# COMP 250

# INTRODUCTION TO COMPUTER SCIENCE

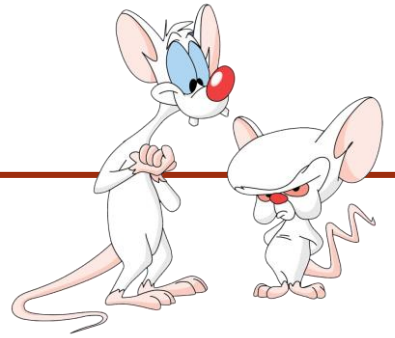


Week 12-2 : Heaps

Giulia Alberini, Fall 2020

Slides adapted from Michael Langer's

# WHAT ARE WE GOING TO DO IN THIS VIDEO?



- Heaps

# HEAPS

# PRIORITY QUEUE

Assume a set of comparable elements or "keys".



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

# PRIORITY QUEUE ADT

- add(key)
- removeMin()

  "highest" priority = "number 1" priority
- peek()
- contains(element)
- remove(element)

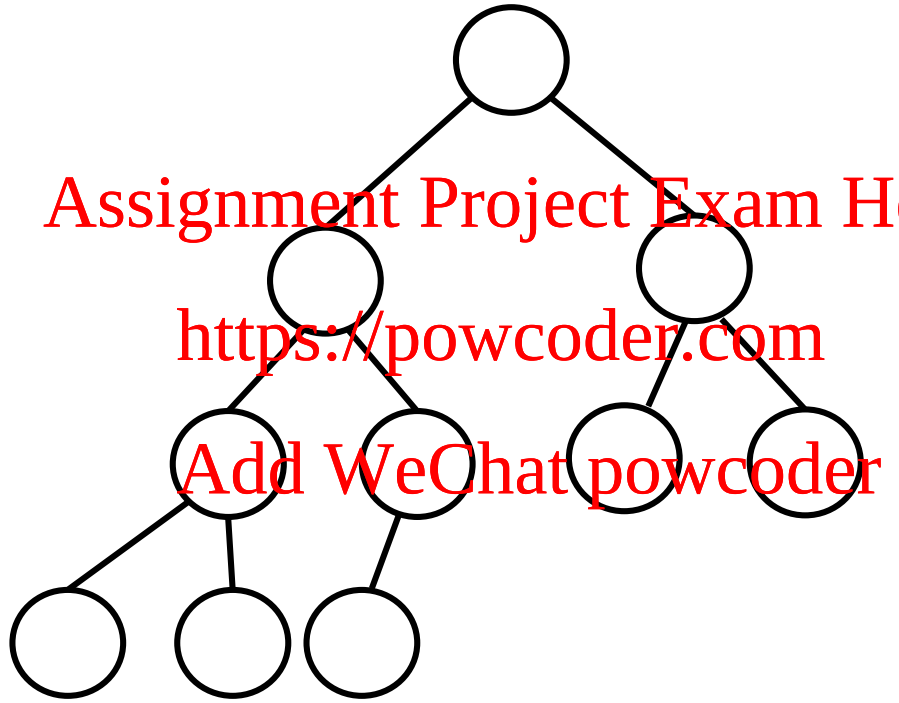
# HOW TO IMPLEMENT A PRIORITY QUEUE ?

- sorted list ?
- binary search tree (last lecture) ?
- balanced binary search tree (COMP 251) ?
- heap (next 2 lectures)

Not the same "heap" you hear about in COMP 206.

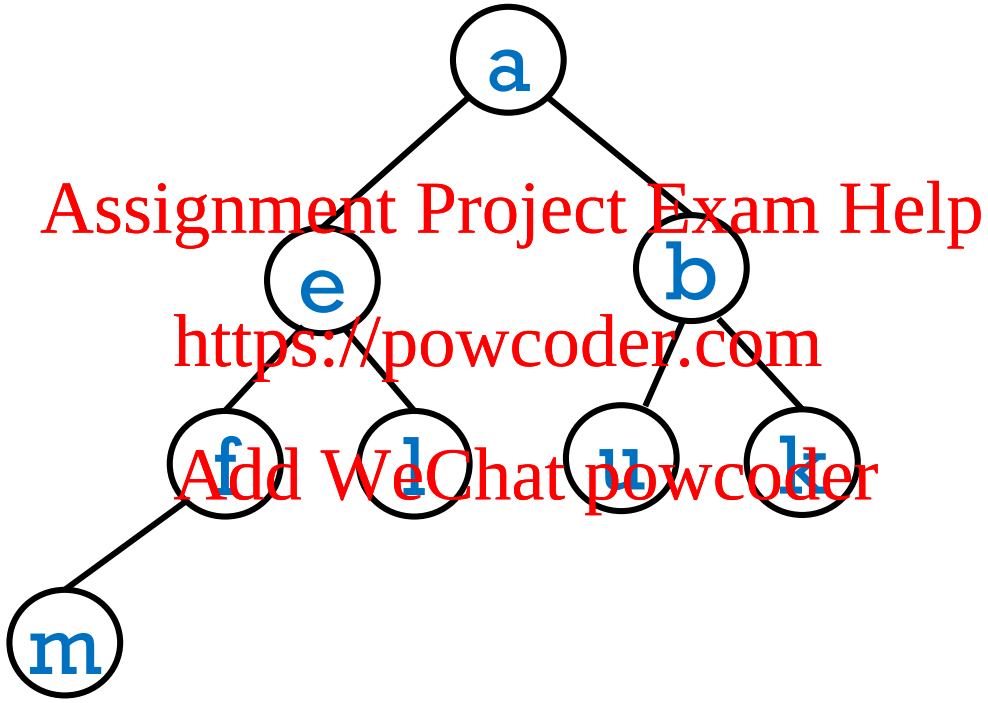
# COMPLETE BINARY TREE (DEFINITION)





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)

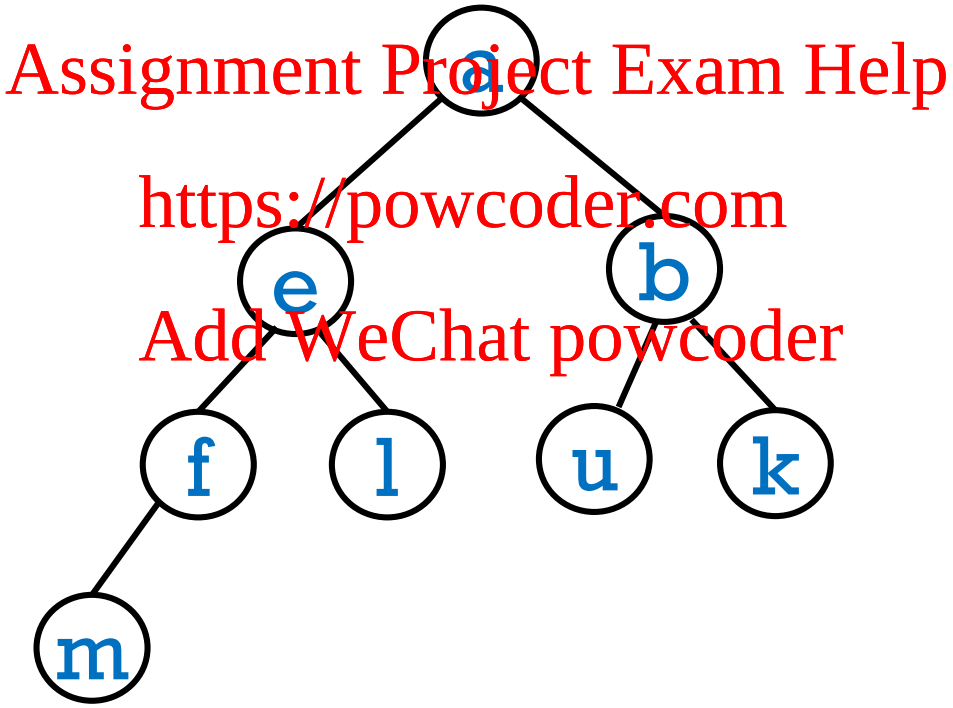





Complete binary tree with unique comparable elements, such that each node's element (key) is less than its children's element (key).

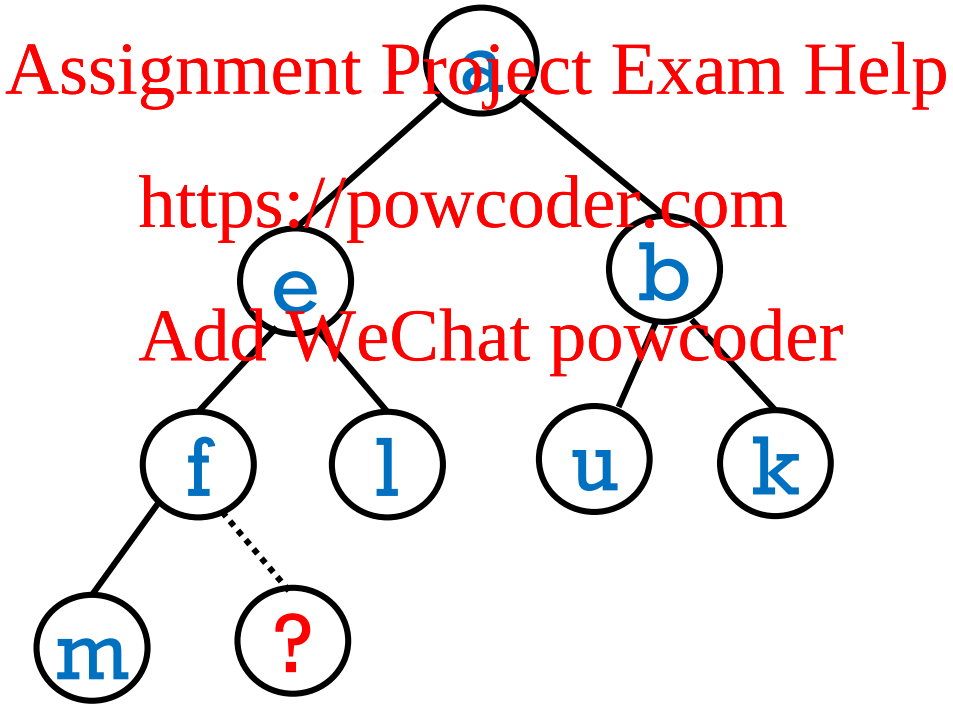
# ADD()

**For example,** `add(c)`



# ADD()

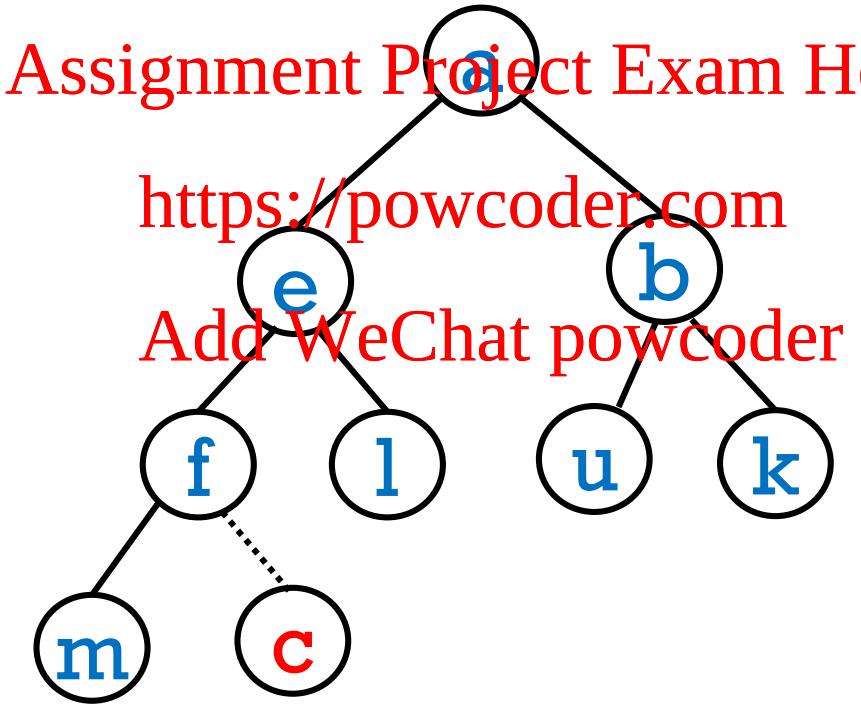
**For example,** `add(c)`



# ADD()

For example, `add(c)`

 What can we do?

