COMP 250

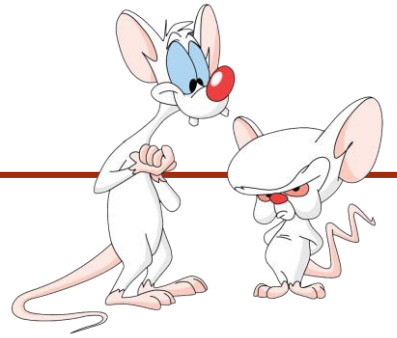
INTRODUCTION TO COMPUTER SCIENCE

Week 12-3 : Heaps

Giulia Alberini, Fall 2020

Slides adapted from Michael Langer's

WHAT ARE WE GOING TO DO IN THIS VIDEO?



- How to build a heap in relatively best and worst case
- write removeMin() using array indices
- Faster algorithm for building a heap

BUILD A HEAP

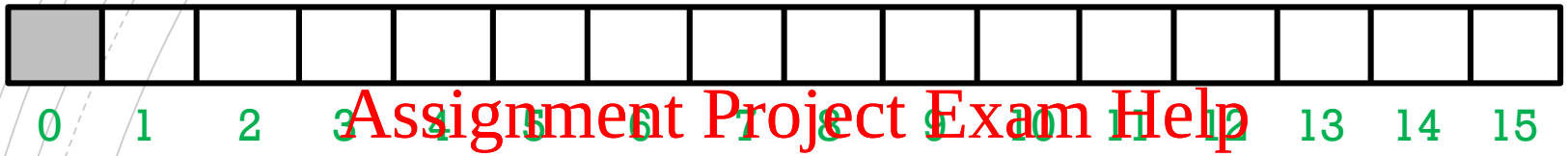
HOW TO BUILD A HEAP?

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 BUILDHEAP IS … ?



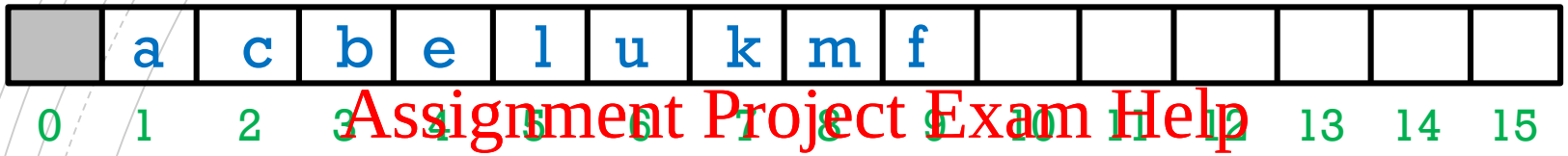
Suppose we want to add some elements to an empty heap:

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 BUILDHEAP IS $O(n)$



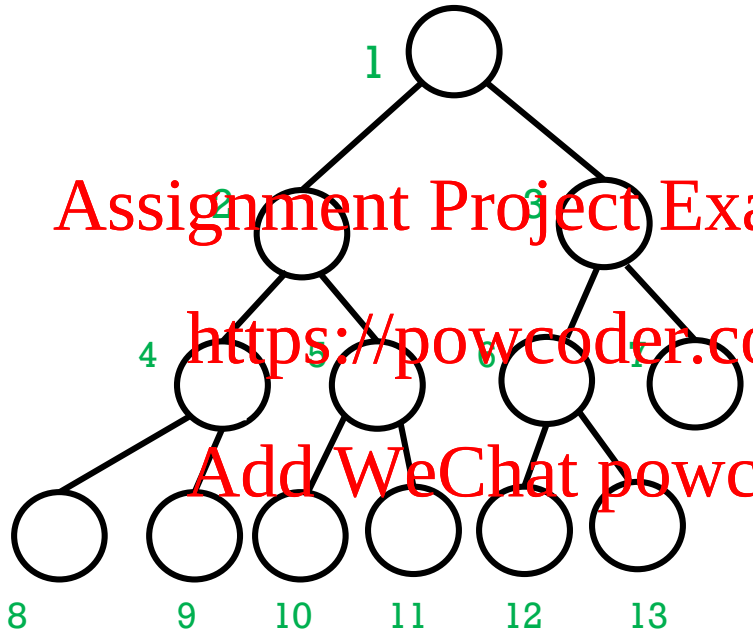
Suppose we want to add some elements to an empty heap:

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 BUILDHEAP IS … ?



level

0

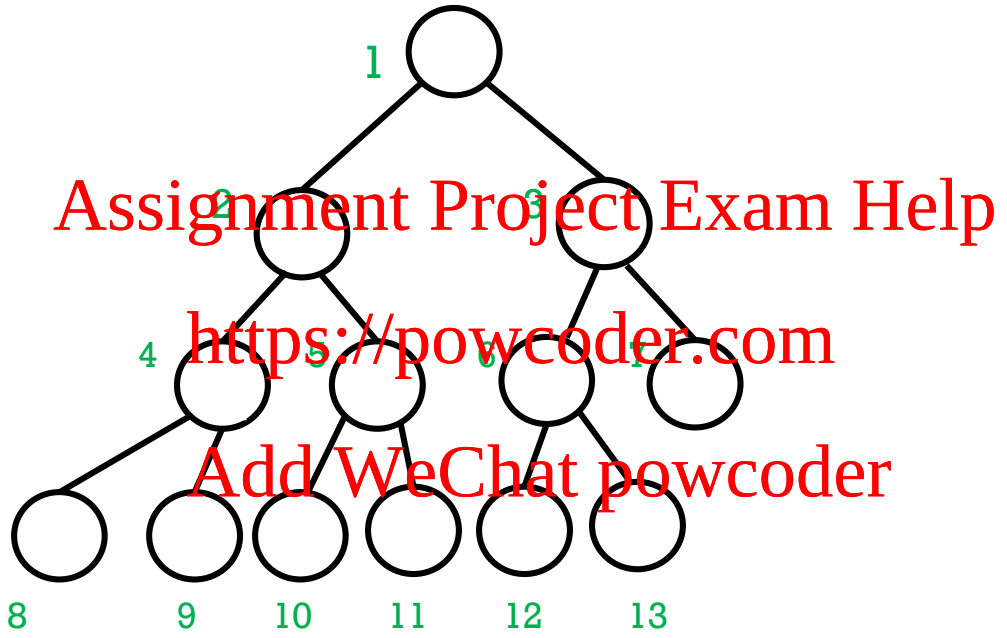
1

2

3

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

WORST CASE OF BUILDHEAP IS … ?



level

0

1

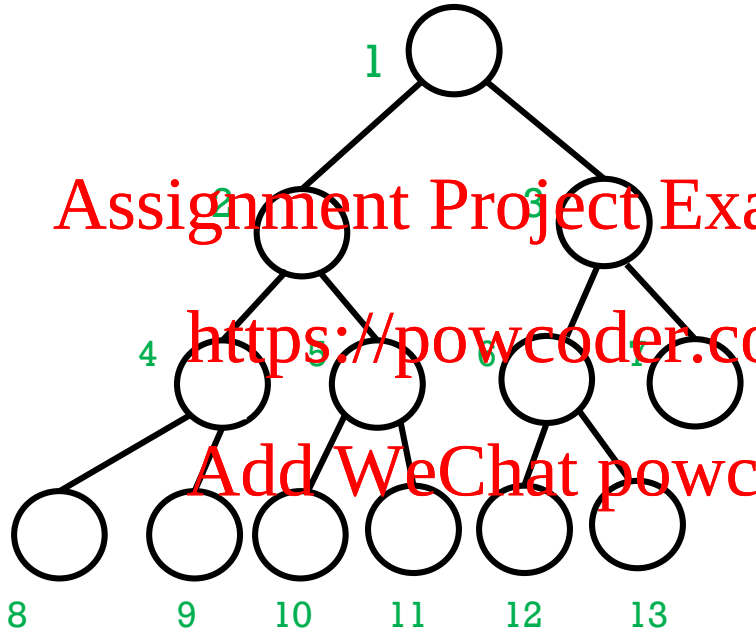
2

3

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 BUILDHEAP IS … ?



