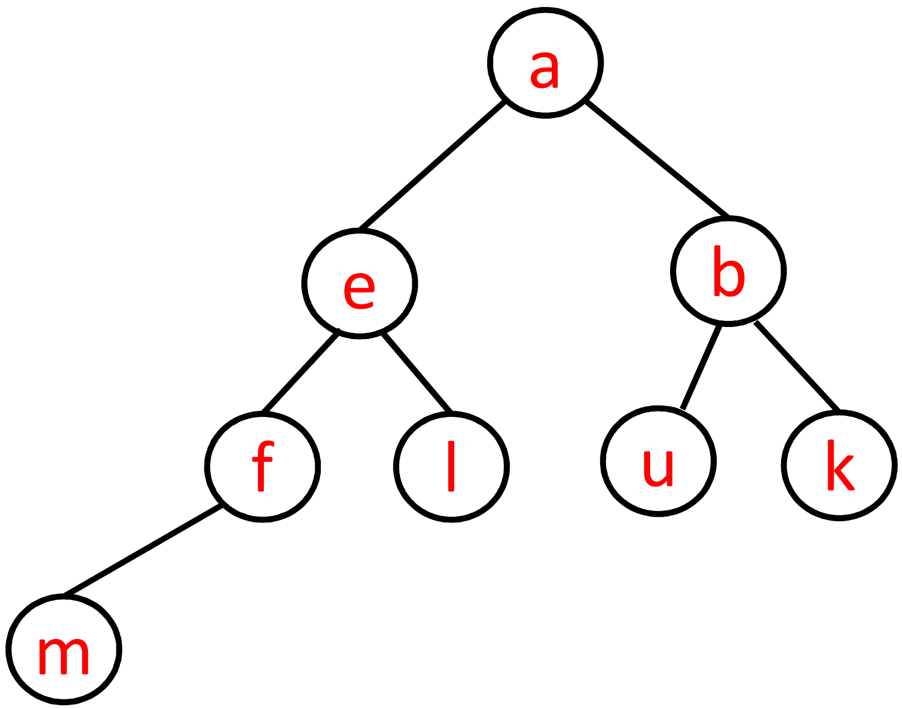
COMP 250

Lecture 28

heaps 2

March 18, 2022

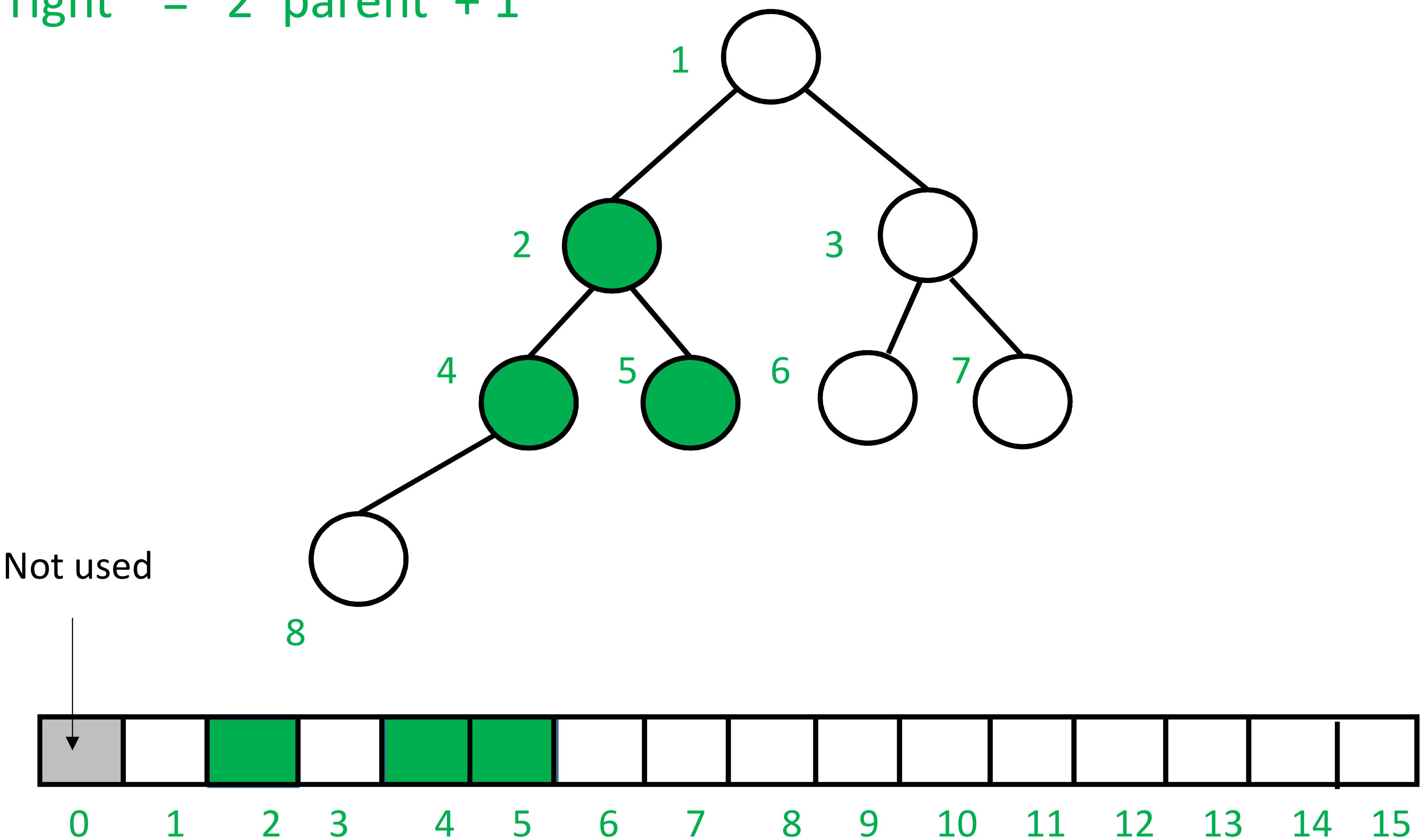
RECALL: min Heap (definition)



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

Heap index relations

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



Plan for today

- building a heap -- best and worst cases
- removeMin() using array indices
- heapsort

Recall: How to build a heap ?

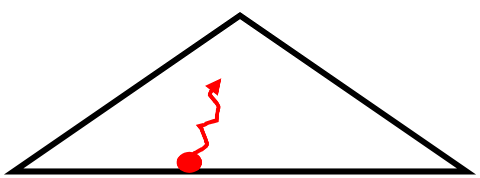
```
buildHeap(list){
  create an array arr[ ] with list.size+1 slots
  for (k = 1; k <= list.size; k++){
      arr[k] = list[k-1]      //  list indices are 0, .. ,size-1
      upHeap( arr, k ) }
  }
}
```

	l	b	c	u	k	e	a	m	f
0	1	2	3	4	5	6	7	8	9

Recall: How to build a heap ?

```
buildHeap(list){
  create an array arr[ ] with list.size+1 slots
  for (k = 1; k <= list.size; k++){
      arr[k] = list[k-1]      //  list indices are 0, .. ,size-1
      upHeap( arr, k ) }
  }
  return arr
}

upHeap(arr, k){            //  from last lecture
     i = k
     while (i > 1) and ( arr[i] <  arr[i / 2] ){
            swapkeys( i, i/2)
            i = i/2
     }
}
```



