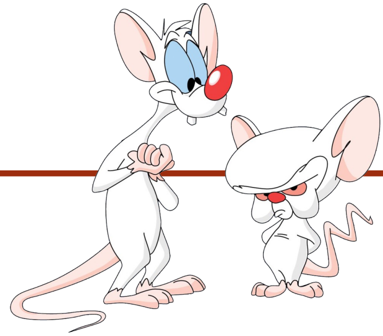
ECSE 250 FUNDAMENTALS OF SOFTWARE DEVELOPMENT

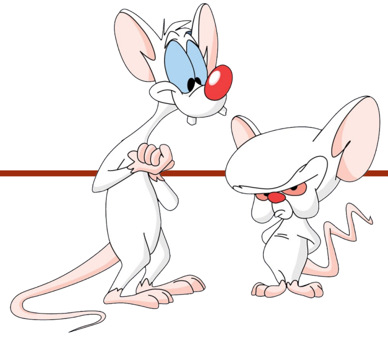
22 – Heaps

Lili Wei
Material by Giulia Alberini and Michael Langer



WHAT ARE WE GOING TO DO TODAY?

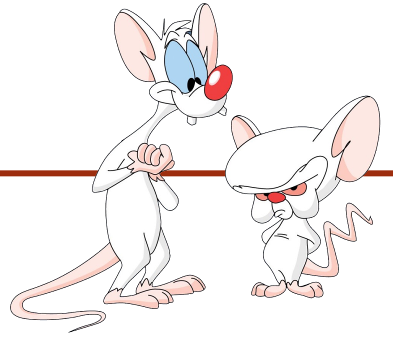
- This is the last lecture that is required in the final exam
- Coming lectures:
  - April 2 (Tentative): Set, Map and Hash Table + Personal advice for software engineering students
  - April 4: Review
  - April 9: Q&A (in classroom)
- Exam: 22 Apr 2024 2:00 PM to 5:00 PM
  - Closed book
  - No crib sheet
  - Multiple choice questions + open ended questions



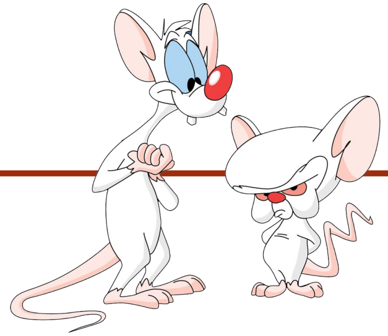
WHAT ARE WE GOING TO DO TODAY?

- Exam: 22 Apr 2024 2:00 PM to 5:00 PM
  - Closed book
  - No crib sheet
  - Multiple choice questions + open ended questions
- No questions are allowed to be asked during the final exam.
  - If you think that there are some problems in the exam questions, write down the problem and your answer assuming the problem is fixed.
  - When grading, we will decide whether the problem is legitimate and whether you answer is correct.
  - Why are no questions allowed for the final exam?
    - The finals are run by the exam office in the gym together with other courses. It will be very disturbing if we want to make any announcements or corrections in the final.

HASH TABLE & SORTING



- A nice note on hash table and sorting:
- https://pages.cs.wisc.edu/~siff/CS367/Notes/sorting.html

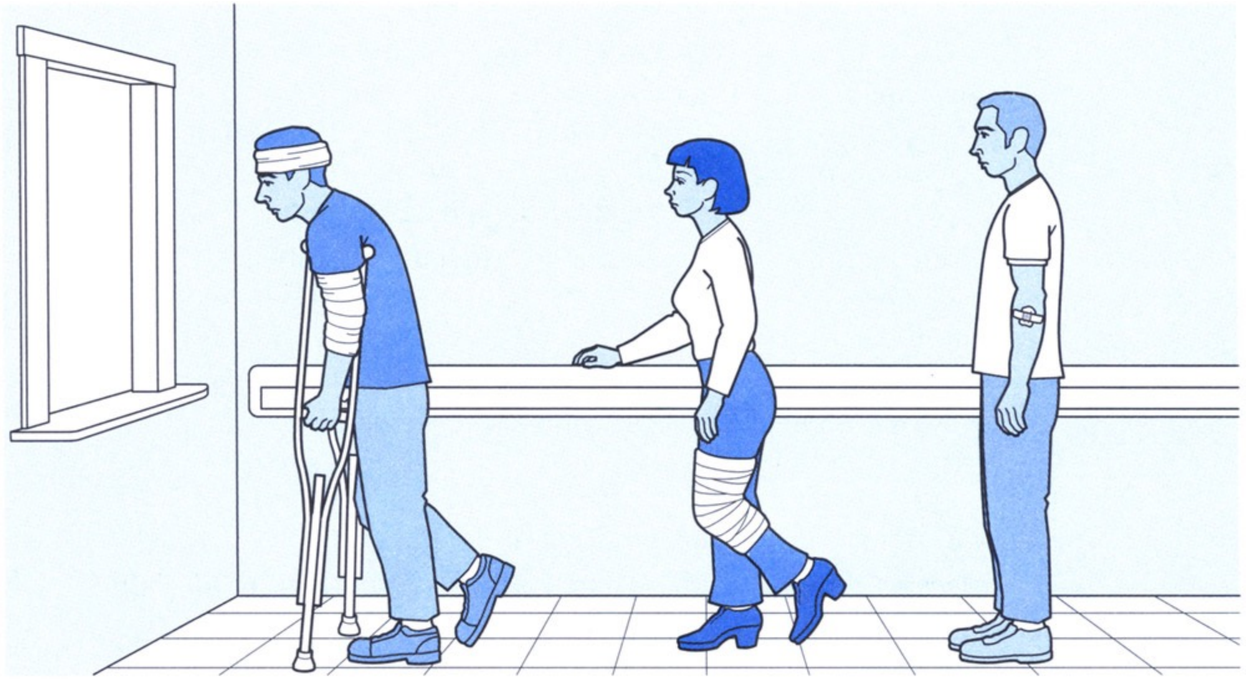


WHAT ARE WE GOING TO DO TODAY?

- Heaps
  - ADD
  - REMOVE
- Implementation of a heap using arrays
  - Add
  - Remove

PRIORITY QUEUE and HEAPS

PRIORITY QUEUE



Priority Queue

- An ADT that supports two operations
  - Add an element
  - Remove the elements with the highest priority

PRIORITY QUEUE ADT

- add(key)
- removeMin()

  “highest” priority = “number 1” priority

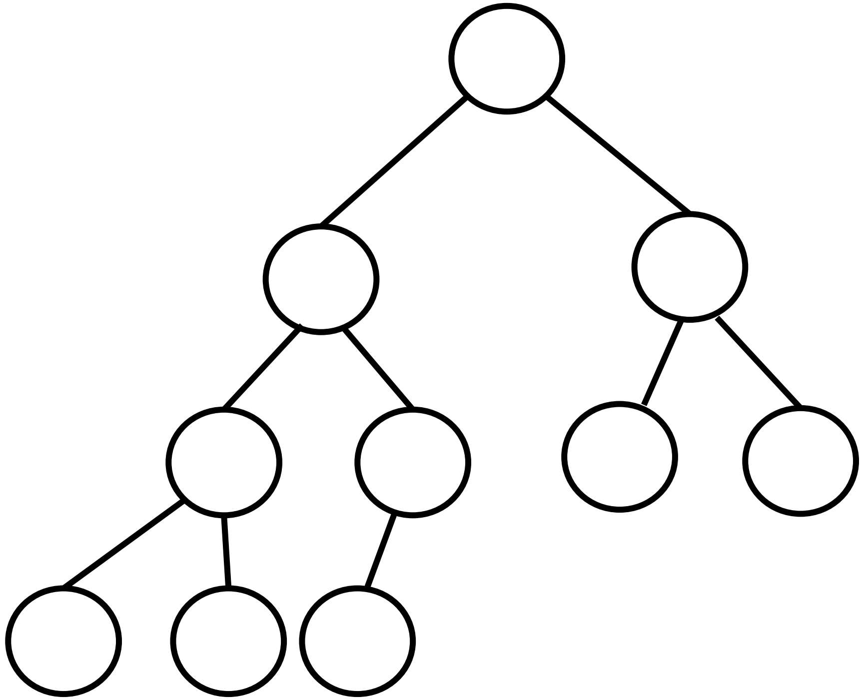
- peek()
- contains(element)
- remove(element)

HOW TO IMPLEMENT A PRIORITY QUEUE ?

- sorted list ?
- binary search tree (last lecture) ?
- balanced binary search tree (add/remove is omitted in this course)?
- heap (today)

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

COMPLETE BINARY TREE

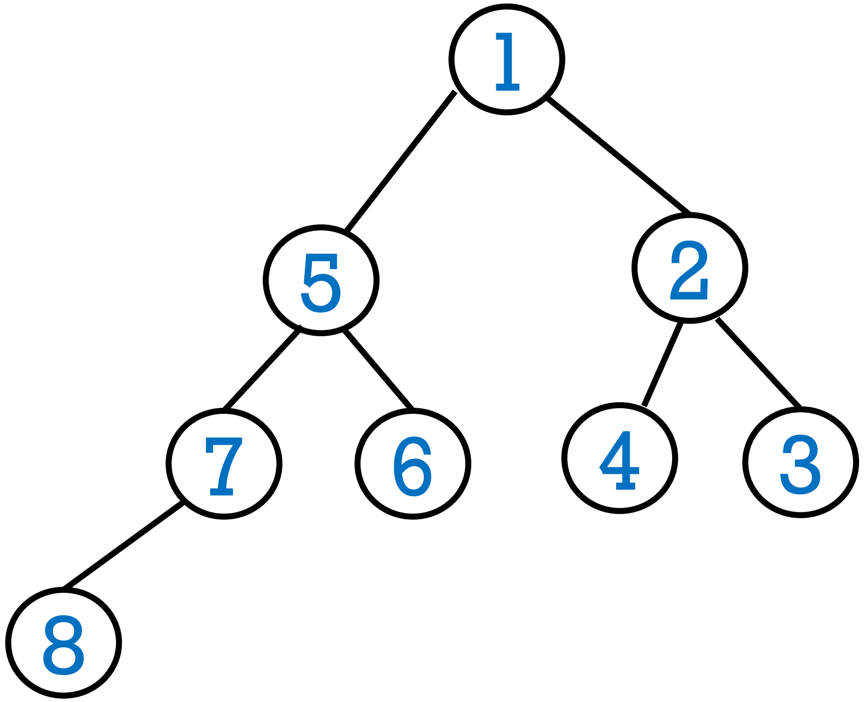


Binary tree of height h such that every level less than h is full, and all nodes at level h are filled from as left as possible.

HEAP (DEFINITION)

Heap

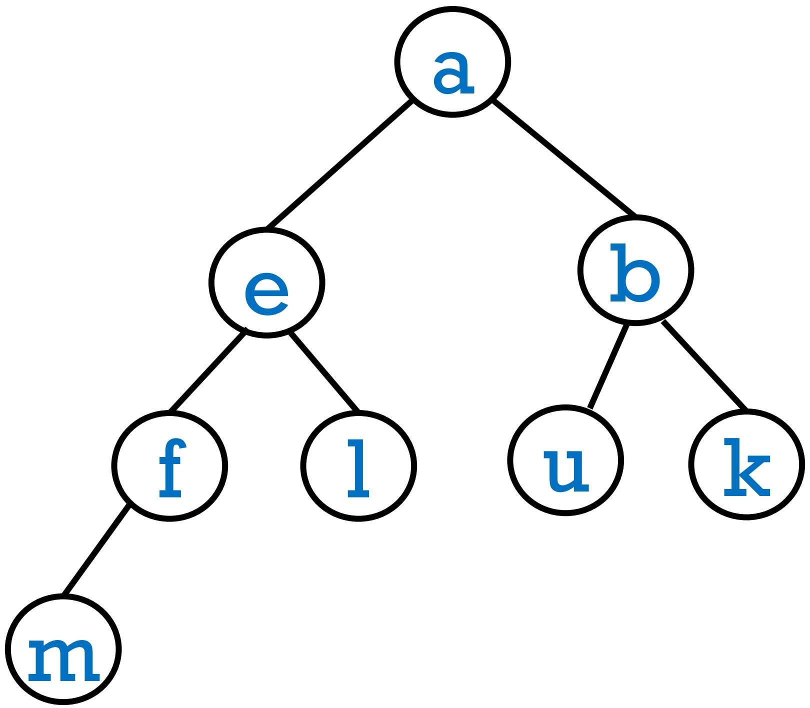
- Complete Binary Tree with comparable keys
- **Max-heap:** parent >= children
- **Min-heap:** parent <= children



ADD

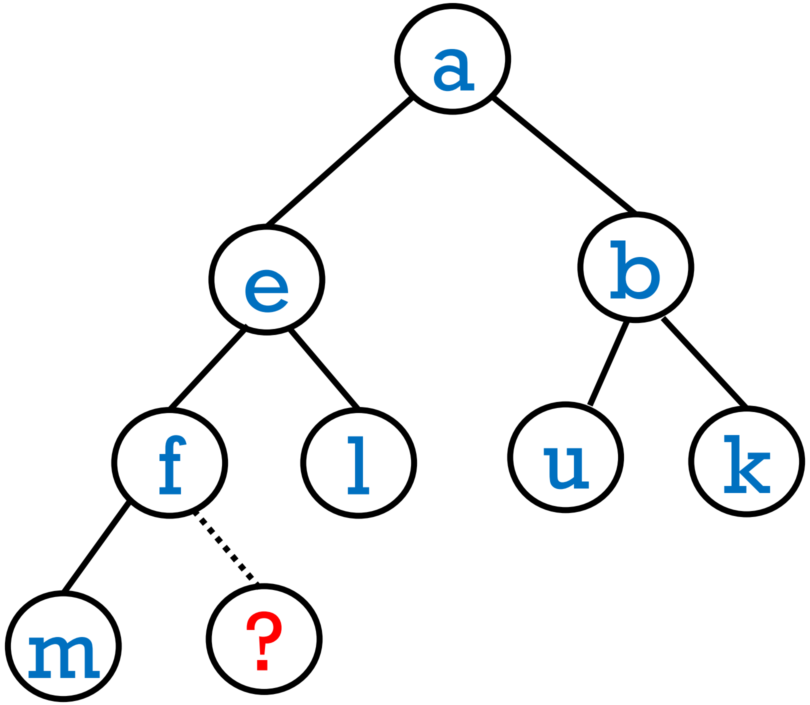
ADD

For example, `add(c)`



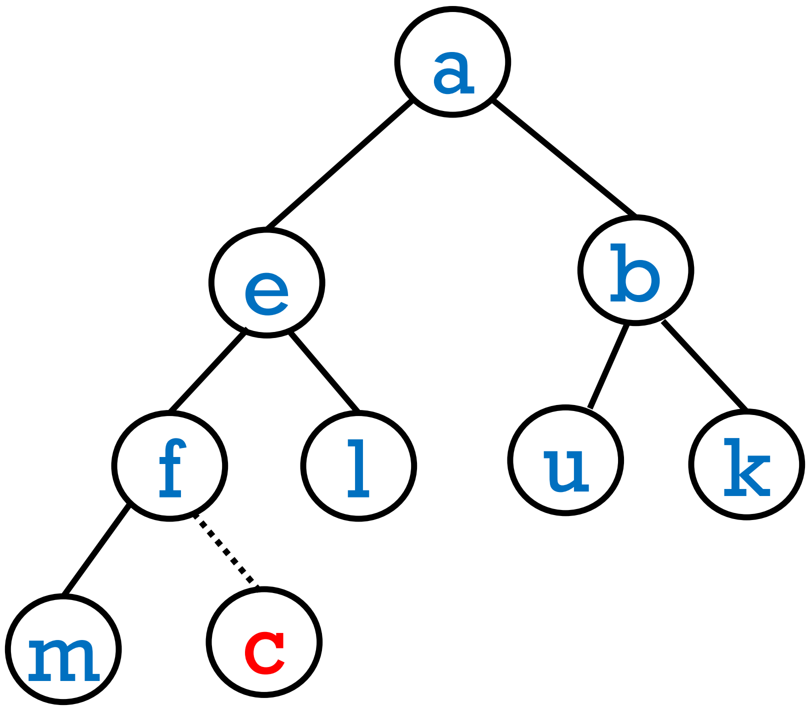
ADD

For example, `add(c)`



ADD

For example, `add(c)`



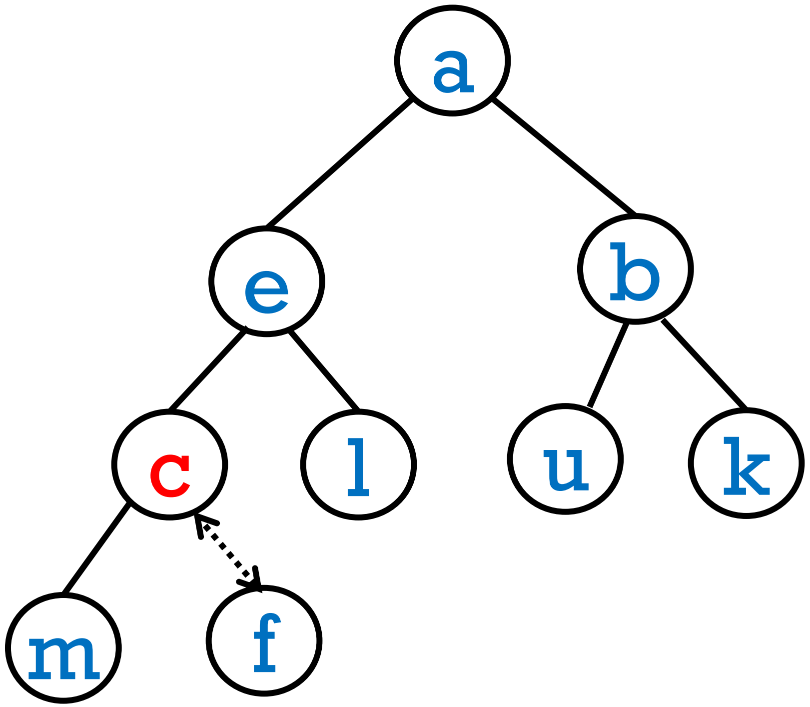
What can we do?

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

f > c

ADD

For example, `add(c)`



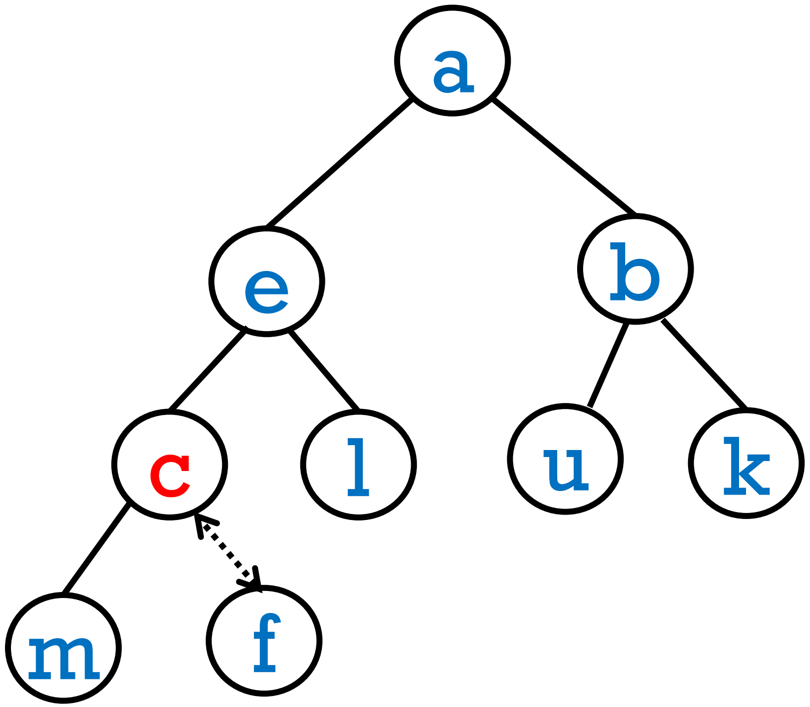
What can we do? ***Heapify***!

