COMP 250

Lecture 26

heaps 1

Nov. 9, 2018

Priority Queue (ADT)

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”.

(Comparable means that there is an ordering, as in the Java Comparable interface.)

Priority Queue ADT

- add(element)

- removeMin()

  “highest” priority = “number 1” priority

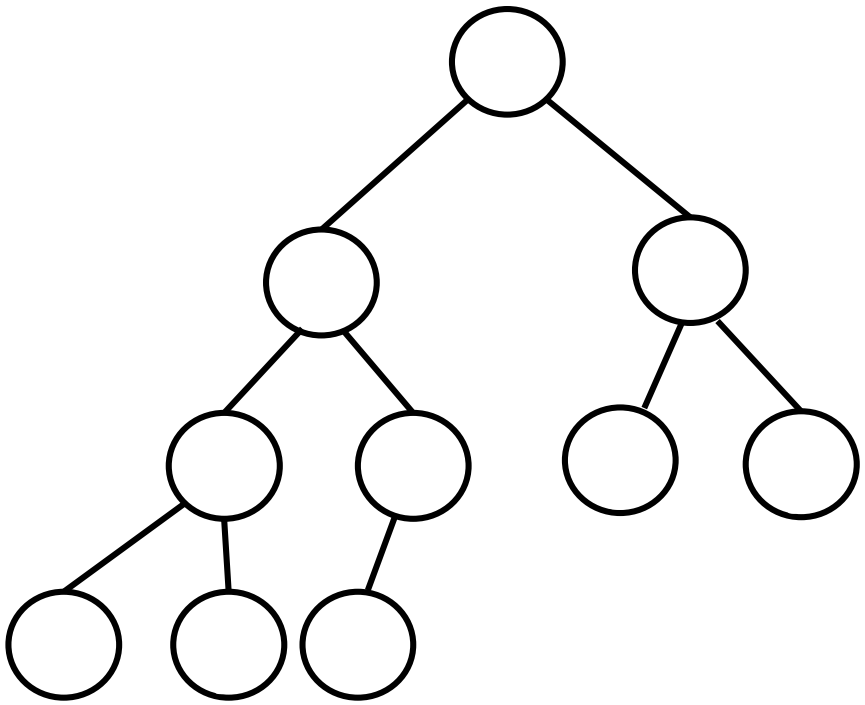
Similar to enqueue(e) and dequeue(), but now dequeue() is called removeMin() and the policy is different from FIFO policy.

How to implement a Priority Queue ?

- BAD: sorted arraylist or linked list
- GOOD: heap (today and next lecture)

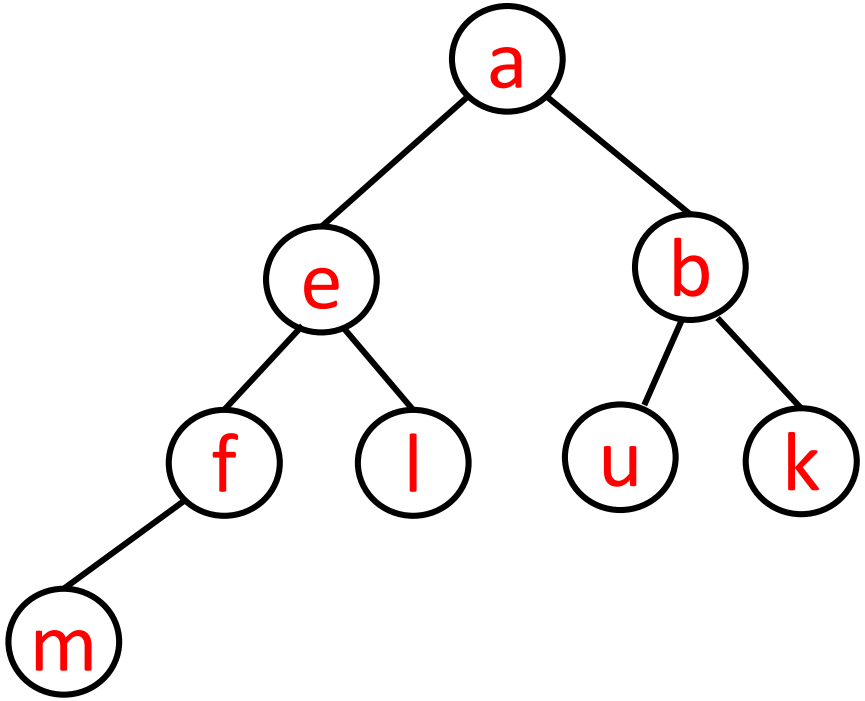
Not the same “heap” you hear about in COMP 206.

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

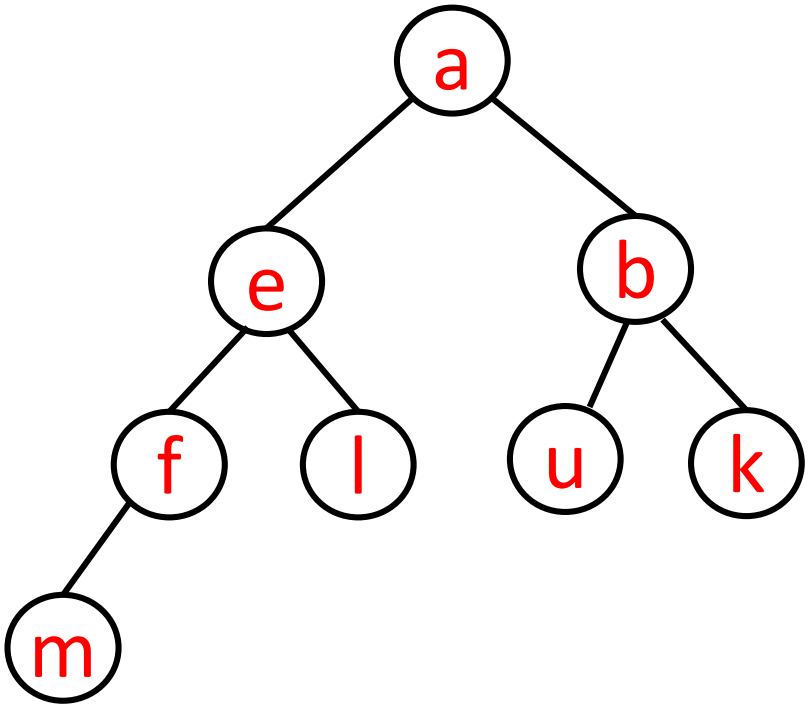
min Heap (definition)



Complete binary tree with unique comparable elements, such that each node's element is less than its children's elements. (**NOT a binary search tree !**)

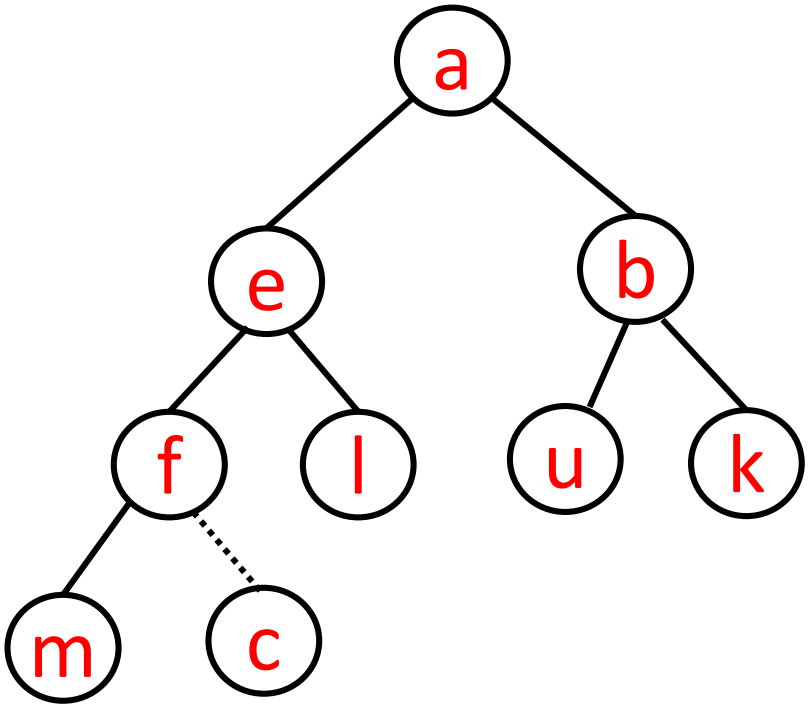
add(element)

e.g. add(c)



add(element)

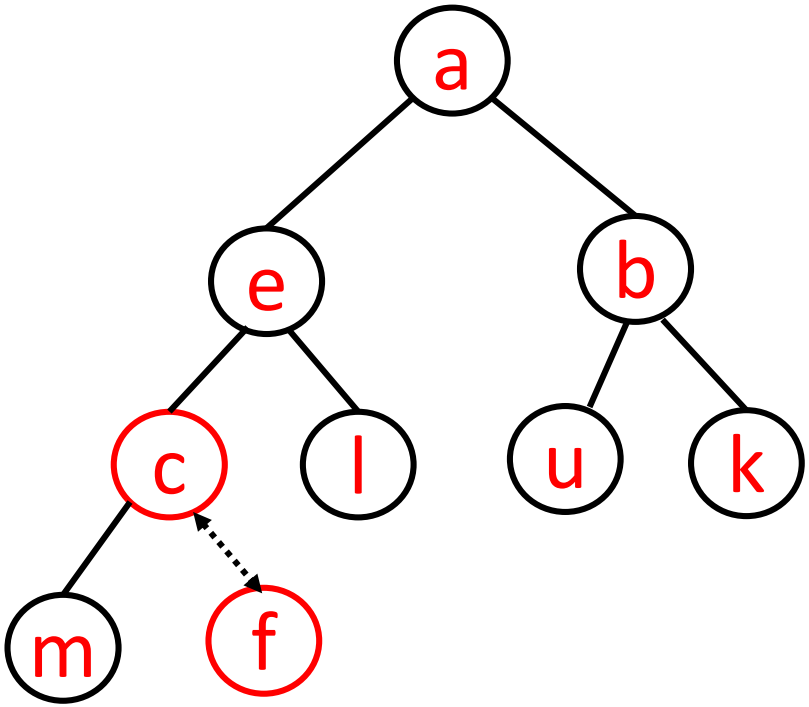
e.g. add(c)



Problem : adding at the next available slot destroys the heap property.

