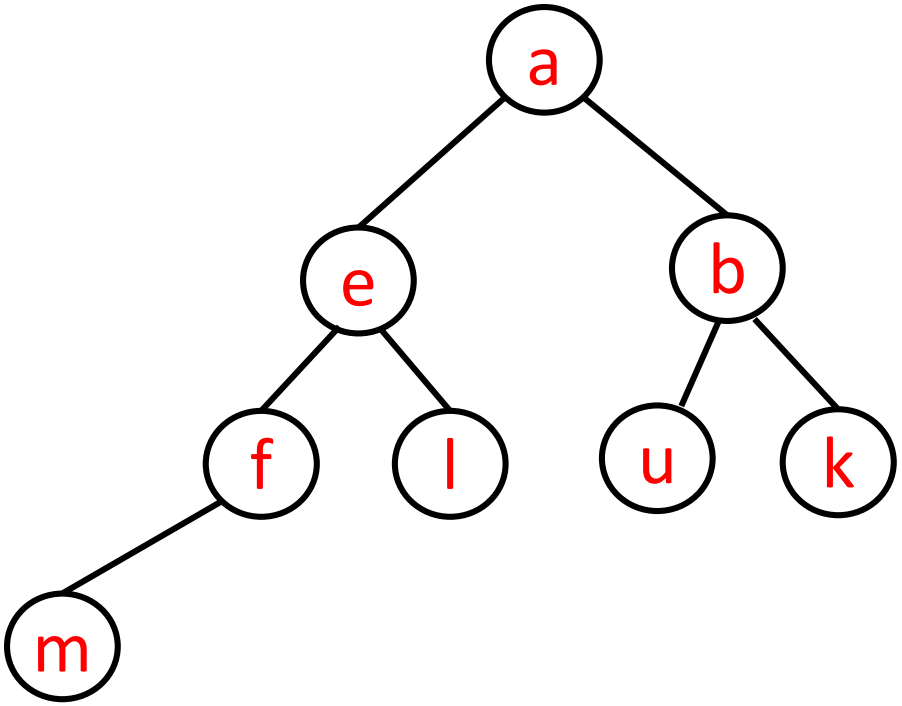
COMP 250

Lecture 27

heaps 2

Nov. 12, 2018

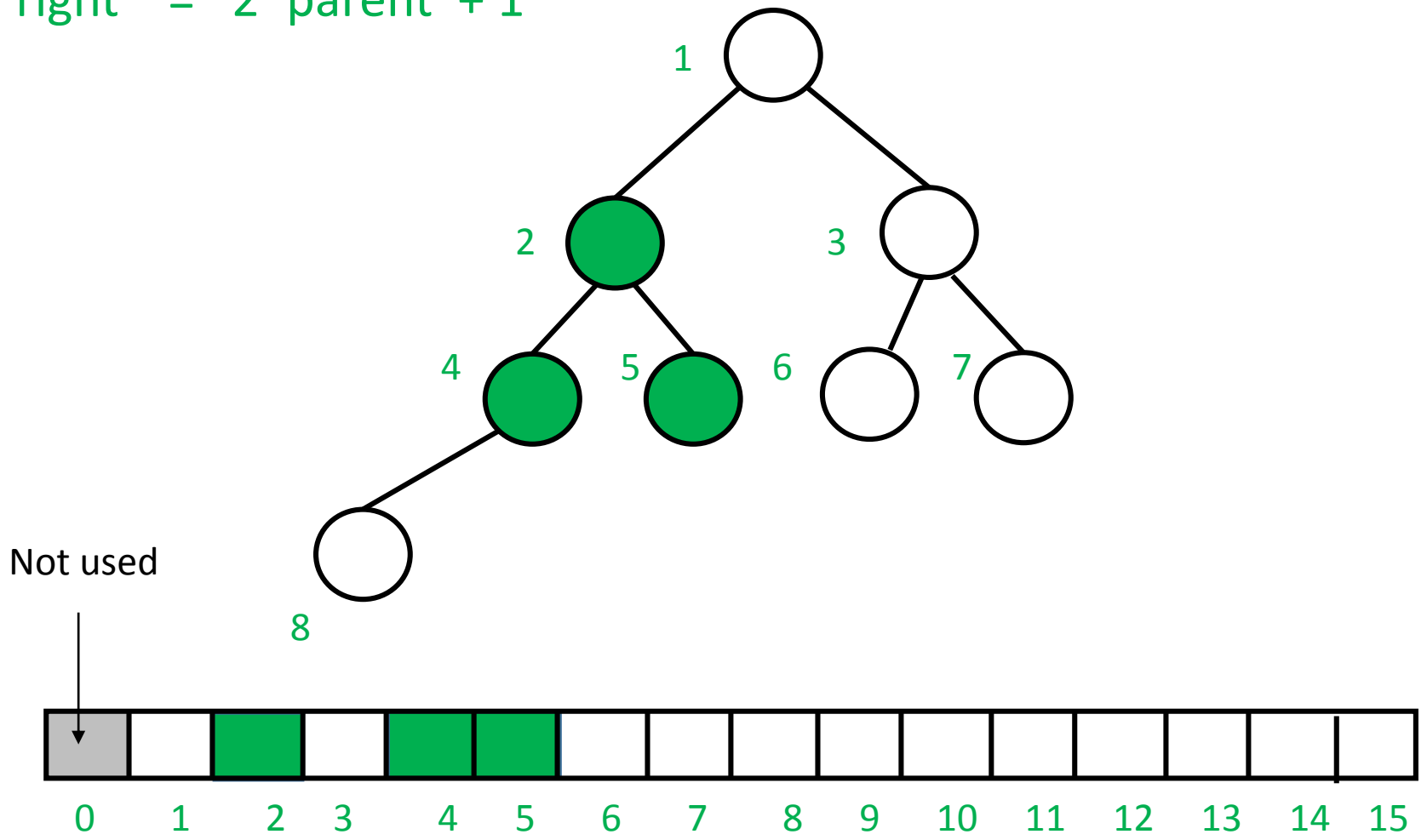
RECALL: min Heap (definition)



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

Heap index relations

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



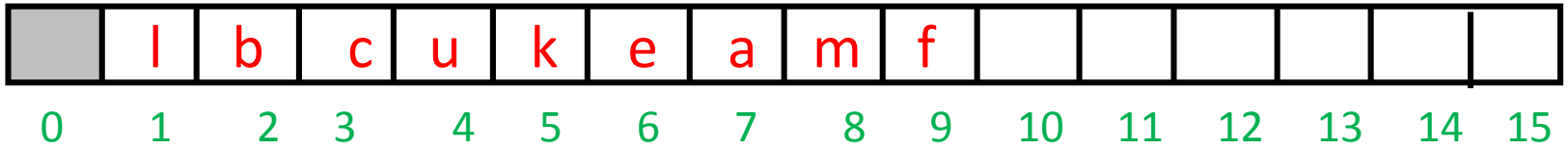
How to build a heap ?

```
buildHeap(){
   //  assume that an array already contains size elements
   for (k = 2;  k <=  size;  k++)
         upHeap( k ) }
}
```

