COMP 250

Lecture 23

priority queue ADT
heaps 1

Nov. 1/2, 2017

Priority Queue

Like a queue, but now we have a more general definition of which element to remove next, namely the one with highest priority.

e.g. hospital emergency room

Assume a set of comparable elements or “keys”.

Priority Queue ADT

- add(element)

- removeMin()

  "highest" priority = "number 1" priority

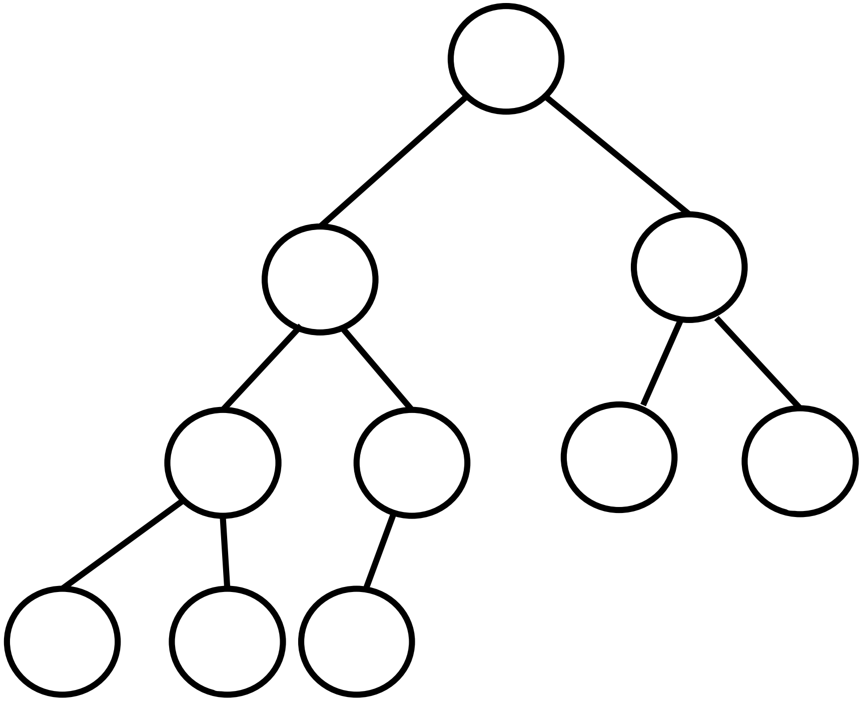
- peek()
- contains(element)
- remove(element)

How to implement a Priority Queue ?

- sorted list ?
- binary search tree (last lecture) ?
- balanced binary search tree (COMP 251) ?
- heap (next 3 lectures)

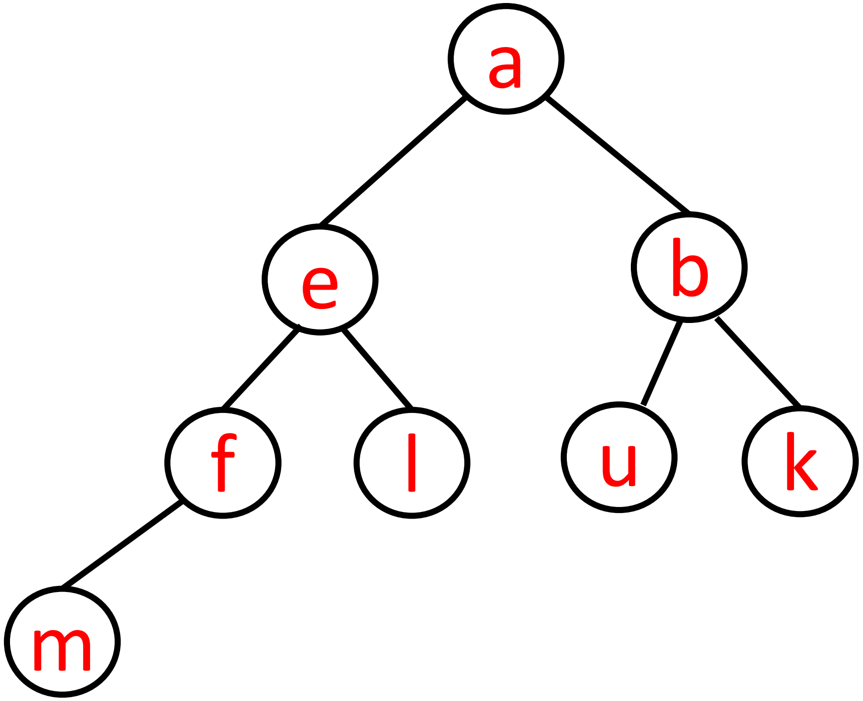
Not the same “heap” you hear about in COMP 206.

Complete Binary Tree (definition)



Binary tree of height h such that every level less than h is full, and all nodes at level h are as far to the left as possible

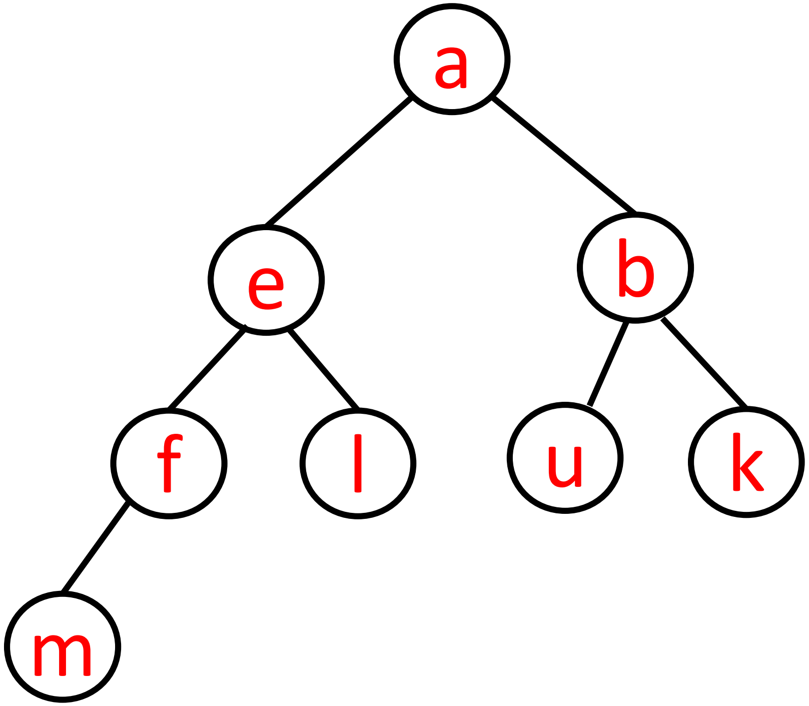
min Heap (definition)



Complete binary tree with unique comparable elements, such that each node's element is less than its children's element.

