How much space does this routine use in the worst case for a given n?

```
public static void use_space(int n)
{
    int b;
    int [] A;

      if (n<=1) return;

      use_space(n/2);
      use_space(n/2);

}
```

Announcements:

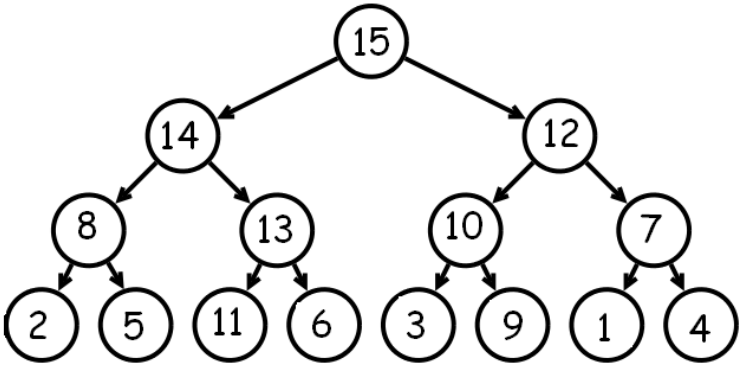
Preliminary final exam schedule has been posted: CSC 225 Sat. Dec. 7, 7pm

Assign. 1B Resubmission: Fri. Oct. 11, 11:55pm.
Assign. 2B: Tues. Oct. 15, 11:55pm.

Midterm: in class, **Wednesday Oct. 23, 2013**.

Lots of old midterms are available from the class web pages. The more recent ones are more relevant.

Heapsort



A Compost Heap

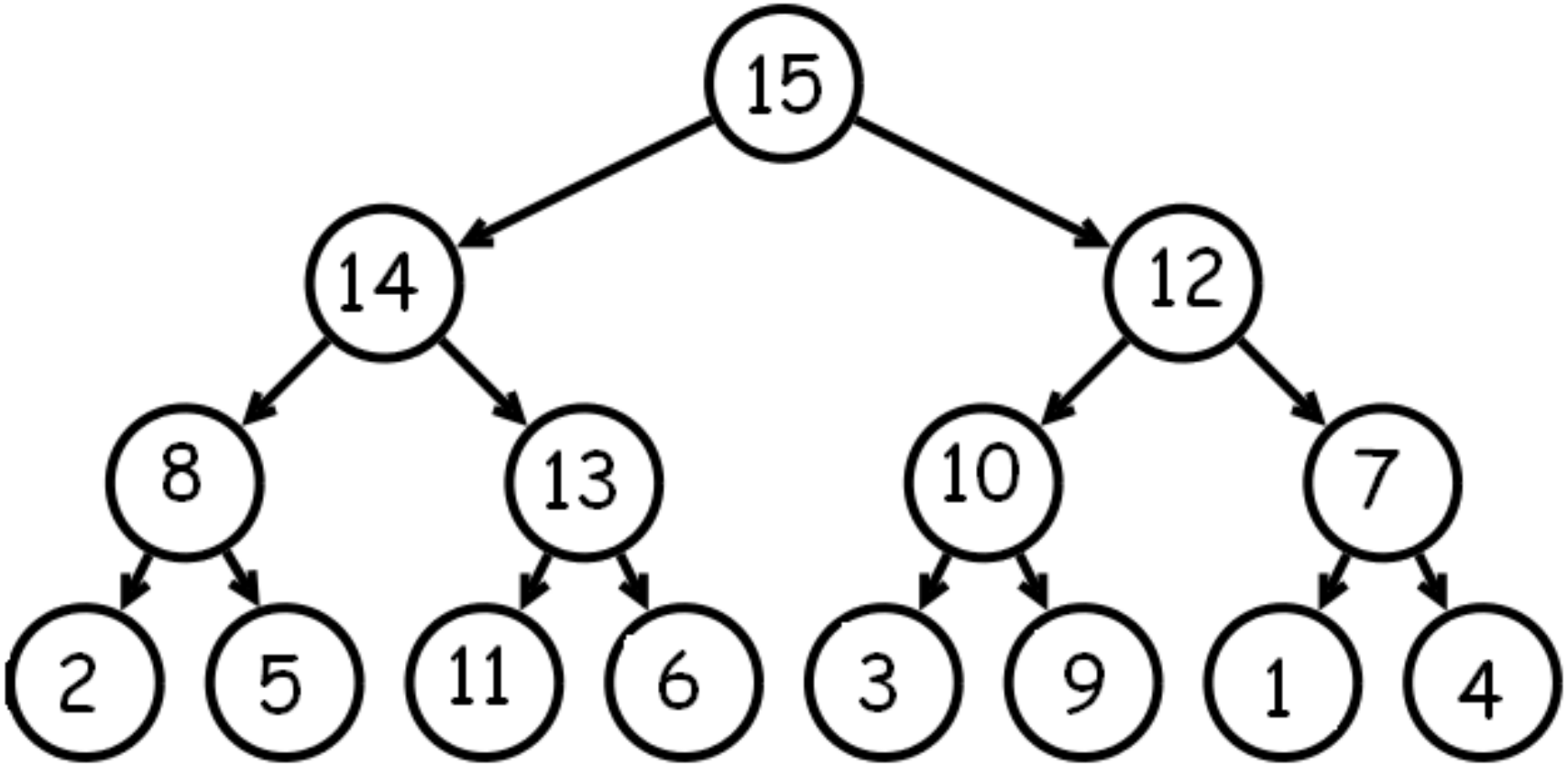
Madame Trash Heap

Pictures from:

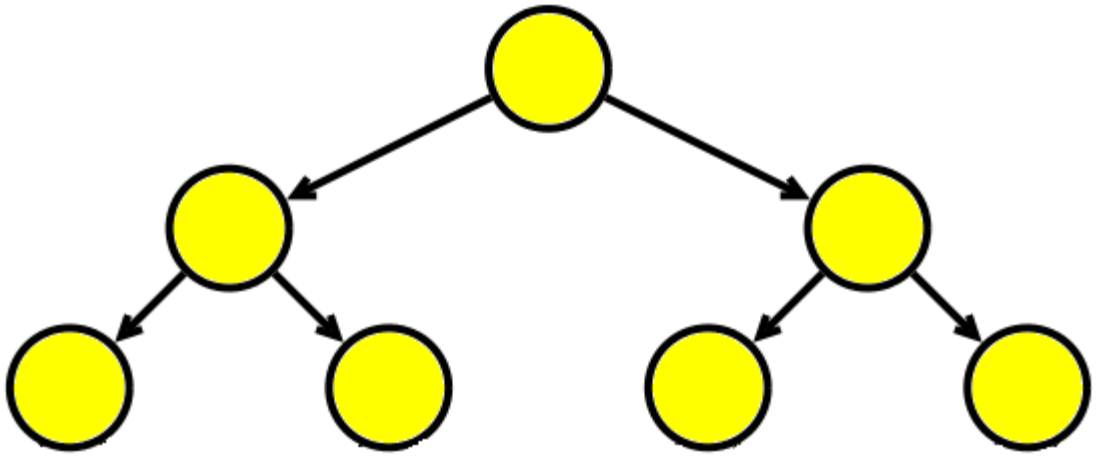
http://www.compostinfo.com/tutorial/methods.htm

http://linguiniontheceiling.blogspot.com/

Max-heap: The data value at each node is greater than or equal to the data values of its children.

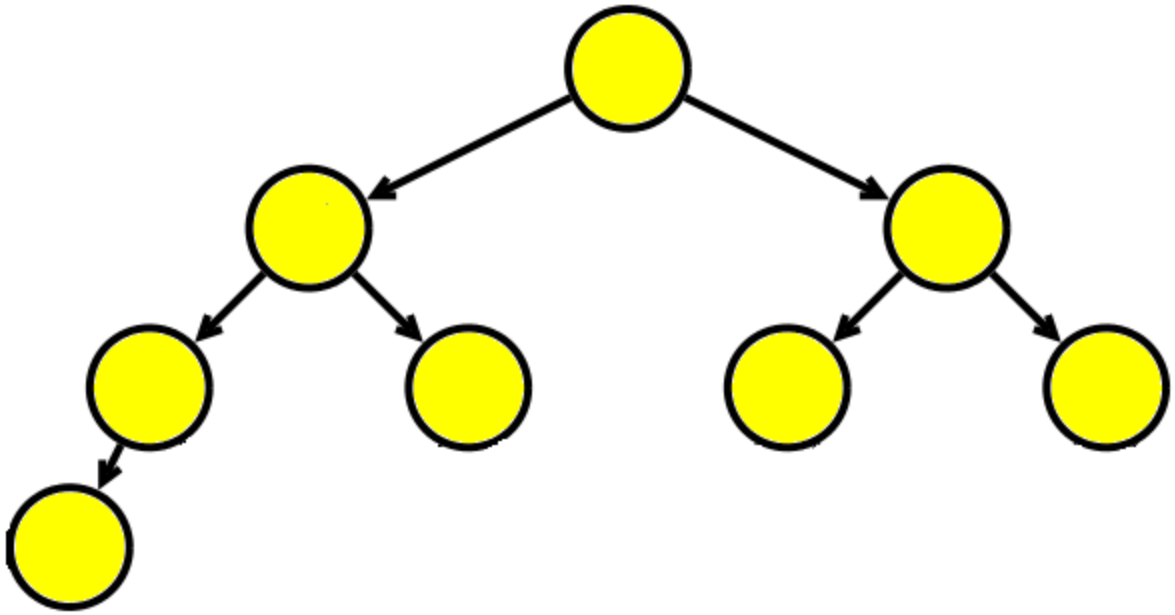


Left complete binary tree- fill in last level of a complete binary tree from left to right.

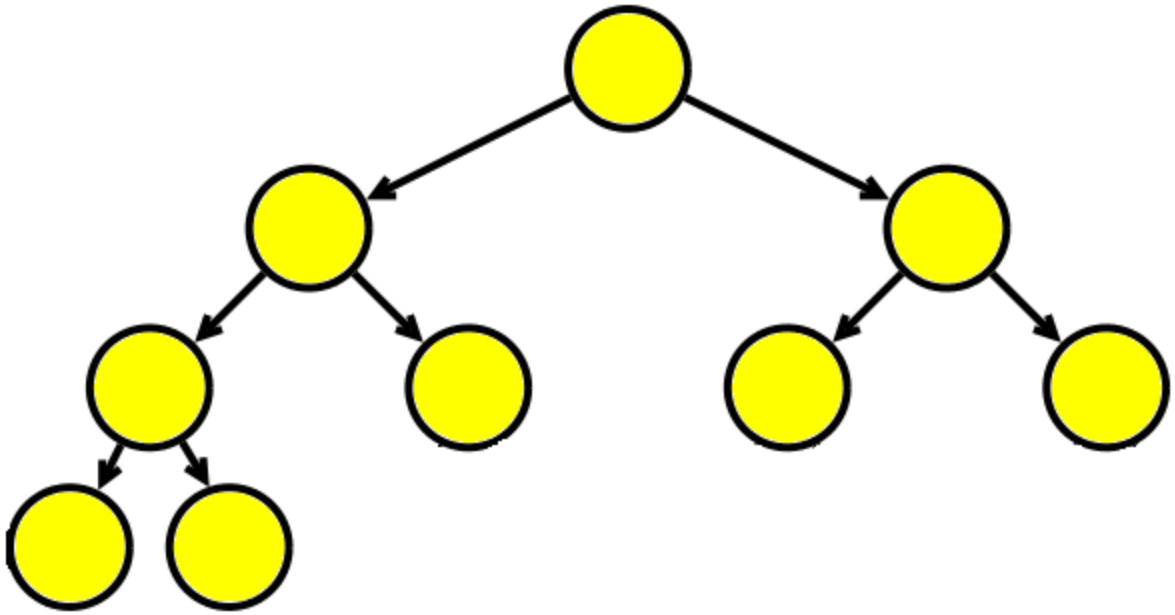


A heap is always stored in a tree with the shape of a left-complete binary tree.