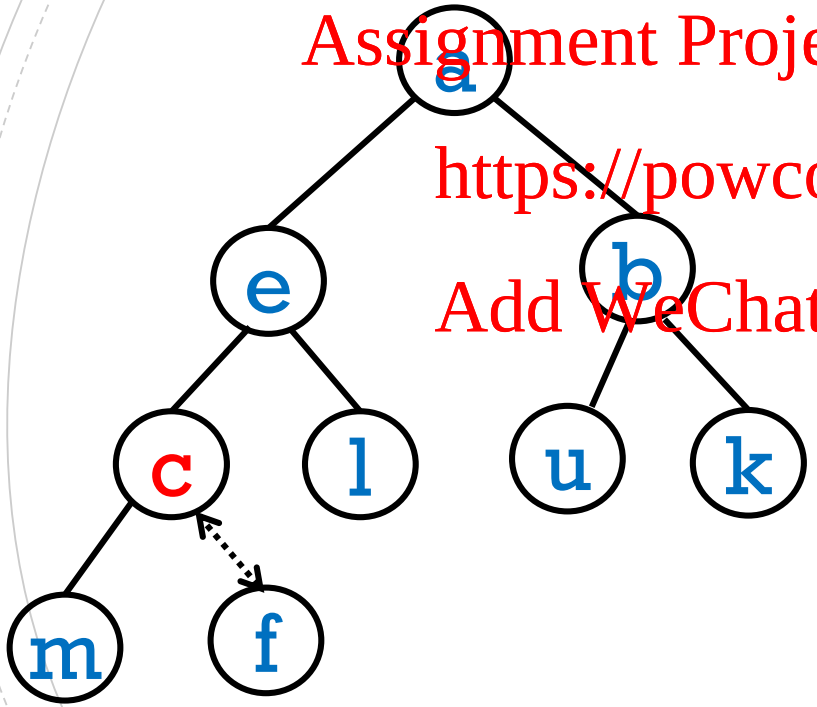





Problem : adding at the next available slot typically destroys the heap property.

# ADD()

**For example,** `add(c)`



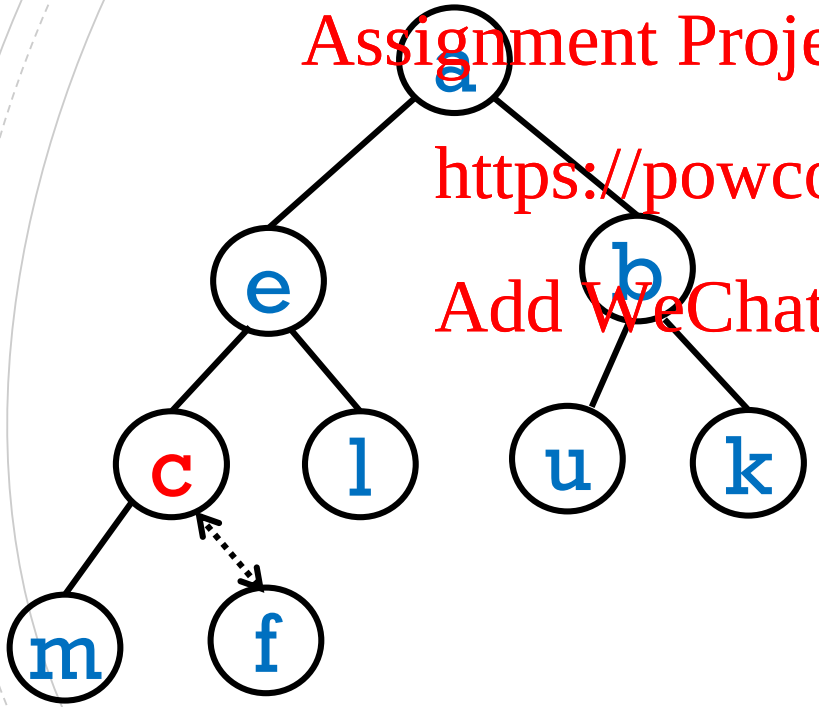

What can we do?

Let's swap c with its parent f.

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

# ADD()

**For example,** `add(c)`






What can we do?

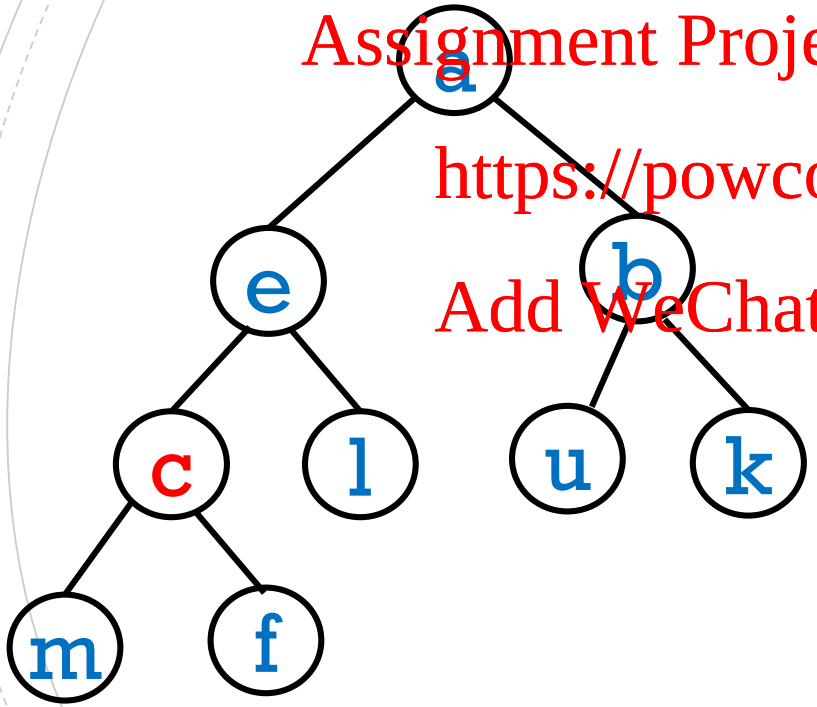
Let's swap **c** with its parent **f**.

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

A: No. Why?

# ADD()

**For example,** `add(c)`

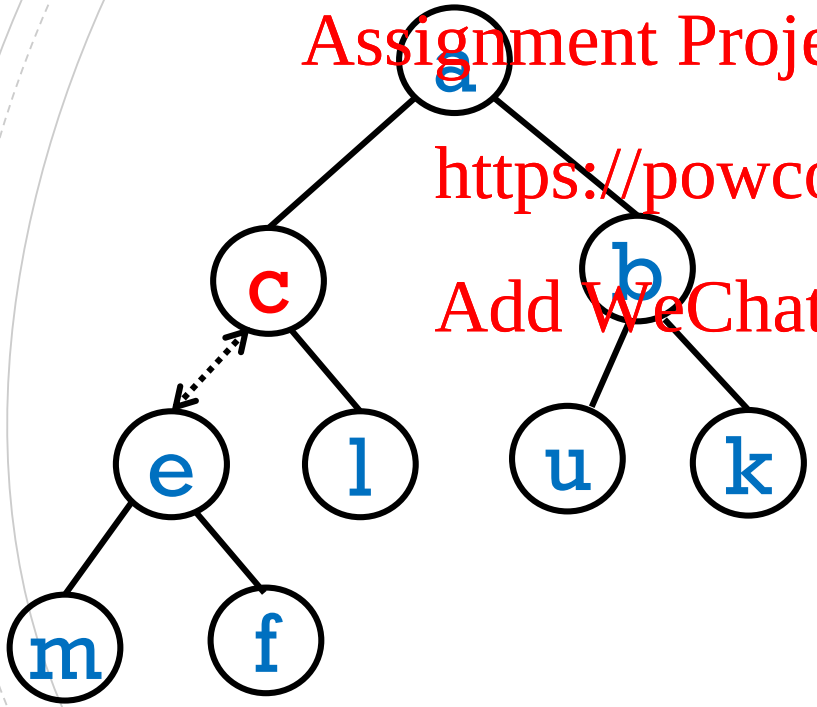



Q: Are we done?

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

# ADD()

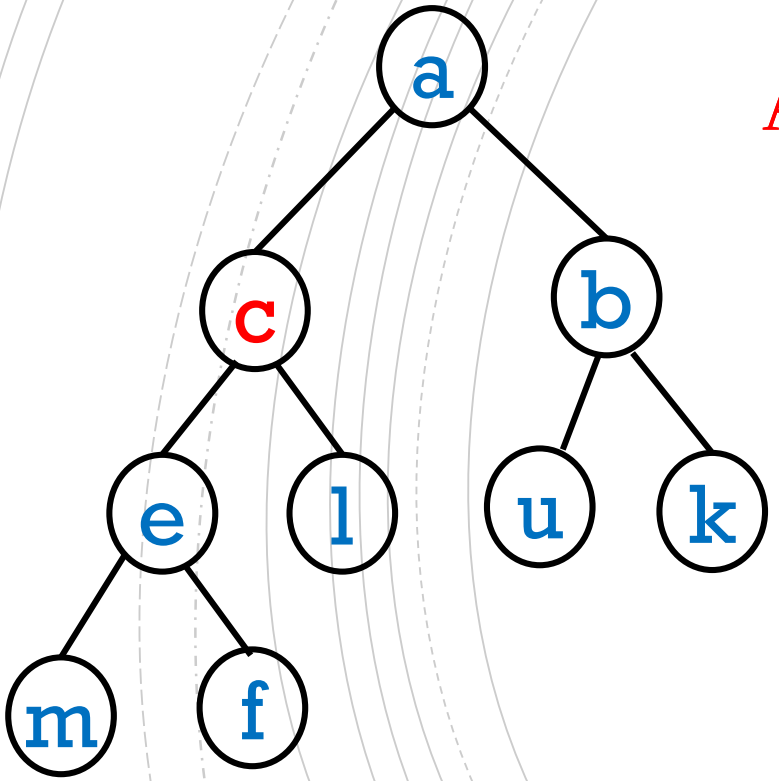
**For example,** `add(c)`




We swap c with its (new) parent e.

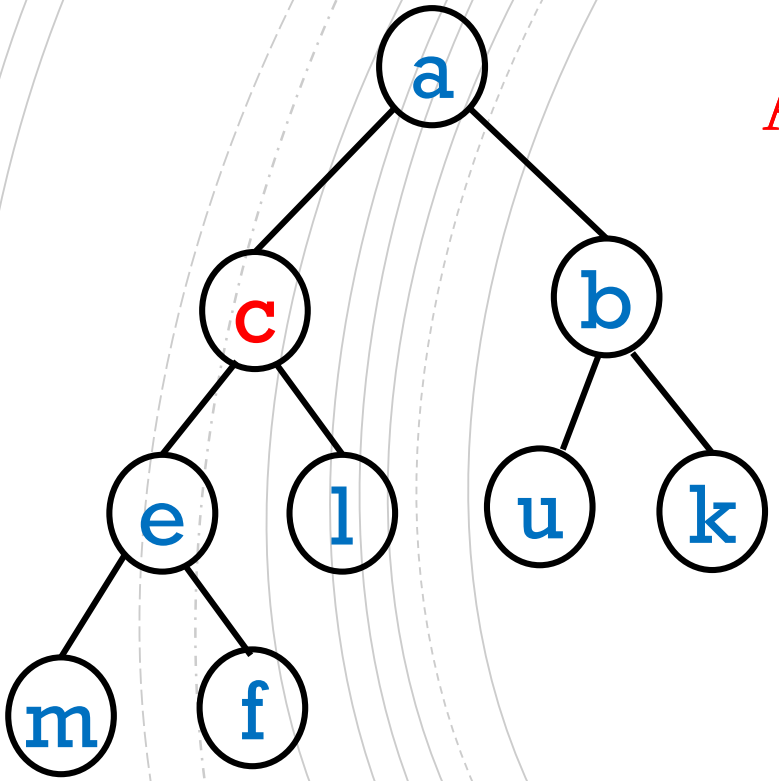
Now we are done because c is greater than its parent a

# ADD() - IMPLEMENTATION



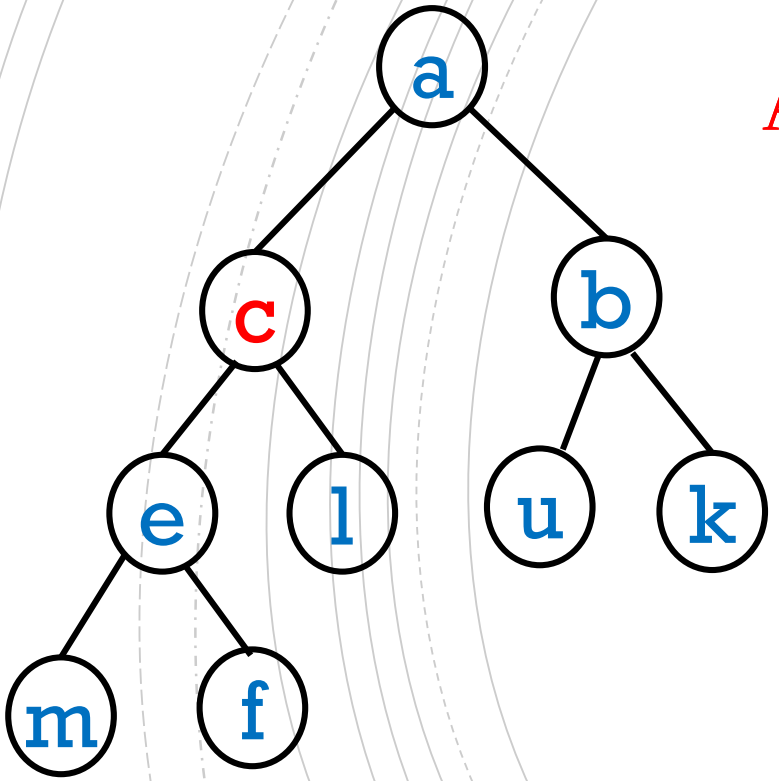

```
add(key) {
    cur = new node at next available leaf position
    cur.key = key

}
```

