4 Heap

Domanda 22 Scrivere una funzione *sndmin*(A) che dato in input un array A organizzato a min-heap, restituisce il successore della radice, ovvero il minimo elemento dello heap maggiore della radice. Se un tale elemento non esiste genera un errore. Assumere che A sia non vuoto e gli elementi in A siano tutti distinti.

Soluzione: È sufficiente ricordare che ogni nodo di un min-heap è minore o uguale ai discendenti per dedurre che il successore della radice, se esiste, è uno dei due figli $A[2]$ o $A[3]$.

```
sndmin(A)
  if A.heapsize > 2
       return min (A[2], A[3])
     else if A.heapsize = 2
       return A[2]
   else
     return error
```

Domanda 23 Scrivere una funzione `IsMaxHeap(A)` che dato in input un array di interi `A[1..n]` che verifica se `A` è organizzato a max-heap e ritorna un corrispondente valore booleano. Valutarne la complessità.

Soluzione: Versione ricorsiva

```
IsMaxHeap(A,i,j) // verifica se l'array A[i,j] e' un max-heap
                    (o meglio, se contiene un max-heap radicato in i,
                     dato che al passo ricorsivo non tutto [i,j]
                     conterra' il sottoalbero scelto
  l = 2*i
  r = 2*i+1
  if l<j
    leftOk = A[i] >= A[l] and IsMaxHeap(l, j)
  else
    leftOk = true

  if r<j
    rightOk = A[i] >= A[r] and IsMaxHeap(r, j)
  else
    rightOk = true

  return leftOk and rightOk
```

Per quanto riguarda la complessità si osservi che $T(n) = c+2T(n/2)$ per un'opportuna costante c e quindi, utilizzando il master theorem, si deduce che la complessità è $O(n)$.

Versione iterativa

```
IsMaxHeap(A,n) // verifica se l'array A[1..n] e' un max-heap
  i=2
  while (i <= n) and (A[i] <= A[i/2])
      i++
  return (i==n+1)
```

Domanda 24 Fornire lo pseudocodice della procedura `HeapExtractMin(A)` per estrarre il minimo elemento da un min-heap A, realizzato tramite un array come visto a lezione (la dimensione corrente dello heap è data dall'attributo $A.heapsize$). Discuterne la complessità.

Soluzione:

```
Heap-Extract-Min(A)    // A e' un min-heap
     if A.heapsize < 1
        error \underflow"
     else
        min = A[1]
        A[1] = A[A.heapsize]
        A.heapsize = A.heapsize - 1
        Min-Heapify(A,1)
        return min

MinHeapify(A,i)
  l = 2i, r =2i+1
  min = i
  if l <= A.heapsize and A[l] < A[m]
     min = l
  if r <= A.heapsize and A[r] < A[m]
     min = r
  if min <> i
     A[i] <-> A[m]
     MinHeapify(A,m)
```

Domanda 25 Dare la definizione di max-heap. Dato un array `A[1..12]` con sequenza di elementi `[60, 6, 45, 95, 30, 24, 15, 80, 19, 38, 21, 70]` si indichi il risultato della procedura `BuildMaxHeap` applicata ad `A`. Si descriva sinteticamente come si procede per arrivare al risultato.

Soluzione: L'array risultante (si vedano le procedure sul libro) è:

`[95, 80, 70, 60, 38, 45, 15, 6, 19, 30, 21, 24]`

5 Alberi e ricorsione

Domanda 26 Dato un albero nel quale i nodi contengono una chiave, si definisca *costo* di un cammino dalla radice ad una foglia, come la somma delle chiavi dei nodi che compaiono nel cammino. Scrivere una funzione `MaxPath(T)` che opera nel modo seguente. Prende in input un albero binario T, con radice $T.root$, e nodi x che hanno come campi $x.k$, $x.l$ e $x.r$, ovvero una chiave, il puntatore al figlio sinistro e destro, rispettivamente. Resituisce la il costo del cammino di costo massimo dalla radice ad una foglia. Valutarne la complessità.

Soluzione:

```
MaxPath(x)
   if x=nil
       return 0
   elseif x.l = nil
       return x.key + MaxPath(x.r)
   elseif x.r = nil
       return x.key + MaxPath(x.l)
   else
       return x.key + max { MaxPath(x.l), MaxPath(x.r) }
```

Domanda 27 Realizzare una procedura `Level(T)` che dato un albero binario `T`, con radice `T.root`, e nodi `x` con campi `x.left`, `x.right` e `x.key`, rispettivamente figlio destro, figlio sinistro e chiave intera, ritorna il numero di nodi per i quali la chiave `x.key` è minore o uguale al livello del nodo (la radice ha livello 0, i suoi figli livello 1 e così via). Valutare la complessità.