How to build a heap ?

```
buildHeap(){
   //  assume that an array already contains size elements
   for (k = 2;  k <=  size;  k++)
         upHeap( k ) }
}

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

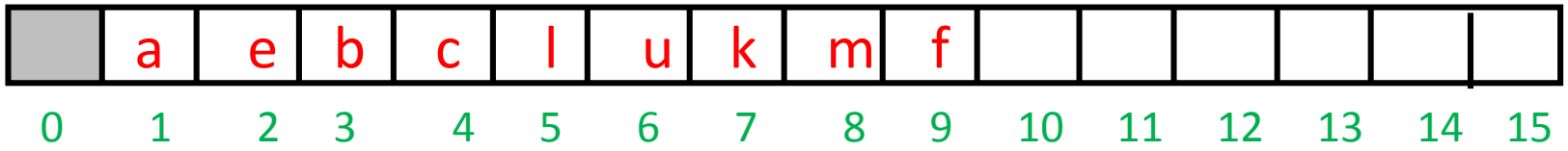
Best case of buildHeap is … ?



Given an array with n elements, how many swaps do we need to upHeap each element?

In the best case, … ?

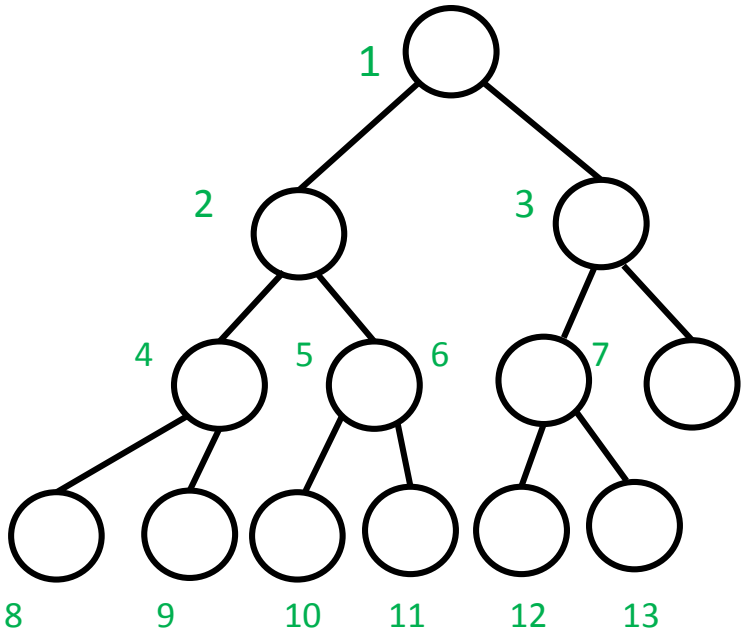
Best case of buildHeap is $O(n)$



In the best case, the elements happen to already satisfy the heap parent-child ordering constraint, and no swaps are necessary.

Why is it $O(n)$ rather than $O(1)$?