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? Heapify!

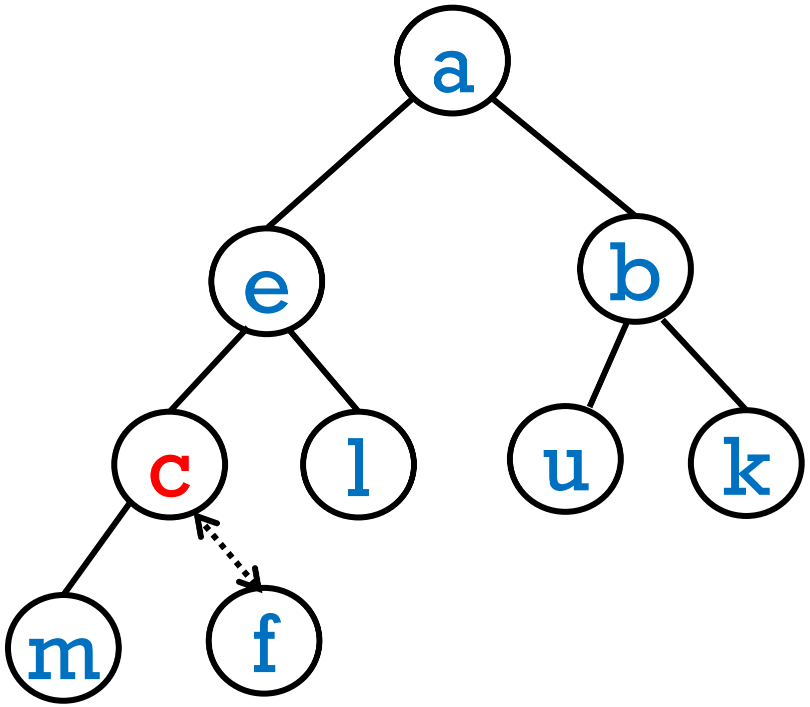
Let's swap c with its parent f.

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)`



What can we do? Heapify!

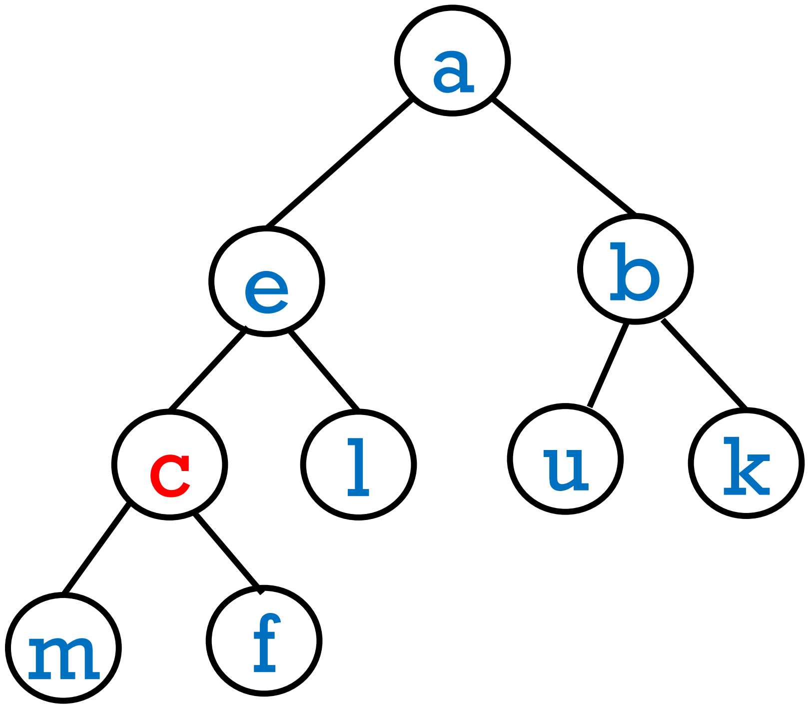
Let's swap c with its parent f.

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

A: No. Why?
We only need to heapify when c's parent is greater than c. For c's siblings, they are children of c's parent. They should be greater than the parent and thus should be greater than c.

ADD

For example, `add(c)`

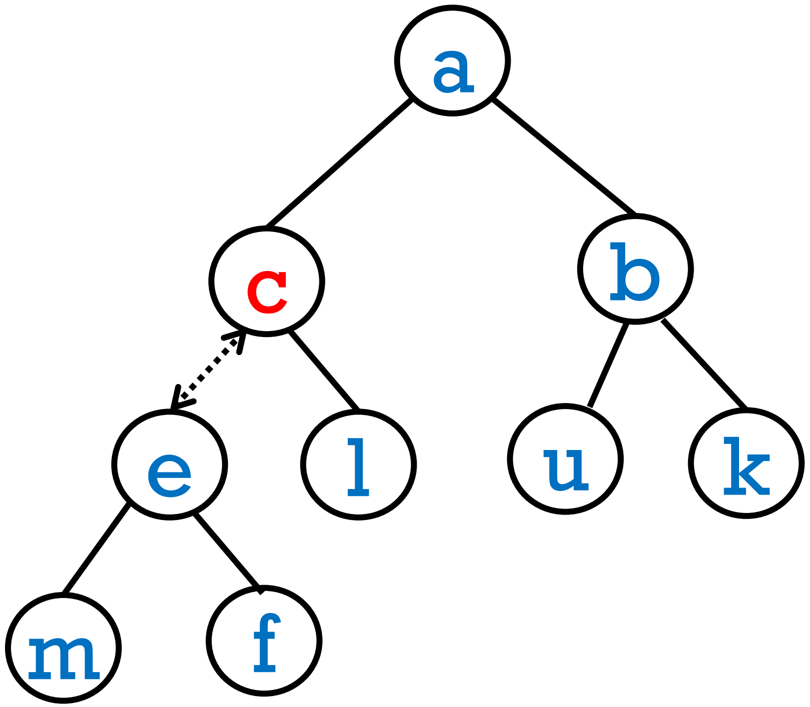


Q: Are we done?

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

ADD

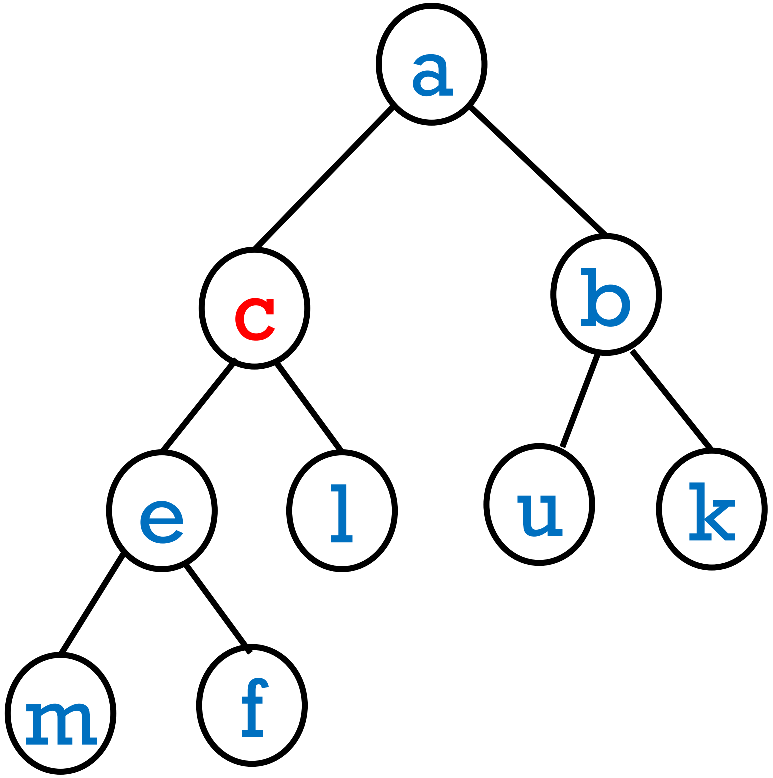
For example, `add(c)`



We swap c with its (new) parent e.

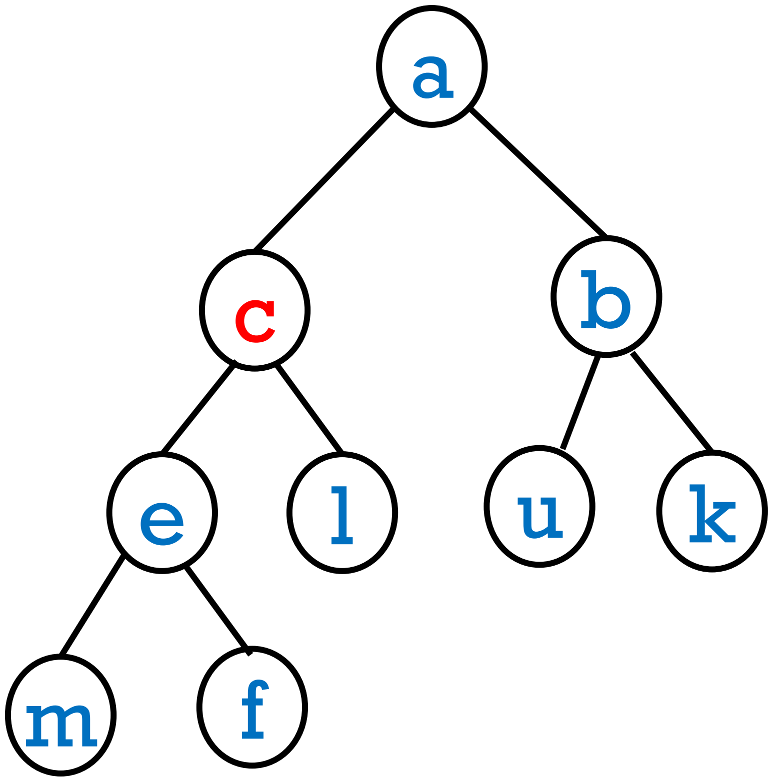
Now we are done because c is greater than its parent a

ADD - IMPLEMENTATION



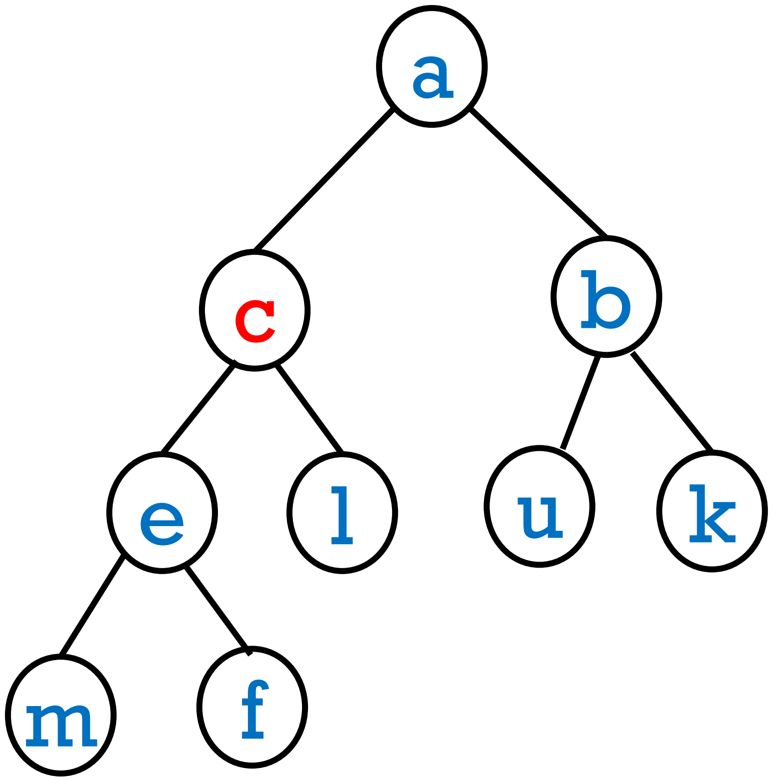

```
add(key) {
    cur = new node at next available leaf position
    cur.key = key



}
```

