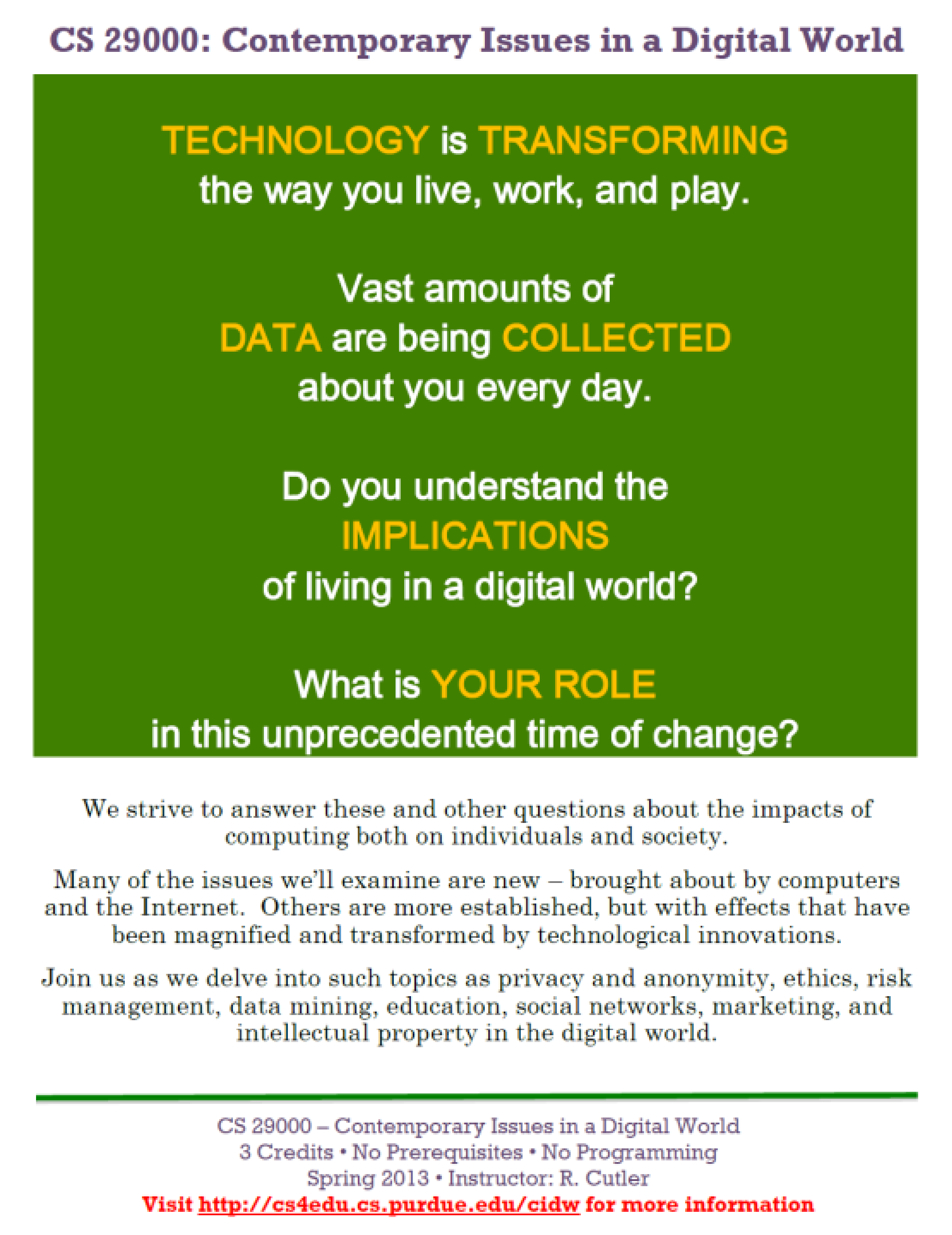
Announcements

- Project 5 is also a team project
  - This will include the brief essay question for the team course of Science
  - The essay question is to be submitted separately and individually
- New course CS 290 00, Spring 2013
  Contemporary Issues in a Digital World
  Instructor: Robb Cutler.
  Syllabus at http://cs4edu.cs.purdue.edu/cidw



CS 29000: Contemporary Issues in a Digital World

TECHNOLOGY is TRANSFORMING
the way you live, work, and play.

