COMP 250

Lecture 27

heaps 1

March 16, 2022

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 (triage)

Assume a set of comparable elements or "keys"

(as with a binary search tree) .

Priority Queue ADT

- add(key)

- 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.

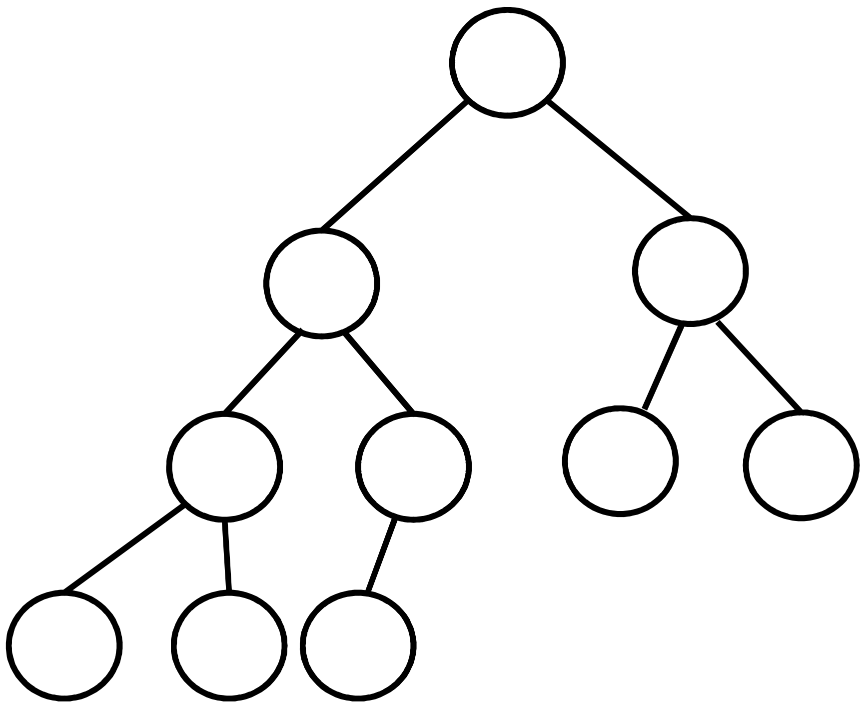
How to implement a Priority Queue ?

- BAD: sorted arraylist or linked list (too slow)

- GOOD: heap (today and next lecture)

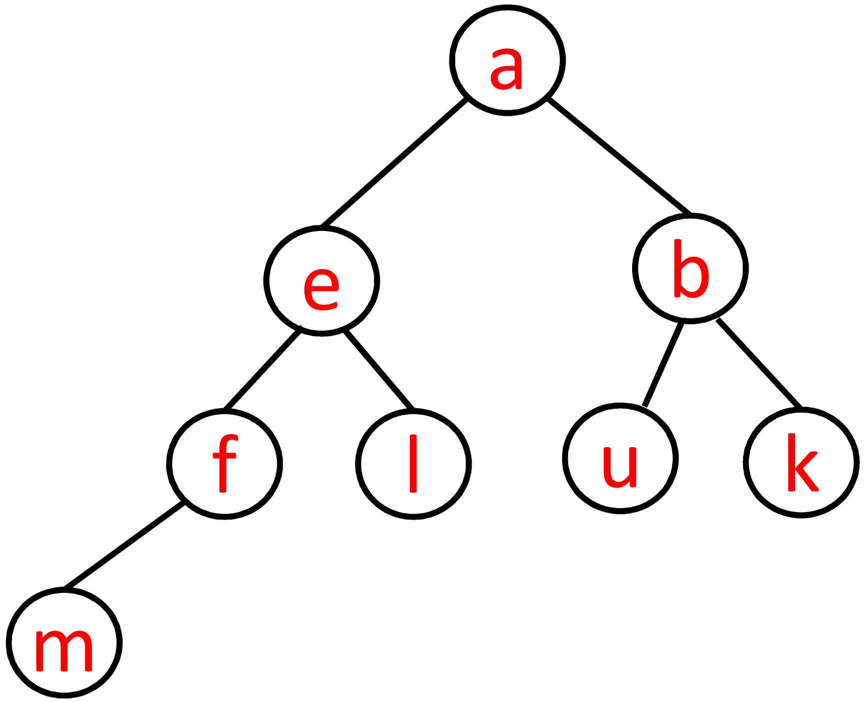
The word “heap” is used in two different ways in computer science. The other way is a “heap” is the part of memory where objects are stored. This is similar the meaning of “heap” used in COMP 206.

Complete Binary Tree (definition)



A *complete binary tree* is a 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

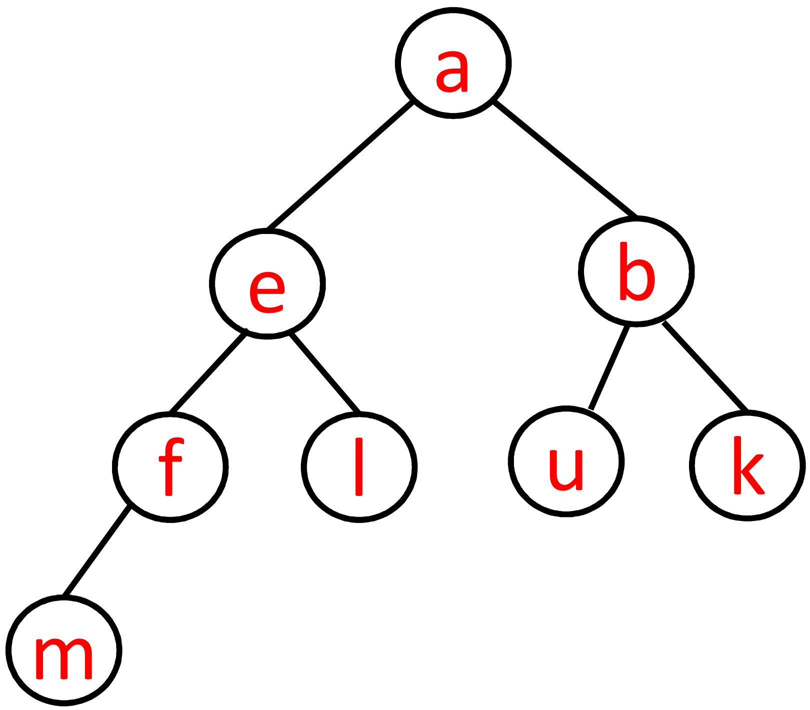
min Heap (definition)



A “min heap” is a complete binary tree with *unique* comparable keys (no duplicates), such that each node’s key is less than its children’s keys. (**NOT a binary search tree !**)

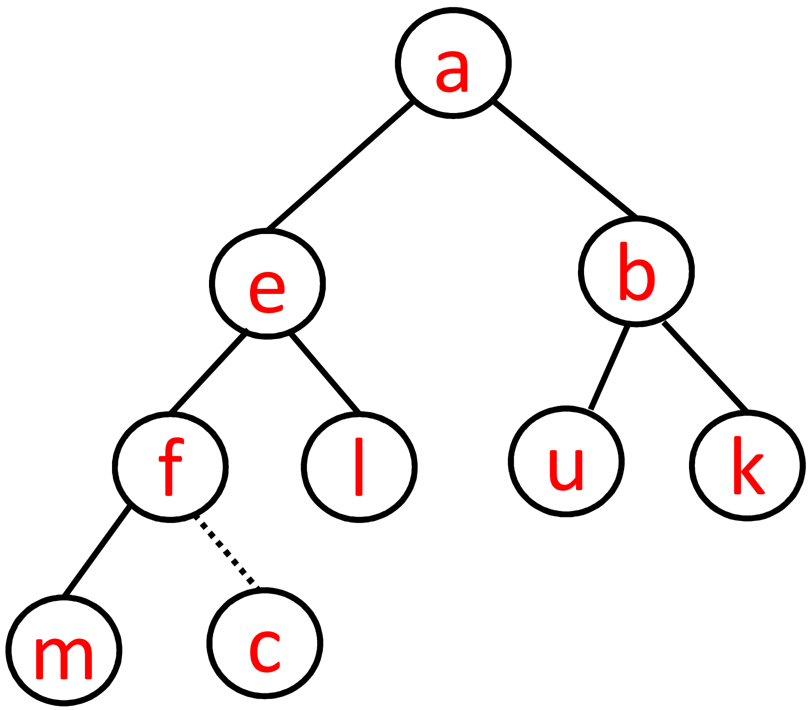
add(key)

e.g. add(c)



add(key)

e.g. add(c)

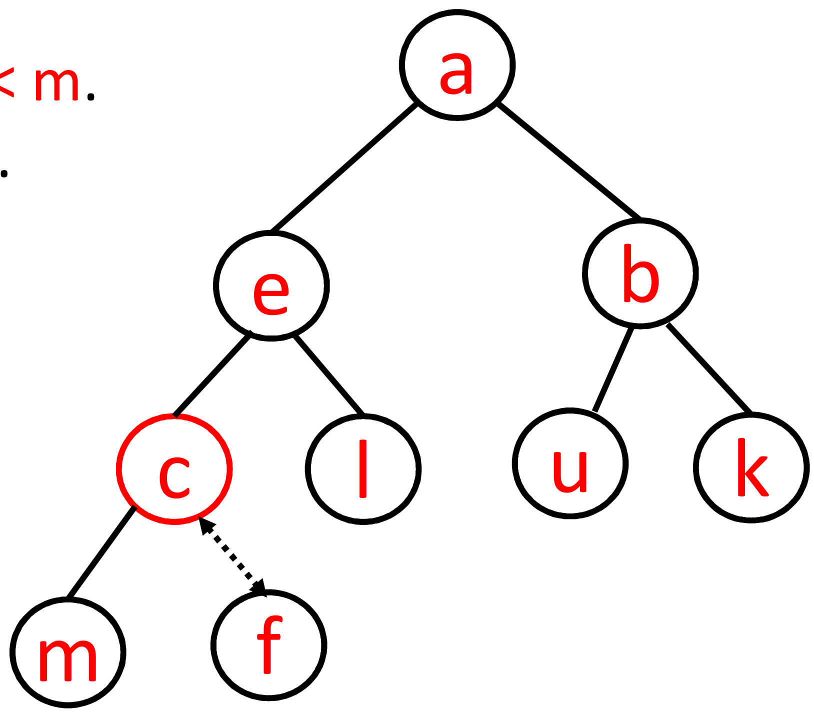


Problem : adding at the next available slot destroys the heap property.

