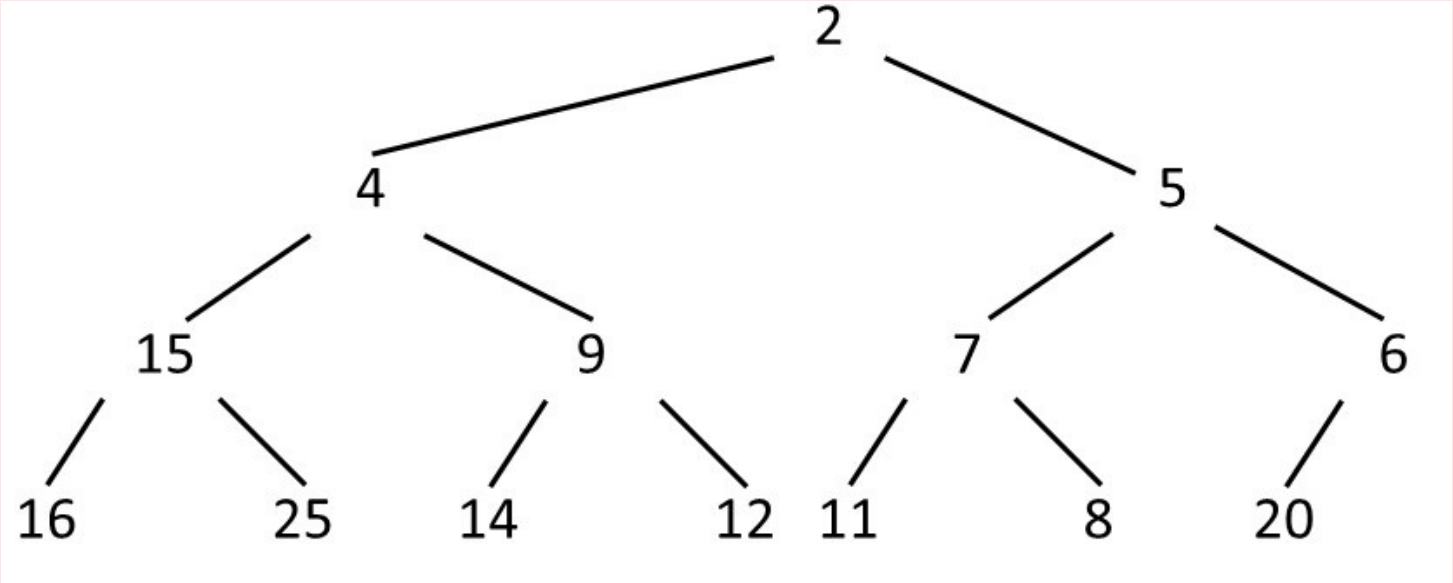
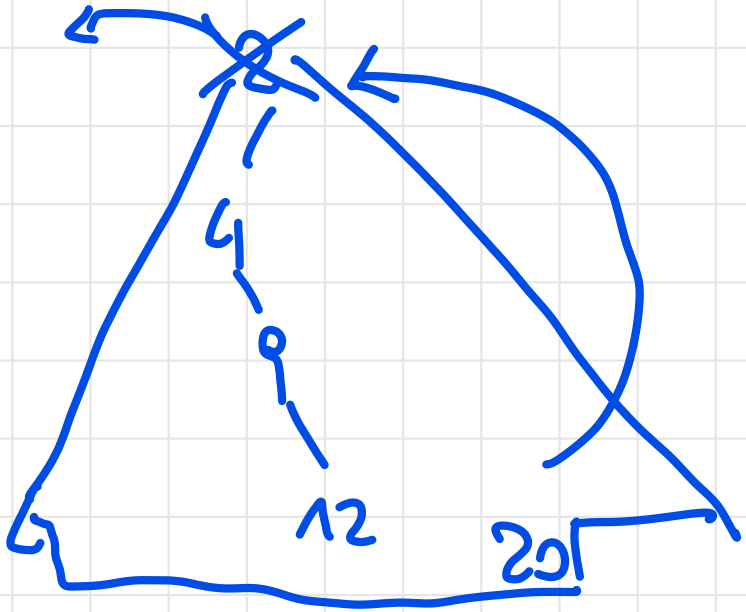
Esercizio