ADD - IMPLEMENTATION




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

}
```

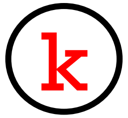
ADD - IMPLEMENTATION




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

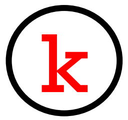
HOW TO BUILD A HEAP?

`add( k )`



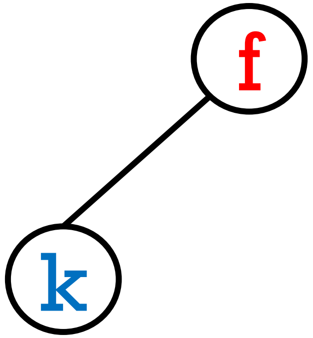
HOW TO BUILD A HEAP?

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



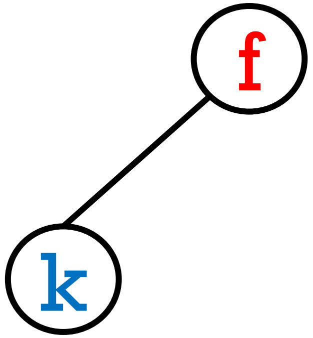
HOW TO BUILD A HEAP?

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



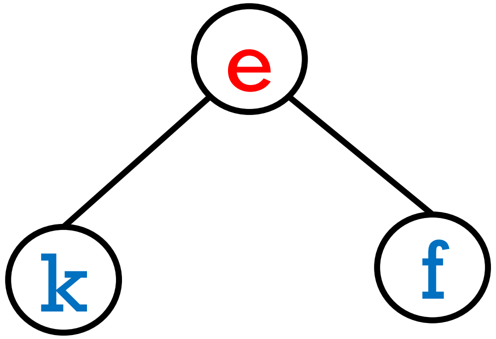
HOW TO BUILD A HEAP?

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



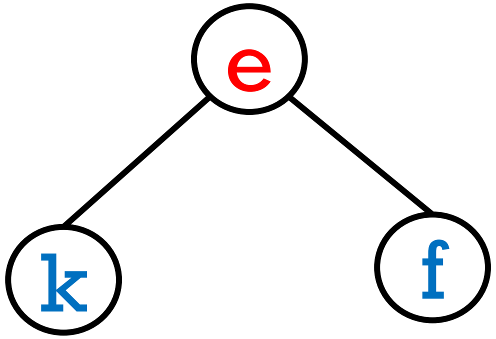
HOW TO BUILD A HEAP?

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



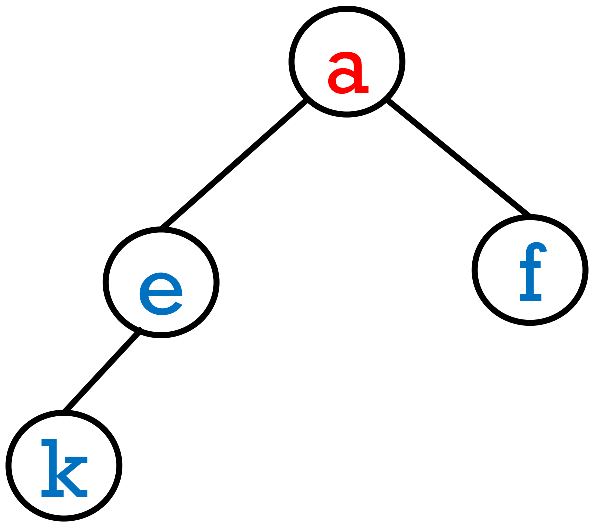
HOW TO BUILD A HEAP?

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



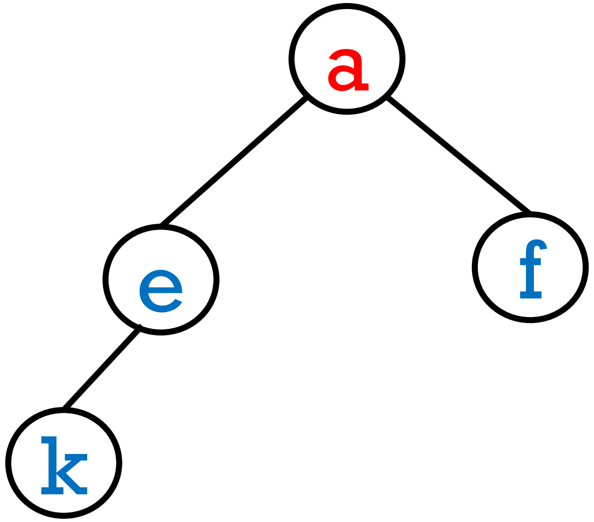
HOW TO BUILD A HEAP?

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



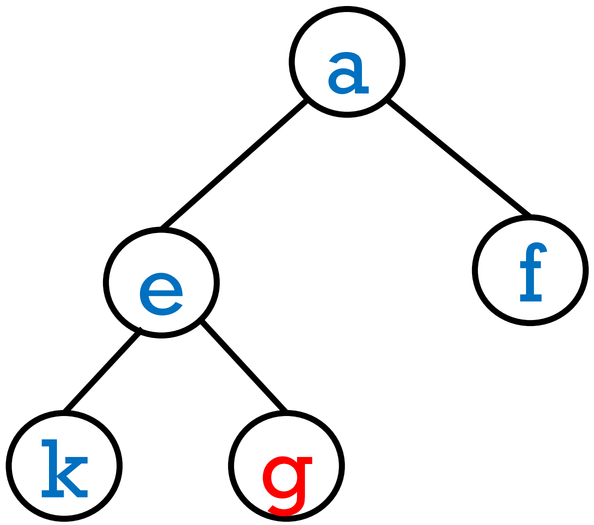
HOW TO BUILD A HEAP?

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)



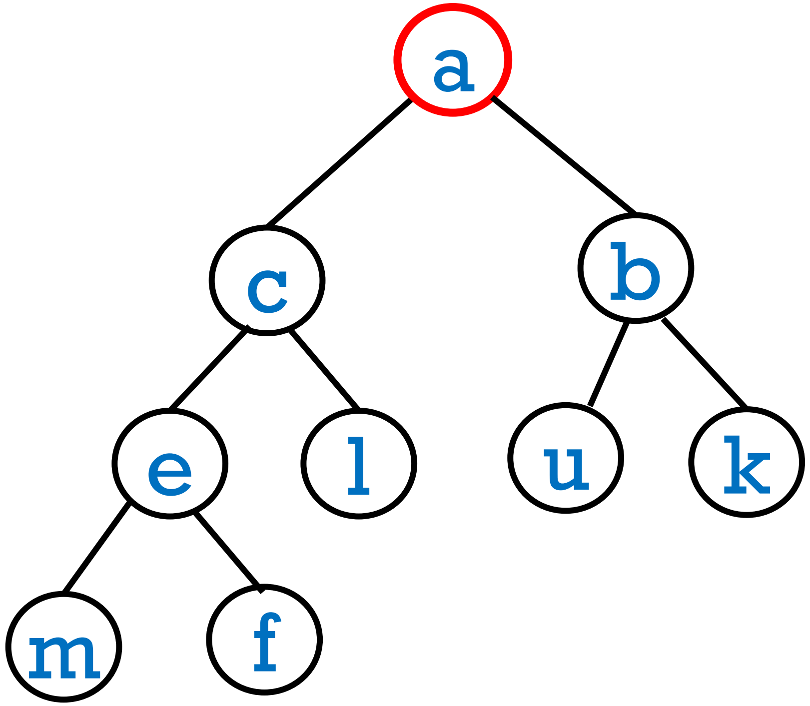
This method of building a heap is slow.

We will see a faster method later

REMOVE

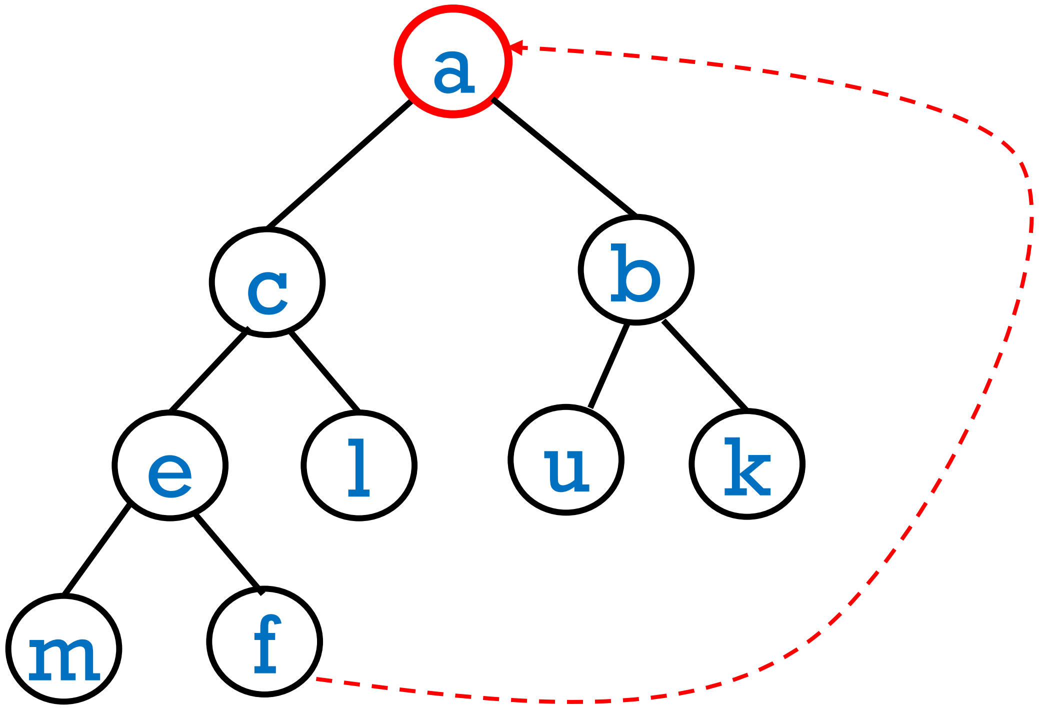
REMOVEMIN

returns root element

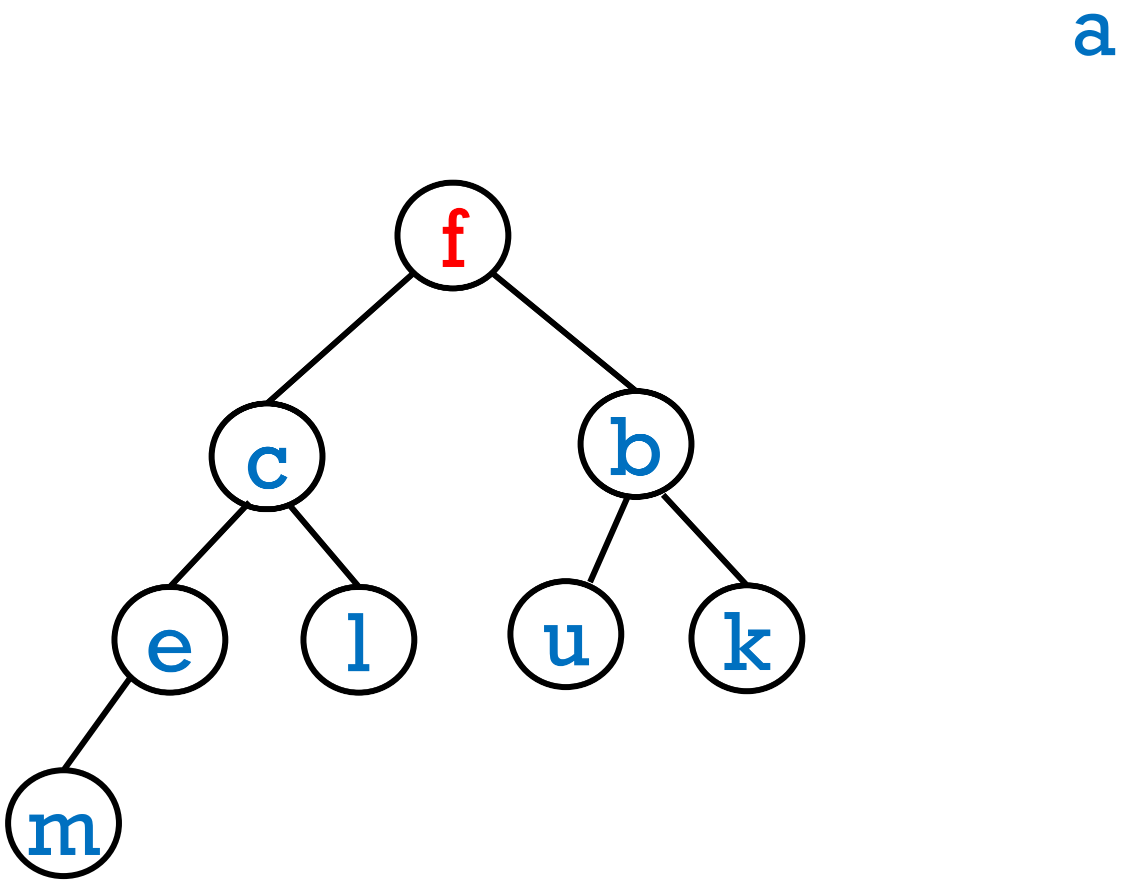


REMOVEMIN

returns root element



REMOVEMIN



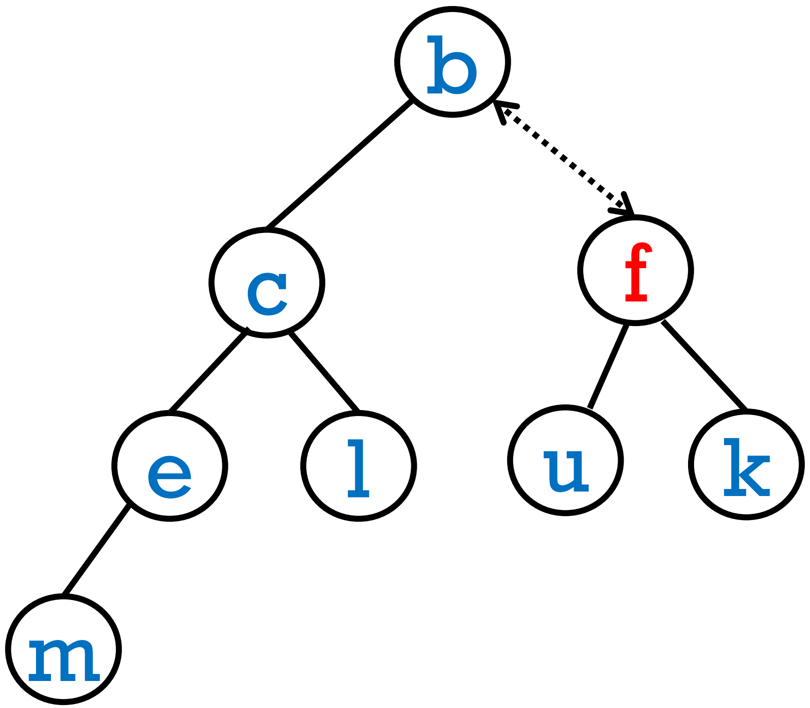
Claim: if the root has two children, then the new root *will* be greater than at least one of its children.

Why?

How to solve this problem?

REMOVEMIN

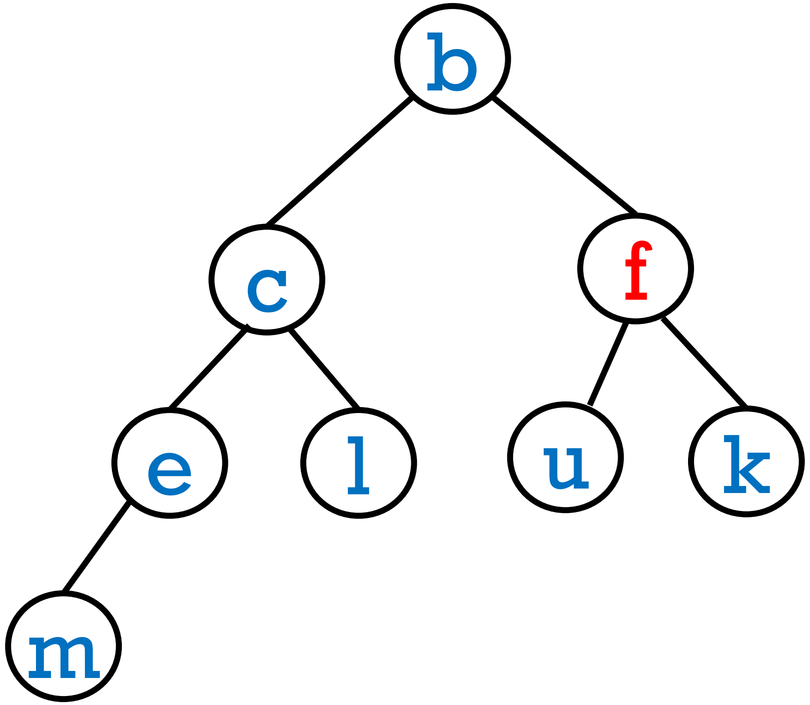
Swap keys with the smaller child!



REMOVEMIN

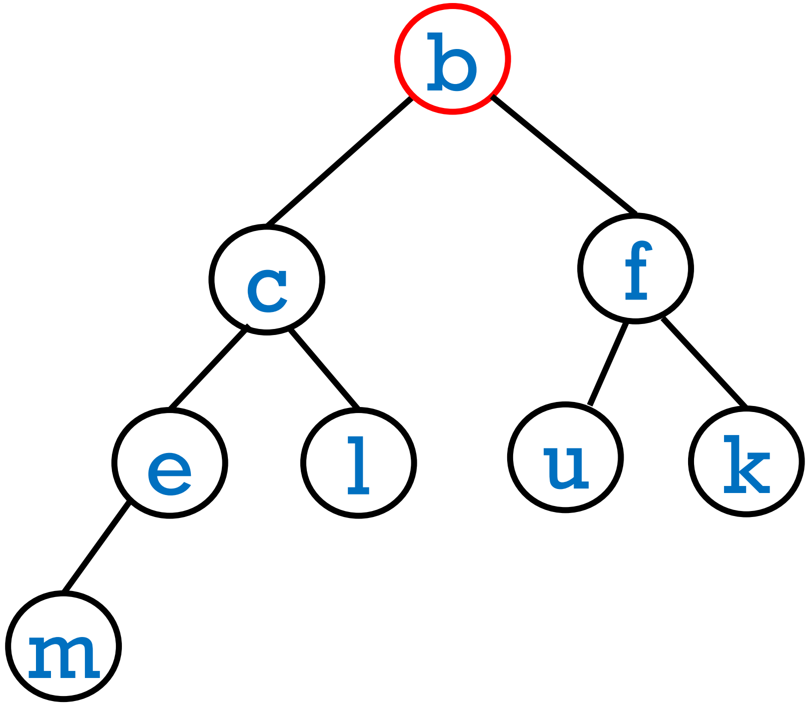
Swap keys with the smaller child!

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



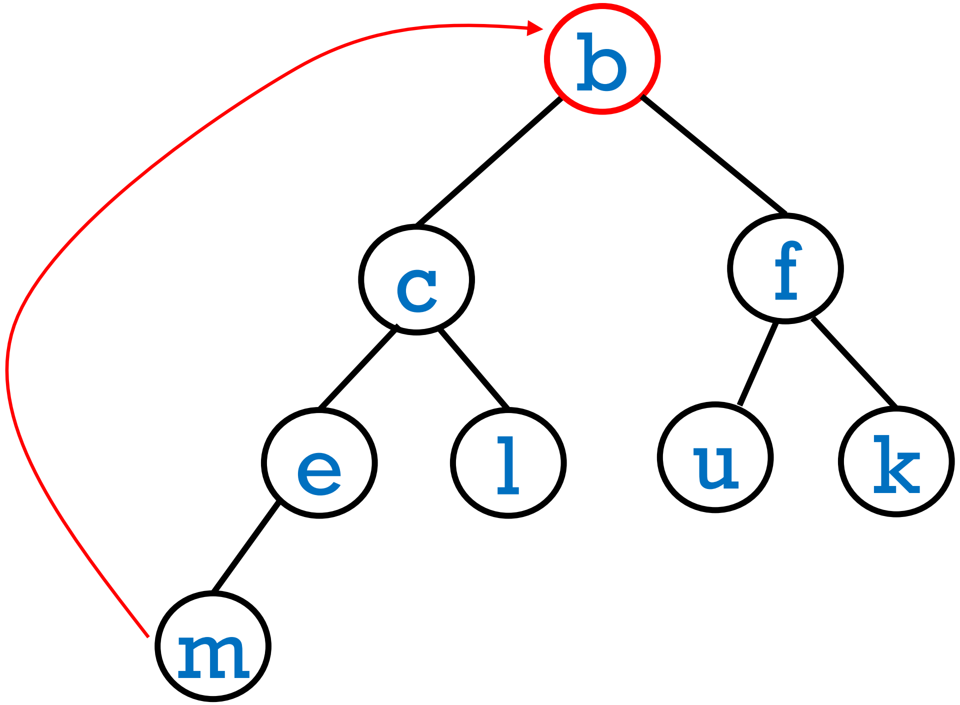
REMOVEMIN

Let's removeMin() again!



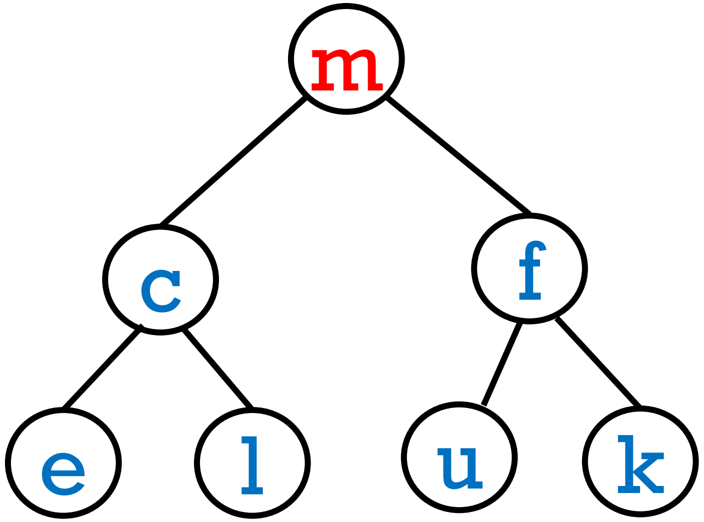
REMOVEMIN

Let's removeMin() again!



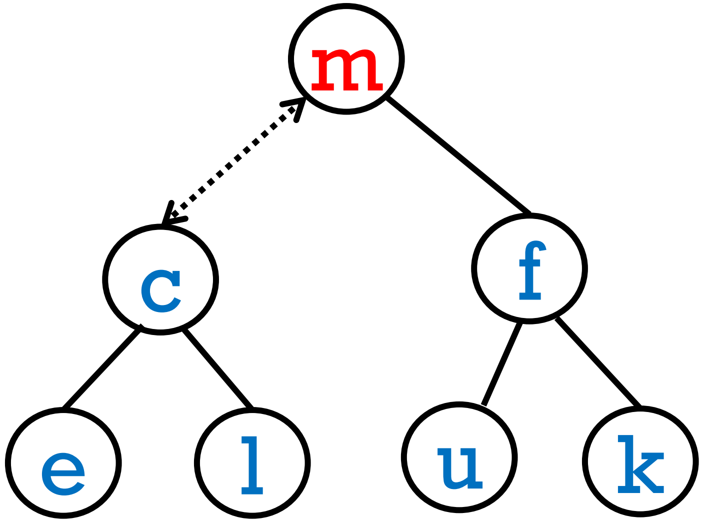
REMOVEMIN

Let's removeMin() again!



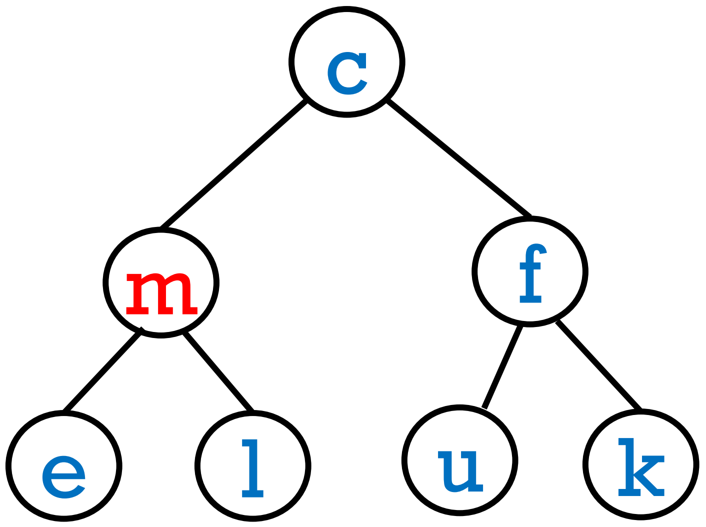
REMOVEMIN

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



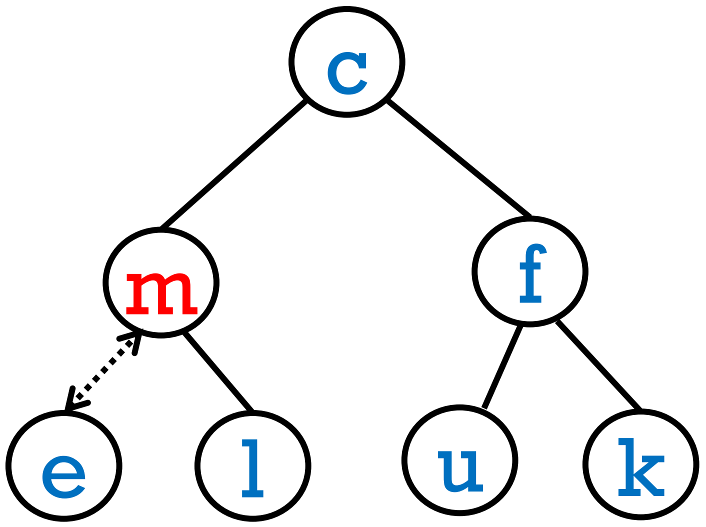
REMOVEMIN

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



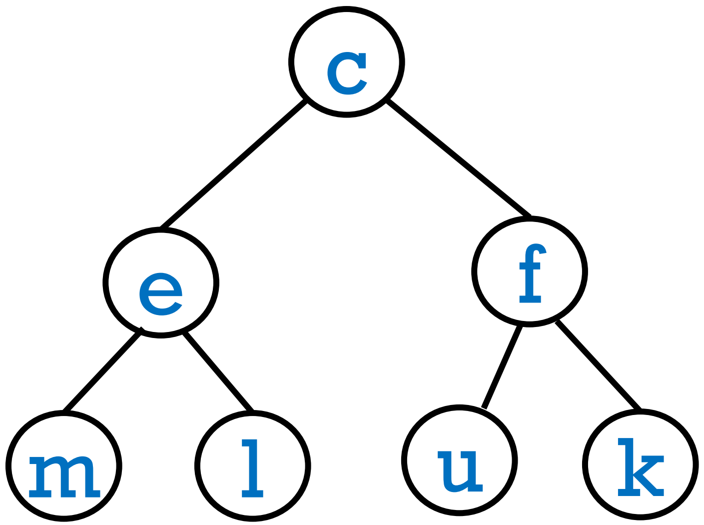
REMOVEMIN

Keep swapping with smaller child, if necessary.

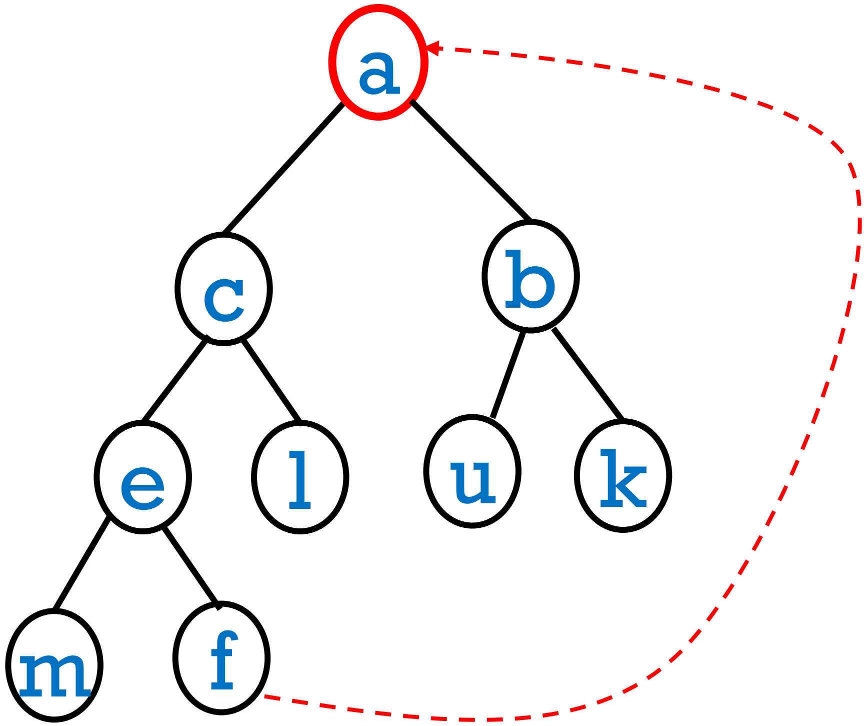


REMOVEMIN

Keep swapping with smaller child, if necessary.

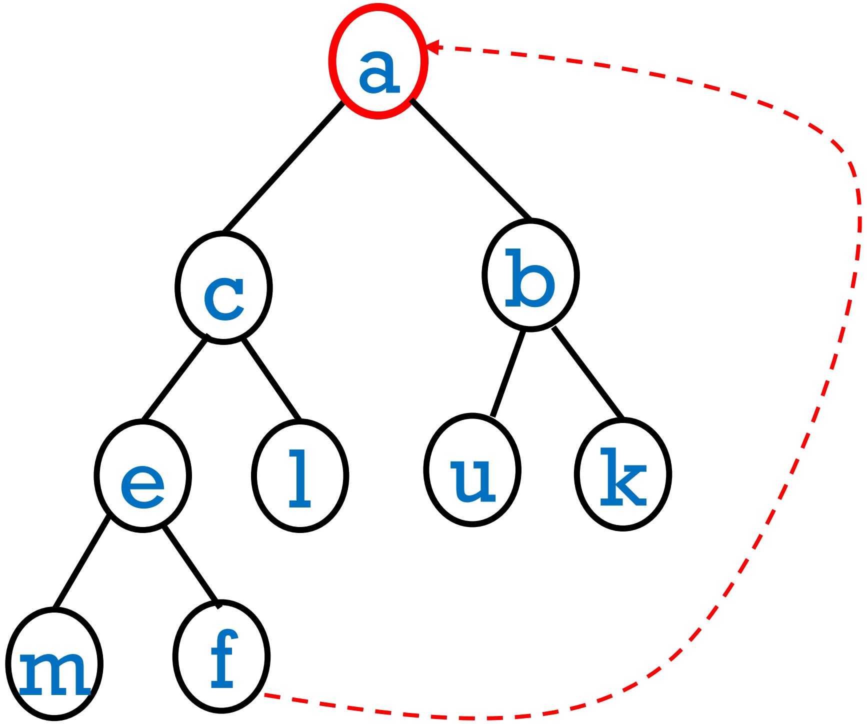


REMOVEMIN() - IMPLEMENTATION



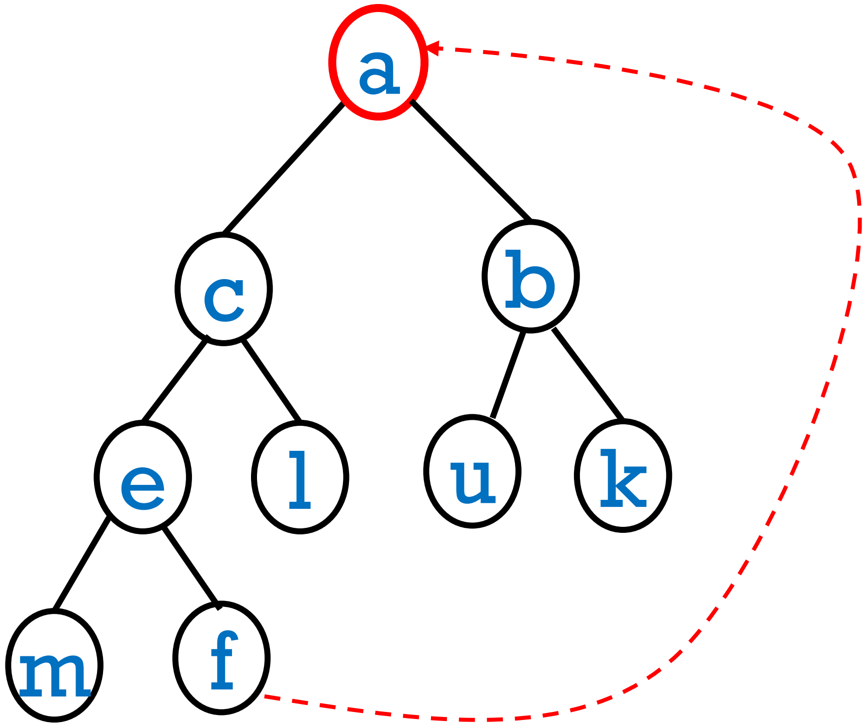