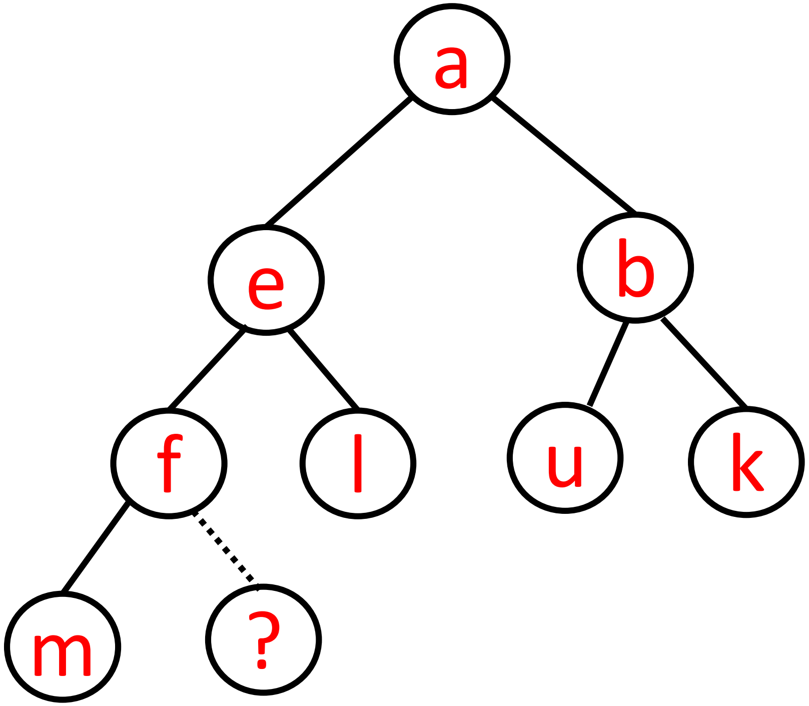
Heap.add(element)

e.g.   add(c)



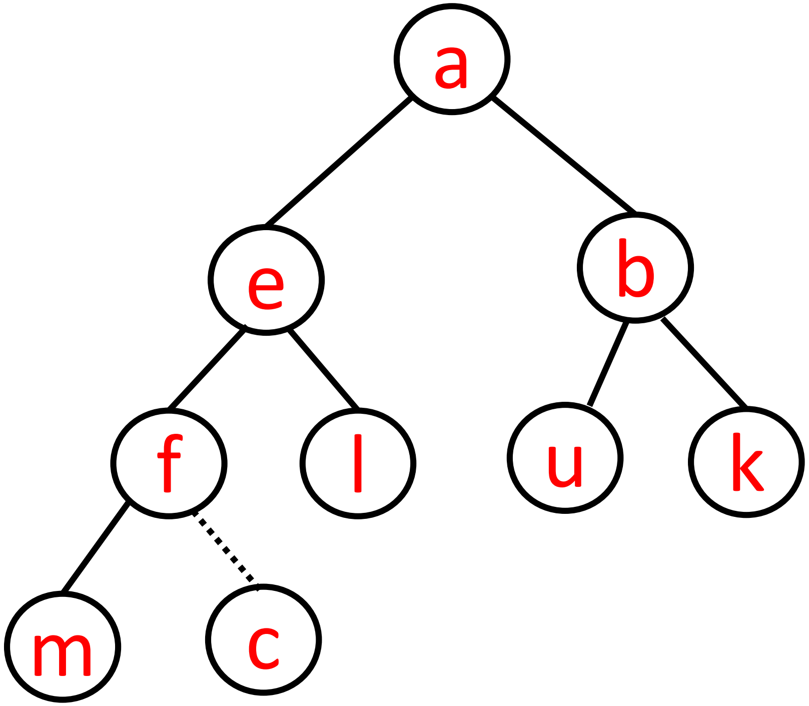
Heap.add(element)

e.g. add(c)



Heap.add(element)

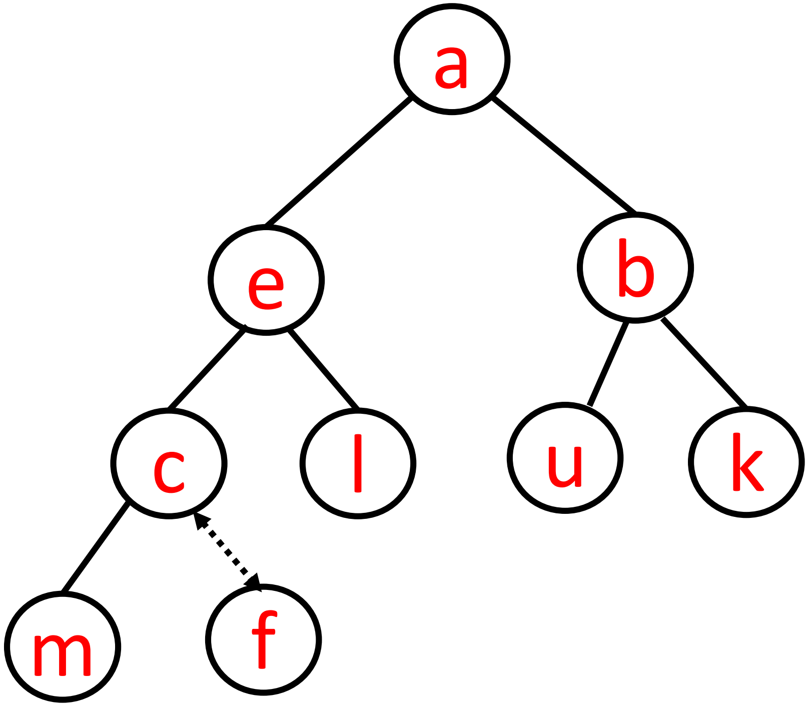
e.g. add(c)



Problem : adding at the next available slot typically will destroy the heap property.

We swap c with its parent f.