Si consideri il seguente Heap

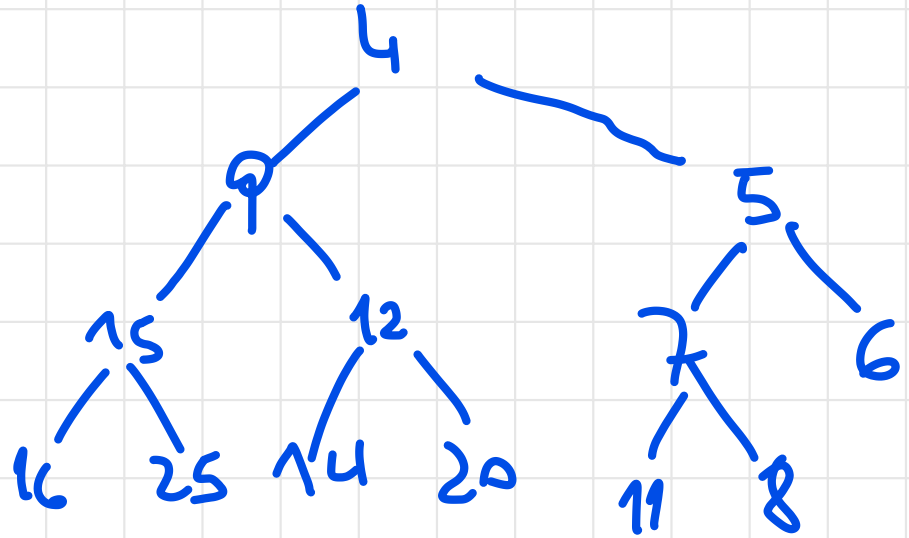


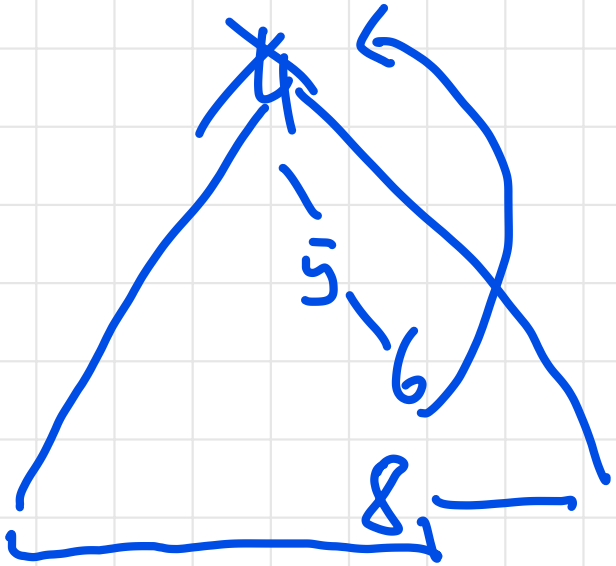
Far vedere lo Heap risultate dopo l'esecuzione delle seguenti operazioni: `removeMin`, `removeMin`, `insert`$(3, *)$, `insert`$(4, *)$.