Vast amounts of
DATA are being COLLECTED
about you every day.

Do you understand the
IMPLICATIONS
of living in a digital world?

What is YOUR ROLE
in this unprecedented time of change?

We strive to answer these and other questions about the impacts of computing both on individuals and society.

Many of the issues we'll examine are new – brought about by computers and the Internet. Others are more established, but with effects that have been magnified and transformed by technological innovations.

Join us as we delve into such topics as privacy and anonymity, ethics, risk management, data mining, education, social networks, marketing, and intellectual property in the digital world.

CS 29000 – Contemporary Issues in a Digital World
3 Credits • No Prerequisites • No Programming
Spring 2013 • Instructor: R. Cutler
Visit http://cs4edu.cs.purdue.edu/cidw for more information

Developing Efficient Algorithms

- Find max once in a list
- Selection sort – O(n^2)
- Find max several times
- Heaps and Priority Queues
- Better repeated min finding
- Heap Sort – O(n log(n))
- Merge Sort – O(n log(n))

Finding max in List

- $O(n)$ algorithm:

```
mp = 0
for j in range(len(L)):
    if L[mp] < L[j]:
        mp = j
```

- Can we do it faster?
- Well, if L were sorted, in descending order, then
  - L[0] would be the max and mp==0 would be the answer
  - The complexity would be $O(1)$
- But sorting takes time; let’s see what we can do

CQ: Arithmetic Series

Let $T(n) \ = \ n + (n-1) + (n-2) + \cdots + 2 + 1$. Then

A. T(n) is O(n)

B. T(n) is O(n^2)

C. T(n) is O(n^3)

Selection Sort

- Algorithm:
  1. Find max in L[0:]
  2. Exchange max with L[0]
  3. Repeat with L[1:], L[2:], …, L[-2]

- In Python:

```
N = len(L)
for k in range(N):
    mp = k
    for j in range(k+1,N):
        if L[mp] < L[j]:
            mp = j
    L[mp], L[k] = L[k], L[mp]
# L is now sorted in descending order
```

How it works

[2,4,3,9,8]	k=0
[2,4,3,9,8]	mp=0, j=1, mp=1
[2,4,3,9,8]	mp=1, j=2
[2,4,3,9,8]	mp=1, j=3, mp=3
[2,4,3,9,8]	mp=3, j=4
[**9**,4,3,2,8]	k=1
[**9**,4,3,2,8]	mp=1, j=2
[**9**,4,3,2,8]	mp=1, j=3
[**9**,4,3,2,8]	mp=1, j=4, mp=4
[**9**,**8**,3,2,4]	k=2
[**9**,**8**,3,2,4]	mp=2, j=3
[**9**,**8**,3,2,4]	mp=2, j=4, mp=4
[**9**,**8**,**4**,2,3]	k=3
[**9**,**8**,**4**,2,3]	mp=3, j=4, mp=4
[**9**,**8**,**4**,**3**,2]	k=4
[**9**,**8**,**4**,**3**,**2**]	k=5

$O(n^2)$ – that is too much !

- Selection sort repeatedly extracts max, O(n) times
- Each max extraction is O(n) comparisons

So selection sort is $O(n^2)$ -- not very good

- Problem:
  - After extracting the max, we get no help extracting the max in the remaining slice
  - Can we fix that?