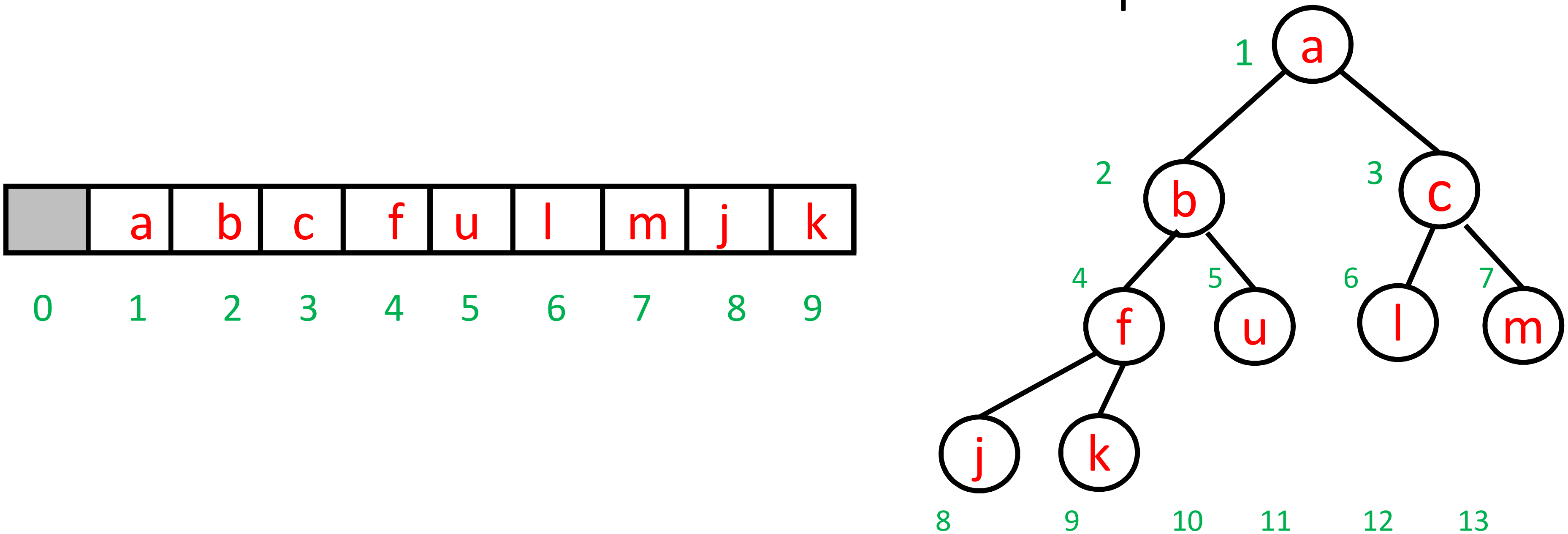
Time Complexity of buildHeap

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

In the best case, … ?

In the worst case, ... ?

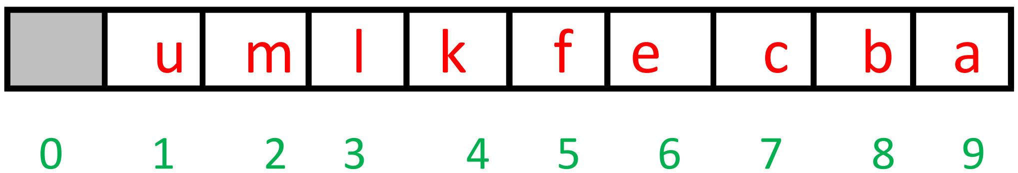
Best case of buildHeap

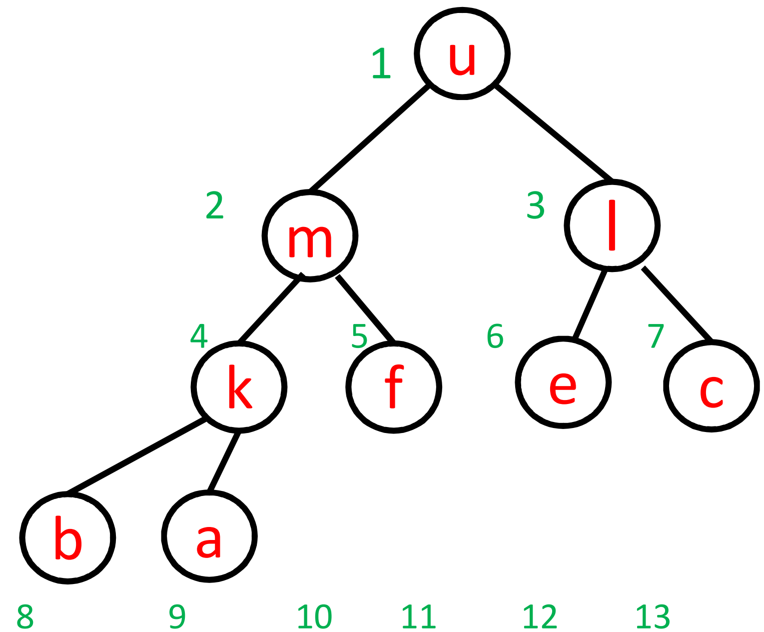


In the best case, *the keys already satisfy the heap property*, and no swaps are necessary.

The time complexity in the best case is $O(n)$, because we need to ensure each node's key is greater than its parent's key.

Worse case of buildHeap ?

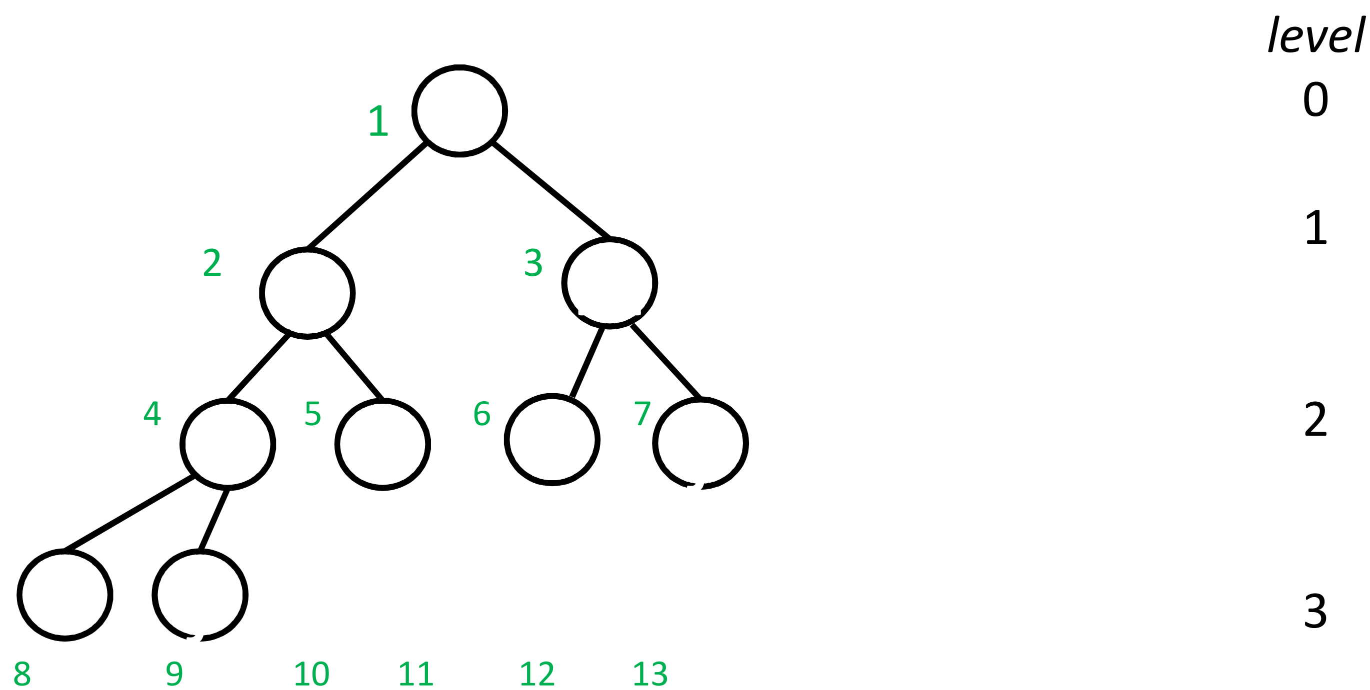




In this example, each parent key is greater than both children keys.

How many upHeap swaps do we need for key at position i ?