We swap c with its parent f.

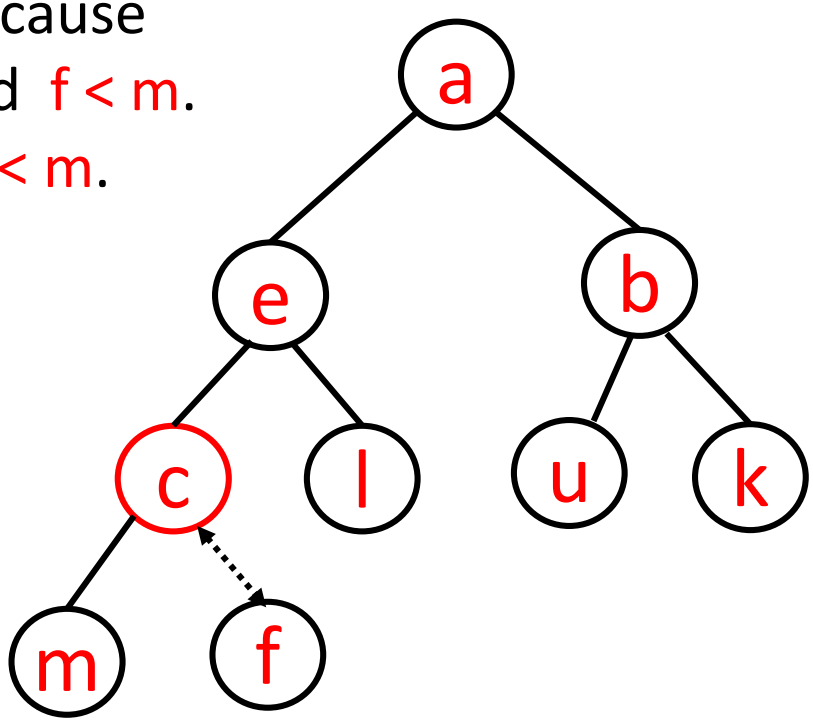
Q: Can this create a problem with c's former sibling, who is now c's child?



We swap c with its parent f.

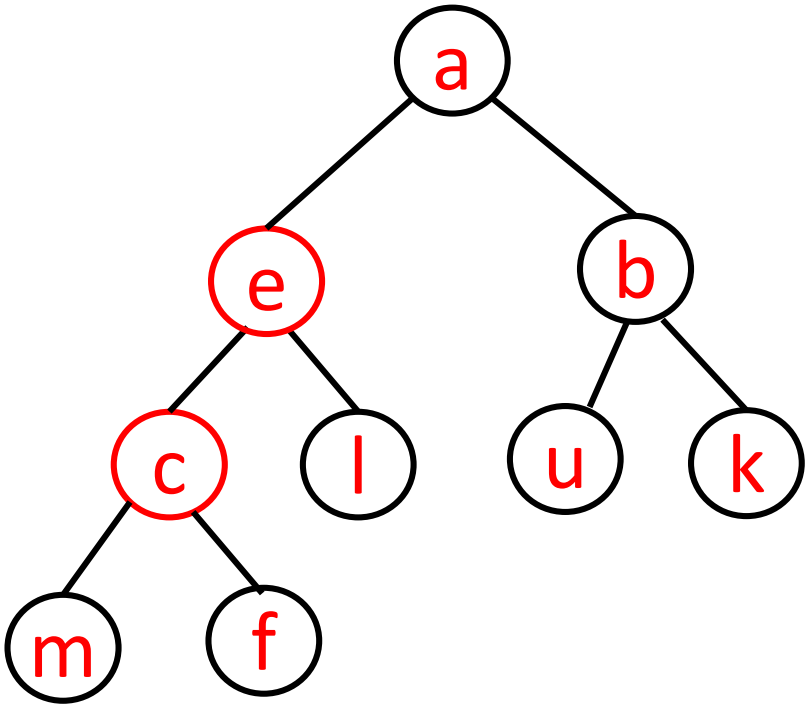
Q: Can this create a problem with c's former sibling, who is now c's child?

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



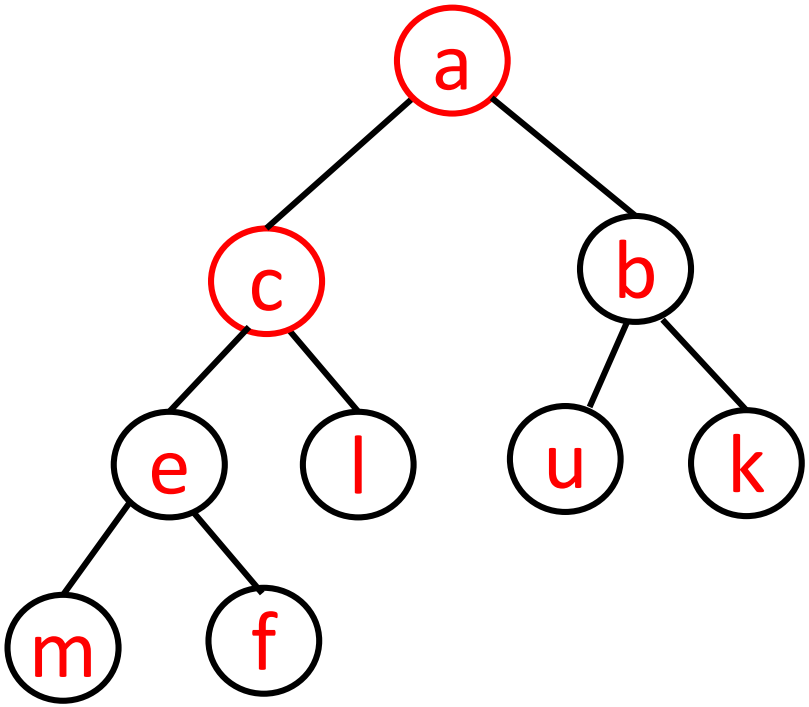
Q: Are we done ?

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



add(element)

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

add(element)

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

    }
}
```

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
    }
}
```

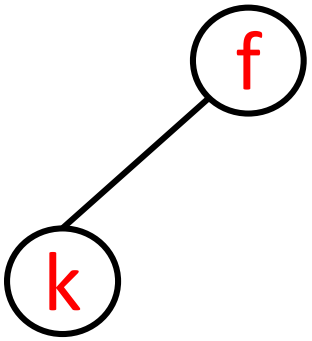
How to build a heap?

add(k)

add(f)

How to build a heap?

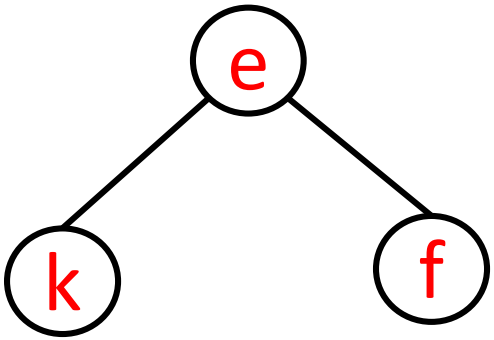
add(k)
add(f)



add(e)

How to build a heap?

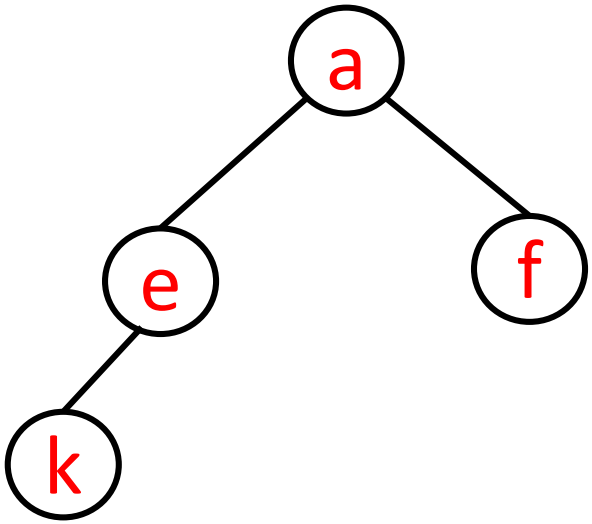
add(k)
add(f)
add(e)



add(a)

How to build a heap?

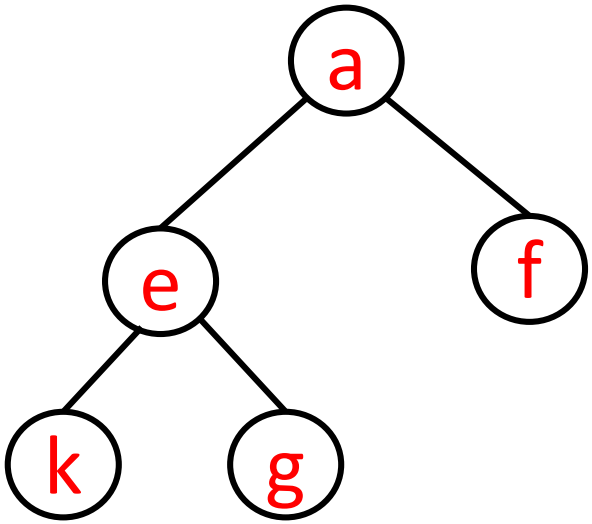
add(k)
add(f)
add(e)
add(a)



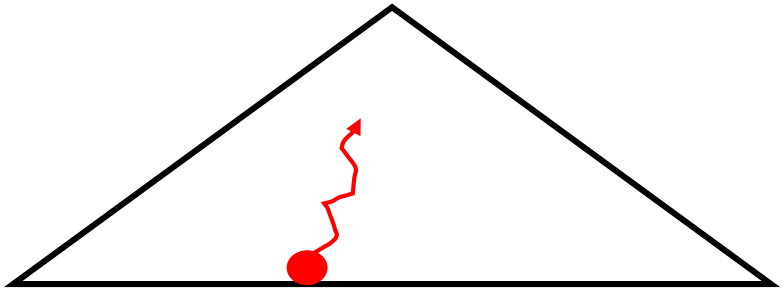
add(g)

How to build a heap?

