Linked Max-Heaps

Realizzare la struttura dati *Max-Heap* mediante strutture linked.

Soluzione. Un generico nodo `x` conterrà i campi:

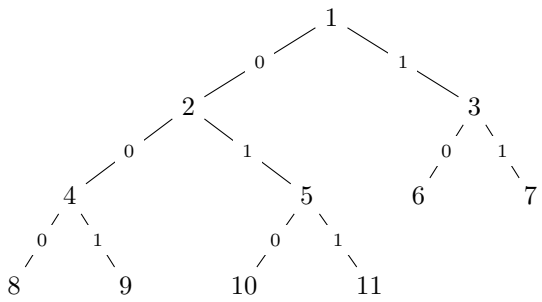
- `x.left`: figlio sinistro
- `x.right`: figlio destro
- `x.p`: genitore

La radice avrà `x.p = nil`, mentre le foglie `x.left = x.right = nil`. Quindi un max-heap è un albero `H`, con campi

- `H.root`
- `H.size`.

Per quanto riguarda le operazioni:

- `Max(H)`, `MaxHeapify(H,x)` e `MaxHeapifyUp(H,x)` restano sostanzialmente inalterate.
- `Insert(H,x)`
  L'elemento va inserito mantenendo la proprietà di albero quasi completo, quindi a destra dell'ultima foglia, se l'ultimo livello non è completo, o come prima foglia di un nuovo livello, se l'ultimo livello è completo. Occorre dunque capire qual è l'ultima foglia dell'ultimo livello. Per questo è opportuno osservare quanto segue. Dato uno heap (nella figura, il numero indica la posizione del nodo, non la chiave), ed etichettando gli archi che connettono al figlio sinistro e destro con 0 e 1 rispettivamente otteniamo:



  allora, se esprimiamo il numero n associato ad un nodo in binario questo risulta del tipo

$$1\underbrace{xxxxxxxxxxxxxx}_{\text{cammino radice-nodo}}$$

  dove $xxxxxxxxxxxxxx$ è la sequenza di etichette nel cammino dalla radice al nodo.

Dunque la posizione nella quale inserire un nuovo nodo, se n è il numero attuale di nodi, si otterrà, esprimendo in binario $n + 1 = (1xxxxxxxxxxxxxx)_2$ ed usando $xxxxxxxxxxxxxx$ per scendere nell'albero: 0 scendi a sinistra, 1 scendi a destra.

Più precisamente, possiamo realizzare una funzione `numToParent(H,k)` che dato uno heap H e un numero di nodo k, restituisce il padre del nodo di posizione k. Per realizzarla si assume di avere una funzione `getBitVector(k)` che dato un numero n restituisce un bitvector che contiene la rappresentazione binaria di n. Il costo di tale funzione sarà $O(\log k)$.

```
numToParent(H, k)                // assume k ≥ 1
   if n == 1
      return nil
   else
      B = getBitVector(k)

      i = B.length                // trova la cifra piu' significativa a 1
      while (i>1 and B[i-1]==0)
         i--

      x = H.root
      for j=i-1 downto 2
        if B[i] == 0
           x = x.left
        else
           x = x.right

      return x
```

A questo punto è facile realizzare la procedura di inserimento, che crea il nuovo nodo, noto il padre. Il nodo sarà figlio sinistro o figlio destro a seconda che il bit meno significativo della posizione sia 0 o 1, rispettivamente.

```
insert(H, x)
   x.left = x.right = nil          // il nodo sara' una foglia

   H.size++                        // incrementa la size
                                   // questa e' la posizione del nuovo nodo

   x.p = numToParent(H, H.size)    // Determina e assegna il parent

   if x.p <> nil
      if (H.size mod 2 == 0)       // se il bit meno significativo e' 0
         x.p.left = x              // il nodo e' figlio sinistro
      else
         x.p.right = p             // altrimenti e' figlio destro

   maxHeapifyUp(x)                 // ripristina la proprieta' di max-heap
```

La complessità è data dalla somma della complessità di `numToParent(H, H.size)` e di `maxHeapifyUp(x)` (il resto ha costo costante). Risulta dunque essere $O(\log n)$.

La funzione `ExtractMax(H)` è analoga alla `Insert`. Il massimo si trova nella radice. Quindi si individua l'ultima foglia e la si sostituisce alla radice.

```
extractMax(H)
   x = numToParent(H, H.size)    // determina il parent dell'ultima foglia

   if x == nil                   // l'albero contiene la sola radice
      H.root = nil
   else

      y.p = nil                  // si prepara a inserire y come root
      y.left = H.root.left
      y.right = H.root.right

      max = H.root               // ricorda il massimo, contenuto nella radice
      H.root = y

      maxHeapify(y)              // ripristina la proprieta' di max-heap

   H.size--                      // in ogni caso si riduce size
```

Anche in questo caso la complessità risulta dunque $O(\log n)$.