Heap Sort

L'heap sort è un tipo di sorting che ha lo stesso approccio incrementale del Selection Sort, ma:

- Seleziona gli elementi dal più grande al più piccolo;
- Usa una struttura dati efficiente;

Per poter implementare questo tipo di sort dobbiamo prima di tutto creare una struttura dati con cui possa lavorare, appunto l'heap.

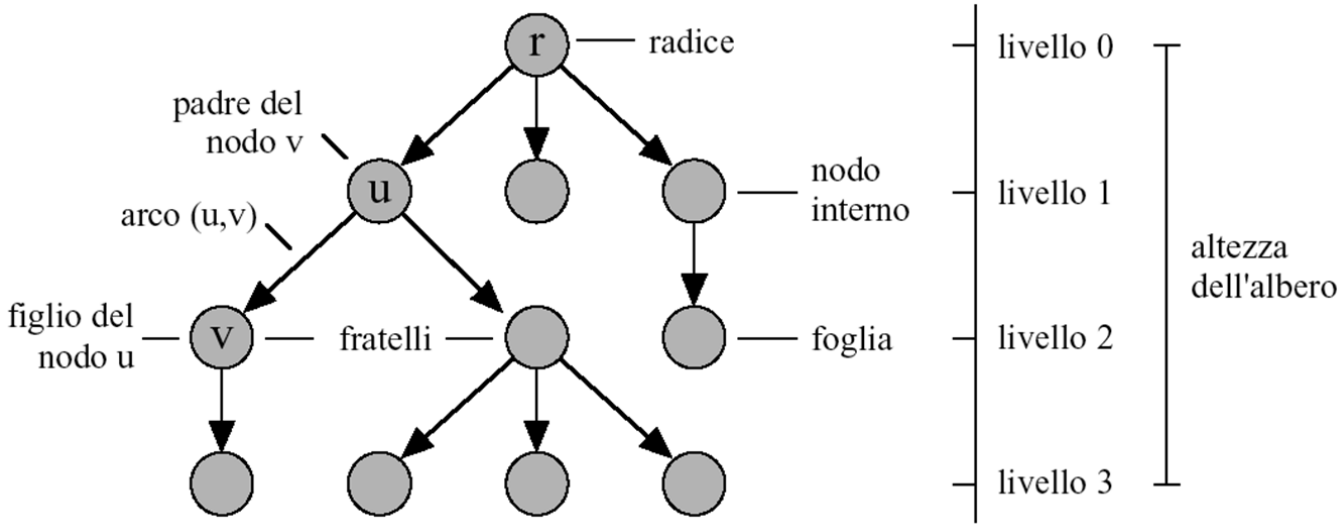
> **Struttura dati**
>
> Organizzazione dei dati che permette di memorizzare la collezione e supportare le operazioni di un tipo di dato usando meno risorse di calcolo possibile.

> **Tipi di dato**
>
> Collezione di oggetti e operazioni di interesse su tale collezione.

Quindi è cruciale per avere un algoritmo efficiente, progettare una struttura H su cui eseguire le operazioni:

- Generazione di H, dato un array A;
- Ricerca del massimo oggetto in H;
- Cancellazione del massimo in H;

Un paio di definizioni sugli alberi



Albero d-ario: albero un cui tutti i nodi interni hanno al più d figli.
$d = 2 \rightarrow$ Albero binario.

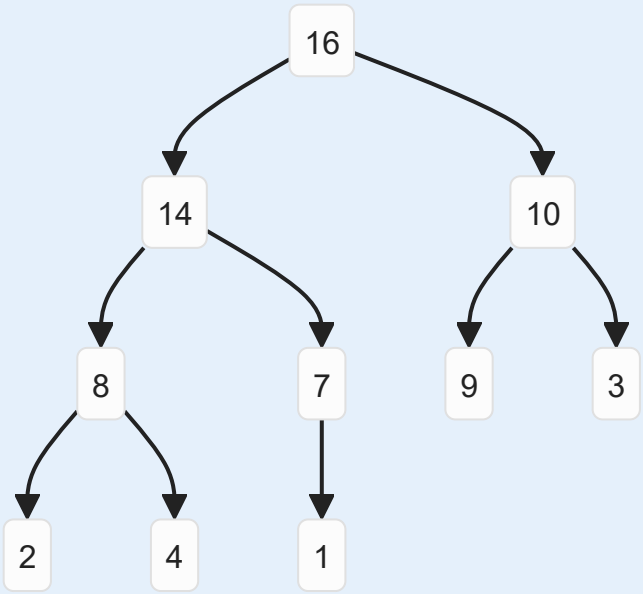
Un albero d-ario è **completo** se tutti i nodi hanno esattamente d figli e le foglie sono tutte allo stesso livello.

Heap

La struttura dati Heap può essere associata ad un insieme S = albero binario radicato con le seguenti proprietà:

- **Completo fino al penultimo livello** e con una struttura rafforzata, cioè le foglie del ultimo livello sono compattate a sinistra.
- Gli elementi di S sono memorizzati nei nodi dell'albero (Ogni nodo v memorizza un solo elemento denotato con $chiave(v)$).
- $chiave(padre(v)) \geq chiave(v) \ \ \forall v$ diverso dalla radice.

