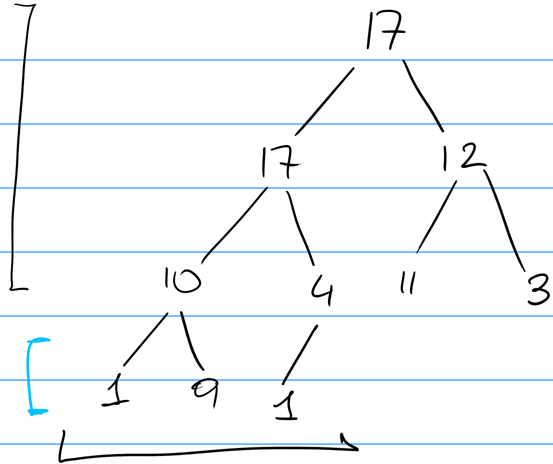
Priority Queue:

	Naïve	Binary Heap
PUSH (x): Aggiungi elemento x	$O(1)$	$O(\log(n))$
GET_MAX (): Ritorna il massimo	$O(n)$	$O(1)$
REMOVE-MAX() : Rimuovi il massimo	$O(n)$	$O(\log(n))$

Binary heap: Un albero binario "quasi-completo" con elementi nei nodi, t.c.: il padre di v è $\geq$ di v.

Completo fino al penultimo livello



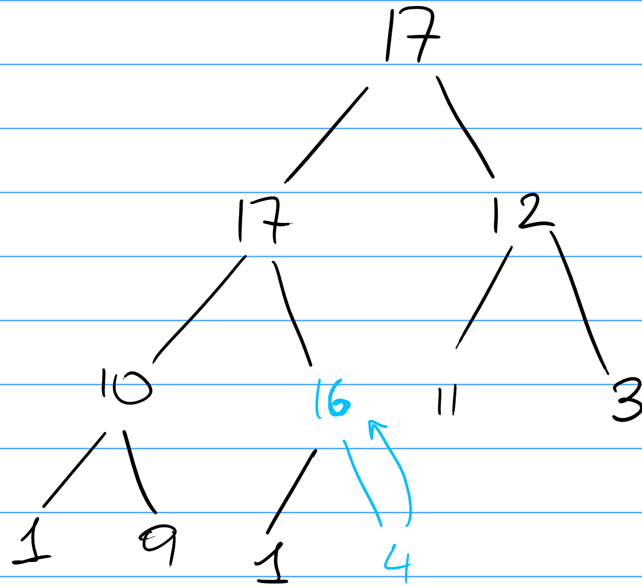
Quasi completo se l'ultimo livello è "un prefisso"

Remark: L'altezza è $\log_2(n)$

GET_MAX() è banale: leggo il numero nella radice.

PUSH(x): Aggiungo x come nuova foglia e lo scambio con il padre finché x è $\geq$ del padre. $\rightsquigarrow$ Alla fine sono un binary heap.

Complessità: $O(\log(n))$.



Why: Remove (12) è difficile in una p-q? Issue: Dove sta il 12?

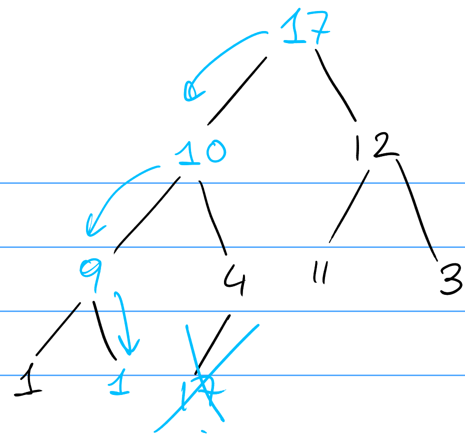
Trick: Se accetto che il tempo diventa ammortizzato $\rightarrow$ posso anche rimuovere.

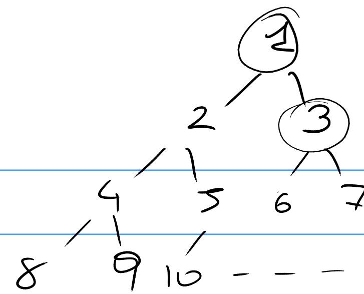
Come: Quando rimuovo, mi segno in un array di booleani che un elemento è uscito.

REMOVE_MAX(): Scambiare la radice con l'ultima foglia, cancellare l'ultima foglia e poi abbassare la radice scambiandola col suo figlio maggiore fin quando non è $\geq$ dei suoi figli.

Alla fine sono un binary heap.

Complessità è $O(\log_2(n))$.





Remark: Le p-q sono nelle stl e sono implementate così.

Remark: Un albero quasi-completo può essere implementato con un vector.

AGGIUNGI / RIMUOVI FOGLIA: push/pop_back.