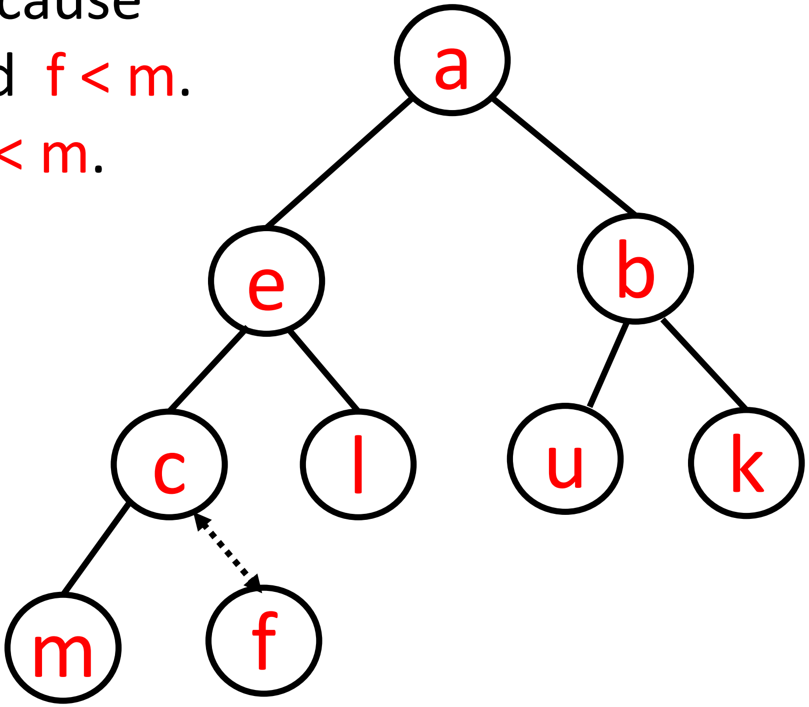
Q: Can this create a problem with c's former sibling, who is now c's child?



We swap c with its parent f.

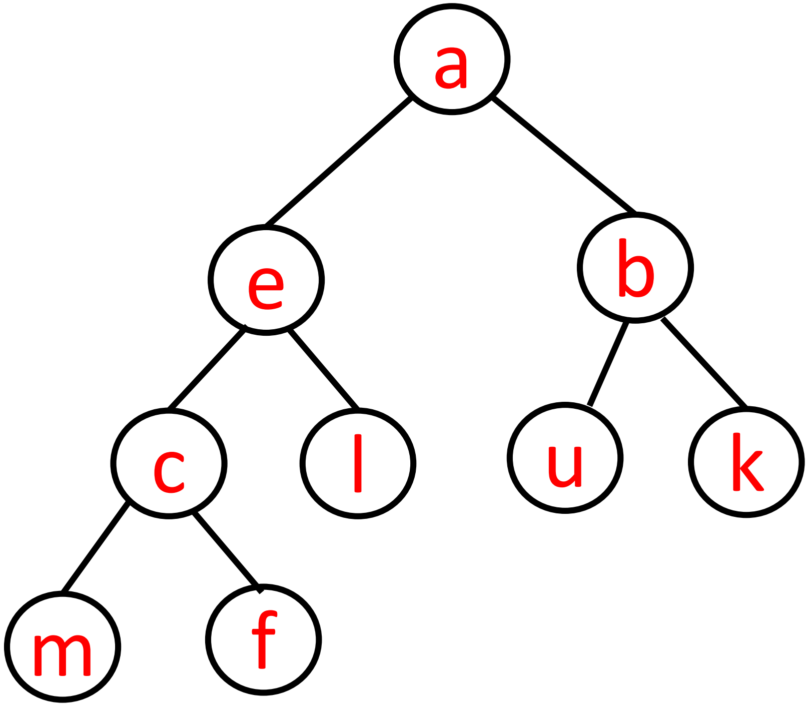
Q: Can this create a problem with c’s former sibling, who is now c’s child?

A: No. Because
$c < f$ and $f < m$.
Thus, $c < m$.



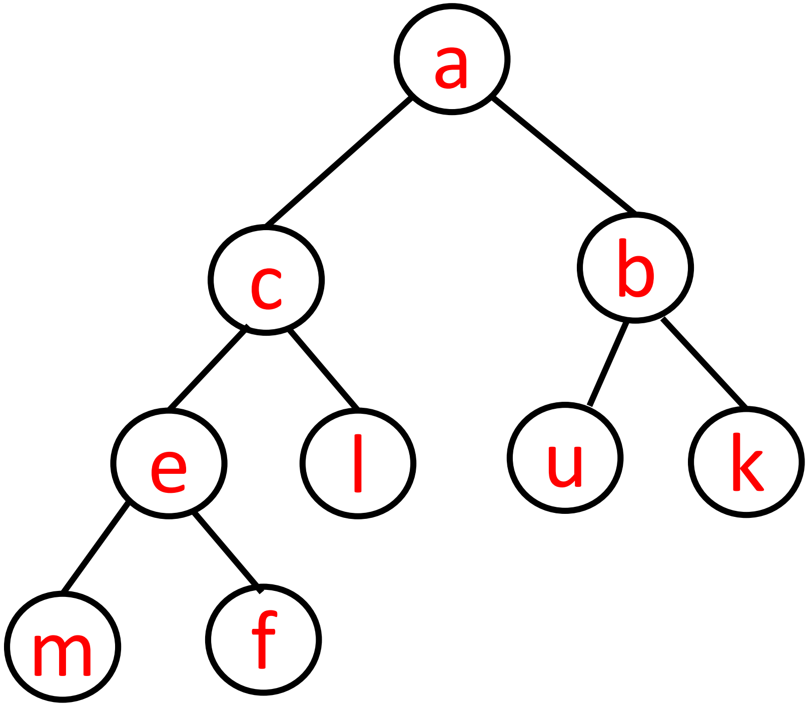
Q: Are we done ?

A: Not necessarily. What about c's parent?



We swap c with its (new) parent e.

Now we are done because c is greater than its parent a



Heap.add(element)

```
add( element ){
    cur = new node at next available leaf position
    cur.element = element

}
```

Heap.add(element)

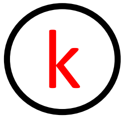
```
add( element ){
    cur = new node at next available leaf position
    cur.element = element
    while (cur != root) and (cur.element < cur.parent.element){



    }
}
```

Heap.add(element)

```
add( element ){
    cur = new node at next available leaf position
    cur.element = element
    while (cur != root) and (cur.element < cur.parent.element){
        swapElement(cur, parent)  //   arguments are nodes
        cur = cur.parent
    }
}
```

How to build a heap?