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

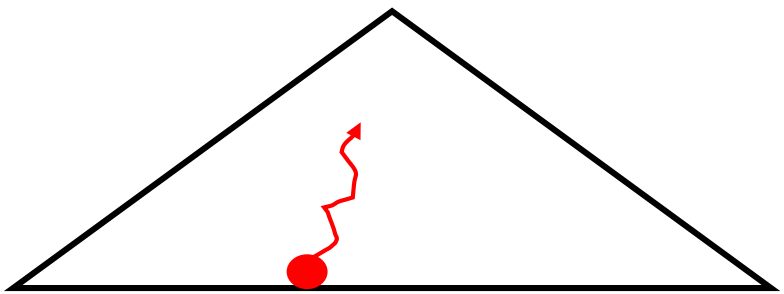


add(element)



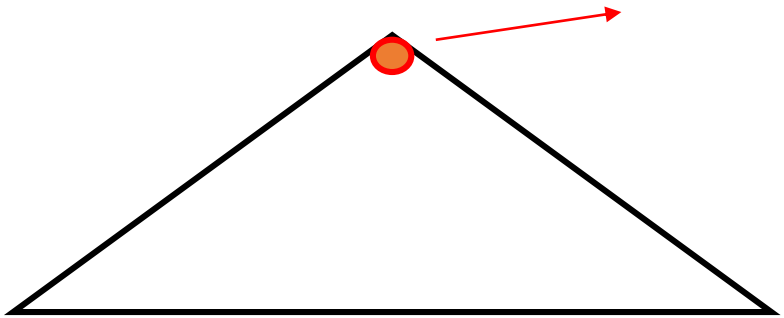
“upHeap”

add(element)



"upHeap"

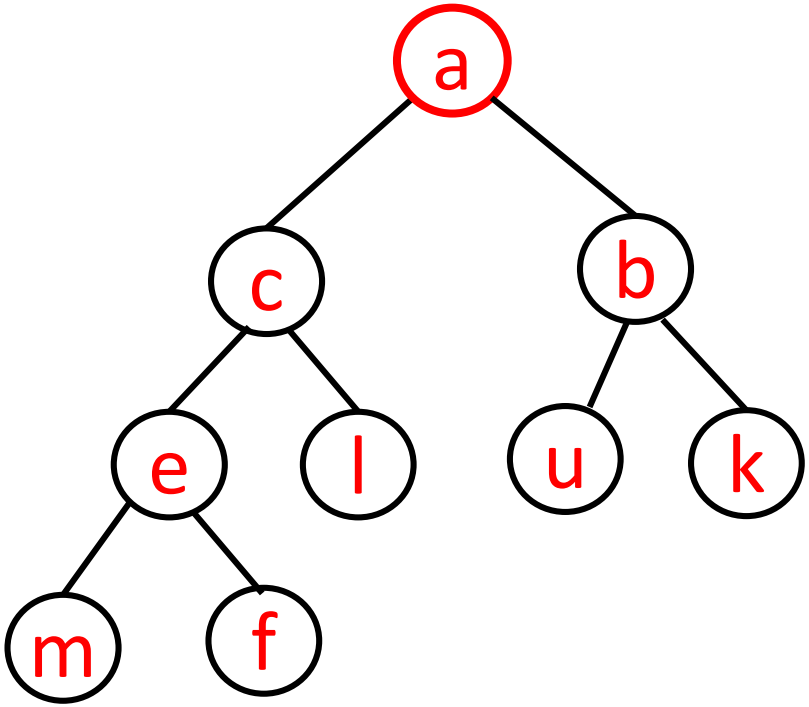
removeMin()



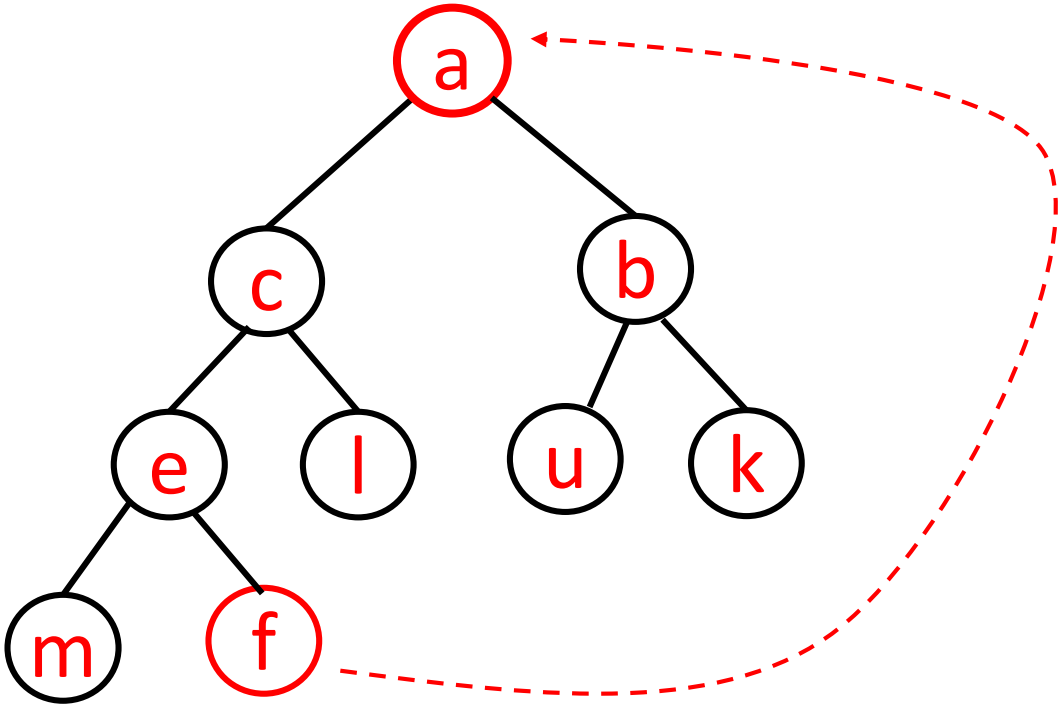
Q: How to do this?

removeMin()

returns root element



removeMin()

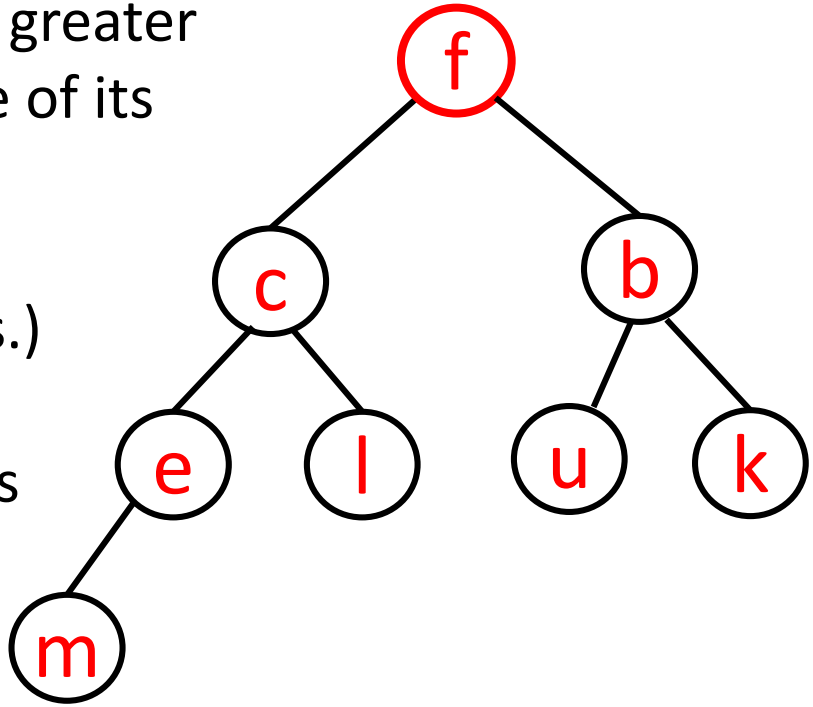


removeMin()

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

Why? (Exercises.)

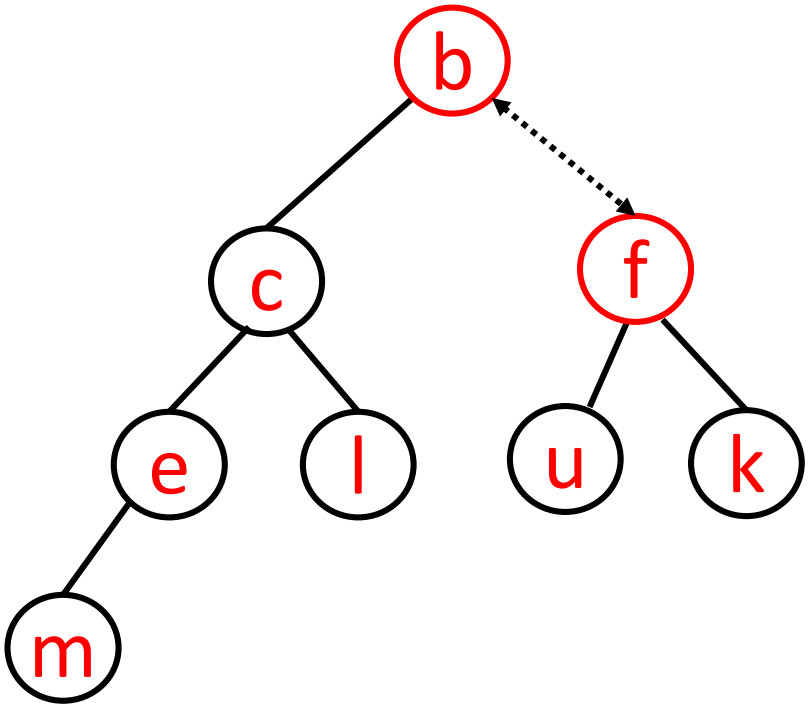
How to solve this problem?