Array: [2 4 5 15 9 7 6 16 25 14 12 11 8 20]

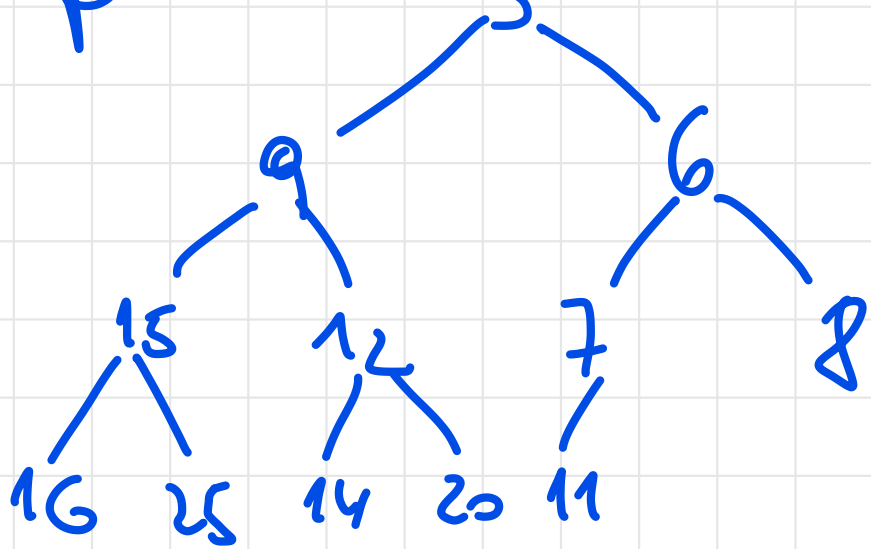


Dfs lewe n.u

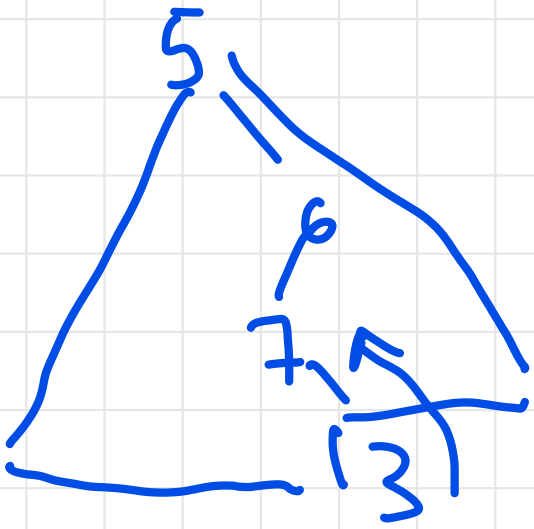




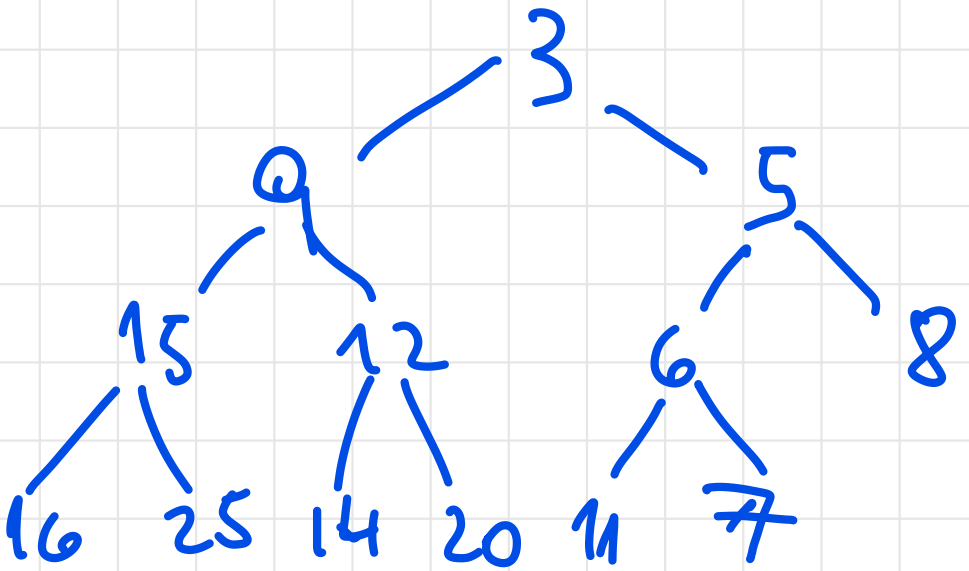
Bfs wuske n.u



Insert(3, *)



Dopo insert(3, *)



Insert(4, *): esercizio

Esercizio

Sia $S = S[0], S[1], \ldots, S[n-1]$ una sequenza di n entry con chiavi intere distinte (ad es., profili IG con i rispettivi numeri di follower), e sia τ un intero in $[1, n]$. Il seguente algoritmo (da completare) trova le τ entry in S con chiave maggiore (*Top-τ entry*).