# ADD() - IMPLEMENTATION




```
add(key){
    cur = new node at next available leaf position
    cur.key = key
    if(root == null) // empty tree
        root = cur

}
```

# ADD() - IMPLEMENTATION




```
add(key){
    cur = new node at next available leaf position
    cur.key = key
    if(root == null) // empty tree
        root = cur
    else {
        while (cur!=root && cur.key<cur.parent.key) {
            swapKeys(cur, cur.parent)
            cur = cur.parent
        }
    }
}
```

# HOW TO BUILD A HEAP?

`add( k )`

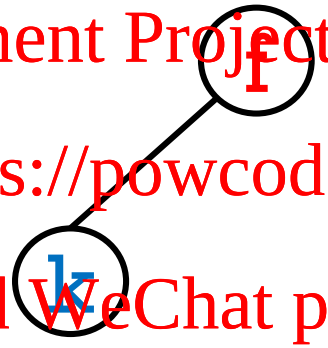
# HOW TO BUILD A HEAP?

```
add( k )
add( f )
```

# HOW TO BUILD A HEAP?

```
add( k )
add( f )
```

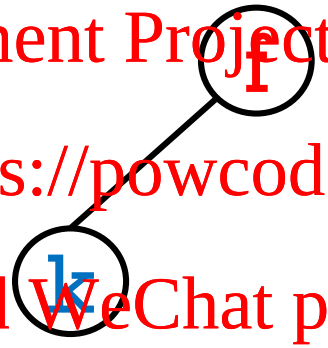




# HOW TO BUILD A HEAP?

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

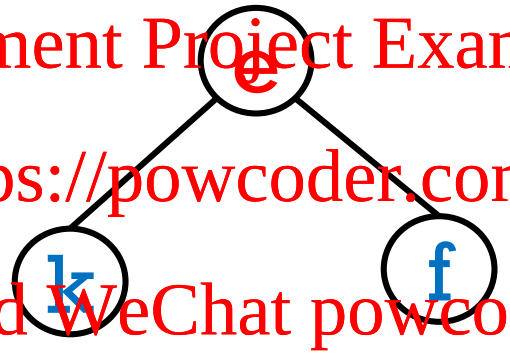




# HOW TO BUILD A HEAP?

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

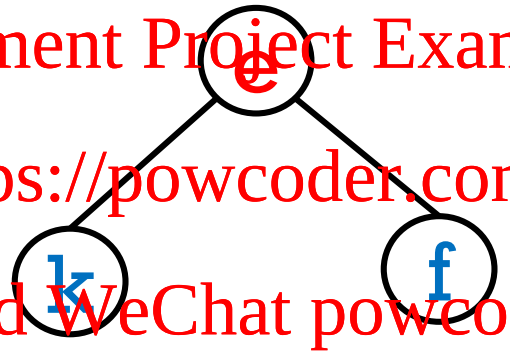




# HOW TO BUILD A HEAP?

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

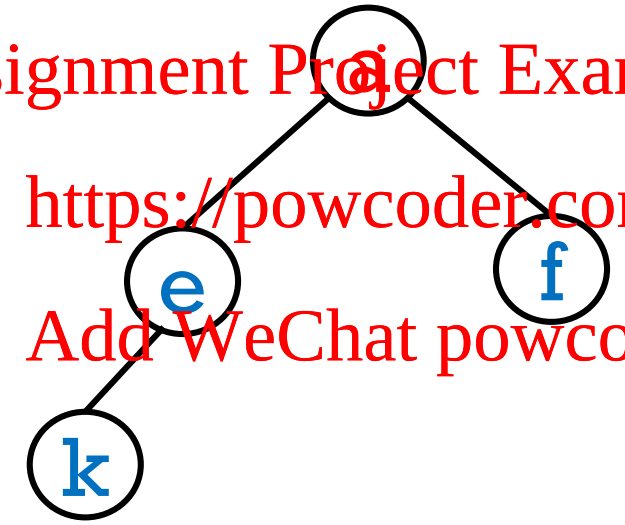




# HOW TO BUILD A HEAP?

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

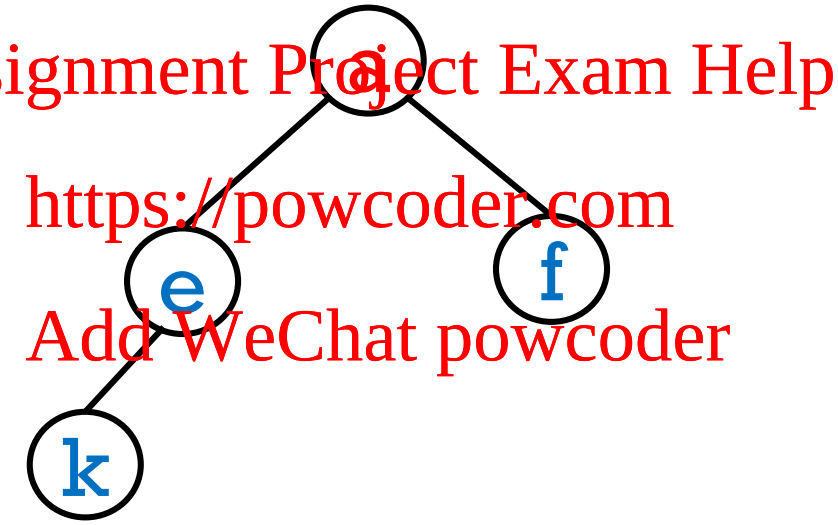




# HOW TO BUILD A HEAP?

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

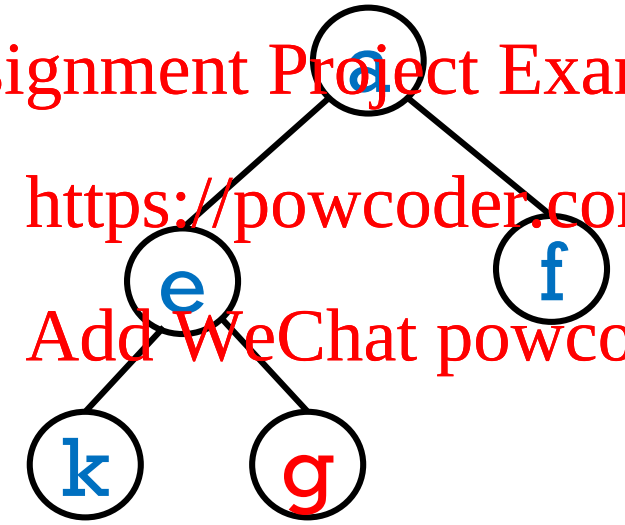




# HOW TO BUILD A HEAP?

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





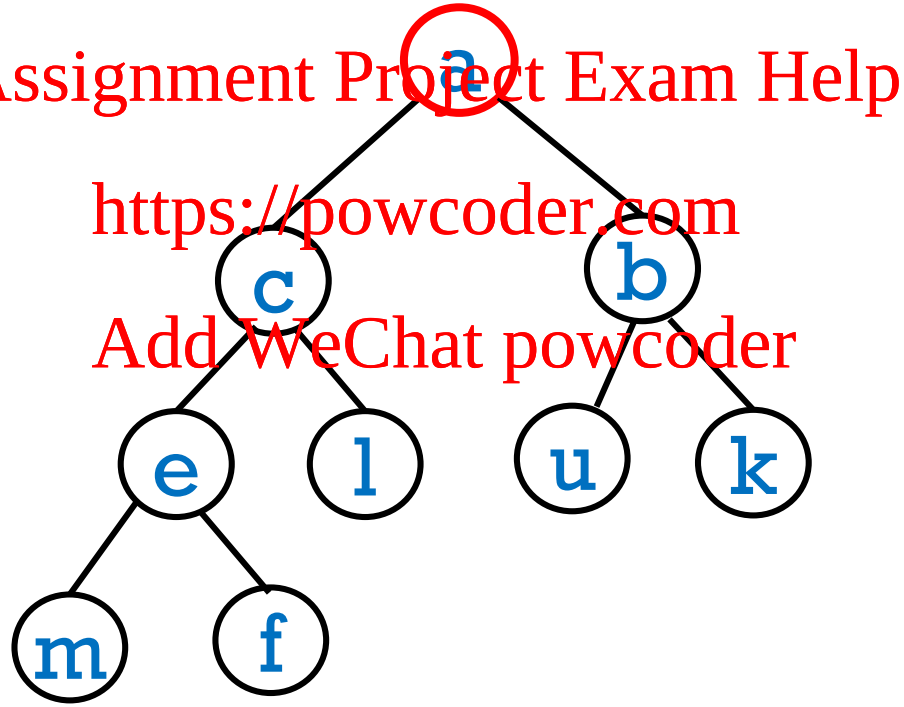



This method of building a heap is slow.

We will see a faster method next video.

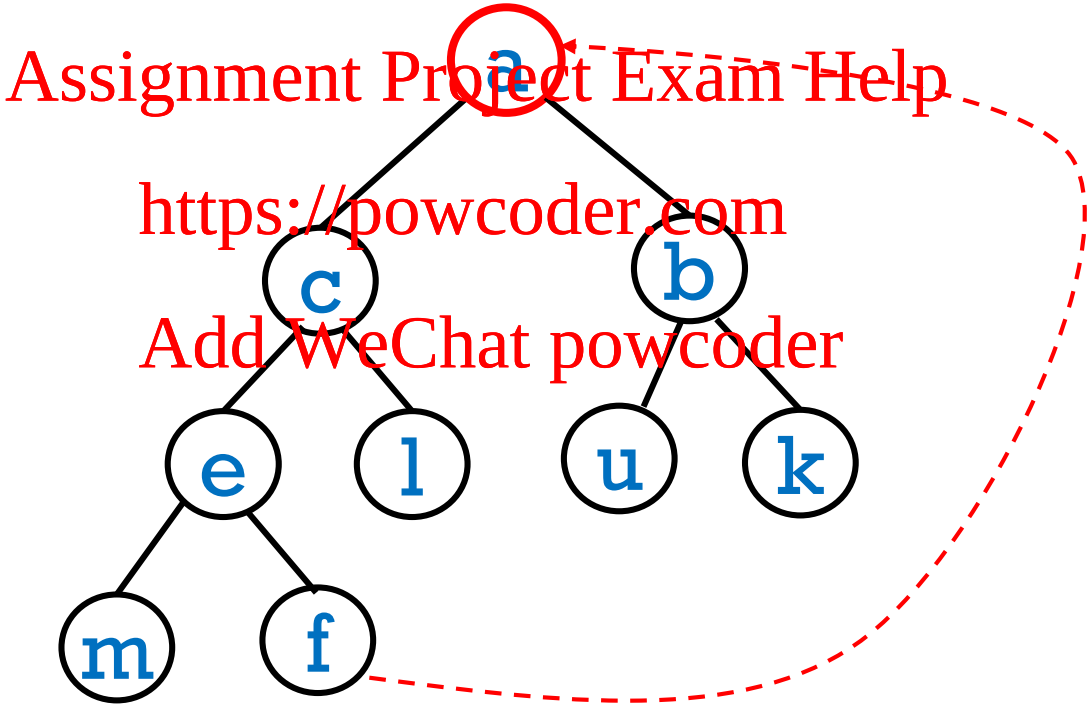
# REMOVEMIN()

returns root element



# REMOVEMIN()

returns root element



# REMOVEMIN()

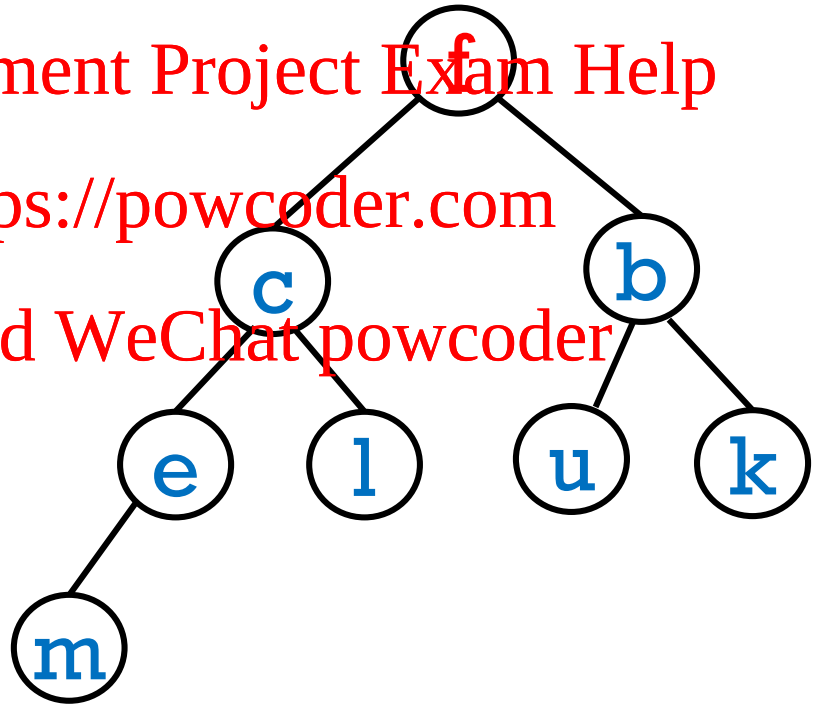
a

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?