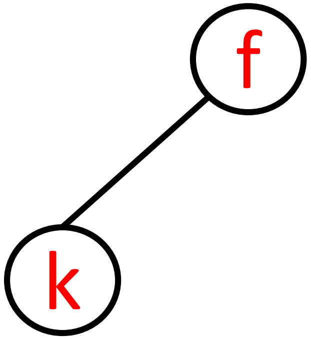
add(k)



add(f)

How to build a heap?

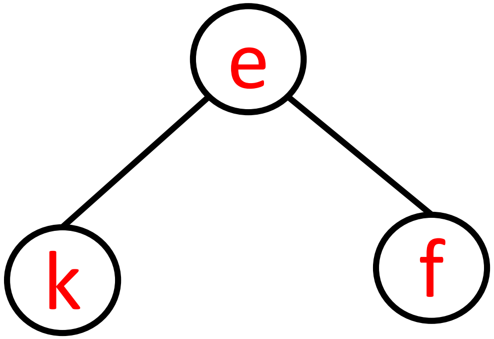
add(k)
add(f)



add(e)

How to build a heap?

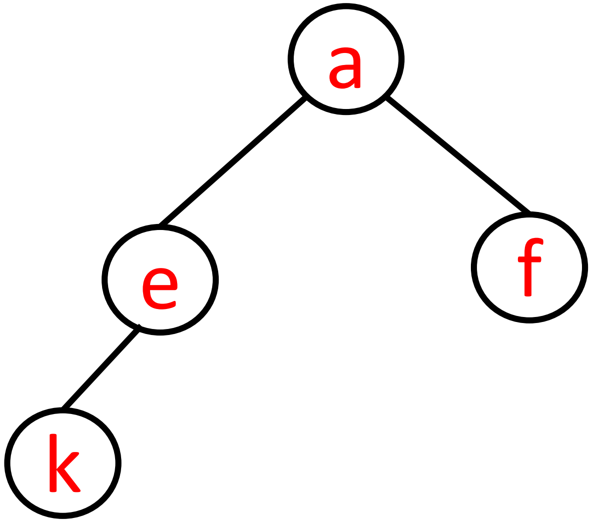
add(k)
add(f)
add(e)



add(a)

How to build a heap?

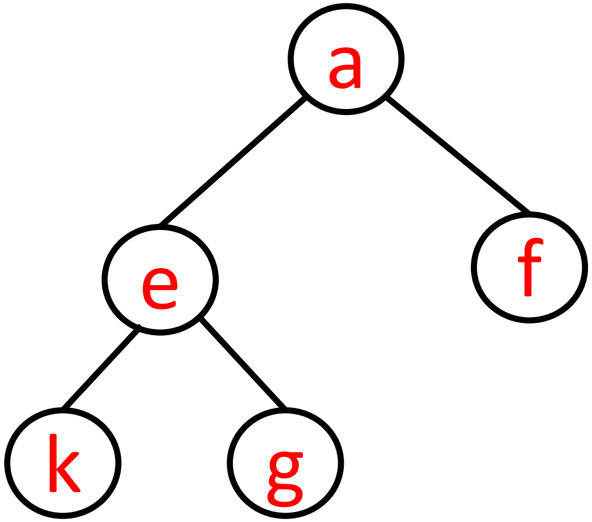
add(k)
add(f)
add(e)
add(a)



add(g)

How to build a heap?

add(k)
add(f)
add(e)
add(a)
add(g)

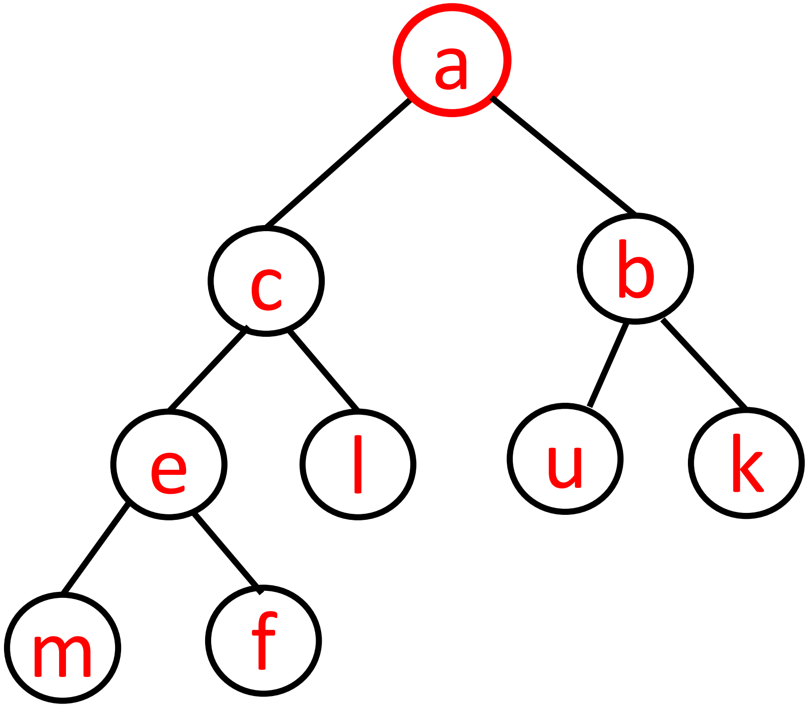


This method of building a heap is slow.

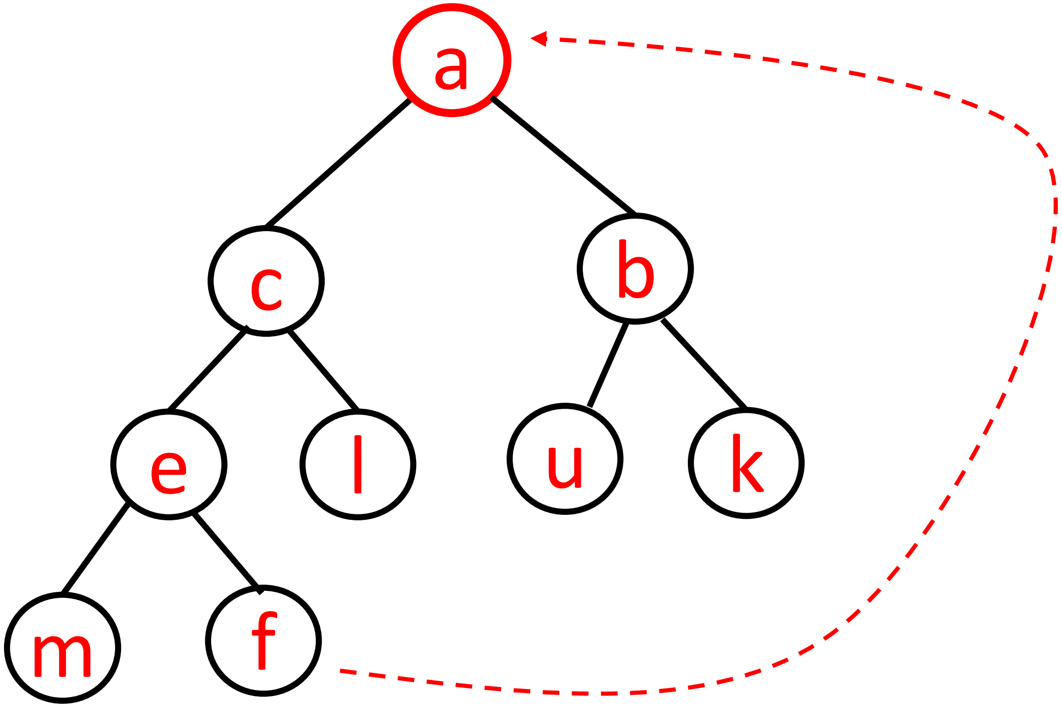
I will show you a faster method two lectures from now.