Algoritmo `Top`(S,τ)
Input: Sequenza S di n entry, intero $\tau \leq n$
Output: τ entry di S con chiave più grande

```
Q ⟵ priority queue vuota;
for i ← 0 to n − 1 do
    if (i < τ) then Q.insert(S[i].getKey(),S[i].getValue());
    else
        { Q.min()
          Q.removeMin()
          Q.insert(...) }
return Q
```

Oss. Pseudocodice basato solo sull'interfaccia PQ.

Esercizio (Continua)

1. Completare l'algoritmo in modo che sia corretto e che Q non contenga mai più di τ entry.
2. Provare la correttezza dell'algoritmo usando un opportuno invariante per il ciclo for.
3. Analizzare la complessità dell'algoritmo.

IDEA: Mantenere in Q le τ entry di $S[0 \div i]$ con chiave più grande (se $i \geq \tau$) altrimenti mantenere tutte, se $i < \tau$

Nell'iterazione i inseriamo $S[i]$ in Q se la chiave più grande del min

- rimuovendo Q.min(), e se
S(i) ha dove numero di Q.min()
non facciamo nulla

Esempio $S = [22\ 3\ 9\ 11\ 6\ 13\ 10\ 18\ 2\ 8]$

$n = 10$ e $\tau = 3$	
i	Q a fine Iterazione i
2	22 3 9
3	22 11 9
4	22 11 9
5	22 11 13
6	22 11 13
7	22 18 13
8	22 18 13
9	22 18 13

① Completamento dello pseudocodice

```
else {
  if (Q.min().getKey() < S[i].getKey()) then
    Q.removeMin()
    Q.insert(S[i].getKey(), S[i].getValue())
  }
}
```

③ Complessità

* Q implementata come lista non ordinata

- fino a τ iterazioni: $\Theta(1)$ op. ciascuna
- successive $n-\tau$ iterazioni: $O(t)$ op. ciascuna

$\Rightarrow$ complessità: $O(n\tau)$

* Q implementata come heap:
  - Ogni iterazione: $O(\log \tau)$ operazioni

  $\Rightarrow$ complessità: $O(n \log \tau)$

* Q implementata come lista ordinata

  $\Rightarrow$ stessa complessità della lista non ordinata dovuta all'init $\Rightarrow O(n\tau)$

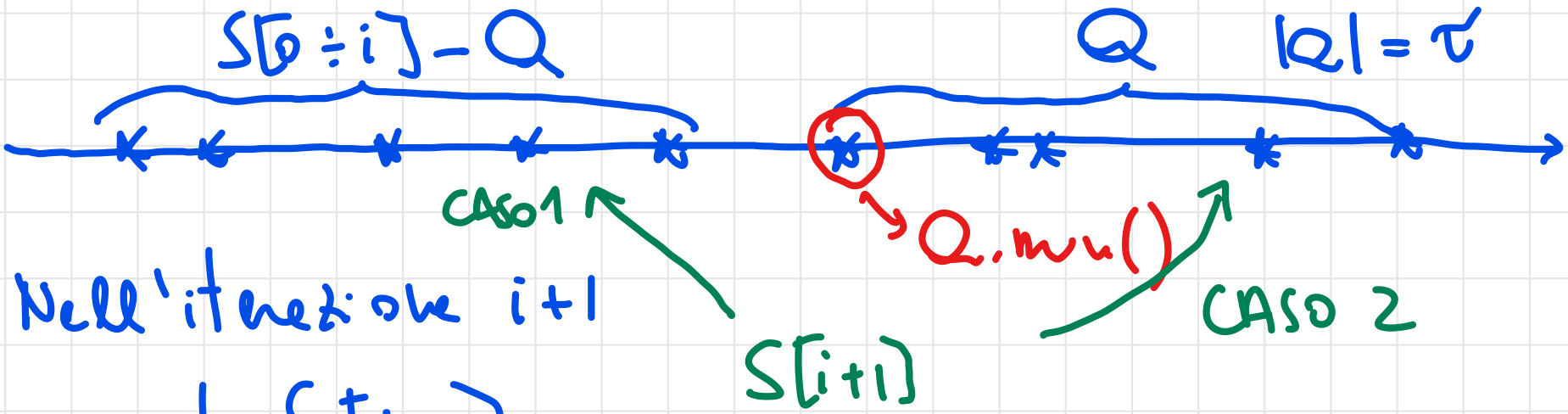
② Correttezza: basata sul seguente invariante che deve valere alla fine dell'iterazione i del for $(i \geq -1)$

