COMP 250

Lecture 25

heaps 3

Nov. 6, 2017

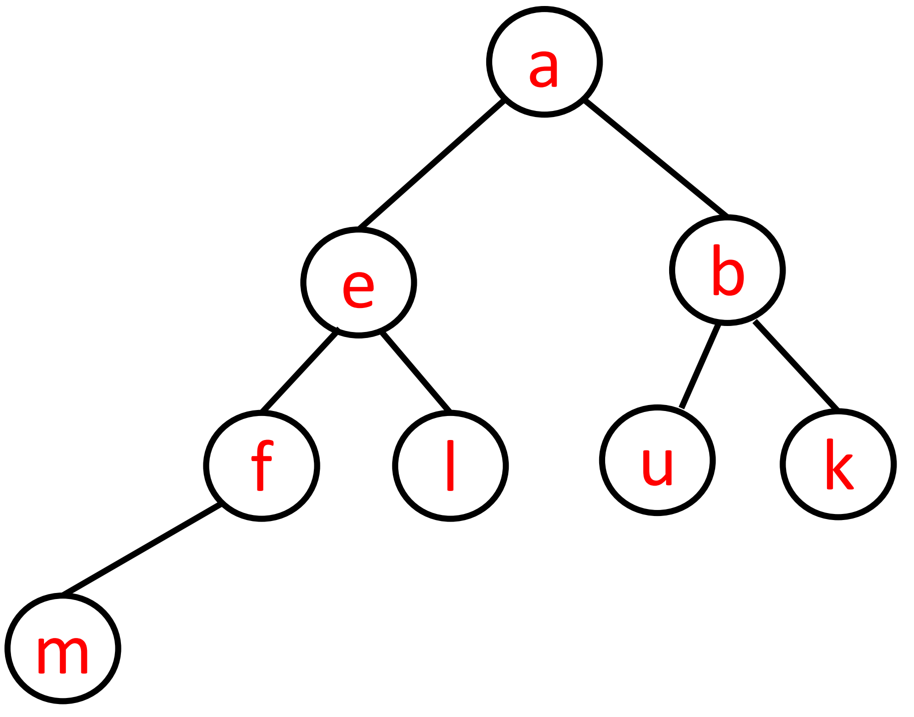
STEM Support

https://infomcgillstem.wixsite.com/stemsupportmcgill

MSSG = McGill Space systems group

http://www.mcgillspace.com/#!/

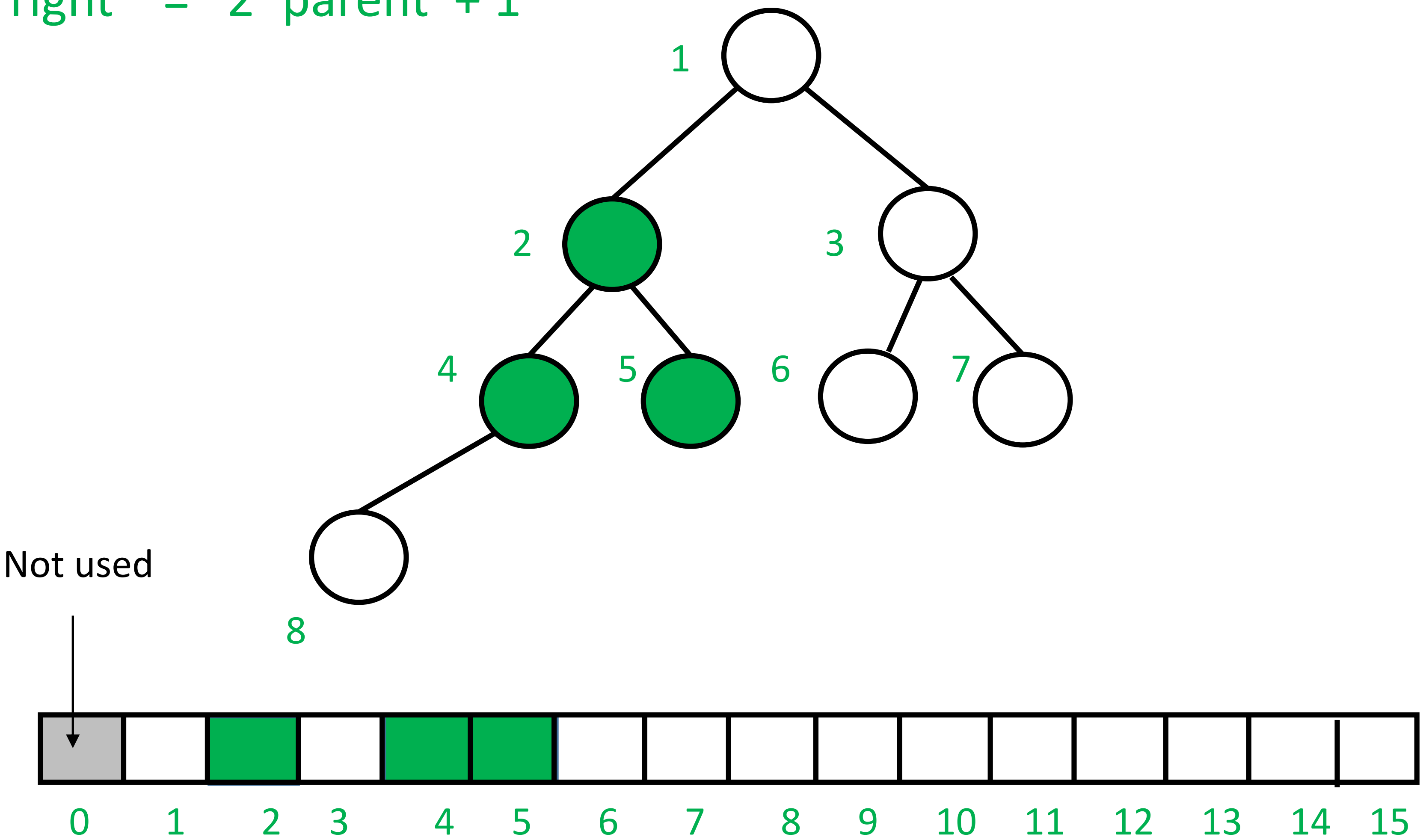
RECALL: min Heap (definition)