We swap c with its parent f.

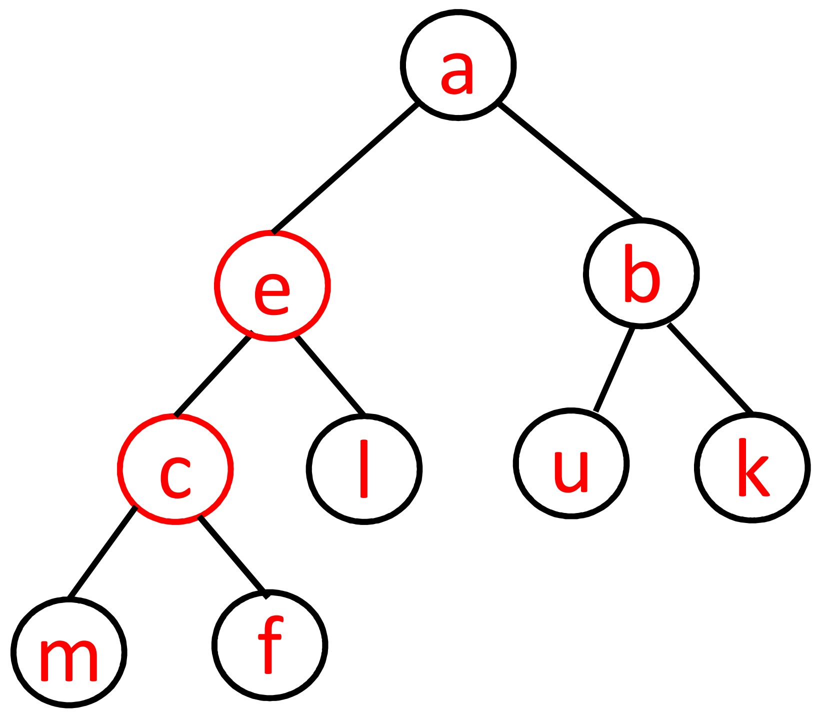
Q: Can this create a problem with c's former sibling, who is now c's child? (It doesn't in this example, but in general ?)

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

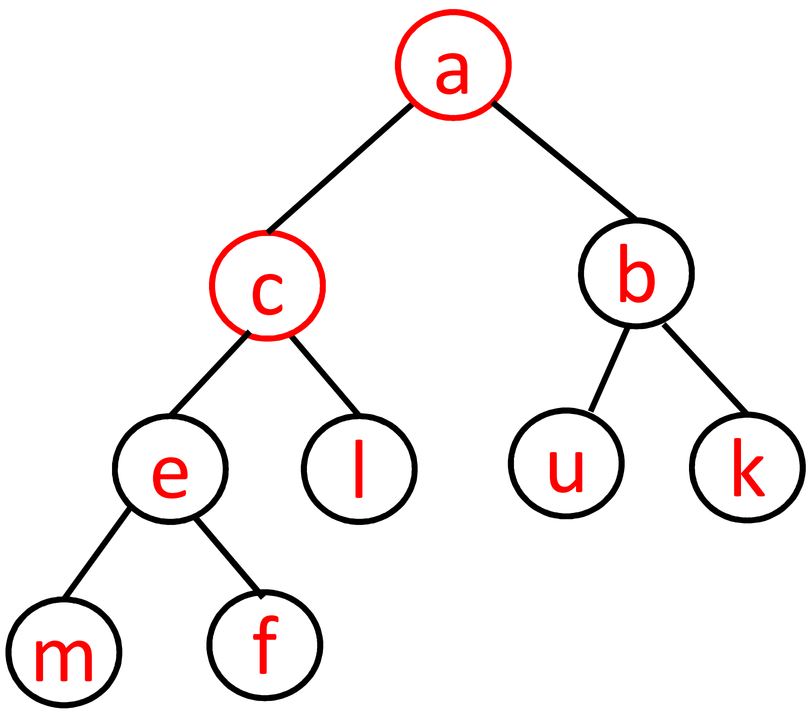


Q: Are we done ?

A: Not necessarily. What about c's parent ? (c < e)



We swap c with its parent e, and now we are done because c is greater than its new parent a .



add(key)

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

}
```

add(key)

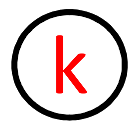
```
add( key ){
    cur = new node at next available leaf position
    cur.key = key
    while (cur != root) and (cur.key < cur.parent.key){