level

0

1

2

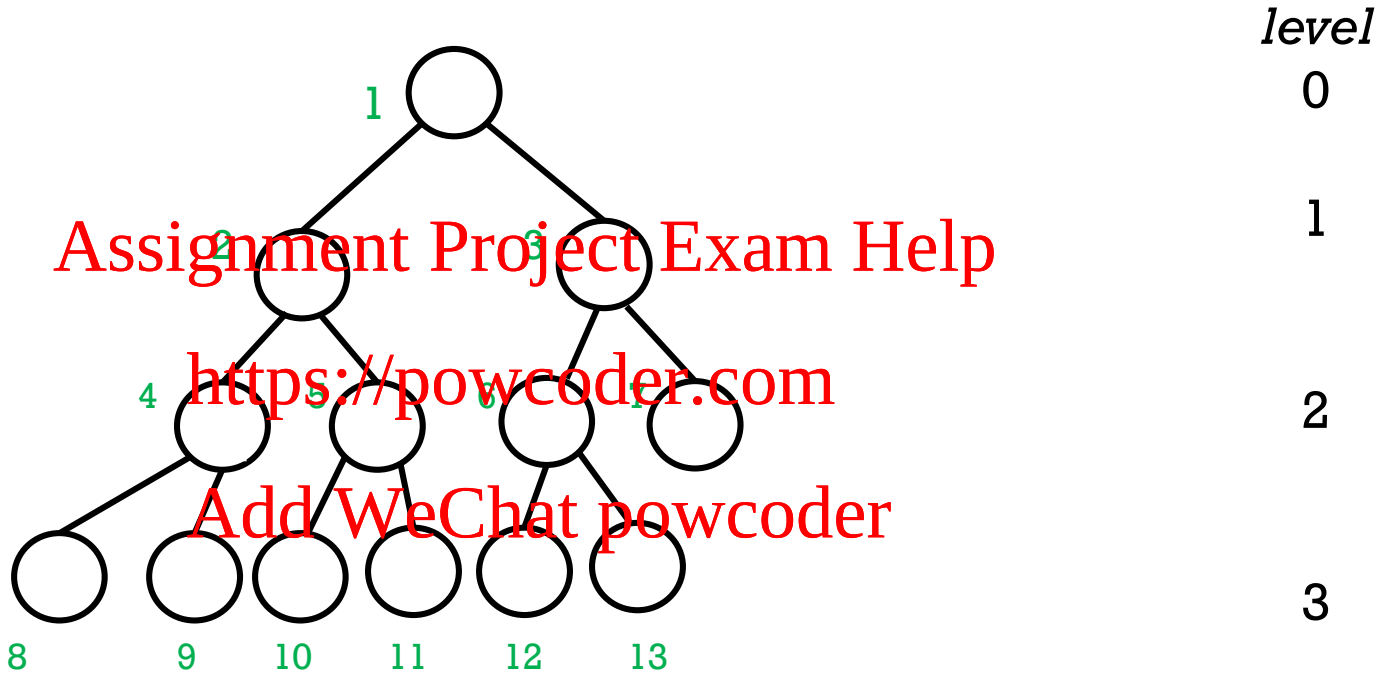
3

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

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

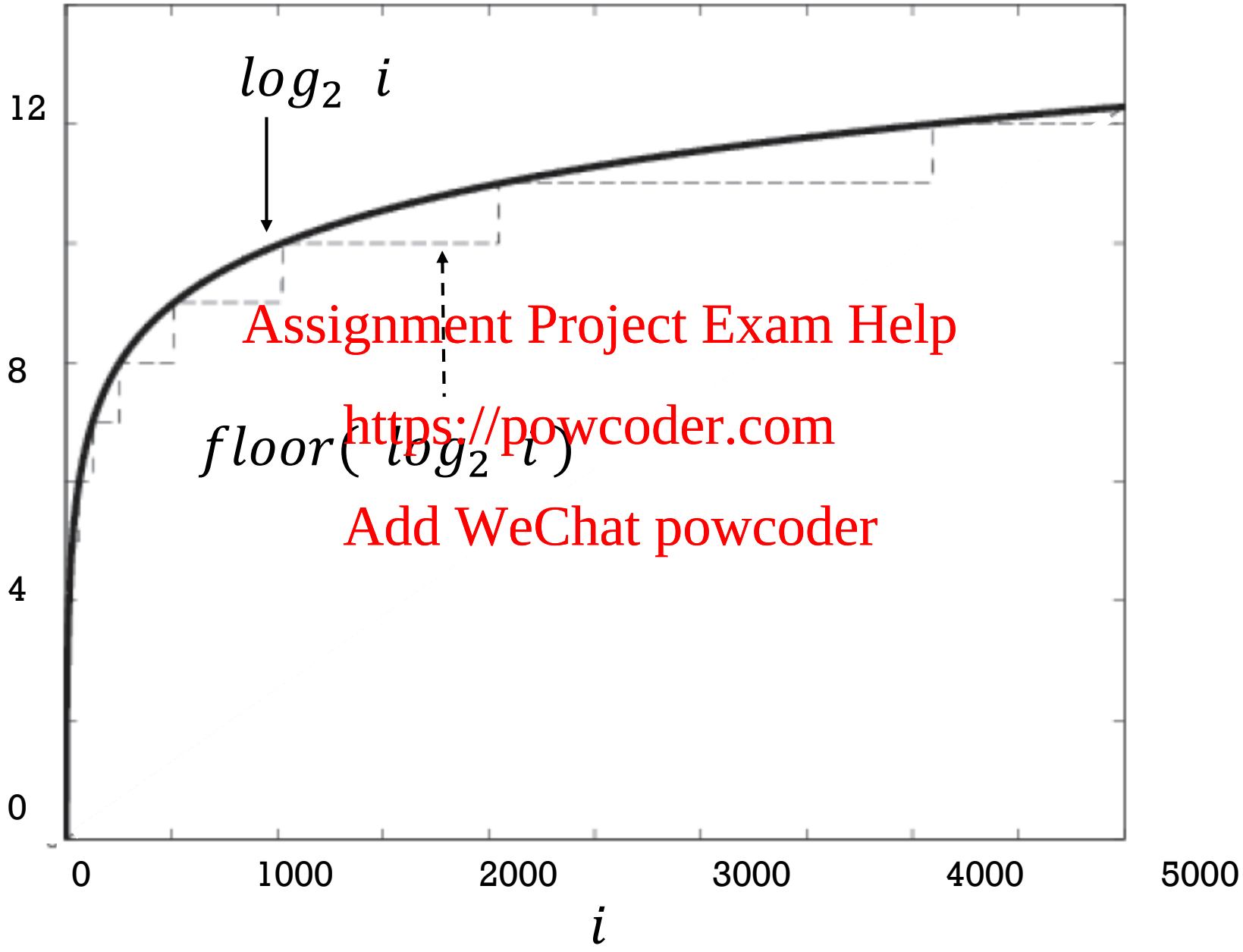
Thus, $level = floor(\log_2 i)$

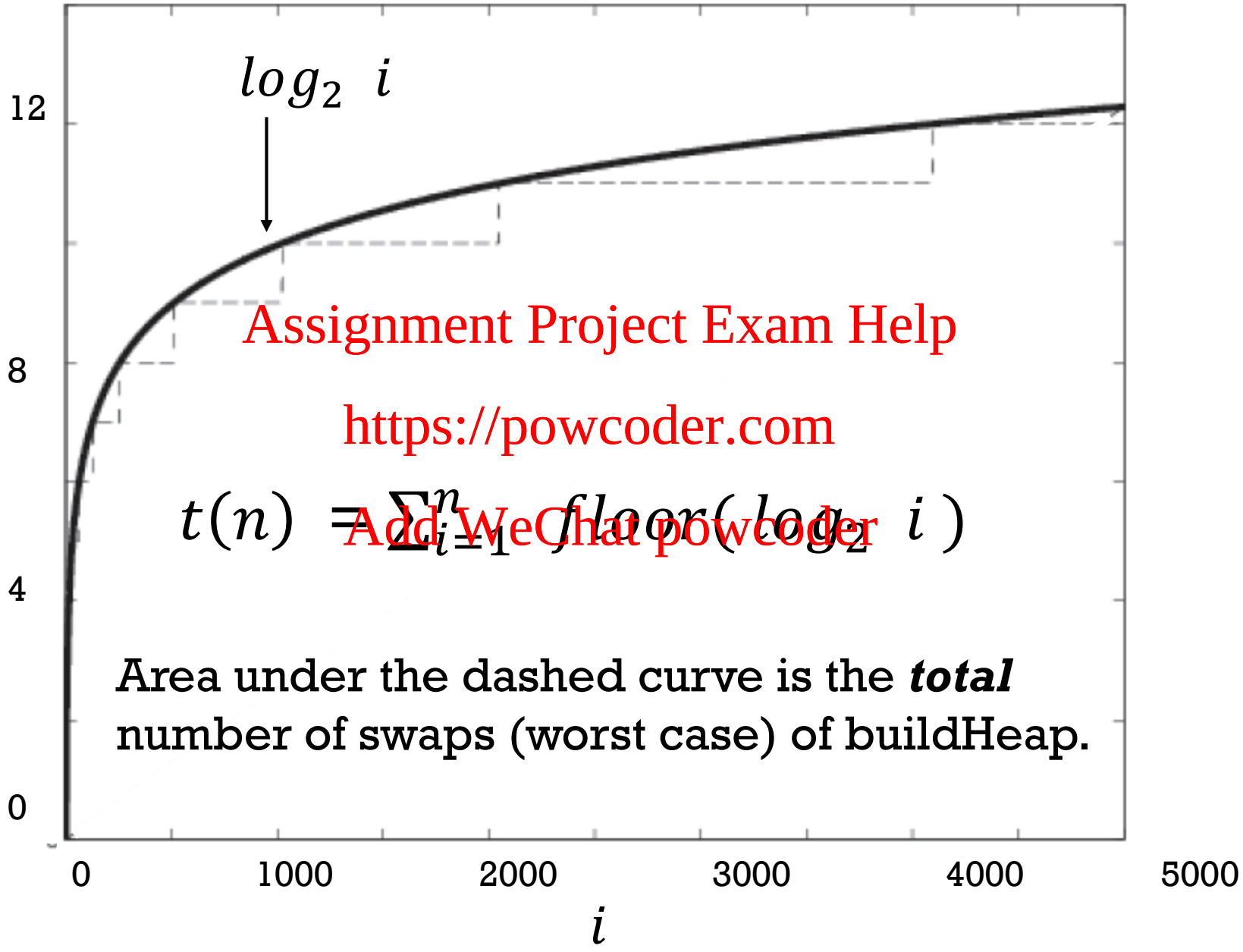
WORST CASE OF BUILDHEAP IS … ?