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

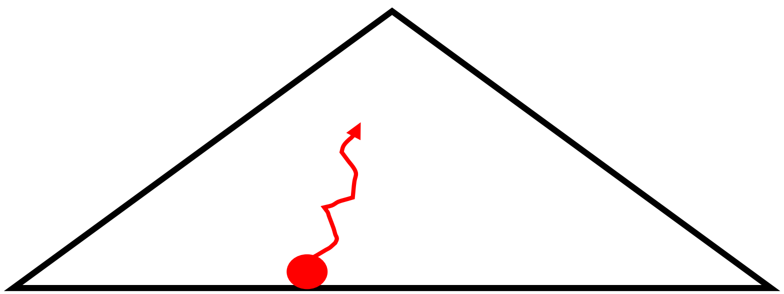
Heap index relations

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



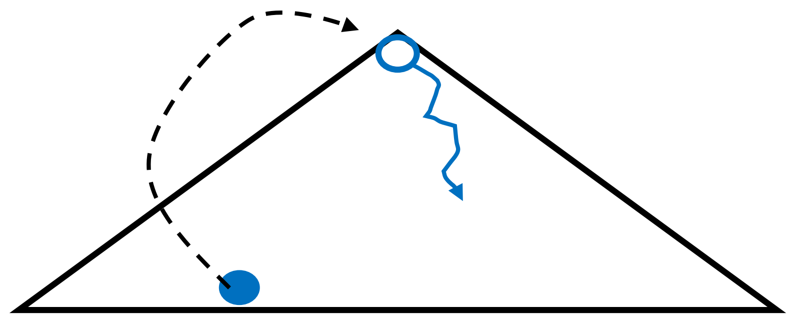
buildHeap()

add()



upHeap(element)

removeMin()



downHeap(1, size)

How to build a heap ? (slight variation)