```
removeMin(){
   temp = root.key
   remove the last leaf node and
   store its key into the root
   cur = root




    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
}
```

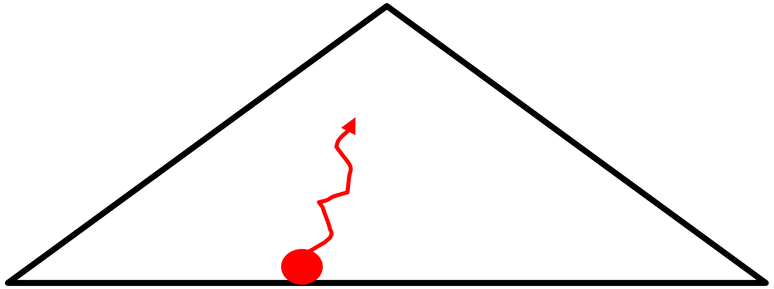
REMOVEMIN() - IMPLEMENTATION




```
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
}
```

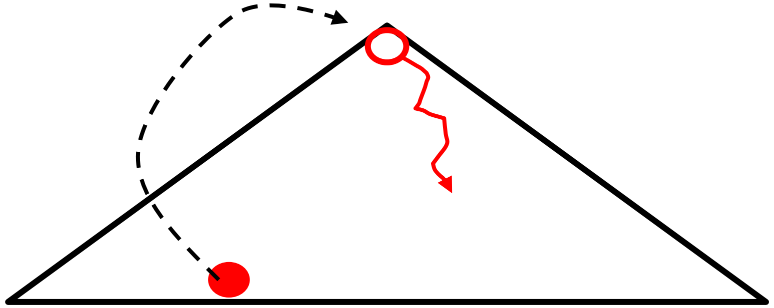
HEAPIFY

add()



“upHeap”

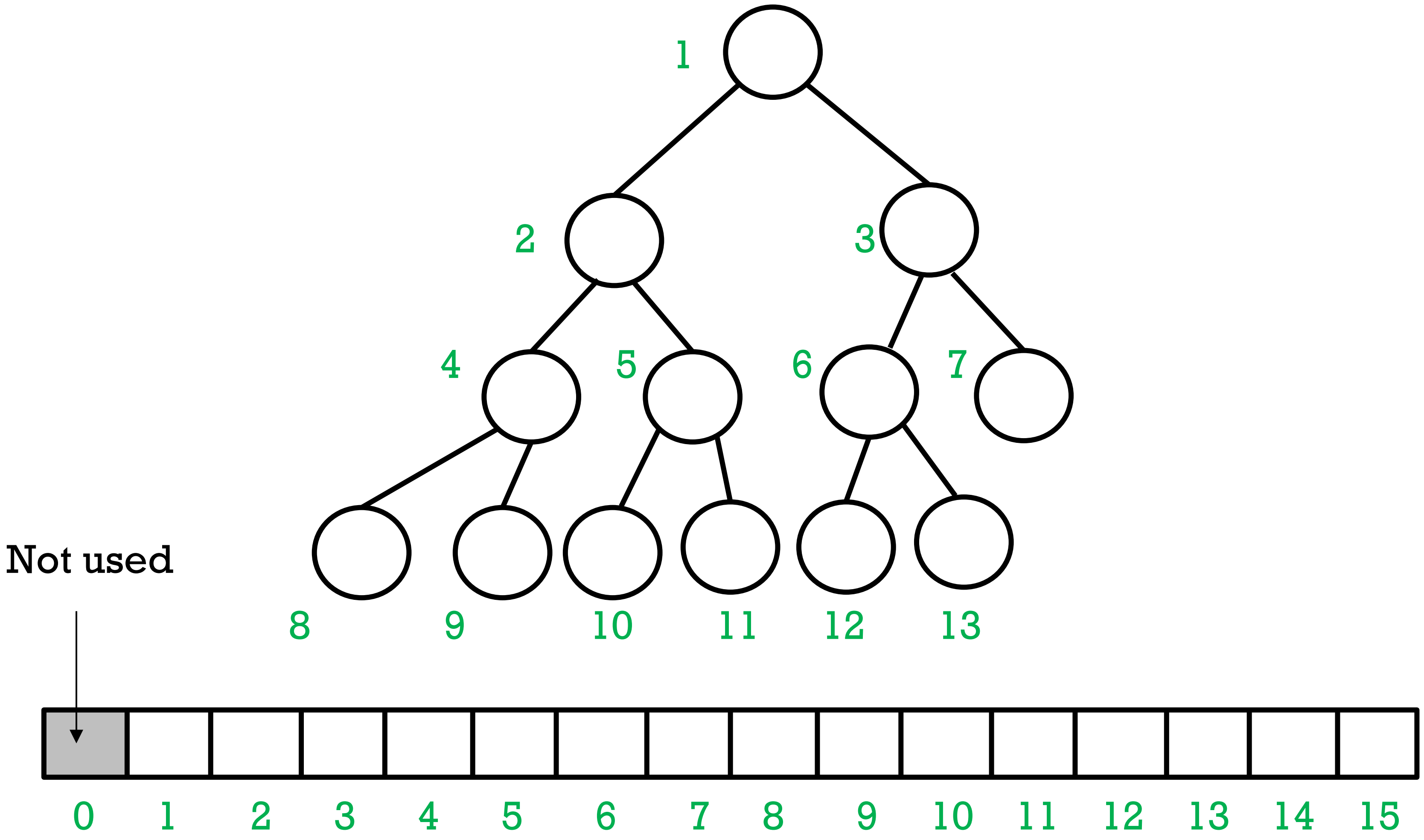
removeMin()



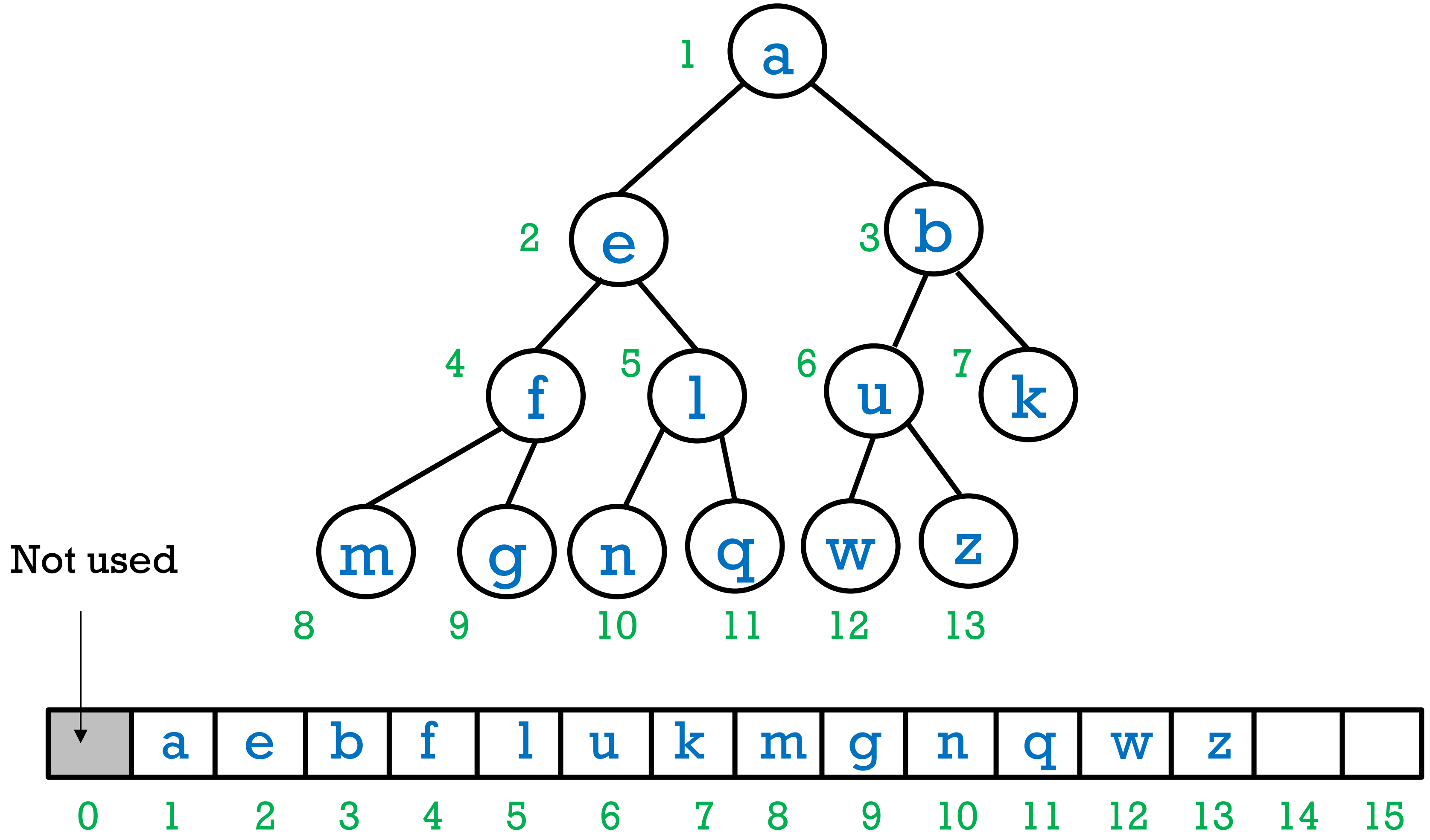
“downHeap”

HEAPS ARRAY IMPLEMENTATION

HEAP (ARRAY IMPLEMENTATION)

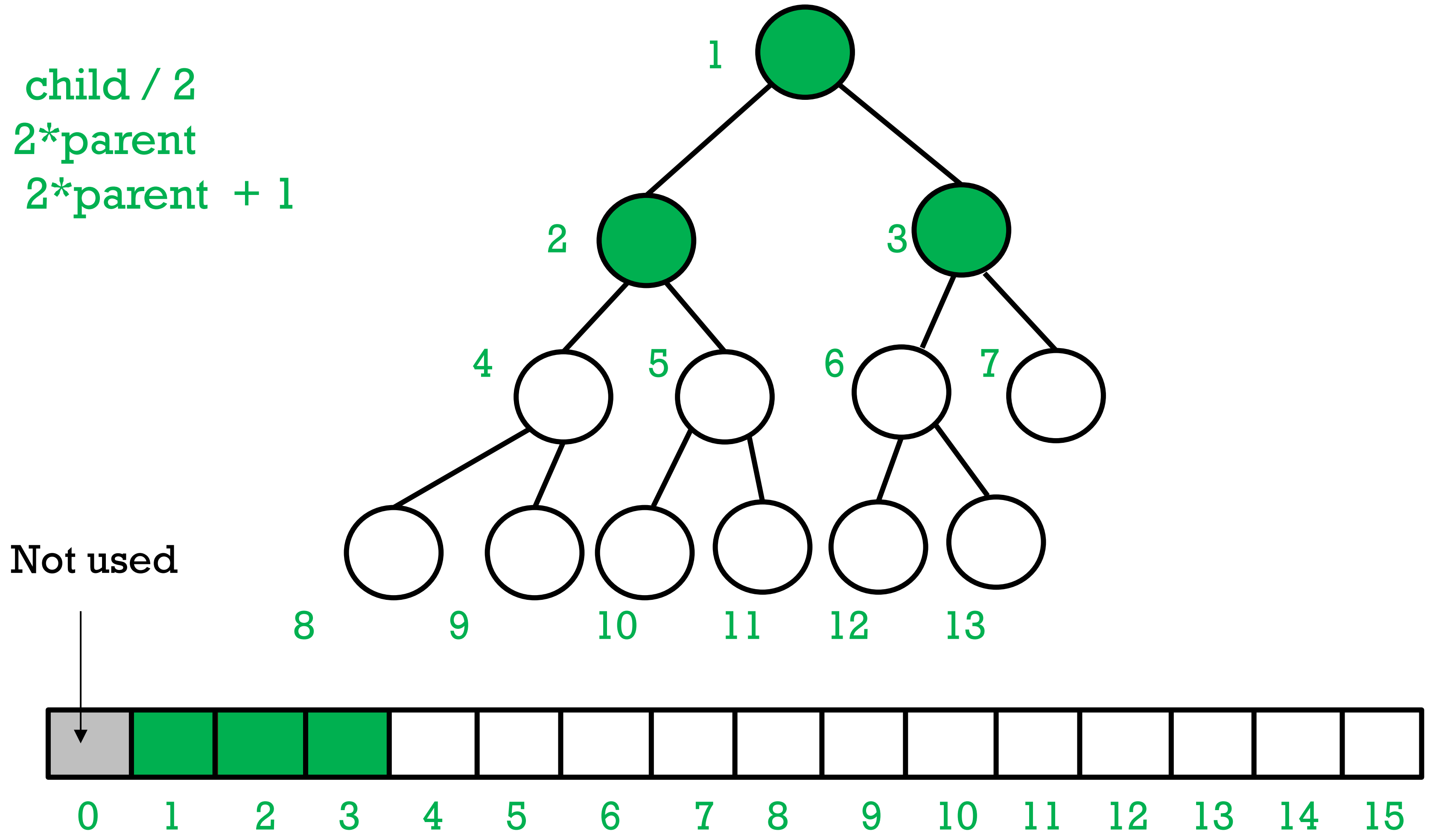


HEAP (ARRAY IMPLEMENTATION)



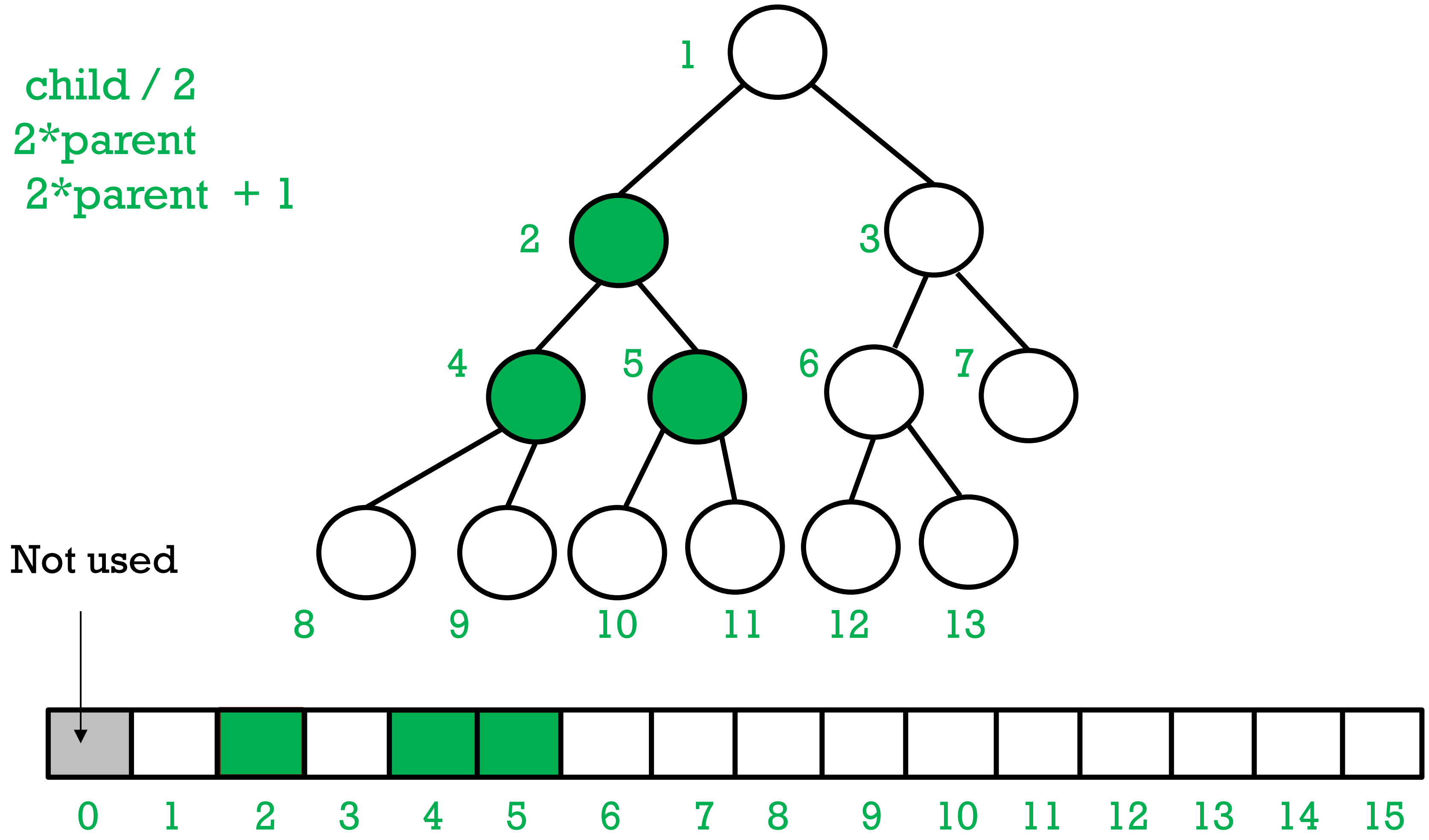
HEAP INDEX RELATIONS

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



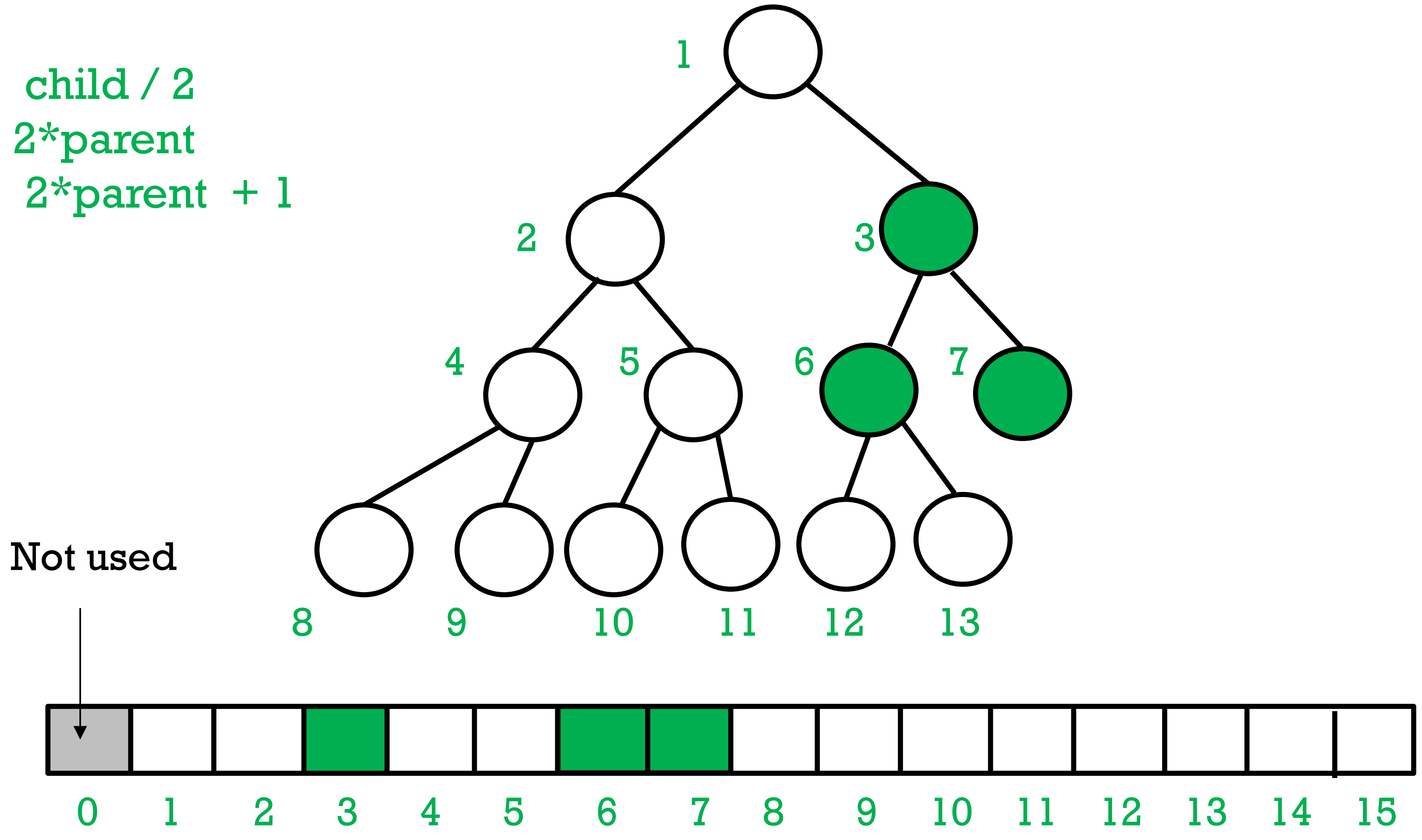
HEAP INDEX RELATIONS

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



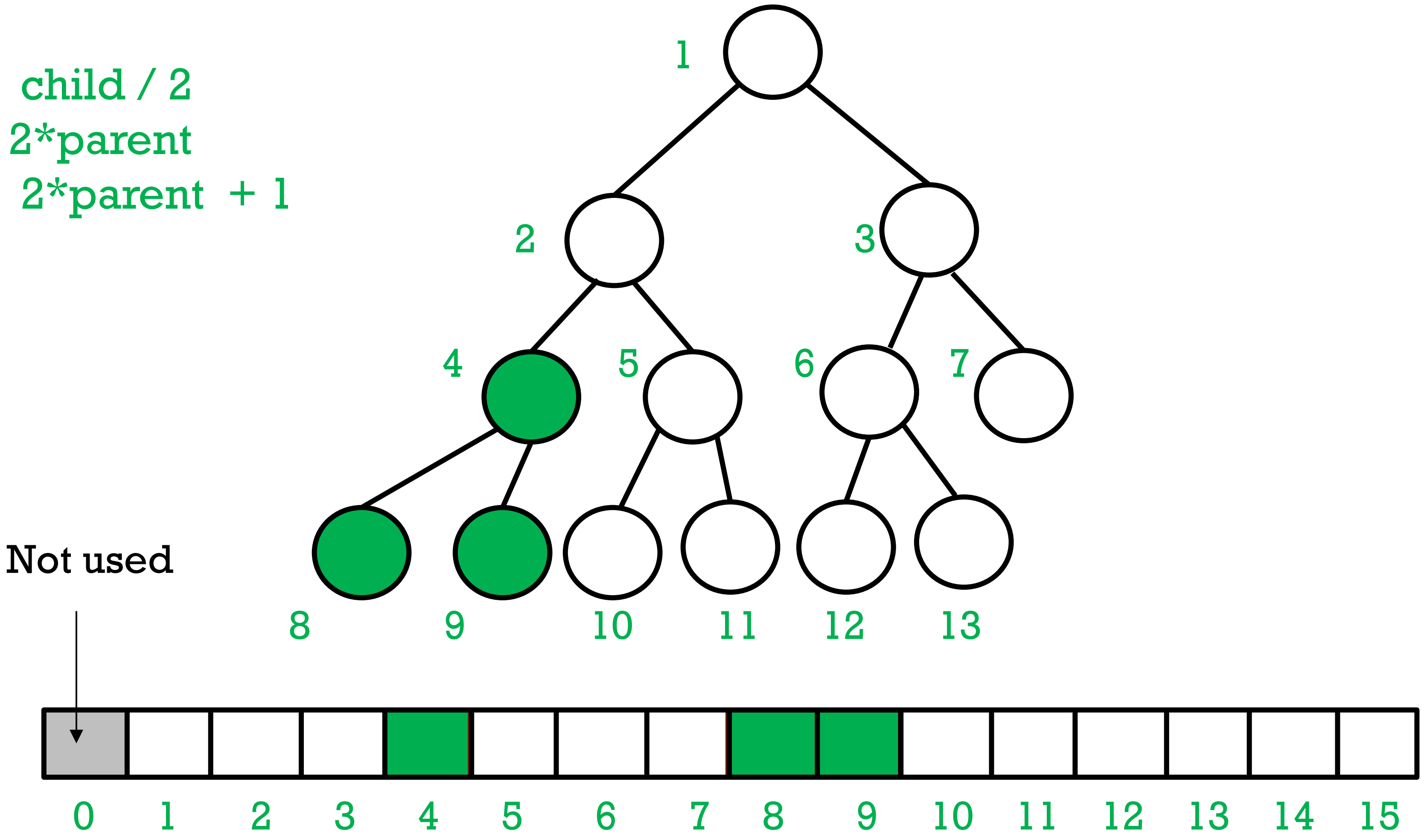
HEAP INDEX RELATIONS

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

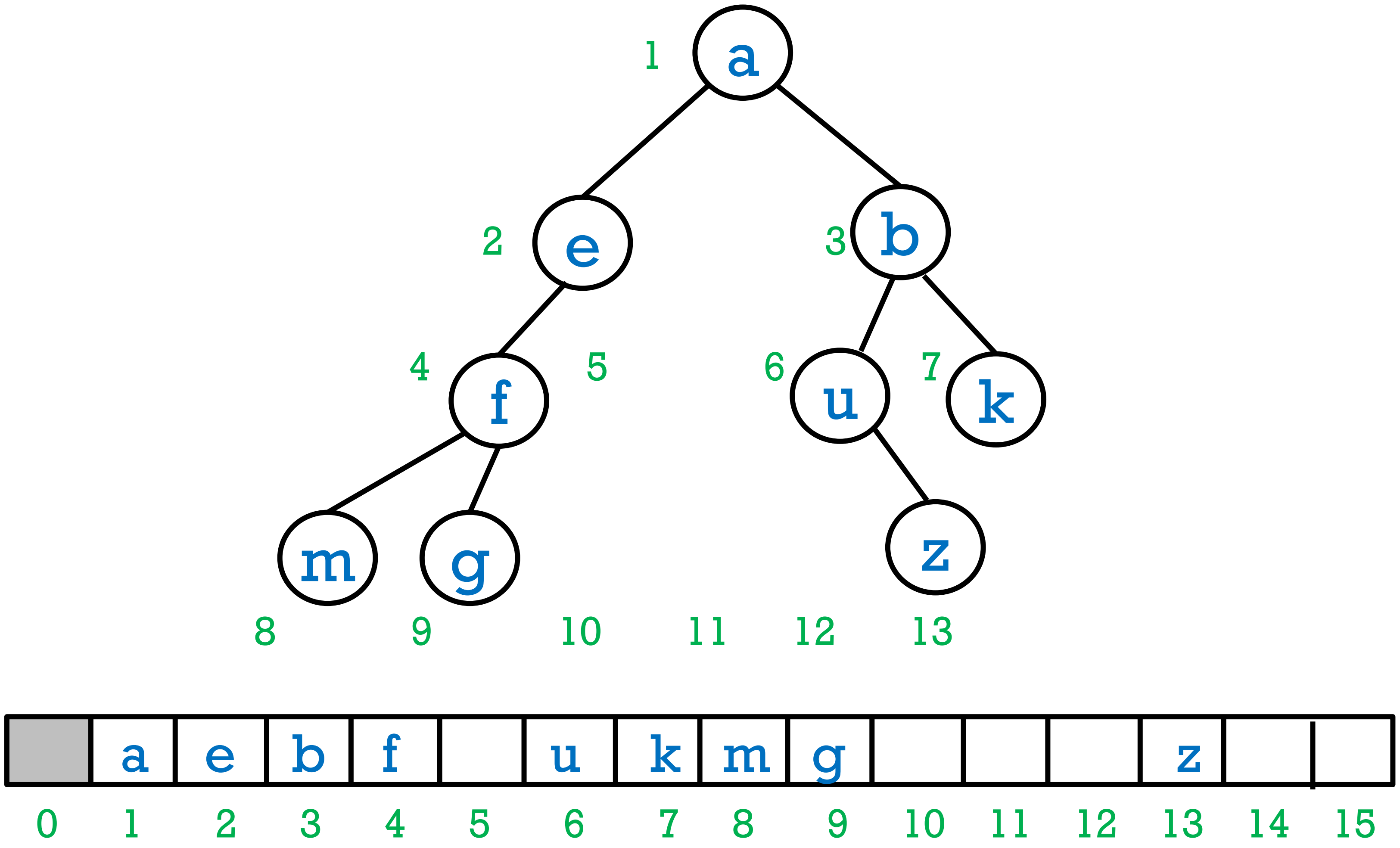


HEAP INDEX RELATIONS

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



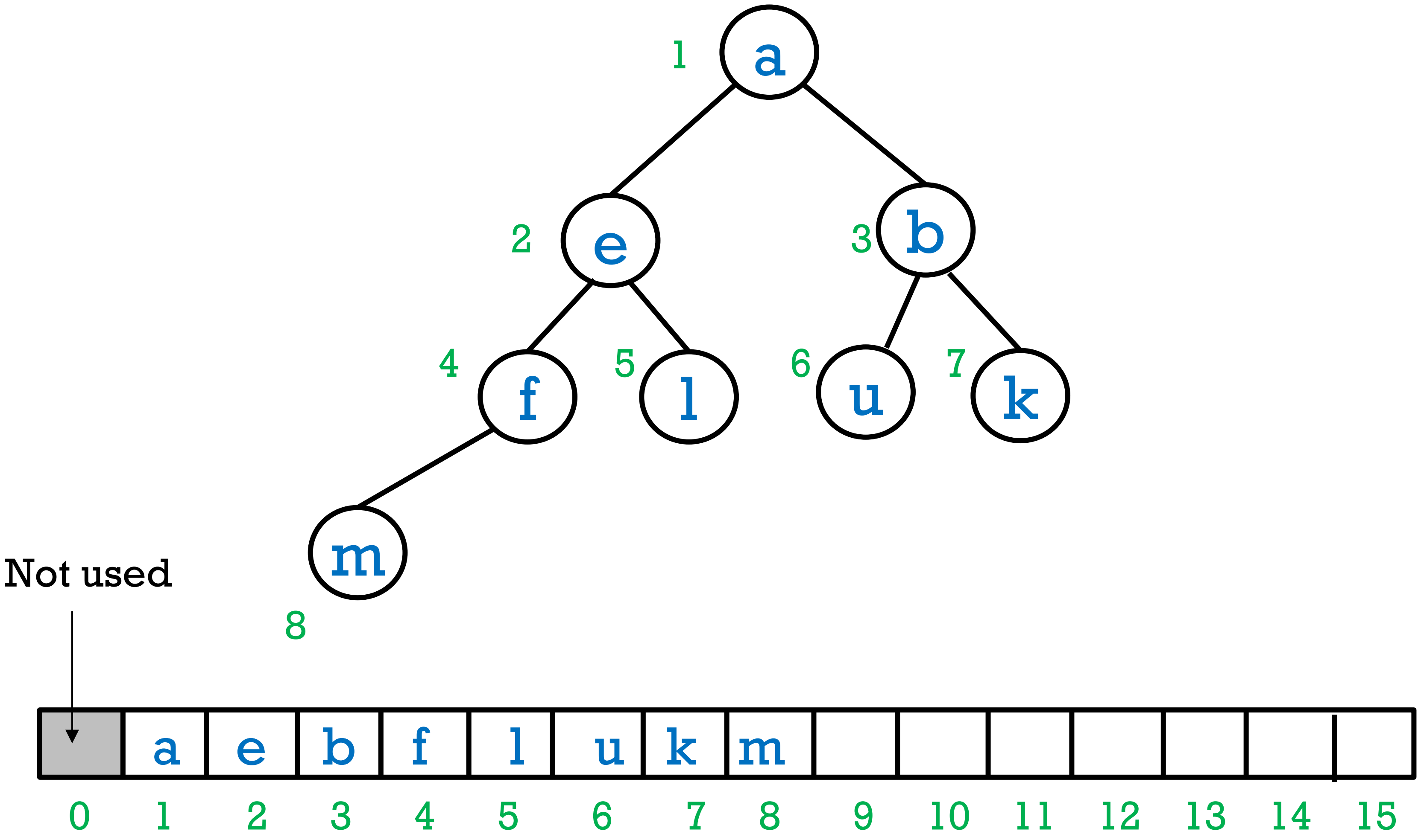
ASIDE: an array data structure can be used for *any* binary tree. But this is uncommon and often inefficient.



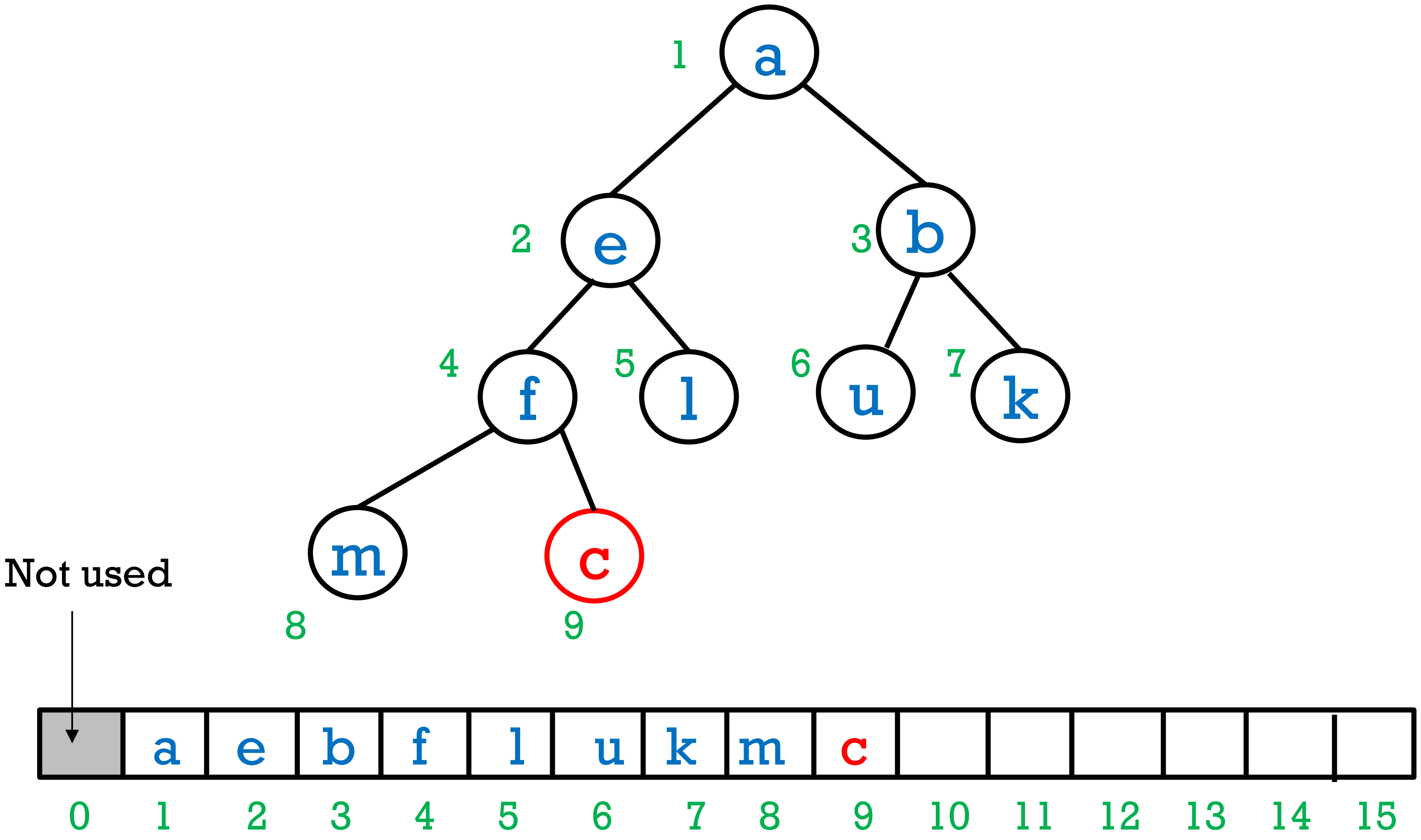
ADD() - IMPLEMENTATION