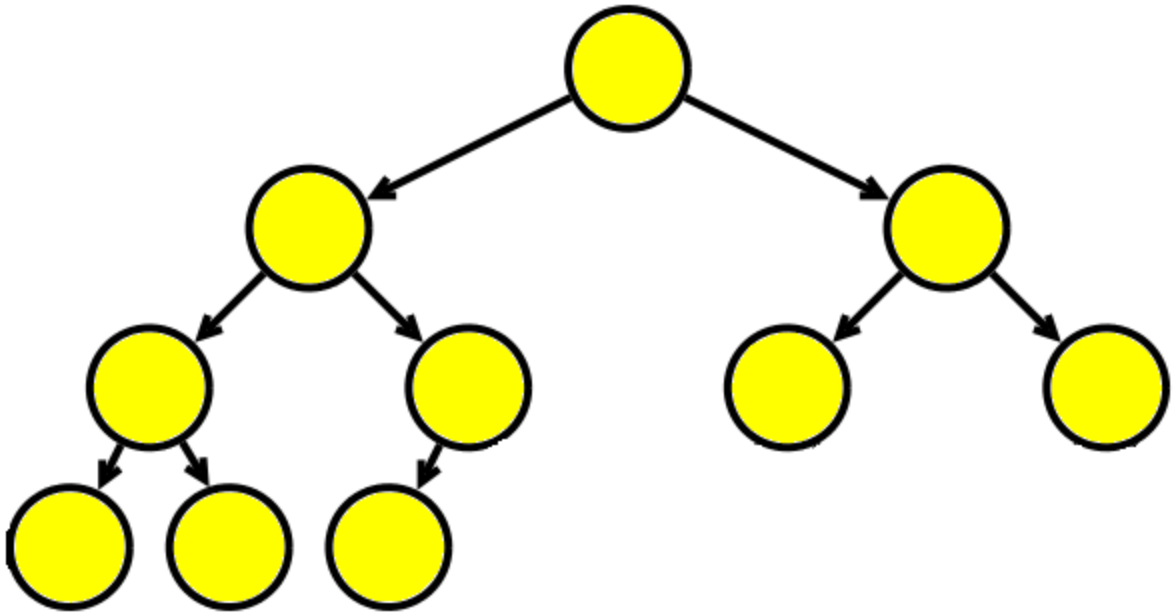
Left complete binary tree- fill in last level of a complete binary tree from left to right.



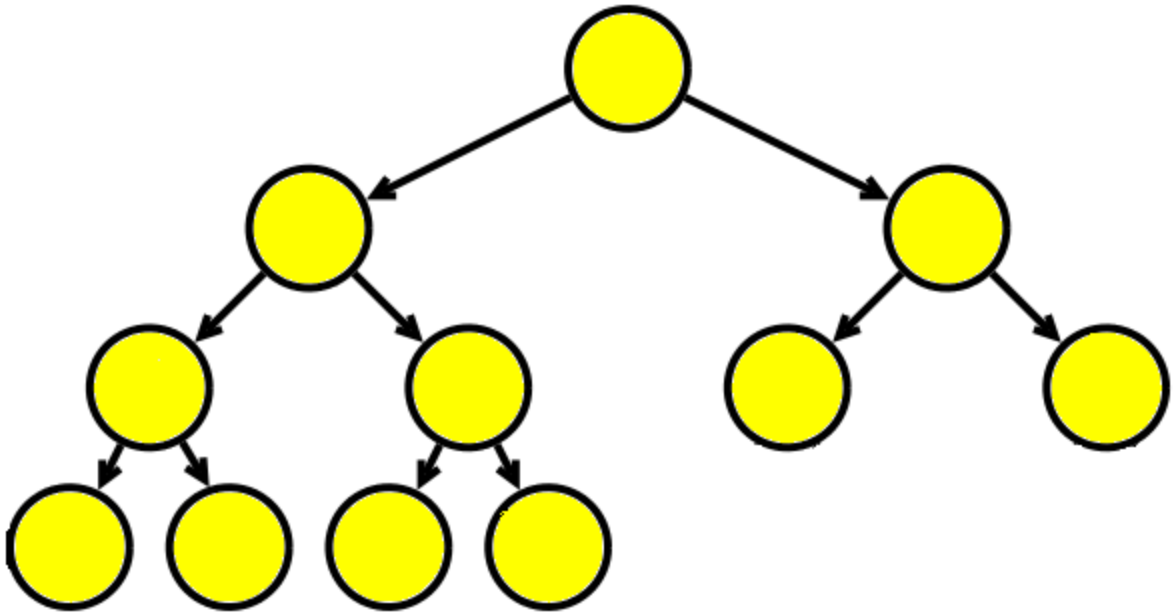
Left complete binary tree- fill in last level of a complete binary tree from left to right.



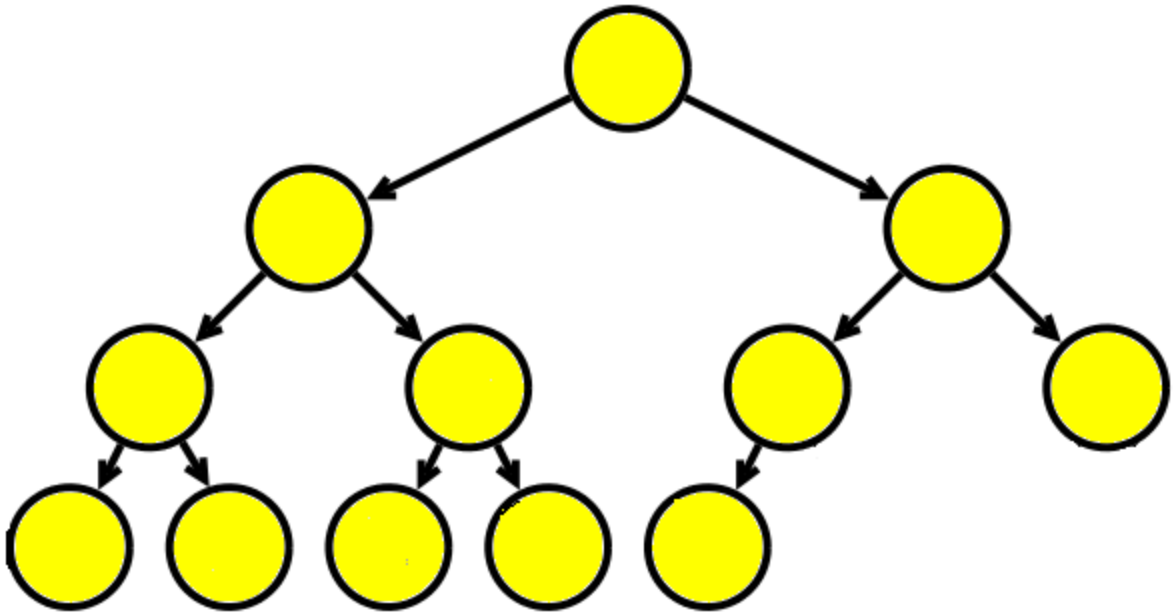
Left complete binary tree- fill in last level of a complete binary tree from left to right.



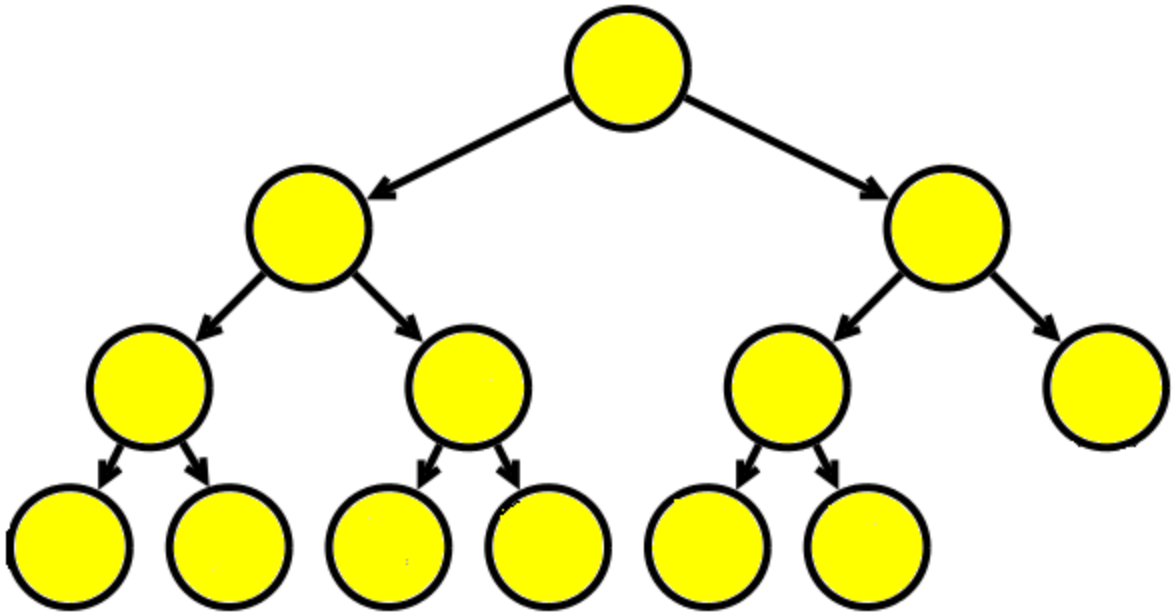
Left complete binary tree- fill in last level of a complete binary tree from left to right.



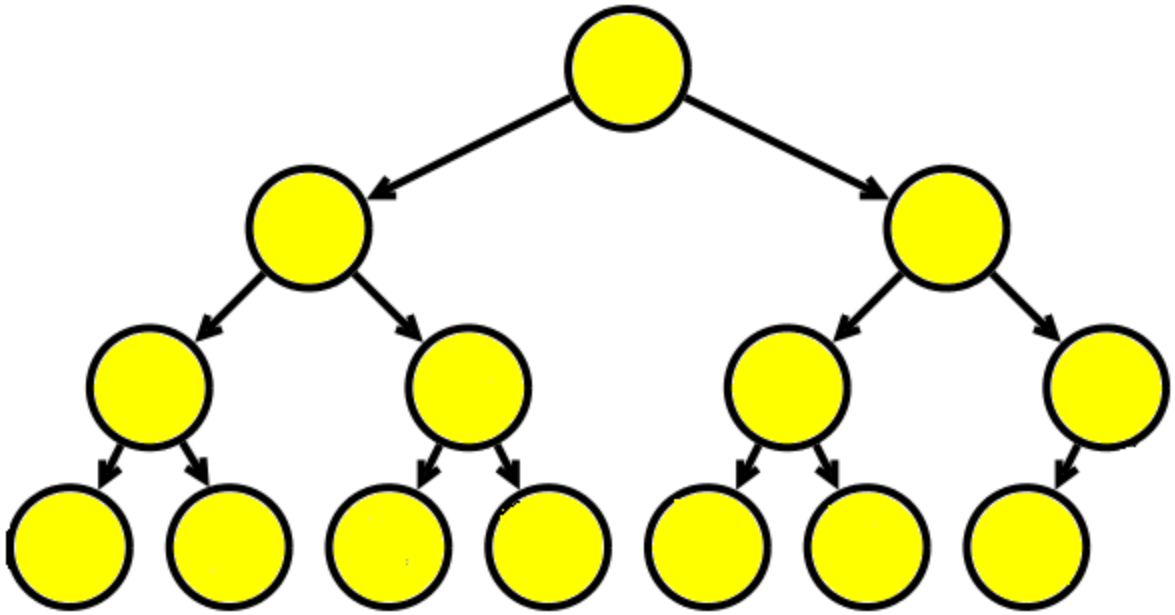
Left complete binary tree- fill in last level of a complete binary tree from left to right.