Heap.removeMin()

returns root element



removeMin()

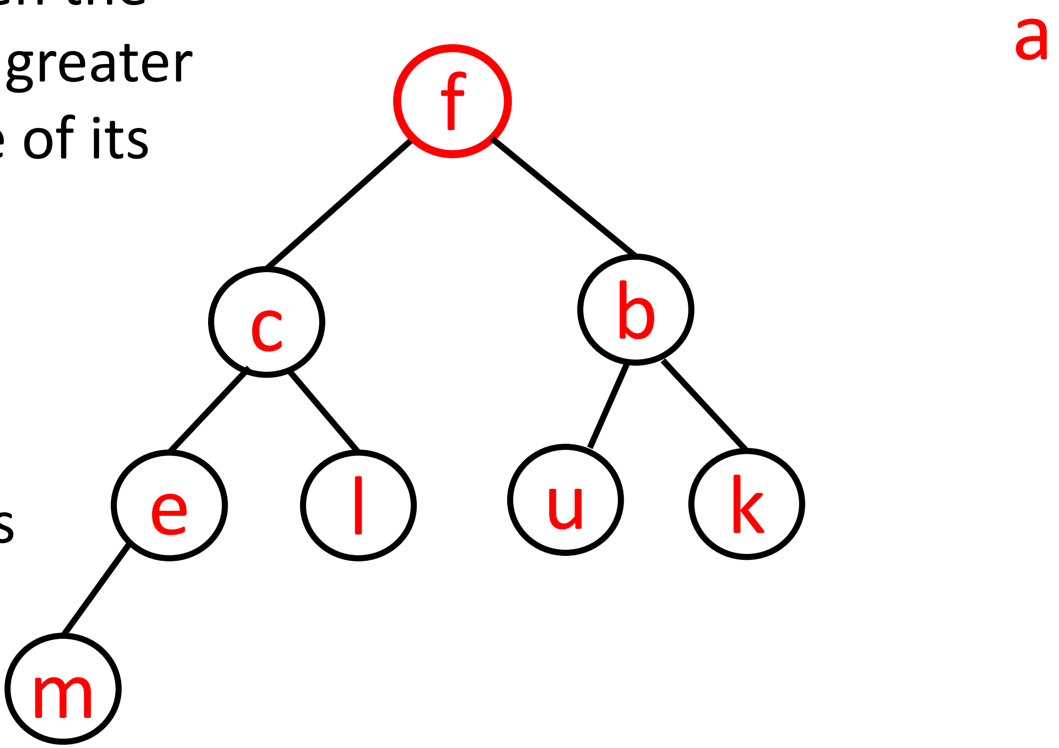


removeMin()

Claim: if the root has two children, then the new root *will* be greater than at least one of its children.

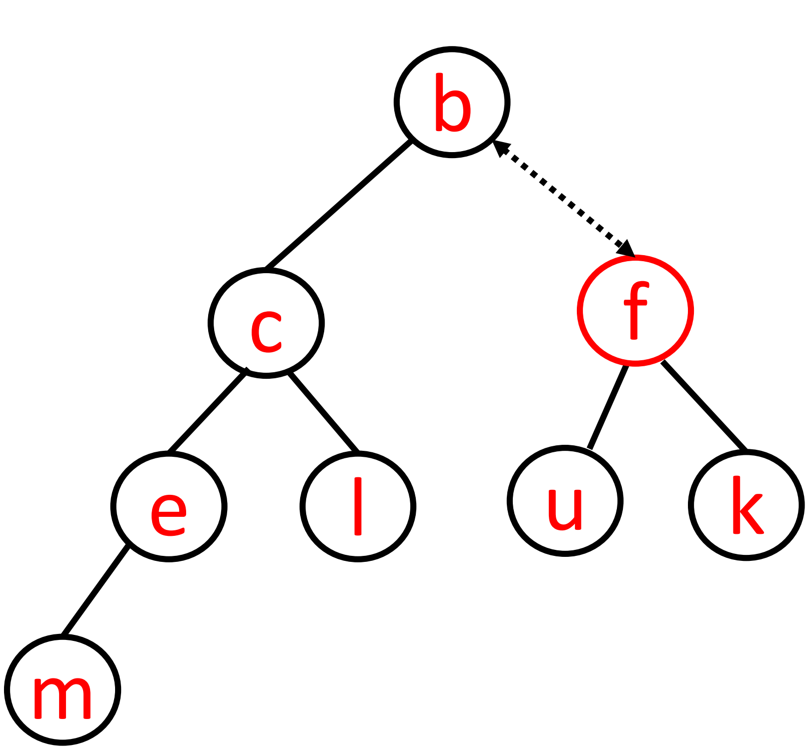
Why?