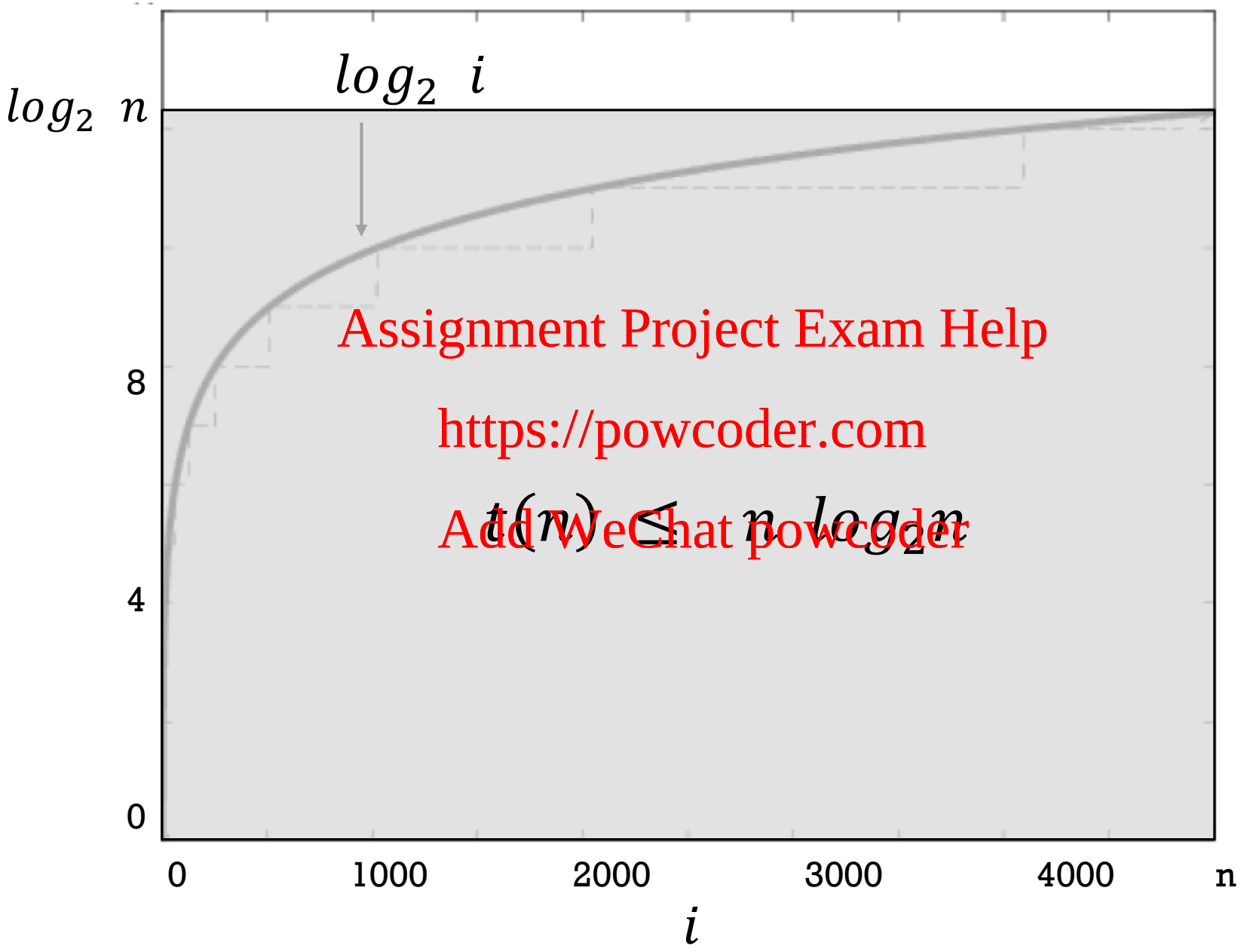
Suppose there are n elements to add, then in the worst case the number of swaps needed to add all the elements is:

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

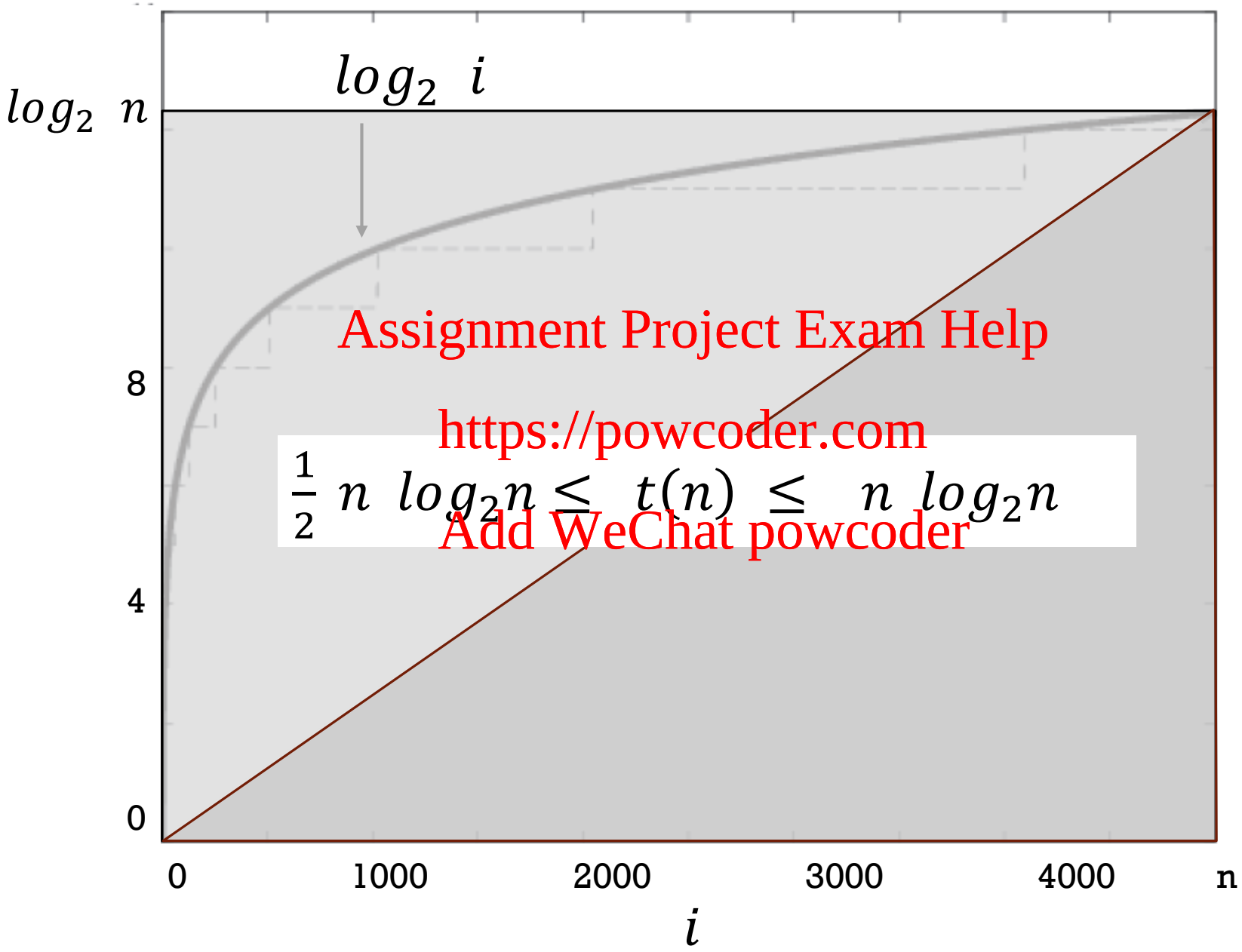




$$t(n) = \sum_{i=1}^{n} floor(log_2\ i)$$

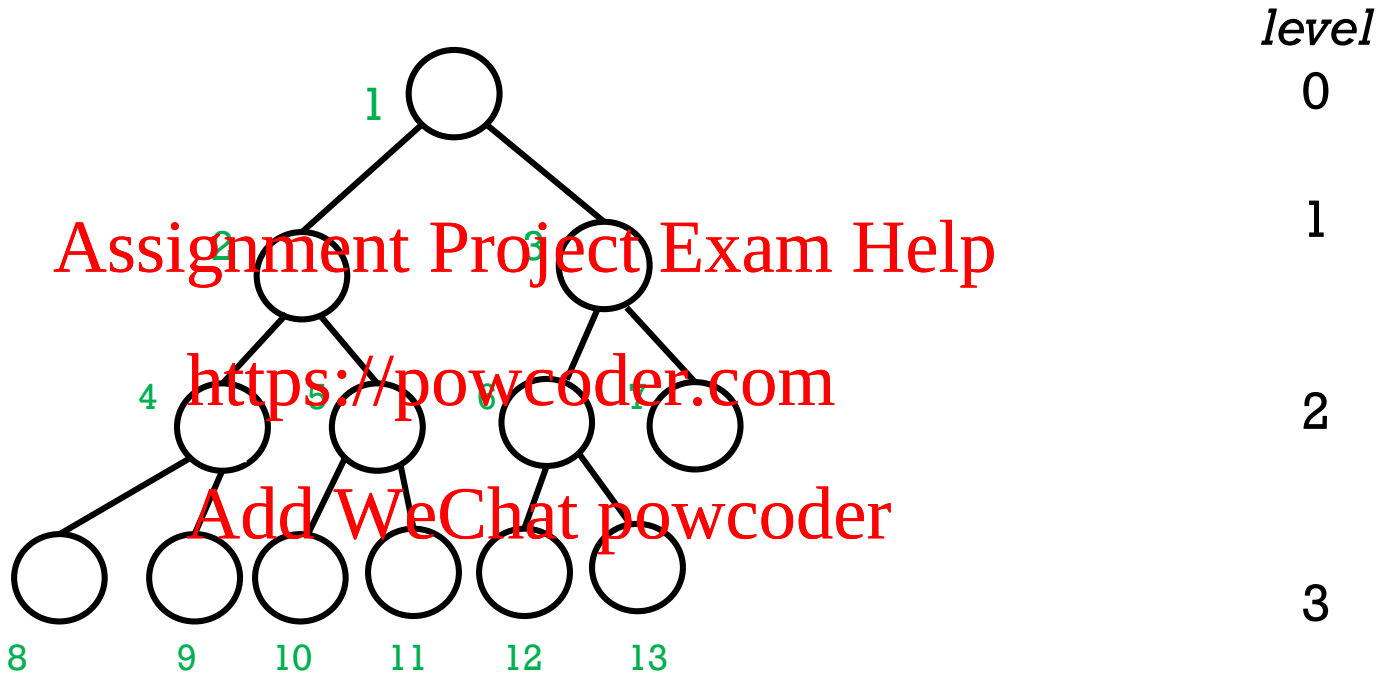


$t(n) \leq n\ log_2 n$



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

WORST CASE OF BUILDHEAP IS $O(n * log_2 n)$



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

removeMin()

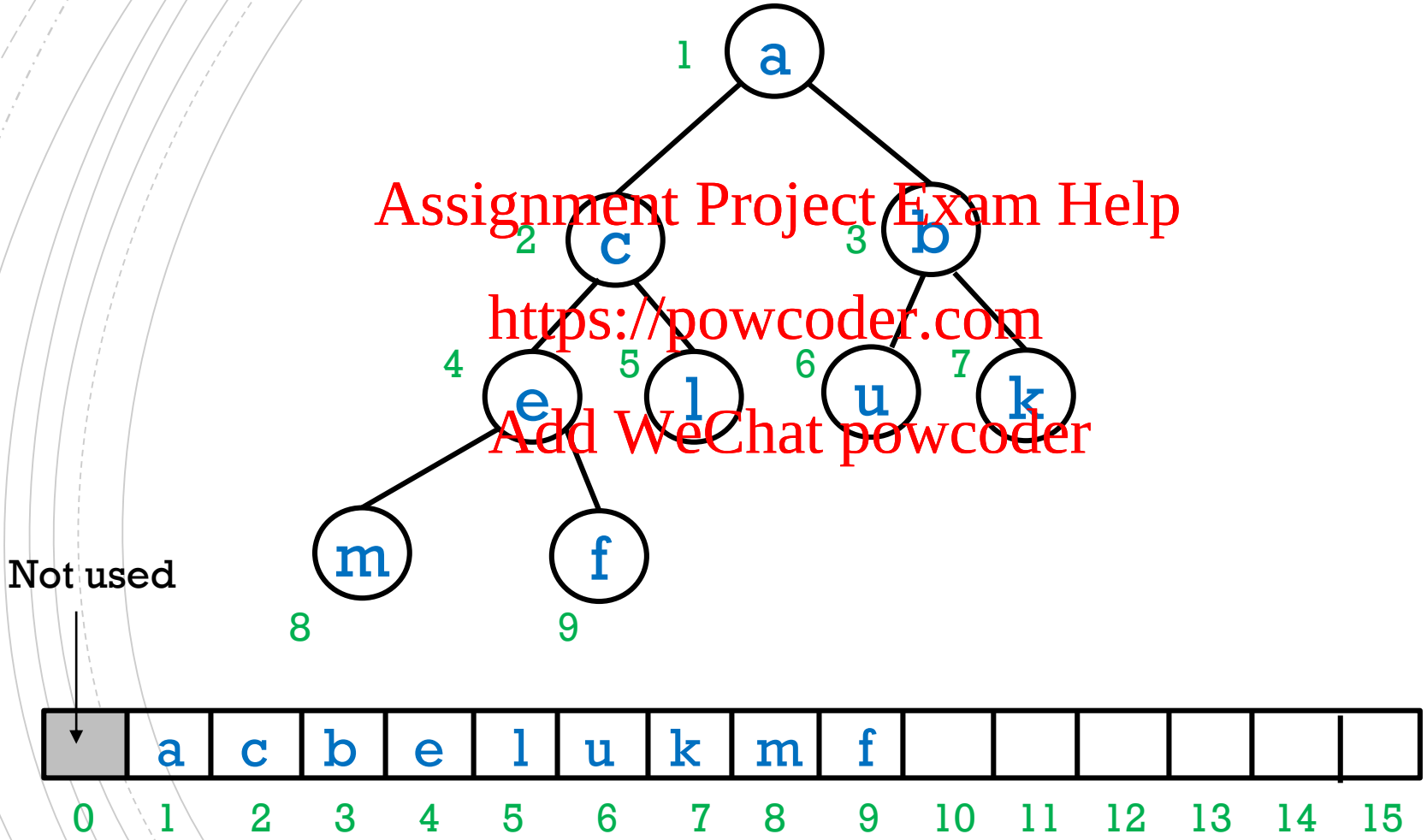
add()

removeMin()

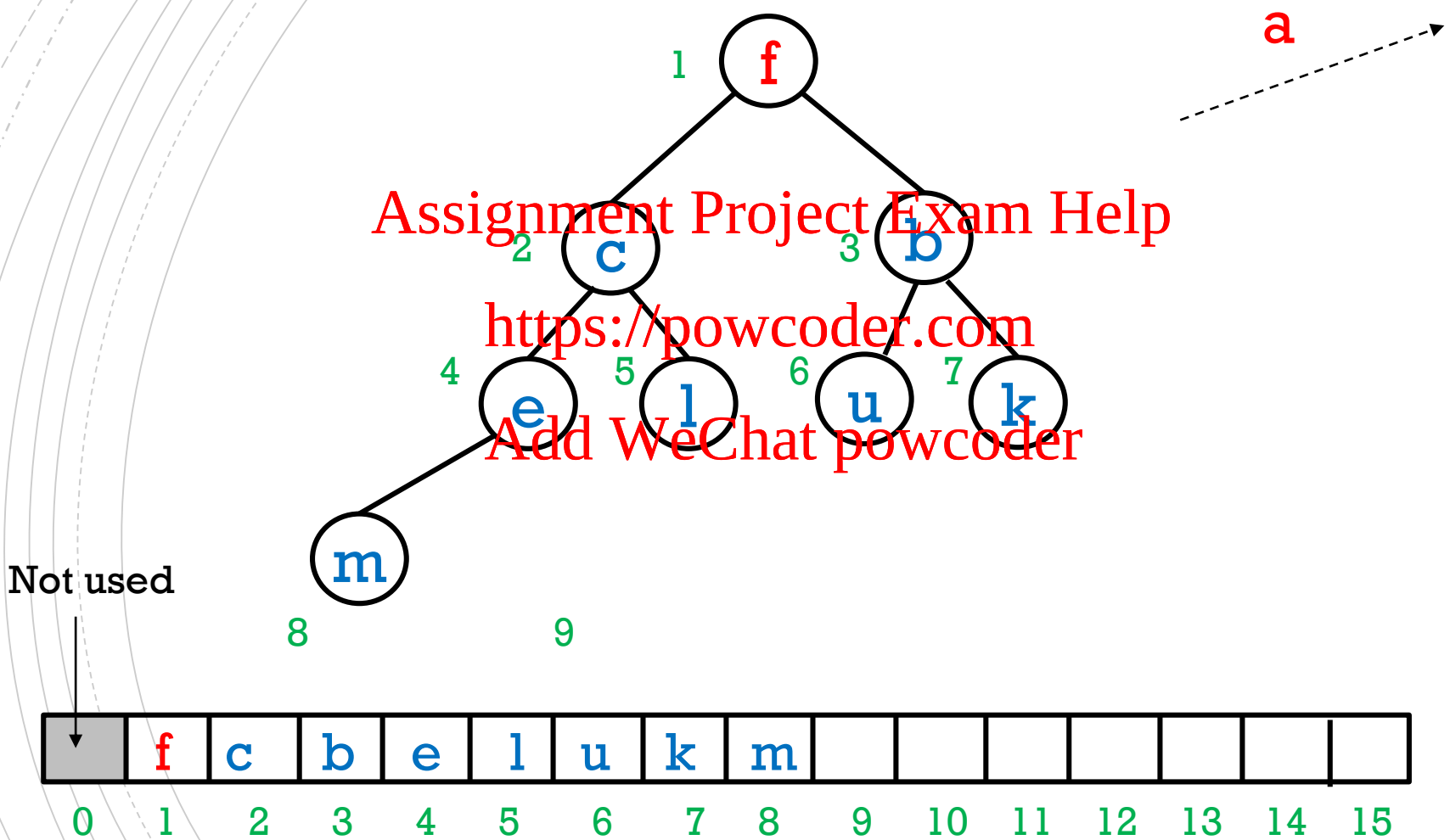
"upHeap"