```
add(key){
   size = size + 1  // number of elements in heap

   // assuming array has room for another element
   heap[ size ] = key

   /*** heapify ***/
   i = size // start from the new node
   while ( i > 1  &&  heap[i] < heap[ i/2 ]){
      swapElements( i, i/2 ) // swap
      i = i/2 // move to parent
   }
}
```

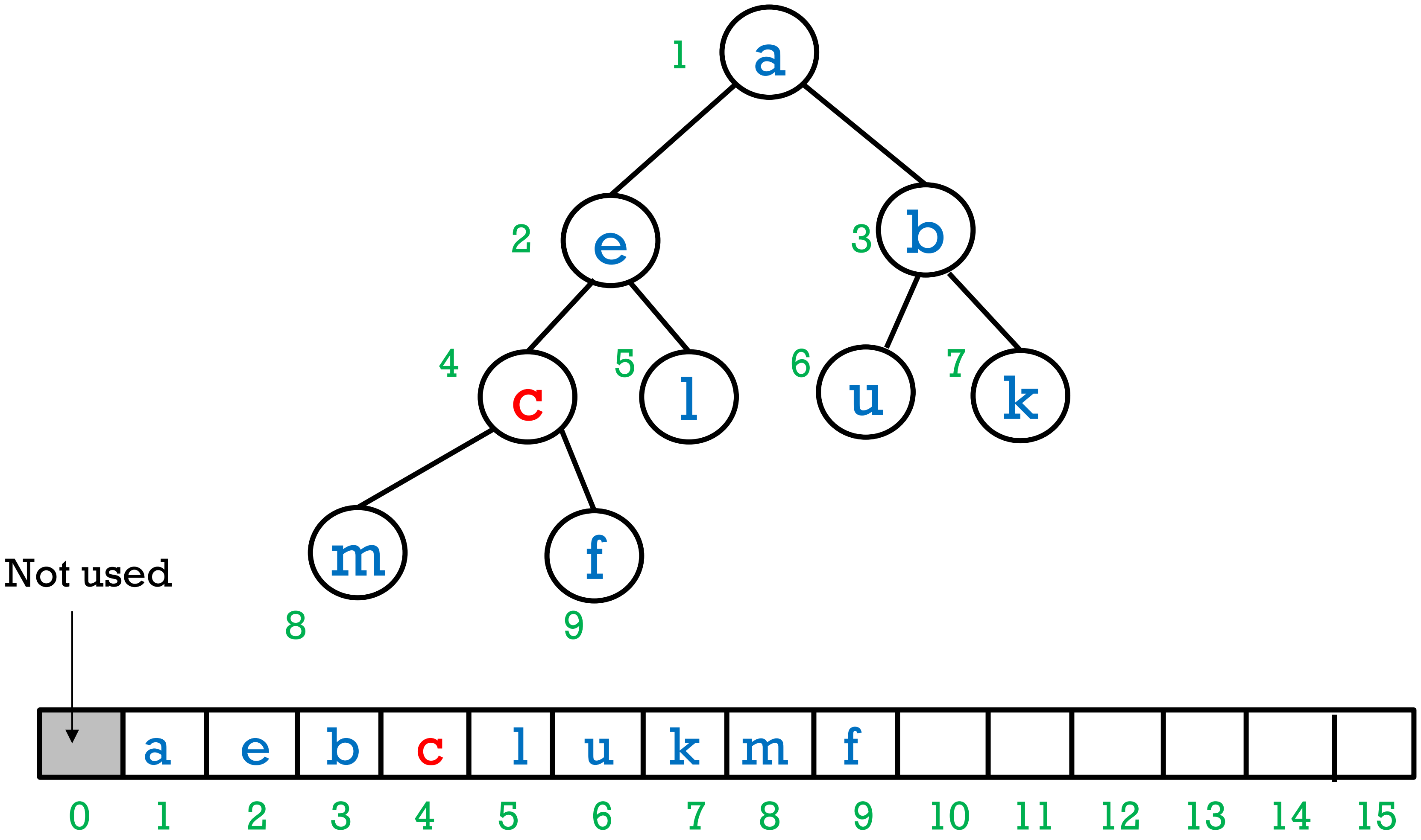
add(c)


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

add(c)

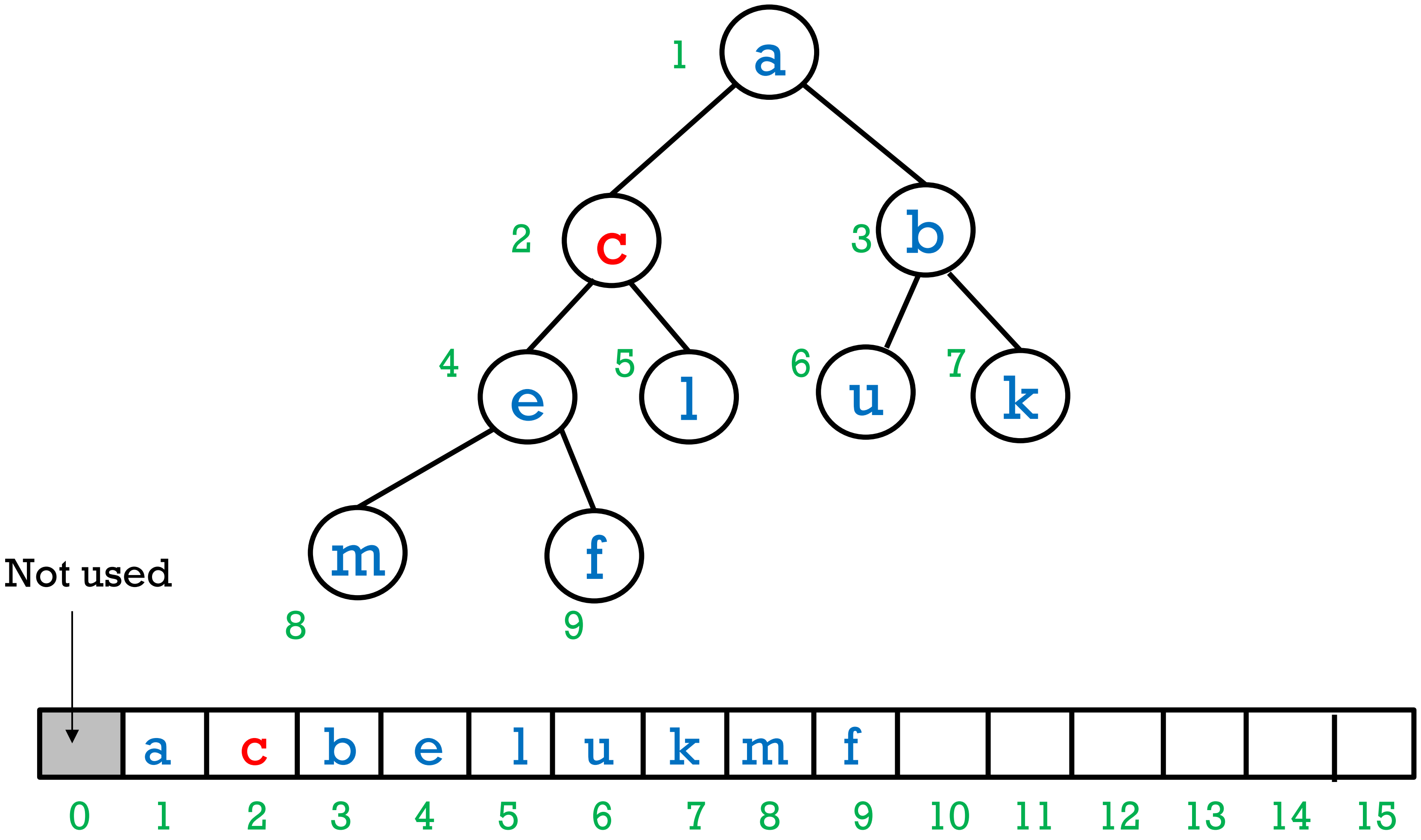

1
a
2
e
3
b
4
f
5
l
6
u
7
k
m
c
8
9
Not used
a
e
b
f
l
u
k
m
c
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

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

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

BUILD A HEAP

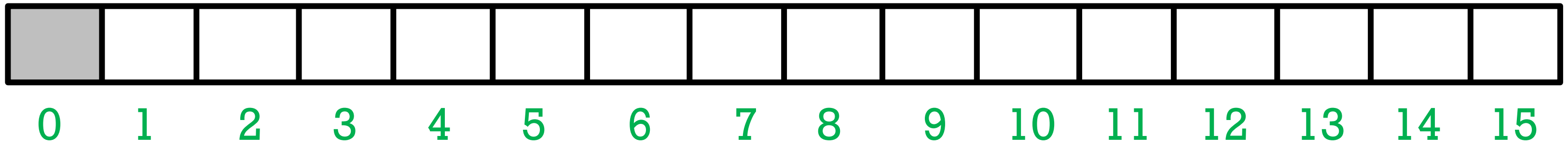
HOW TO BUILD A HEAP?

Suppose we have a list with n elements, we can create an empty heap and use add() to add one element at a time to the heap:

```
buildHeap(list){
   create new heap array //capacity > list.size()
   for (k = 0; k < list.size(); k++)
      add( list[k] )  // add the element to the heap
}
```

Note that you could write the buildHeap algorithm slightly differently by putting all the list elements into the array at the beginning, and then `upheaping' each one.

BEST CASE OF BUILDING A HEAP IS … ?



Suppose we want to add some elements to an empty heap:

a c b e l u k m f

How many swaps do we need to add each element?

In the best case, …

BEST CASE OF BUILDING A HEAP IS O(N)

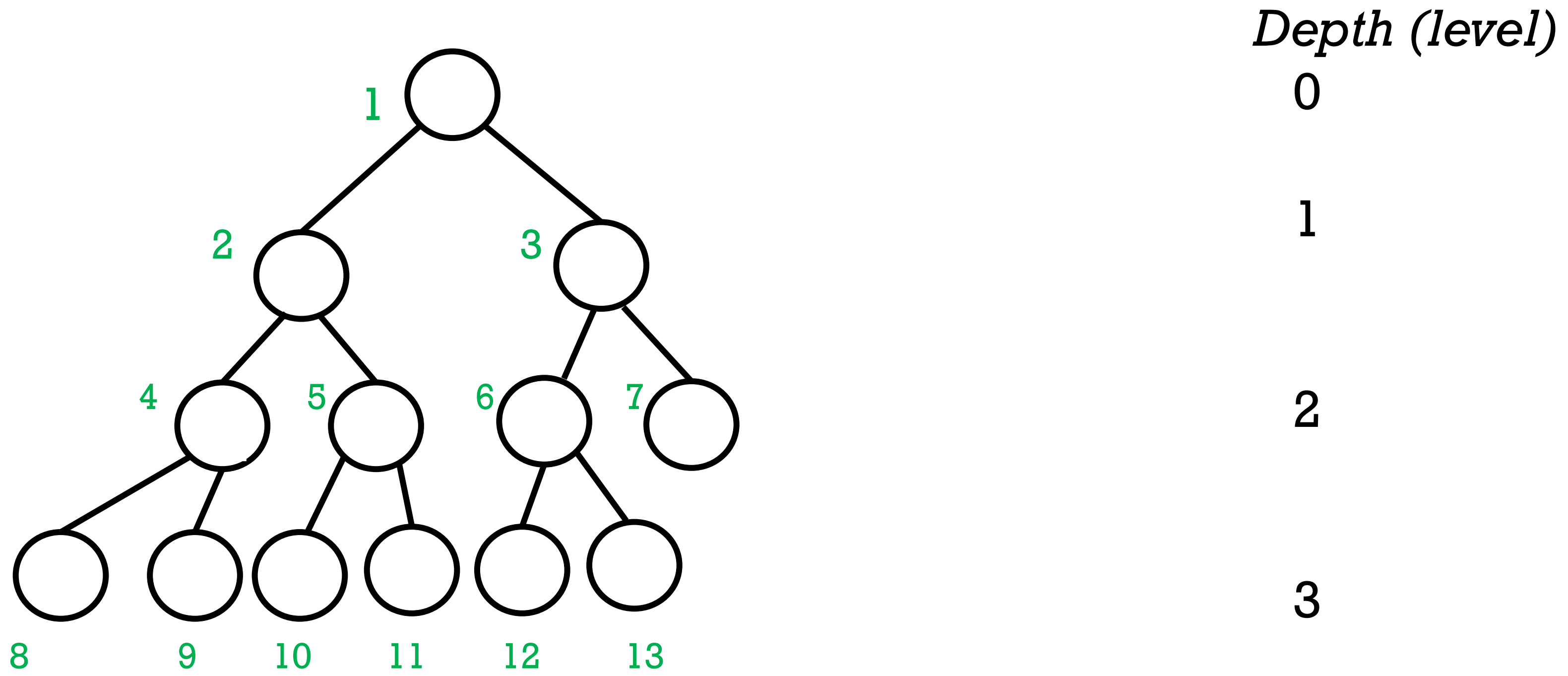
	a	c	b	e	l	u	k	m	f						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Suppose we want to add some elements to an empty heap:
a c b e l u k m f

How many swaps do we need to add each element?

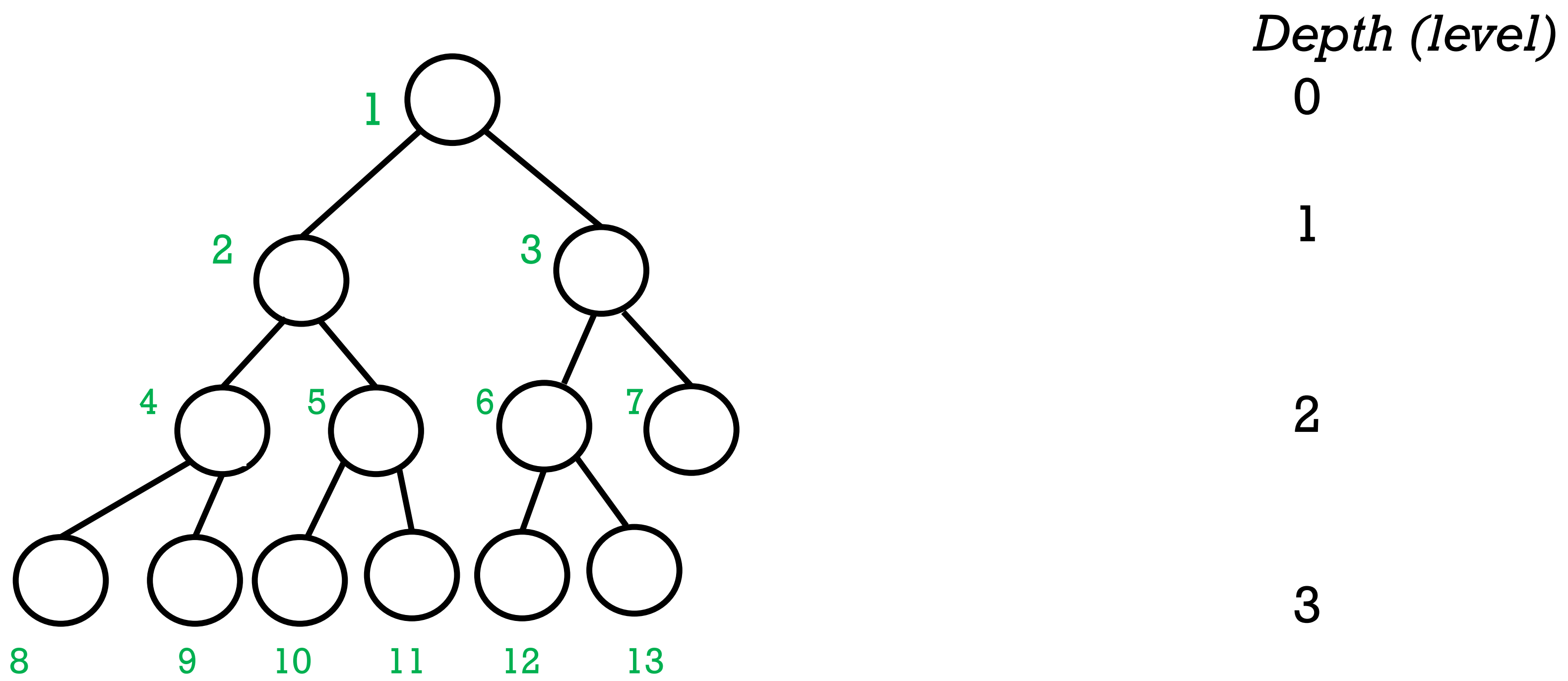
In the best case, the order of elements that we add is already a heap, and no swaps are necessary.

WORST CASE OF BUILDING A HEAP IS … ?



How many swaps do we need to add the i-th element?

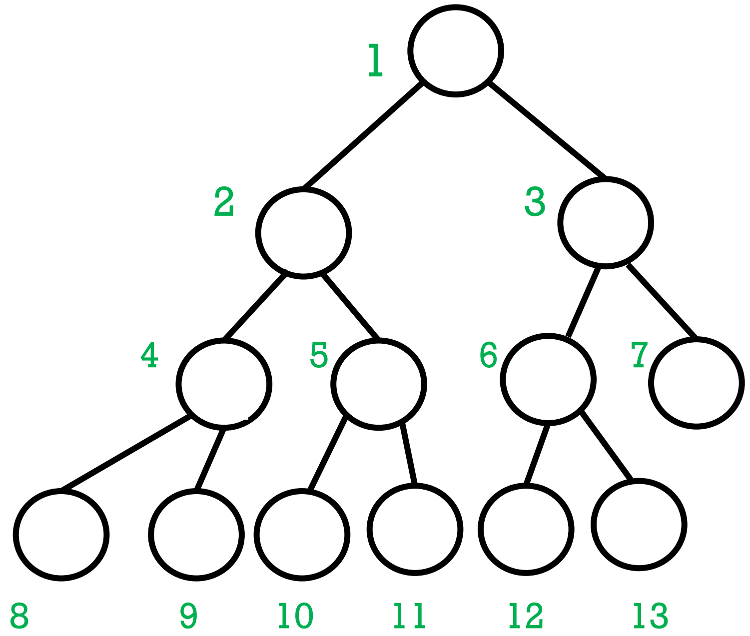
WORST CASE OF BUILDING A HEAP IS … ?



How many swaps do we need to add the i-th element?
Element i gets added to some $level$, such that:

$$2^{level} \leq i < 2^{level+1}$$

WORST CASE OF BUILDING A HEAP IS … ?

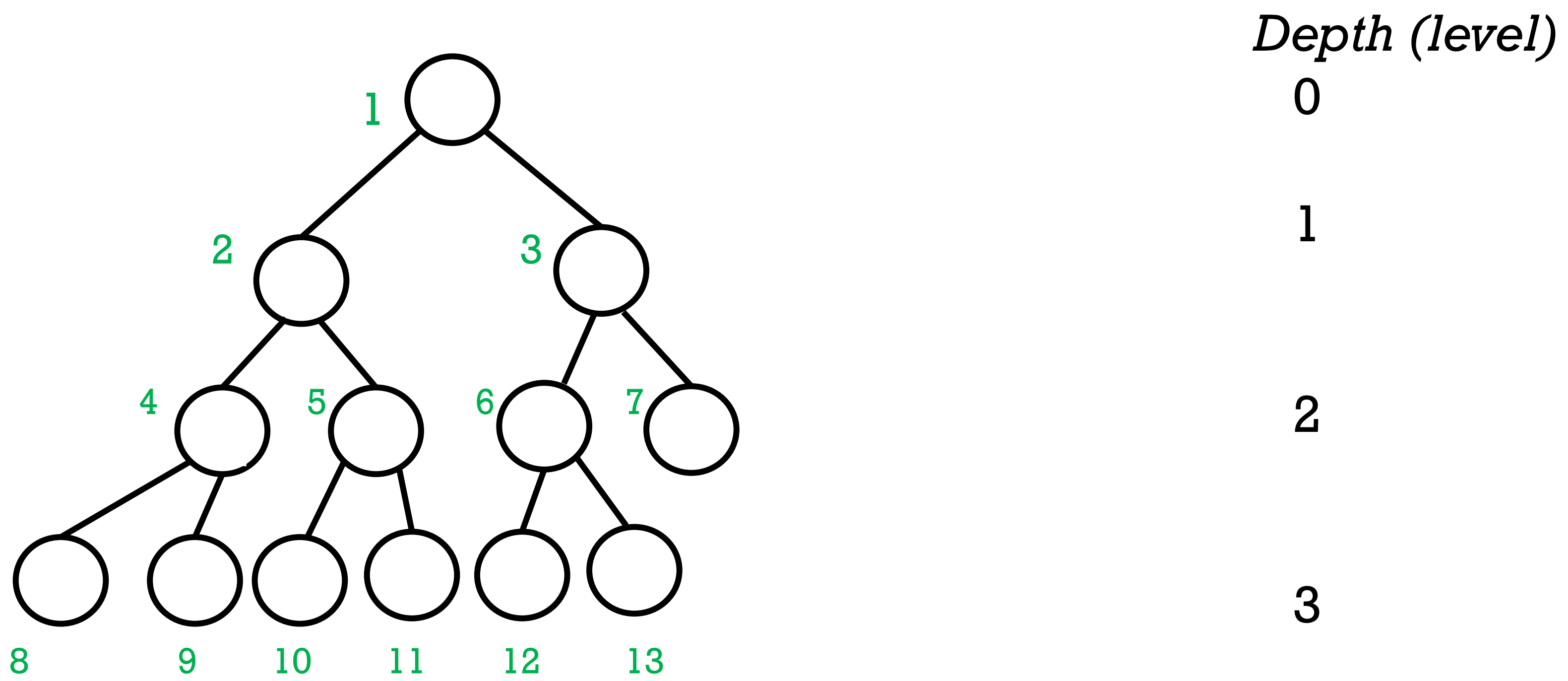


Depth (level)
0
1
2
3

$$2^{level} \leq i < 2^{level+1}$$
$$level \leq \log_2 i < level + 1$$

Thus, $level = floor(\log_2 i)$

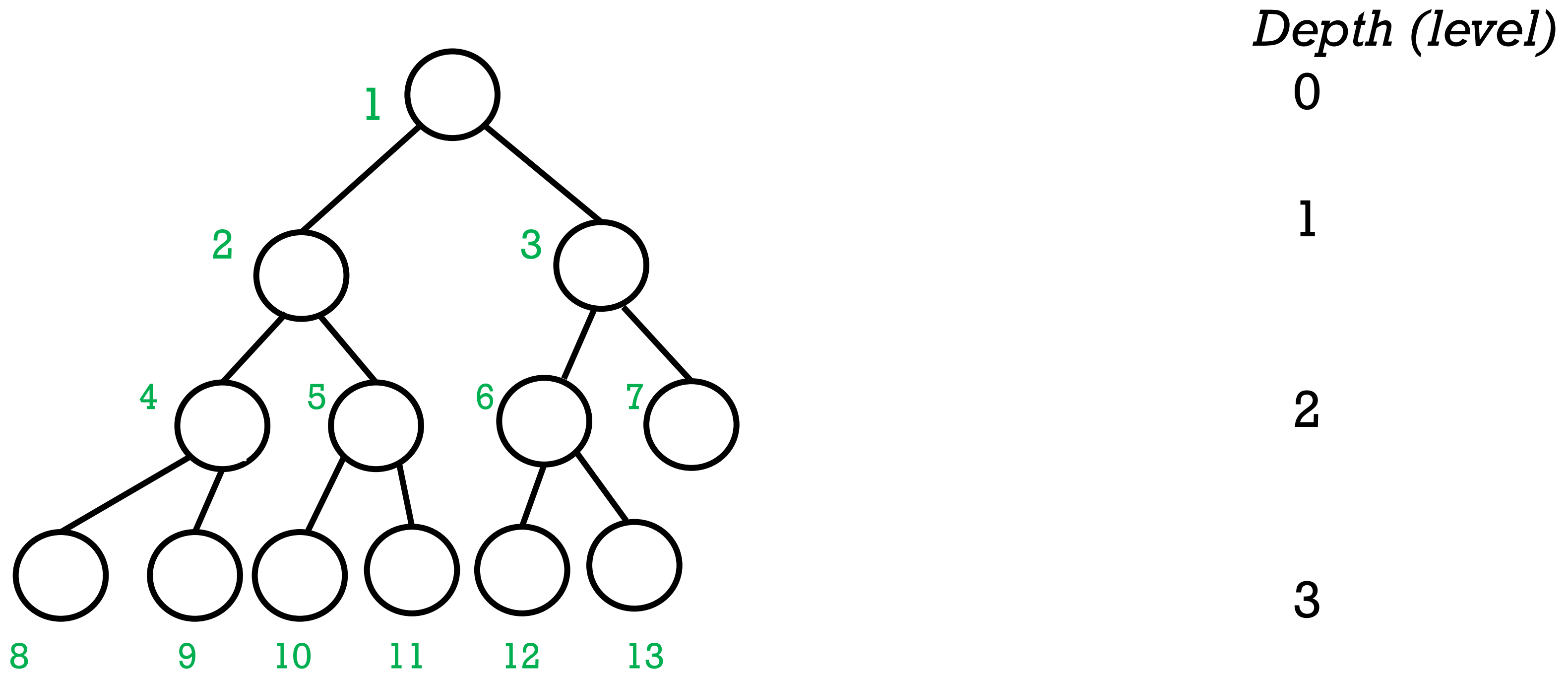
