node {
    int k;        // chiave
    Node* p;      // padre
    Node* l;      // figlio sinistro
    Node* r;      // figlio destro
}
```

Algoritmo

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

toTreeRec(A, i, parent)
    if i > A.heapsize
        return nil

    // Crea nodo corrente
    x = new Node
    x.k = A[i]
    x.p = parent

    // Ricorri sui figli
    x.l = toTreeRec(A, 2*i, x)      // figlio sinistro
    x.r = toTreeRec(A, 2*i+1, x)    // figlio destro

    return x
```

Correttezza

- **Caso base**: Se i > A.heapsize, non c'è nodo (return nil)
- **Caso ricorsivo**:
  - Crea nodo con chiave A[i]
  - Imposta parent correttamente
  - Ricorre sui figli sinistro (2i) e destro (2i+1)
  - La struttura riproduce esattamente l'heap array

Complessità

Θ(n)

- Visita ogni nodo esattamente una volta
- Operazioni per nodo: O(1)
- Totale: Θ(n)

---

5. INSERIMENTO SEQUENZIALE IN MAX-HEAP - ESEMPIO

Problema: Mostrare la sequenza di max-heap ottenuti inserendo sequenzialmente gli elementi 4, 5, 2, 1, 6, 3 in un max-heap inizialmente vuoto.

Sequenza di inserimenti

Passo 1: Insert(4)

```
Heap: [4]
Array: [4]
```

Passo 2: Insert(5)

```
Prima: [4, 5]
Dopo MaxHeapifyUp: [5, 4]
```

5 > 4, quindi swap con padre.

Passo 3: Insert(2)

```
Heap: [5, 4, 2]
```

2 < 5, nessun swap necessario.

Passo 4: Insert(1)

```
Heap: [5, 4, 2, 1]
```

1 < 4, nessun swap necessario.

Passo 5: Insert(6)

```
Prima: [5, 4, 2, 1, 6]
Swap con padre: [5, 6, 2, 1, 4]
Swap con padre: [6, 5, 2, 1, 4]
```

6 > 4 → swap; 6 > 5 → swap; 6 è radice.

Passo 6: Insert(3)

```
Prima: [6, 5, 2, 1, 4, 3]
Swap con padre: [6, 5, 3, 1, 4, 2]
```

3 > 2 → swap con padre; 3 < 6 → stop.

Heap finale: [6, 5, 3, 1, 4, 2]

Confronto con MIN-HEAP

Inserendo la stessa sequenza 4, 5, 2, 1, 6, 3 in un **min-heap**:

Sequenza min-heap:

1. Insert(4): [4]
2. Insert(5): [4, 5]
3. Insert(2): [2, 5, 4] (2 sale alla radice)
4. Insert(1): [1, 2, 4, 5] (1 sale alla radice)
5. Insert(6): [1, 2, 4, 5, 6]
6. Insert(3): [1, 2, 3, 5, 6, 4]

Min-heap finale: [1, 2, 3, 5, 6, 4]

Osservazioni

- Ogni inserimento costa **O(log n)** (altezza heap)
- n inserimenti costano **O(n log n)** totale
- BuildMaxHeap su array esistente costa **Θ(n)** ed è più efficiente

---

RIEPILOGO COMPLESSITÀ

Operazione	Complessità
sndmin(A)	Θ(1)
IsMaxHeap(A) - ricorsivo	O(n)
IsMaxHeap(A) - iterativo	Θ(n)
intersect(A1, A2, n)	O(n log n)
toTree(A)	Θ(n)
Insert (singolo)	O(log n)

Operazione	Complessità
n Insert sequenziali	O(n log n)