a is returned

removeMin()

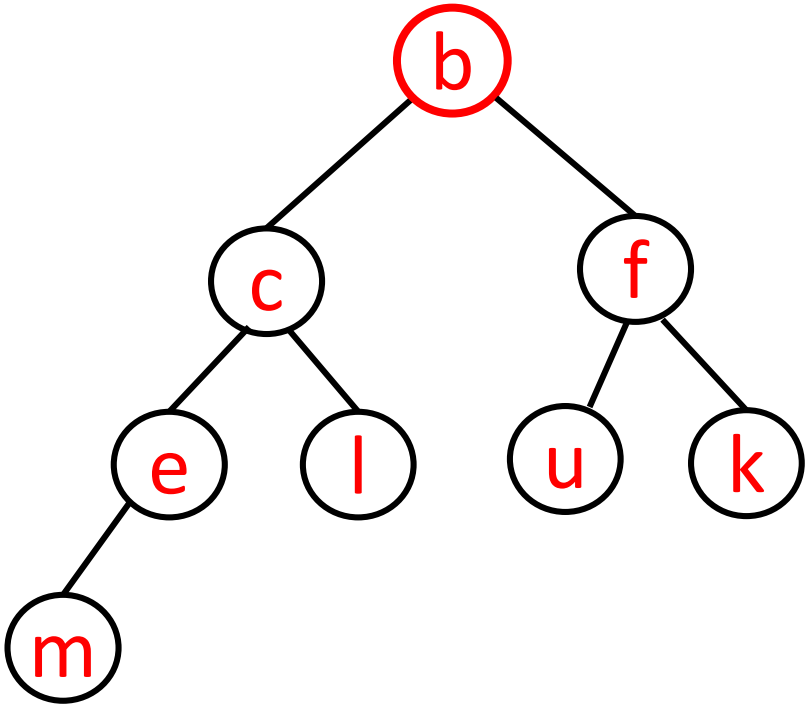
Swap elements with smaller child.



Keep swapping with smaller child, if necessary.

removeMin()

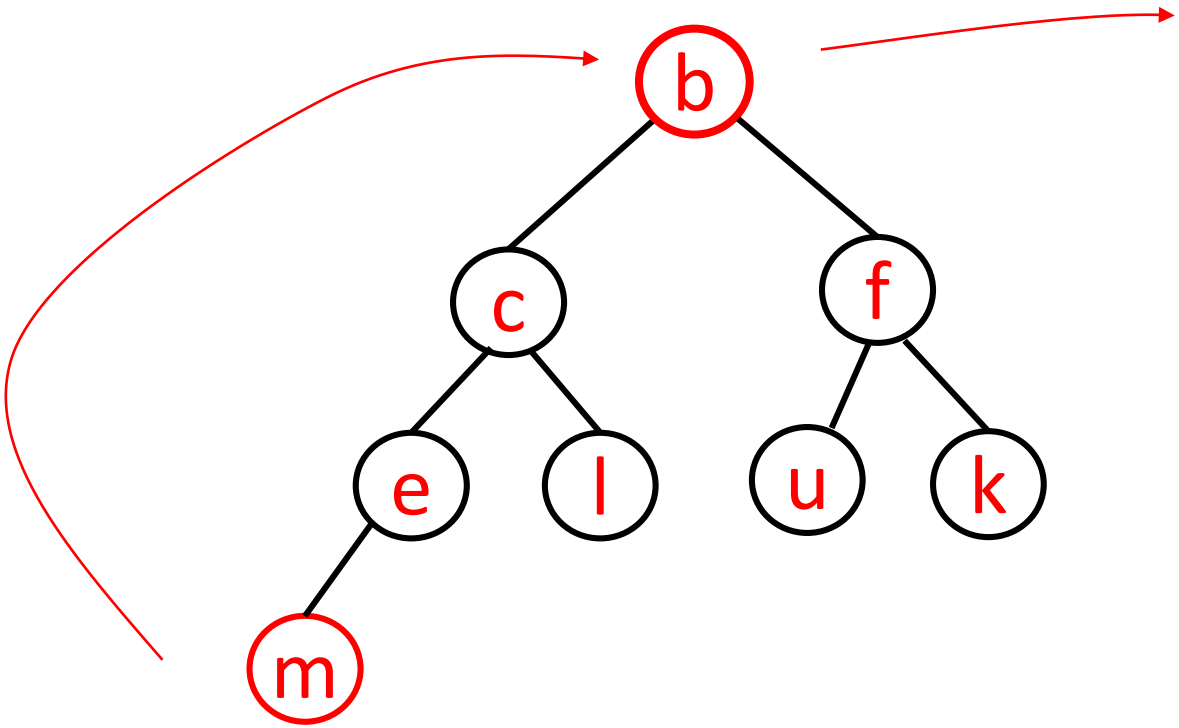
Let's do it again.



removeMin()

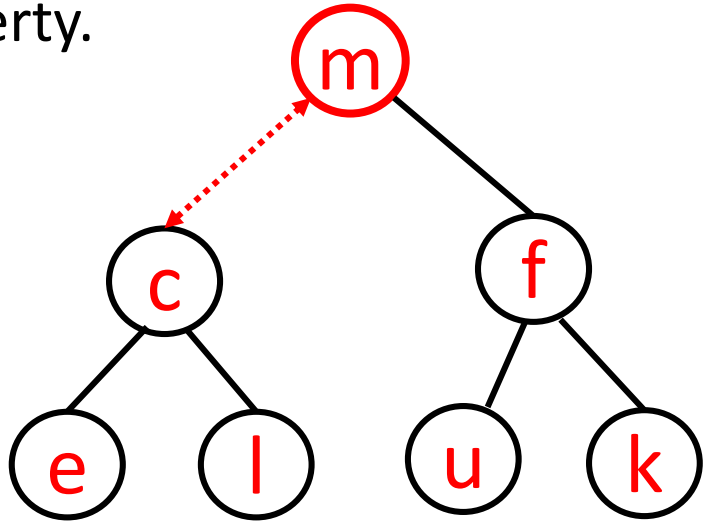
Let's do it again.

b is returned



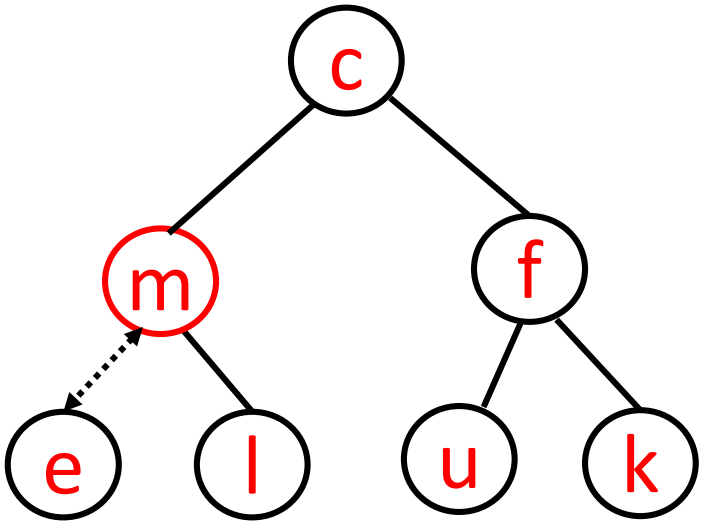
removeMin()

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

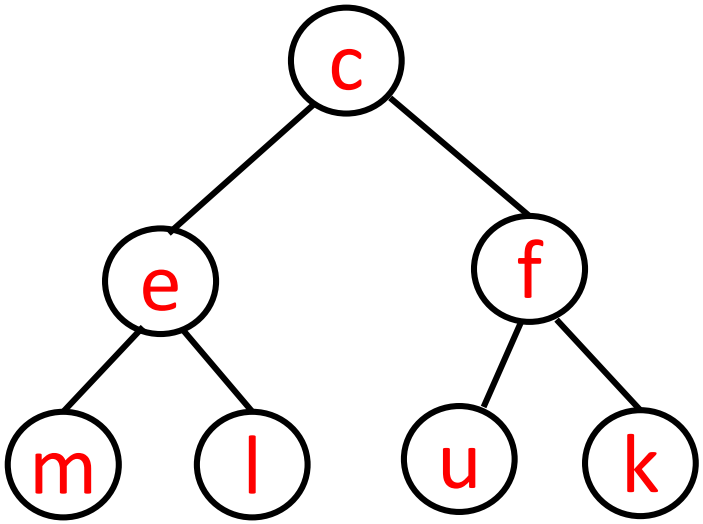


removeMin()

Keep swapping
with smaller child,
if necessary.



removeMin()



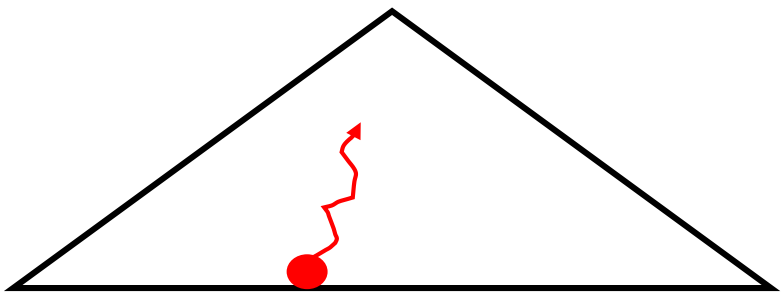
```
removeMin(){
    tmp = root.element
    remove last leaf node and put its element into the root
    cur = root
    while
    {
    }
    return tmp
}
```

```
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
}
```