Worse case of buildHeap ?



level

0

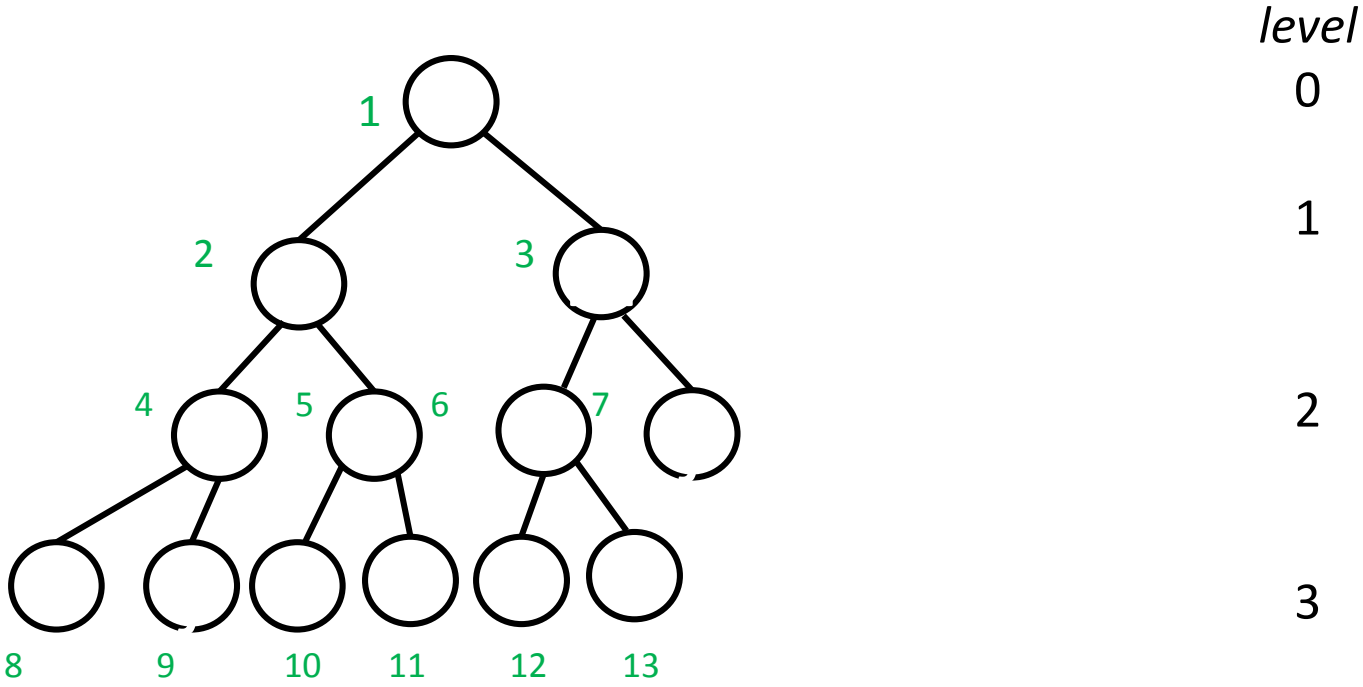
1

2

3

How many upHeap swaps do we need for element i ?

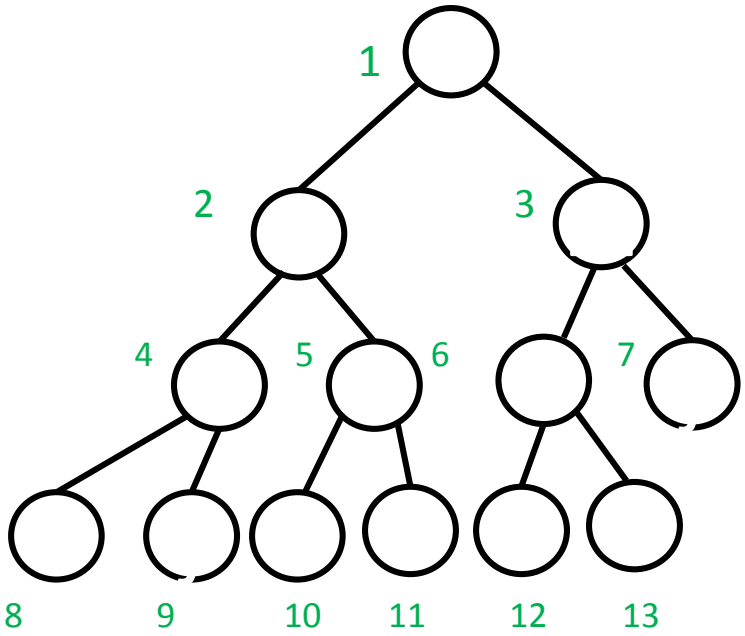
Worse case of buildHeap ?



How many upHeap swaps do we need for element i ?
Element i is at level, such that:

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

Worse case of buildHeap ?



level

0

1

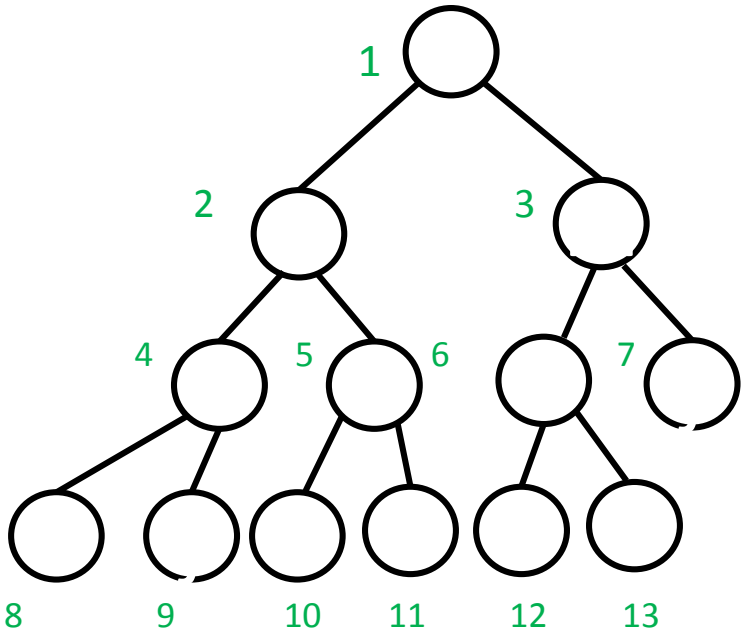
2

3

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

$$level \leq log_2 i < level + 1$$

Worse case of buildHeap ?

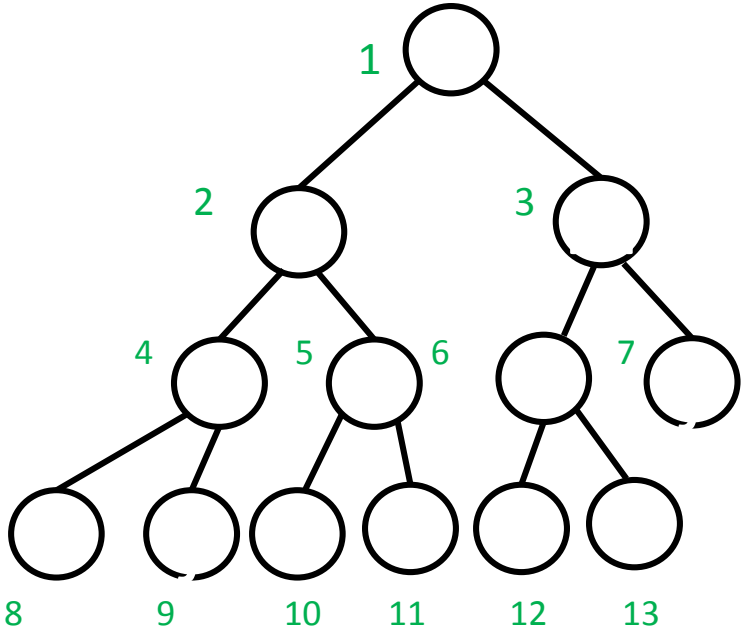


$$2^{level} \leq i < 2^{level+1}$$