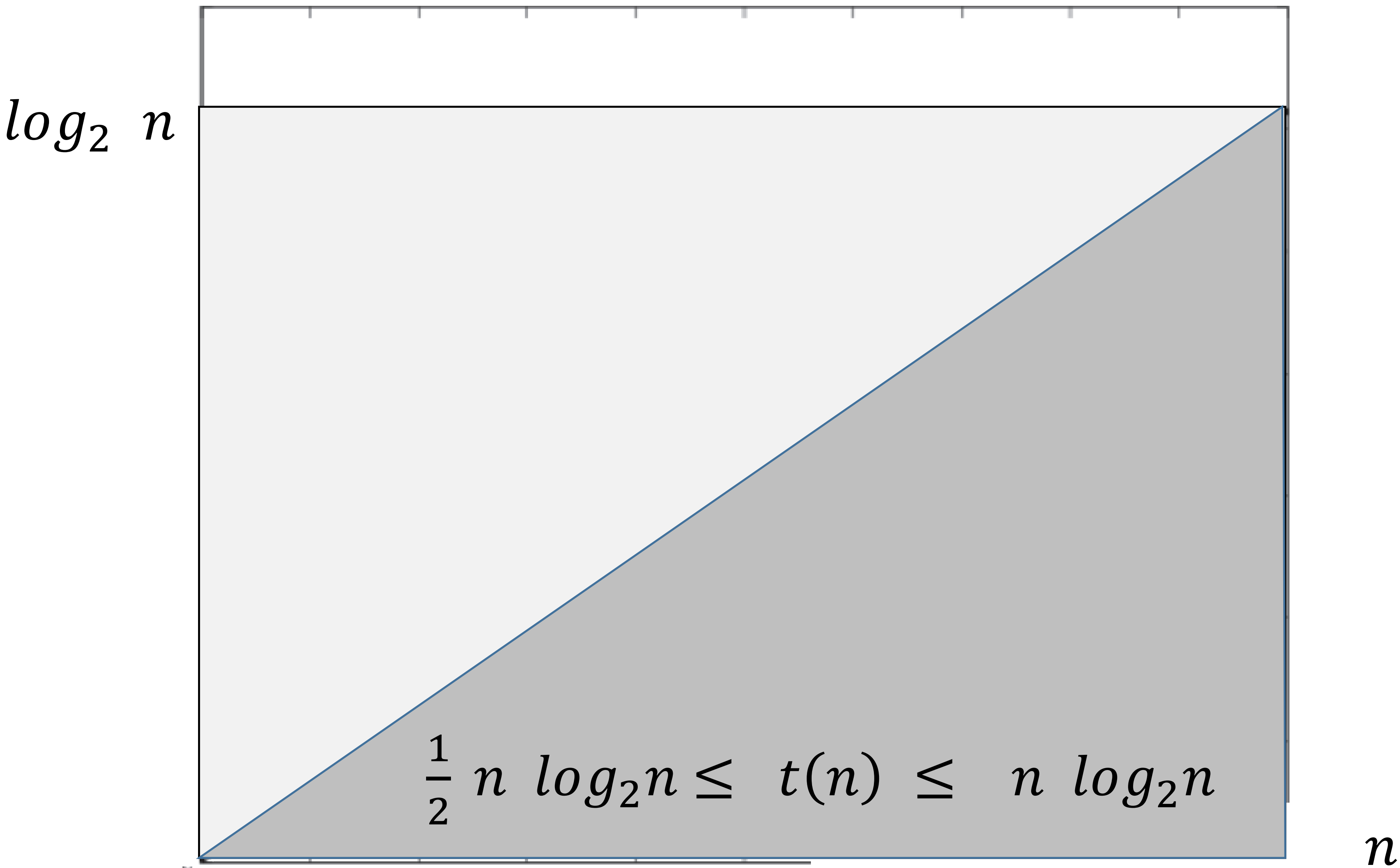
```
buildHeap(){
   //  assume that an array already contains size elements
   for (k = 2;  k <=  size;  k++)
        upHeap( k ) }
}
```

How to build a heap ? (slight variation)

```
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
      }
}
```

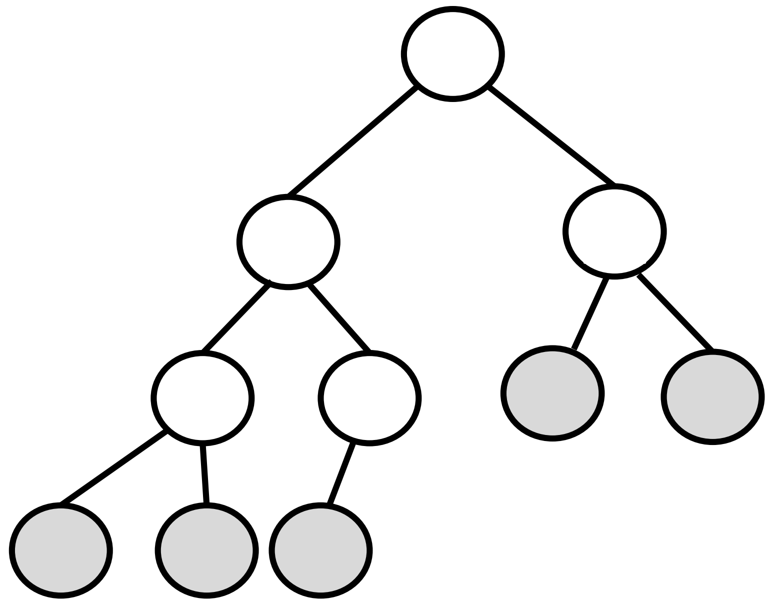
Recall last lecture: Worse case of buildHeap



Thus, worst case: buildHeap is $\Theta(n\, log_2\ n)$

Next, I will show you a $\Theta(n)$ algorithm for building a heap.