How to solve this problem?



removeMin()

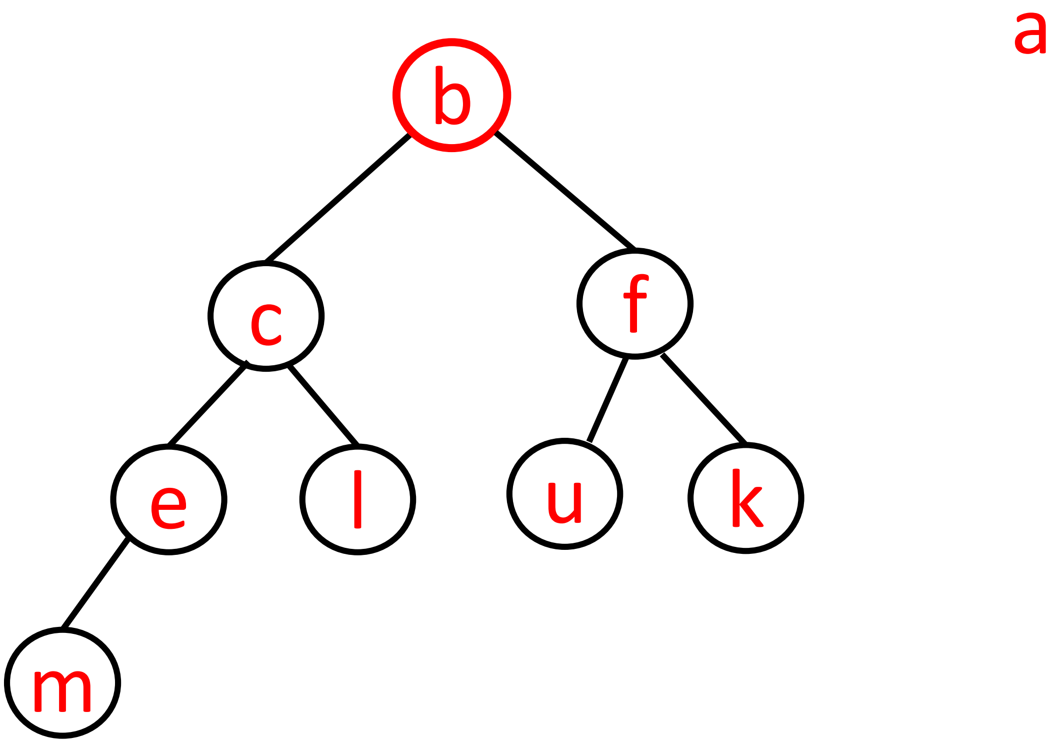
Swap elements with smaller child.



Keep swapping with smaller child, if necessary.

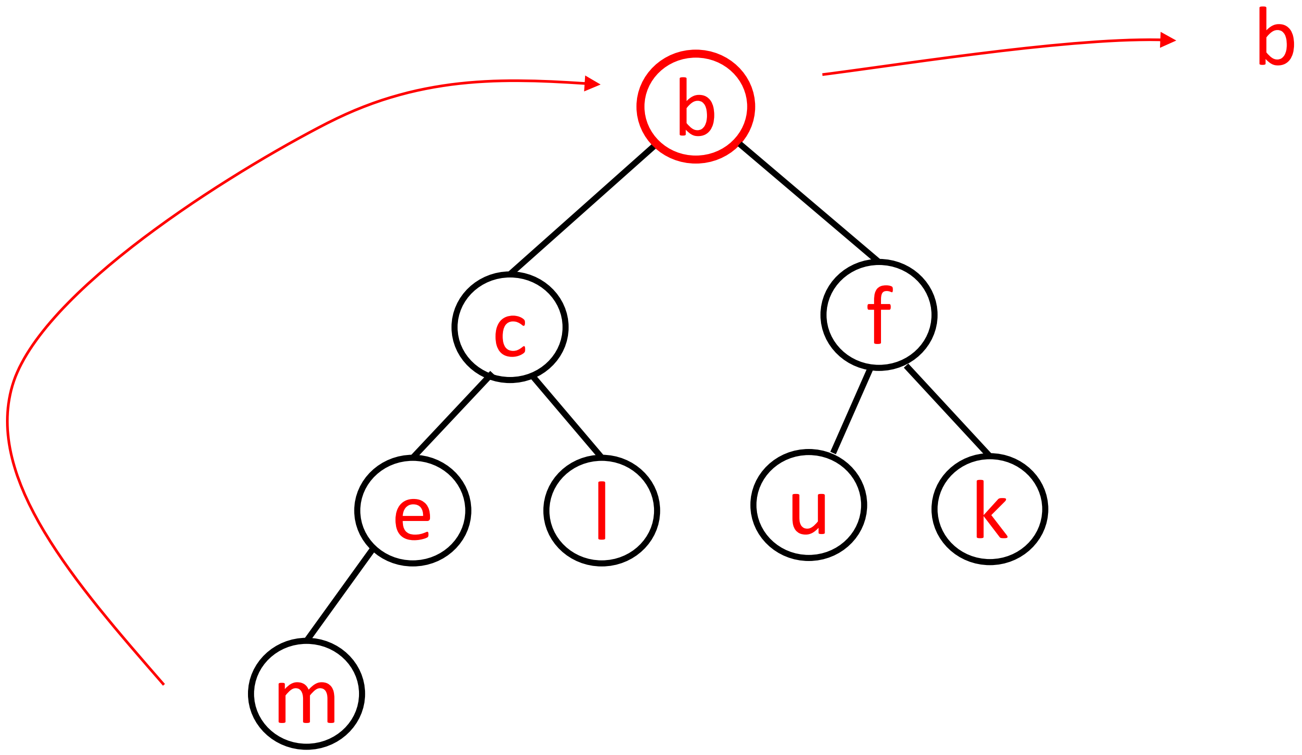
removeMin()

Let’s do it again.



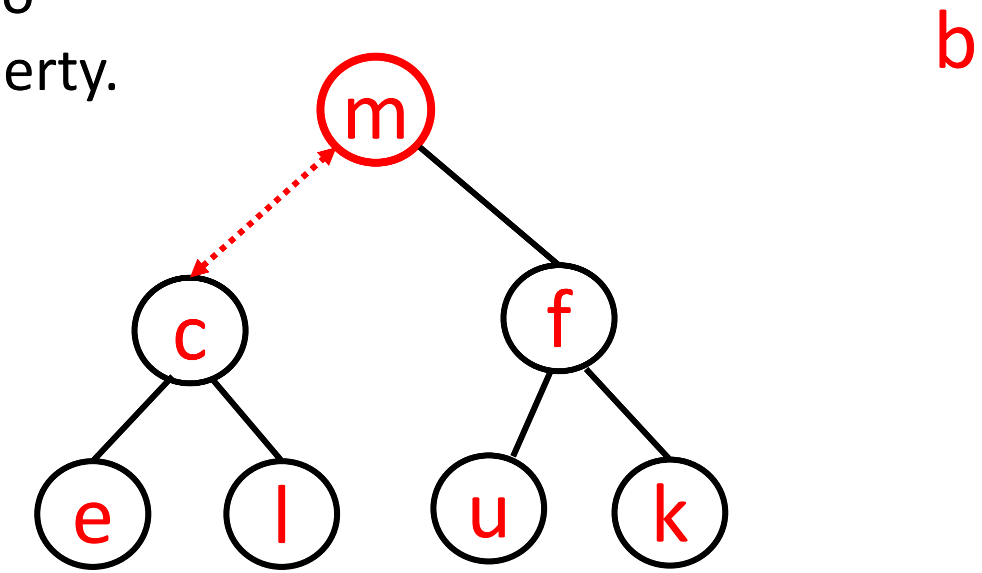
removeMin()

Let’s do it again.