    }
}
```

add(key)

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

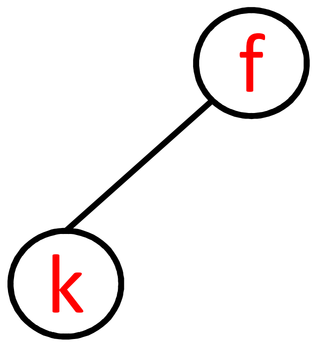
How to build a heap?

add(k)
add(f)   ?



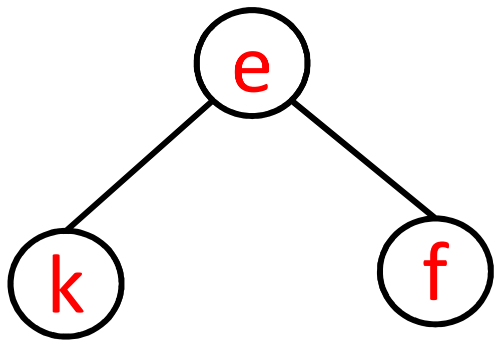
How to build a heap?

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



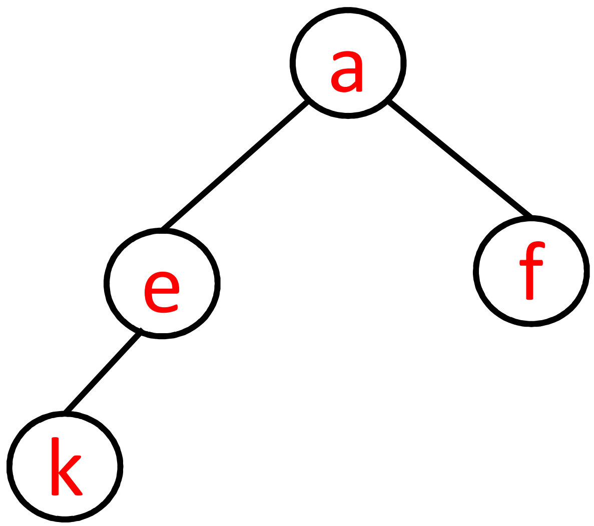
How to build a heap?

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



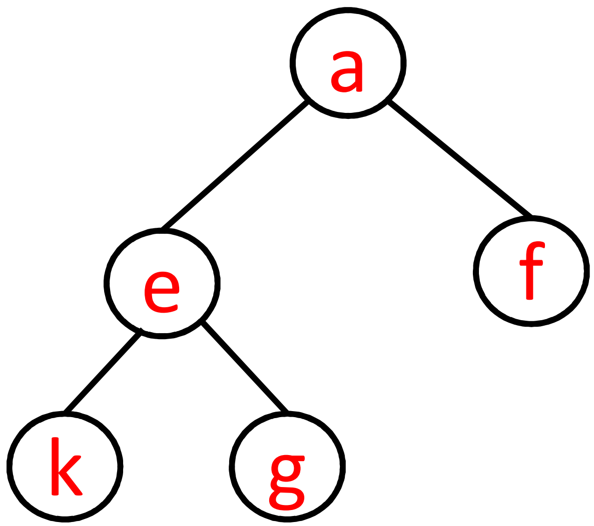
How to build a heap?

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

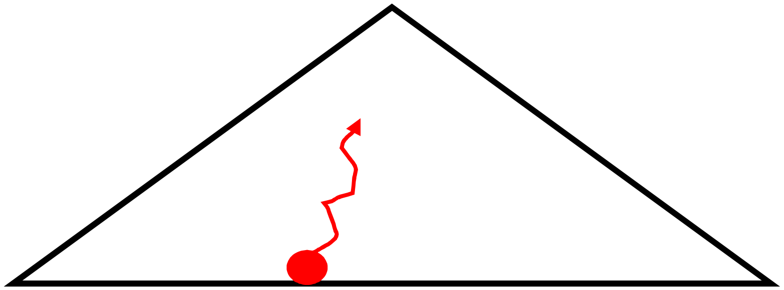


How to build a heap?

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

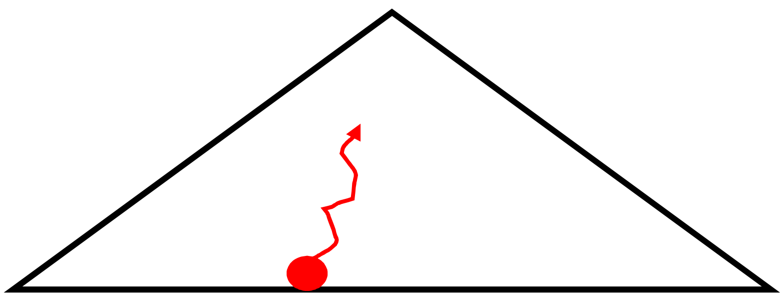


add(key)



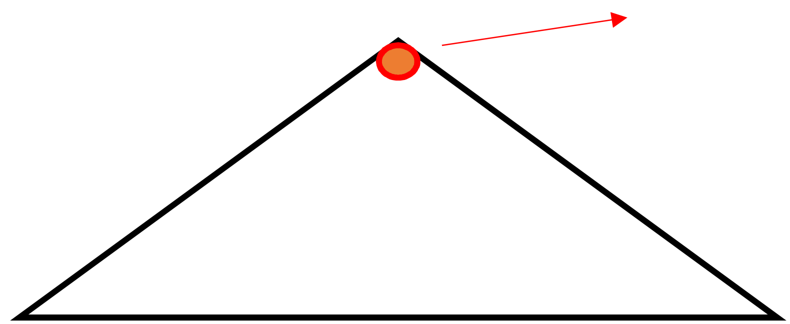
“upHeap”

add(key)



“upHeap”