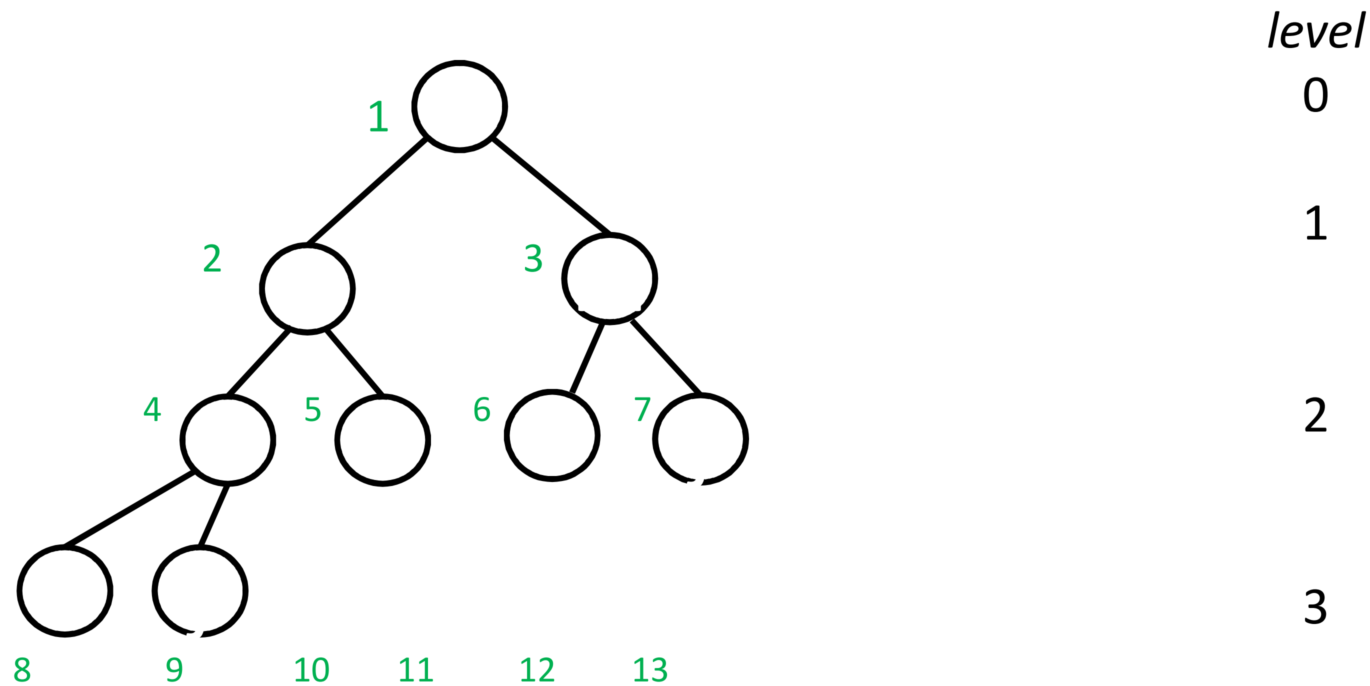
Worse case of buildHeap ?



How many upHeap swaps do we need for key at position i ?
Position i is at some *level*, such that:

$$2^{level} \leq i < 2^{level+1}, \quad \text{so} \quad level = \;?$$

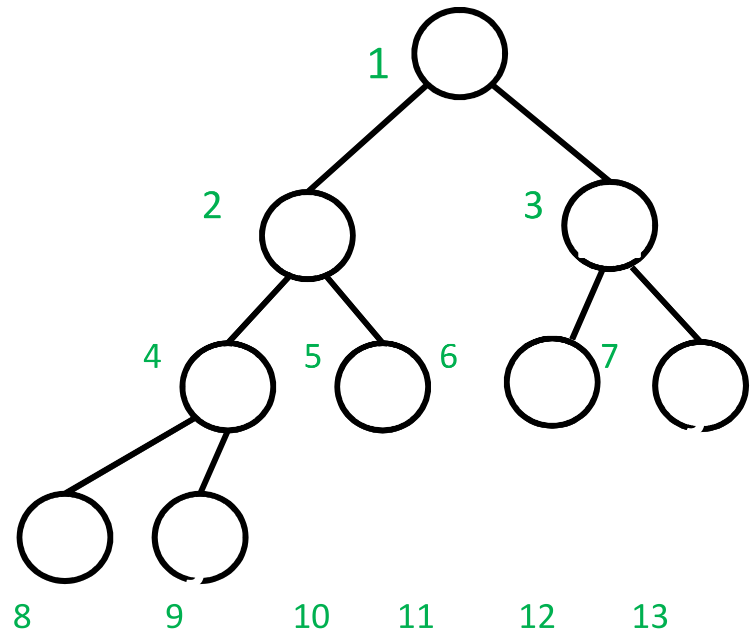
Worse case of buildHeap ?



How many upHeap swaps do we need for key at position i ?
Position i is at some *level*, such that:

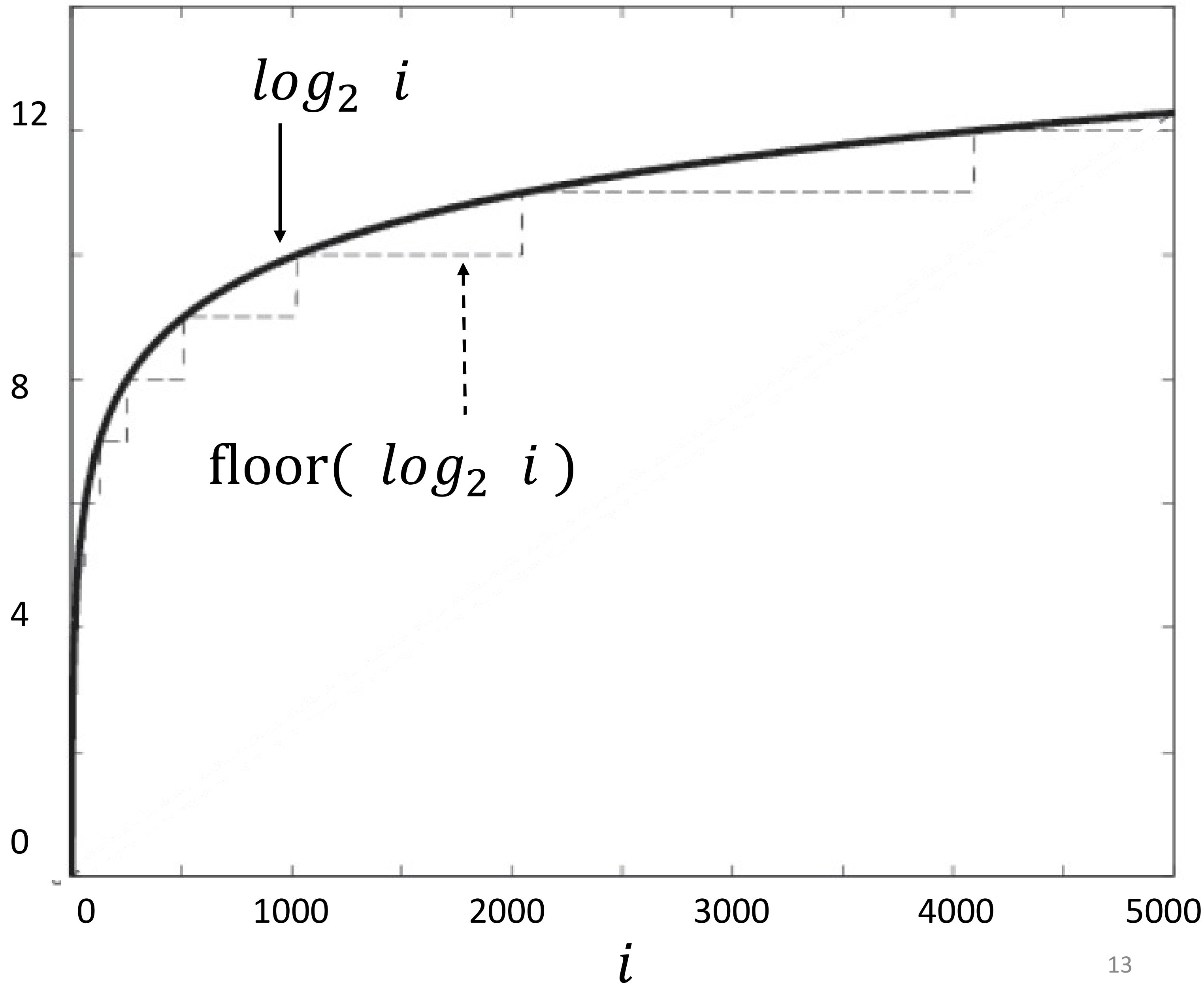
$$2^{level} \leq i < 2^{level+1}, \text{ so } level = floor(\log_2 i)$$