How to build a heap ? (**fast**)



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)

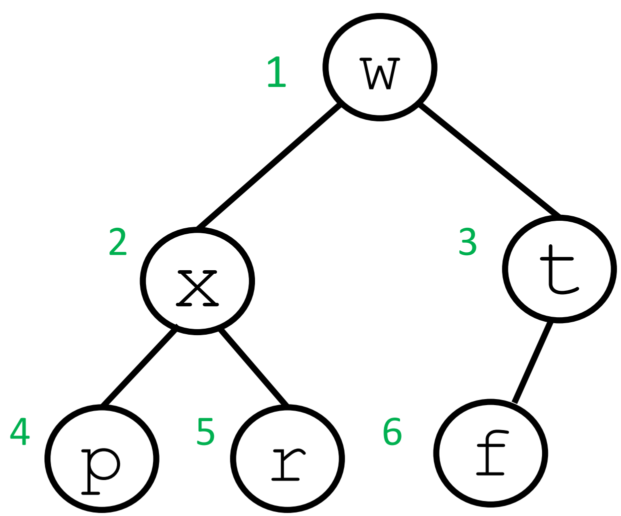
```
buildHeapFast(){

 //  assume that heap[ ] array contains size elements

    for (k = size/2;   k >= 1;    k--)
        downHeap( k, size )
}
```

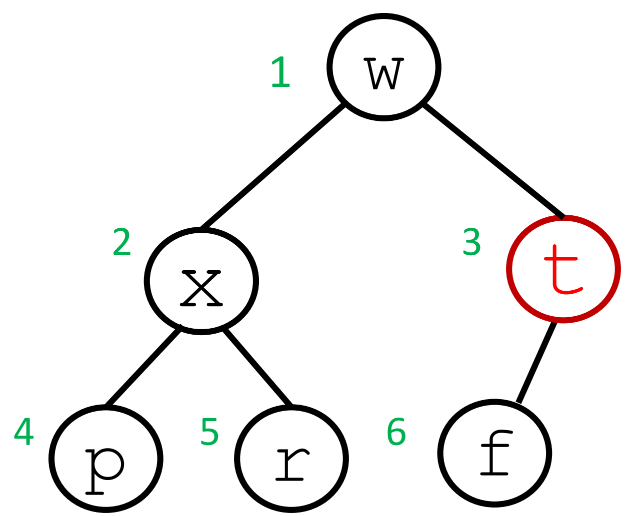
1	2	3	4	5	6
w	x	t	p	r	f

k = 3



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

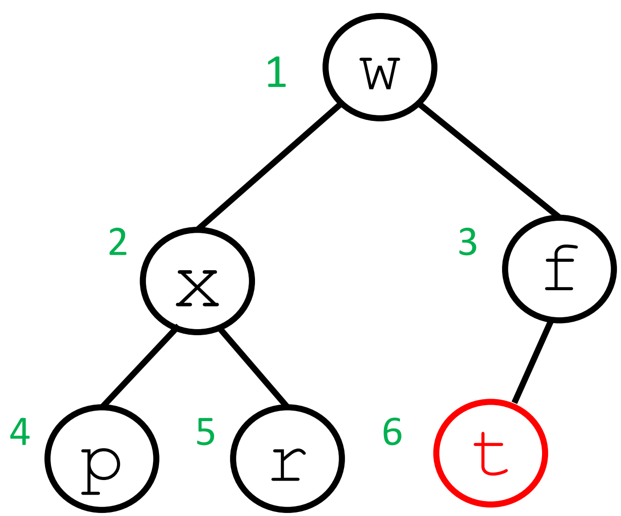
k = 3



downHeap(3, 6)

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

k = 3

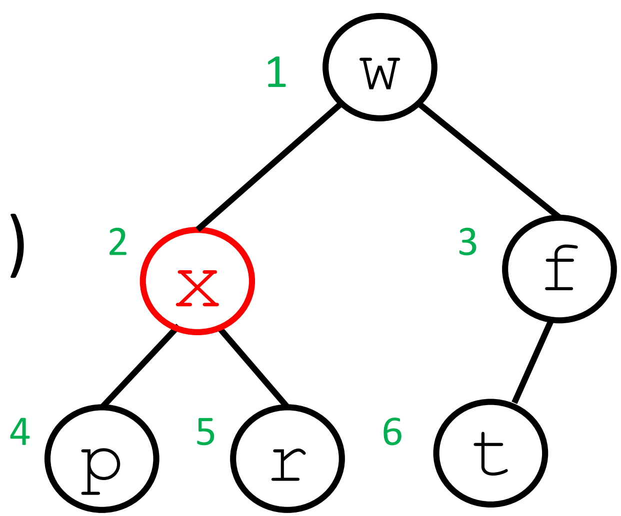


downHeap(3, 6)