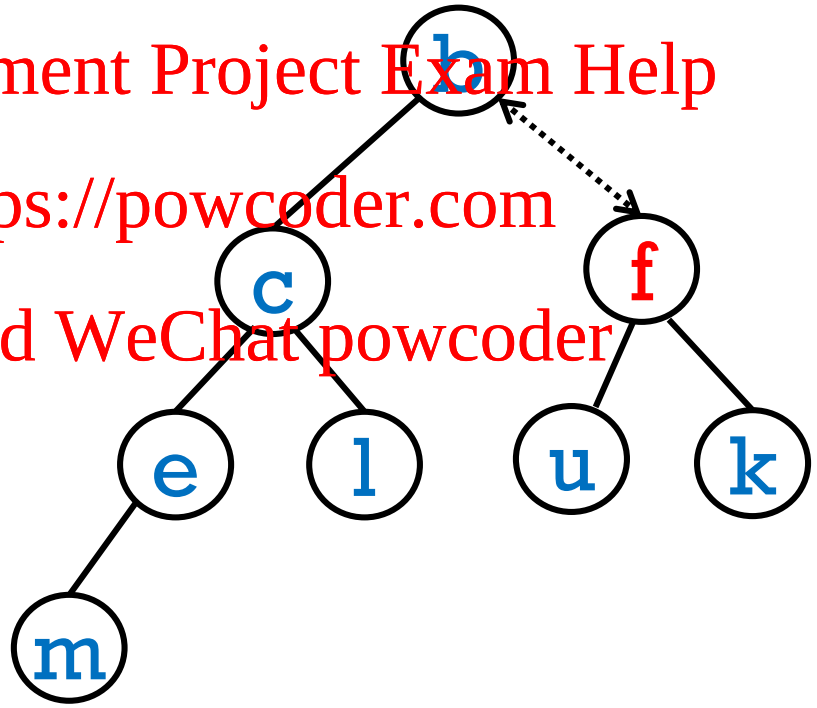




# REMOVEMIN()

Swap keys with the smaller child!



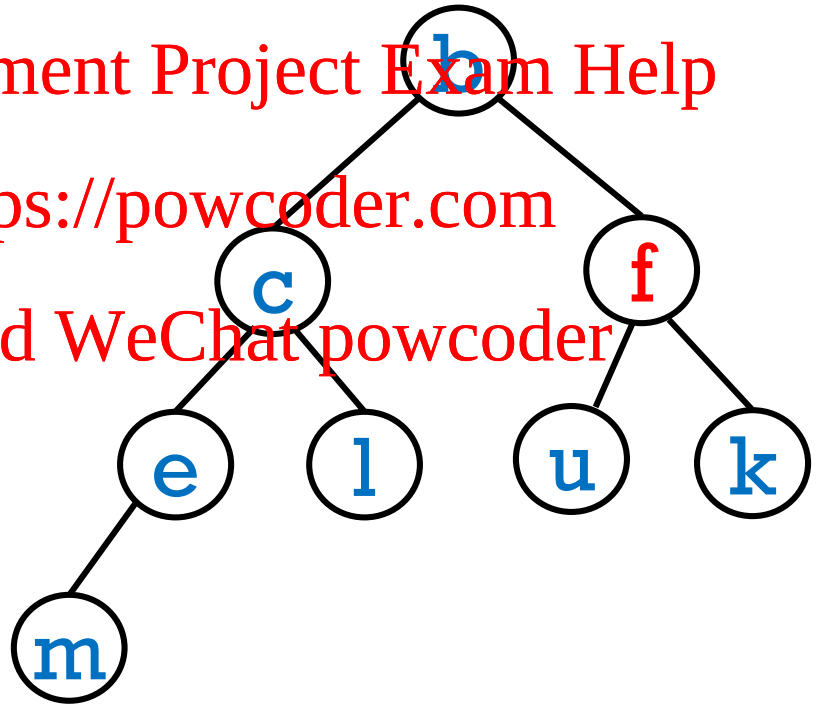


# REMOVEMIN()

Swap keys with the smaller child!

Keep swapping with keys with the smaller child until it's necessary.

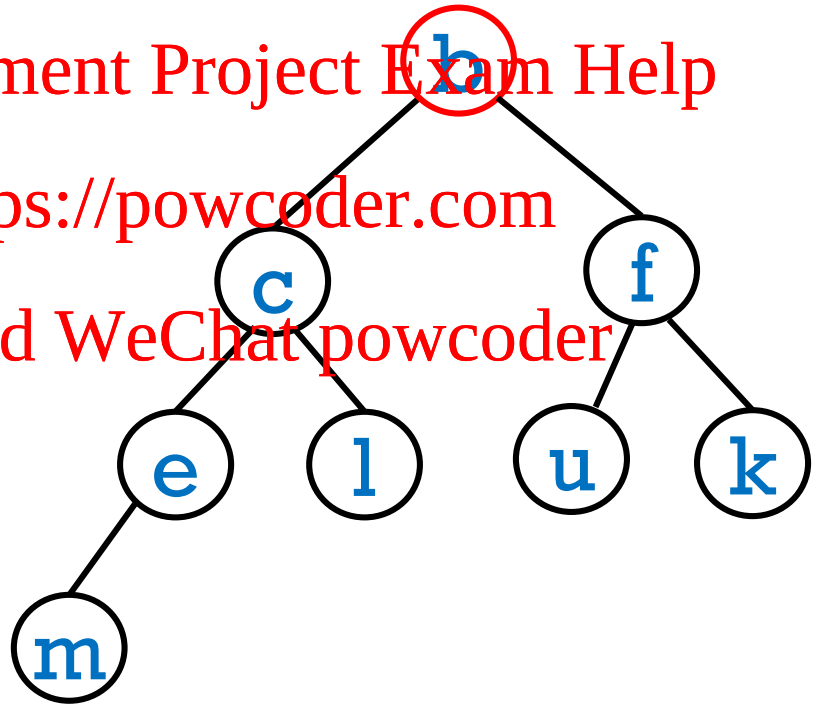




# REMOVEMIN()

Let's removeMin() again!

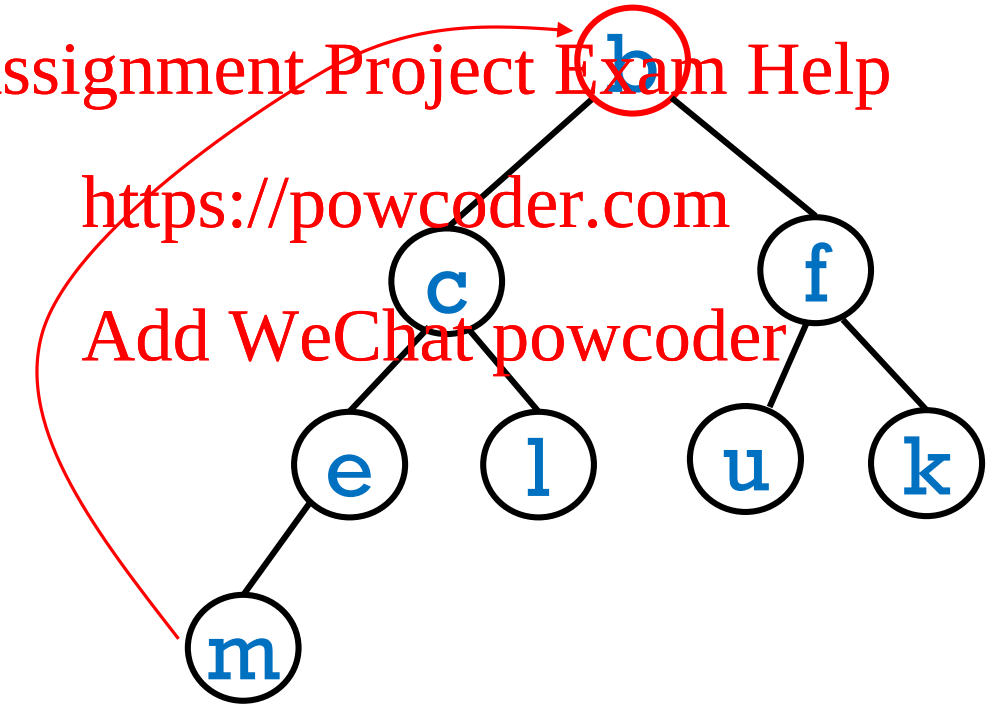




# REMOVEMIN()

Let's removeMin() again!

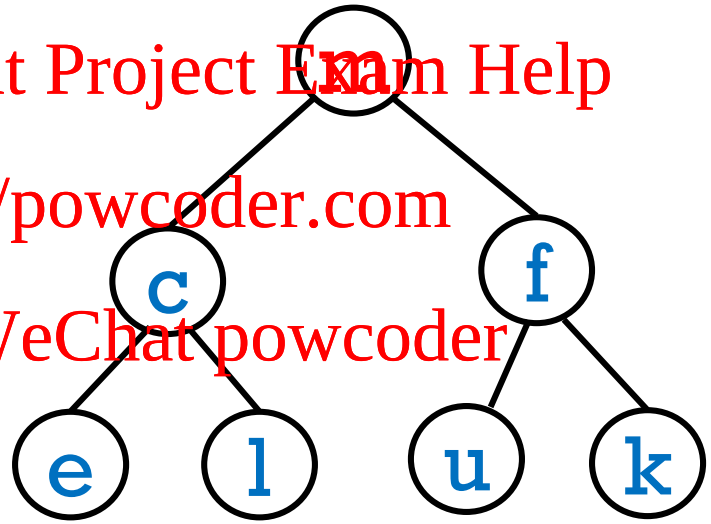




# REMOVEMIN()

Let's removeMin() again!

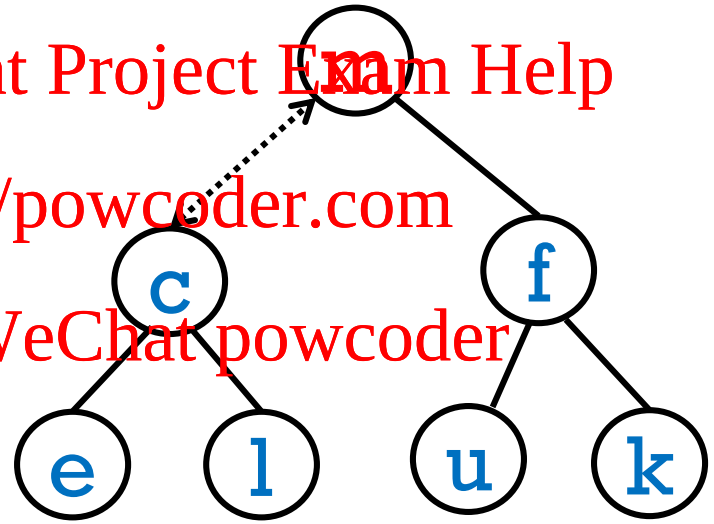




# REMOVEMIN()

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

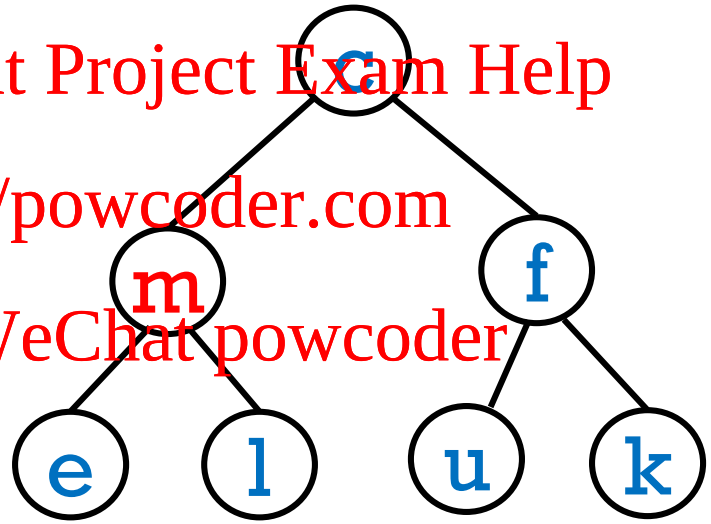




# REMOVEMIN()

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

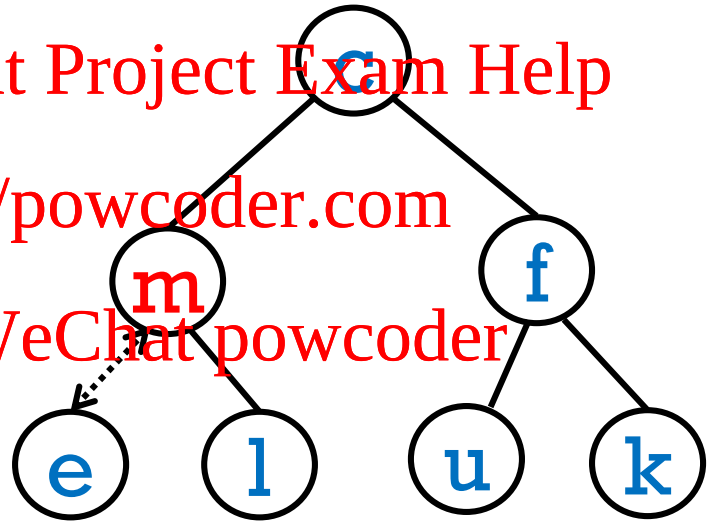




# REMOVEMIN()

Keep swapping with smaller child, if necessary.