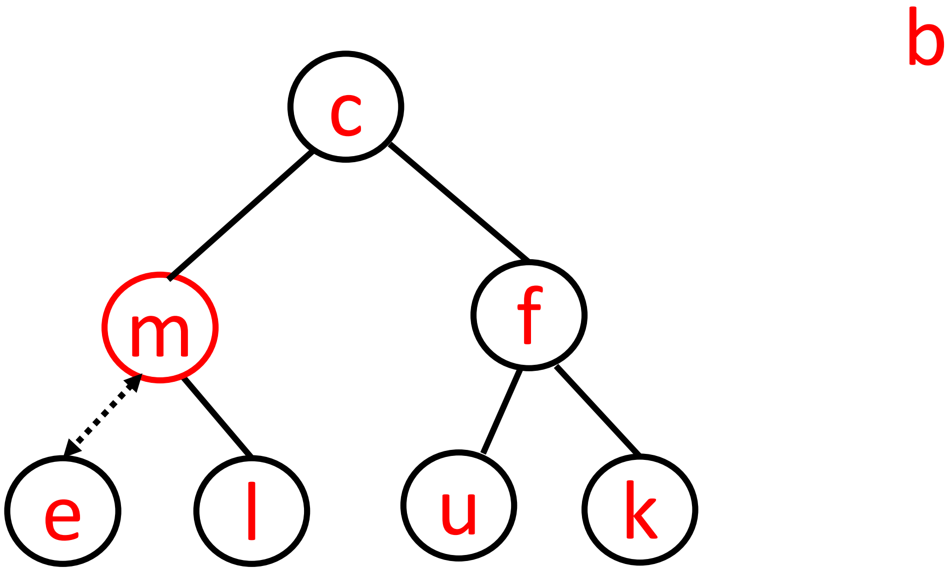


removeMin()

Now swap with smaller child, if necessary, to preserve heap property.

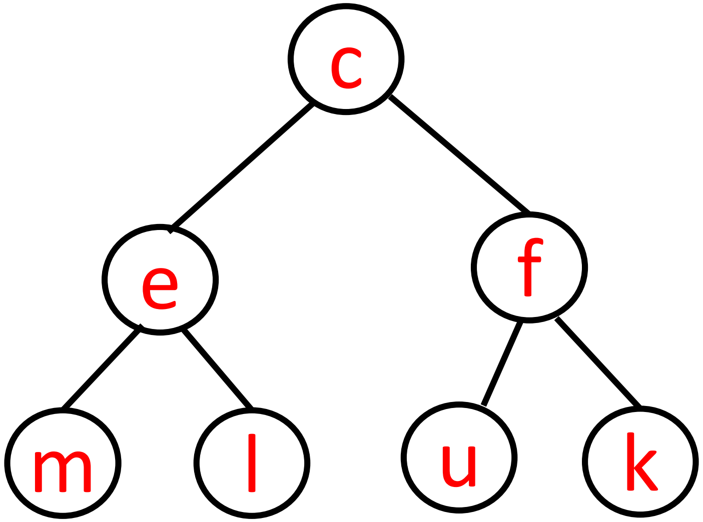


removeMin()



Keep swapping with smaller child, if necessary.

removeMin()



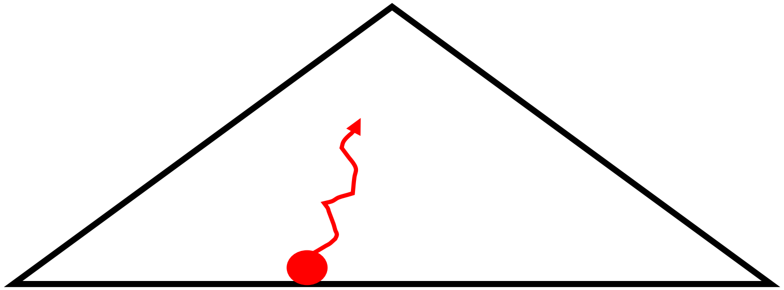
b

```
removeMin(){
    tmp = root.element
    remove last leaf node and put its element into the root
    cur = root
    while [redacted]

    {

    }
   return tmp
}
```

```
removeMin(){
     tmp = root.element
     remove last leaf node and put its element into the root
     cur = root
     while ( (cur has a left child) and
                ( (cur.element > cur.left.element) or
                  (cur has right child and cur.element > cur.right.element)) )
     {


     }
    return tmp
}
```

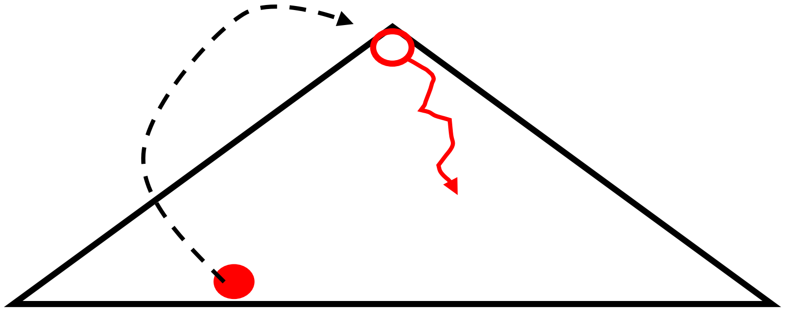
```
removeMin(){
    tmp = root.element
    remove last leaf node and put its element into the root
    cur = root
    while ( (cur has a left child) and
            ( (cur.element > cur.left.element) or
              (cur has right child and cur.element > cur.right.element)) )
    {   minChild = child with the smaller element
        swapElement(cur, minChild)
        cur = minChild
    }
   return tmp
}
```

add(element)



“upHeap”

removeMin()



“downHeap”