"downHeap"

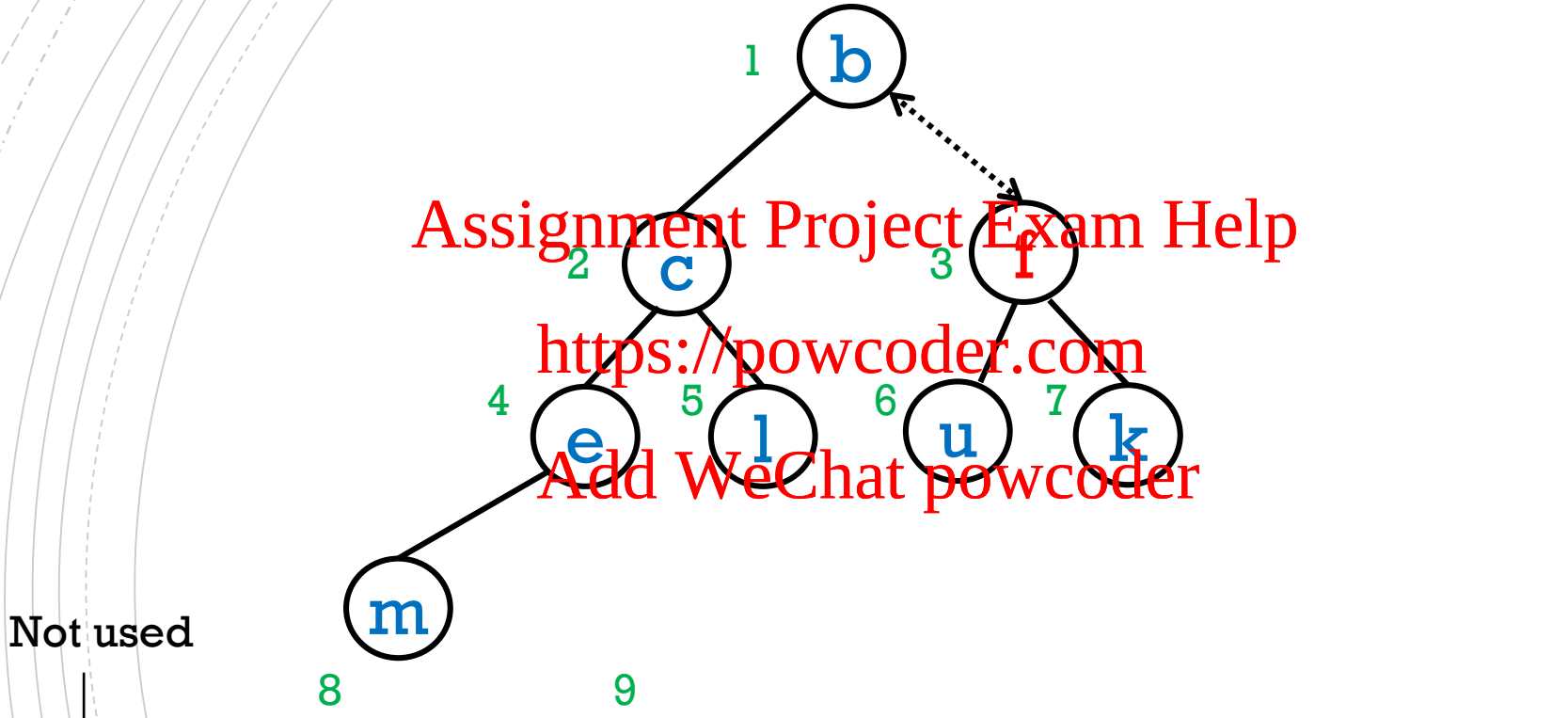
E.G. removeMin()



E.G. `removeMin()`



E.G. `removeMin()`



REMOVEMIN() - IMPLEMENTATION

Let `heap` be the underlying array, and 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 // 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]   // heap[0] not used.
      heap[1] = heap[size]
      heap[size] = null // not necessary
      size = size - 1
      downHeap(1, size)
      return  tmpElement
}
```

DOWNHEAP() - IMPLEMENTATION

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









    }
}
```

DOWNHEAP() - IMPLEMENTATION

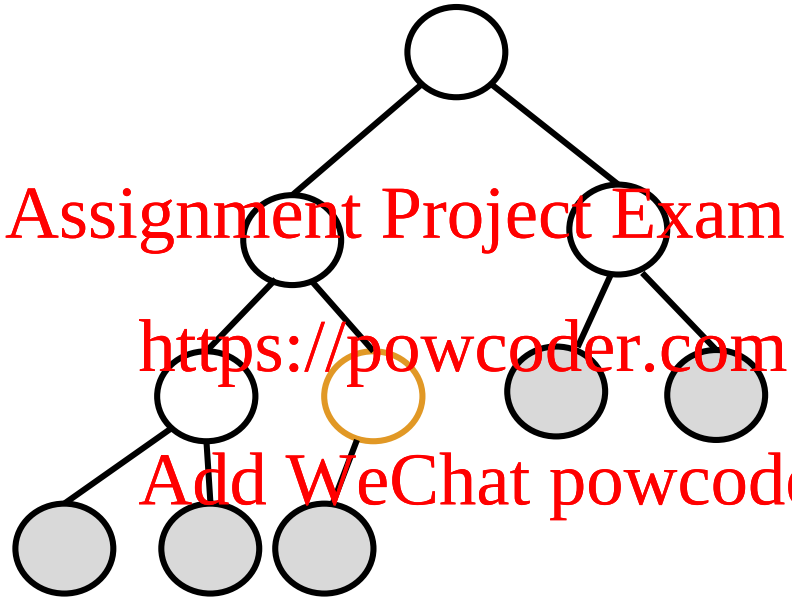
```
downHeap( startIndex , maxIndex ){
    i = startIndex
    while (2*i <= maxIndex){ // if there is a left child
        child = 2*i
        if (child < maxIndex)  { // if there is a right sibling
            if (heap[child + 1] < heap[child]) // if rightchild < leftchild
                child = child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

BUILD A HEAP (fast)

HOW TO BUILD A HEAP ? (FAST)



Observations:

- Half the nodes of a heap are leaves.
  (Each leaf is a heap with one node)
- The last non-leaf node has index size/2.

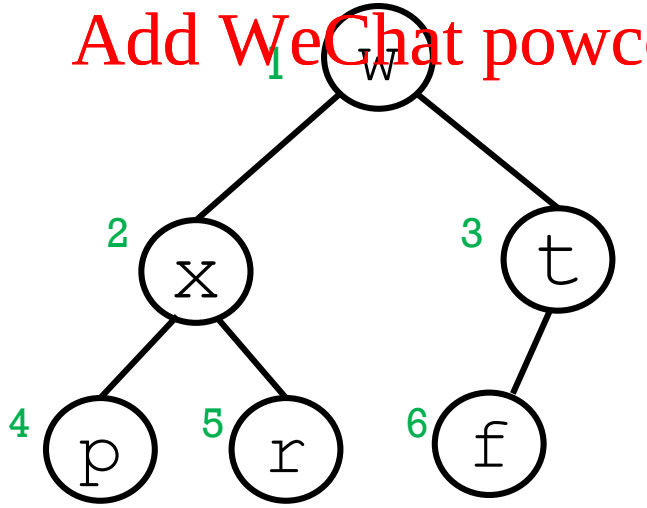
HOW TO BUILD A HEAP ? (FAST)


