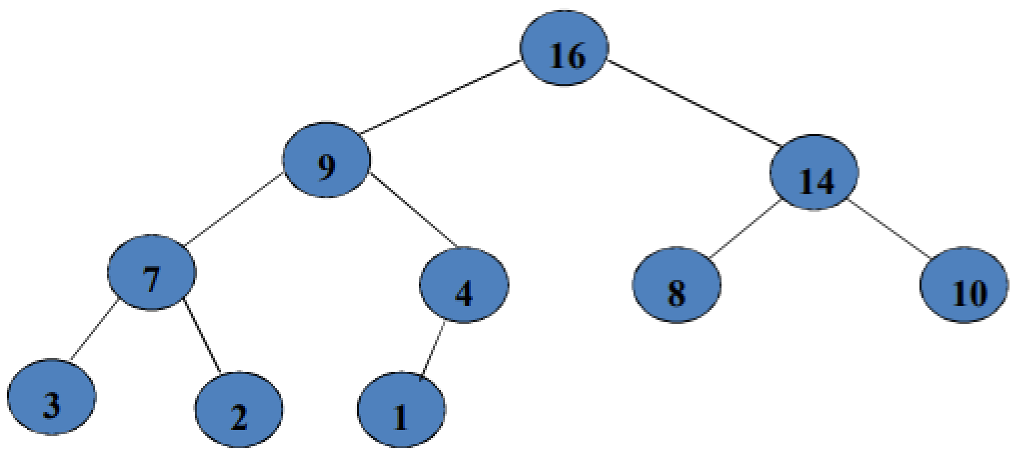
Heap Massimo

E' un albero binario completo tranne per l'ultimo livello, dove è riempito solo a sinistra
Inoltre la **chiave del padre** è $\geq$ delle **chiavi dei figli**(sia sinistri che destri)

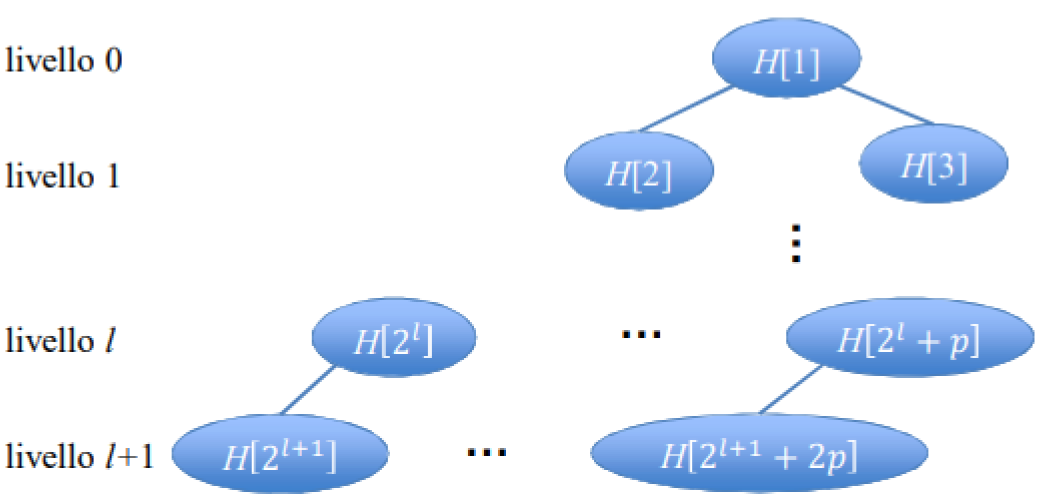


Lo possiamo rappresentare anche come un array dove *H*[1] è la radice dell'albero(con elem 16)

$H =$	16	9	14	7	4	8	10	3	2	1
	1	2	3	4	5	6	7	8	9	10

Qua l'indice parte da 1, di solito è 0. H è l'heap

Tuttavia per spostarsi facilmente tra l'array, dobbiamo prima cercare la relazione tra **Parent**, **Left** e **Right**.



Se indichiamo con ***l*** un generico livello, il **primo** nodo nel livello ***l*** sarà nella posizione 2^l, dunque $H[2^l]$

Invece per trovare il **figlio**:

- **sinistro**: raddoppiamo l'indice→ $H[2^{l+1}]$
- **destro**: raddoppiamo l'indice e aggiungiamo 1 → $H[2^{l+1}+1]$

Infine per trovare il **parent** basta calcolare la **metà**(arrotondata per difetto) dell'indice del nodo

```
PARENT(H, i)
    ▷ Pre: 1 ≤ i ≤ H.N
    ▷ Post: restituisce la posizione del genitore se esiste, 0 altrimenti
return ⌊i/2⌋

LEFT(H, i)
    ▷ Pre: 1 ≤ i ≤ H.N
    ▷ Post: restituisce la posizione del figlio sinistro se esiste, i altrimenti
if 2i ≤ H.N then
    return 2i
else
    return i
end if
RIGHT(H, i)
    ▷ Pre: 1 ≤ i ≤ H.N
    ▷ Post: restituisce la posizione del figlio destro se esiste, i altrimenti
if 2i + 1 ≤ H.N then
    return 2i + 1
else
    return i
end if
```

Pseudocodice che, dato in input l'heap e l'indice di un nodo, restituisce rispettivamente il parent,il figlio sinistro e il figlio destro

Proprietà

In un **Heap** con n elementi(dunque $H[1..n]$), le foglie occupano il semivettore $H[\lfloor n/2 \rfloor + 1 ... n]$

Se $n = 1$ allora $\lfloor n/2 \rfloor + 1 = 1$ ed $H[1]$ è il solo nodo.