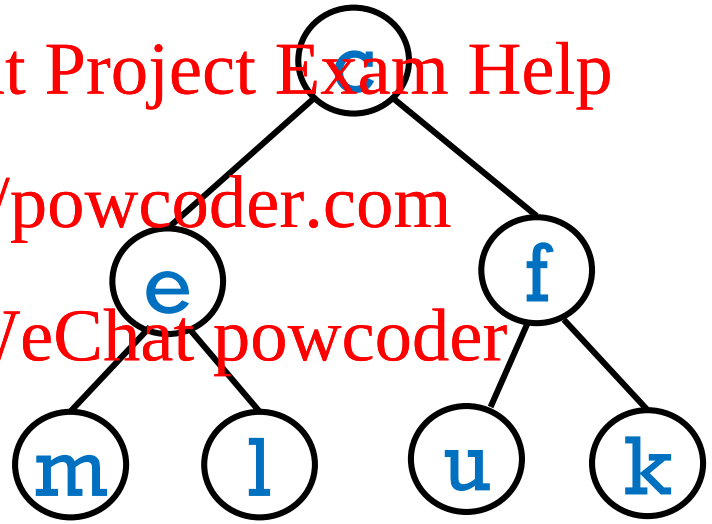




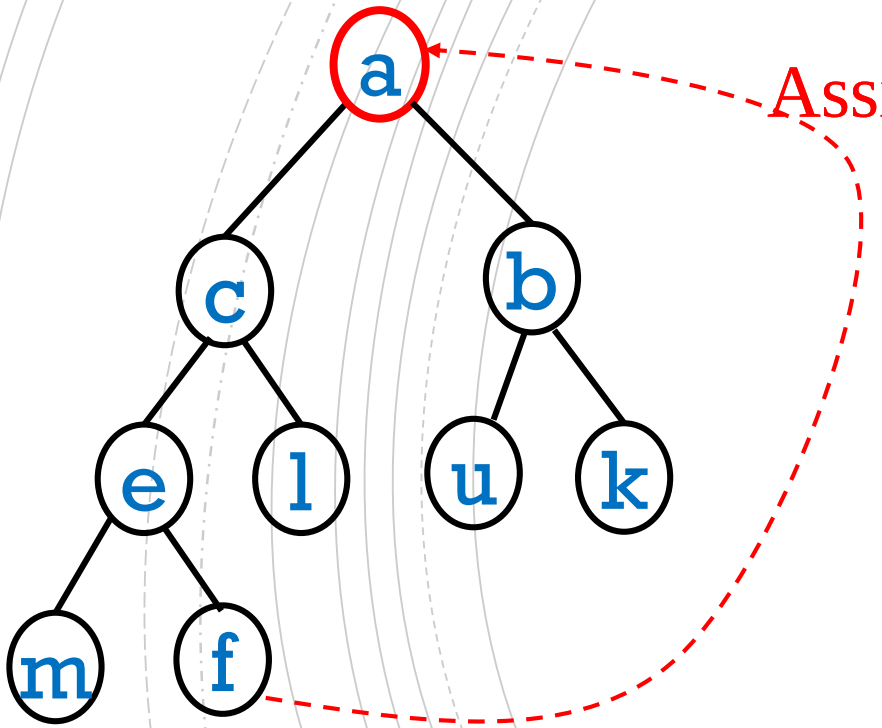
# REMOVEMIN()

Keep swapping with smaller child, if necessary.



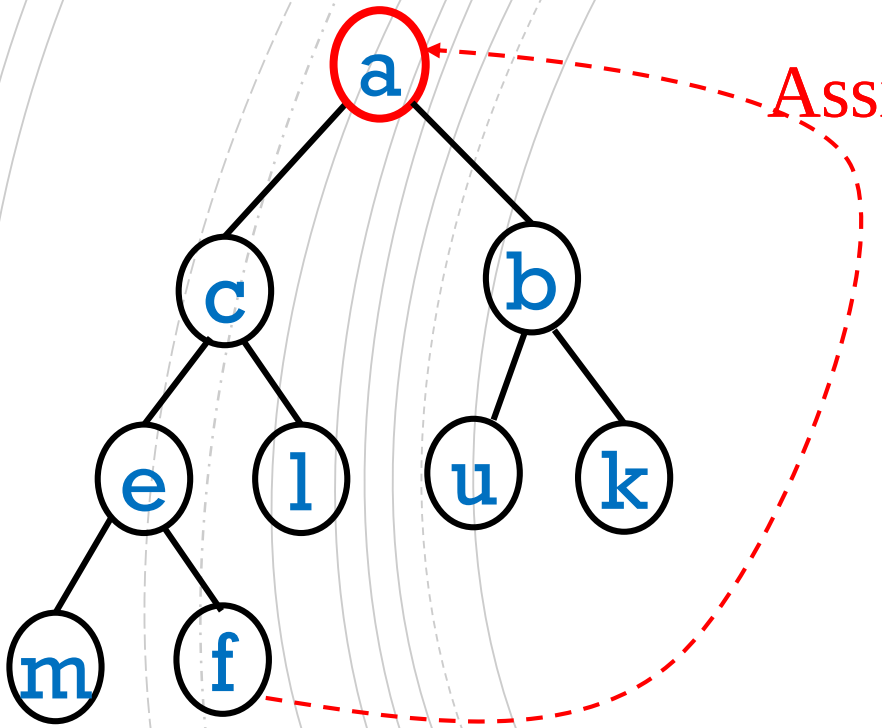


# REMOVEMIN() - IMPLEMENTATION




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

    return temp
}
```

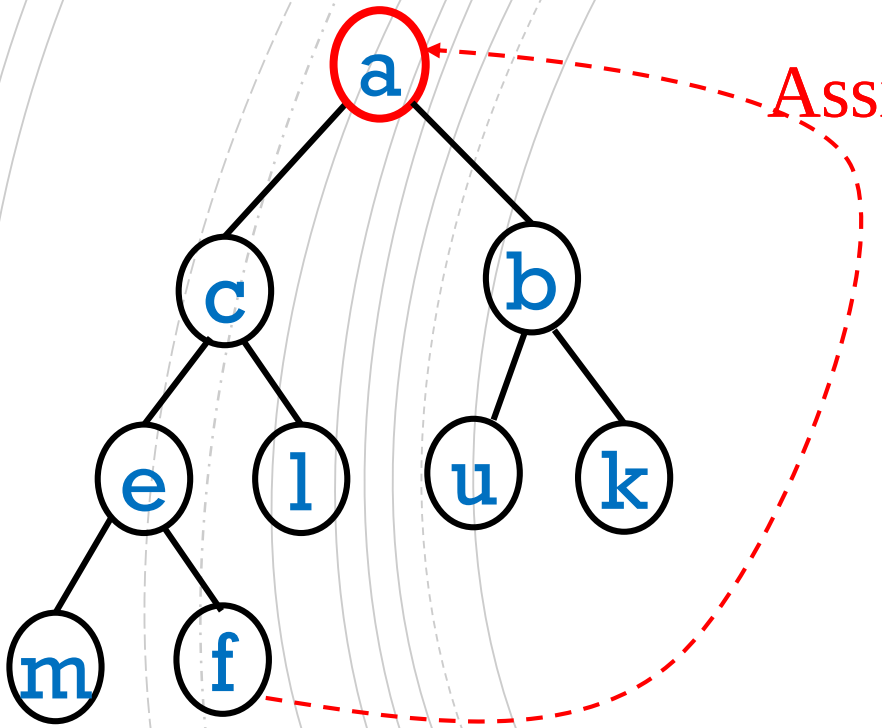
# REMOVEMIN() - IMPLEMENTATION






```
removeMin(){
    temp = root.key
    remove the last leaf node and
    store its key into the root
    cur = root
    while((cur.left!=null && cur.key > cur.left.key)
    || (cur.right!=null && cur.key > cur.right.key)) {