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

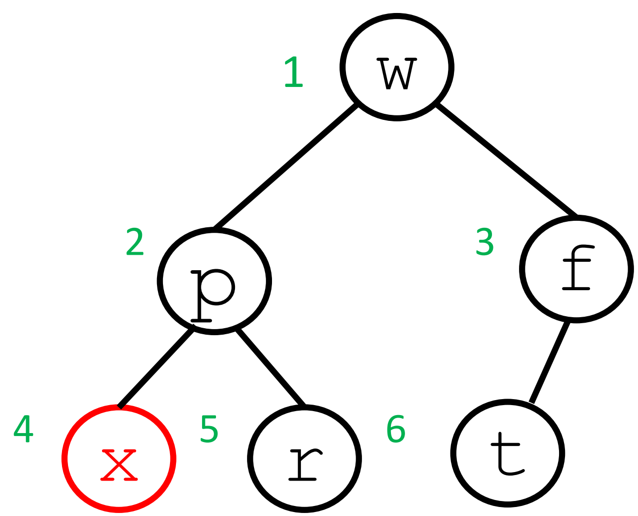
k = 2

downHeap(2, 6)



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

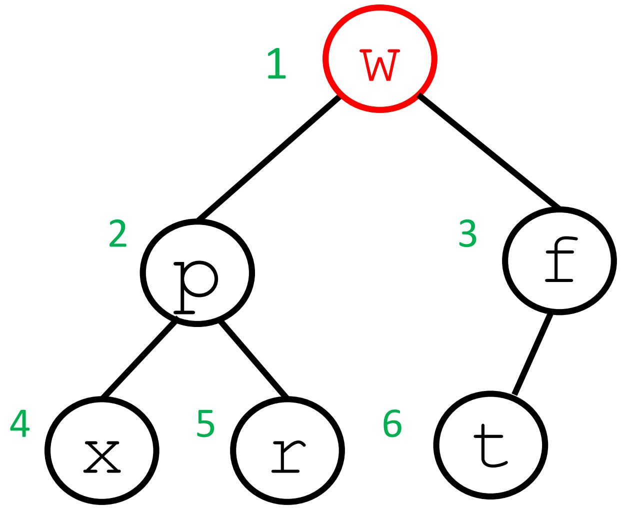
k = 2



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

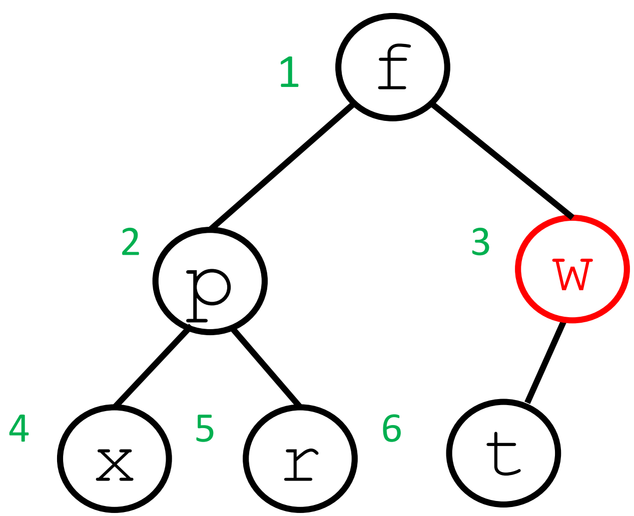
k = 1

downHeap(1, 6)