```
buildHeapFast(){
    // assume that heap[] array contains size elements
    for (k = size/2; k >= 1; k--)
        downHeap( k, size )
}
```

EXAMPLE

1	2	3	4	5	6
w	x	t	p	r	f

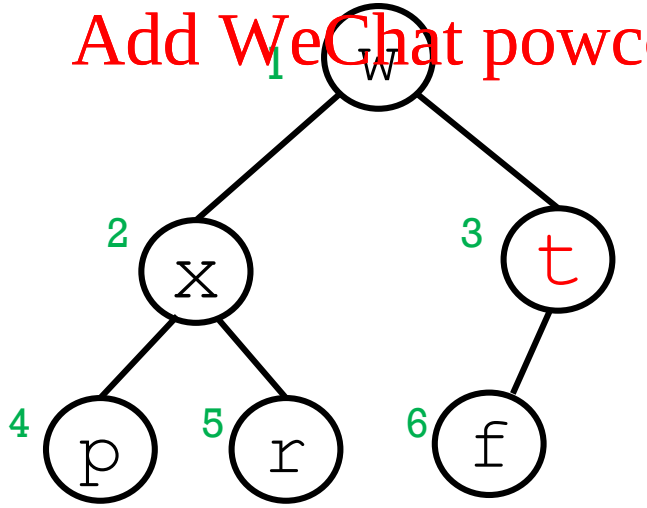
k = 3



EXAMPLE

1	2	3	4	5	6
w	x	t	p	r	f

k = 3

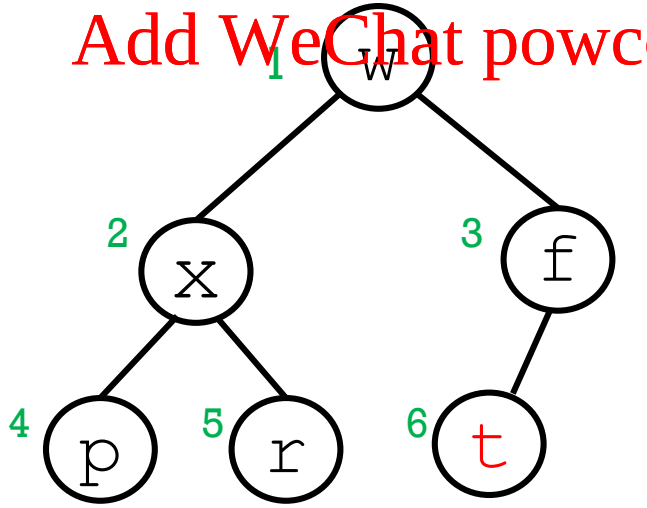


downHeap(3, 6)

EXAMPLE

1	2	3	4	5	6
w	x	f	p	r	t

k = 3



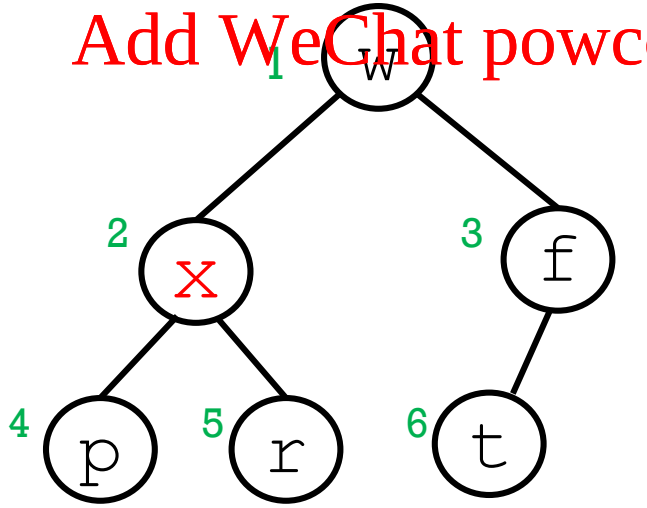
downHeap(3, 6)

EXAMPLE

1	2	3	4	5	6
w	x	f	p	r	t

k = 2

downHeap(2, 6)

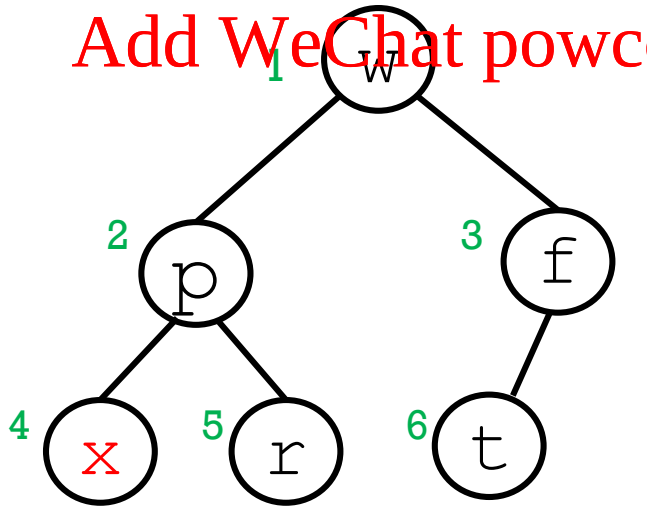


EXAMPLE

1	2	3	4	5	6
w	p	f	x	r	t

k = 2

downHeap(2, 6)

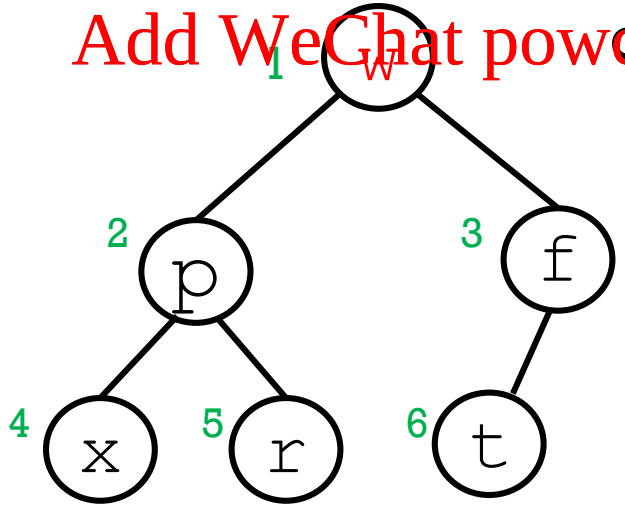


EXAMPLE

1	2	3	4	5	6
w	p	f	x	r	t

k = 1

downHeap(1, 6)

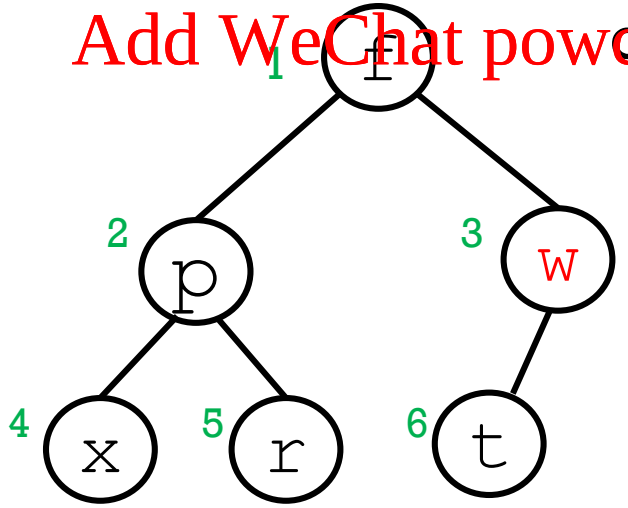


EXAMPLE

1	2	3	4	5	6
f	p	w	x	r	t


```
k = 1
```

```
downHeap( 1,6 )
```

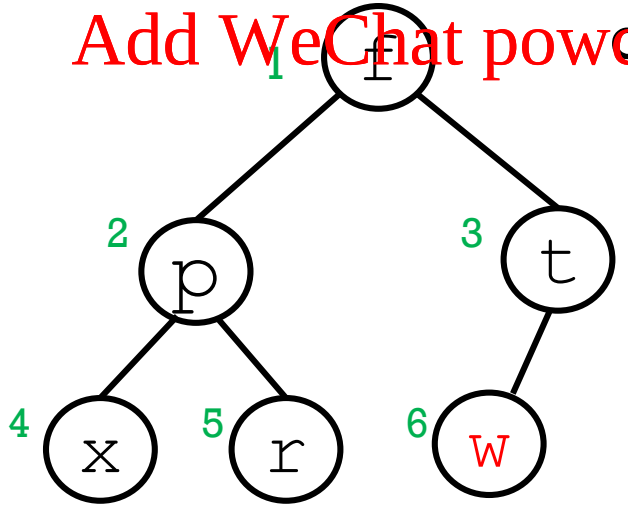


EXAMPLE

1	2	3	4	5	6
f	p	t	x	r	w