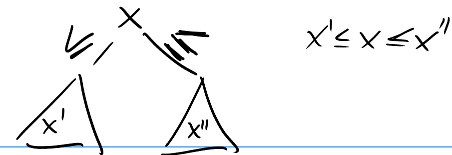
radice: at(1)

Dato v, i suoi figli sono $2v$ e $2v+1$.

il padre di v è $\lfloor \frac{v}{2} \rfloor$.

SET: (più potente delle p-q ma più complicato e più lento x3)

Operazione		NAIVE	BBST
INSERT(x):	Aggiunge l'elemento x.	$O(1)$	
REMOVE(x):	Rimuove -- --.	$O(n)$	$O(\log(n))$.
lower_bound(x):	Restituisce l'elemento minore $\geq x$. ↑ può essere usato per capire se x è nell'insieme	$O(n)$	



BST: E' un albero binario con elementi nei nodi t.c.
lower_bound(x) in O(height).
Balanced = height e' O(log(n)).

Remark: set delle stl e' un bbst.
Remark: Dato x, quanti elementi < x ci sono nel set? – Questo non si puo' fare con set.
– Si puo' fare con ordered_set.

HASH-SET:

	NAIVE	HASH	
INSERT (x): //	O(1)	O(1)	
REMOVE (x): //	O(n)	O(1)	← lento
IS_INSIDE (x): Se un elemento sta nel container	O(n)	O(1)	

Se gli elementi fossero numeri tra [0, L) → un array lungo L di booleani andrebbe bene.
Idea: Ci si riconduce a questo caso.
Ci viene data h: {mondo degli elementi} → N.
E fisso una lunghezza L. e alloco un array (arr) di vector lungo L.
Quando devo gestire x ↦ guardo nel "cassetto" arr[h(x) % L]

INSERT(x): arr[h(x)%L].pb(x).
REMOVE(x): // .remove(x)
IS_INSIDE(x): // .IS_INSIDE(x)

} O(arr[h(x)%L].size()).

Se h è "non regolare" allora $\#\{arr[i].size()\} = O\left(\frac{\text{elementi}}{L}\right)$ — $O(1)$

Come scelgo L? Raddoppio L (e ricostruisco tutto) quando $L = \#$ elementi.

<u>Remark</u>: Sono nelle stl, ma con un'implementazione diversa.

LCA o BINARY-LIFTING

Dato un albero radicato,

LIFT(v, d): Restituisce il d-esimo antenato di v.

Naive: $O(d)$

Binary lifting: in $O(\log(n))$.

<u>OSSERVAZIONI</u>:
- Ogni numero intero n si scrive come somma di $O(\log(n))$ potenze di 2 distinte
- LIFT $(v, x+y)$ = LIFT (LIFT (v, x), y).

Quindi il trucco è precalcolare <u>LIFT $(v, 2^k)$</u> per ogni v e per ogni $0 \leq k < \log_2(n)$.

Così facendo, $LIFT(v, d) = LIFT(v, 2+8+64+\dots)$

$= LIFT(LIFT(LIFT(v,2),8),64)\dots$

$\rightarrow O(\log(n))$

Come precalcolo? $LIFT(v, 2^k) = LIFT(LIFT(v, 2^{k-1}), 2^{k-1})$.

Preprocessing: $O(n \log(n))$.

Query: $O(\log(n))$.

anc[v][k]

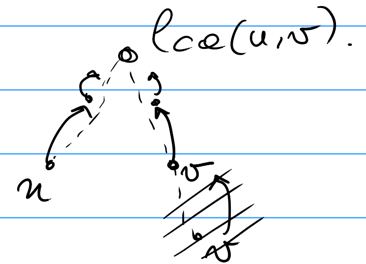
```
int lift(int v, int d) {
    for (int e = log(n)-1; e >= 0; e--) if (d & (1<<e)) v = anc[v][e];
    return v;
}
```

Se l'e-esimo bit è 1 in d.

Funzionerebbe anche $0 \rightarrow \log(n)-1$.

LCA: Dati due vertici u, v, $lca(u,v)$ è il più basso vertice che è un antenato di entrambi.

Trovare $lca(u,v)$ in $O(\log(n))$.



Step 0: Portare u, v alla stessa altezza.

Step 0.5: Se ora $u=v \rightarrow$ fine.

Step 1: Altrimenti sollevate u, v della più grande pot. di 2 che li mantiene distinti

& ripetete.

Step 2: Ora padre(u) = padre(v) che è l'lca che cercavamo.

```
lca (u, v) {
   Assumo dep(u) = dep(v) e u ≠ v.
→  for (int e = log n-1; e >= 0; e--) if (anc[u][e] != anc[v][e]) { u = anc[u][e];
                                                                   v = anc[v][e]; }
   return padre(u);
}
```