Worse case of buildHeap

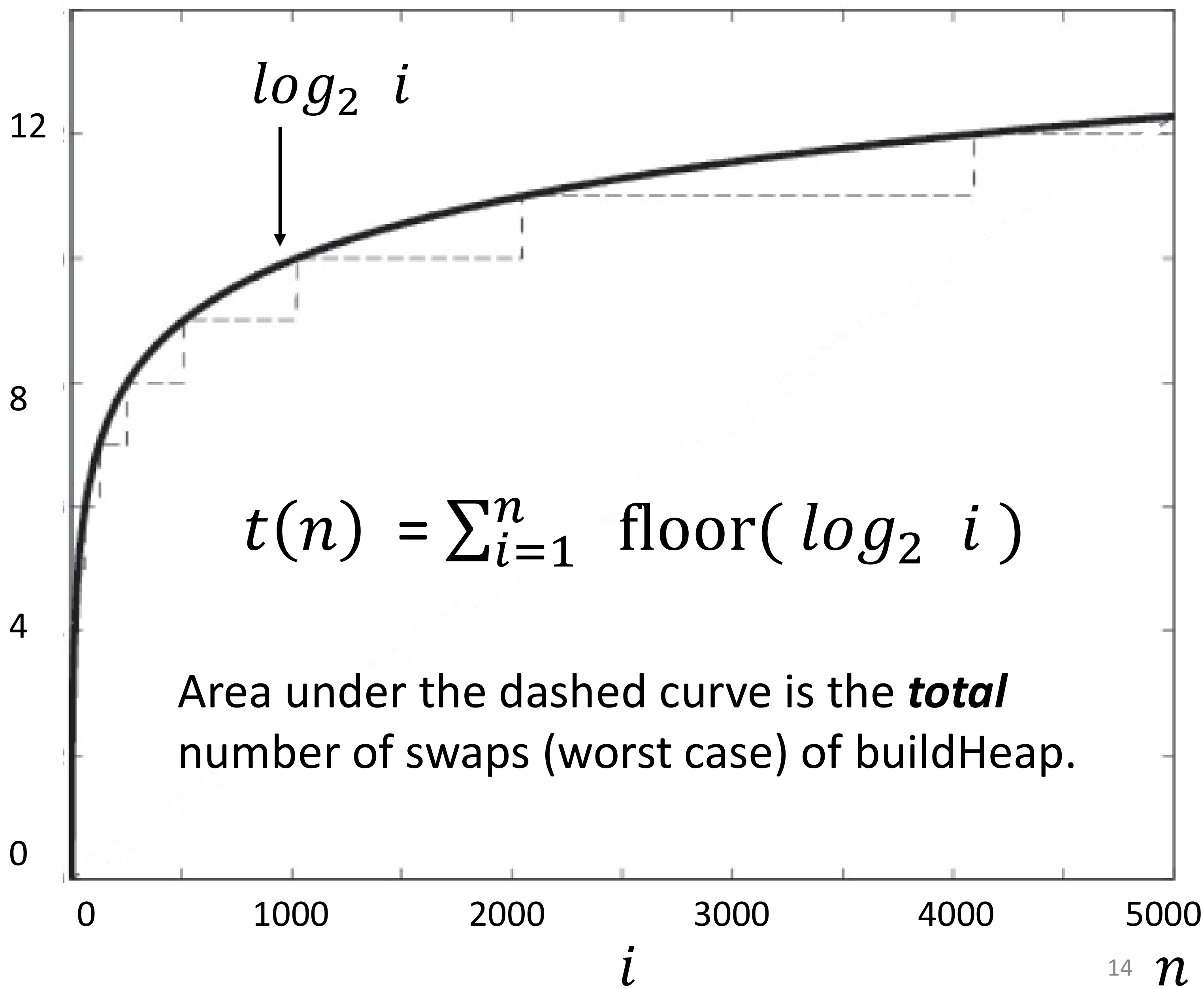


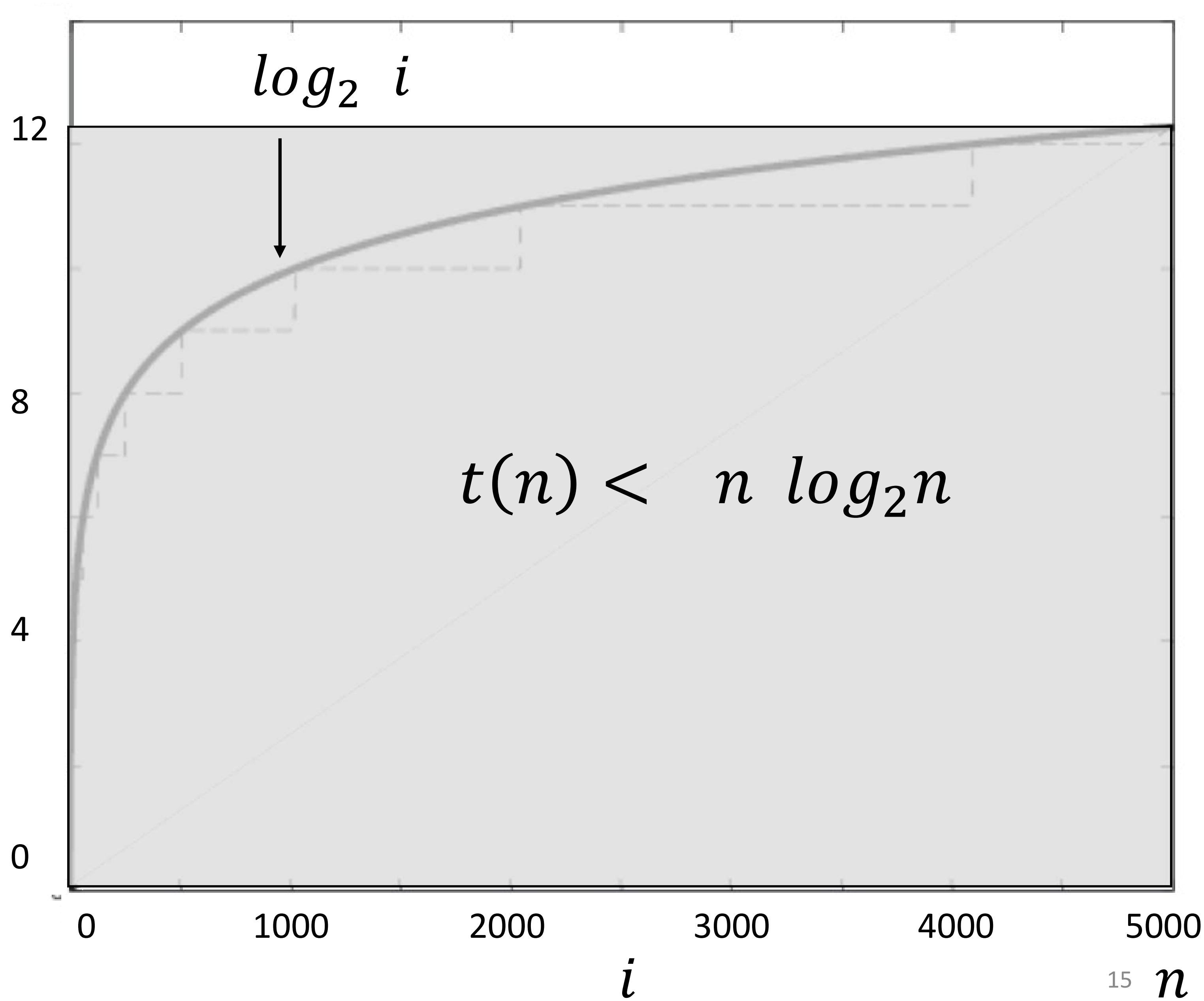
Key at position i requires at most $\text{floor}(\, log_2 \;\; i \,)$ swaps.