    }
    return temp
}
```

# REMOVEMIN() - IMPLEMENTATION






```
removeMin(){
    temp = root.key
    remove the last leaf node and
    store its key into the root
    cur = root
    while((cur.left!=null && cur.key > cur.left.key)
    || (cur.right!=null && cur.key > cur.right.key)) {
        minChild = child with smaller key
        swapKeys(cur, minChild)
        cur = minChild
    }
    return temp
}
```

add()

removeMin()



“upHeap”

“downHeap”

# REMOVE()

Q: What about remove(key) ?

# REMOVE()

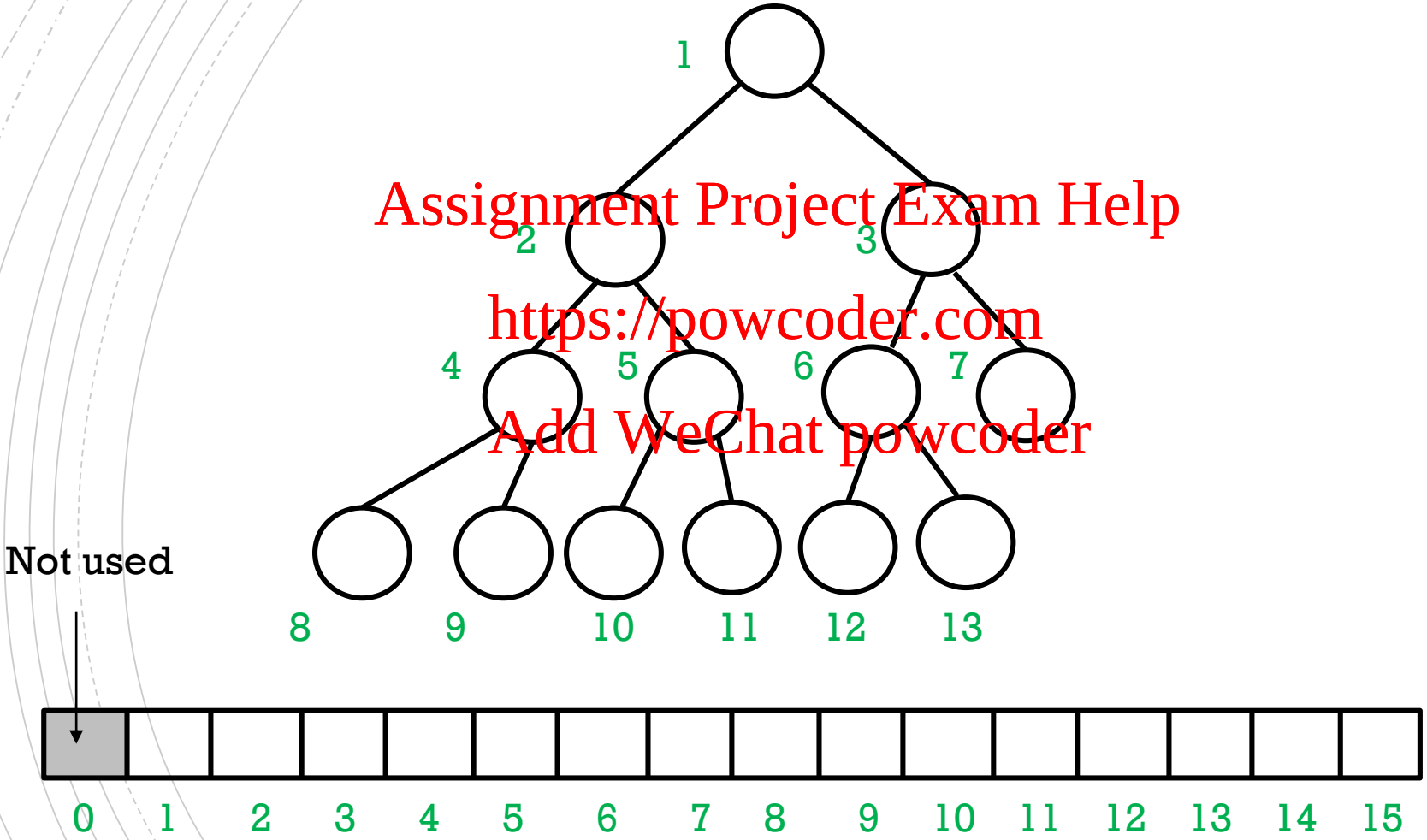
Q: What about remove(key) ?



A: Worst case $O(n)$

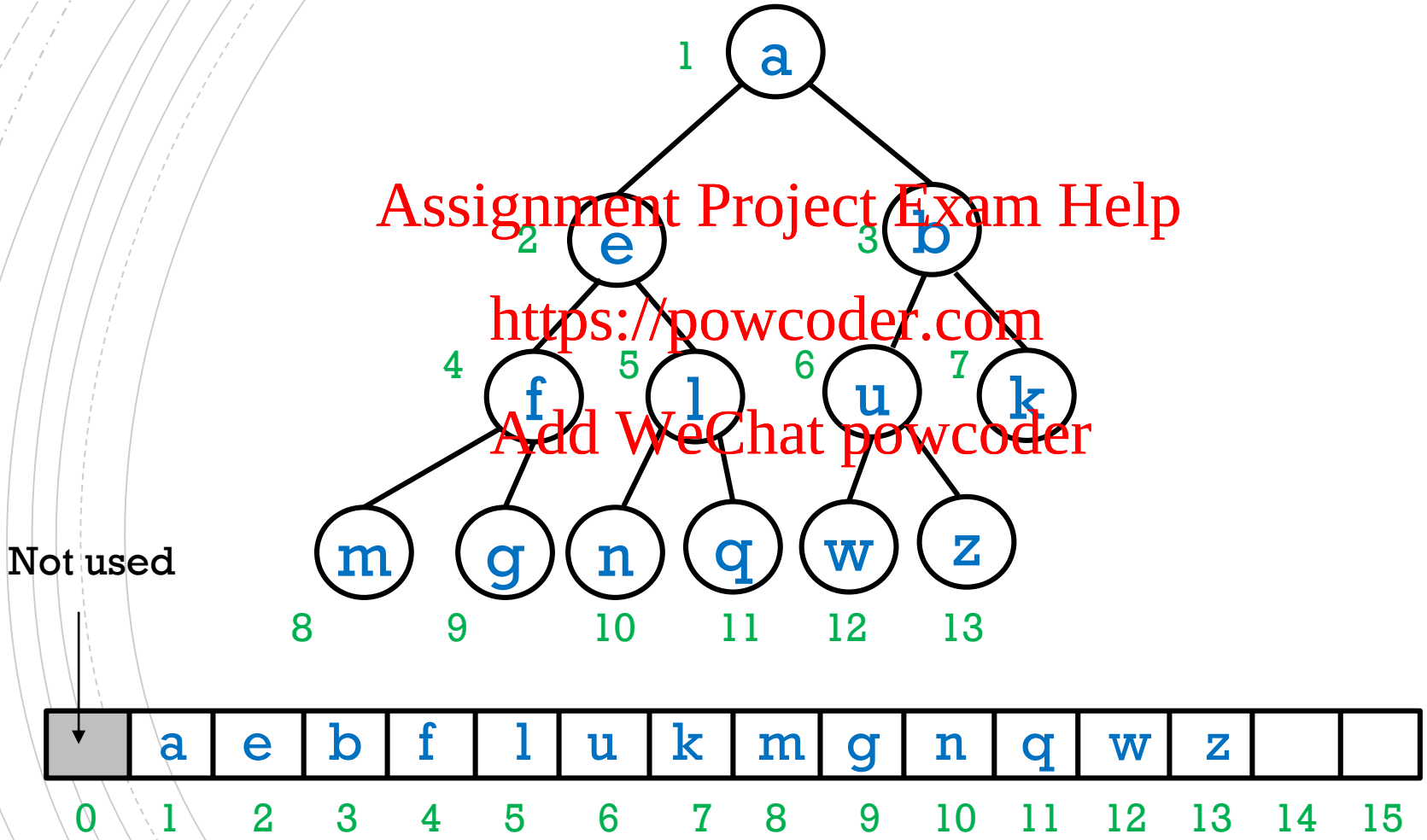
# HEAP (ARRAY IMPLEMENTATION)





# HEAP (ARRAY IMPLEMENTATION)

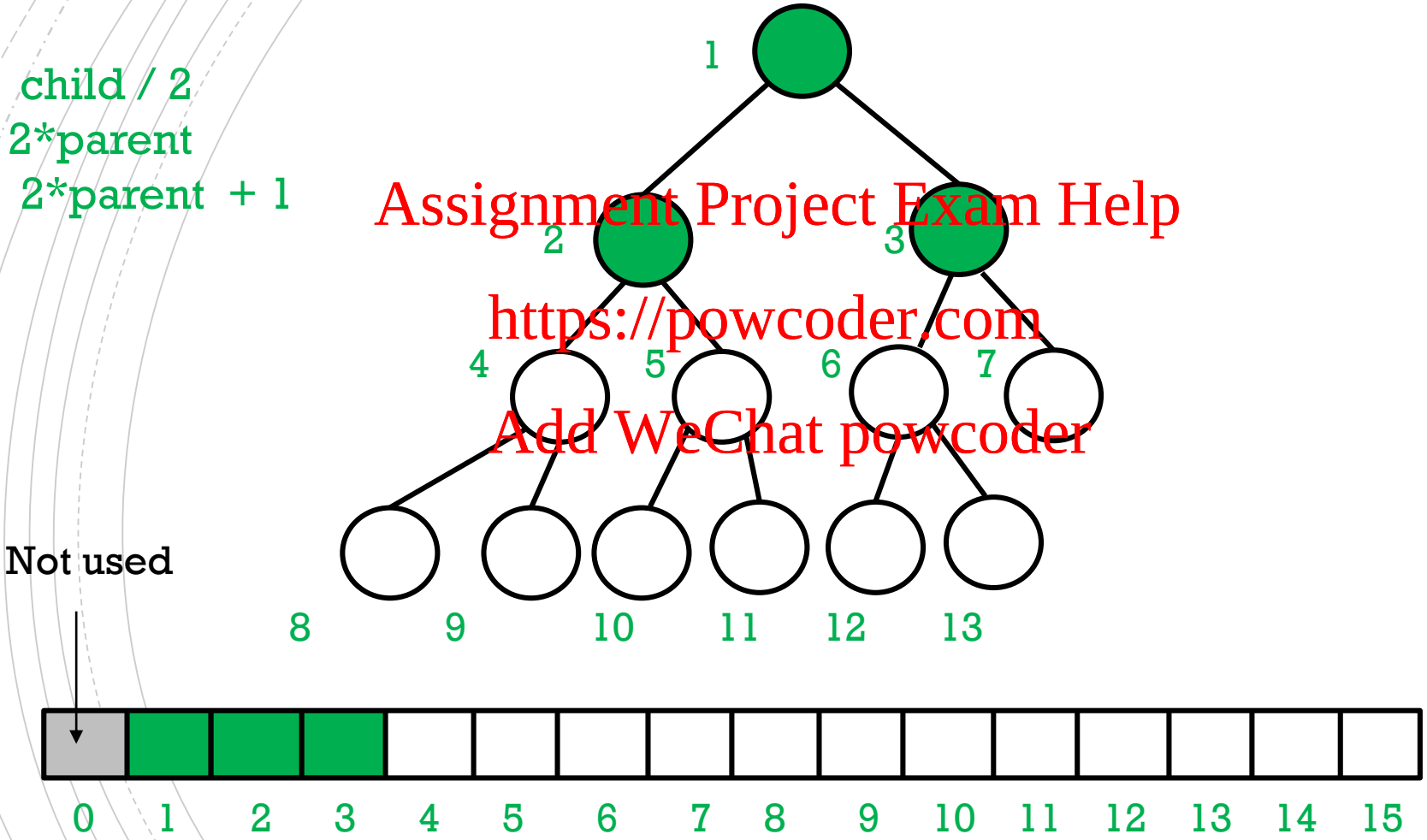




# HEAP INDEX RELATIONS

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

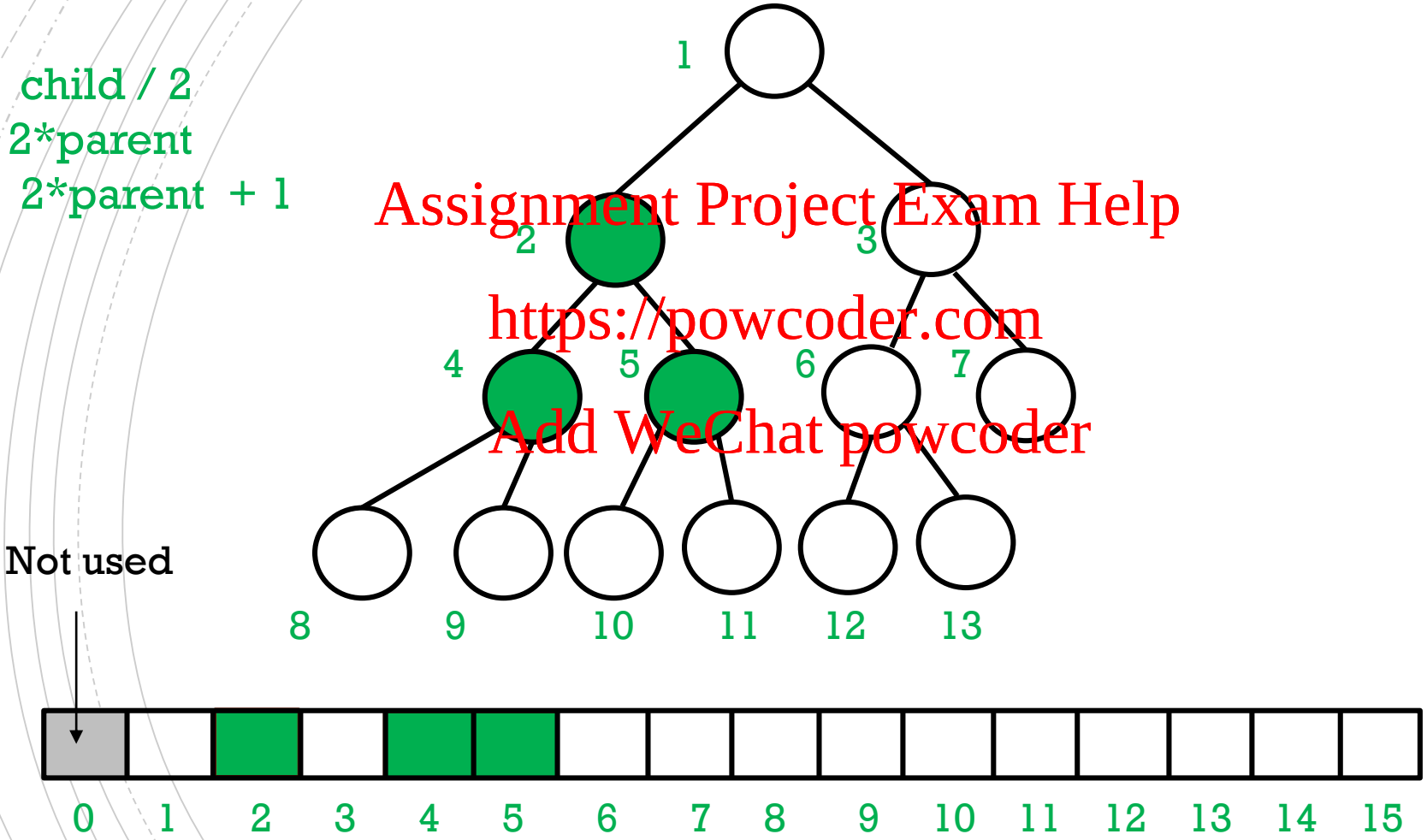




# HEAP INDEX RELATIONS

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

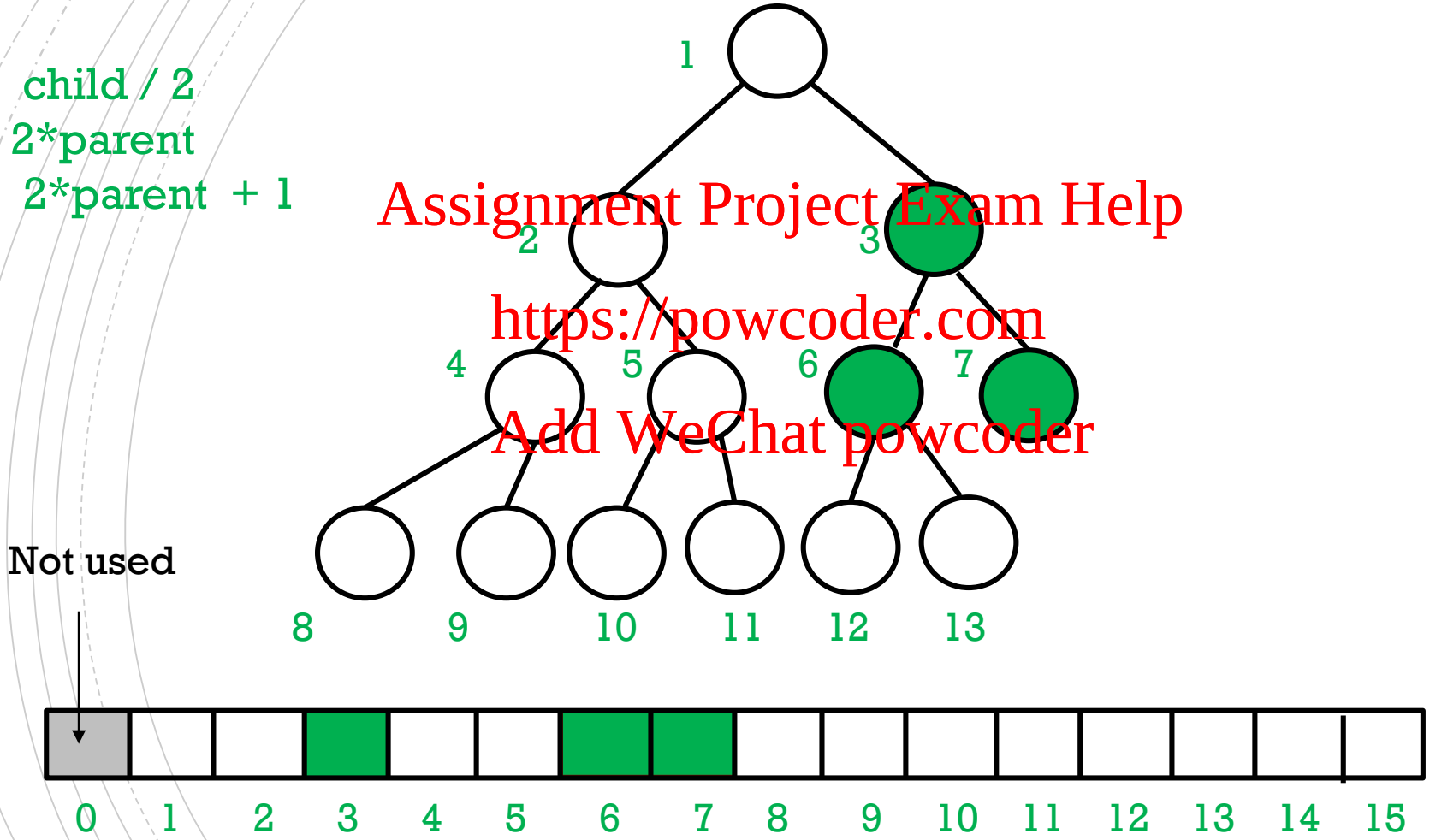




# HEAP INDEX RELATIONS

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

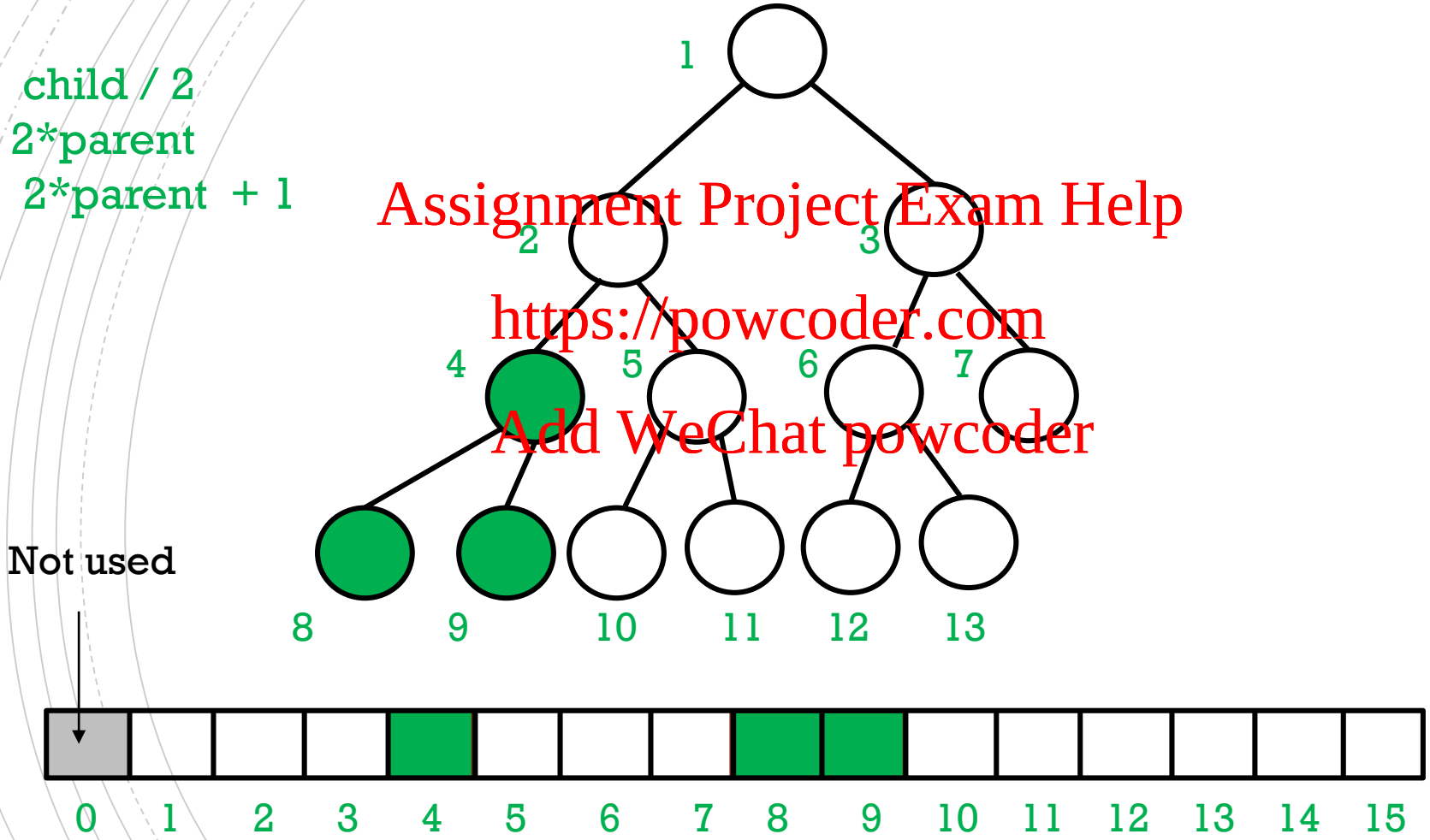




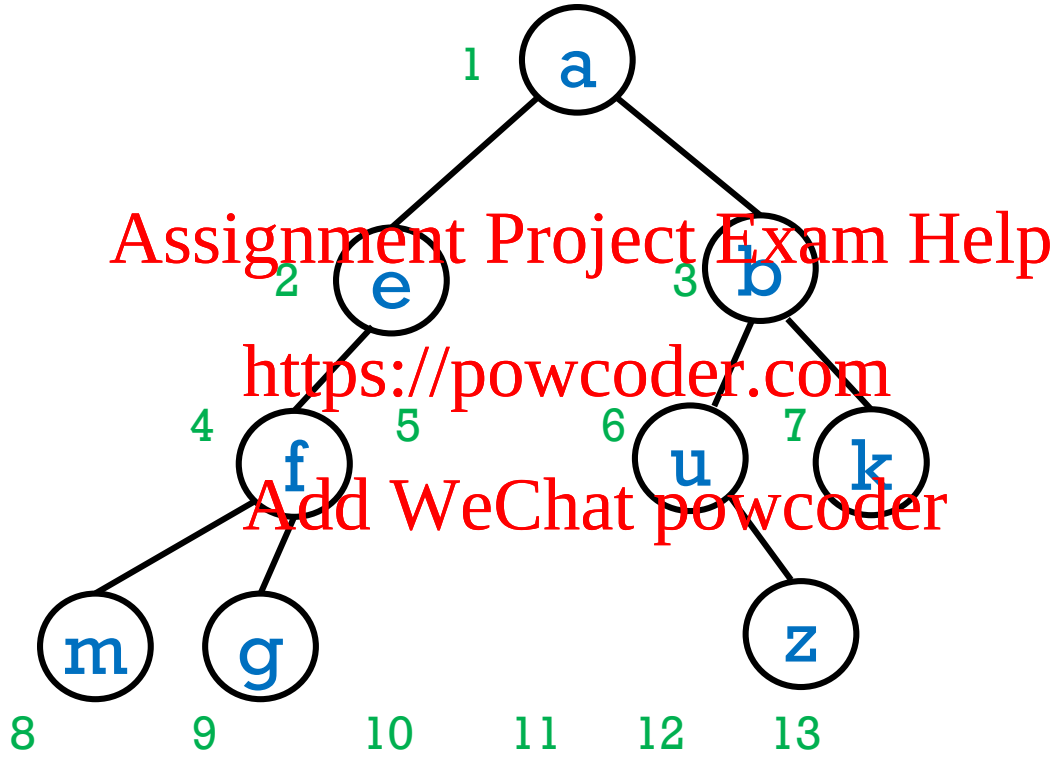
# HEAP INDEX RELATIONS

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

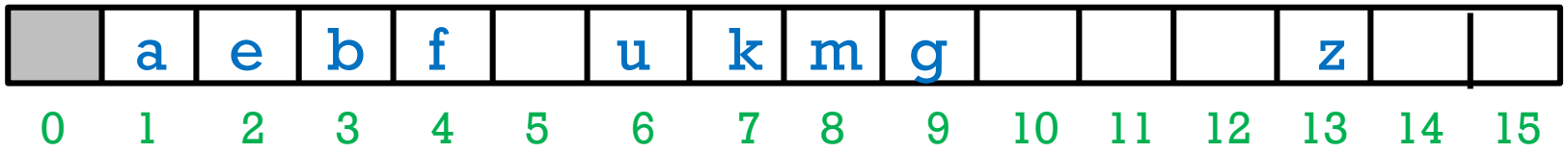




ASIDE: an array data structure can be used for *any* binary tree. But this is uncommon and often inefficient.



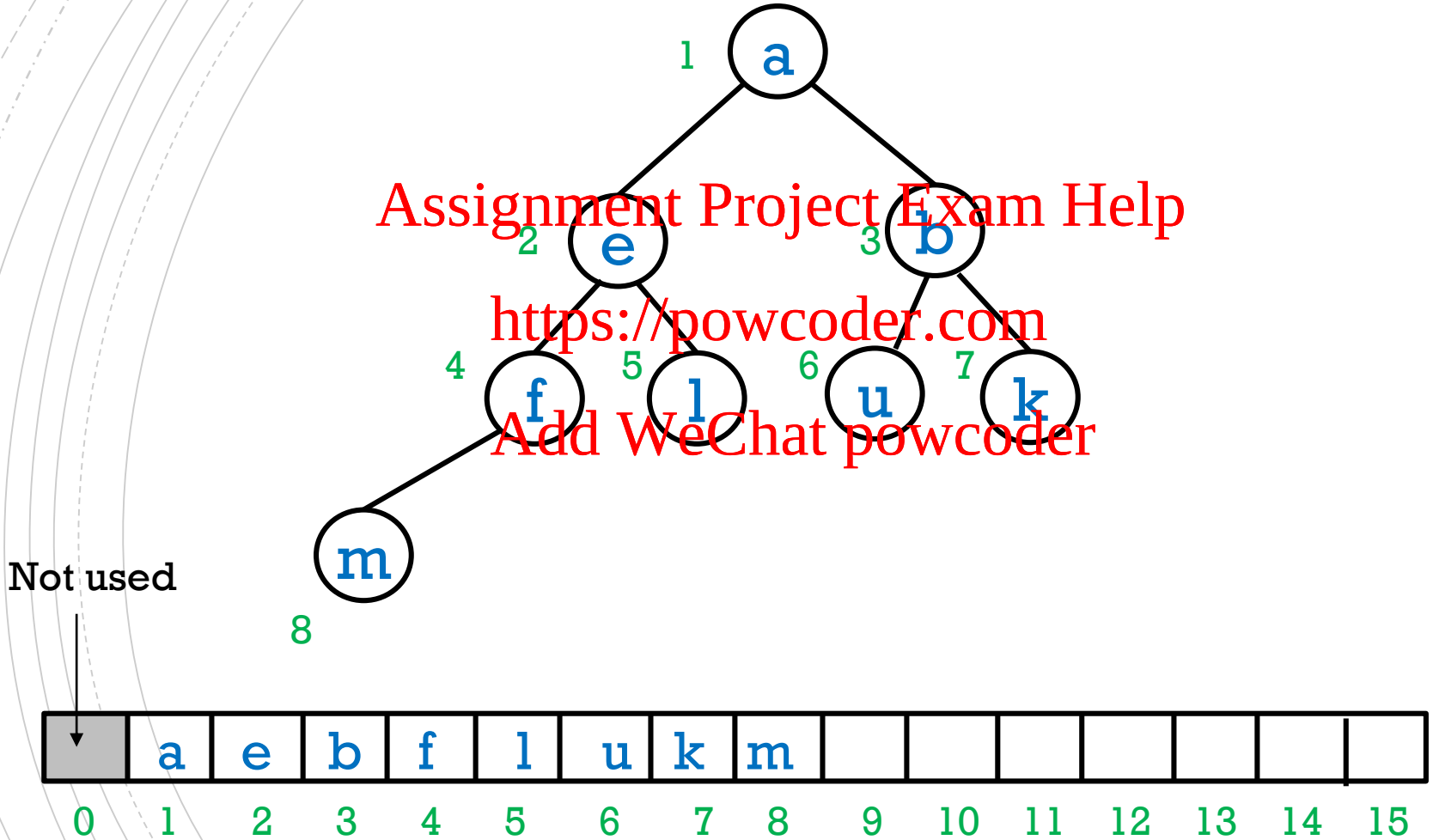


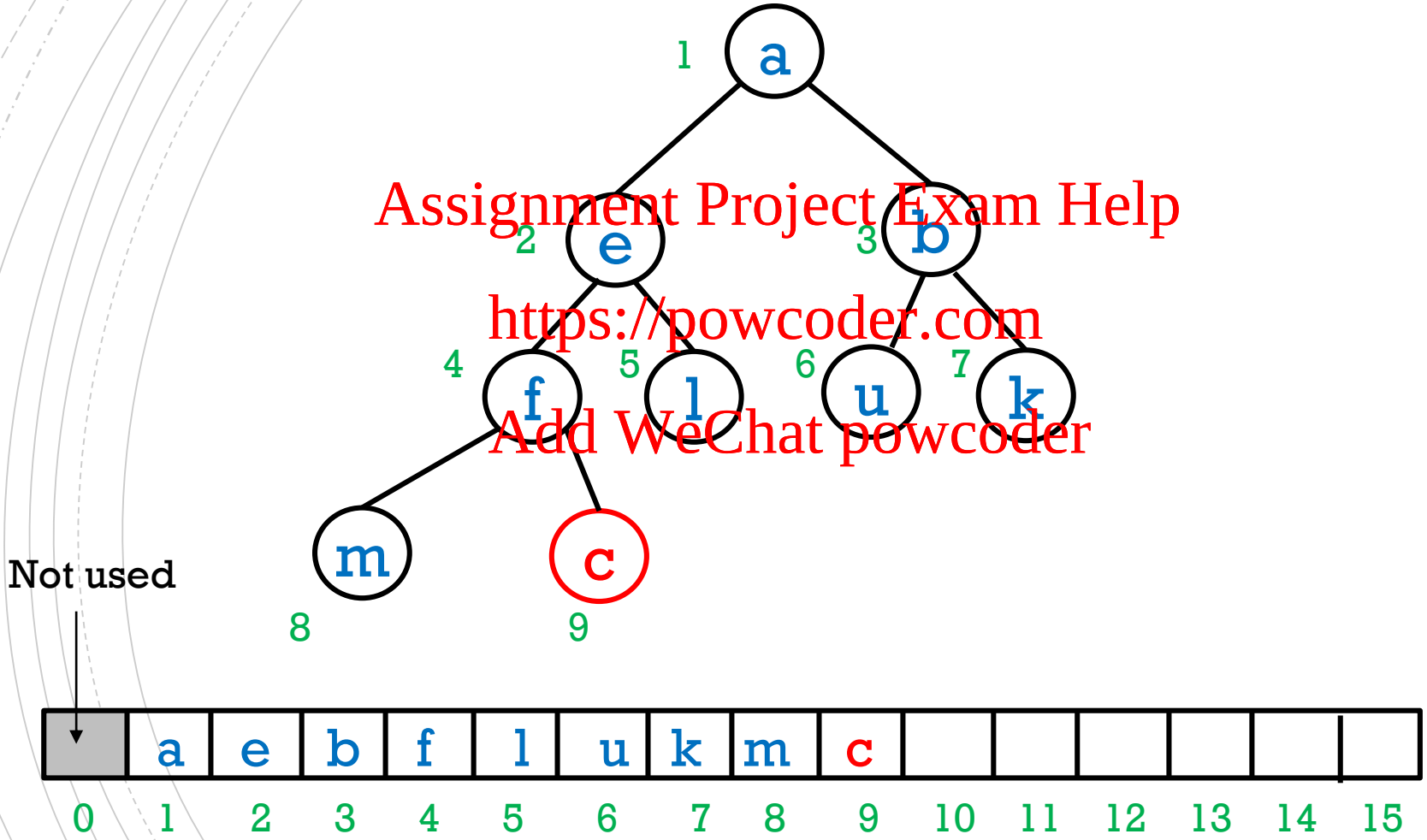
# ADD() - IMPLEMENTATION

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

    i = size

    //  the following is sometimes called "upHeap"
    while ( i > 1  &&  heap[i] < heap[ i/2 ]){
        swapElements( i, i/2 )
        i = i/2
    }
}
```