```
k = 1
```

```
downHeap( 1,6 )
```



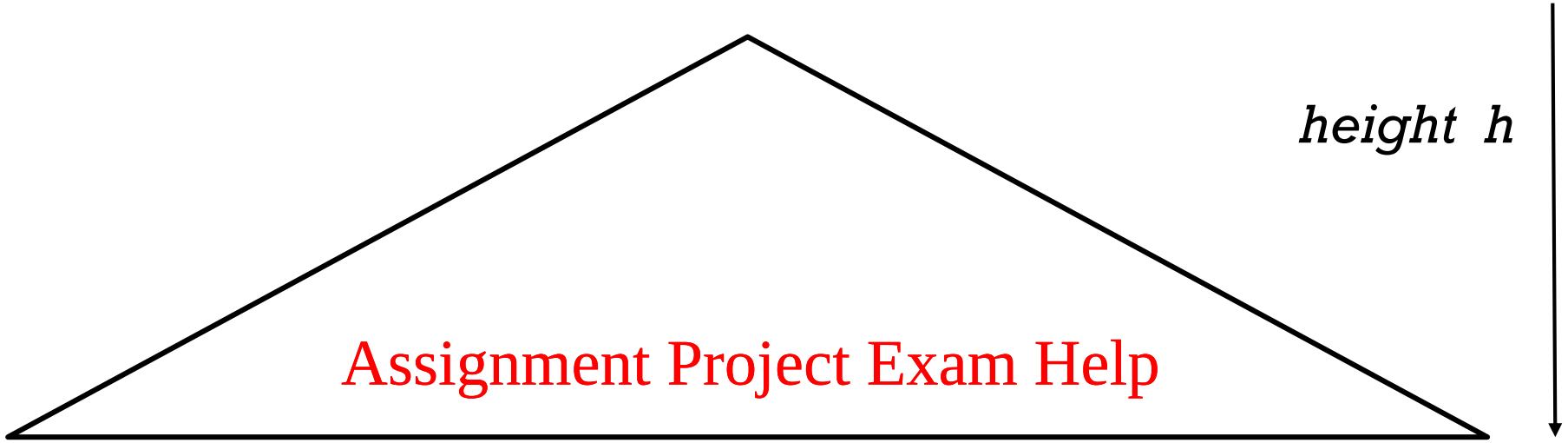
BUILDHEAPFAST() – IMPLEMENTATION