Q contiene le $\min\{i+1, \tau\}$ entry di $S[0 \div i]$ con chiave maggiore

Dimostriamo che vale alla fine di ogni iterazione

* $i < \tau$ banalmente vero
* $i \longrightarrow i+1 \quad (i \geq \tau - 1)$

Alla fine dell'iterazione i la situazione è questa:



Nell'iterazione i+1

quando S[i+1]

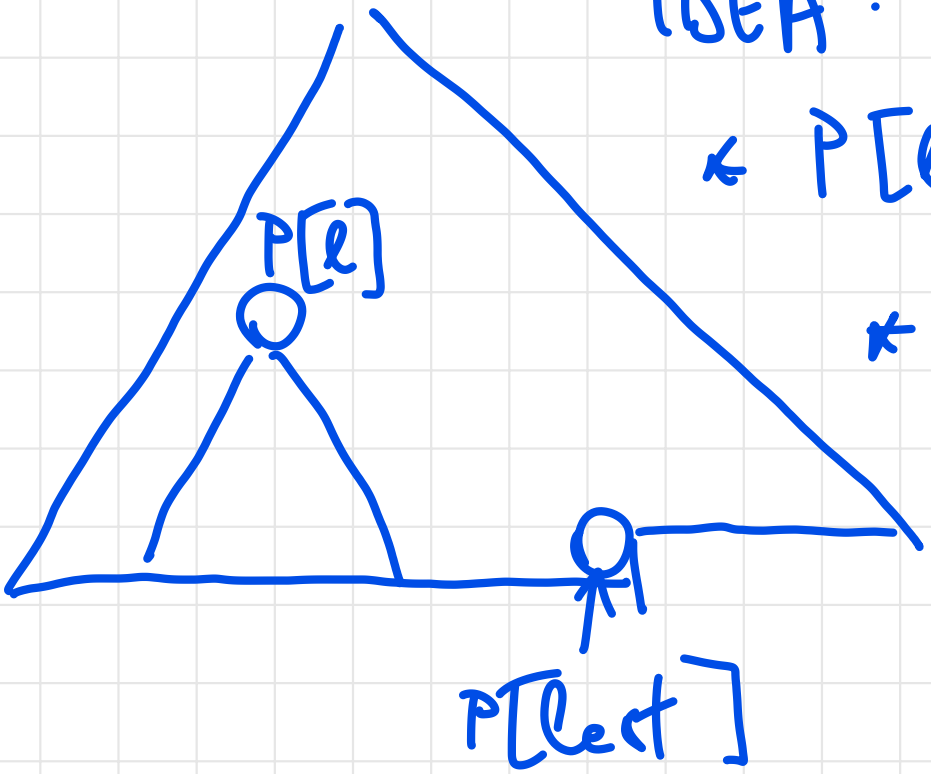
- Nel CASO 1: l'operazione non fa nulla
- Nel CASO 2: l'operazione rimuove Q.mu e inserisce S[i+1] in Q