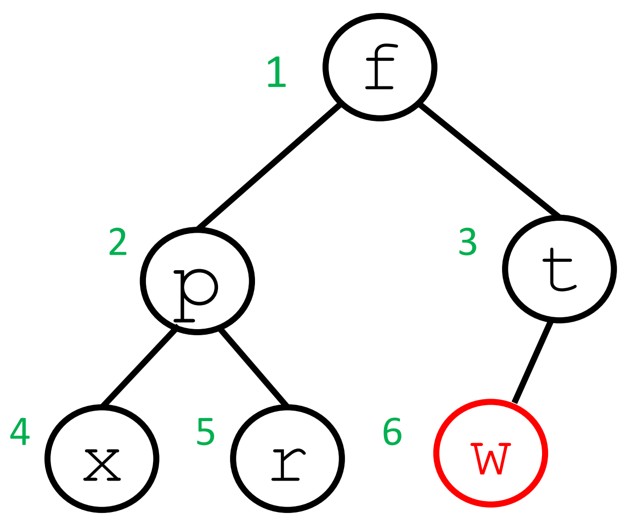
1	2	3	4	5	6
f	p	w	x	r	t

k = 1



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

k = 1

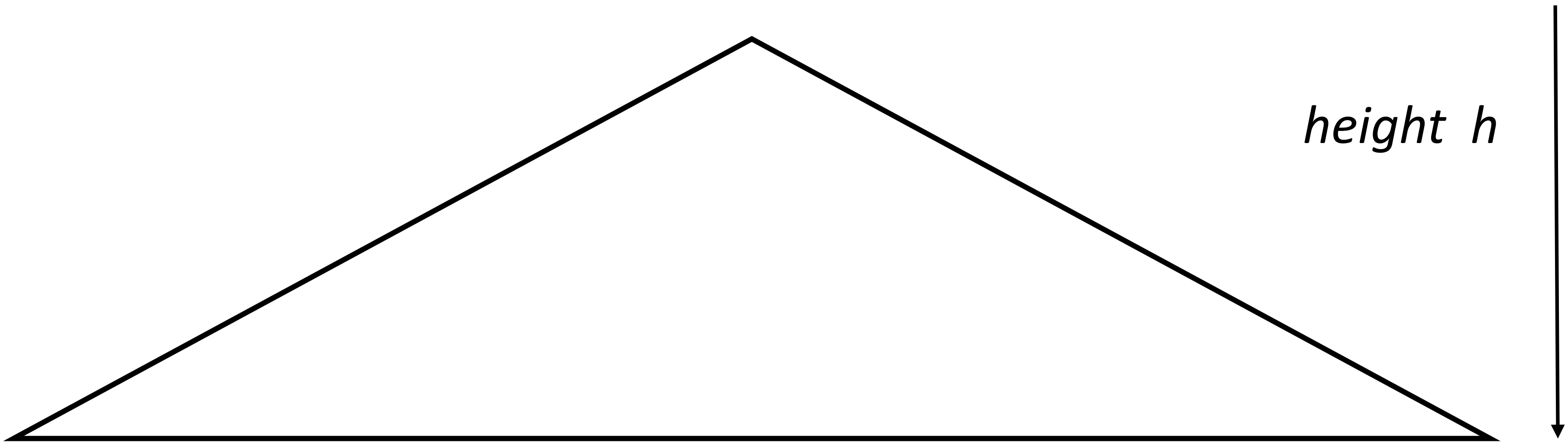


```
buildHeapFast(list){
    copy list into a heap array
    for (k = size/2;   k >= 1;    k--)
        downHeap( k, size )
}
```

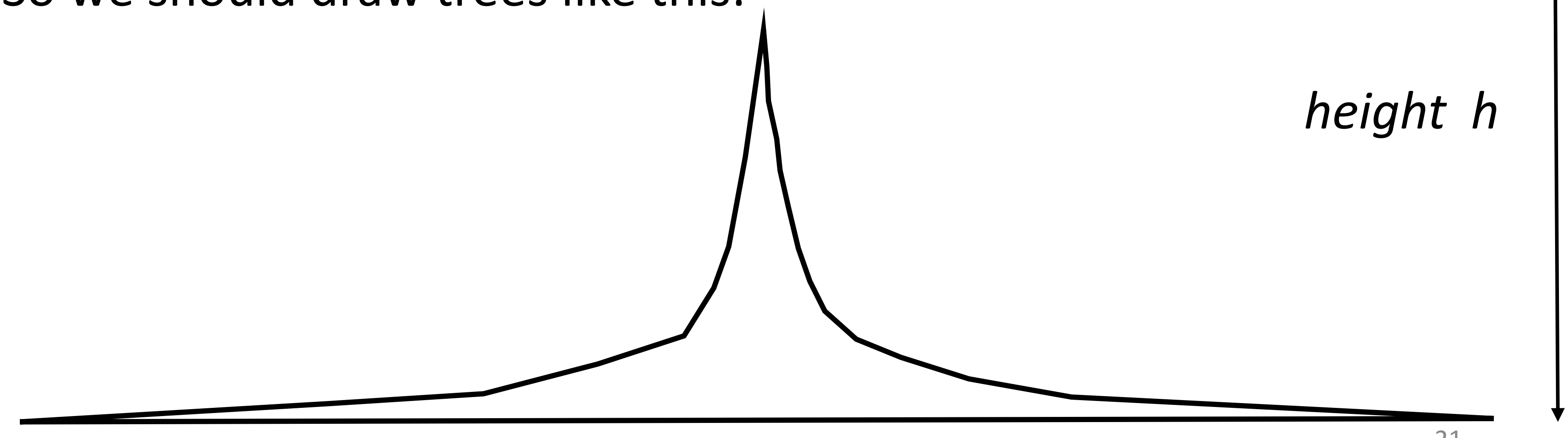
Claim: this algorithm is $\Theta(n)$.