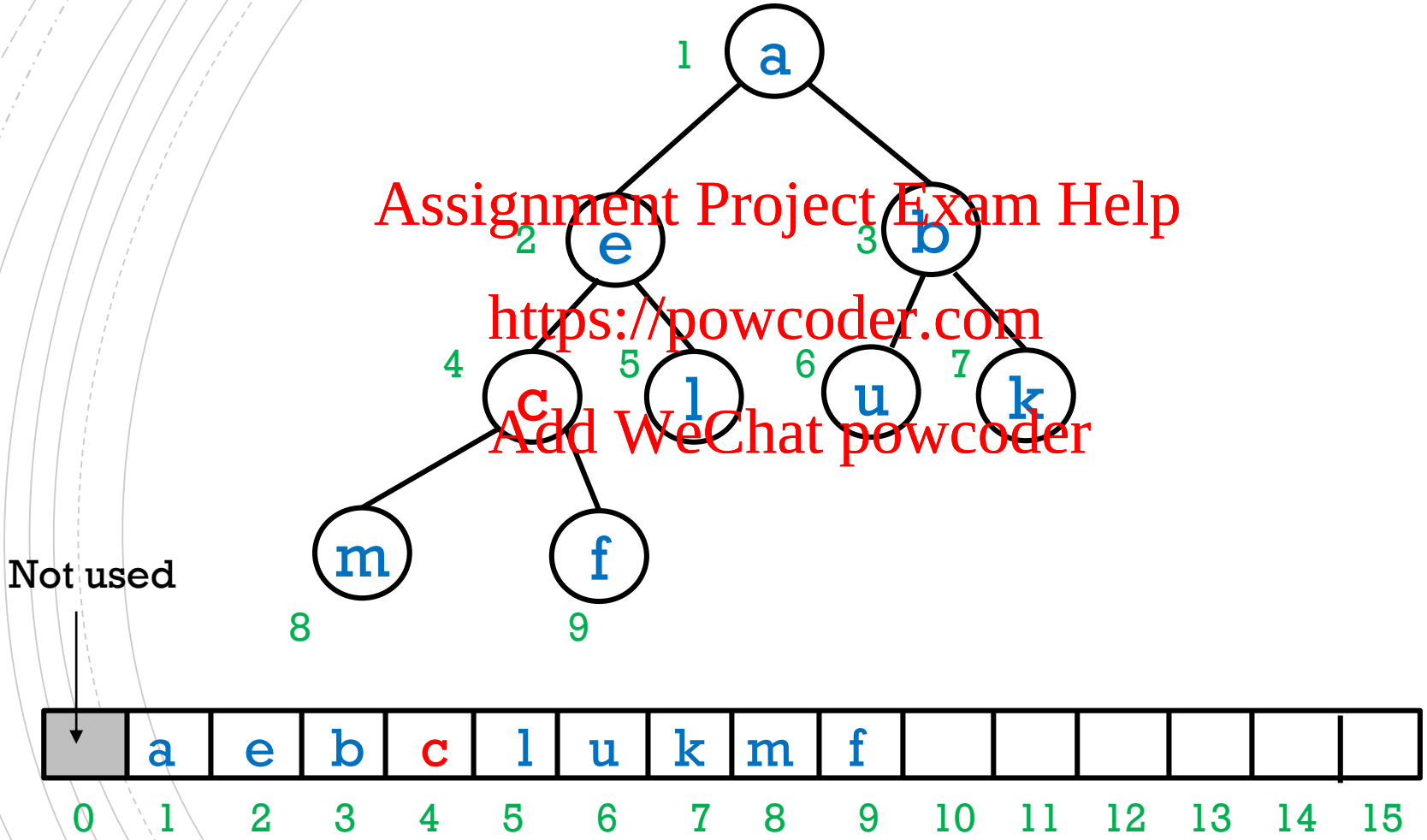
# E.G. add (c)