```
buildHeapFast(list){
    // copy elements from list to heap array
    for (k = size/2; k >= 1; k--)
        downHeap( k, size )
}
```


Claim: this algorithm is $O(n)$.

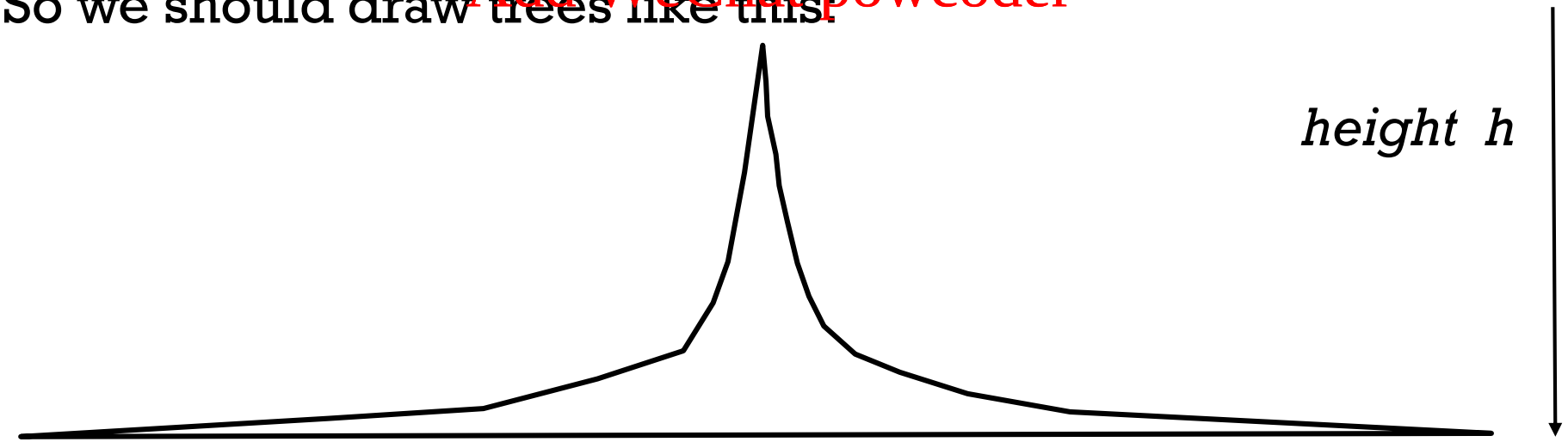
What is the intuition for why this algorithm is so fast?

We tends to draw binary trees like this:

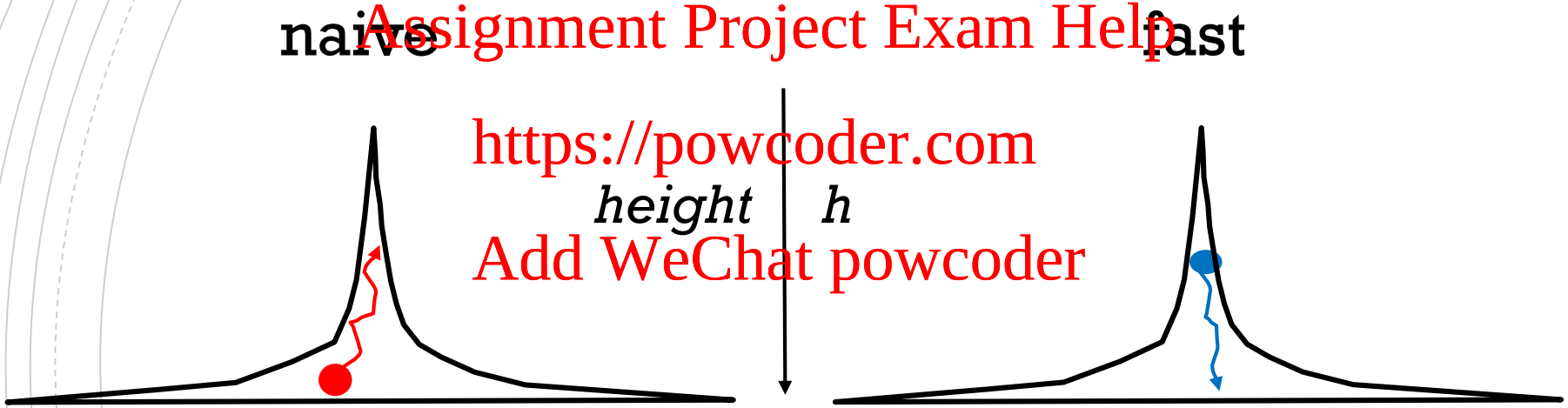


But the number of nodes doubles at each level.

So we should draw trees like this:



BUILDHEAP ALGORITHMS



Most nodes swap ~*h* times in worst case.

Few nodes swap ~h times in worst case.

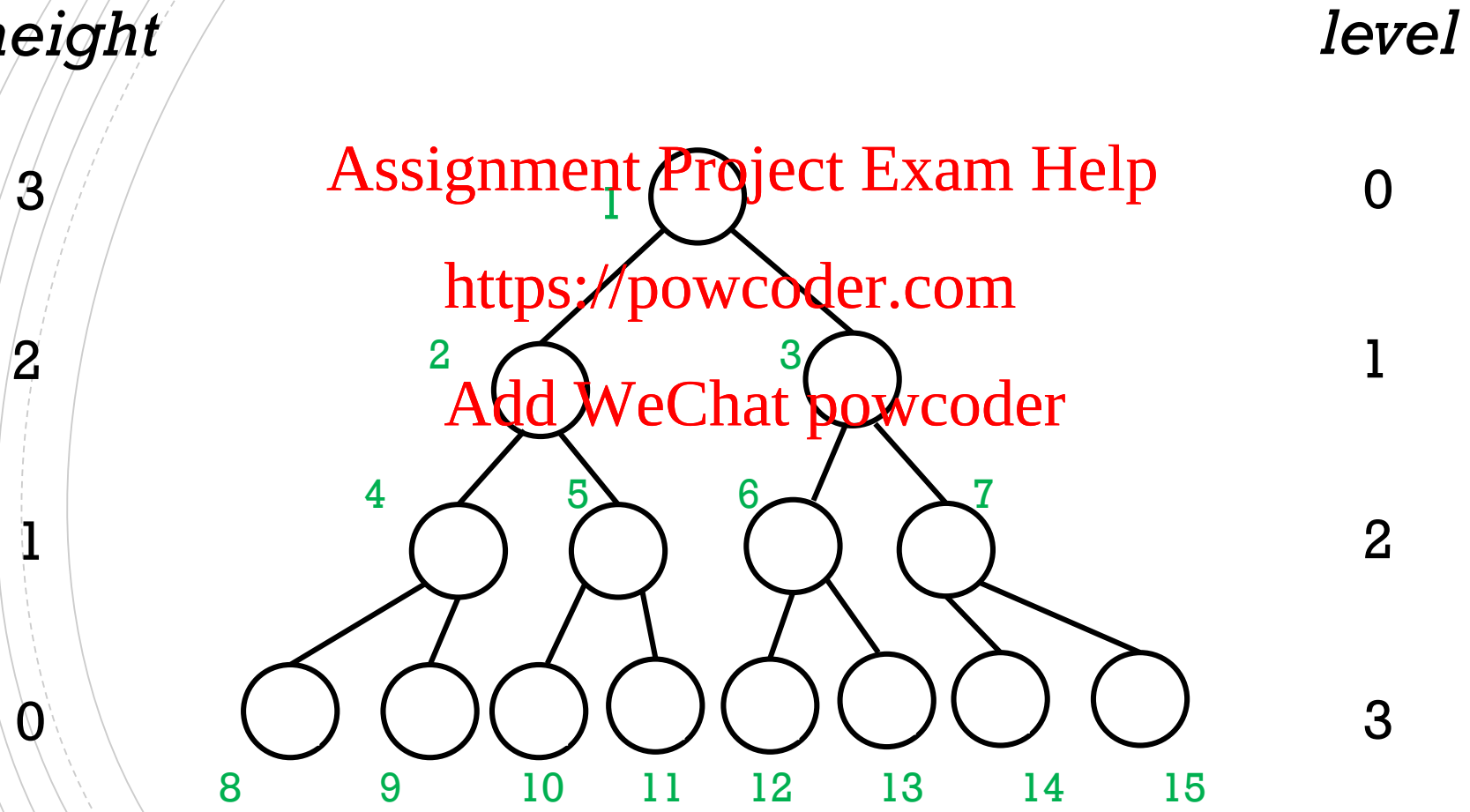
HOW TO SHOW BUILDHEAPFAST IS $O(n)$?

The worst case number of swaps needed to downHeap node i is the height of that node.

$$t(n) = \sum_{i=1}^{n} height\ of\ node\ i$$

½ of the nodes do no swaps.
¼ of the nodes do at most one swap.
1/8 of the nodes do at most two swaps….

ASSUME THE LAST LEVEL IS FULL



WORSE CASE OF BUILDHEAPAPFAST ?

- How many elements at *level* l ? $(l \in 0, \ldots, h)$
- What is the height of each *level* l node?

WORSE CASE OF BUILDHEAPFAST ?

- How many elements at $level\ \ l$? $(l \in 0, \ldots, h)$
  - 2^l
- What is the height of each $level\ l$ node?
  - $h - l$

$$t(n) = \sum_{i=1}^{n} height\ of\ node\ i$$

$$= \ ?$$

WORSE CASE OF BUILDHEAPFAST ?

- How many elements at $level\ l$? $(l \in 0, \ldots, h)$
  - 2^l
- What is the height of each $level\ l$ node?
  - $h - l$

$$t(n) = \sum_{i=1}^{n} height\ of\ node\ i$$

$$= \sum_{l=0}^{h} (h - l)\ 2^l$$

$$t_{worstcase}(h) = \sum_{l=0}^{h} (h-l)\ 2^l$$

$$= h\sum_{l=0}^{h} 2^l - \sum_{l=0}^{h} l\ 2^l$$

Easy

Difficult

(number of nodes)

(sum of node levels)

$$t_{worstcase}(h) = \sum_{l=0}^{h} (h-l)\ 2^l$$

$$= h \sum_{l=0}^{h} 2^l - \sum_{l=0}^{h} l\ 2^l$$

(See next slide)

$$= h(2^{h+1}-1) - (h-1)2^{h+1} - 2$$

$$
\begin{aligned}
\sum_{l=0}^{h} l\ 2^l &= \sum_{l=0}^{h} l\ (2^{l+1} - 2^l) \qquad \text{(trick)} \\
&= \sum_{l=0}^{h} l\ 2^{l+1} - \sum_{l=0}^{h} l\ 2^l \\
&= \sum_{l=0}^{h} l\ 2^{l+1} - \sum_{l=0}^{h-1} (l+1)\ 2^{l+1} \\
&= h\ 2^{h+1} + \sum_{l=0}^{h-1} (l - (l+1))\ 2^{l+1} \\
&= h\ 2^{h+1} - 2\sum_{l=0}^{h-1} 2^l \\
&= h\ 2^{h+1} - 2(2^h - 1) \\
&= (h-1)2^{h+1} + 2
\end{aligned}
$$

Second term index goes to h-1 only

$$t_{worstcase}(h) = \sum_{l=0}^{h} (h-l)\ 2^l$$

$$= h\sum_{l=0}^{h} 2^l - \sum_{l=0}^{h} l\ 2^l$$

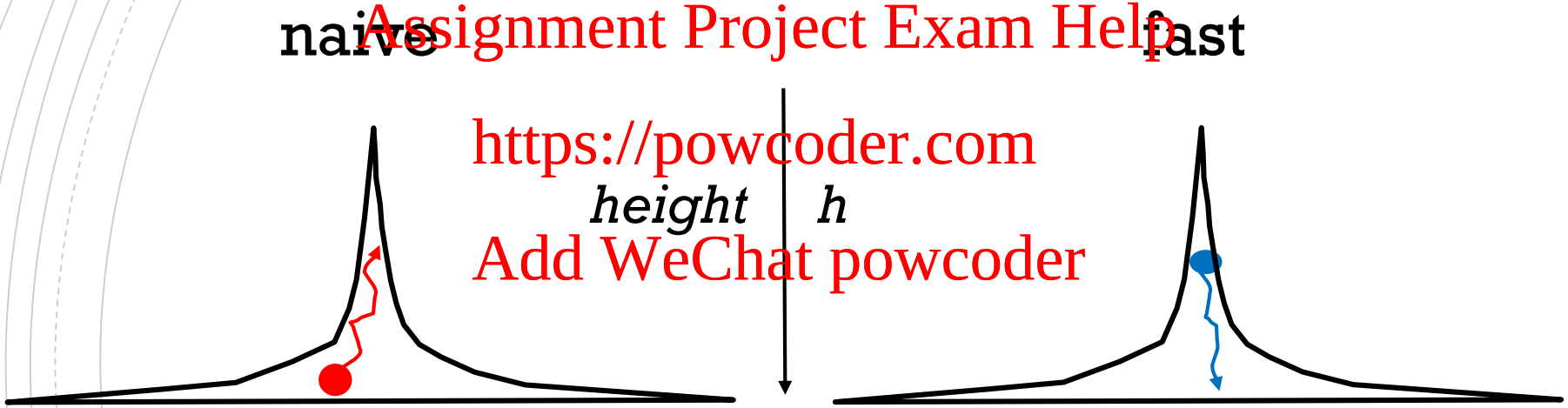
$$= h(2^{h+1}-1) - (h-1)2^{h+1} - 2 \quad \text{from above}$$

$$= 2^{h+1} - h - 2$$

Since $n = 2^{h+1} - 1$, we get :

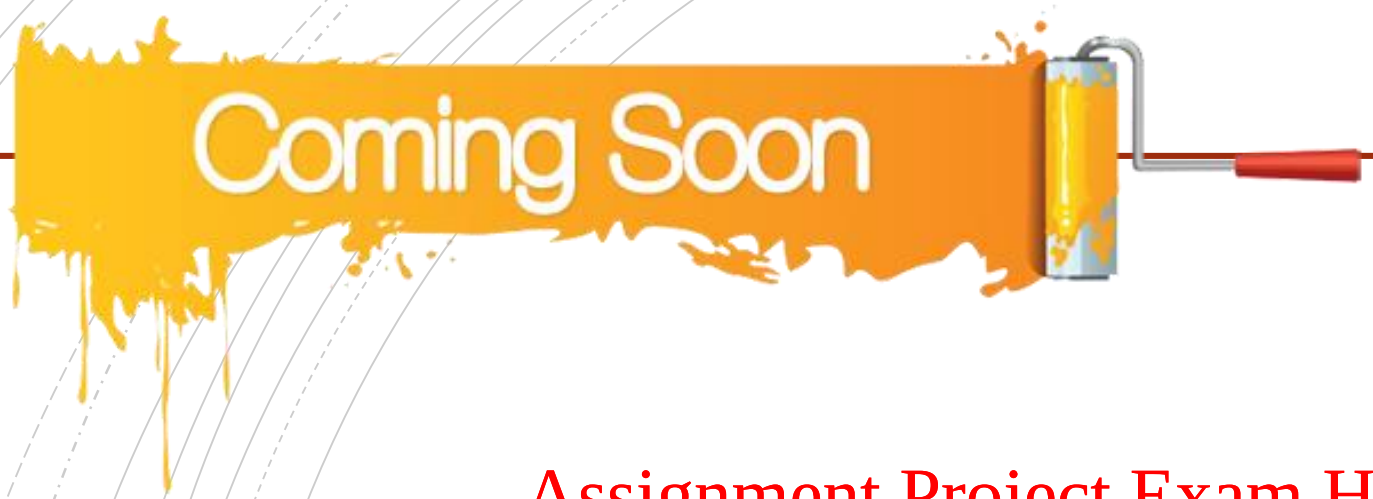
$$t_{worstcase}(n) = n - \log(n+1)$$

SUMMARY: BUILDHEAP ALGORITHMS



$O(n \ log_2 \ n)$

$O(n)$



In the next videos:

- Hashing
- Graphs