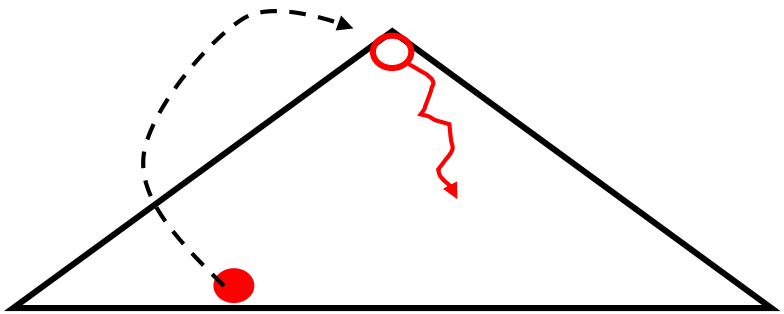
```
removeMin(){
    tmp = root.element
    remove last leaf node and put its element into the root
    cur = root
    while ( (cur has a left child) and
            ( (cur.element > cur.left.element) or
              (cur has right child and cur.element > cur.right.element)) )
    {     minChild = child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

add(element)



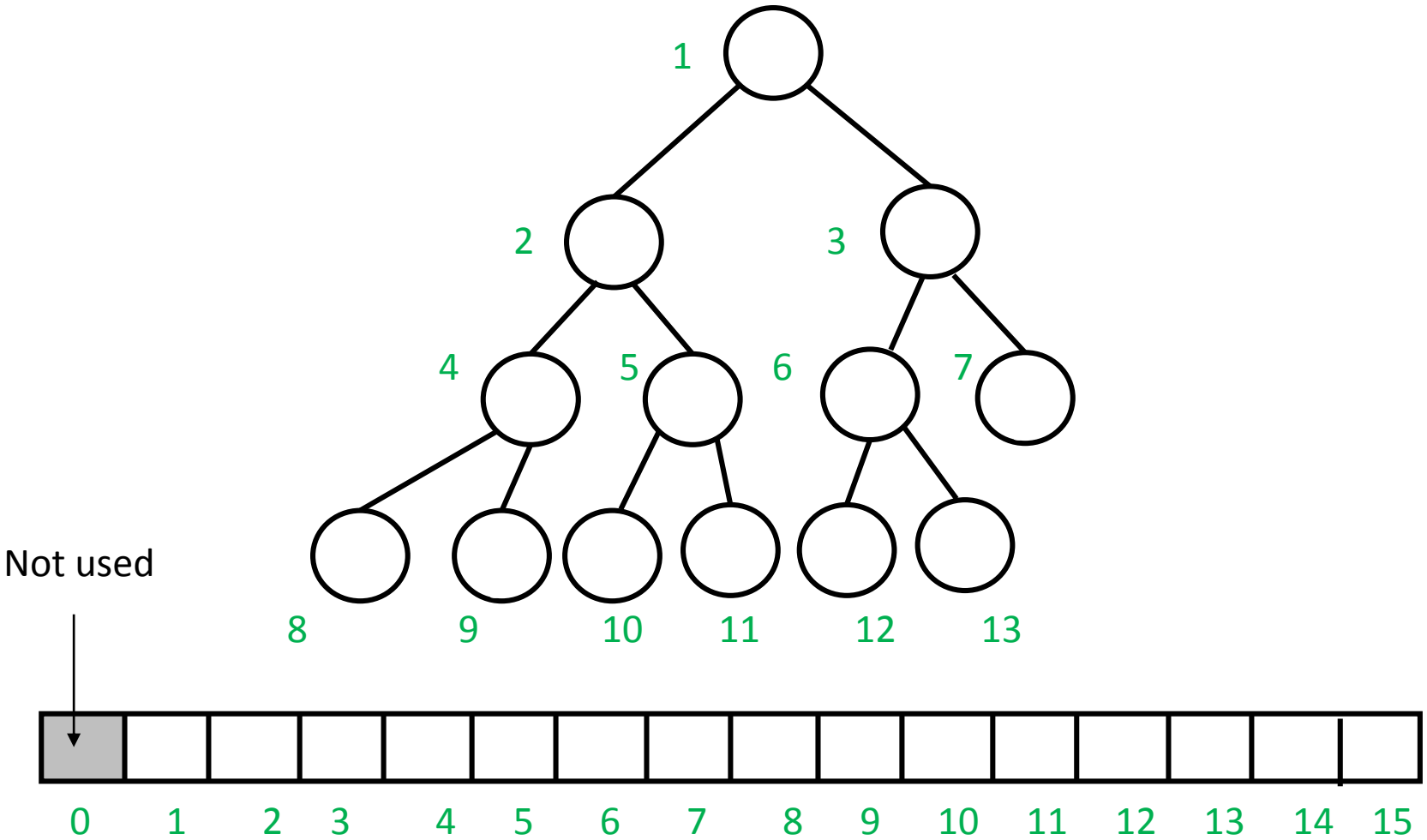
“upHeap”

removeMin()

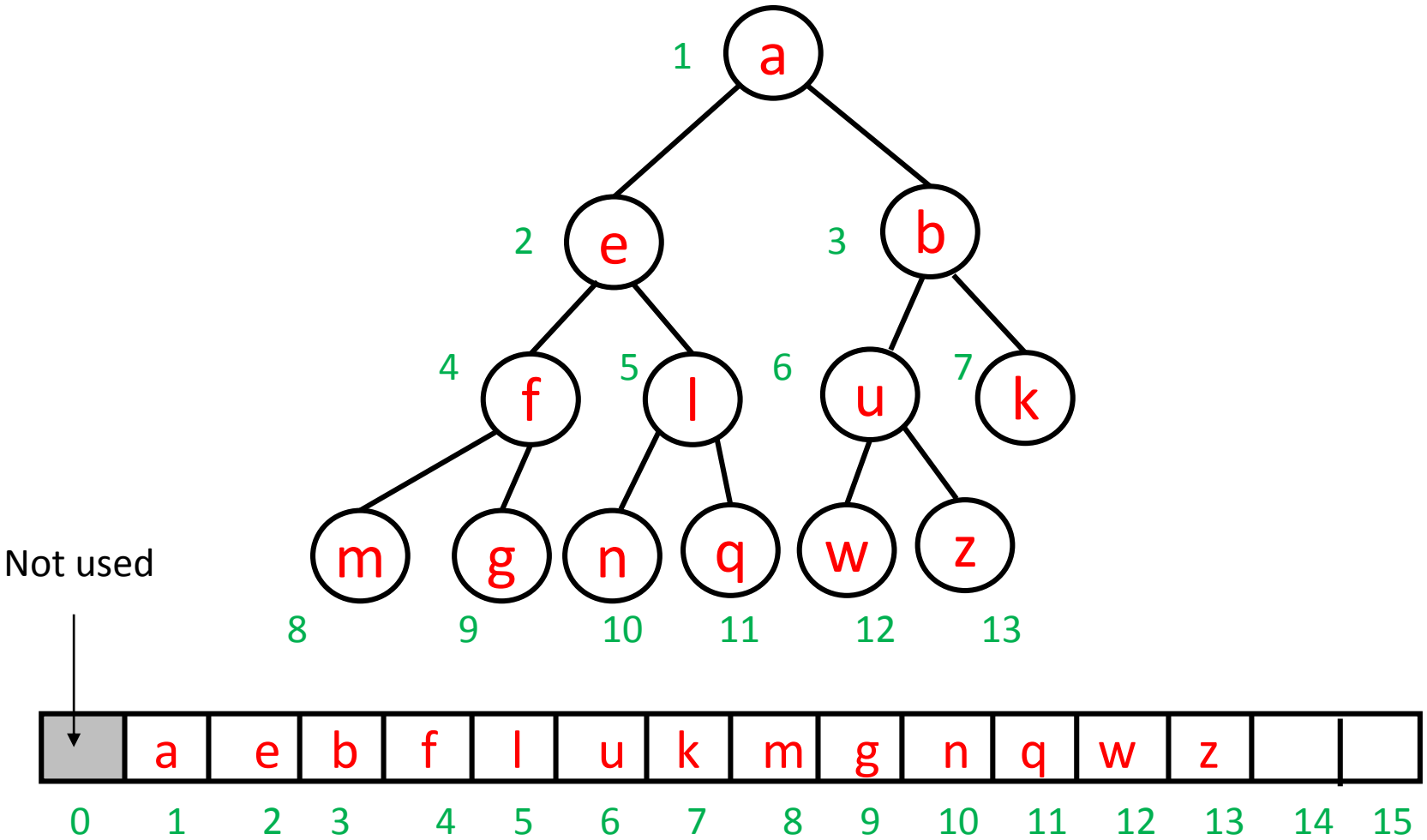


“downHeap”