Esempio:



L'ordinamento è dato solo verticalmente.

Proprietà degli heap:

1. Il massimo è contenuto nella radice.
2. L'albero con n nodi ha altezza $O(log(n))$.
3. Gli heap con struttura rafforzata possono essere rappresentati con un array di dimensione n.

Dimostrazione della proprietà 2:

Sappiamo che l'albero è alto h e che fino ad $h-1$ abbiamo un albero binario completo. Quindi il numero dei nodi n sarà:

$$n \geq 1 + \sum_{i=0}^{h-1} 2^i = 1 + 2^h - 1 = 2^h \rightarrow h \leq log_2(n)$$

.

Per rappresentare un Heap in un array dobbiamo tenere conto delle seguenti proprietà:

1. $sin(i) = 2i$, cioè il nodo alla sinistra dell'indice i in posizione due volte il valore di i.
2. $des(i) = 2i + 1$, cioè il nodo alla destra di i è in posizione due volte il valore di i +1.
3. $padre(i) = \lfloor i/2 \rfloor$, cioè il padre del nodo i si trova in posizione $i/2$.

⚠ **Heap-size[A] ≠ Length[A]**

Procedura FixHeap

La seguente procedura ci permette di per mantenere la proprietà dell'heap in un array:

```
def FixHeap(i,A): #indice di inizio e array


    #Valori di indici bassati su array che parte da zero
    s = 2*i+1 #Figlio sinistro del nodo
    d = 2*i+2 #Figlio destro del nodo

    if(s < len(A) and A[s] > A[i]): #Se s si trova in A ed è maggiore del
padre
        massimo = s
    else:
        massimo = i

    if(d < len(A) and A[d] > A[i]): #Se d si trova in A ed è maggiore del
padre
        massimo = d

    if(massimo != i):

        A[i],[massimo] = A[massimo],[i] #Scambio il massimo con i
        FixHeap(massimo,A)#Richiamo funzione su istanza minore

    return A
```

Procedura di estrazione del massimo

L'idea della procedura è la seguente:

1. Copiamo nella radice la chiave contenuta nella foglia più a destra dell'ultimo livello.
2. Rimuoviamo la foglia, quindi decrementiamo Heap-size.
3. Ripristinare le proprietà del Heap con fixHeap sulla radice

Tutto questo in un tempo d'esecuzione di $O(log(n))$.

```
def EstraiMax(A):
    max_val = A[0] # Salvo il massimo
    A[0] = A[len(A) - 1] # Sostituisco il massimo con l'ultimo elemento
```

```
    A.pop() # Rimuovo l'ultimo elemento
    FixHeap(0, A) # Richiamo la funzione di correzione
    return max_val
```

Costruzione dell'heap

L'idea per la costruzione dell'heap si basa sul chiamare ricorsivamente la funzione necessaria alla creazione dell'heap sui sottoalberi destri e sinistri della radice.
Questo è un codice in python che la svolge:

```
def heapify(H):
    # Partiamo dall'ultimo nodo non foglia e applichiamo FixHeap
    for i in range(len(H) // 2 - 1, -1, -1):
        FixHeap(i, H)
    return H
```

Facciamo adesso l'analisi della complessità di heapify:

- Sia h l'altezza di un heap contenente n elementi.
- Sia $n' \geq n$ l'intero tale che un heap con n' elementi ha altezza h ed è completo fino all'ultimo nodo.

Vale quindi la seguente disuguaglianza: $T(n) \leq T(n')$ e $n' \leq 2n$
Quindi il tempo d'esecuzione è:

$$T(n') = 2T(\frac{n'-1}{2}) + O(log(n')) \leq 2T(\frac{n'}{2}) + O(log(n'))$$

Dal **Teorema Master** otteniamo:

$$T(n') = O(n')$$

Quindi:

$$T(n) \leq T(n') = O(n') = O(2n) = O(n)$$

La particolarità di questa struttura dati è che ci permette di estrarre il massimo dell'array in poco tempo, ma se volessi invece trovare il minimo?

Si può costruire una struttura min-heap che ha il compito inverso semplicemente invertendo le proprietà delle chiavi, cioè $chiave(padre(v)) \leq chiave(v)$.

HeapSort

L'idea dietro all'heapsort è la seguente:

- Costruire un heap tramite heapify.
- Estrarre ripetutamente il massimo $n-1$ volte, ad ogni estrazione memorizza il massimo nella posizione dell'array che si è appena liberata.

Di seguito lo pseudocodice:

```
heapSort (A)

1. Heapify(A)
2. Heapsize[A]=n
3. for i=n down to 2 do
4. scambia A[1] e A[i]
5. Heapsize[A] = Heapsize[A] -1
6. fixHeap(1,A)
```

Il motivo per cui usiamo un max-heap invece di un min-heap è che con il max-heap possiamo implementare la struttura senza usare memoria costante.

Teorema:

L'algoritmo HeapSort ordina in loco un array di lunghezza n in tempo $O(nlog(n))$ nel caso peggiore.