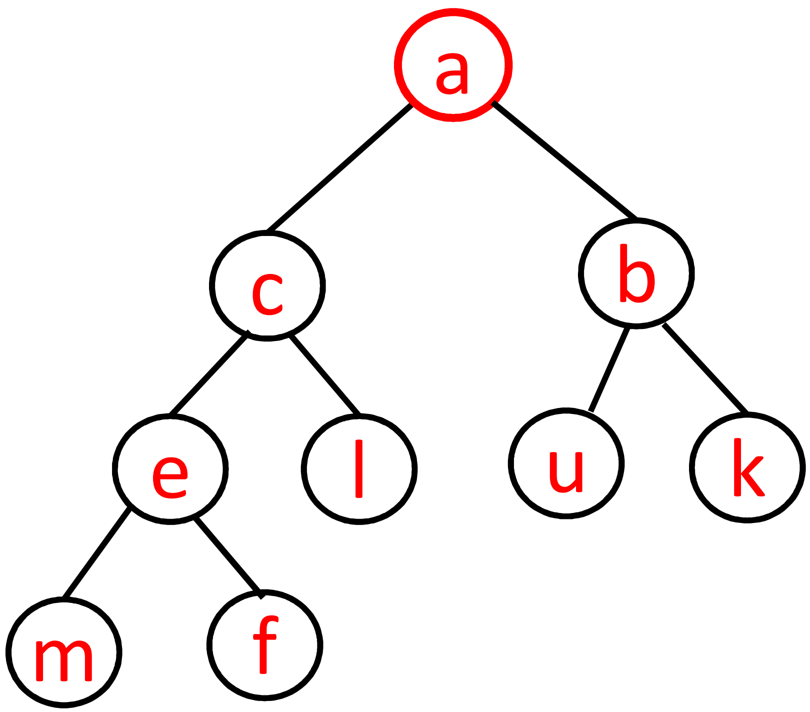
removeMin()



Q: How to do this?

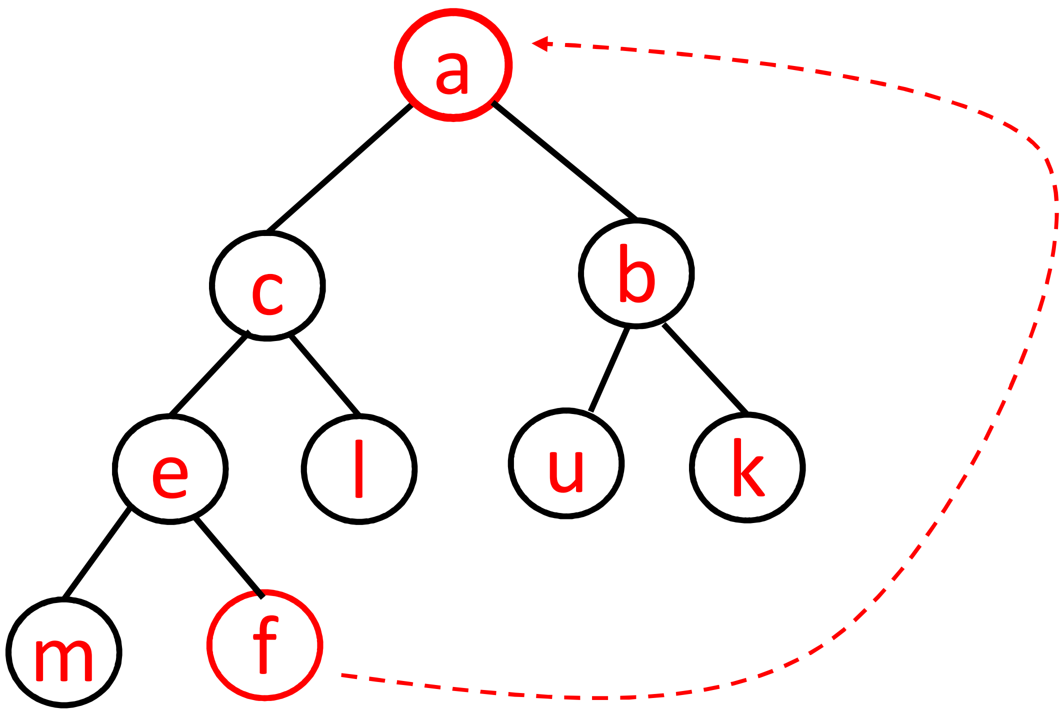
removeMin()

It returns the root key.



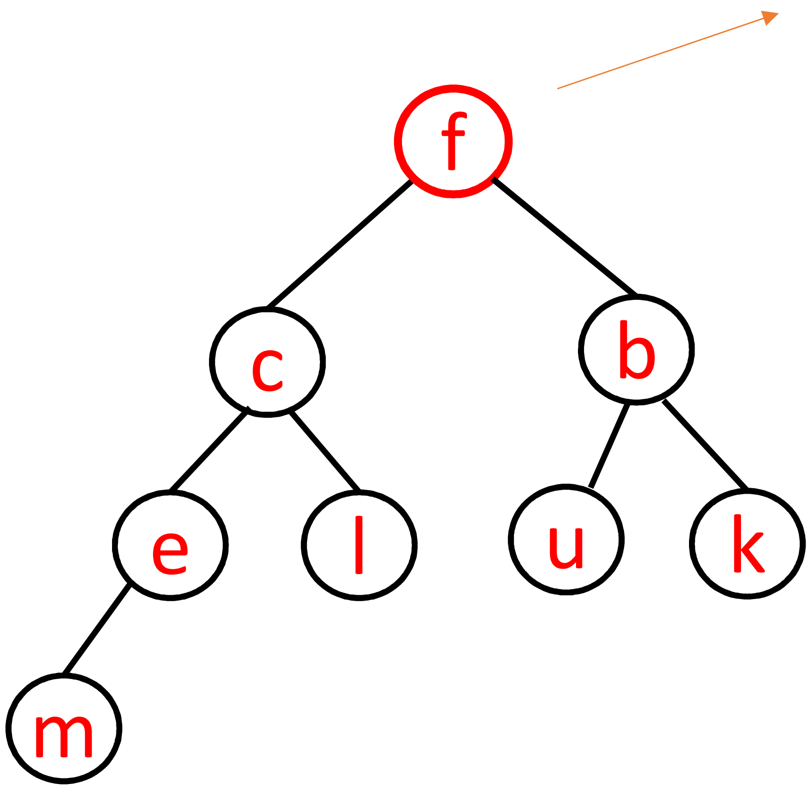
How can we do this?

removeMin()



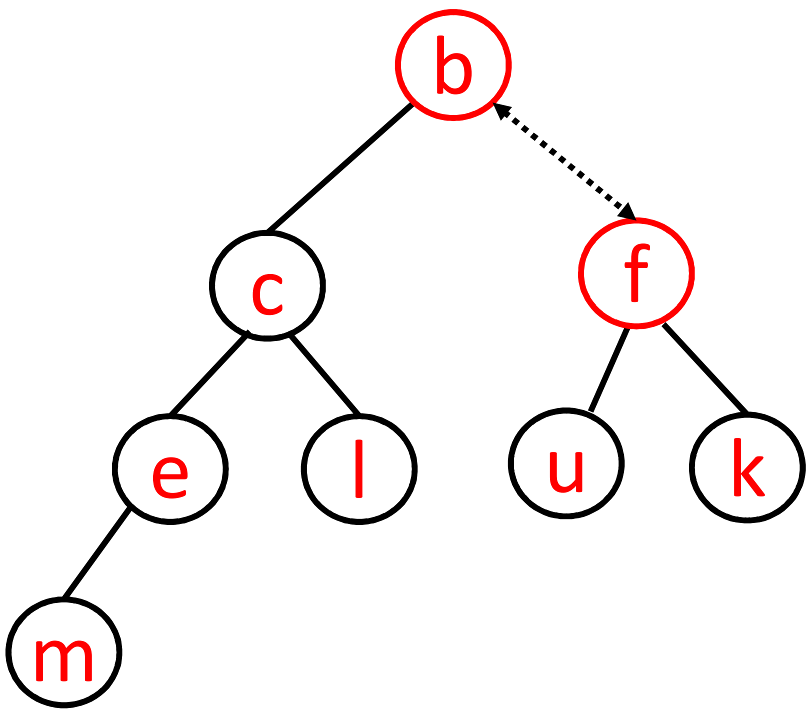
removeMin()

a will be returned



removeMin()

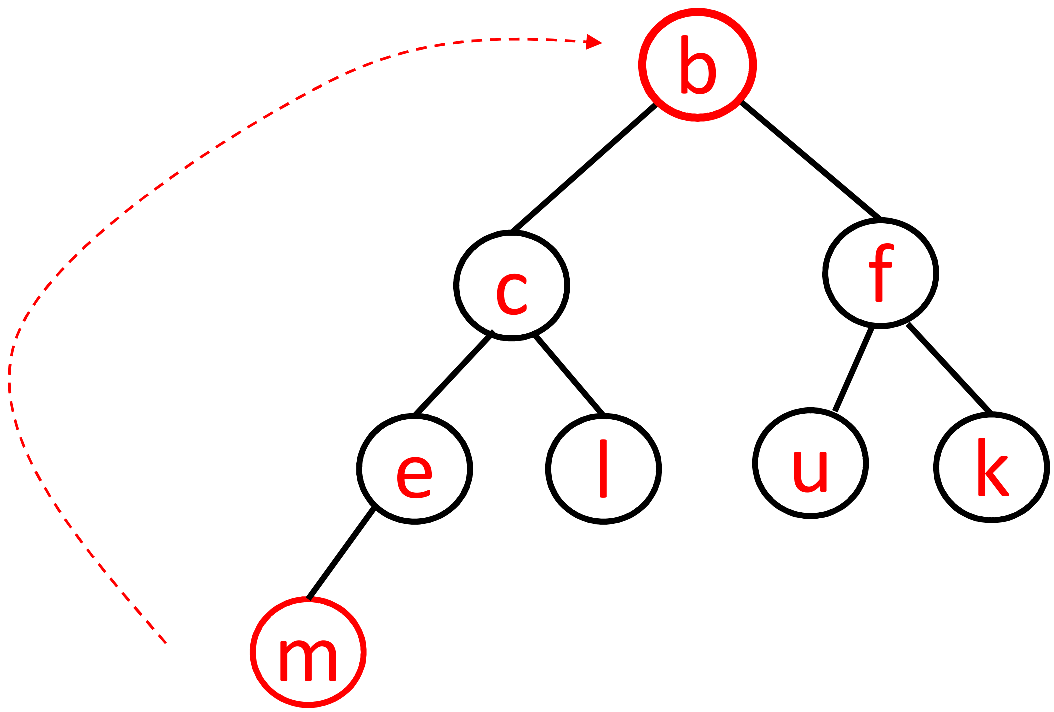
Swap keys with smaller child.



Keep swapping with smaller child, if necessary.

removeMin()

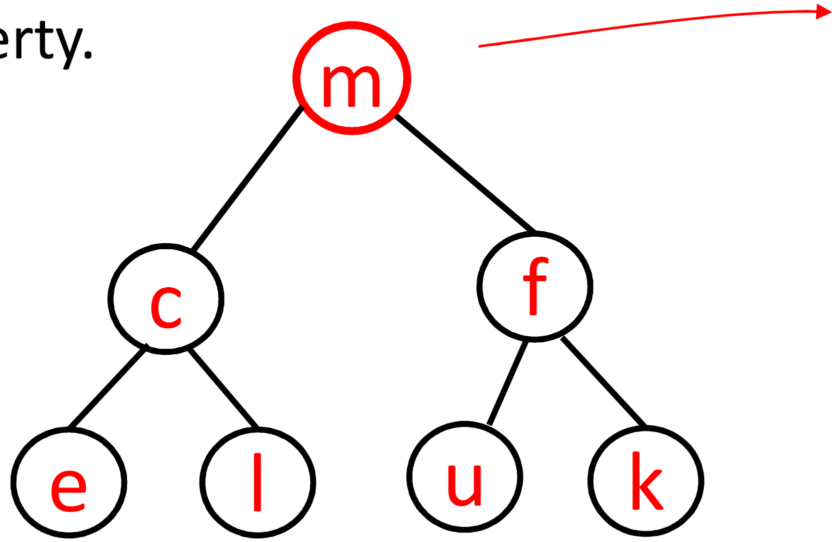
Let’s call removeMin again...



removeMin()

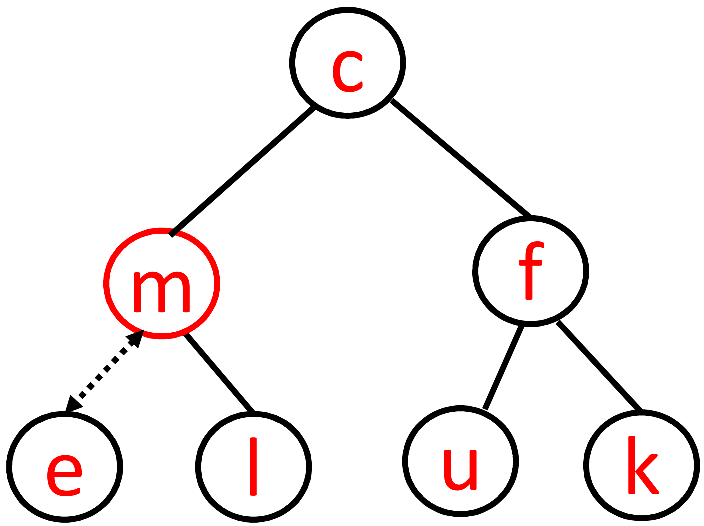
Now swap with smaller child (if necessary) to preserve heap property.

b will be returned

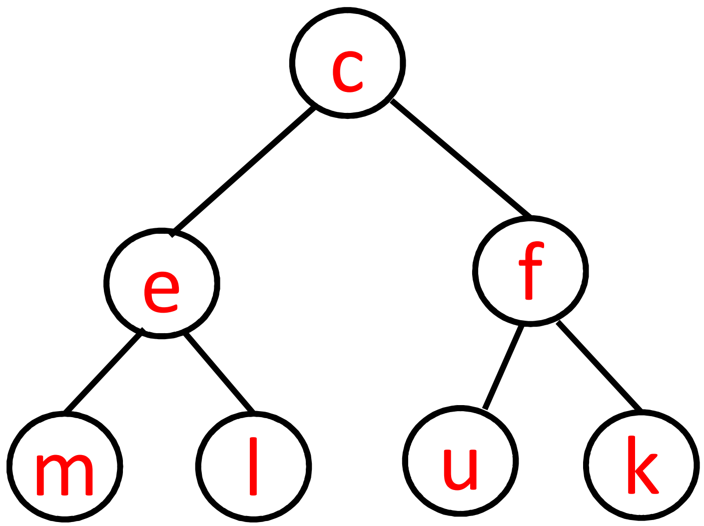


removeMin()

Keep swapping with smaller child, if necessary.



removeMin()