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

We tends to draw binary trees like this:



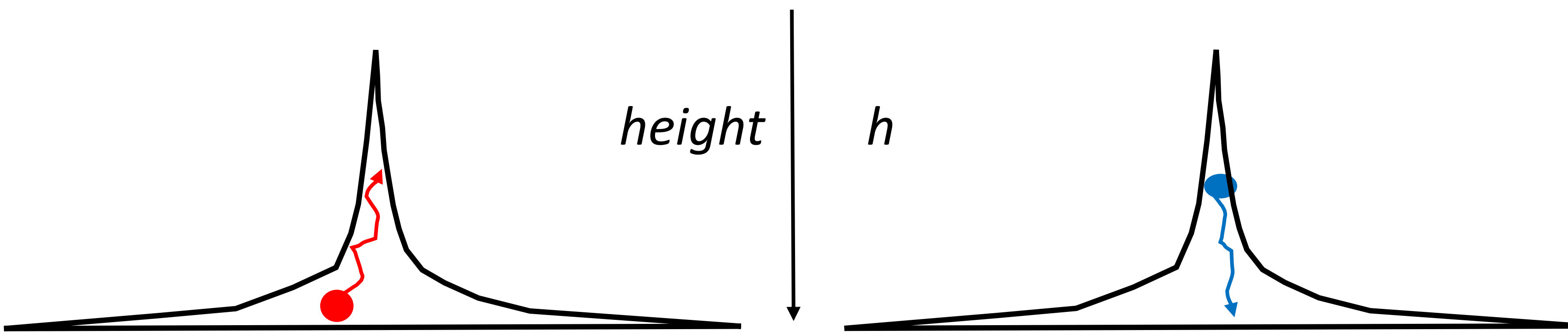
But the number of nodes doubles at each level.
So we should draw trees like this:



buildheap algorithms

last lecture

today



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

Few nodes swap ~h times in worst case.

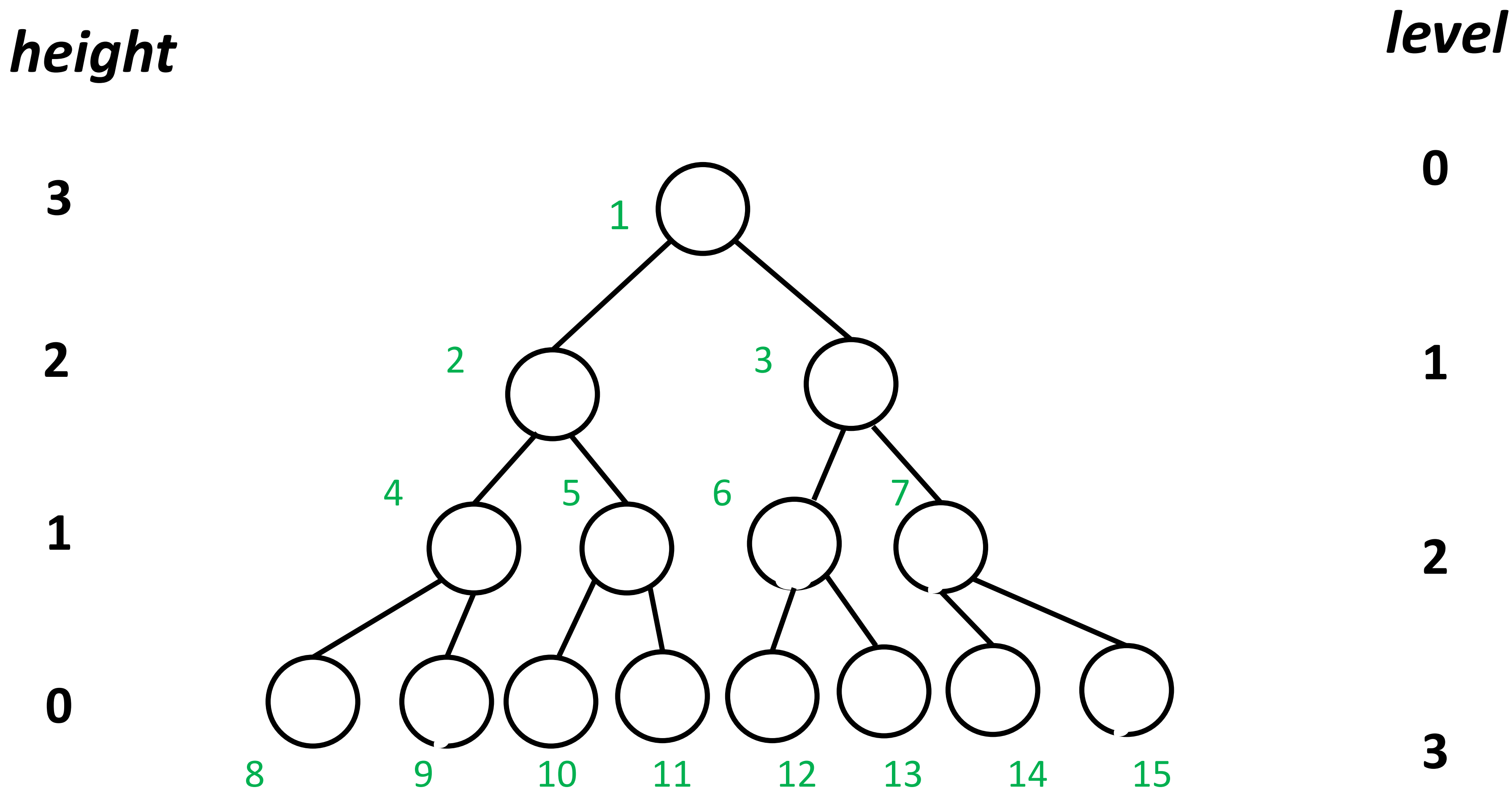
How to show buildHeapFast is $\Theta(n)$?