$$level \leq \log_2 i < level + 1$$

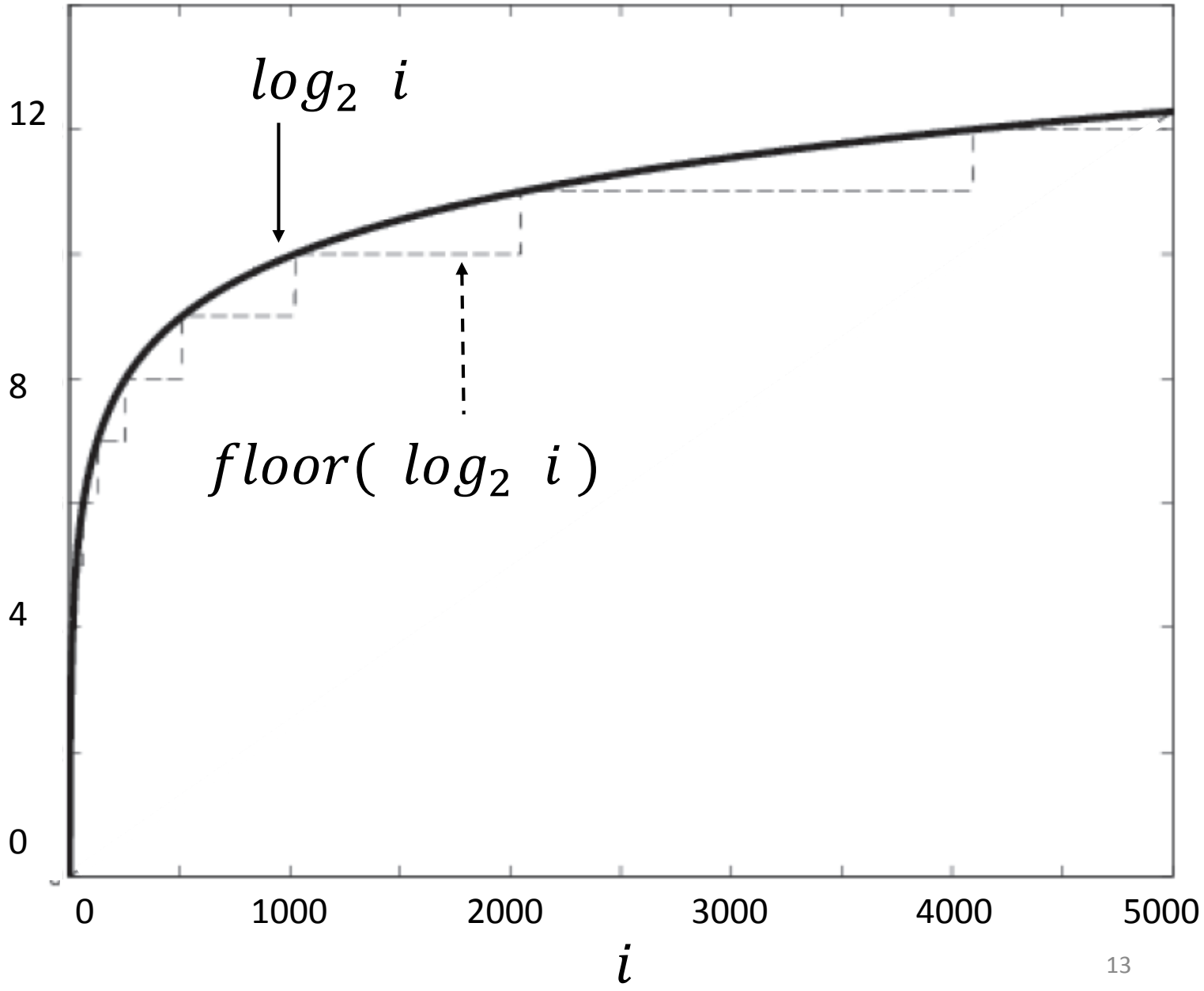
Thus, $level = floor(\log_2 i)$

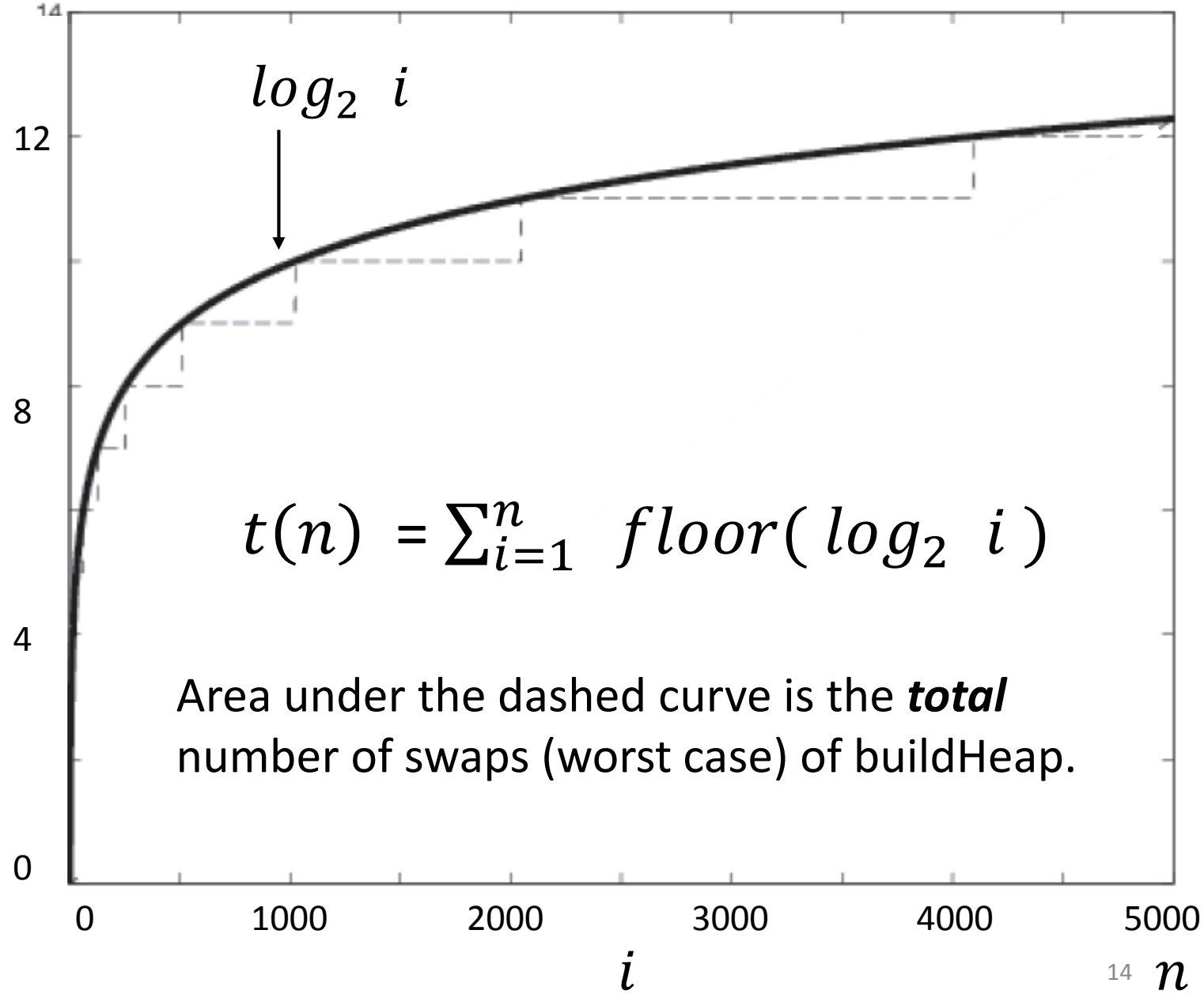
Worse case of buildHeap

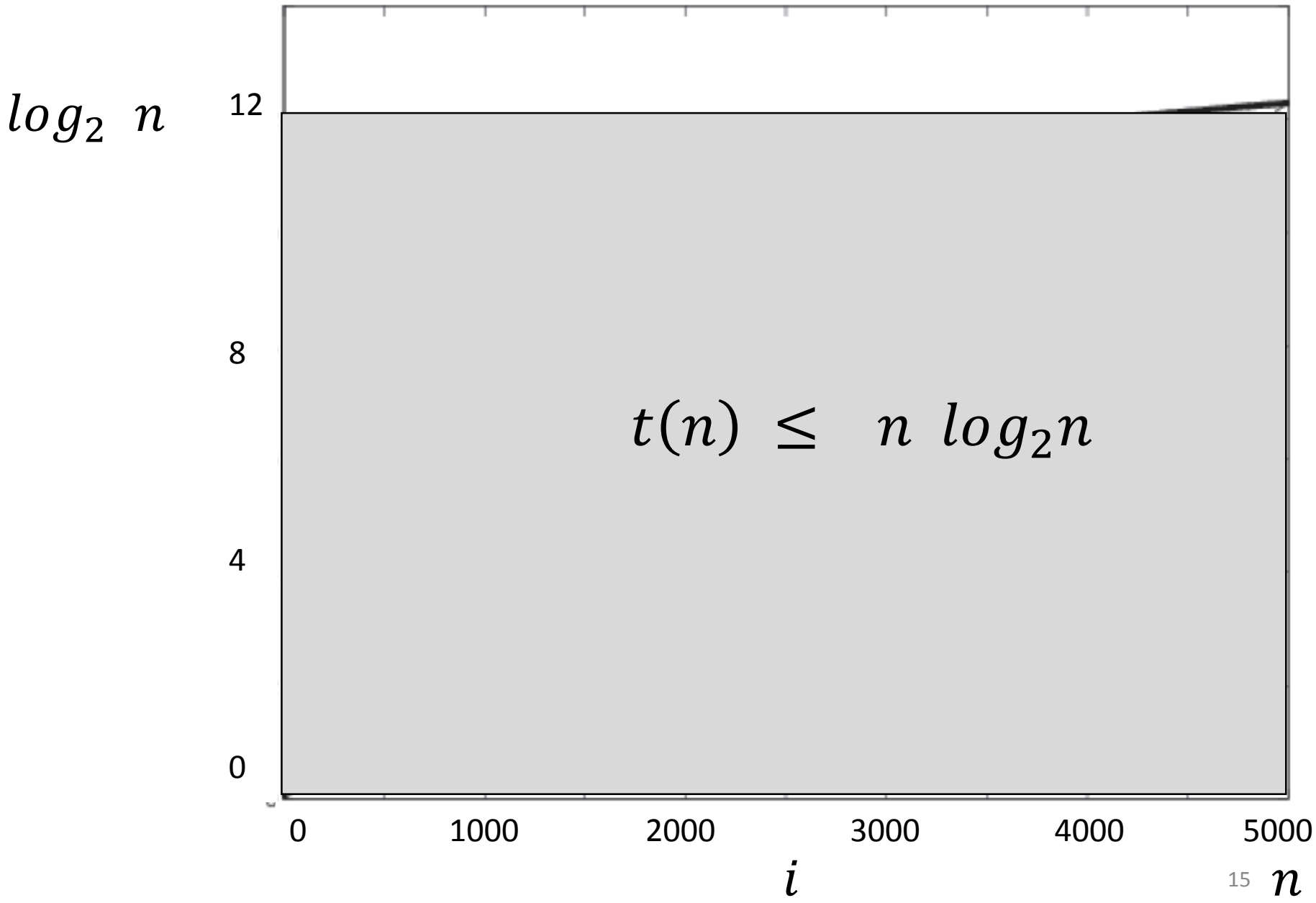


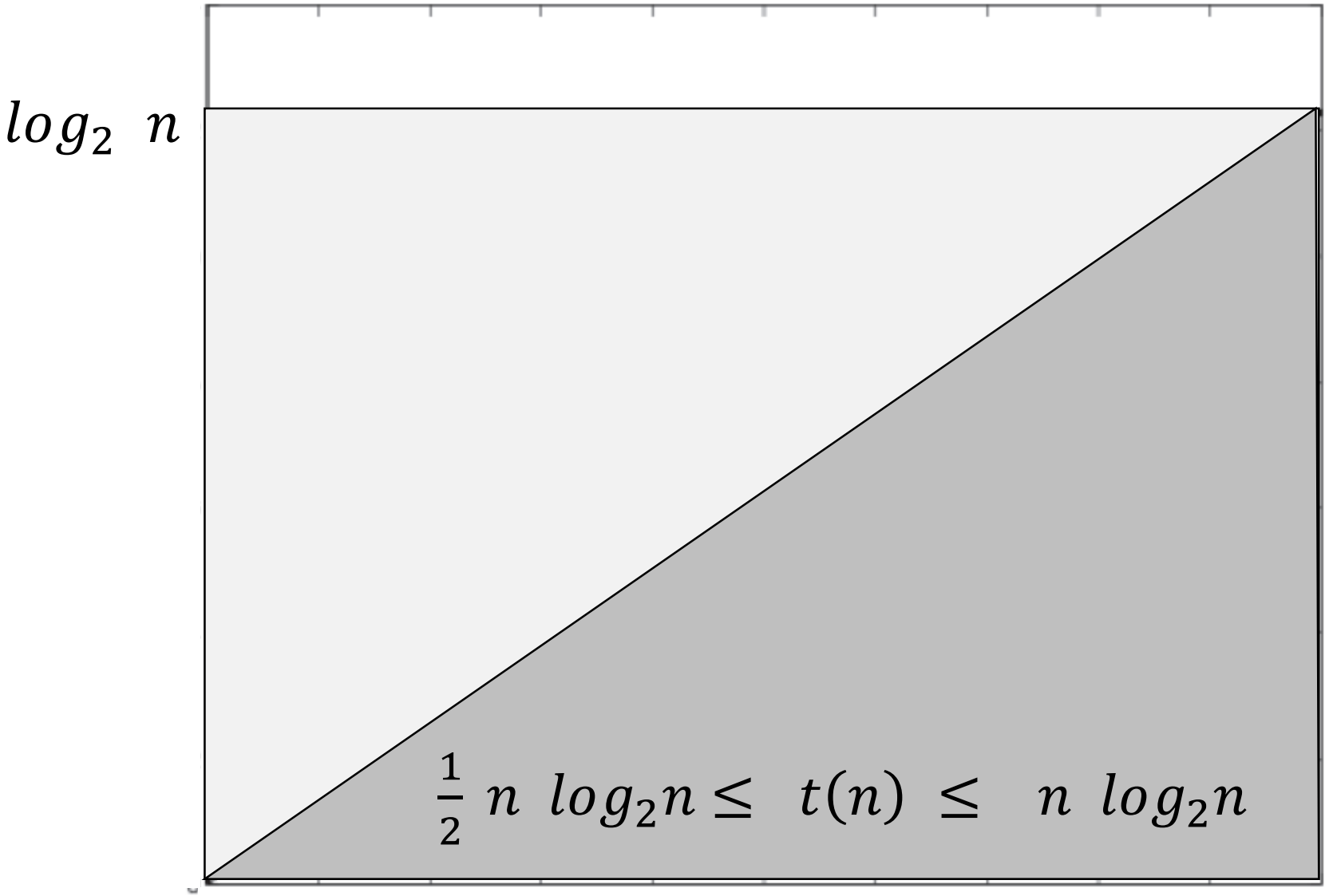
Worst case number of swaps needed to build a heap using upHeap.

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


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


14
12
8
4
0
$log_2\ i$
$t(n) = \sum_{i=1}^{n} floor(\ log_2\ i\ )$
Area under the dashed curve is the ***total*** number of swaps (worst case) of buildHeap.
0
1000
2000
3000
4000
5000
i
n