```
removeMin(){
    tmp = root.key

    return tmp
}
```

```
removeMin(){
    tmp = root.key
    remove last leaf node and put its key into the root
    cur = root

        Now adjust the heap if necessary.

    return tmp
}
```

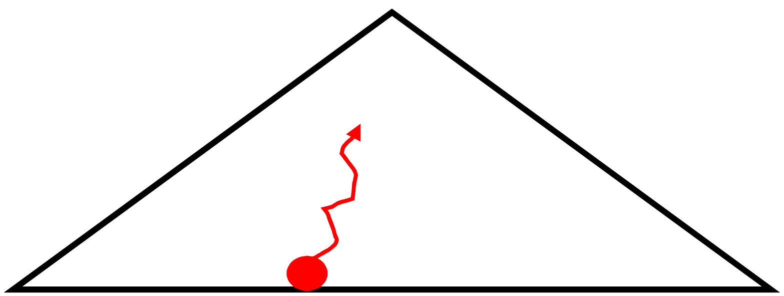
```
removeMin(){
    tmp = root.key
    remove last leaf node and put its key into the root
    cur = root
    while ( (cur has a left child) and (cur.key > cur.left.key)) or
              (cur has right child and (cur.key > cur.right.key) )  )
    {


    }
    return tmp
}
```

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



    }
    return tmp
}
```

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

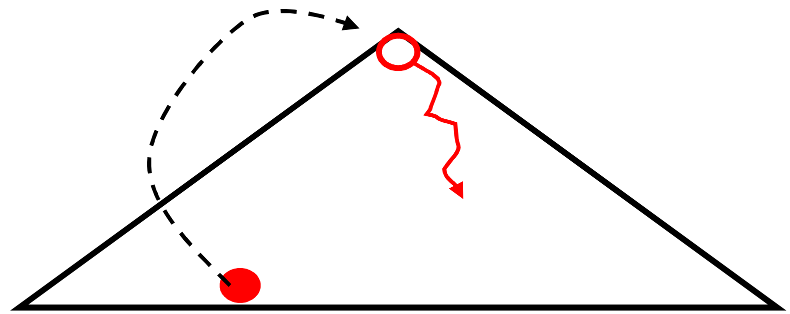
We have just sketched out…

add(key)