Soluzione:

```
// ritorna il numero di nodi y del sottoalbero radicato in x, tali che
//      y.key <= livello

Level(x, level)
   if x == nil
      return 0
   else
      left = Level(x.left, level+1)
      right = Level(x.right, level+1)
      if x.key <= level
               return left + right + 1
      else
         return left + right



// chiamata di base
Level(T)
        return Level (T.root, 0)
```

Si tratta di una visita dell'albero, quindi la complessità è lineare $\Theta(n)$.

Domanda 28 Sia T un albero binario i cui nodi x hanno i campi *x.left*, *x.right*, *x.key*. L'albero si dice un *sum-heap* se per ogni nodo x, la chiave di x è maggiore o uguale sia alla somma delle chiavi nel sottoalbero sinistro che alla somma delle chiavi nel sottoalbero destro.

Scrivere una funzione `IsSumHeap(T)` che dato in input un albero T verifica se `T` è un sum-heap e ritorna un corrispondente valore booleano. Valutarne la complessità.

Soluzione: La soluzione può essere

```
IsSumHeap(T)
  return IsSumHeap-rec(T.root)

IsSumHeap-rec(x) // verifica se l'albero radicato in x e' un sum-heap
                    e ritorna true/false e la somma delle chiavi nel
                    sottoalbero radicato in x
  if x = nil
     return true, 0
  else
     isSumHeapL, sumL = IsSumHeap-rec(x.left)
     isSumHeapR, sumR = IsSumHeap-rec(x.right)
     return (x.key >= sumL) and (x.key >= sumR),
            x.key + sumL + sumR
```

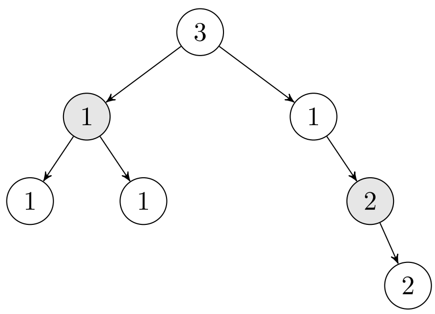
Per quanto riguarda la complessità, se l'albero è bilanciato, $T(n) = c + 2T(n/2)$ per un'opportuna costante c e quindi, utilizzando il master theorem, si deduce che la complessità è $\Theta(n)$. Se l'albero non è bilanciato, si può osservare che si tratta di una visita (costo lineare) o più precisamente scrivere la ricorrenza

$$T(n) = T(k) + T(n - k - 1) + c$$

e provare, per sostituzione, che $T(n) = an + b$ è soluzione per opportune costanti a, b.

Esercizio 10 Un nodo x di un albero binario T si dice *fair* se la somma delle chiavi nel cammino che conduce dalla radice dell'albero al nodo x (escluso) coincide con la somma delle chiavi nel sottoalbero di radice x (con x incluso). Realizzare un algoritmo ricorsivo *printFair(T)* che dato un albero T stampa tutti i suoi nodi fair. Supporre che ogni nodo abbia i campi *x.left*, *x.right*, *x.p*, *x.key*. Valutare la complessità dell'algoritmo.

Un esempio: i nodi grigi sono fair



Soluzione: L'algoritmo può essere il seguente:

```
printFair(x,path)       // x = node of the tree
                        // path=sum of the keys in the path from the root to x
```

```
                         // Action: print the fair nodes and
                        //         returns the sum of the keys in the subtree
    if (x == nil)
       return 0

    left   = printFair(x.l, path + x.key)
    right  = printFair(x.r, path + x.key)
    sumTree = left + right + x.key
    if (path == sumTree)
       print x
    return sumTree
```

e viene chiamato come `printFair(T.root, 0)`.

Si tratta di una visita, quindi con costo $O(n)$ (più precisamente ottenibile con il master theorem come soluzione della ricorrenza $T(n) = 2T(n/2) + c$.

Esercizio 11 Sia dato un albero i cui nodi contengono una chiave intera $x.key$, oltre ai campi $x.l$, $x.r$ e $x.p$ che rappresentano rispettivamente il figlio sinistro, il figlio destro e il padre. Si definisce *grado di squilibrio* di un nodo il valore assoluto della differenza tra la somma delle chiavi nei nodi foglia del sottoalbero sinistro e la somma delle chiavi dei nodi foglia del sottoalbero destro. Il grado di squilibrio di un albero è il massimo grado di squilibrio dei suoi nodi.

Fornire lo pseudocodice di una funzione `sdegree(T)` che calcola il grado di squilibrio dell'albero T (si possono utilizzare funzioni ricorsive di supporto). Valutare la complessità della funzione.