Thus, the worst case number of swaps needed to build a heap of size n using upHeap is $\sum_{i=1}^{n} \; \text{floor}(\, log_2 \;\; i \,)$

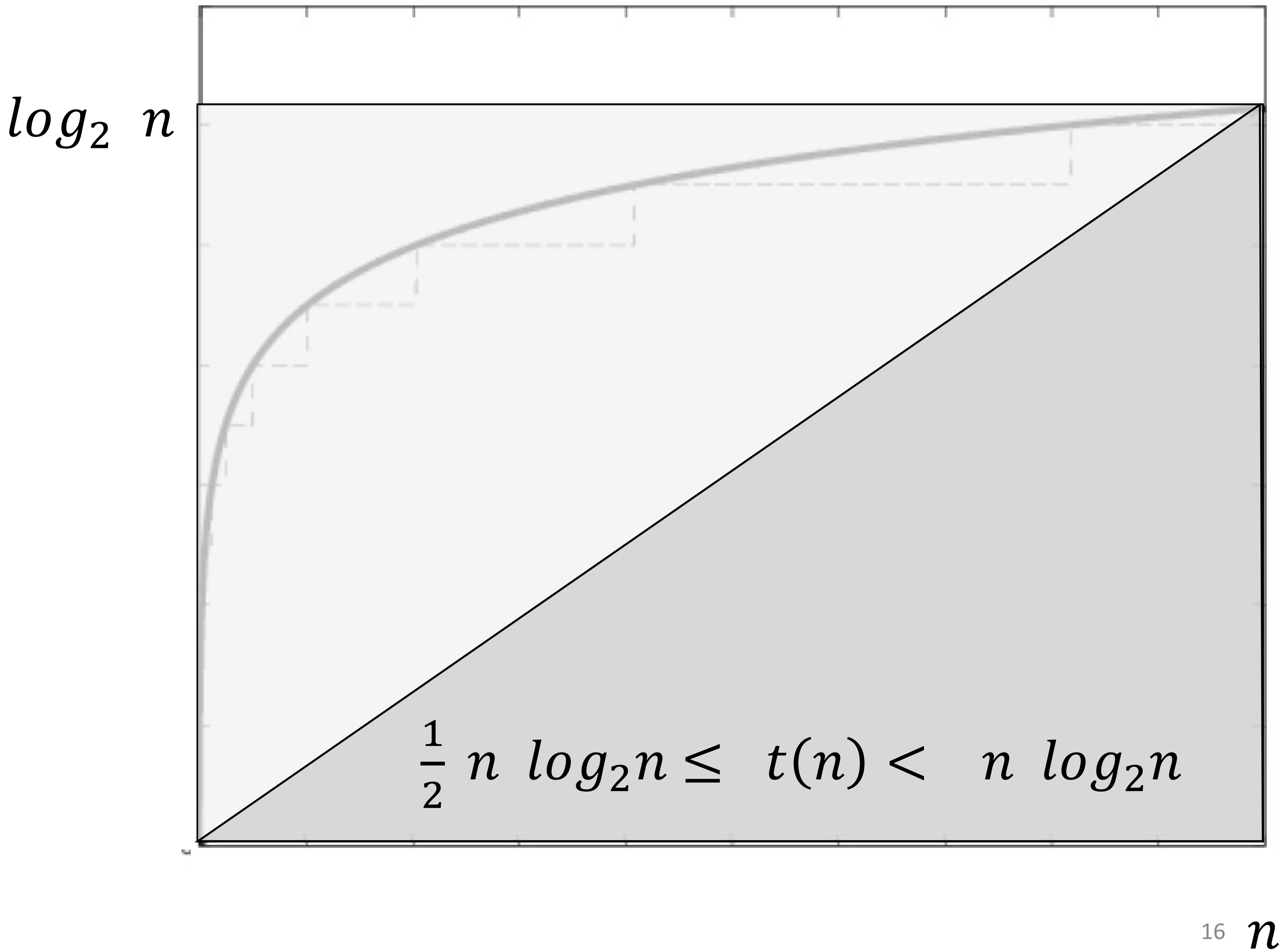


$log_2\ i$
floor($log_2\ i$)
12
8
4
0
0
1000
2000
3000
4000
5000
i


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



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



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

The worst case number of swaps of buildHeap is between $\frac{1}{2}\, n \;\; log_2 n$ and $n \;\; log_2 n$.

So the worst case is $\mathrm{O}(n \;\; log_2 n)$.

This worst case can occurs, for example, if the given list is ordered from large to small.

Plan for today

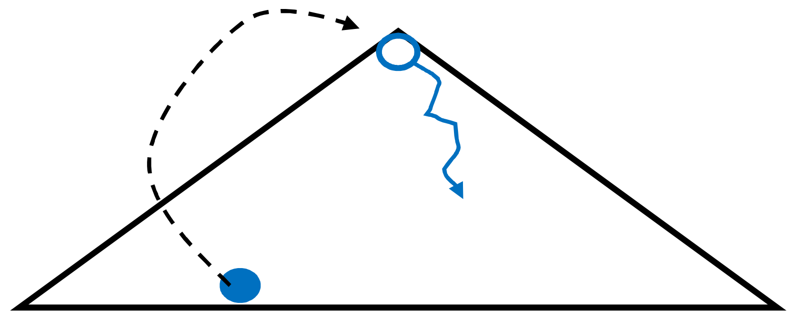
- building a heap -- best and worst cases
- removeMin() using array indices
- heapsort

Recall from last lecture

add(key)

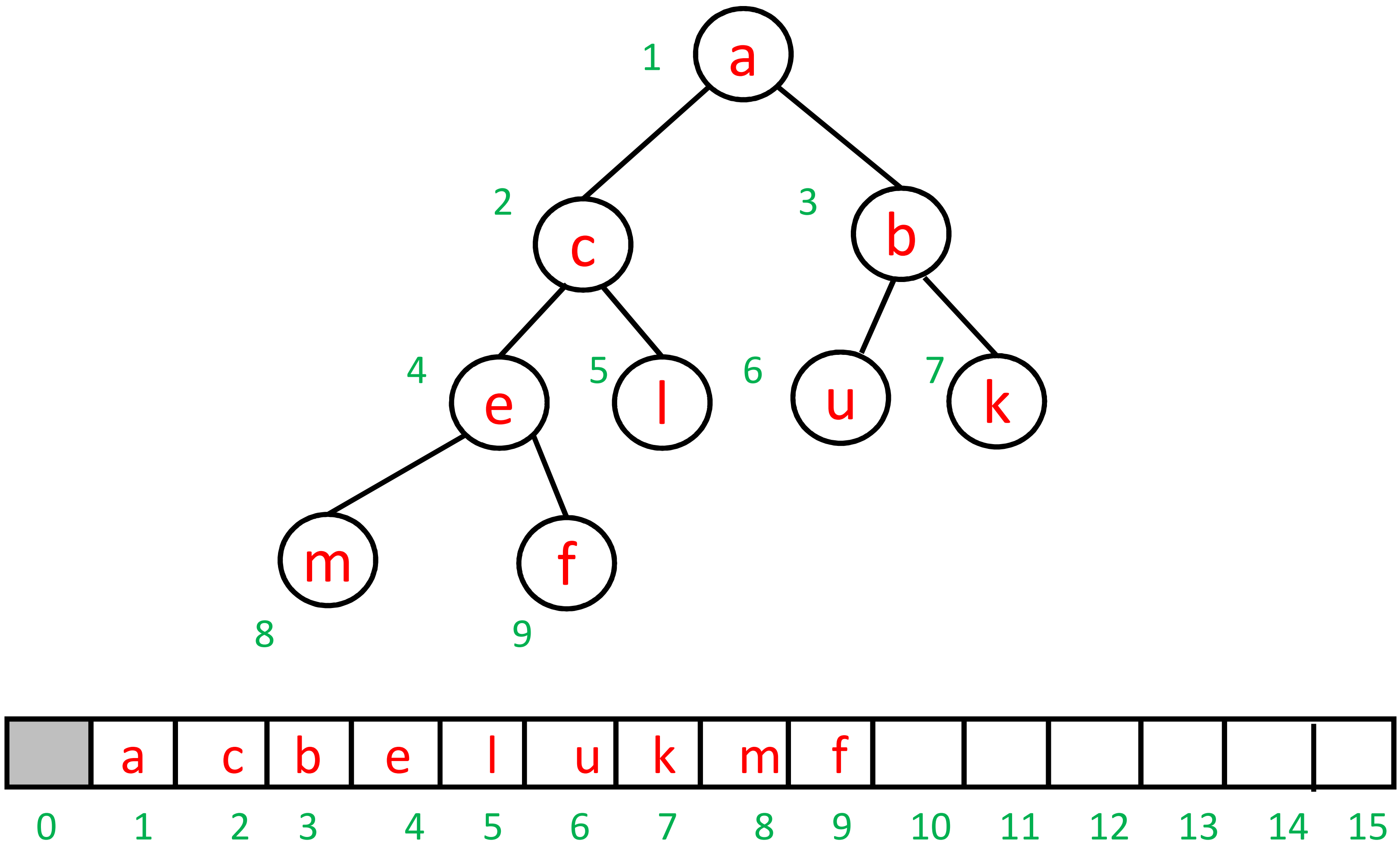
"upHeap"