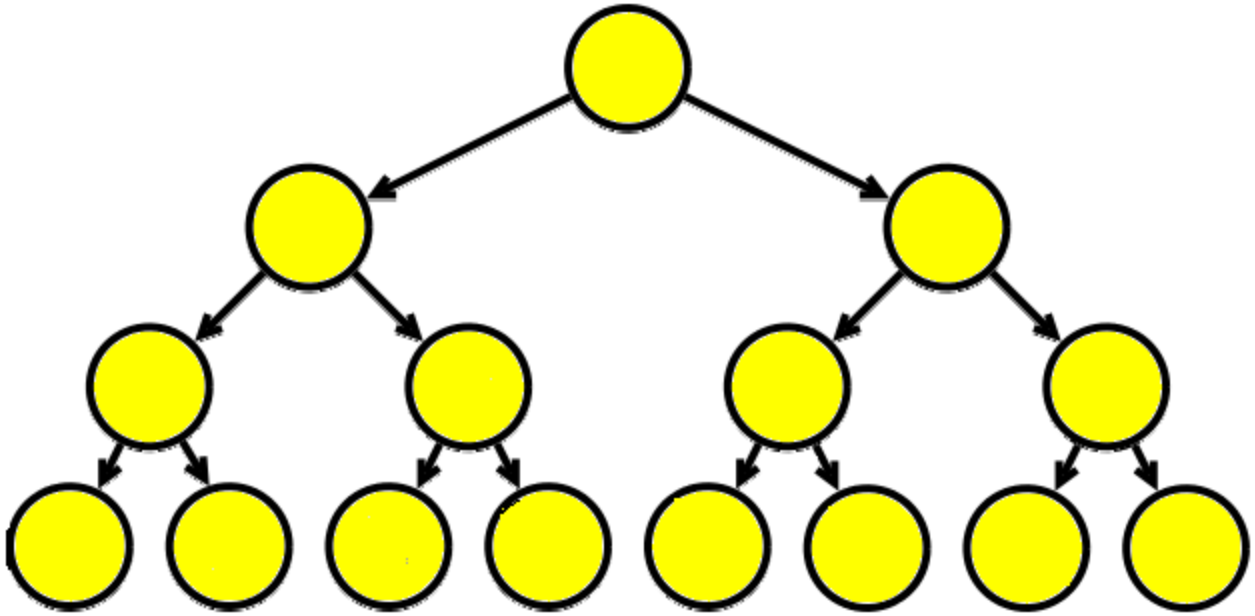
Left complete binary tree- fill in last level of a complete binary tree from left to right.



Left complete binary tree- fill in last level of a complete binary tree from left to right.



Left complete binary tree- fill in last level of a complete binary tree from left to right.



One way to build a heap:

Add the new entry in the position which is the next slot of a left-complete binary tree. Then bubble-up to restore the heap property.

Max-heap: The data value at each node is greater than or equal to the data values of its children.

Bubble-up pseudocode (swim):

While the current node is not the root and current.data is greater than the data value of the parent of current

{

1. Swap the data values in nodes current and the parent of current.
2. Set the current node to be the parent of current.

}

Then to sort (like a Max Sort but now the heap is used to determine the max):

Repeat:

Delete the max from the heap.

Fix the heap.

Until the heap is empty.

Problem of the Day

1. Insert the following keys into a heap using a bubbleUp approach:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

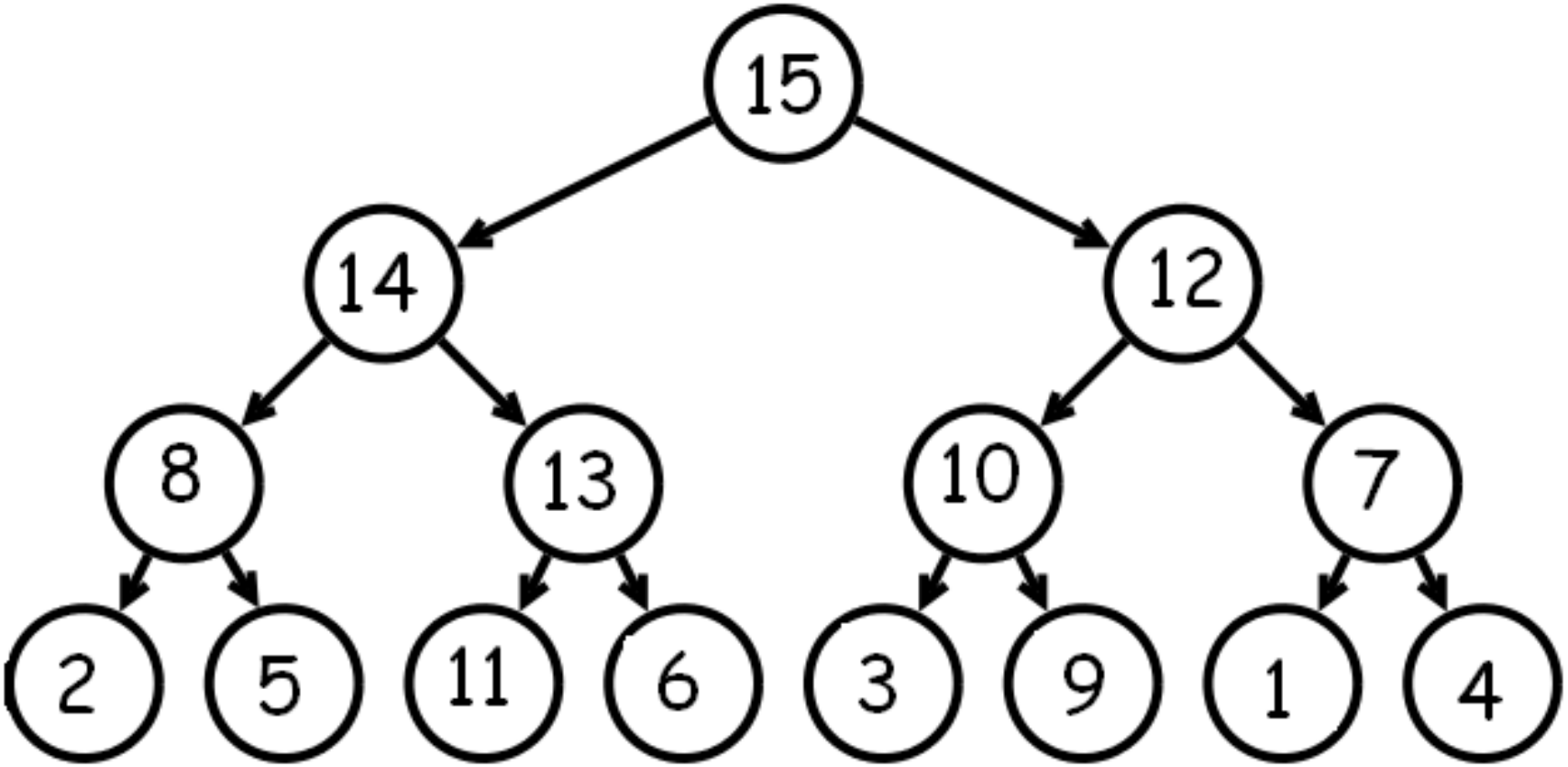
2. How much time does the compare method from Assignment 2A take on:

x= 2 2 2 … 2

Y= 0 0 0 …. 0 2 2 2 … 2

where x and y each have n 2's and y has n^5 leading zeroes?

Apply repeated deleteMax operations to this heap in order to sort the data:



DeleteMax:

To fix the heap after deleting the max:

Take the data value which is in the slot which will disappear when the left-complete binary tree has one node removed.

Place it at the top of the heap (where the data value was just deleted).

Then Bubble-down this data value in order to restore the heap property.

To Bubble-down (sink):

Repeat:

Swap the data value with the largest of its two children until either the data value moves to the bottom of the tree, or it is greater than or equal to its children.

Implementation note: Near the bottom of the tree, a node may have 0 or 1 children instead of the usual two children.

Timing:

To build the heap: The roughly n/2 nodes at the bottom of the heap may all have to bubble up to the top of the heap. So the time to insert all the nodes is in $\Theta(n \log_2(n))$ in the worst case.

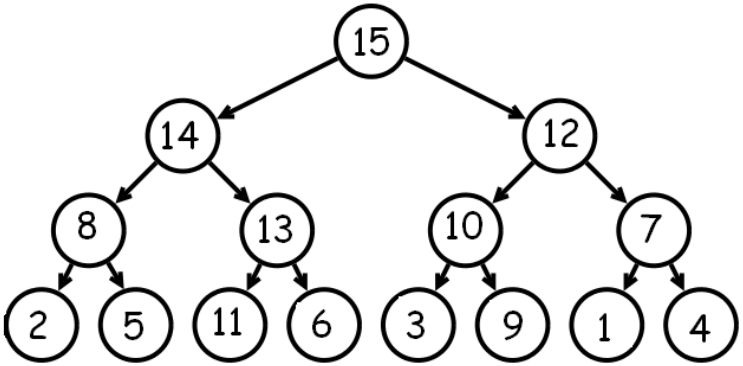
The DeleteMax operation is repeated n times and in the worst case, results in the value bubbling down to the bottom of the tree. Since roughly half the nodes may take $\log_2(n)$ time, the time complexity in the worst case is in $\Theta(n \log_2(n))$.

Show the result of inserting these values into a heap one at a time using bubble up with each insertion:

1, 2, 3, 4, 5, 6, 7

If you get finished early: finish off the heap sort by doing 7 deleteMax operations using bubble down to fix the heap at each step.