Heap (array implementation)

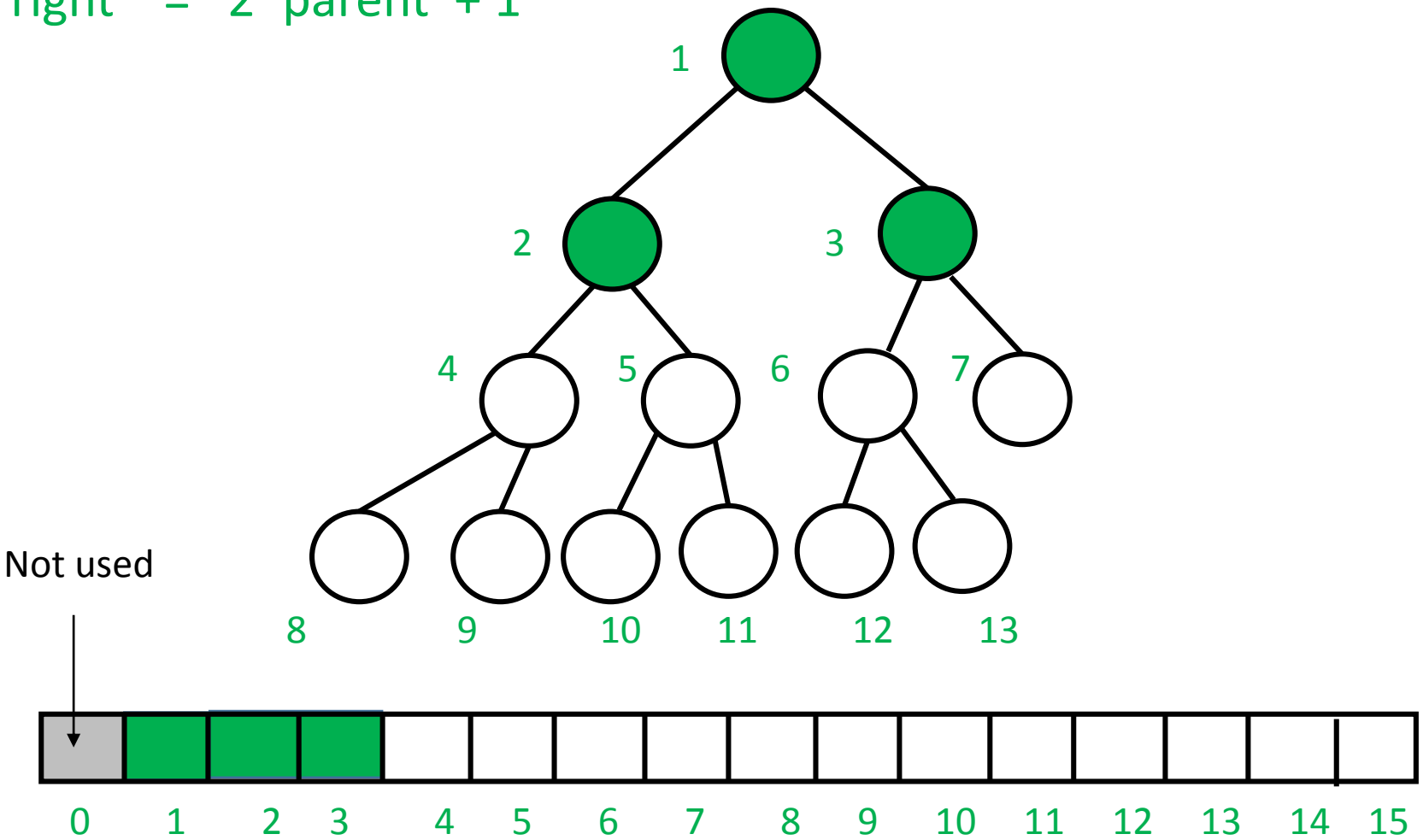


Heap (array implementation)



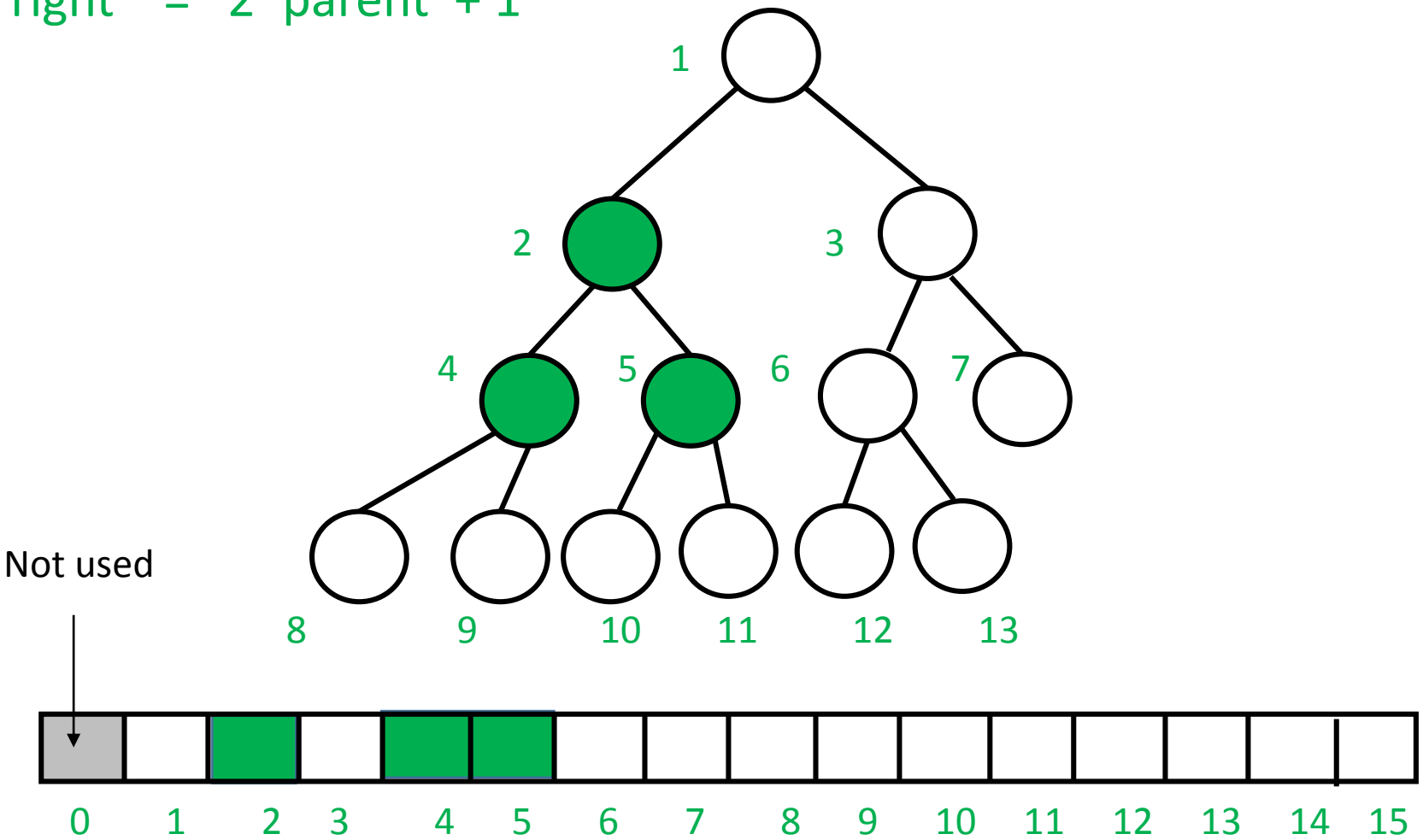
Heap index relations

parent = child / 2
left = 2*parent
right = 2*parent + 1



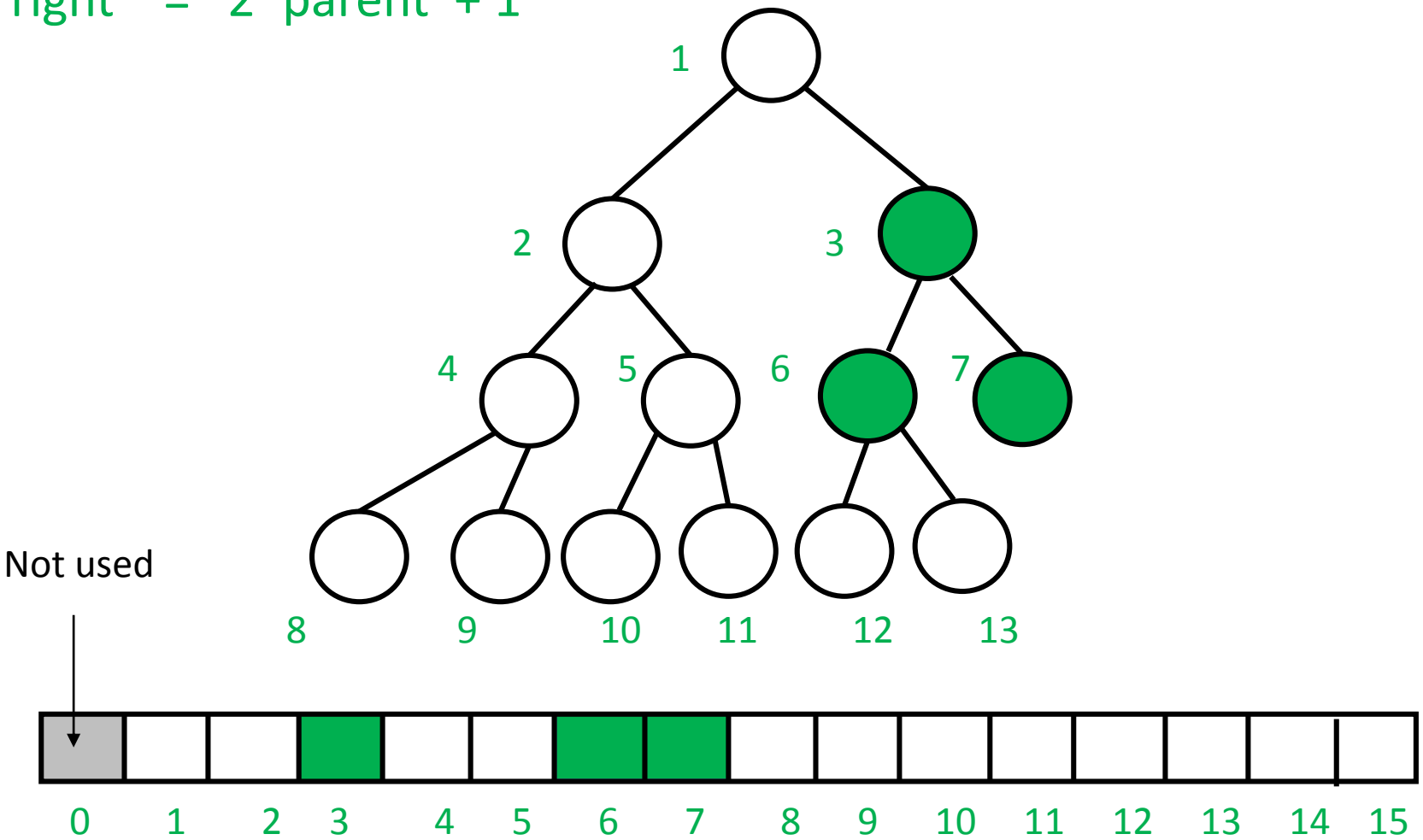
Heap index relations

parent = child / 2
left = 2*parent
right = 2*parent + 1



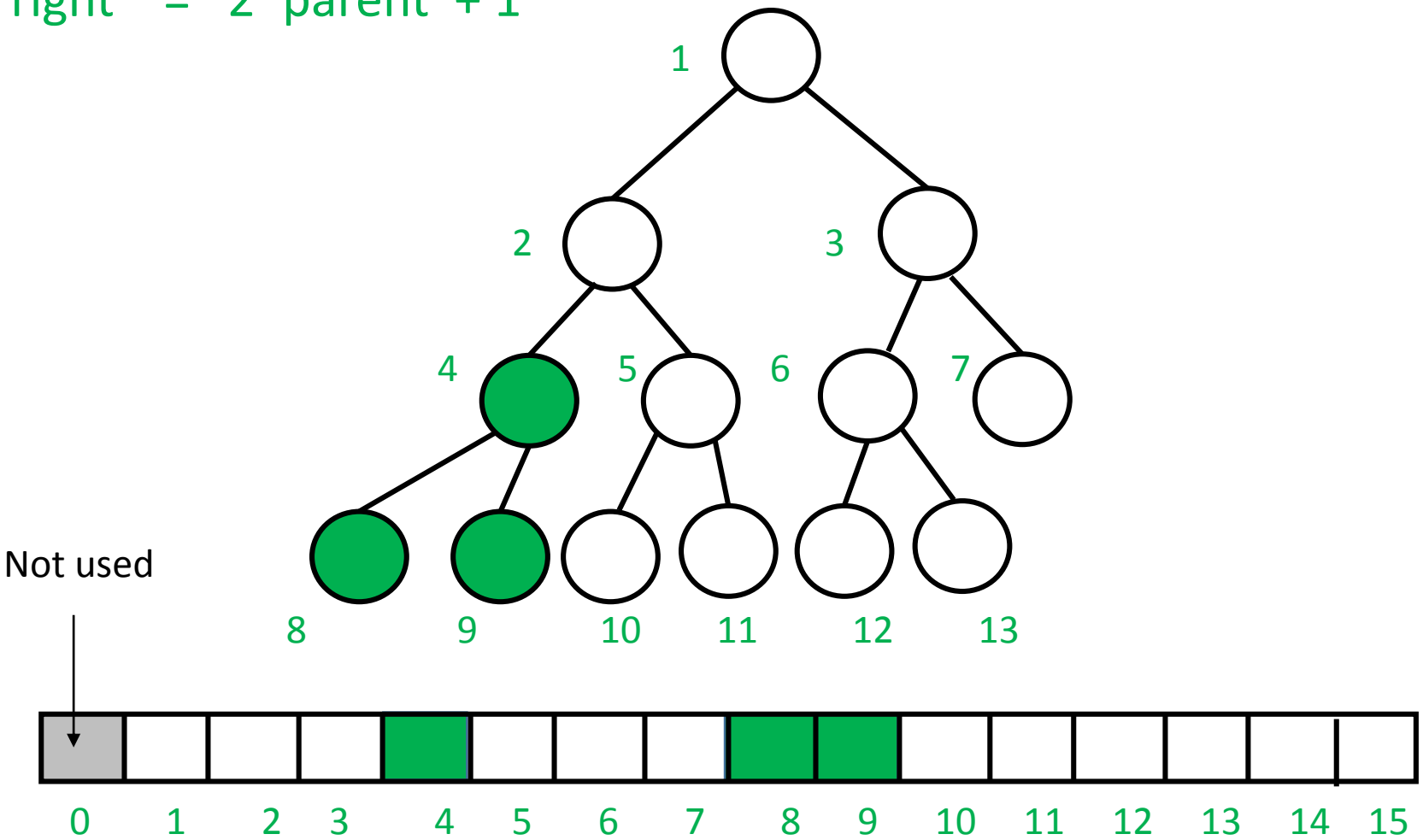
Heap index relations

parent = child / 2
left = 2*parent
right = 2*parent + 1



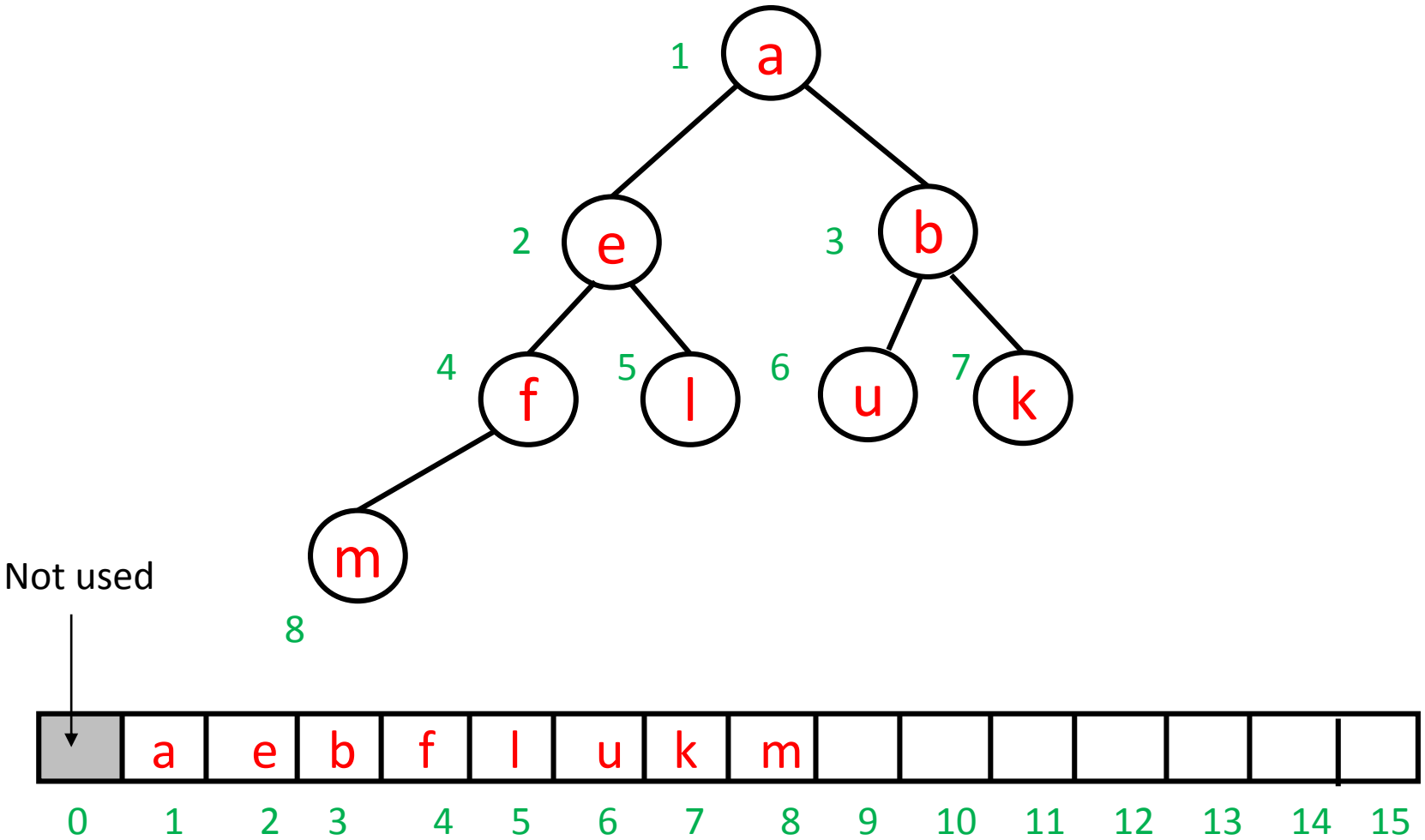
Heap index relations