In entrambi i casi l'invariante continua a valere alla fine dell'iterazione i+1

Alla fine del for, l'invariante assicura che
Q contiene le tuple di S che chiave maggiore

Esercizio

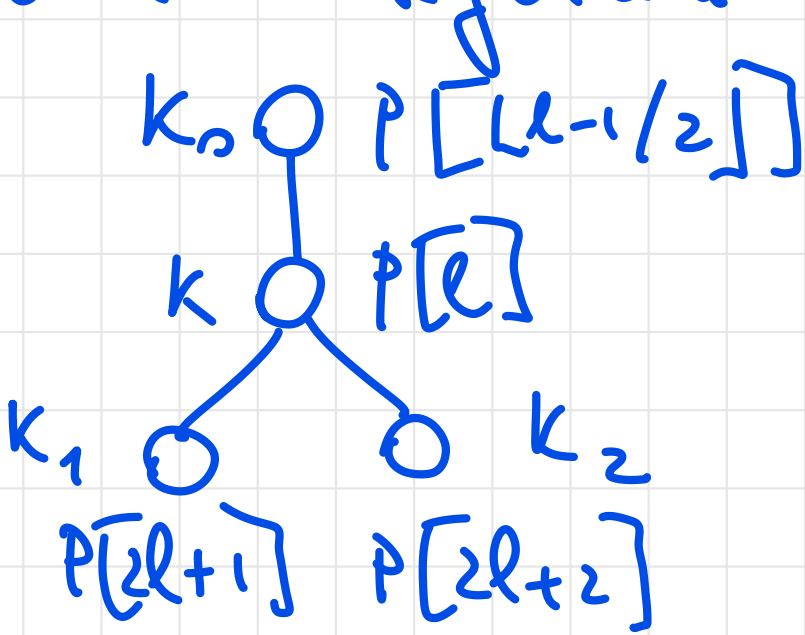
Progettare e analizzare un algoritmo per rimuovere da uno heap P con n entry una entry $P[\ell]$, per un qualche $0 \leq \ell < n$, ripristinando poi le proprietà dello heap.



IDEA:

* P[ℓ] ← P[last--]
* ripristinare le H.O.P. con un up-heap bubbling o un down-heap bubbling

Dopo aver eseguito $P[i] \leftarrow P[heap\text{-}size--]$
la situazione è la seguente



CASO 1: $k_0 \leq k \leq \min\{k_1, k_2\}$

$\Rightarrow$ nessuna violazione della H.O. P

Caso 2: $k < k_0 \leq \min\{k_1, k_2\}$

$\Rightarrow$ up-heap bubbling

Caso 3: $k_0 \leq \min\{k_1, k_2\} < k$

$\Rightarrow$ down-heap bubbling

Oss. Nei casi 2 e 3 la violazione della HOP locata in $P[i]$ e i suoi contenuti (Caso 2) o discendenti (Caso 3)

Algoritmo removeEntry(P,ℓ)
input Heap P con n entry, intero ℓ (0 $\leq \ell \leq$ n-1)
output Heap P con n-1 entry ottenuto rimuovendo P[ℓ]

```
P[ℓ] ⟵ P[last--]; i ⟵ ℓ;

/* Caso 2:  up-heap bubbling a partire da P[j] */

while ((i≥1) and (P[i].getKey() < P[⌊(i-1)/2⌋].getKey())) do {
    swap(P[i],P[⌊(i-1)/2⌋])
    i ⟵ ⌊(i-1)/2⌋
}

/* Caso 3:  down-heap bubbling a partire da P[j] */

j ⟵ indexMinChild(P,i);
while ((j != null) and (P[i].getKey() > P[j].getKey())) do {
    swap(P[i],P[j])
    i ⟵ j
    j ⟵ indexMinChild(P,i);
}
```