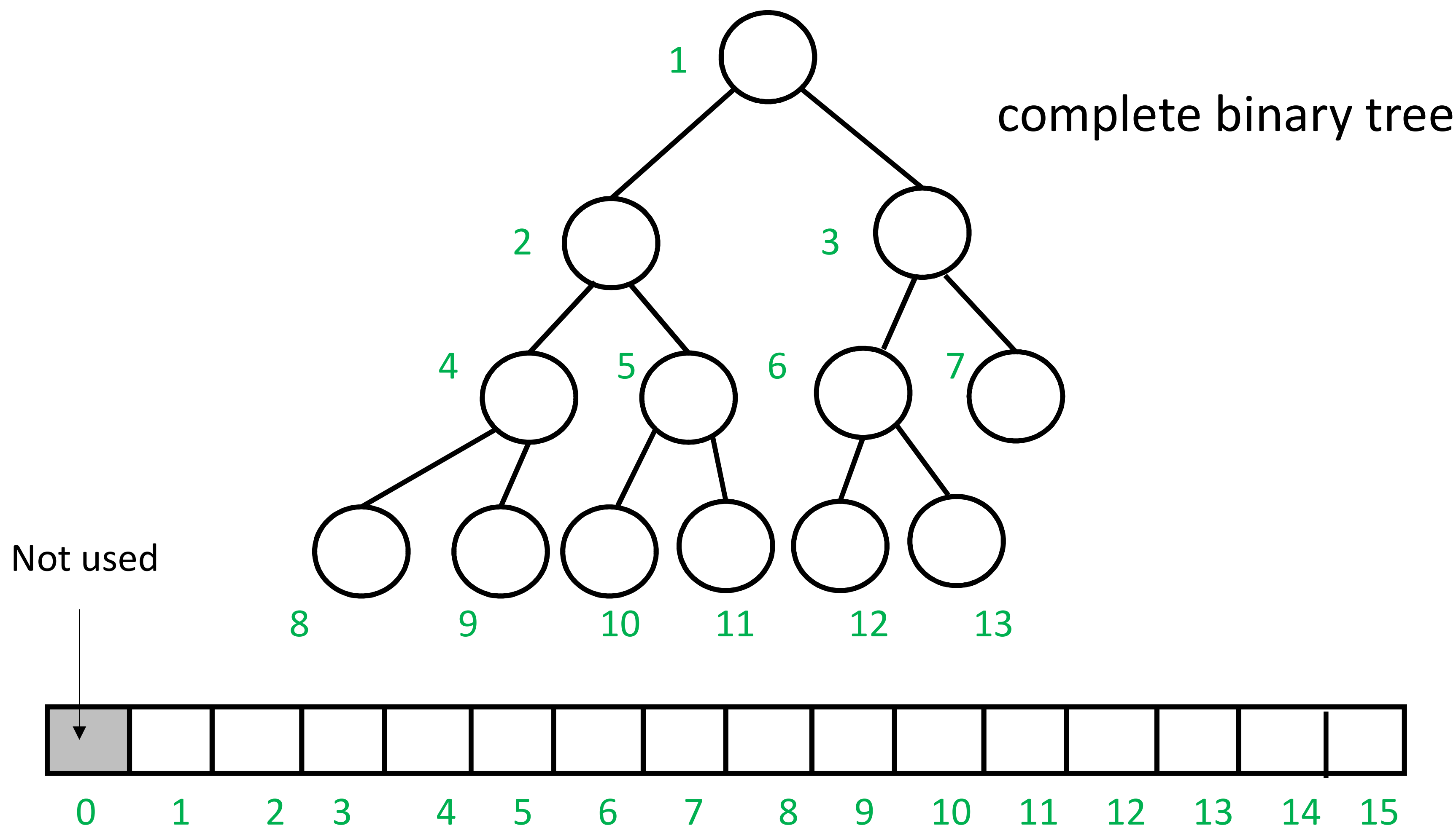
“upHeap”

removeMin()

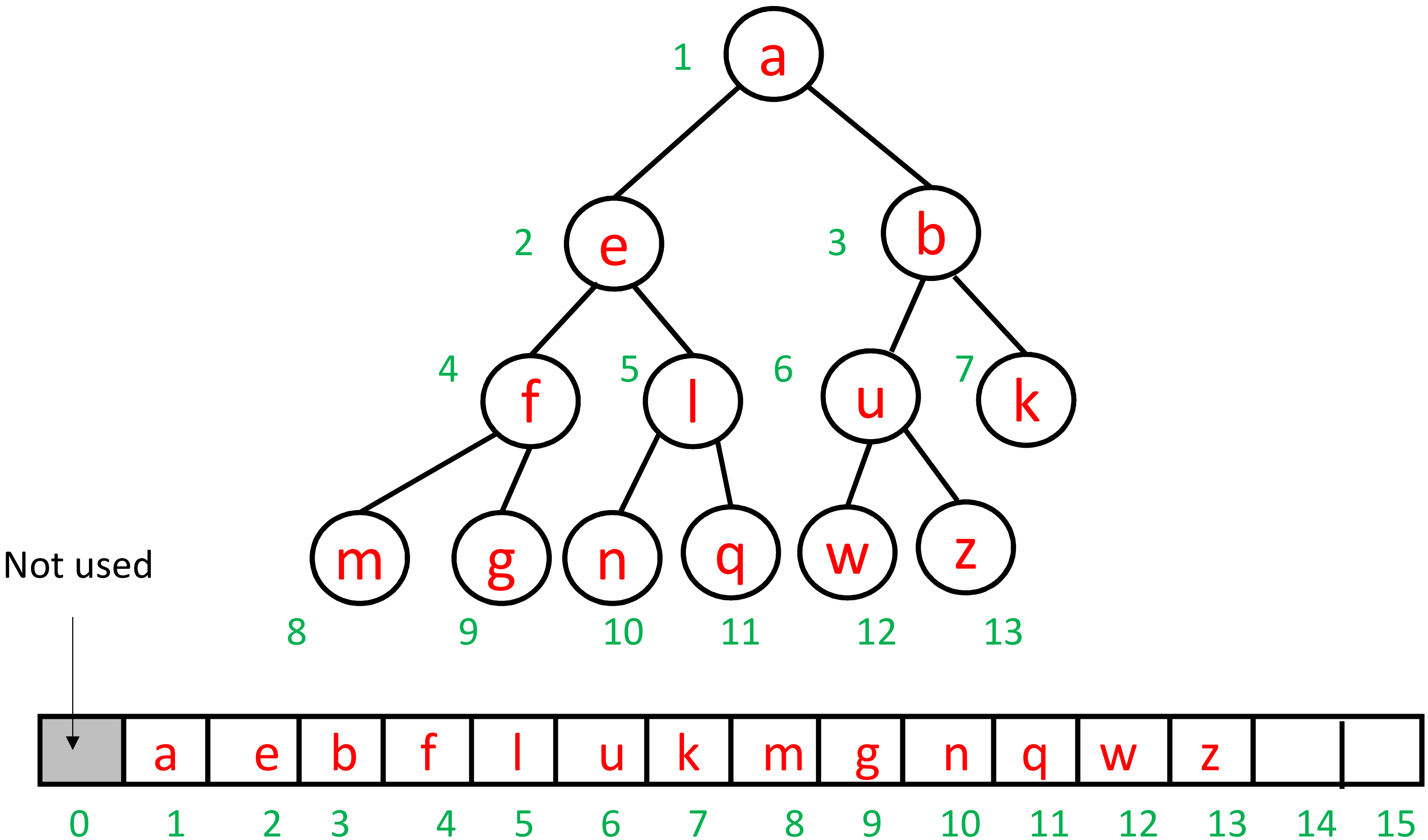


“downHeap”

Heap (array implementation)

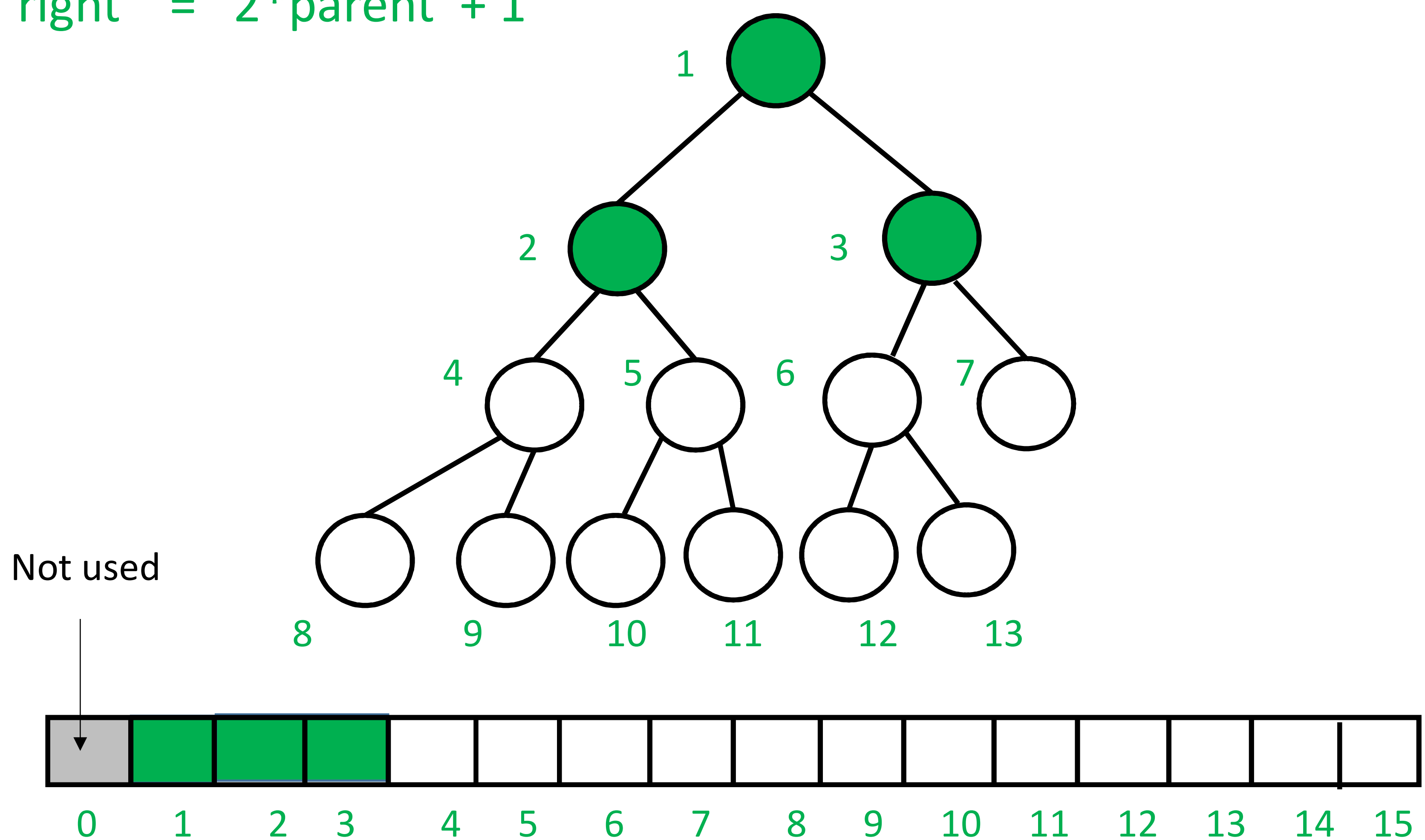


Heap (array implementation)