parent = child / 2
left = 2*parent
right = 2*parent + 1

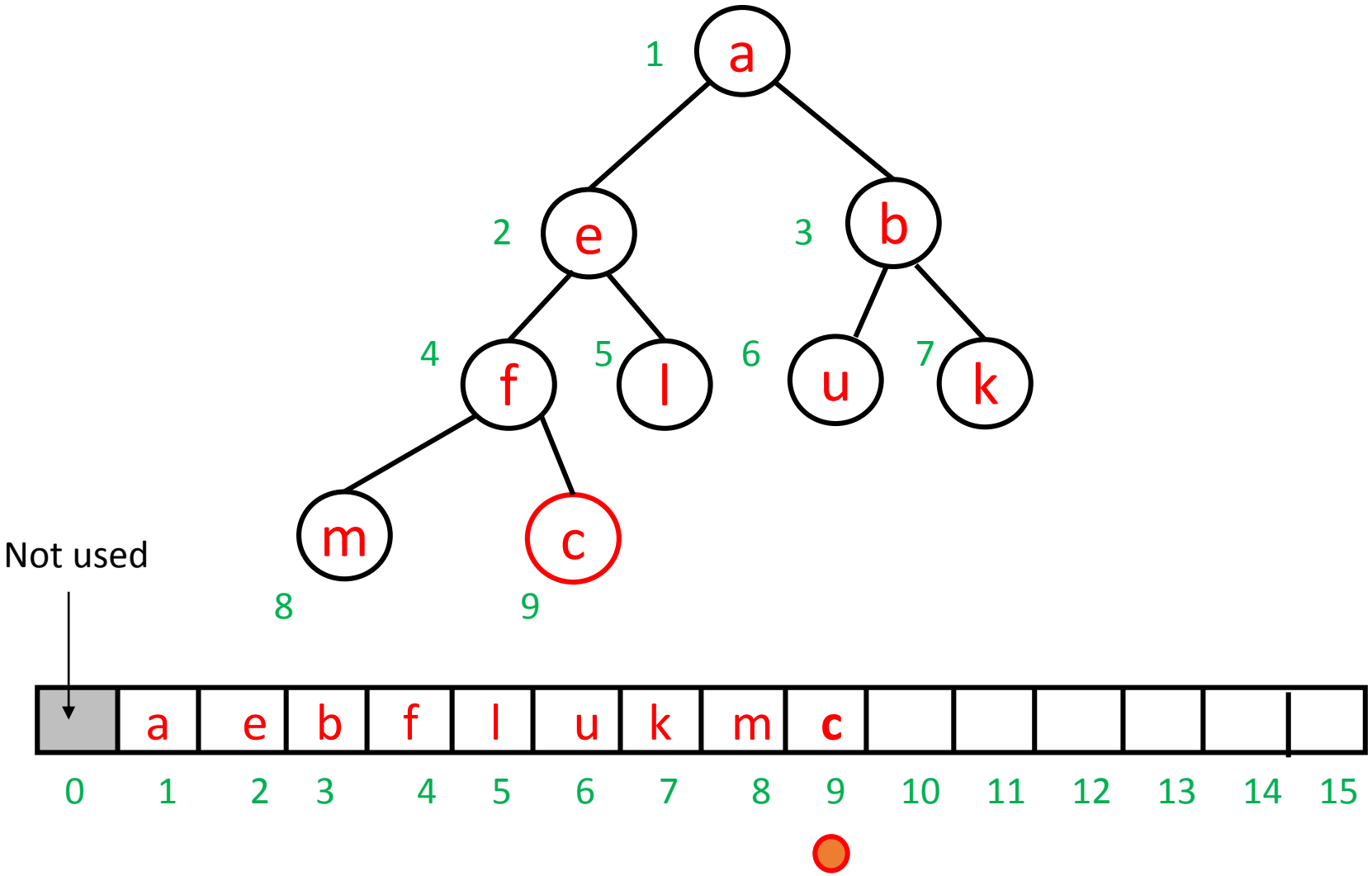


Time permitting…

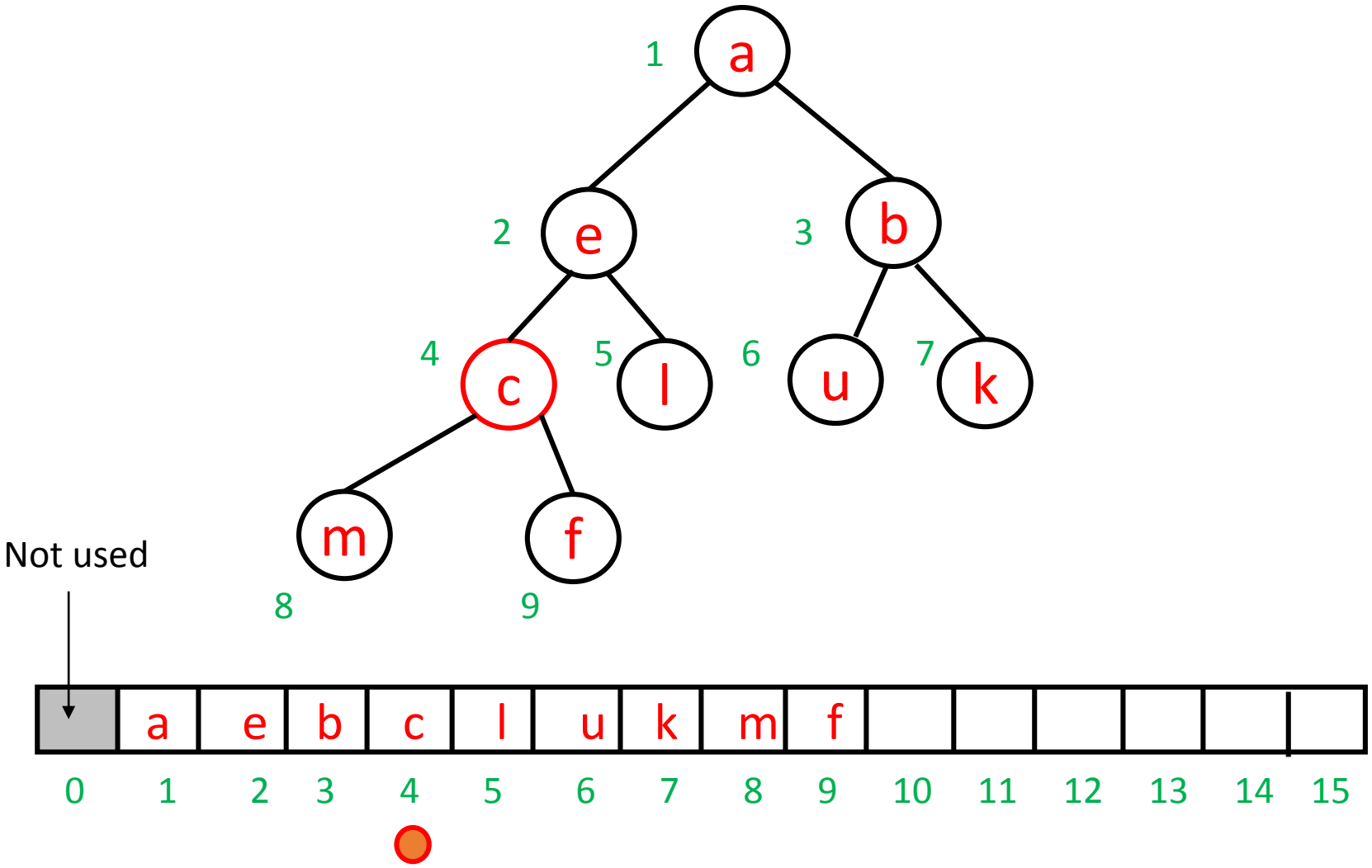
e.g. add(c)


1
a
2
e
3
b
4
f
5
l
6
u
7
k
m
8
Not used
a
e
b
f
l
u
k
m
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

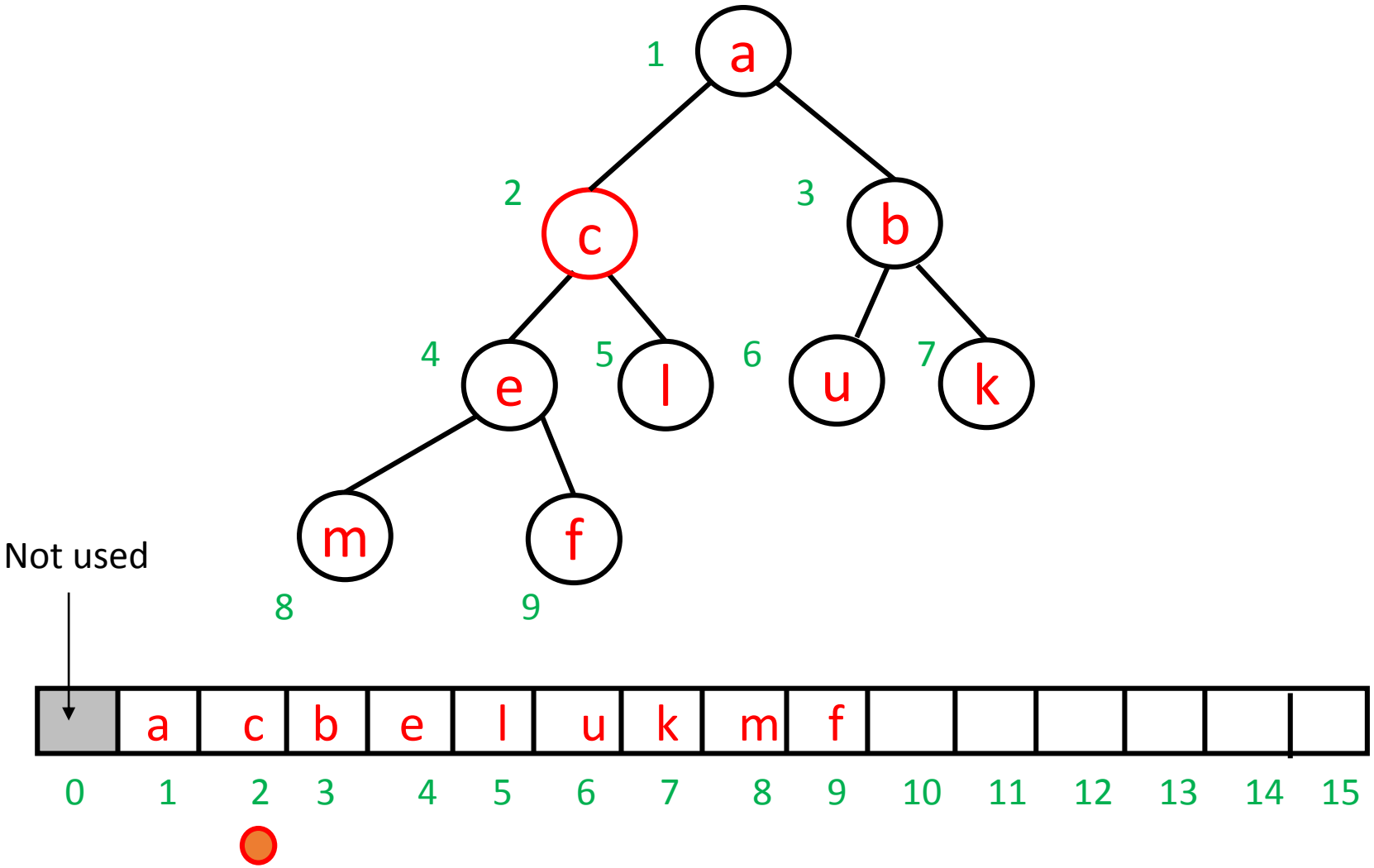
e.g. add(c)


1
a
2
e
3
b
4
f
5
l
6
u
7
k
m
c
8
9
Not used
a
e
b
f
l
u
k
m
c
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

e.g. add(c)


1
a
2
e
3
b
4
c
5
l
6
u
7
k
m
f
8
9
Not used
a
e
b
c
l
u
k
m
f
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

e.g. add(c)


1
a
2
c
3
b
4
e
5
l
6
u
7
k
m
f
8
9
Not used
a c b e l u k m f
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

```
add(element ){
   size = size + 1          // number of elements in heap
   heap[ size ] = element   // assuming array
                            //  has room for another element
   i = size

  //  the following is sometimes called "upHeap"

   while ( i > 1  and  heap[i] < heap[ i/2 ]){
       swapElements( i, i/2 )
       i = i/2
  }
}
```