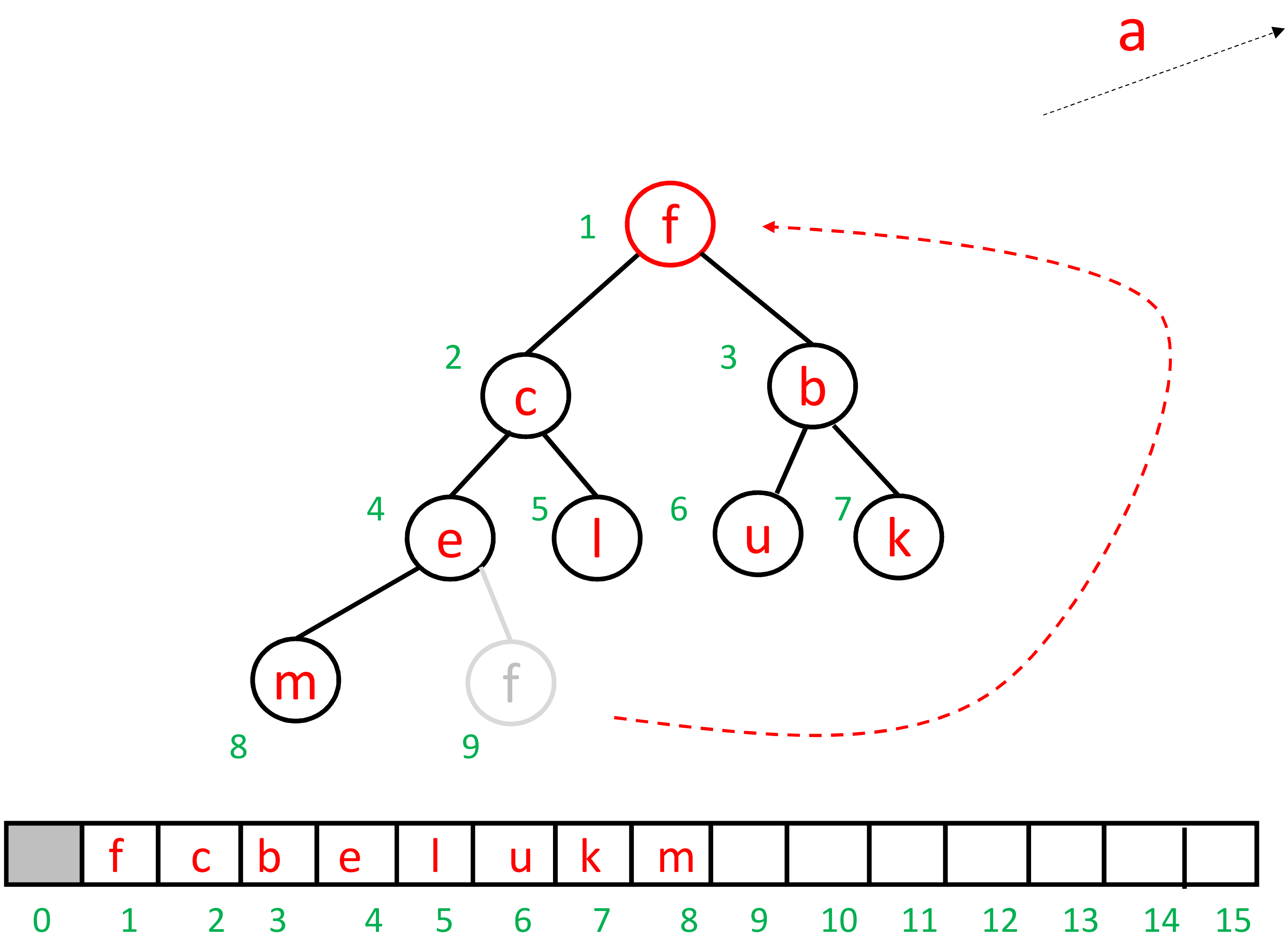
removeMin()



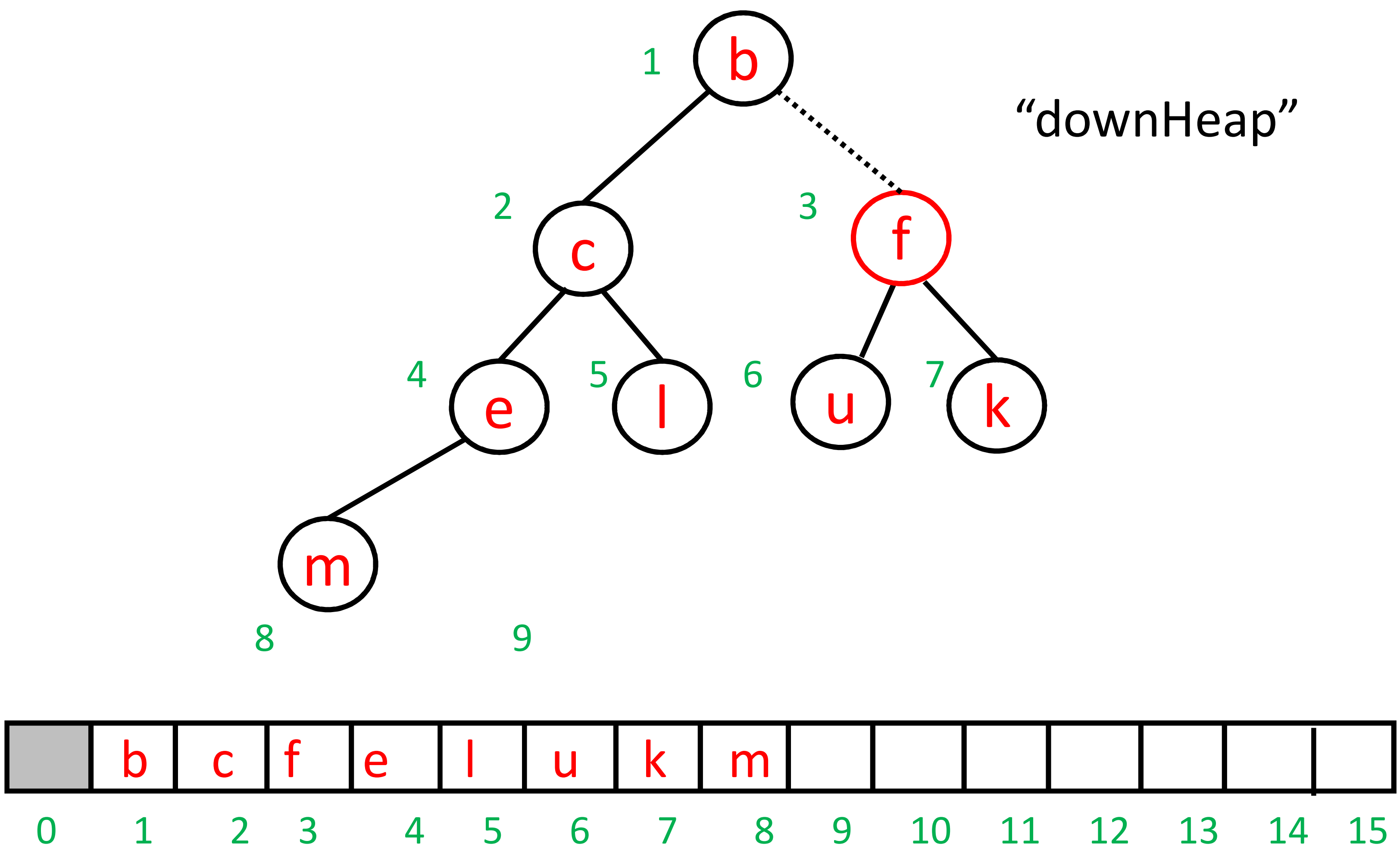
"downHeap"

e.g. removeMin()




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

“downHeap”



removeMin() with array indexing

Let arr[ ] be the array.