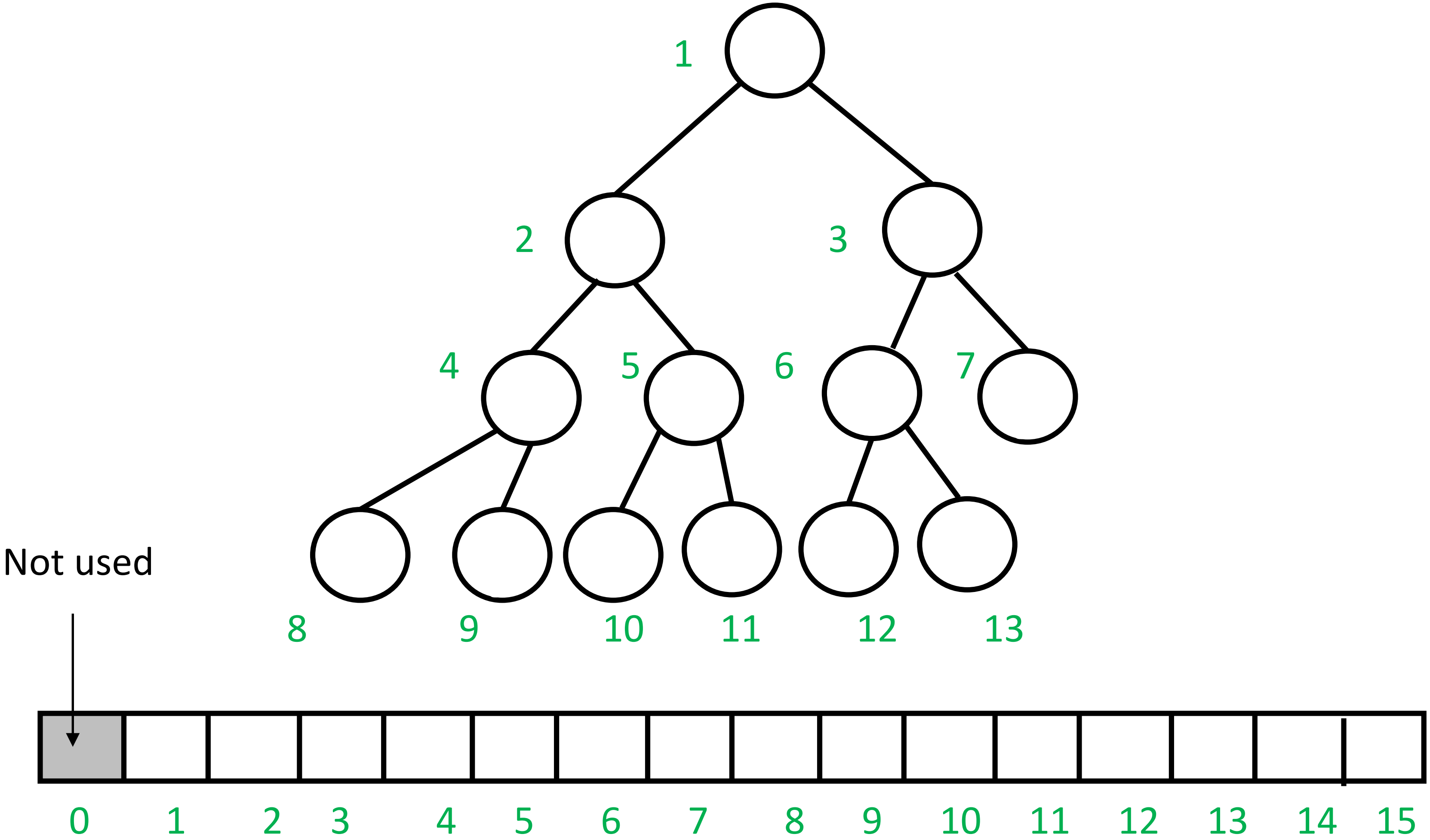
Q: What about remove(element) ?

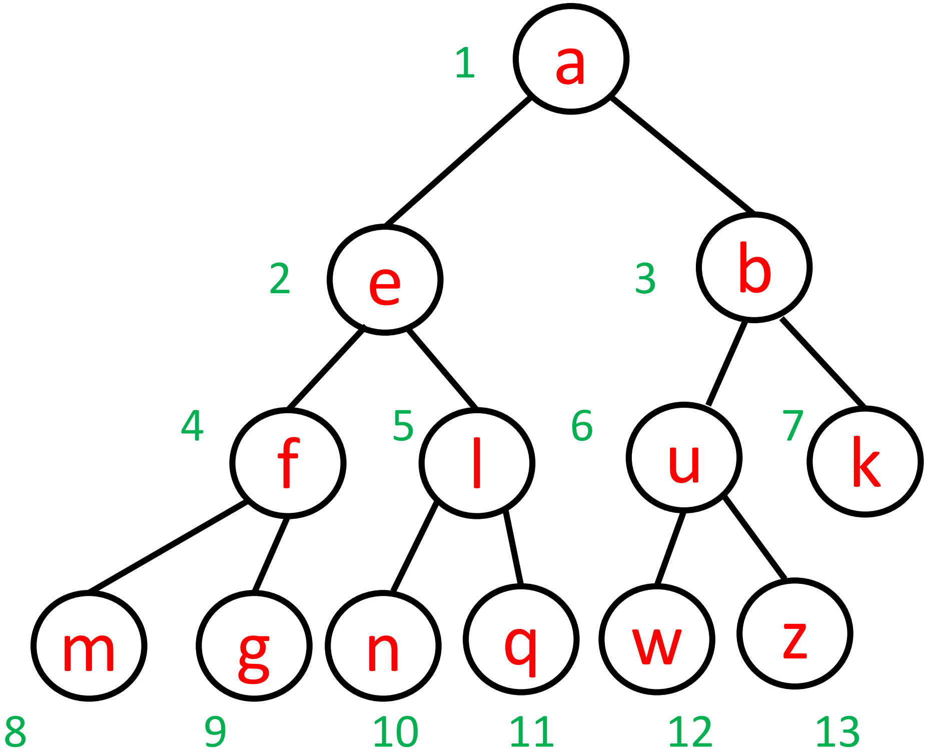
Q: What about remove(element) ?

A: Worst case $\Theta(n)$

Best case (not discussed)

Heap (array implementation)





Not used

	a	e	b	f	l	u	k	m	g	n	q	w	z		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Next two lectures

- write add(element) and removeMin() using array indices

- best and worst case

- faster algorithm for building a heap