Se $n > 1$, $H[i]$ è una foglia se e solo se $n < 2i$, cioè

$$\begin{aligned} n &< 2i \\ \lfloor n/2 \rfloor &< \lfloor i \rfloor = i \\ \lfloor n/2 \rfloor + 1 &\leq i \end{aligned}$$

e $i \leq n$.

Dimostrazione della proprietà

Inserimento

Si aggiunge l'elemento come foglia nell'heap. Lo si fa poi risalire lungo il ramo in cui è stato aggiunto finchè non viene ricostruito l'heap(ovvero finchè non ritorna a valere la **proprietà delle chiavi**)

HeapInsert(H,x) aggiunge x come foglia in H; quindi la fa risalire lungo il ramo cui è stato aggiunto sinché non sia ricostruito lo heap.

```
HEAPINSERT(H, x)
     ▷ Pre: H è un heap
     ▷ Post: H è un heap con x inserito
H.N ← H.N + 1
p ← H.N
H[p] ← x
while p > 1 ∧ H[p] > H[PARENT(H, p)] do
   scambia H[p] e H[PARENT(H, p)]
   p ← PARENT(H, p)
end while
```

$\boxed{\text{HeapInsert è } O(\log n)}$

Pseudocodice dell'insert. Ha una complessità logaritmica

Heap Minimo

E' come l'**Heap Massimo** ma la relazione delle chiavi è inversa:
la **chiave del padre** è $\leq$ **chiave dei figli**(sia sinistri che destri)

Estrazione(Heap massimo)

L'estrazione toglie l'elemento dalla radice.

1. Prima si rimpiazza la radice con la foglia più a destra
2. Viene fatta scendere la radice lungo l'albero finchè non sia maggiori di entrambi i figli(ovvero finchè non ritorna ad essere un heap).
   Nel discendere si sceglie sempre il figlio con il valore maggiore della chiave.
   Questa fase serve per rendere **heap massimo** l'albero.

Heapify: rendere heap(massimo) un albero

E' un algoritmo che, dato un albero, lo rende un heap

$\text{Heapify}(H, i)$
 ▷ Pre: $1 \leq i \leq H.N$, i sottoalberi con radice in $\text{Left}(H, i)$ e $\text{Right}(H, i)$ sono heap
 ▷ Post: l'albero con radice in i è heap
$m \leftarrow$ index of $\text{Max}\{H[i], H[\text{Left}(H, i)], H[\text{Right}(H, i)]\}$
if $m \neq i$ **then**
 scambia $H[m]$ e $H[i]$
 $\text{Heapify}(H, m)$
end if

E' analogo al procedimento descritto nella seconda fase dell'estrazione dell'elemento della radice.
Questo algoritmo ha complessità logaritmica

Avendo definito **Heapify**, l'algoritmo di estrazione diventa:

$\text{HeapExtract}(H)$
 ▷ Pre: H è un heap
 ▷ Post: H è un heap con etichetta massimo eliminata
$H[1] \leftarrow H[H.N]$
$H.N \leftarrow H.N - 1$
$\text{Heapify}(H, 1)$

Non introducendo alcuna complessità aggiuntiva oltre a quella dell'Heapify, la complessità rimane logaritmica

Priority Queue

E' una struttura dati astratta(**ADT**) dove, usando un heap, ogni elemento viene associato ad una priorità(in questo caso la **chiave**).

```
datatype PriorityQueue, Element;
constructors:
   EmptyQueue: PriorityQueue
   Insert: PriorityQueue, Element -> PriorityQueue
   ExtractMaximum: PriorityQueue -> PriorityQueue
observations:
   Maximum: PriorityQueue -> Element
semantics:
   Insert(S,x) = S ∪ {x}
   Maximum(S) = x tale che Priorità(x) =
                   max{Priorità(y)| y ∈ S}
   ExtractMaximum(S) = S \ {Maximum(S)}
```

Riassunto della Priority Queue. Abbasta autoesplicativa

Le funzioni **insert** e **extract_maximum** son le stesse dell'**heap**
Per quanto riguarda la funzione **maximum**, basta ritornare la radice.

Heap-Sort

Possiamo usare la struttura dati **Heap** per riordinare un vettore di elementi attraverso l'algortimo di sorting **Heap-Sort**.
L'**Heap-Sort** ricorda, per struttura, l'algoritmo **Selection-Sort**.

Prima di tutto, bisogna costruire, a partire da un array qualsiasi, un heap.
Dunque definiamo il seguente metodo che esegue ciò appena detto:

```
BuildHeap(V: array)
for i ← ⌊length(V)/2⌋ downto 1 do
   Heapify(V,i)
```

Una volta reso **Heap**, possiamo effettuare il sorting.

```
HeapSort(V: array)
BuildHeap(V) // riorganizza V in uno heap
for i ← size(V) downto 2 do
   scambia V[1] e V[i]
   HeapSize(V) ← HeapSize(V) - 1
   Heapify(V,1)
```

Ha complessità $O(n \log n)$, dunque è un algoritmo **ottimo**