$log_2\ n$
12
8
4
0
$t(n) \leq\ \ n\ log_2 n$
0
1000
2000
3000
4000
5000
i
n


$log_2\ n$
$\frac{1}{2}\ n\ log_2 n \leq\ t(n)\ \leq\ \ n\ log_2 n$
n

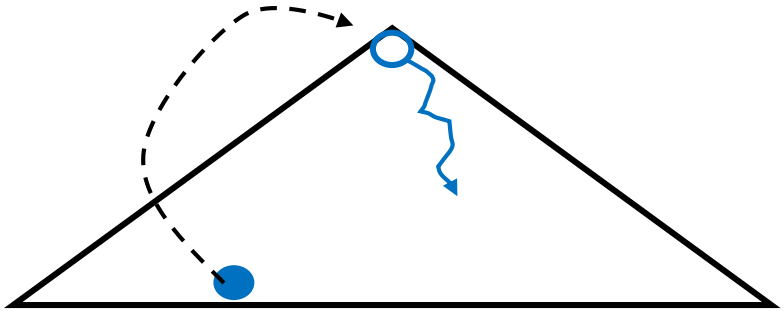
The worst case of buildHeap is $O(n \ log_2 n)$

Recall from last lecture

add(element)

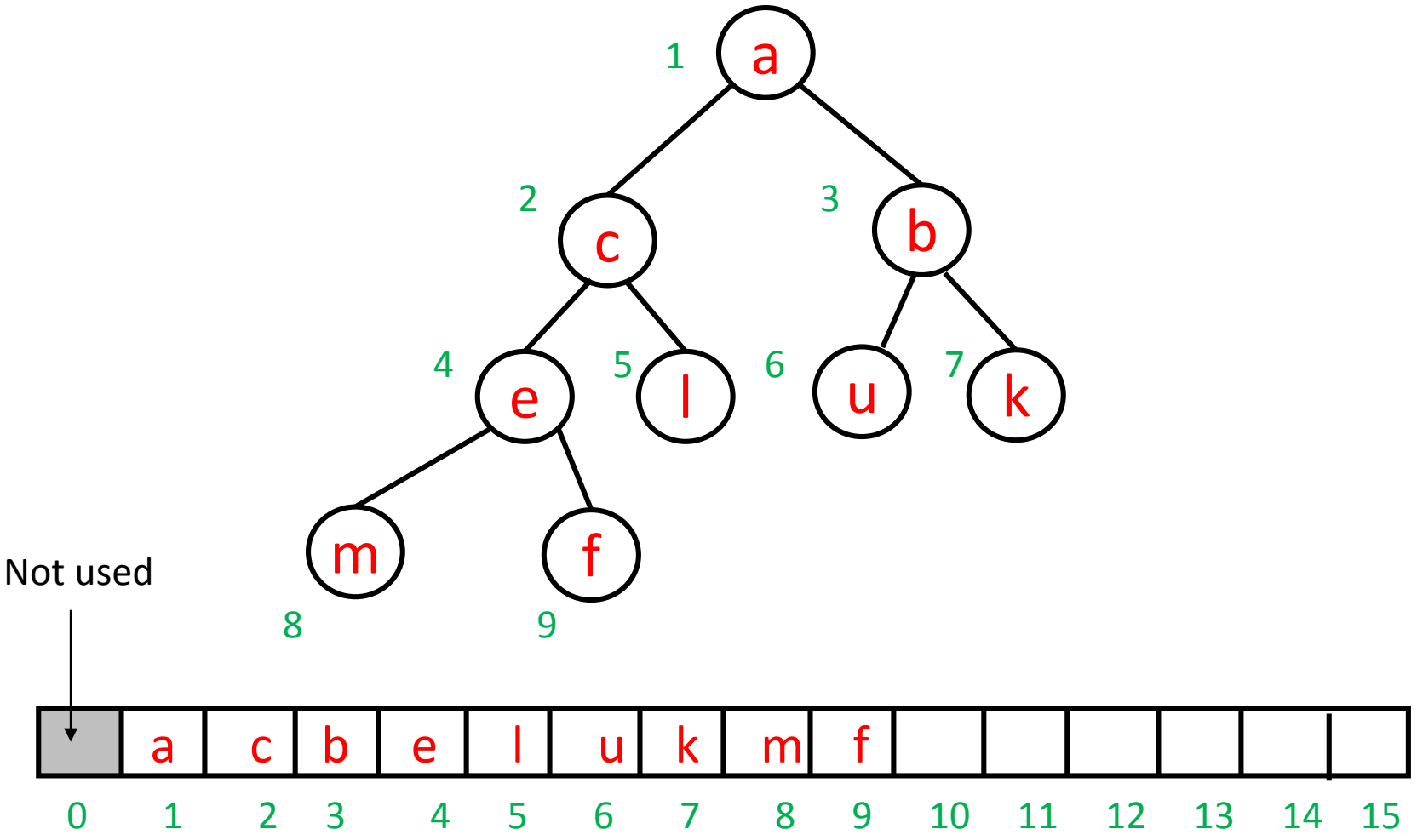
“upHeap”

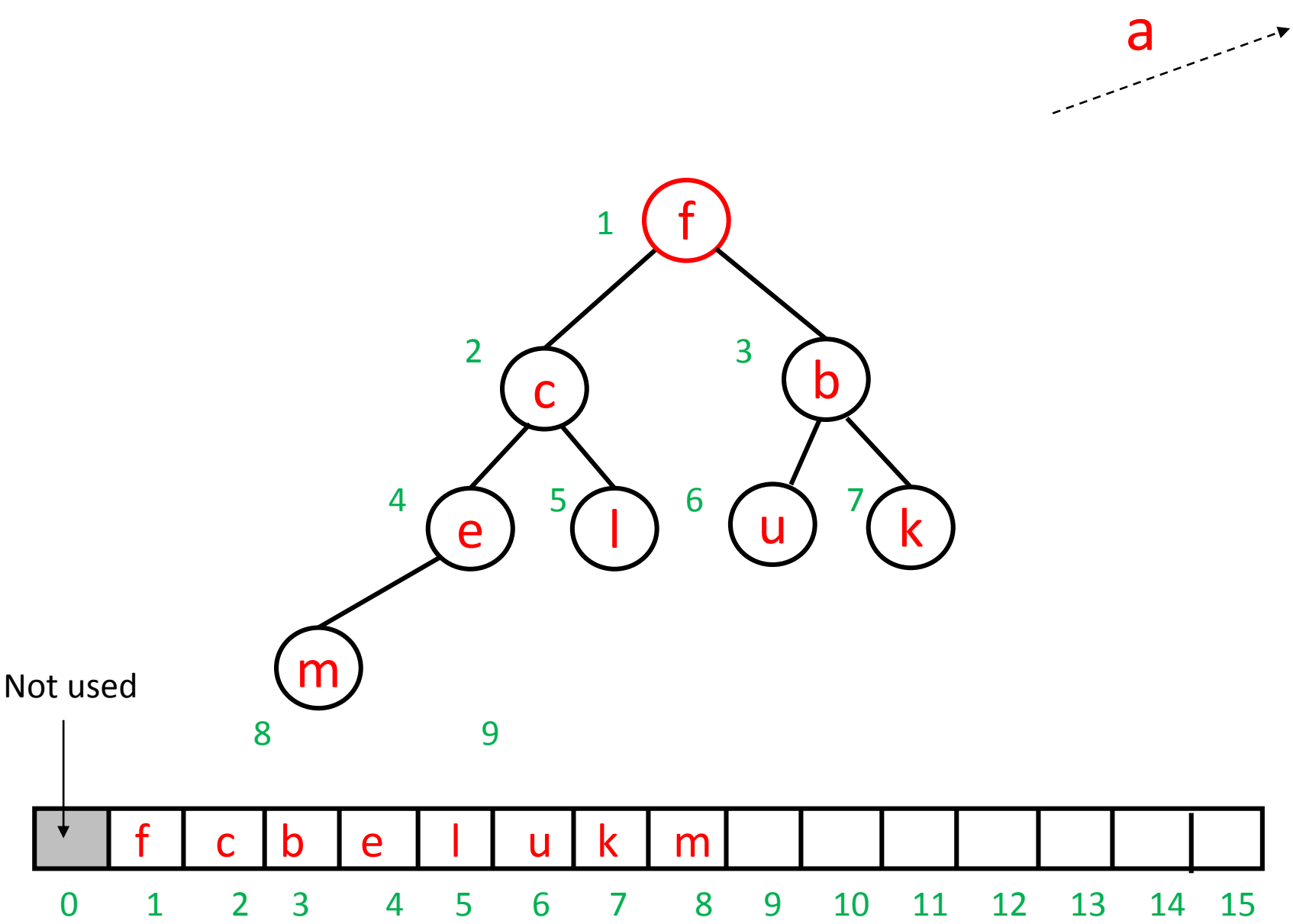
removeMin()

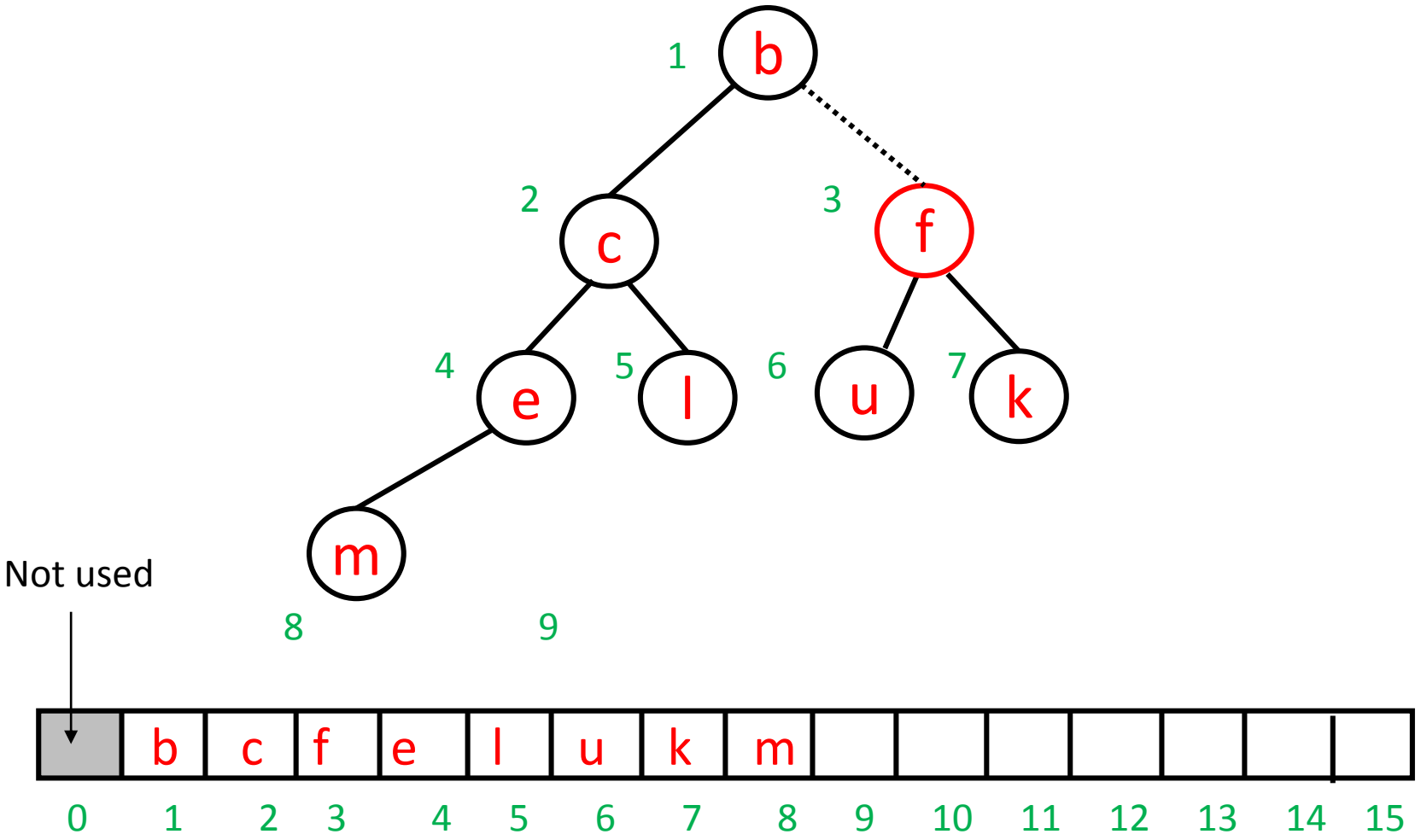


“downHeap”

e.g. 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


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()

Let heap[ ] be the array.
Let size be the number of elements in the heap.