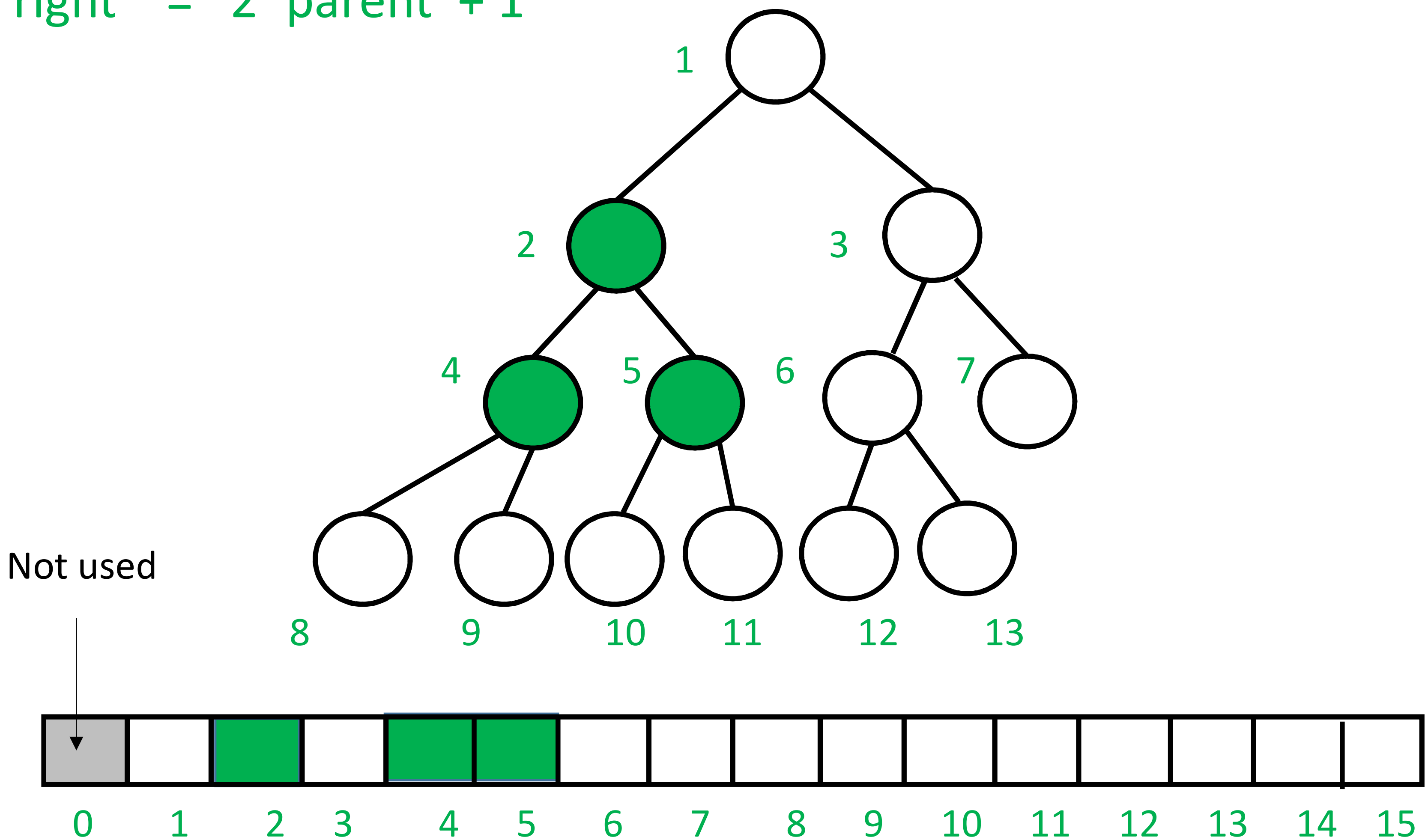
Heap index relations

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



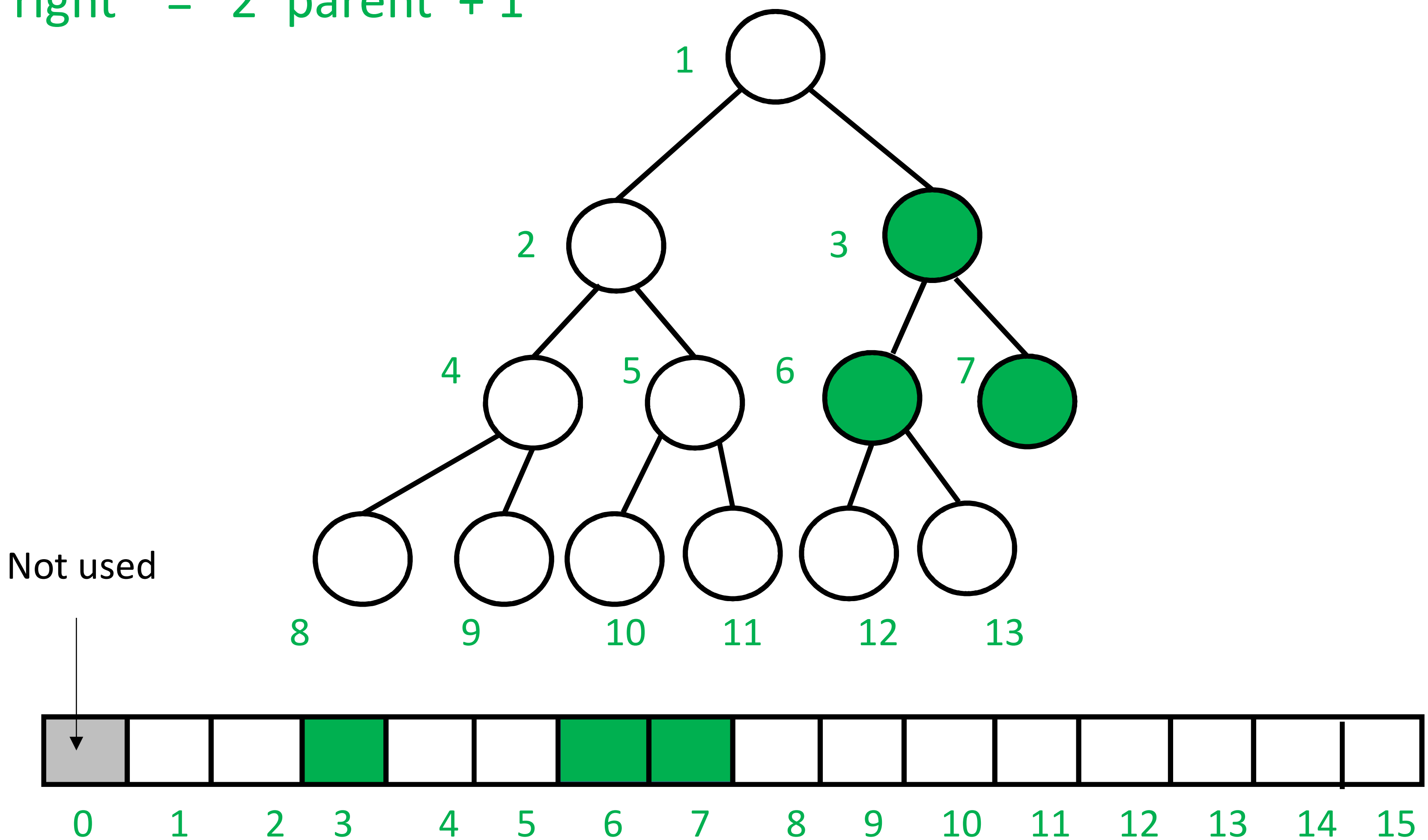
Heap index relations

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



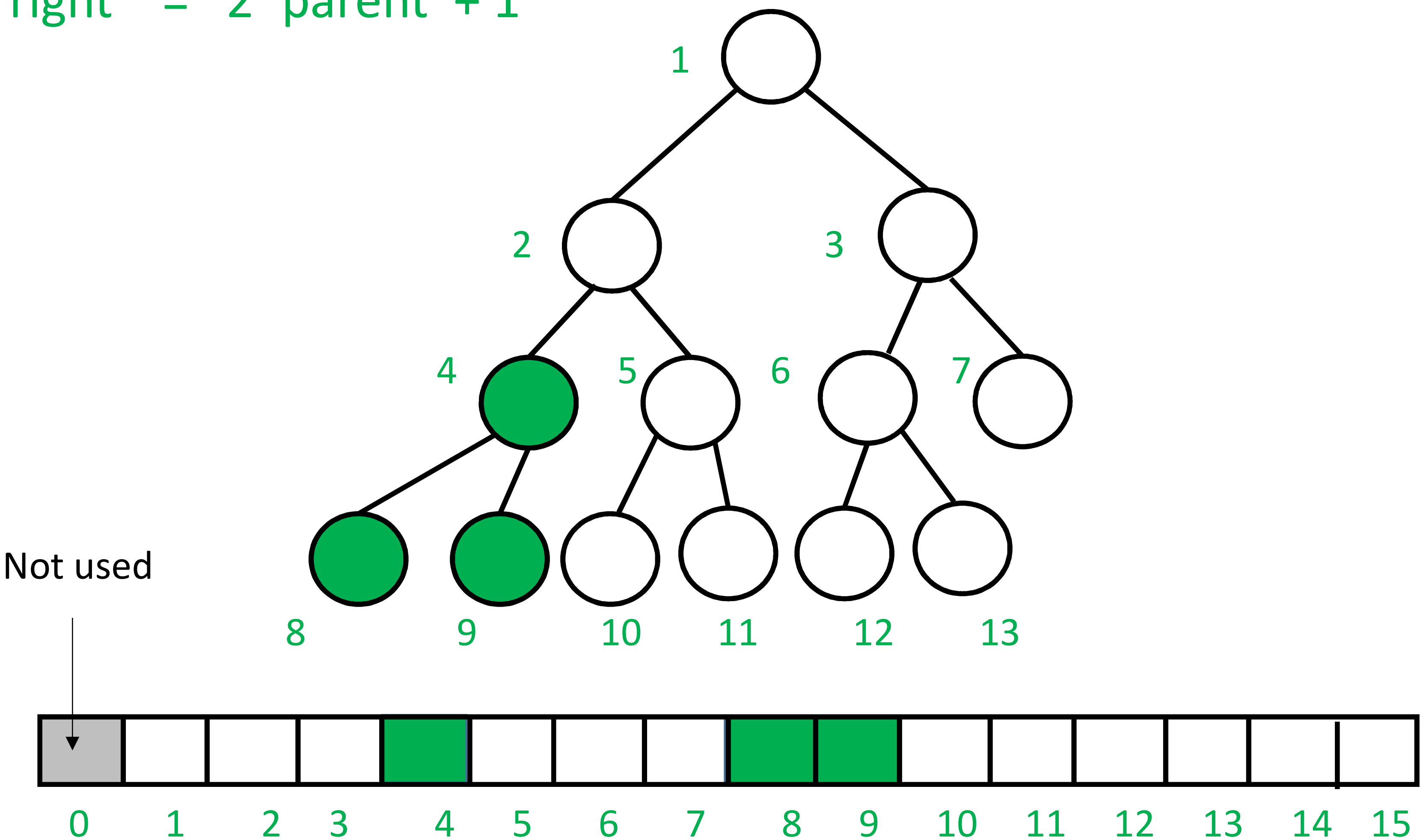
Heap index relations

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

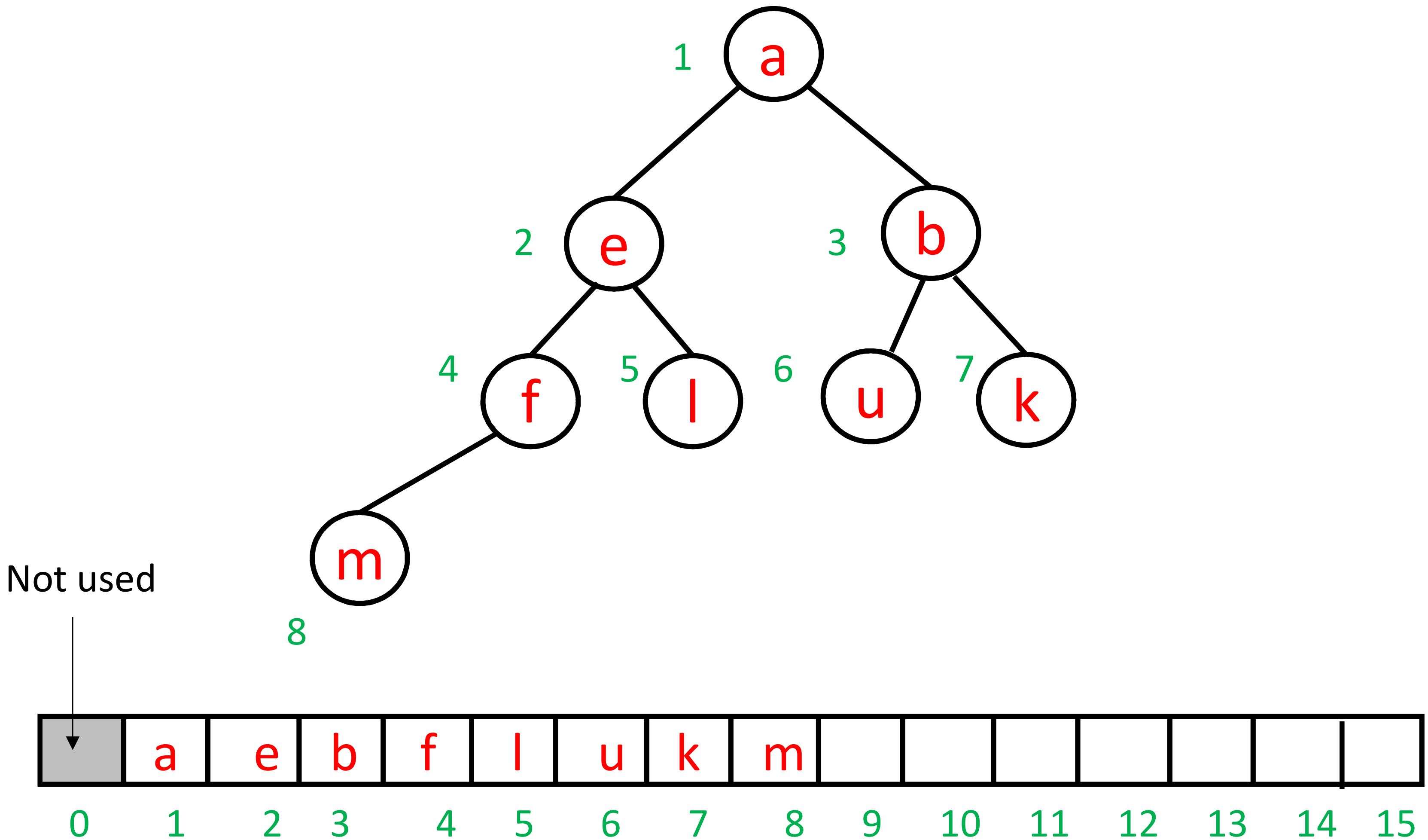


Heap index relations

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

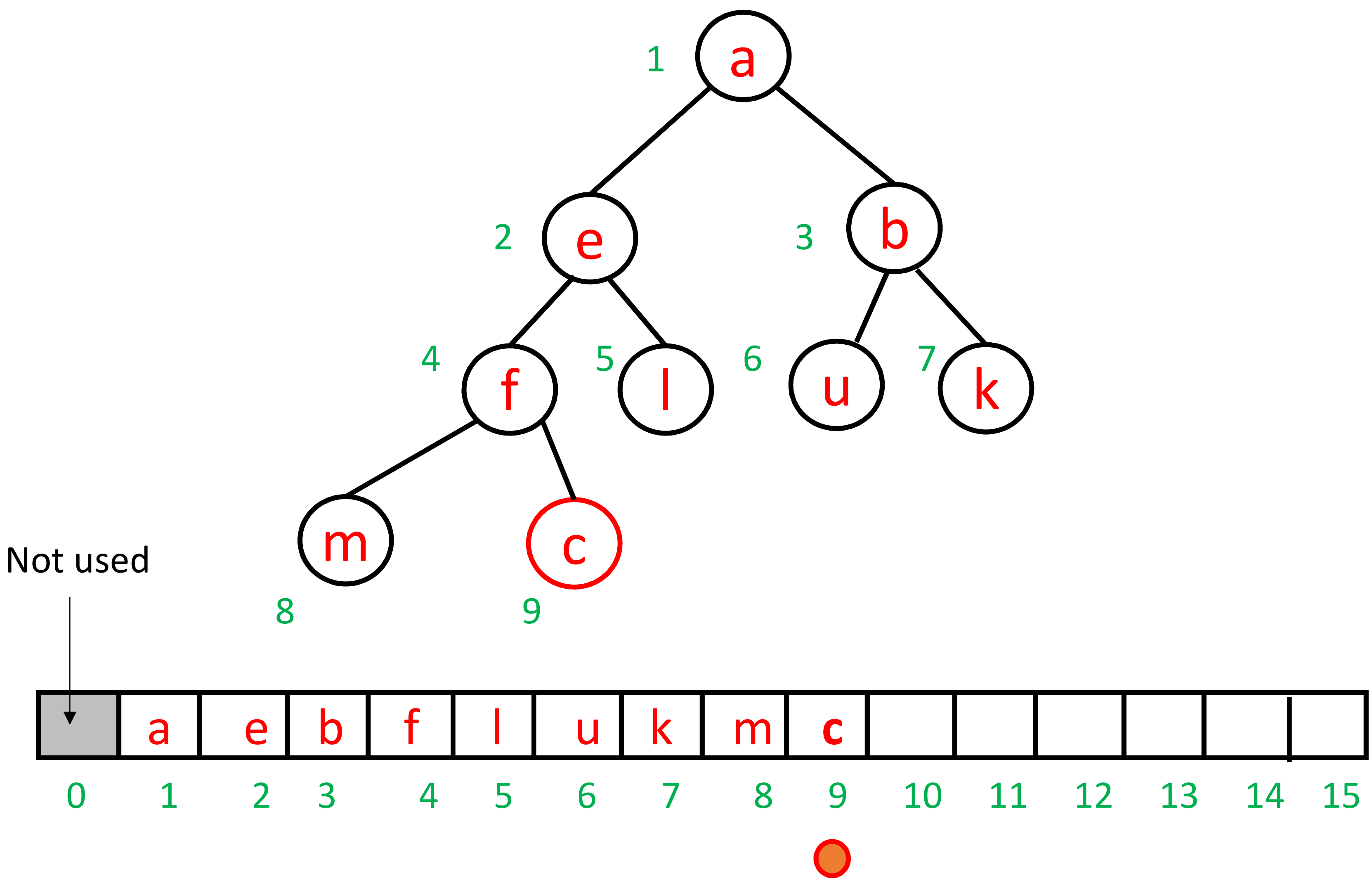


e.g. add(c)