WORST CASE OF BUILDING A HEAP IS ... ?



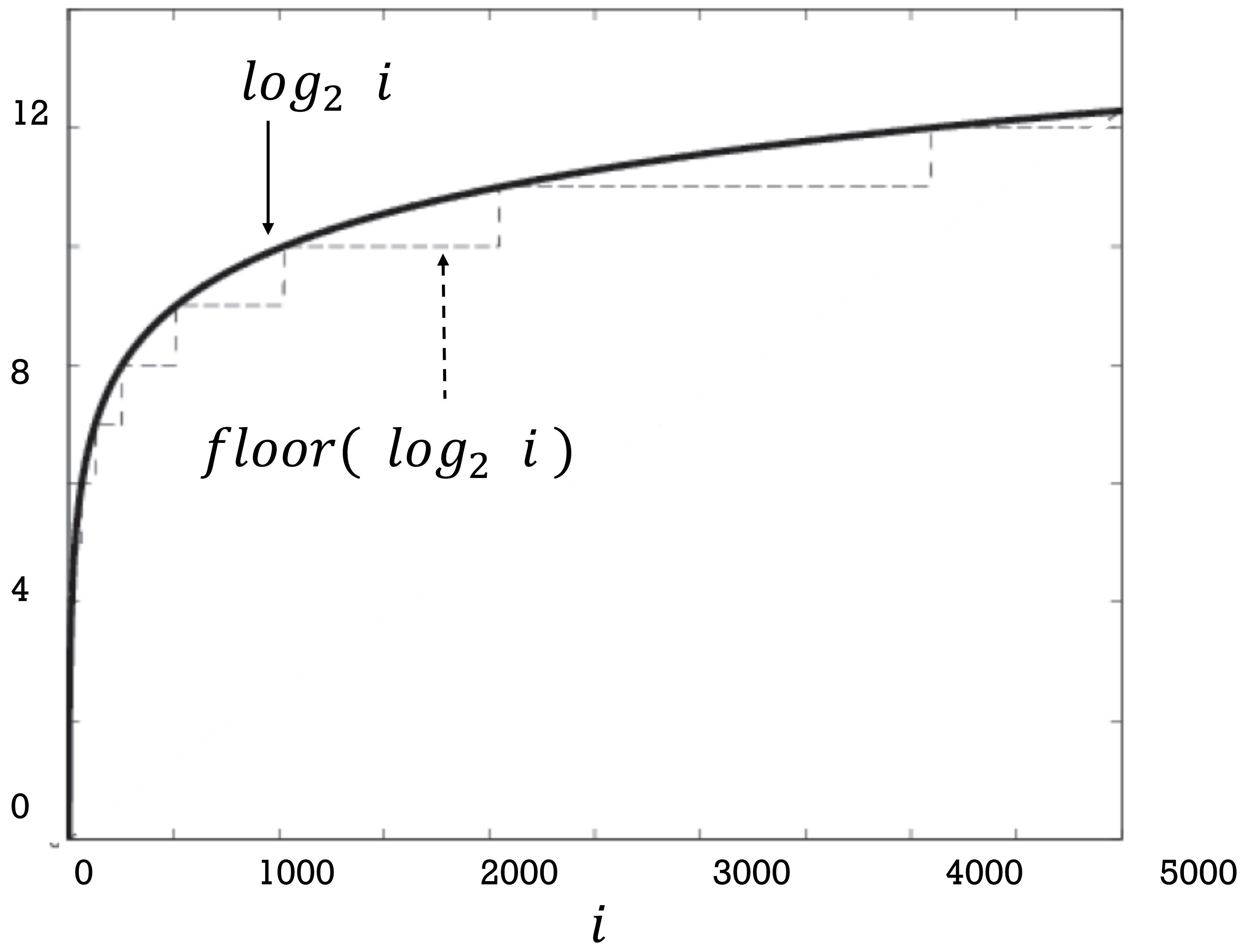
Suppose there are n elements to add, then in the worst case the number of swaps needed to add all the elements is:

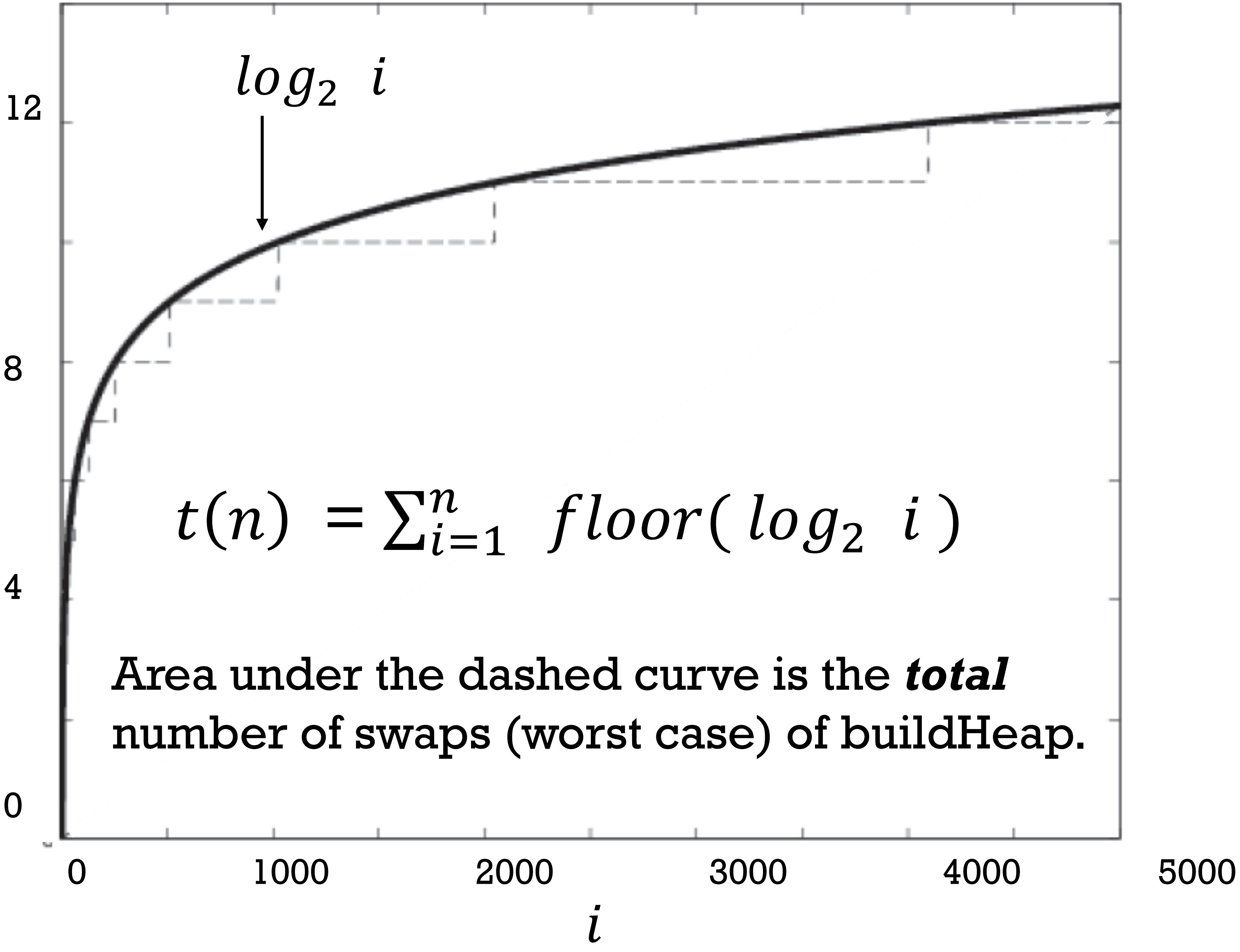
$$t(n) = \sum_{i=1}^{n} floor(\log_2 i)$$

WORST CASE OF BUILDING A HEAP



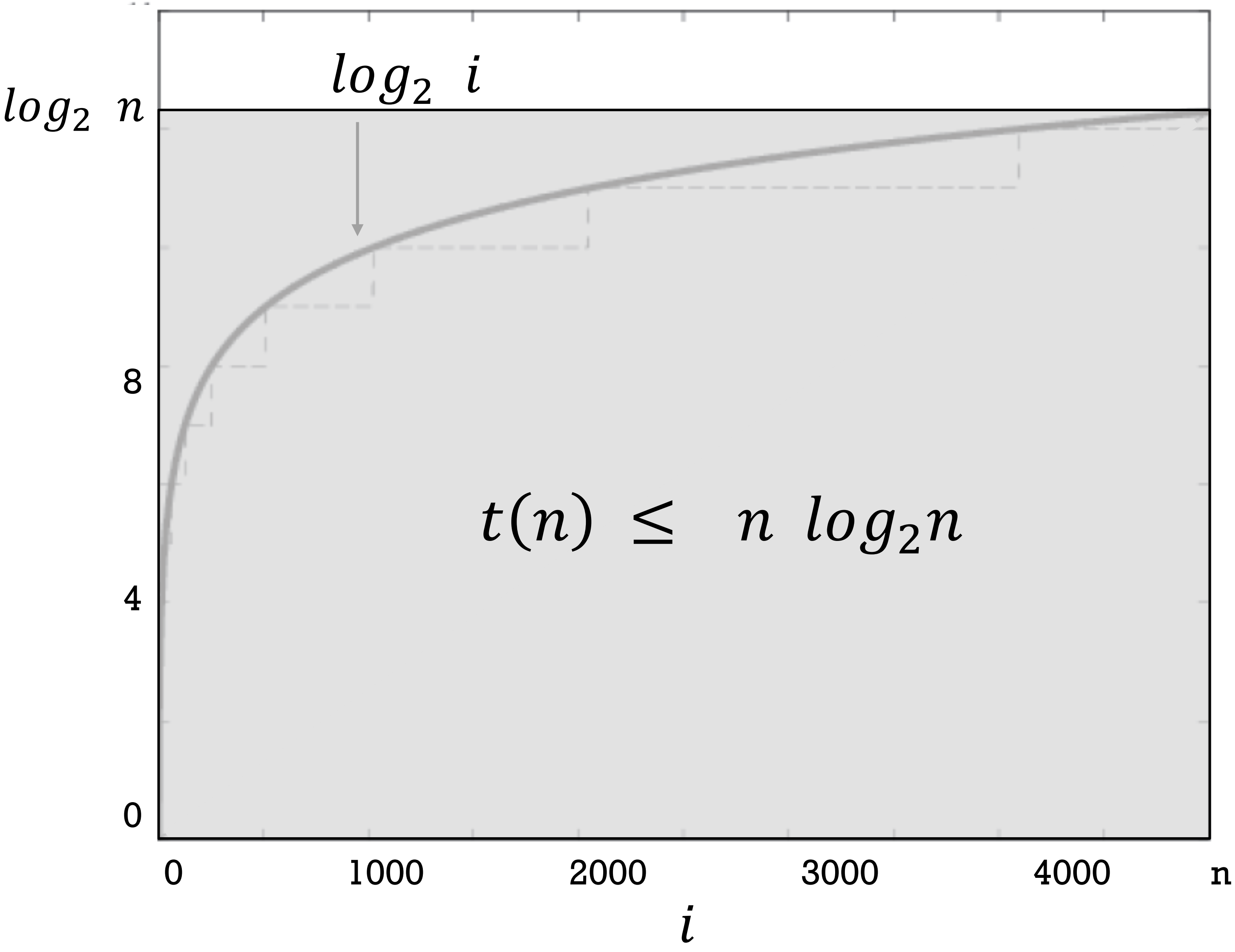
Thus, in the worst case scenario for buildHeap() is $O(n * \log n)$

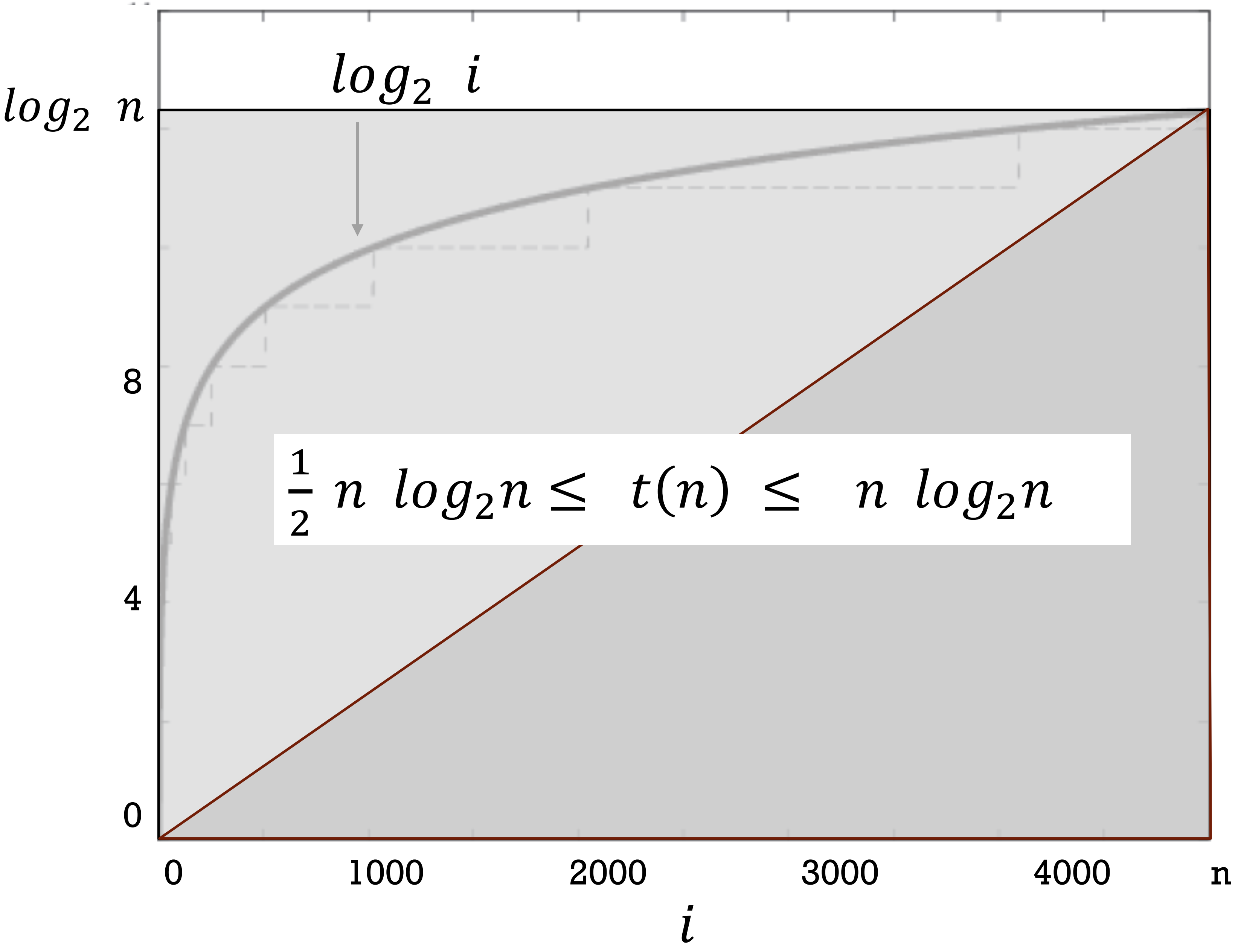

$log_2\ i$
$floor(\ log_2\ i\ )$
12
8
4
0
0
1000
2000
3000
4000
5000
i



$$t(n) = \sum_{i=1}^{n} floor(\, log_2\ i\,)$$

Area under the dashed curve is the ***total*** number of swaps (worst case) of buildHeap.


$log_2\ i$
$log_2\ n$
8
4
0
0
1000
2000
3000
4000
n
i
$t(n) \le n\ log_2 n$

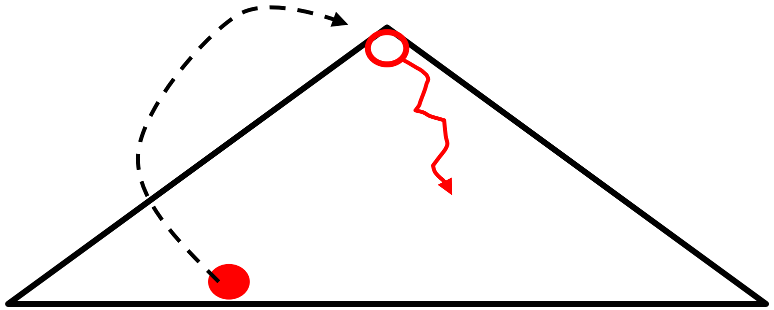

$log_2\ i$
$log_2\ n$
8
4
0
0
1000
2000
3000
4000
n
i
$\frac{1}{2}\, n\ log_2 n \leq\ t(n)\ \leq\ n\ log_2 n$

removeMin()
ARRAY IMPLEMENTATION

add()

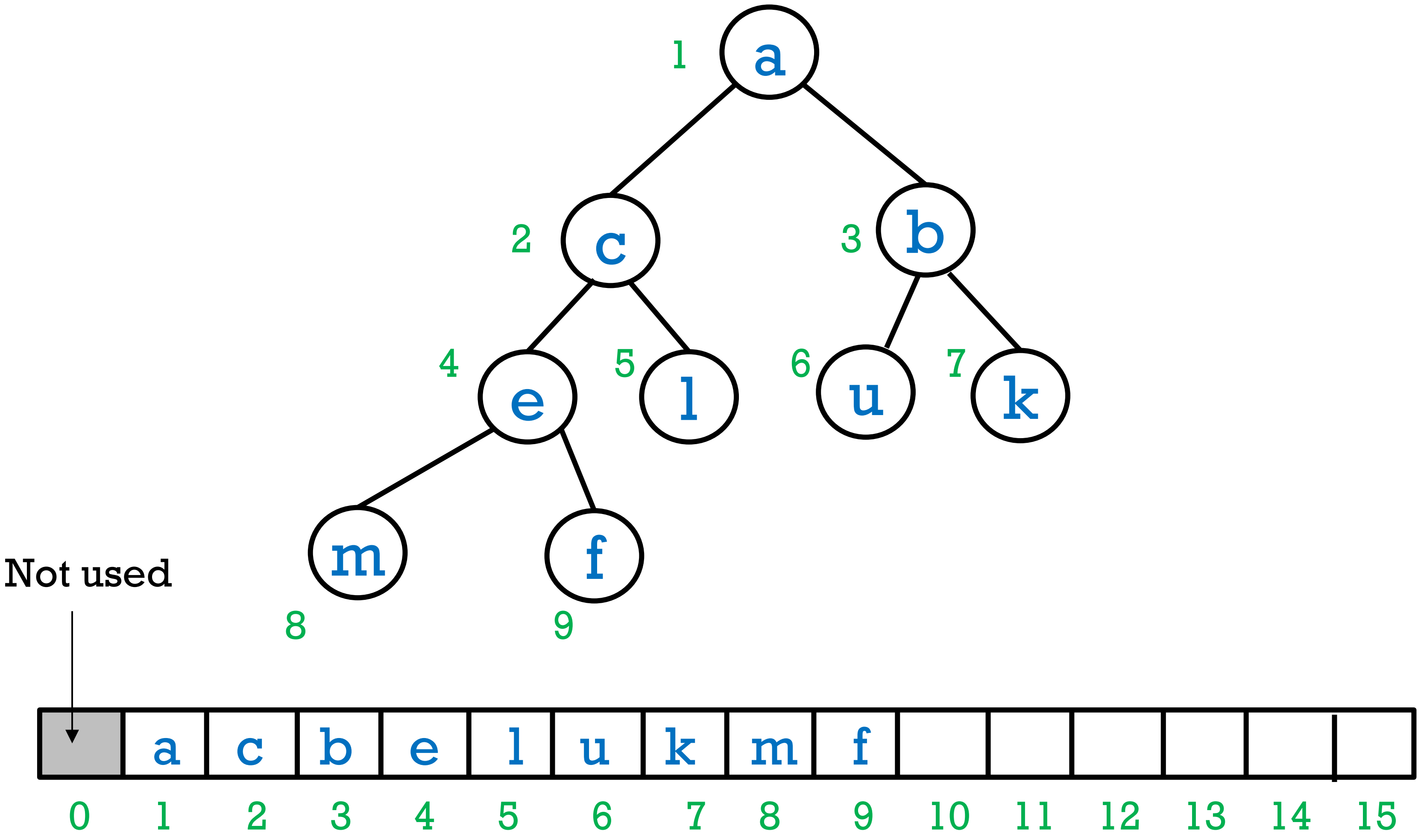
“upHeap”

removeMin()



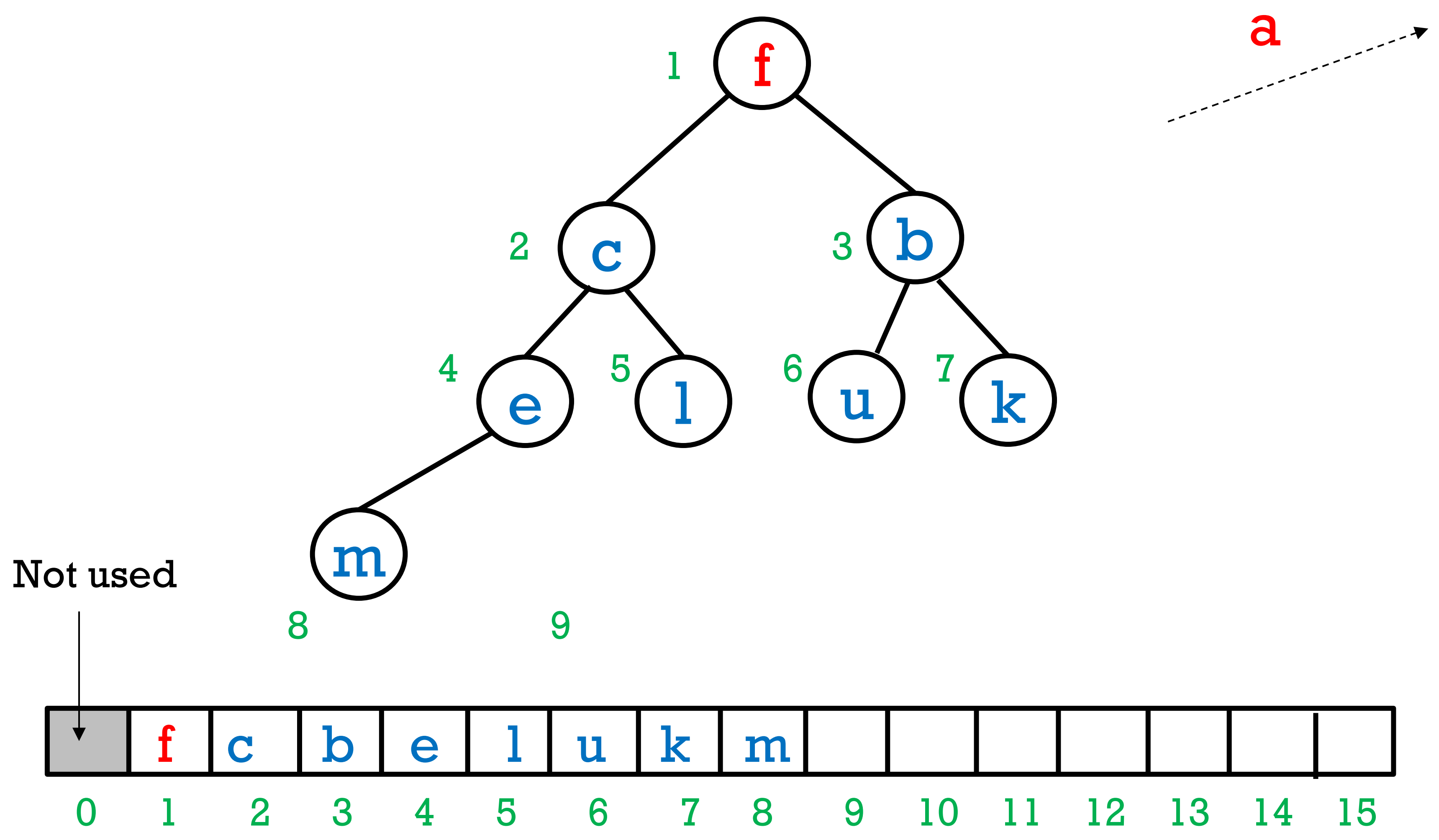
“downHeap”

removeMin()

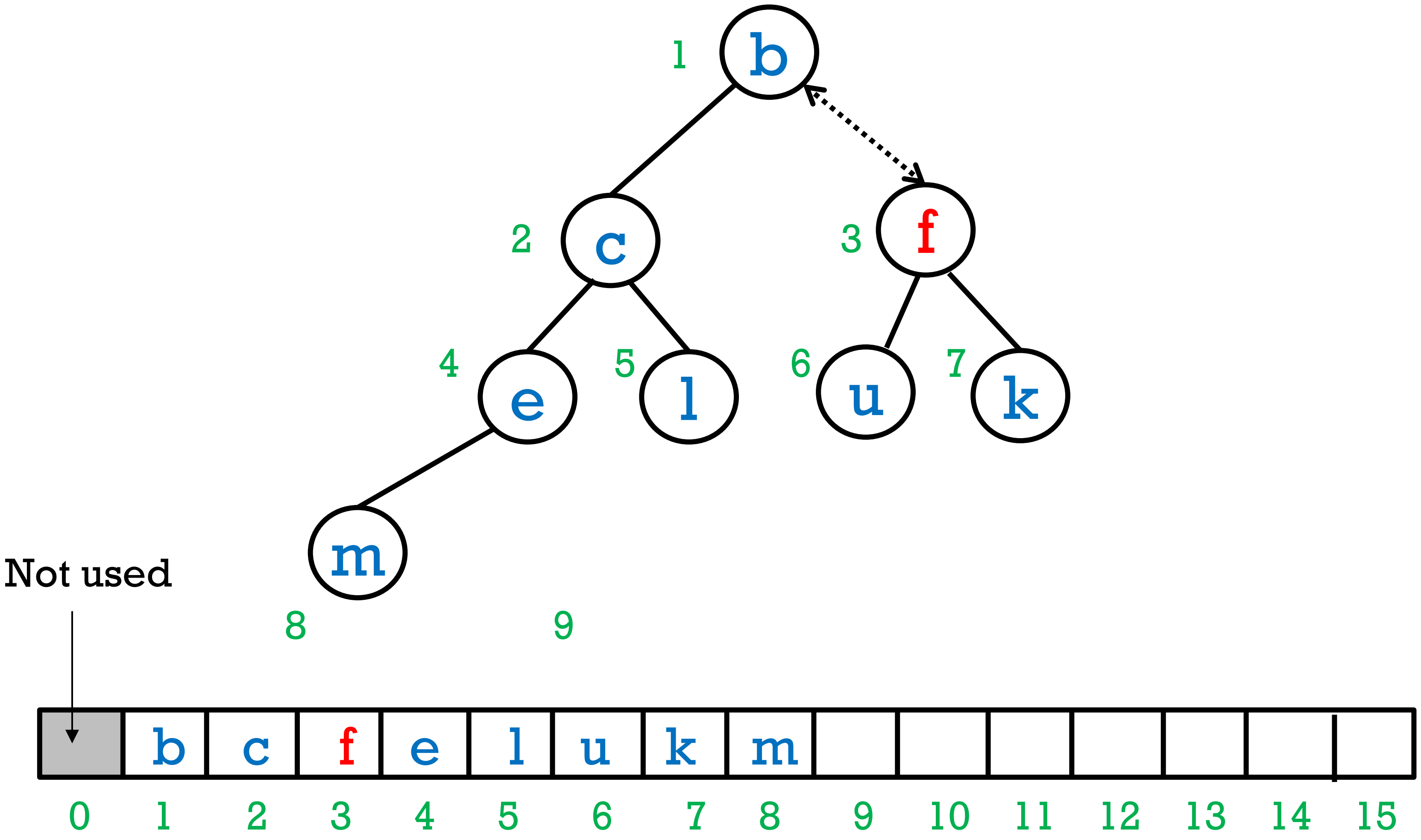


1
a
2
c
3
b
4
e
5
l
6
u
7
k
m
f
8
9
Not used
a
c
b
e
l
u
k
m
f
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

removeMin()


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

removeMin()



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

REMOVEMIN() - IMPLEMENTATION

Let `heap` be the underlying array, and let `size` be the number of elements in the heap.