```
removeMin( ){
    tmpElement = heap[1]          //   heap[0] not used.
    heap[1] = heap[size]
    
}
```

removeMin()

Let heap[ ] be the array.
Let size be the number of elements in the heap.

```
removeMin( ){
    tmpElement = heap[1]          //  heap[0] not used.
    heap[1] = heap[size]
    heap[size] = null
    size = size - 1
    downHeap( size )              //   next slide
    return  tmpElement
}
```

```
downHeap( maxIndex ){

   i = 1
   while ( 2*i  <= maxIndex){          // if there is a left child
      child = 2*i
      Find the smaller child  (left or right?)
    }
}
```

```
downHeap( maxIndex ){

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

```
downHeap( maxIndex ){

  i = 1
  while ( 2*i <= maxIndex){          // if there is a left child
     child = 2*i
     if child < size {               // 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 return                    //   avoid infinite loop.
   }
}
```

Heapsort

Given a list with $n =$ size elements:

Build a heap.

Call removeMin() n times.

On the next slide(s), we will see an elegant way to do it.

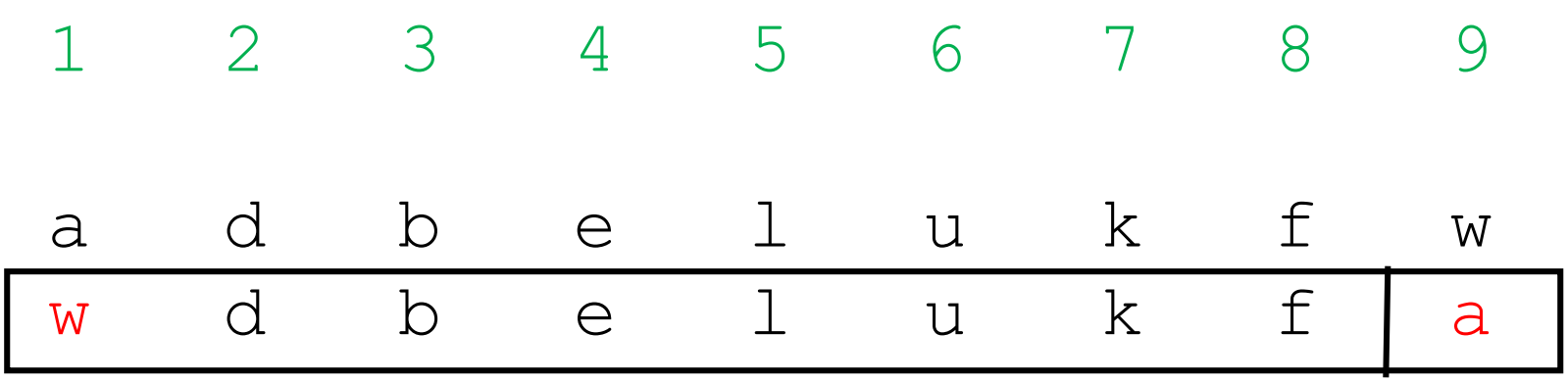
Heapsort

```
heapsort(list){

    buildheap(list)

    for i = 1 to size-1{
        swapElements( heap[1], heap[size + 1 - i])
        downHeap( 1,  size - i)
    }
    return reverse(heap)
}
```

1	2	3	4	5	6	7	8	9
a	d	b	e	l	u	k	f	w

This shows the array after we have built the heap.
Now we execute the following:

```
for i = 1 to size-1{
      swapElements( heap[1], heap[size + 1 - i])
      downHeap( 1,  size - i)
  }
```