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

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

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....

Let’s do the calculation for a tree that whose last level is full.



Worse case of buildHeap**Fast** ?

How many elements at $level\ \ l$? $(l \ \in\ 0, \ldots, \ h)$

What is the height of each $level\ \ l$ node?

Worse case of buildHeap**Fast** ?

$level\ \ l$ has 2^l elements, $l \in 0, \ldots, h$

$level\ \ l$ nodes have height $h - l$.

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

$= \quad ?$

Worse case of buildHeap**Fast** ?

$level\ \ l$ has 2^l elements, $l \in 0, \ldots, h$

$level\ \ l$ nodes have height $h - l$.

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

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

$$
\begin{aligned}
t_{worstcase}(h) &= \sum_{l=0}^{h} (h-l)\, 2^l \\
&= h \sum_{l=0}^{h} 2^l - \sum_{l=0}^{h} l\, 2^l
\end{aligned}
$$

Easy

(number of nodes)

Difficult

(sum of node depths)

$$
\begin{aligned}
t_{worstcase}(h) &= \sum_{l=0}^{h} (h-l)\, 2^l \\
&= h\sum_{l=0}^{h} 2^l - \boxed{\sum_{l=0}^{h} l\, 2^l} \\
&= h(2^{h+1}-1) - \boxed{(h-1)2^{h+1} - 2}
\end{aligned}
$$

(See next slide)

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

Second term index goes to h-1 only

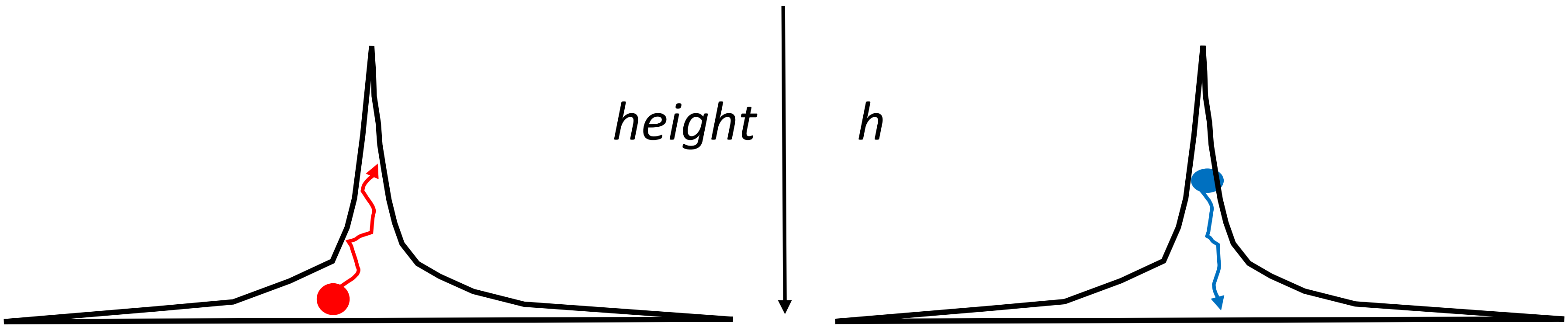
$$
\begin{aligned}
t_{worstcase}(h) &= \sum_{l=0}^{h} (h-l)\, 2^l \\
&= h \sum_{l=0}^{h} 2^l - \sum_{l=0}^{h} l\, 2^l \\
&= h(2^{h+1}-1) - (h-1)2^{h+1} - 2 \qquad \text{from above} \\
&= 2^{h+1} - h - 2
\end{aligned}
$$

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

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

Summary: buildheap algorithms

last lecture | today



$O(n\ \ log_2\ n)$ | $O(n)$

Heapsort

Given a list with size elements:

Build a heap.

Repeatedly call removeMin() and put the removed elements into a list.

“in place” Heapsort

Given an array heap[ ] with size elements:

```
heapsort(){
   buildheap( )
   for i = 1 to size{
      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

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	

```
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
b   d   w   e   l   u   k   f | a
```

```
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
b   d   w   e   l   u   k   f  | a
b   d   k   e   l   u   w   f  | a
```

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	

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	

```
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
d    f    k    e    l    u    w  | b    a
```

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
d	f	k	e	l	u	w |	b	a
d	e	k	f	l	u	w |	b	a

```
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
```

```
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
```

```
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
```

Heapsort

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