```
removeMin( ){
      tmpElement = heap[1]   // save the min element
      heap[1] = heap[size]   // replace min with the last element
      heap[size] = null // not necessary



      return  tmpElement
}
```

REMOVEMIN() - IMPLEMENTATION

Let `heap` be the underlying array, and let `size` be the number of elements in the heap.

```
removeMin( ){
     tmpElement = heap[1]   // save the min element
     heap[1] = heap[size]   // replace min with the last element
     heap[size] = null // not necessary
     size = size - 1        // similar to "remove" in an ArrayList
     downHeap(1, size)      // heapify
     return  tmpElement
}
```

DOWNHEAP() - IMPLEMENTATION

```
downHeap( startIndex , maxIndex ){
   i = startIndex
   while (2*i <= maxIndex){ // if there is a left child
      child = 2*i









   }
}
```

DOWNHEAP() - IMPLEMENTATION

```
downHeap( startIndex , maxIndex ){
   i = startIndex
   while (2*i <= maxIndex){ // if there is a left child
      child = 2*i
      if (child < maxIndex)  { // if there is a right sibling
         if (heap[child + 1] < heap[child]) // if rightchild < leftchild
            child = child + 1
      }





   }
}
```

DOWNHEAP() - IMPLEMENTATION

```
downHeap( startIndex , maxIndex ){
   i = startIndex
   while (2*i <= maxIndex){ // if there is a left child
      child = 2*i
      if (child < maxIndex) { // if there is a right sibling
         if (heap[child + 1] < heap[child]) // if rightchild < leftchild
            child = child + 1
      }
      if (heap[child] < heap[i]){ // Do we need to swap with child?
         swapElements(i , child)
         i = child
      } else
         break
   }
}
```

This is the last lecture that is required in the final exam

Coming lectures:

- April 2 (Tentative): Set, Map and Hash Table + Personal advice for software engineering students
- April 4: Review
- April 9: Q&A (in classroom)