Let size be the number of keys in the heap.

```
removeMin( arr ){
    tmp = arr[1]
    arr[1] = arr[size]
    size = size - 1

    return tmp
}
```

removeMin() with array indexing

Let arr[ ] be the array.

Let size be the number of keys in the heap.

```
removeMin( arr ){
    tmp = arr[1]
    arr[1] = arr[size]
    size = size – 1                 //  note the new value
    downHeap( arr, size )           //  next slides
    return tmp
}
```

```
downHeap( arr, size ){          //  size parameter explained later

  i = 1
  while ( 2*i <= size){           // check if there is a left child

       Identify the smaller child  (left or right?)

       Swap if necessary.

  }
}
```

```
downHeap( arr, size ){

  i = 1
  while ( 2*i <= size){                    // check if there is a left child
     child = 2*i            //  left child's index
     if child < size {      //  ... then there is a right child
          if ( arr[child + 1] < arr [child])      // right < left ?
               child = child + 1                   //  choose smaller child
     }

  }
}
```

```
downHeap( arr, size ){

  i = 1
  while ( 2*i <= size){                    // check if there is a left child
     child = 2*i              //  left child's index
     if child < size {        //  ... then there is a right child
          if ( arr[child + 1] < arr [child])      // right < left ?
               child = child + 1                  //  choose smaller child
     }
     if ( arr[child] <  arr[ i ]){      //  swap with child, if necessary.
        swapkeys(arr,  i , child)
        i = child
     }
     else return                       //   avoids infinite loop.
   }
}
```

Plan for today

- building a heap -- best and worst cases
- removeMin() using array indices
- heapsort

Heapsort

Given a list with n keys

- Build a heap using an array.

Heapsort

Given a list with n keys

- Build a heap using an array.

- Call removeMin() n times, for $i = 1\ to\ n$.
  For the i^{th} remove, store the removed key in array slot $n + 1 - i$.
  This sorts the keys in the reversed order.

- Reverse the order of keys.
  You could build a maxHeap and removeMax instead.

Heapsort

Here is the algorithm. Let's walk through an example.

```
heapsort(list){
   arr = buildheap(list)
   n = list.size
   for i = 1  to n − 1 {
       swapkeys( arr, 1,  n + 1 – i)      ←  note size parameter
       downHeap( arr,  n − i)
   }
   return  reverse(arr)
}
```

Example of input list:

b d a f l u k e w

```
heapsort(list){
   arr = buildheap(list)    →  next slide
   n = list.size
   for i = 1  to n − 1 {
       swapkeys( arr, 1,  n + 1 – i)
       downHeap( arr,  n − i)
   }
   return  reverse(arr)
}
```

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

This is now a heap.

```
heapsort(list){
   arr = buildheap(list)       // done (see above)
   n = list.size
   for i = 1 to n − 1 {
      swapkeys( arr, 1, n + 1 − i)
      downHeap( arr, n − i)
   }
   return reverse(arr)
}
```

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

$i = 1$

```
heapsort(list){
   arr = buildheap(list)
   n = list.size
   for i = 1  to n − 1 {
      swapkeys( arr, 1,  n + 1 − i)
      downHeap( arr,  n − i)
   }
   return  reverse(arr)
}
```

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

$i = 1$

```
heapsort(list){
   arr = buildheap(list)
   n = list.size
   for i = 1  to n − 1 {
      swapkeys( arr, 1,  n + 1 – i)
      downHeap( arr,  n − i)
   }
   return  reverse(arr)
}
```

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

$i = 1$

```
heapsort(list){
   arr = buildheap(list)
   n = list.size
   for i = 1  to n − 1 {
      swapkeys( arr, 1,  n + 1 − i)
      downHeap( arr,  n − i)
   }
   return  reverse(arr)
}
```

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
f	d	k	e	l	u	w	b	a

$i = 2$

```
heapsort(list){
   arr = buildheap(list)
   n = list.size
   for i = 1  to n − 1 {
       swapkeys( arr, 1,  n + 1 – i)
       downHeap( arr,  n − i)
   }
   return  reverse(arr)
}
```

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	f	k	e	l	u	w	b	a

$i = 2$

```
heapsort(list){
   arr = buildheap(list)
   n = list.size
   for i = 1  to n − 1 {
      swapkeys( arr, 1,  n + 1 − i)
      downHeap( arr,  n − i)
   }
   return  reverse(arr)
}
```

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	$i = 2$

```
heapsort(list){
    arr = buildheap(list)
    n = list.size
    for i = 1 to n − 1 {
        swapkeys( arr, 1, n + 1 − i )
        downHeap( arr, n − i)
    }
    return reverse(arr)
}
```

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	$i = 8$

The keys are in the reverse order. So we need to reverse their order and return.

Heapsort (worst case)

```
heapsort(list){
   arr = buildheap(list)
   n = list.size
   for i = 1  to n − 1 {
      swapkeys( arr, 1,  n + 1 – i)
      downHeap( arr,  n − i)
   }
   return  reverse(arr)
}
```

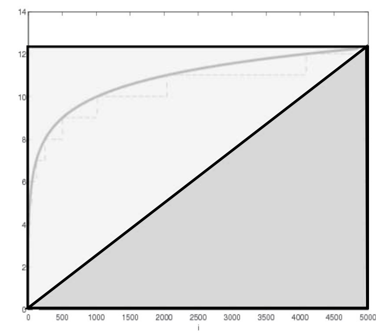
Worse case is that we have to swap all the way from level 0 to the level of node $n - i$.

e.g. This happens if the heap we build is already sorted.

Heapsort (worst case)

heapsort(list){
  arr = buildheap(list)
  n = list.size
  for i = 1 to $n - 1$ {
    swapkeys(arr, 1, n + 1 – i)
    downHeap(arr, $n - i$)
  }
  return reverse(arr)
}

$$\sum_{i=1}^{n-1} \text{floor}(\log(\, i\,))$$



$$n - 1$$

$$\sum_{i=1}^{n-1} \text{floor}\,(\log(n - i)\,)$$

$$n$$

This is the same as the above summation!

Heapsort (worst case)

$t(n)$ = $c_1\, n + \; c_2\; n \log n$ in the worst case.

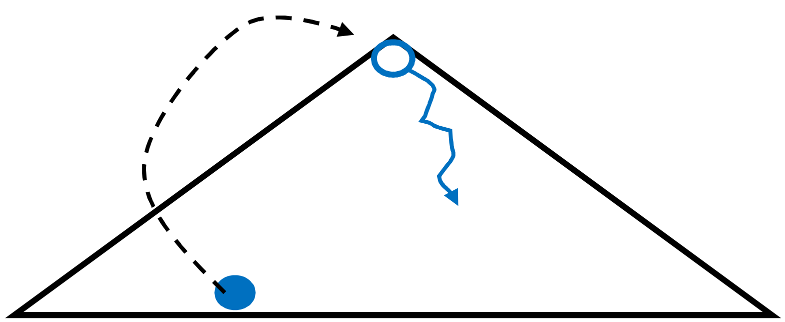
So, we say $t(n)$ is $O(n \; \log n)$ in the worst case.

This worst case is the same as mergesort, and it is better than quicksort's worst case which is $O\left(n^2\right)$.

Heapsort (best case?)

Heapsort is $O(n \ \log n)$ even in best case. Intuitively, why ?