Heapsort



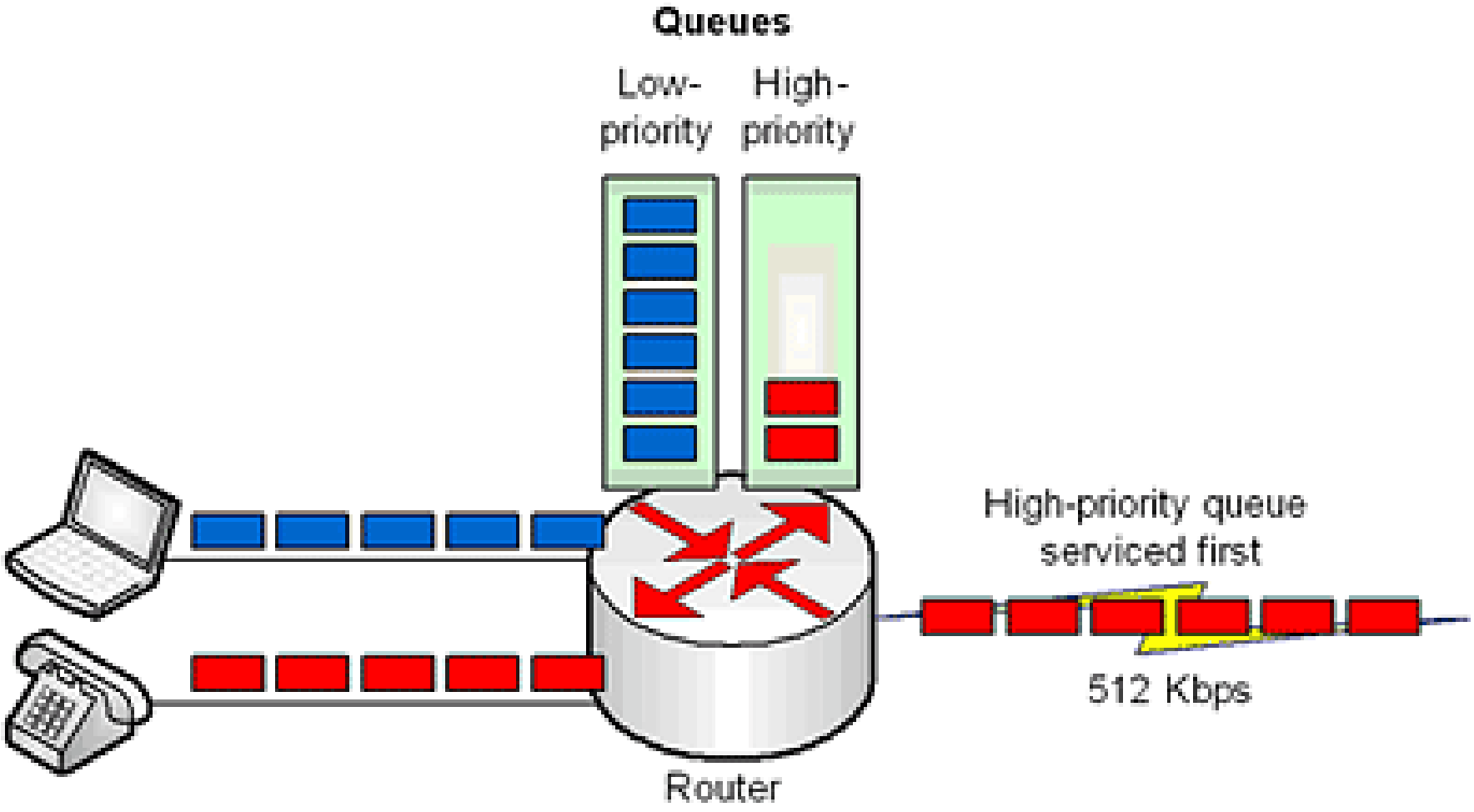
A Compost Heap

Madame Trash Heap

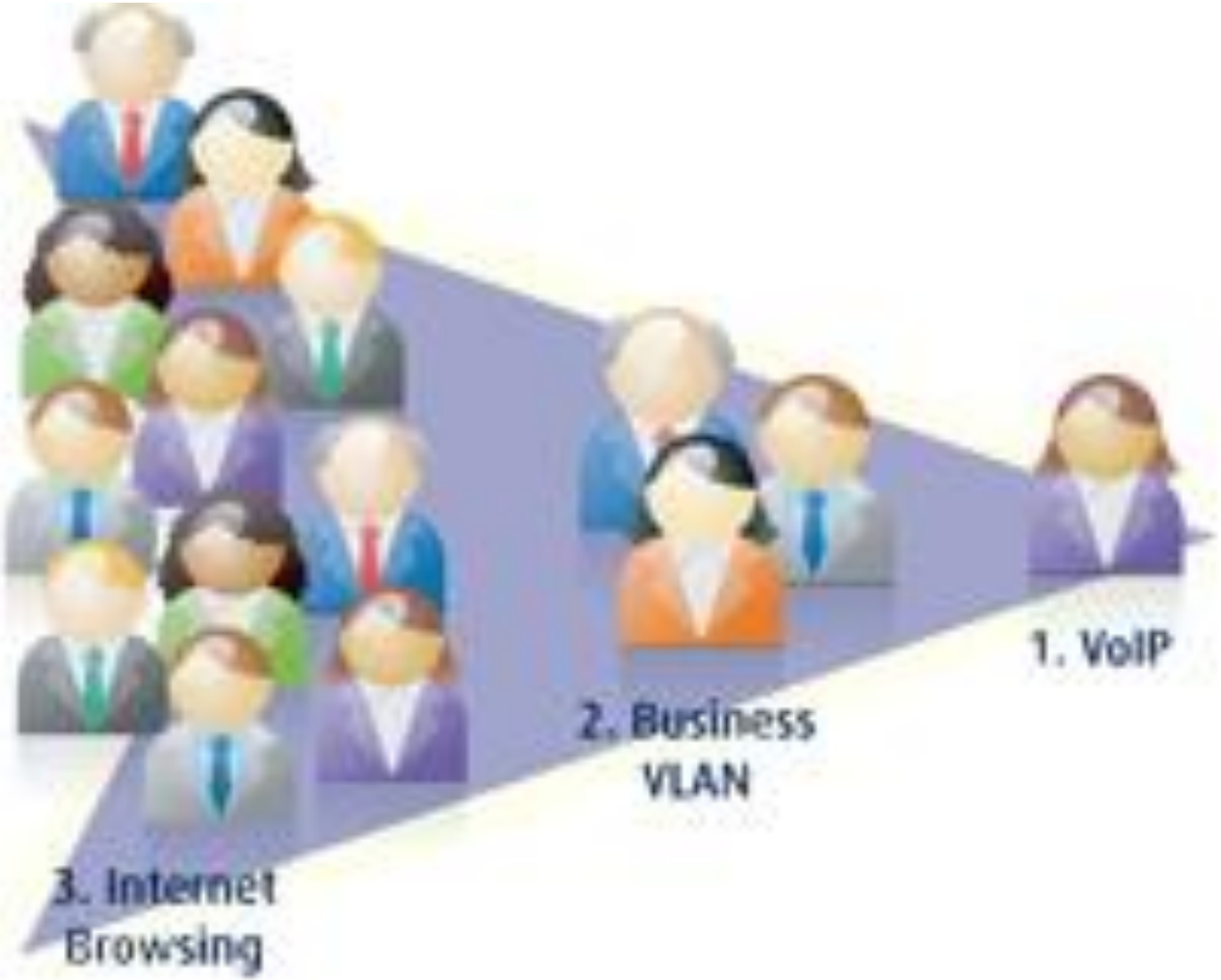
Pictures from:

http://www.compostinfo.com/tutorial/methods.htm

http://linguiniontheceiling.blogspot.com/

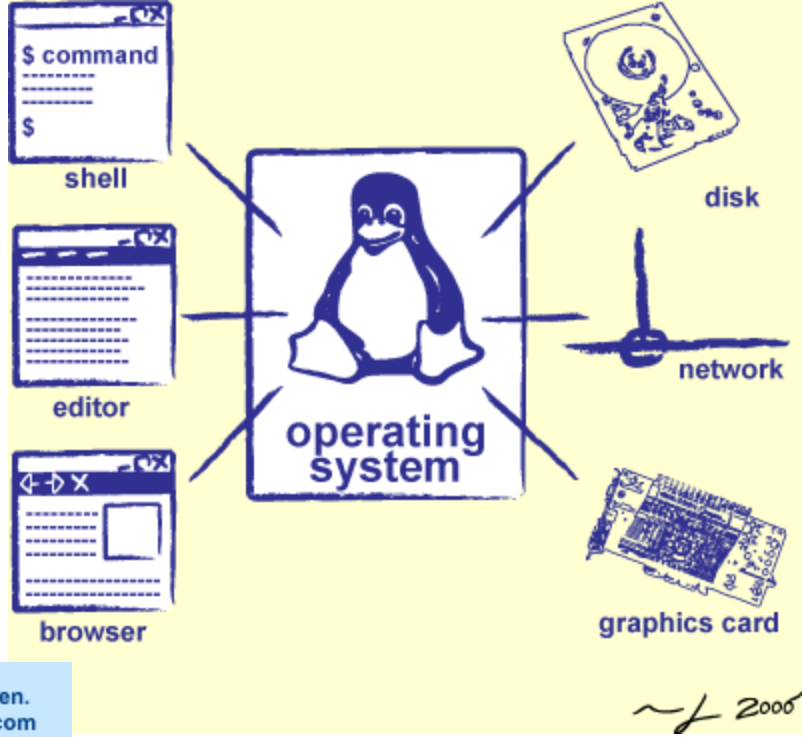


http://www.biztechmagazine.com/article.asp?item_id=318



http://www.eionwireless.com/vip_release/10_reasons.html

http://swc.scipy.org/lec/img/shell01/operating_system.png



"It's not just you. Most operating systems are unstable."

http://www.cse.chalmers.se/EDU/OS/PIC/06-09-2001.gif

What does a worst case example look like?

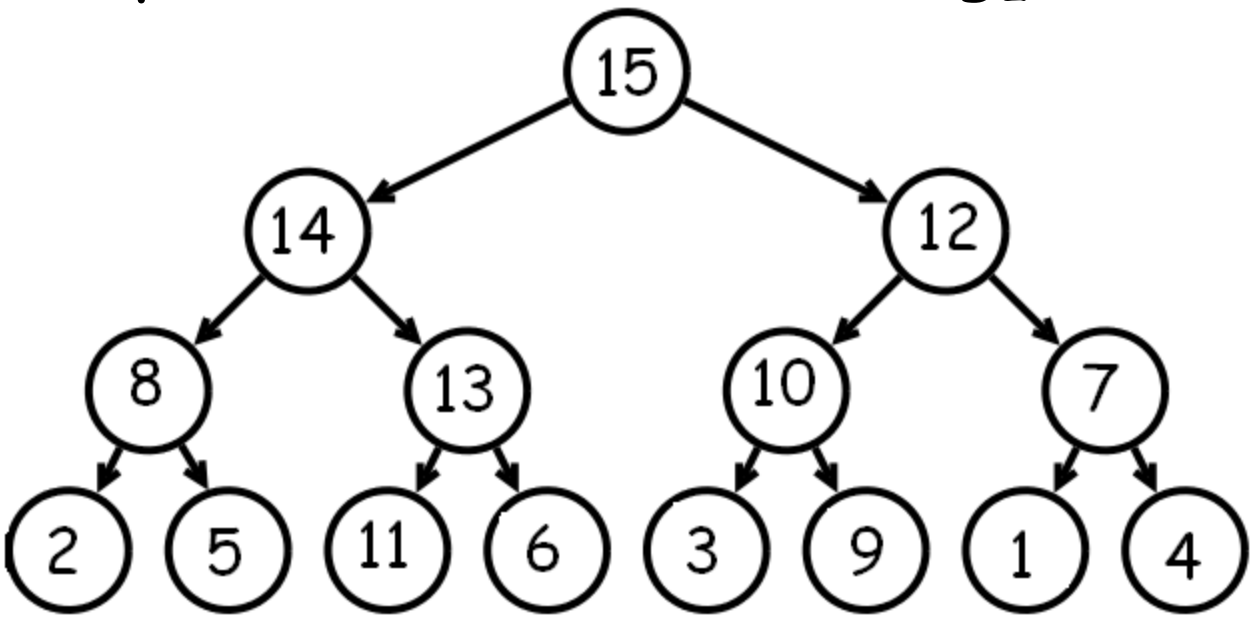
How much work is done to create the heap in the worst case using the bubble up strategy?

S(n) is proportional to the time taken:

$S(n) = 1*1 + 2*2 + 4*3 + 8*4 + \ldots + 2^h*(h+1)$

where $n = 1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1$.

How can we prove that $S(n) \in \Theta(n \log_2(n))$?



Assume that T, f and g are functions mapping the natural numbers {0, 1, 2, 3, ...} into the reals.