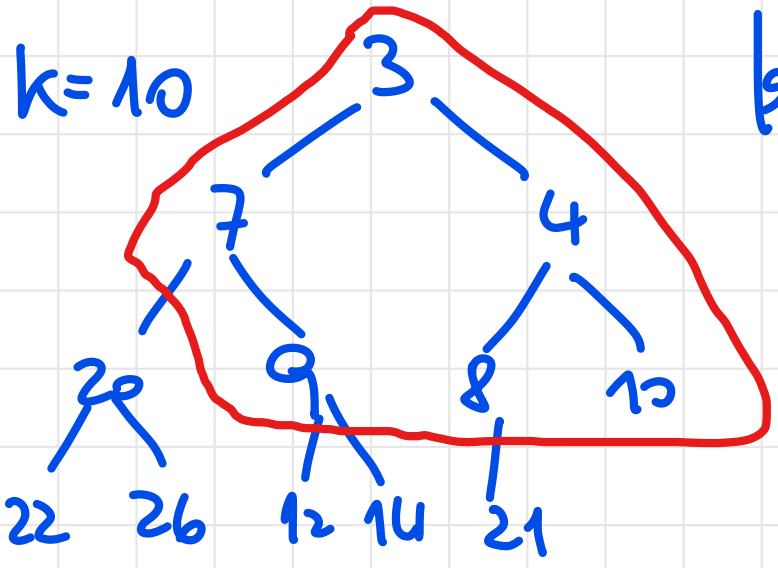
Oss. Nel caso 1 nessuno dei 2 cicli viene eseguito, mentre nei casi 2 e 3 solo il ciclo corrispondente al caso viene eseguito

Complessità: $O(\log n)$ dato che ciascun ciclo richiede $O(\log n)$ operazioni

Esercizio (C-9.34 [GTG14])

Progettare un algoritmo che dato uno heap T con n entry e una chiave k, stampi tutte le entry in T con chiave $\leq k$. La complessità deve essere proporzionale al numero di entry stampate ($+1$ nel caso siano 0).

Esempio



Oss. le entry con chiave $\leq k$ formano un sottoalbero T' (eventualmente vuoto) che se non è vuoto è radicato nella radice di T

$\Rightarrow$ IDEA: visita in preordine di T'

Oss. Estrarre le entry con chiave minore stampabile fino alla prima entry con chiave $> k$ ha un costo $O(m \log n)$ (m = # entry stampate) che in generale è più che lineare in m a differenza di quanto richiesto

```
Algoritmo Print(T, i, k) // stampa elementi
input  } esercizio                    i = 0
output }
if (T[i].getkey() ≤ k) then {
   Stampa(T[i])  // visita
   if ((2i+1 ≤ last) AND (T[2i+1].getKey() ≤ k)
        then Print(T, 2i+1, k)
   if ((2i+2 ≤ last) AND (T[2i+2].getKey ≤ k)
        then Print(T, 2i+2, k)
}
```

Complessità: assumendo che la stampa di una entry sia una operazione elementare la complessità è $\Theta(n+1)$ $n = \#$ entry stampate, dato che l'algoritmo esegue la visita in preordine di T!

Esercizio (C-9.36 [GTG14])

Siano T_1 e T_2 due alberi binari (non necessariamente completi e implementati tramite struttura linkata) i cui nodi memorizzano delle entry in modo da soddisfare la heap-order property. Descrivere un algoritmo per combinare T_1 e T_2 in un unico albero binario T i cui nodi contengano tutte le entry di T_1 e T_2, mantenendo la heap-order property. La complessità dell'algoritmo deve essere $O(h_1 + h_2)$, dove h_1 e h_2 rappresentano le altezze di T_1 e T_2, rispettivamente.

Esercizio

Sia $S = S[0], S[1], \ldots, S[n-1]$ una sequenza contenente n chiavi distinte e sia m un indice intero tale che $1 \leq m \leq n/\log_2 n$. Progettare un algoritmo che in tempo $O(n)$ determini la m-esima chiave più piccola di S. (Analizzare la complessità dell'algoritmo per far vedere che è $O(n)$.)