This shows the array after we have built the heap.
Now we execute the following:

```
for i = 1 to size-1{
      swapElements( heap[1], heap[size + 1 - i])
      downHeap( 1,  size - i)
  }
```

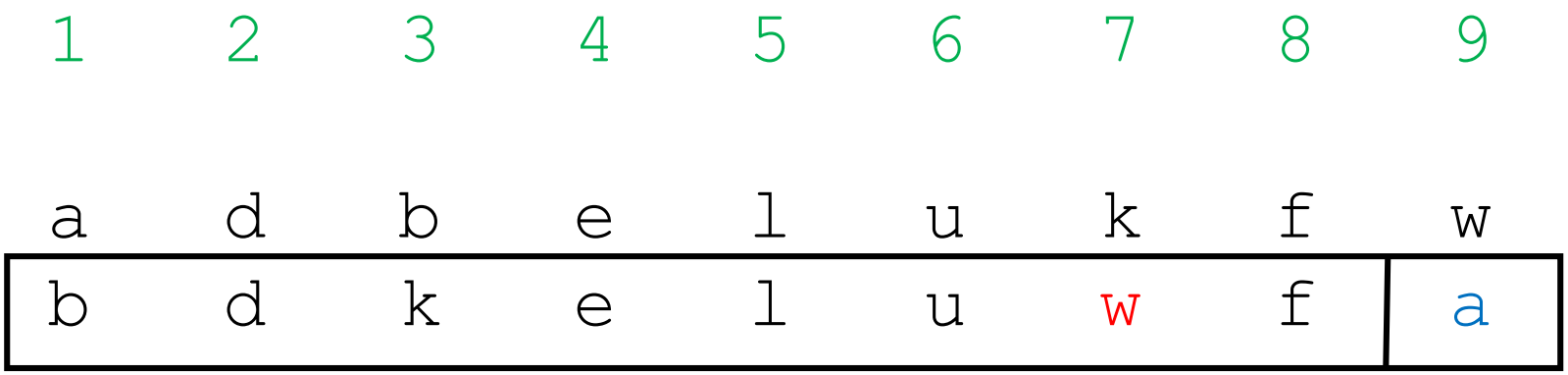

i = 1

1	2	3	4	5	6	7	8	9
a	d	b	e	l	u	k	f	w
b	d	w	e	l	u	k	f	a

This shows the array after we have built the heap.
Now we execute the following:

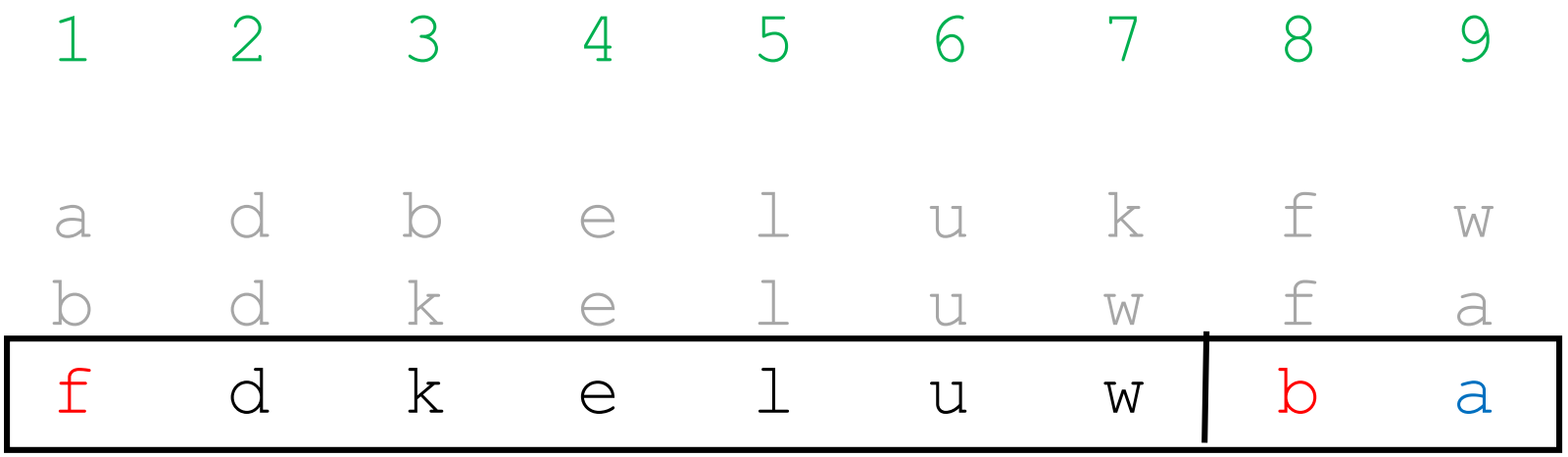
```
for i = 1 to size-1{
    swapElements( heap[1], heap[size + 1 - i])
    downHeap( 1,  size - i)
  }
```

i = 1



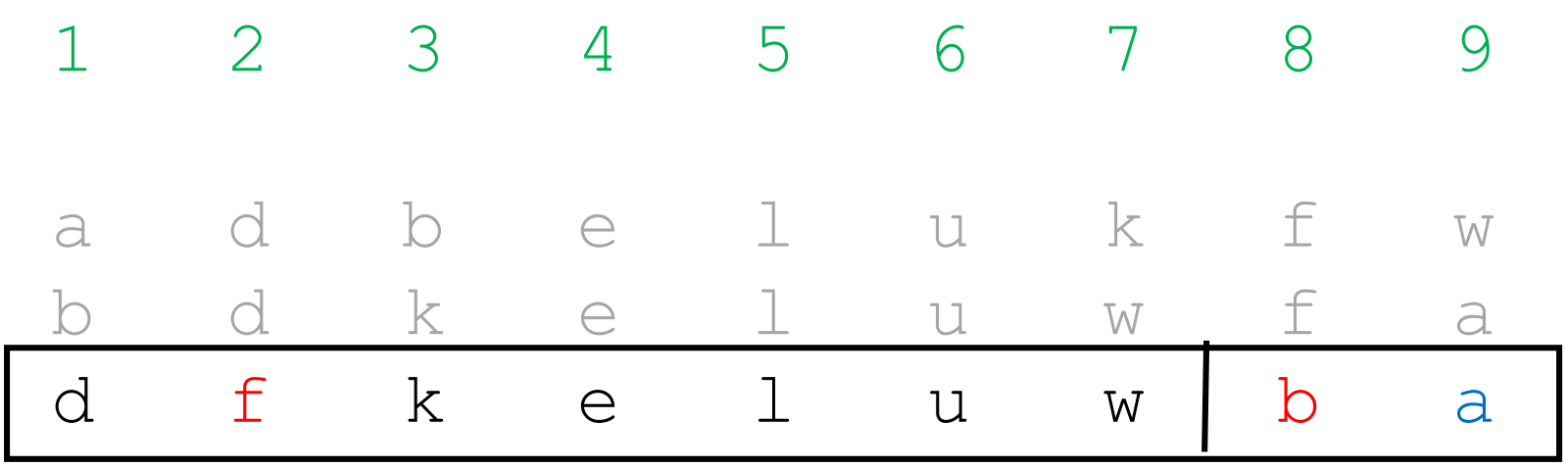

```
for i = 1 to size-1{
      swapElements( heap[1], heap[size + 1 - i])
      downHeap( 1,  size - i)
  }
```