Definition: "Omega" A function T(n) is in $\Omega(f(n))$ if there exist constants $n_0 \geq 0$, and $c > 0$, such that for all $n \geq n_0$, $T(n) \geq c * f(n)$.

Recall: $n = 1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1$.

Proof that $S(n) \in \Omega(n \log_2(n))$:

Use technique for getting a lower bound on a sum: $S(n) =$

$$1*1 + 2*2 + 4*3 + 8*4 + \ldots + 2^{h-1}*(h) + 2^h*(h+1) \geq$$

$$0 + 0 + 0 + 0 + \ldots + 0 + 2^h*(h+1)$$

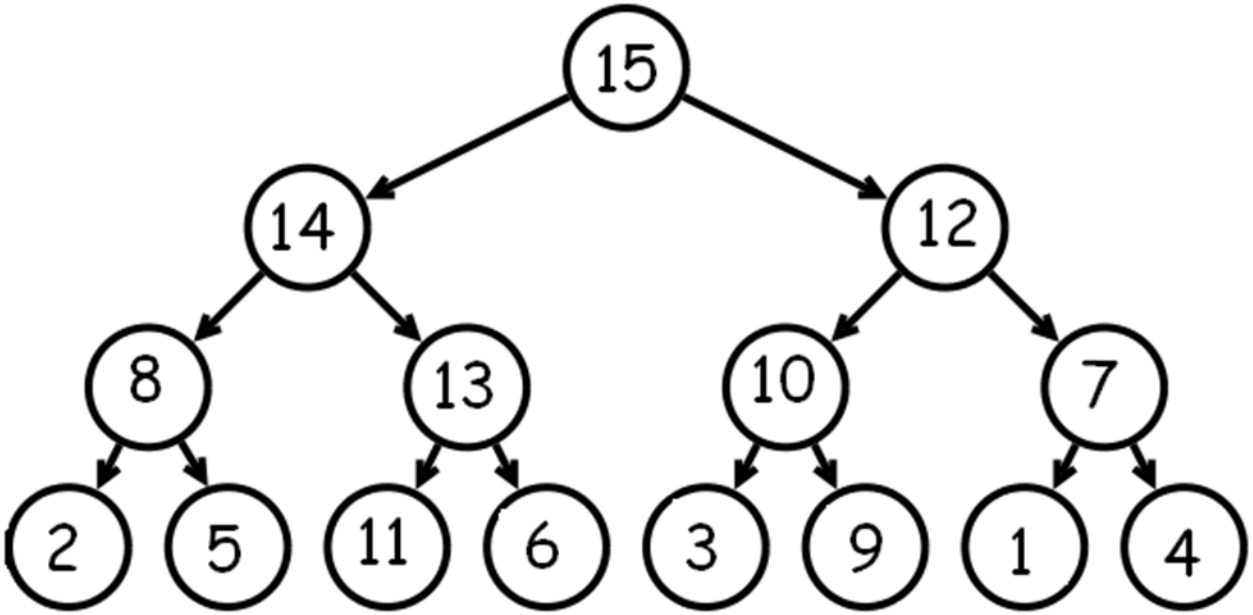
$$= (n+1)/2 * \log_2(n+1) \geq \tfrac{1}{2} * n * \log_2(n)$$

since $(n+1)/2 \geq \frac{1}{2} * n$ and $\log_2(n+1) \geq \log_2(n)$.

This is true for $n \geq 1$, so for the proof that $S(n) \in \Omega(n \log_2(n))$ we have that $c = \frac{1}{2}$ and $n_0 = 1$.

This is a typical lower bounding argument.

The key concept: Because for approximately $\frac{1}{2}$ of the insertions the time is in $\Omega(f(n))$, the total time is in $\Omega(n^* f(n))$.



Assume that T, f and g are functions mapping the natural numbers {0, 1, 2, 3, ...} into the reals.

Definition: "Big Oh" A function T(n) is in O(f(n)) if there exist constants $n_0 \geq 0$, and $c > 0$, such that for all $n \geq n_0$, $T(n) \leq c * f(n)$.

Recall: $n = 1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1$.

Proof that $S \in O(n \log_2(n))$:

Use technique for getting a upper bound on a sum:

$S = 1*1 + 2*2 + \ldots + 2^{h-1}*(h) + 2^h*(h+1) \leq$

$1*(h+1) + 2*(h+1) + \ldots + 2^{h-1}*(h+1) + 2^h*(h+1)$

$= (h+1) * [2^{h+1} - 1] = n * \log_2(n+1)$

It is obvious that it is not true that

$n * \log_2(n+1) \leq 1* \; n * \log_2(n)$.

So for our proof, let's try to find n large enough so that: $n * \log_2(n+1) \leq 2* \; n * \log_2(n)$.

Find n large enough: $\log_2(n+1) \leq 2^* \log_2(n)$

Making the math easier by looking at integers:

h	n	n+1	$\log_2(n+1)$	$\lfloor \log_2(n) \rfloor$	$2^* \lfloor \log_2(n) \rfloor$
0	1	2	1	0	0
1	3	4	2	1	2
2	7	8	3	2	4
3	15	16	4	3	6
4	31	32	5	4	8

We can see that for $n \geq 3$ ($h \geq 1$),

$\log_2(n+1) \leq 2^* \lfloor \log_2(n) \rfloor \leq 2^* \log_2(n)$

Theoretical justification that for $n \geq 3$,

$\log_2(n+1) \leq 2 * \lfloor \log_2(n) \rfloor \leq 2 * \log_2(n)$

Recall: $n = 1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1$.

$\log_2(n+1) = h+1$

$\lfloor \log_2(n) \rfloor = h$

So we have

$h+1 \leq 2 * h$

which is true for $h \geq 1$ (meaning $n \geq 3$)

And the last inequality is true because of the properties of the floor function.

Recall: $n = 1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1$.

Proof that $S \in O(n \log_2(n))$:

Use technique for getting a upper bound on a sum:

$S = 1*1 + 2*2 + \ldots + 2^{h-1}*(h) + 2^h*(h+1) \leq$

$1*(h+1) + 2*(h+1) + \ldots + 2^{h-1}*(h+1) + 2^h*(h+1)$

$= (h+1) * [2^{h+1} - 1] = n * \log_2(n+1)$

$\leq 2* n * \log_2(n)$ for $n \geq 3$.

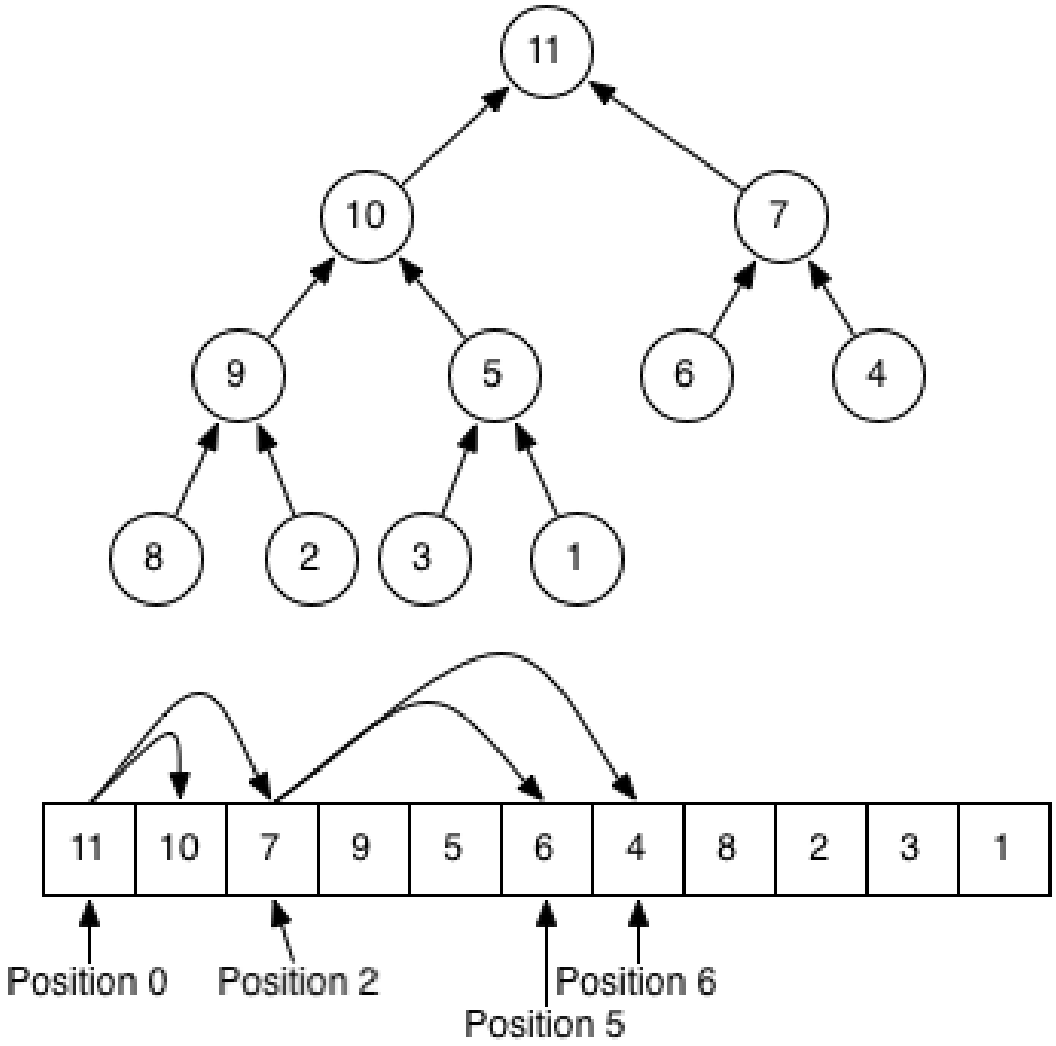
So for the proof that S is in

$O(n \log_2(n))$ we have that $c = 2$ and $n_0 = 3$.

Assume that T, f and g are functions mapping the natural numbers {0, 1, 2, 3, ...} into the reals.

Definition: "Theta" The set $\Theta(g(n))$ of functions consists of $\Omega(g(n)) \cap O(g(n))$.

Heaps can be stored in arrays with implicit parent/child pointers.