The first step of heapsort is to build a heap. Once you have a heap, *approximately half the keys lie at the deepest level and these tend to be the largest keys.* So, each time you call removeMin and move a key from the bottom to the top, it will tend to downHeap back down close to bottom. So the majority of the n keys will require close to $\log n$ swaps!



Heapsort versus Quicksort ?

Heapsort is $O(n\ \log n)$ in both best and worst case.

Quicksort is $O(n\ \log n)$ in best case but $O\left(n^2\right)$ in worst case.

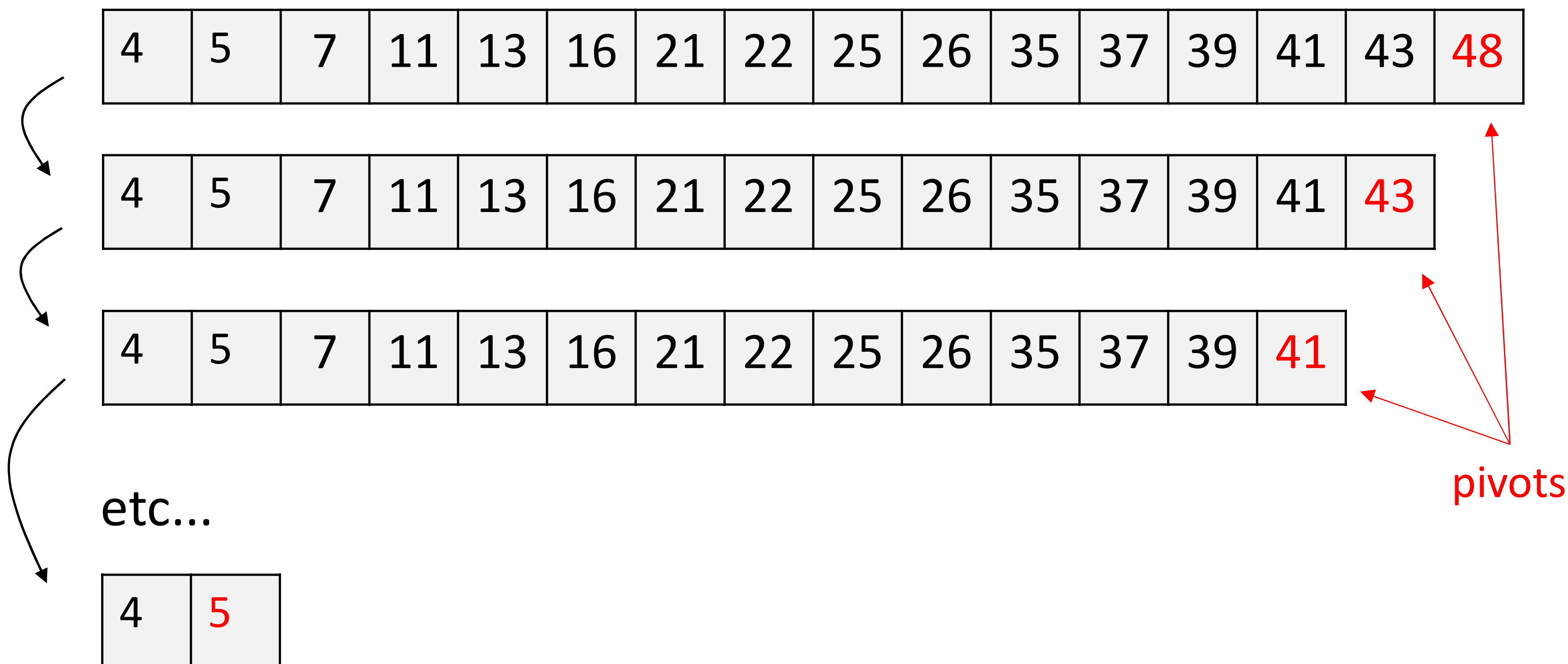
Yet quicksort "quicker" than heapsort *in practice. How* ?

ASIDE: The following slides are not on the final exam, and you will learn more about it in COMP 251. I mention it now for your interest only.

Quicksort worst case

An example of when Quicksort is $O(n^2)$: the list is already sorted.

In this case, each partition splits the list into two lists of size $n - 1$ and 0.



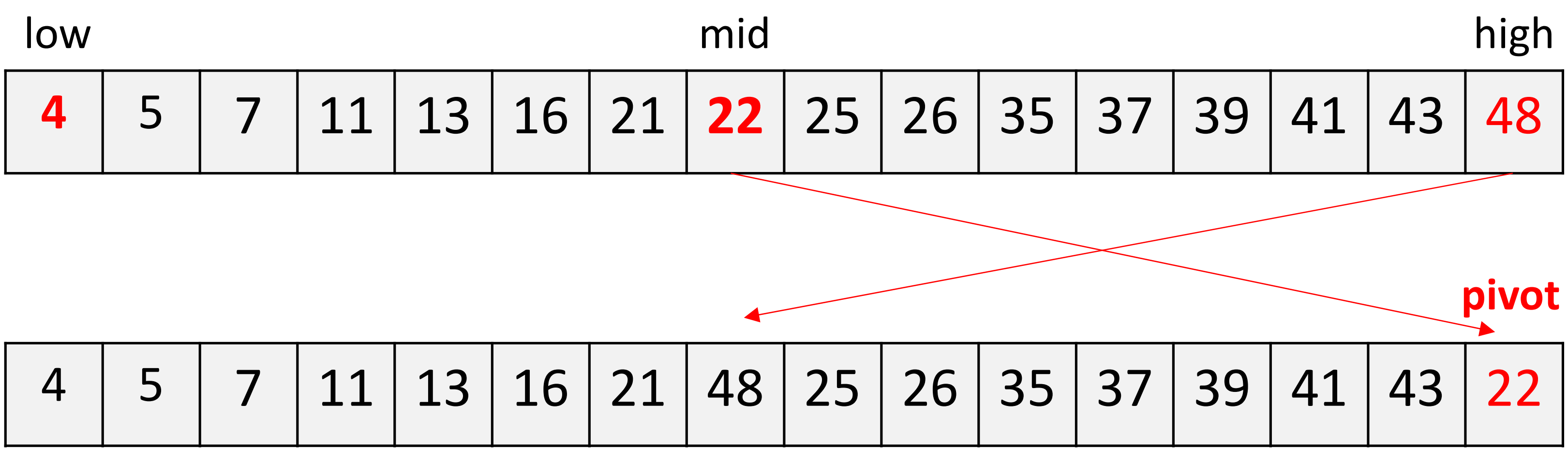
Recall Quicksort ("in place", using an array)

```
quicksort(list, low, high ){      // void
   if  low < high  {
      wall  =   partition (list, low, high)
      quicksort(list, low, wall - 1)
      quicksort(list, wall + 1,  high)
    }
}

partition(list, low , high )
  pivot =  list[high]
  wall = low - 1
  for (i = low ; i <= high;   i++)
      if ( list[i]  <=  pivot ){
          wall ++
          list.swap(wall, i)
      }
   return wall
}
```

The pivot was chosen to be the last element in the array. **But this is not necessary. Instead we can swap the element at high with another element (next slide)**

If we knew which element was the median, we could use it. But finding the median of n numbers takes $\mathrm{O}(n)$ time in the worst case, which would defeat the purpose! Instead, we choose the median of a few of the elements, namely those in positions {low, mid, high}. e.g. median(**4, 22, 48**) is 22. It takes three comparisons i.e. O(1) to do so. We then swap this median with the last element, and otherwise the quicksort algorithm is the same. This is called the "median of 3" method.



This will give a much better partition.
For more general examples, it is much more *likely* to give a better partition.

Heapsort versus Quicksort ?

Heapsort is $O(n \log n)$ in both best and worst case.

Quicksort is $O(n \log n)$ in best case but $O(n^2)$ in worst case.

Yet quicksort "quicker" than heapsort *in practice. How* ?

So when people talk about quicksort, then are including a method like 'median of three' (or random) for choosing the pivot. This hugely speeds up quicksort in practice, especially in the "worst case" just mentioned.

Coming up...

Lectures

Mon. & Wed   March 21 & 23

Maps & Hashing

Fri. March 25

Graphs 1

Assessments

Quiz 4   (lectures 20-25)

today

Assignment 4 will be posted next week