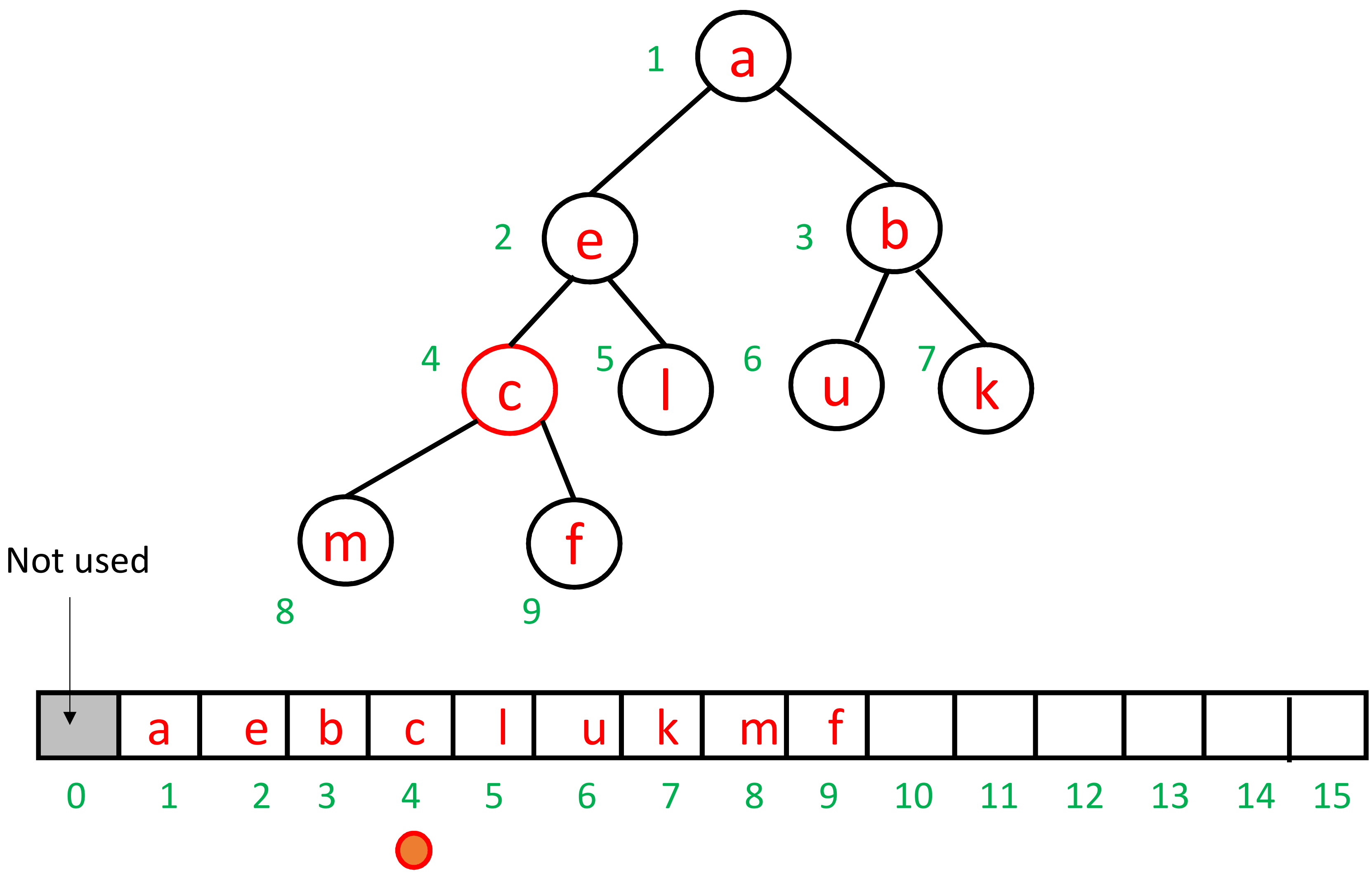
1 a
2 e
3 b
4 f
5 l
6 u
7 k
8 m
Not used
a e b f l u k m
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

e.g. add(c)



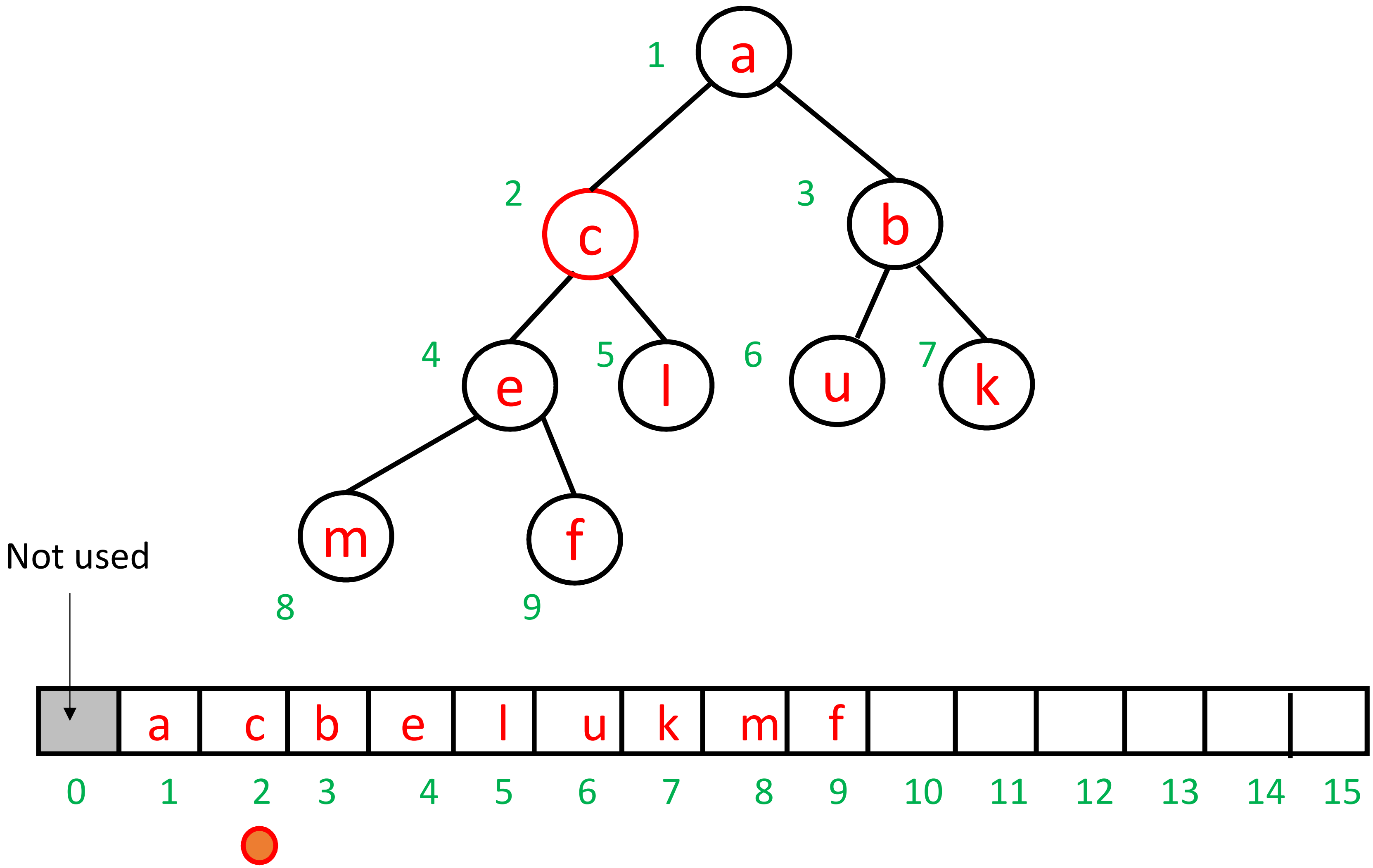
1 a
2 e
3 b
4 f
5 l
6 u
7 k
m
c
8
9
Not used
a e b f l u k m c
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

e.g. add(c)



1 a
2 e
3 b
4 c
5 l
6 u
7 k
8 m
9 f
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(key ){
    size = size + 1          // number of keys in heap
    heap[ size ] = key   // assuming array
                                 //  has room for another key
    i = size

   //  now  "upHeap"

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

Coming up...

Lectures

Fri. March 18 (lecture 28)

Building a heap, Heapsort

Mon. & Wed March 21 & 23

Maps & Hashing

Fri. March 25

Graphs 1

Assessments

Assignment 3

due today

Quiz 4 (lectures 20-25)

Fri. March 18

Assignment 4 posted next week