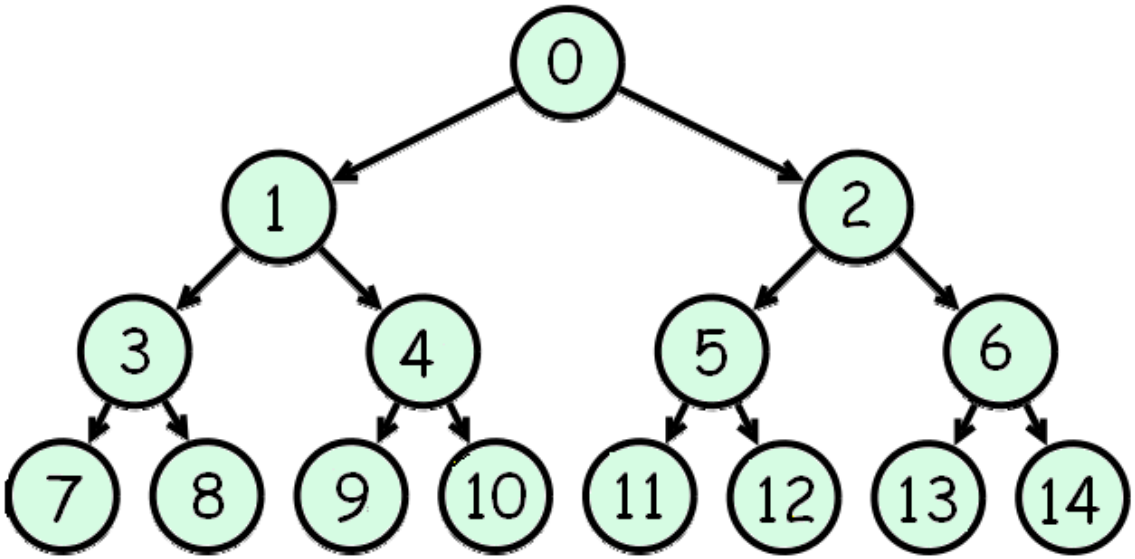


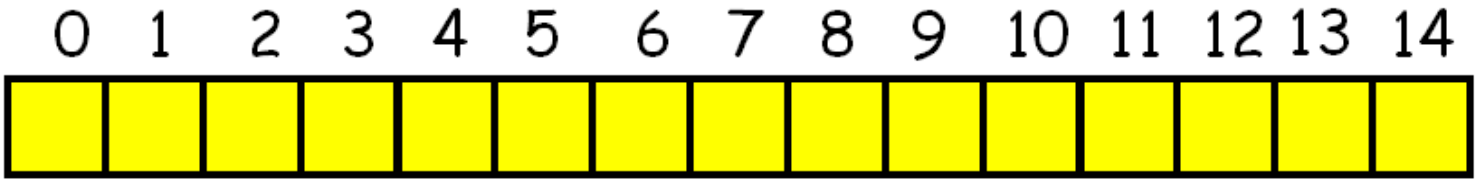
http://scienceblogs.com/goodmath/heap-array-example.png

Our text starts the array at 1 probably as an anachronism from Pascal. We will start at 0 since we are programming in Java or C.

What is the array index of:

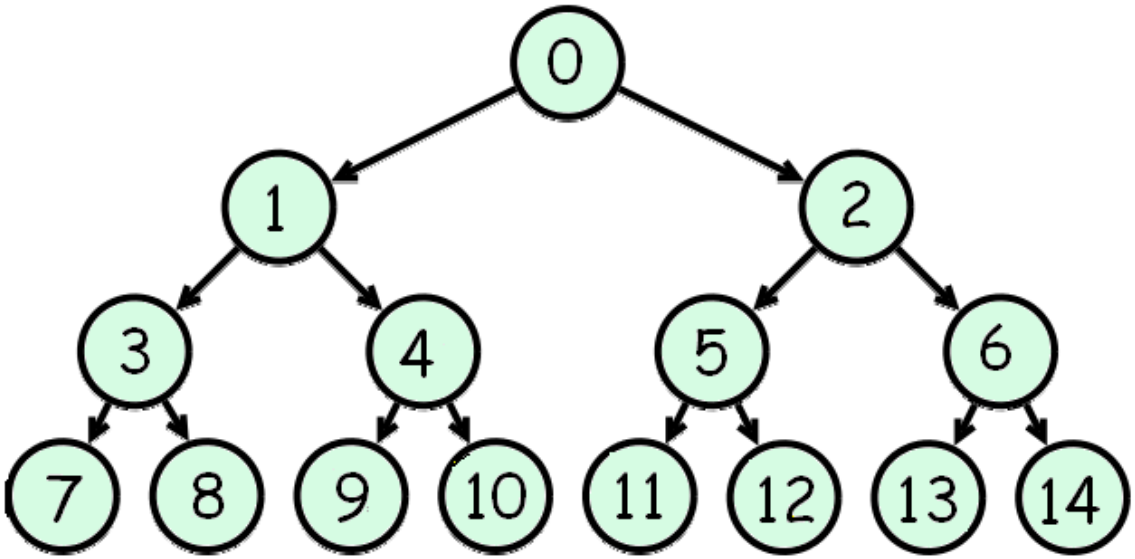
The parent of the node in position k?

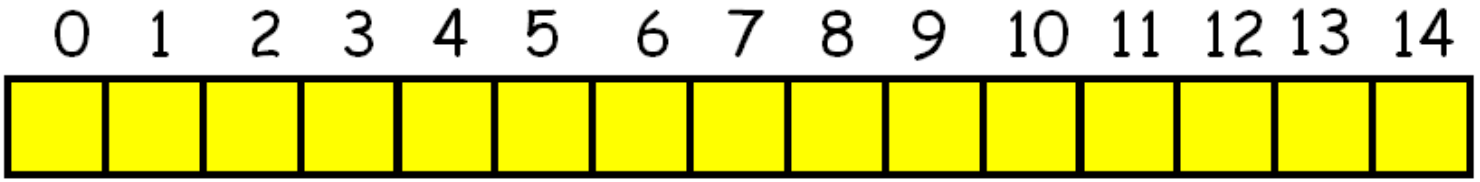




What is the array index of:

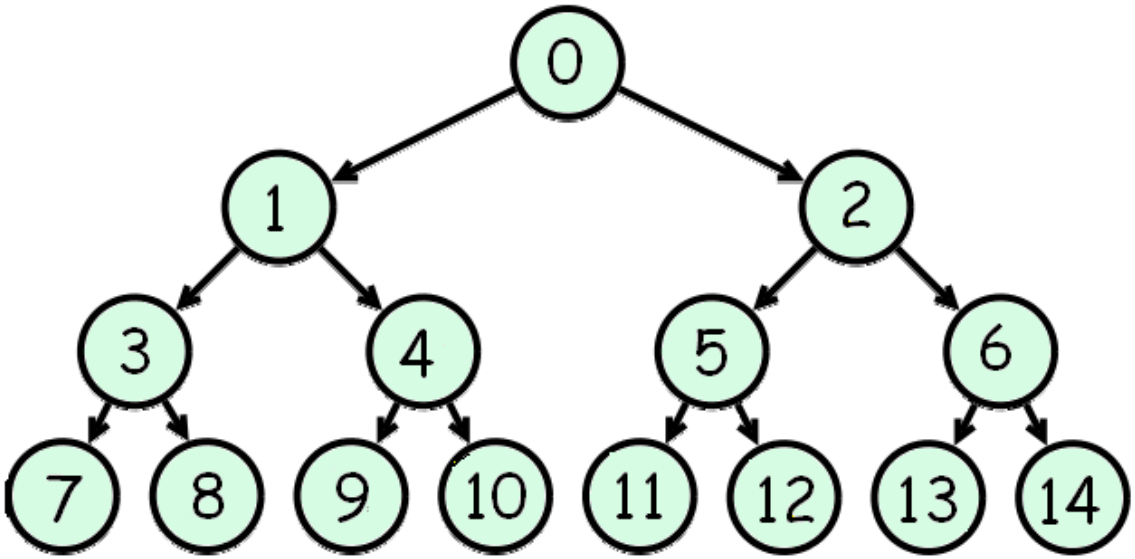
The left child of the node in position k?

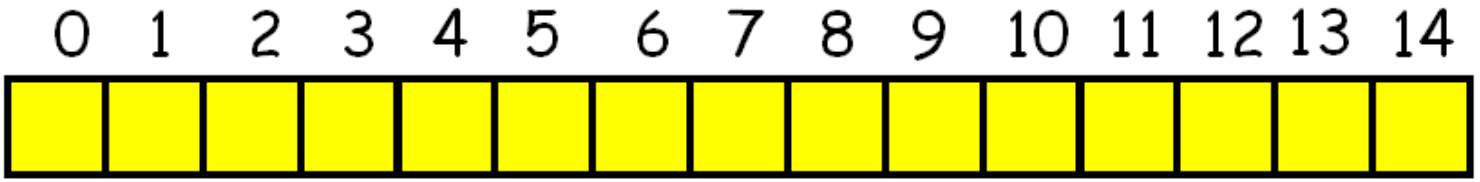




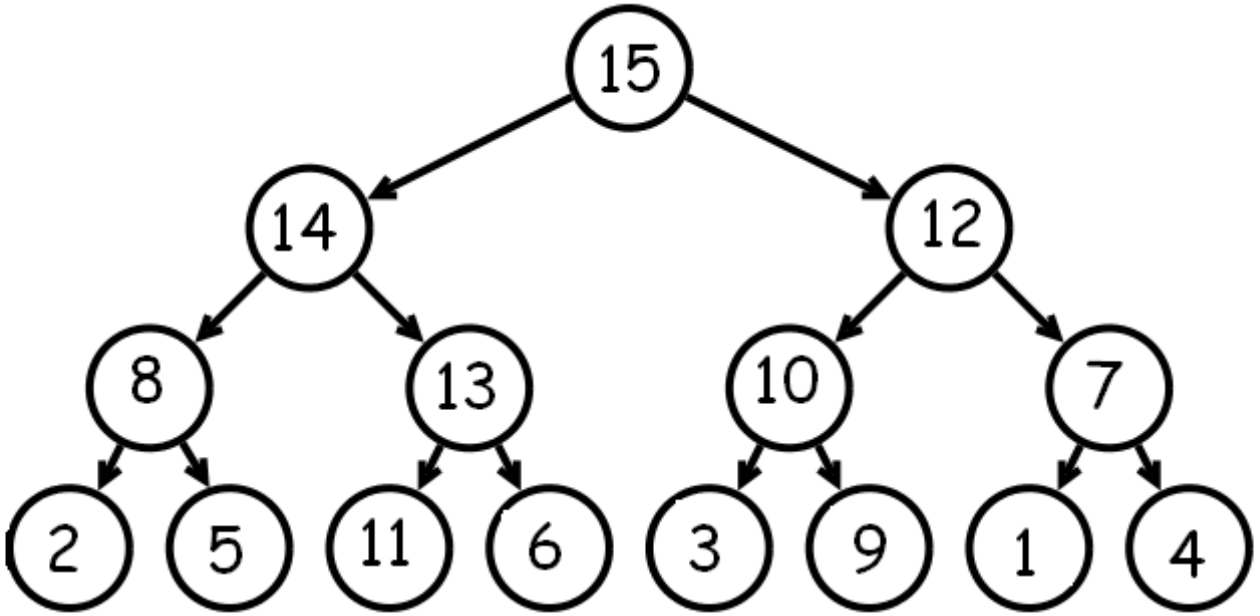
What is the array index of:

The right child of the node in position k?