Soluzione:

```
// computes the sum of the leaf nodes of the subtree and the sdegree
// for the node x (returns two values)

sdegree(x)

  if (x == nil)
     sum = 0
     degree = 0
  elif (x.left == nil) and (x.right = nil)      # leaf
     sum = x.key
     degree = 0
  else
     suml, degreel = sdegree(x.l)
     sumr, degreer = sdegree(x.r)
     sum = suml + sumr
     degree = max { degreel, degreel, abs(suml - sumr) }

  return sum, degree
```

Esercizio 12 Si consideri un albero binario T, i cui nodi x hanno i campi $x.l$, $x.r$, $x.p$ che rappresentano il figlio sinistro, il figlio destro e il padre, rispettivamente. Un *cammino* è una sequenza di nodi $x_0, x_1, \ldots, x_n$ tale che per ogni $i = 1, \ldots, n$ vale $x_{i+1}.p = x_i$. Il cammino è detto *terminabile* se $x_n.l = nil$ oppure $x_n.r = nil$. Diciamo che l'albero è 1-bilanciato se tutti i cammini terminabili dalla radice hanno lunghezze che differiscono al più di 1. Scrivere una funzione `bal1`(T) che dato in input l'albero T verifica se è 1-bilanciato e ritorna un corrispondente valore booleano. Valutarne la complessità.

Soluzione: La soluzione può basarsi su di una funzione ricorsiva che calcola per un nodo x la lunghezza massima e minima dei cammini terminabili da x

```
bal1(T)   // verifica se l'albero radicato in x e' 1-bilanciato,
          // ovvero i cammini dalla radice terminati da nil hanno
          // lunghezze che differiscono al piu di 1

          // si basa su di una funzione ricorsica che calcola la
          // lunghezza minima e massima dei cammini.
 min,max = bal1rec(T.root)

 return max-min <= 1

bal1rec(x)
 if x = nil
    return (0,0)
 else
    (left_min,left_max) = bal1(x.left)
    (right_min,r_right_max) = bal1(x.right)
    return max(left_min, right_min) + 1, max(left_max,right_max) + 1
```

Per quanto riguarda la complessità si osservi che $T(n) = c+T(k)+T(n-k-1)$ per un'opportuna costante c e quindi, la complessità è $O(n)$.

Esercizio 13 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 `k-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:

```
k-bound(T,k)
  return k-bound-rec(T.root)

k-bound-rec(x,k)   // Verifica se l'albero radicato in x e' k-bounded
                    Ritorna true/false e la somma delle chiavi in un
                    cammino da x alla radice con somma massima
  if x = nil
     return true, 0
  else
     if x.left = nil
        isKBound, max = k-bound-rec(x.right,k)
        max  = max + x.key
     else if x.right = nil
        isKBound, max = k-bound-rec(x.left,k)
        max  = max + x.key
```

```
     else
        isKBoundL, maxL = k-bound-rec(x.left,k)
        isKBoundR, maxR = k-bound-rec(x.right,k)
        max  = max { maxR, maxL } + x.key
        isKBound =  isKBoundL and isKBoundR and max <= k

    return isKBound, max
```

La complessità è $\Theta(n)$.

Domanda 29 Scrivere una funzione *complete*(T) che dato in input un albero binario verifica se è completo (ovvero ogni nodo interno ha due figli e tutte le foglie hanno la stessa distanza dalla radice).

Soluzione:

```
complete(x) // verifica se l'albero radicato in x e' completo: ritorna
            // l'altezza dell'albero in caso positivo e -1 in caso negativo
 if x = nil
    return 0
 else
    countl = complete(x.left)
    countr = complete(x.right)

    if (countr == -1) or (countl == -1) or (countl <> countr)
       return -1
    else
       return countl+1
```

Per quanto riguarda la complessità si osservi che $T(n) = c+T(k)+T(n-k-1)$ per un'opportuna costante c e quindi, la complessità è $O(n)$.

Domanda 30 Scrivere una funzione `diff(T)` che dato in input un albero binario `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: Il programma si basa su una una funzione ricorsiva `diff-rec(x)` che restituisce il la lunghezza del minimo e massimo cammino per il sottoalbero radicato in x

```
diff(T)
   min,max = diffRec(T.root)
   return

diff-rec(x)
   if x = nil
      return 0,0
   else
      minl, maxl = diff-rec(x.left)
      minr, maxr = diff-rec(x.right)
      return min(minl,minr)+1, max(maxl,maxr)+1
```

Per quanto riguarda la complessità si osservi che si tratta di una visita dell'albero, quindi $\Theta(n)$.