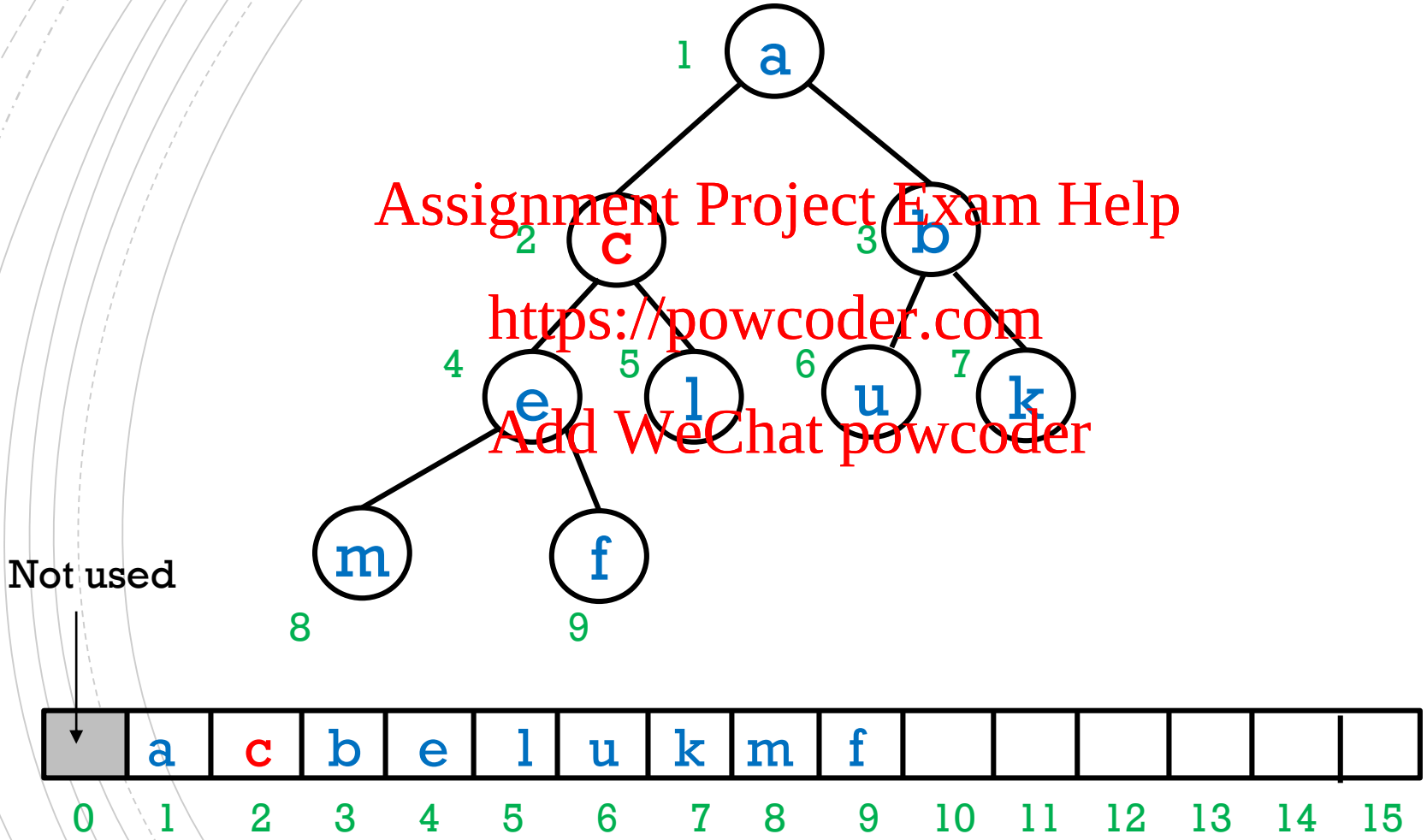
# E.G. add(c)



# E.G. add(c)

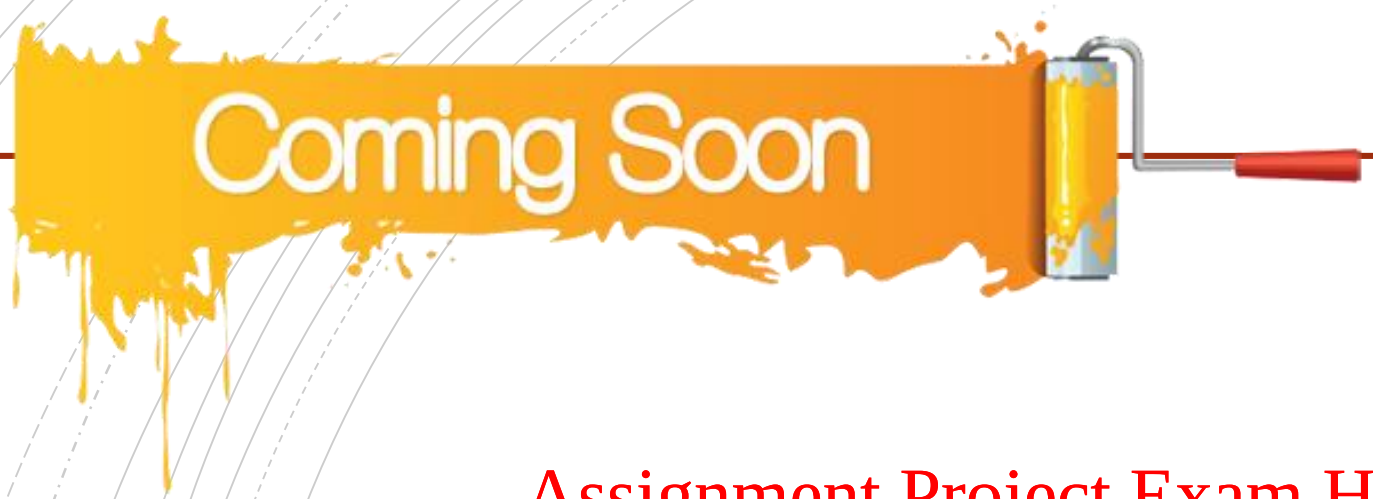


# E.G. add (c)



# NEXT VIDEO

- write  removeMin() using array indices
- discuss best and worst case
- faster algorithm for building a heap






In the next video:



- More on heaps