i = 1



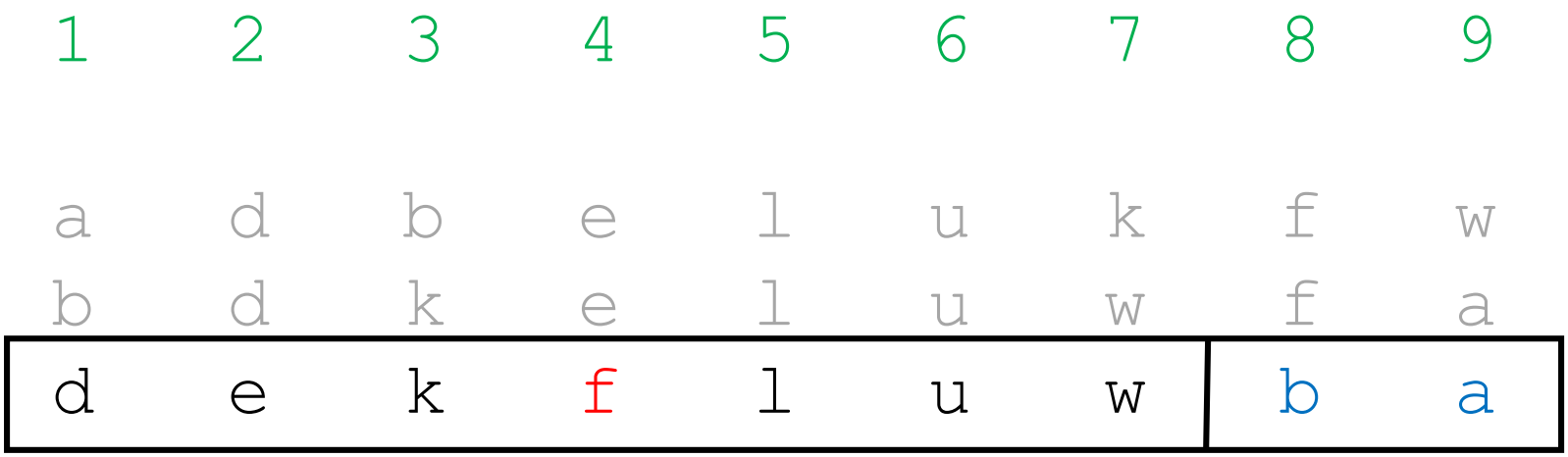

```
for i = 1 to size-1{
      swapElements( heap[1], heap[size + 1 - i])
      downHeap( 1,  size - i)
  }
```


i = 2



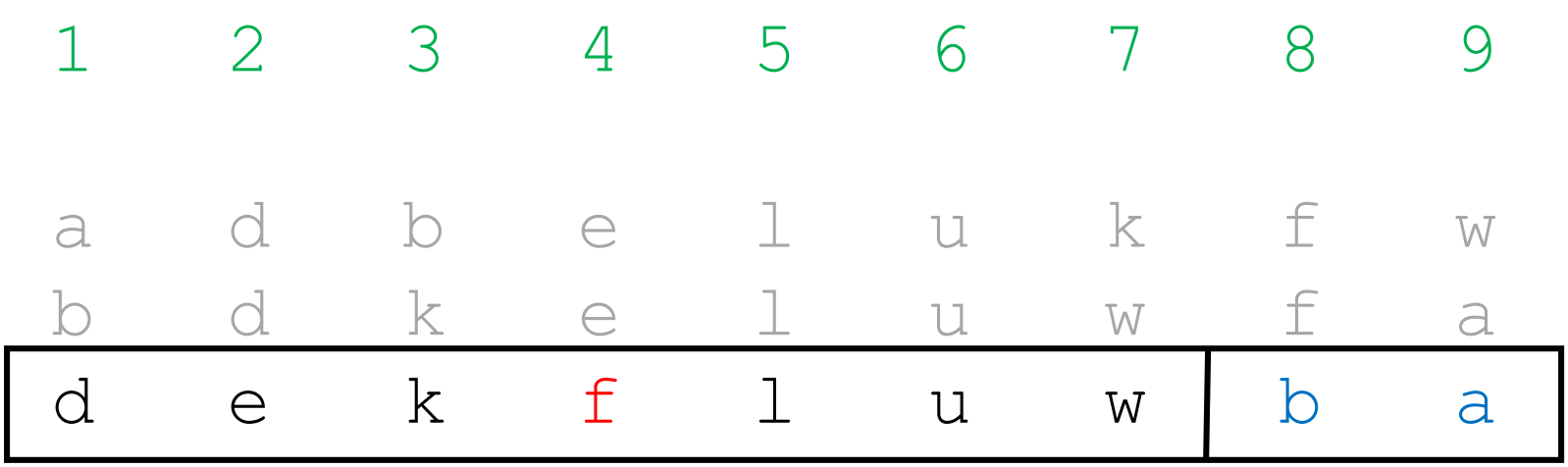

```
for i = 1 to size-1{
    swapElements( heap[1], heap[size + 1 - i])
    downHeap( 1,  size - i)
  }
```

i = 2




```
for i = 1 to size-1{
    swapElements( heap[1], heap[size + 1 - i])
    downHeap( 1,  size - i)
  }
```

i = 2




```
for i = 1 to size-1{
      swapElements( heap[1], heap[size + 1 - i])
      downHeap( 1,  size - i)
  }
```

i = 2

1	2	3	4	5	6	7	8	9
a	d	b	e	l	u	k	f	w
b	d	k	e	l	u	w	f	a
d	e	k	f	l	u	w	b	a
w	e	k	f	l	u	d	b	a

```
for i = 1 to size-1{
      swapElements( heap[1], heap[size + 1 - i])
      downHeap( 1,  size - i)
  }
```


i = 3

1	2	3	4	5	6	7	8	9
a	d	b	e	l	u	k	f	w
b	d	k	e	l	u	w	f	a
d	e	k	f	l	u	w	b	a
e	w	k	f	l	u	d	b	a

```
for i = 1 to size-1{
    swapElements( heap[1], heap[size + 1 - i])
    downHeap( 1,  size - i)
  }
```

i = 3

1	2	3	4	5	6	7	8	9
a	d	b	e	l	u	k	f	w
b	d	k	e	l	u	w	f	a
d	e	k	f	l	u	w	b	a
e	f	k	w	l	u	d	b	a

```
for i = 1 to size-1{
      swapElements( heap[1], heap[size + 1 - i])
      downHeap( 1,  size - i)
  }
```


i = 3

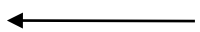
1	2	3	4	5	6	7	8	9
a	d	b	e	l	u	k	f	w
b	d	k	e	l	u	w	f	a
d	e	k	f	l	u	w	b	a
e	f	k	w	l	u	d	b	a
u	f	k	w	l	e	d	b	a

```
for i = 1 to size-1{
    swapElements( heap[1], heap[size + 1 - i])
    downHeap( 1,  size - i)
  }
```


i = 4

1	2	3	4	5	6	7	8	9
a	d	b	e	l	u	k	f	w
b	d	k	e	l	u	w	f	a
d	e	k	f	l	u	w	b	a
e	f	k	w	l	u	d	b	a
f	u	k	w	l	e	d	b	a

```
for i = 1 to size-1{
    swapElements( heap[1], heap[size + 1 - i])
    downHeap( 1,  size - i)
 }
```



i = 4

1	2	3	4	5	6	7	8	9
a	d	b	e	l	u	k	f	w
b	d	k	e	l	u	w	f	a
d	e	k	f	l	u	w	b	a
e	f	k	w	l	u	d	b	a
f	l	k	w	u	e	d	b	a

```
for i = 1 to size-1{
    swapElements( heap[1], heap[size + 1 - i])
    downHeap( 1,  size - i)
 }
```

i = 4

1	2	3	4	5	6	7	8	9
a	d	b	e	l	u	k	f	w
b	d	k	e	l	u	w	f	a
d	e	k	f	l	u	w	b	a
e	f	k	w	l	u	d	b	a
f	l	k	w	u	e	d	b	a
k	l	u	w	f	e	d	b	a
l	w	u	k	f	e	d	b	a
u	w	l	k	f	e	d	b	a
w	u	l	k	f	e	d	b	a
w	u	l	k	f	e	d	b	a

Tip: To avoid making a mistake in an exam situation, I suggest you draw a sequence of trees rather than a sequence of arrays.

e.g. Suppose you are asked what is the state of the array after 3 passes through the heapsort loop.

Heapsort

```
heapsort(list){
    buildheap(list)
    for i = 1 to size-1{
        swapElements( heap[1], heap[size + 1 - i])
        downHeap( 1,  size - i)
    }
    return reverse(heap)
}
```

Exercise: time complexity of heap sort ?

Heapsort (worst case)

```
heapsort(list){
    buildheap(list)
    for i = 1 to size-1{
        swapElements( heap[1], heap[size + 1 - i])
        downHeap( 1,  size - i)
    }
    return reverse(heap)
}
```

$n \log(n)$

$c + \log(n - i)$

n

Exercise: time complexity of heap sort ?

Heapsort (worst case)

$$t(n) = n\,logn + cn + \underbrace{\sum_{i=1}^{n}(\log(n-i)\,)}_{\frac{1}{2}\,n\,log_2 n \leq \quad \leq \quad n\,log_2 n} + n$$

$t(n)$ is $O(n\ \log n)$

Similar to mergesort.