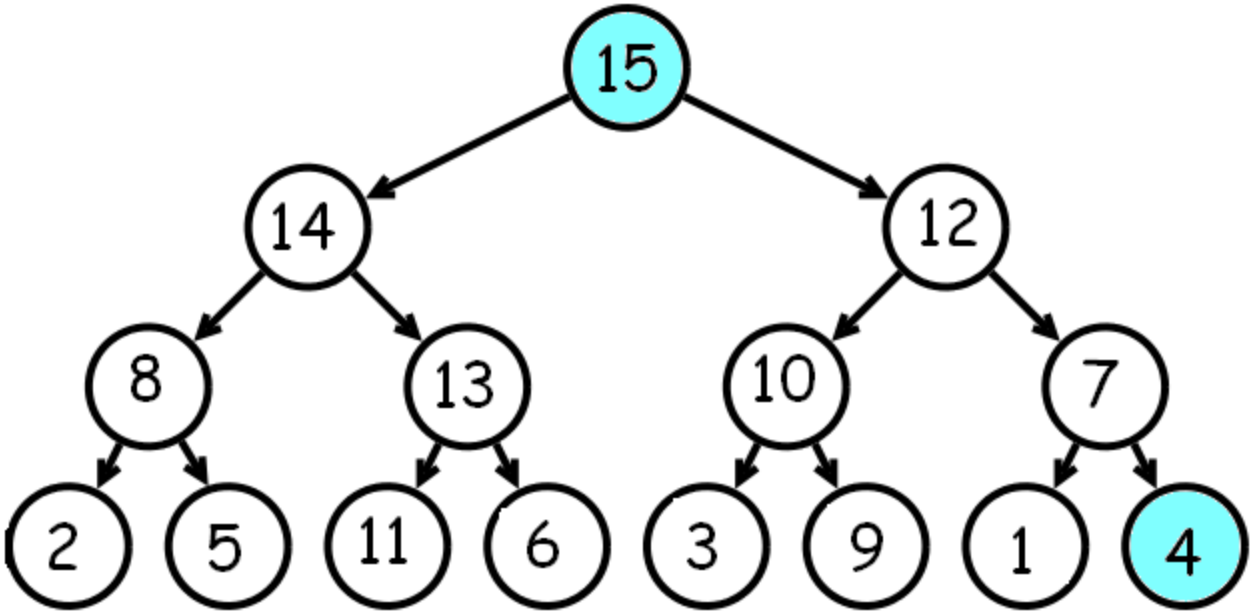


Example:

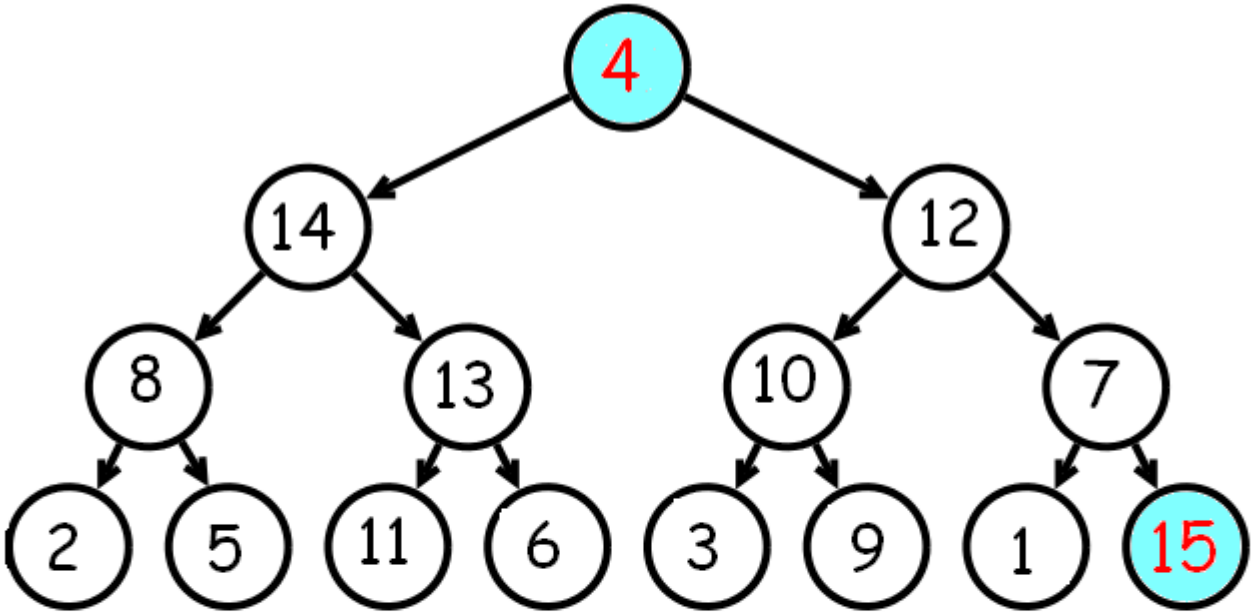


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
15	14	12	8	13	10	7	2	5	11	6	3	9	1	4

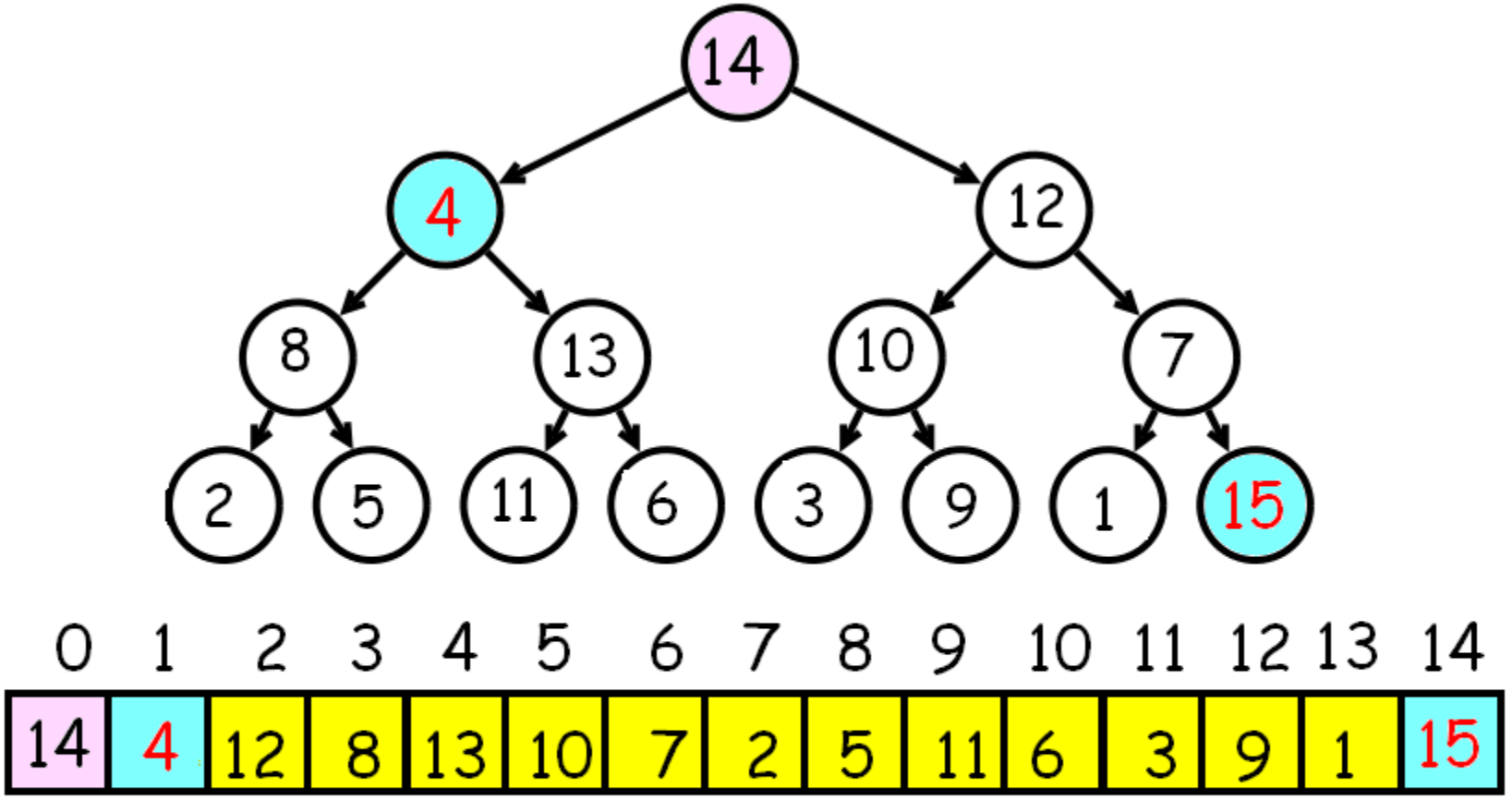
Deleting the max using an array:

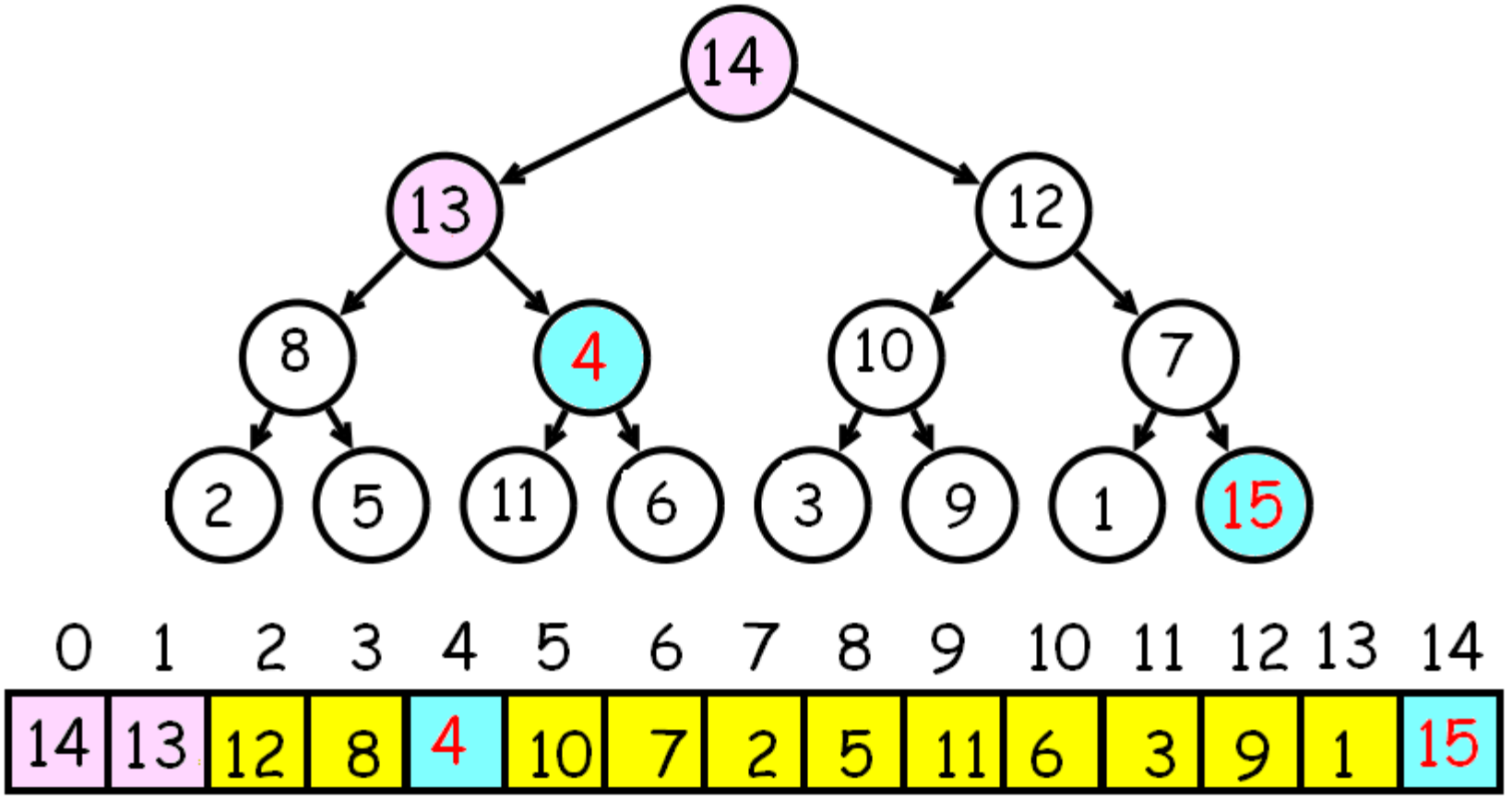


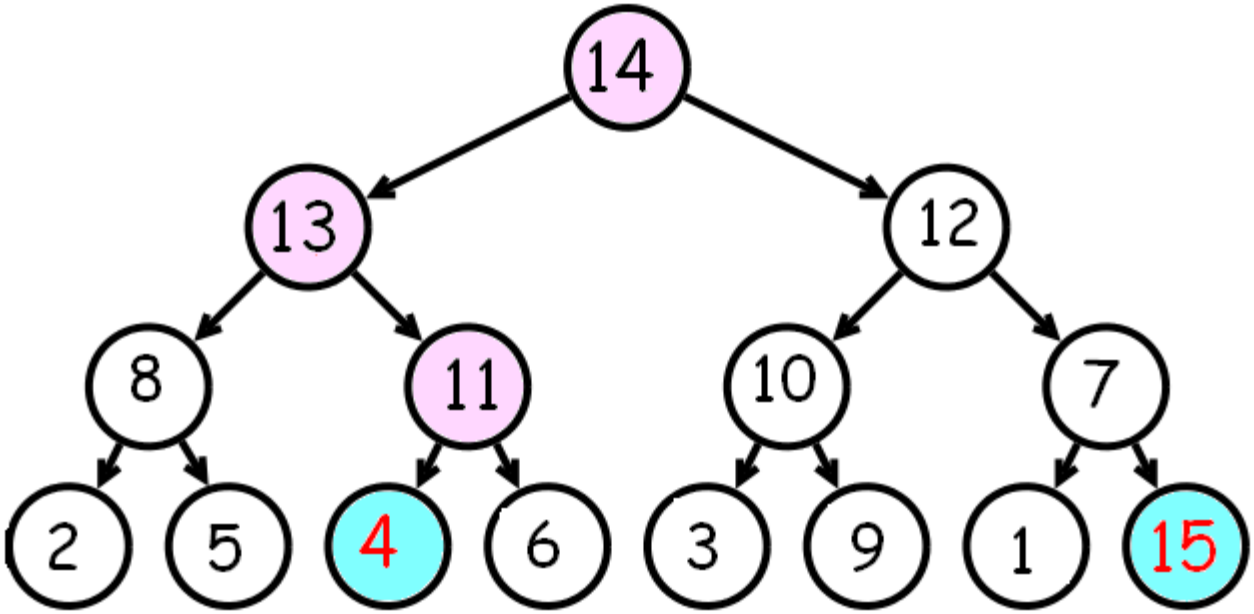
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
15	14	12	8	13	10	7	2	5	11	6	3	9	1	4



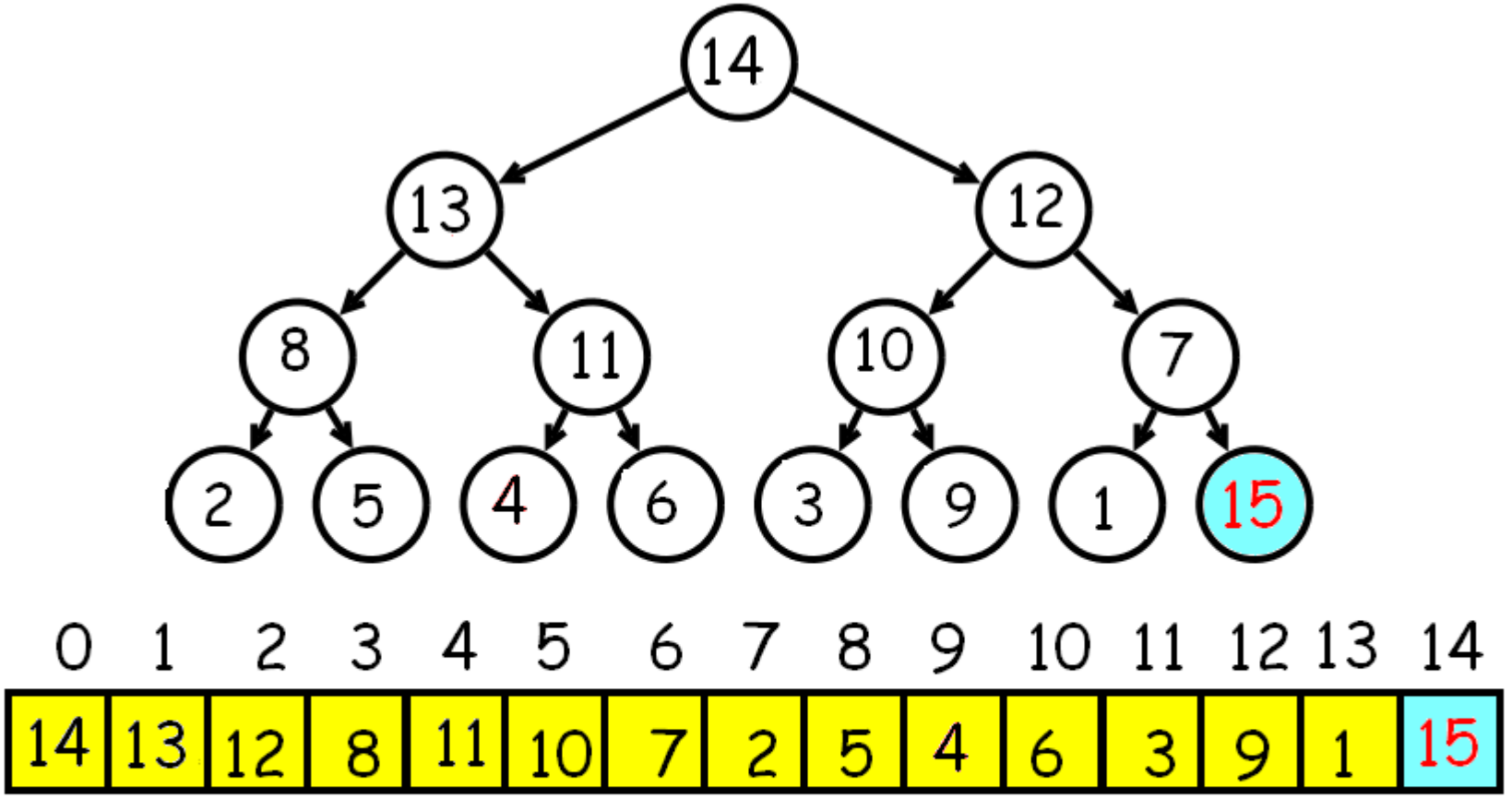
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
4	14	12	8	13	10	7	2	5	11	6	3	9	1	15


14
4
12
8
13
10
7
2
5
11
6
3
9
1
15
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
14 4 12 8 13 10 7 2 5 11 6 3 9 1 15

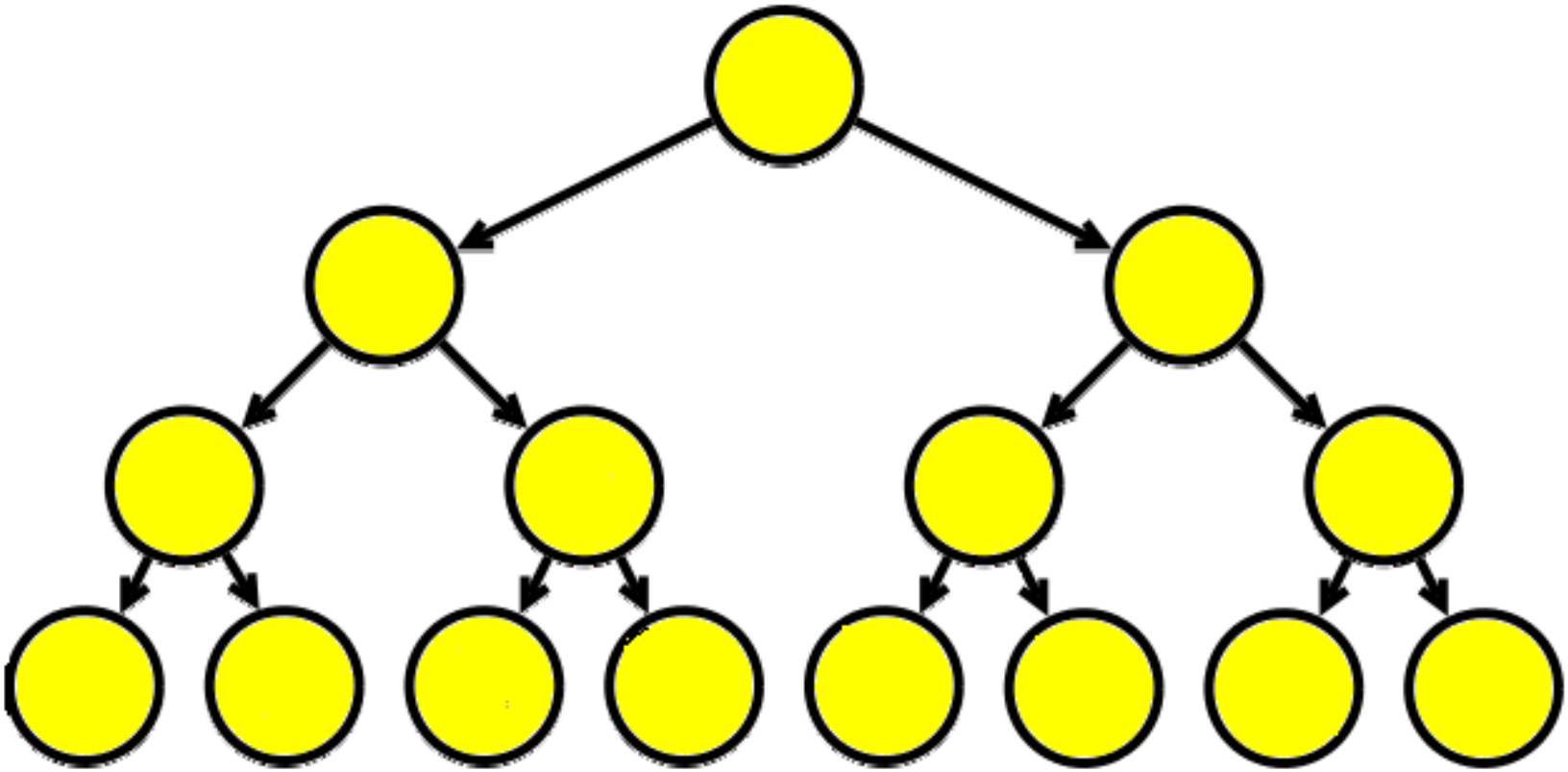

14
13
12
8
4
10
7
2
5
11
6
3
9
1
15
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
14 13 12 8 4 10 7 2 5 11 6 3 9 1 15



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	13	12	8	11	10	7	2	5	4	6	3	9	1	15


14
13
12
8
11
10
7
2
5
4
6
3
9
1
15
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
14 13 12 8 11 10 7 2 5 4 6 3 9 1 15

It's possible to build a heap in O(n) time:



To heapify at a node r:

Heapify the left subtree.

Heapify the right subtree.

Bubble down the data value at node r.

Non-recursively:

For (i= ?? ; i >=0; i--)

    Bubble-down the data value at position i.

How long does this take in the worst case?

Let T(h) be the time to heapify a complete binary tree of height h.

T(0)= 0 since nothing has to be done with only one node present to make it a heap.

$T(h) = h + 2\ T(h-1)$

Since we first make the left and right subtrees into a heap in 2T(h-1) time and then bubble-down the data value at the root taking time proportional to h.

$T(h) = h + 2\ T(h-1),\ T(0) = 0.$

This is not a fun recurrence to solve, but you should be able to get to the point where the recurrence is expressed as a sum.

You also should be able to prove by induction that the solution to this recurrence is:

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

What does this say in terms of n about the worst case time complexity for constructing a heap using this approach?

Suppose that one node in the priority queue (heap) has its data value changed.

What should you do to fix a max-heap if its data value:

1. increases?

2. decreases?

3. How much time does this take in the worst case?

How hard is it to find a minimum key value in a max-heap?