Digression

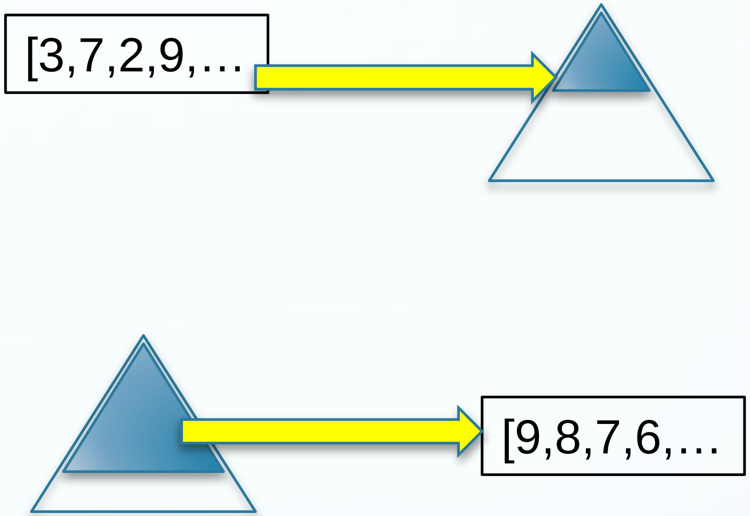
- Queuing up at the gate in the airport:
  - First customer at the gate is first customer on the plane
  - First one into the queue is first one out
- But (almost all) airlines run a priority queue:
  - Of all customer in the queue they take the one(s) with the highest priority first!
- So, items in the queue have each a priority. We want to take the items by decreasing priority.
  - That is like extracting max repeatedly…

Two Phases using a Priority Queue

1. Put all items in the input list into a priority queue

2. Extract items by decreasing magnitude and put them into the output list

We get a sorted list that way



[3,7,2,9,…

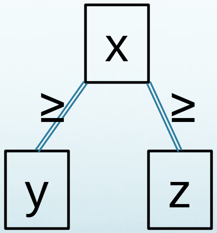
[9,8,7,6,…

Priority Queue Ingredients

1. A flatly-encoded full binary tree
   - The node labels are the numbers we are trying to sort
2. An ordering property of the tree nodes
   - Each time we insert into or delete from the queue this ordering has to be maintained/restored

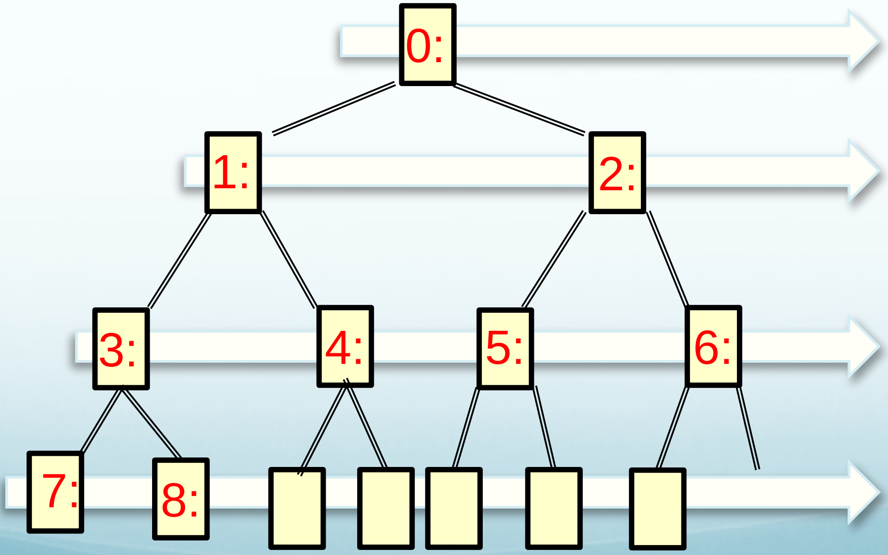
Priority Queue Ordering

- A special binary tree: filled layer by layer
  - That way we don’t have to nest lists in the encoding
  - Accessing nodes by an index computation
- Tree nodes are numbers (priorities)
- Locally, higher priority is above lower priority in the tree:
  - x ≥ y and x ≥ z:

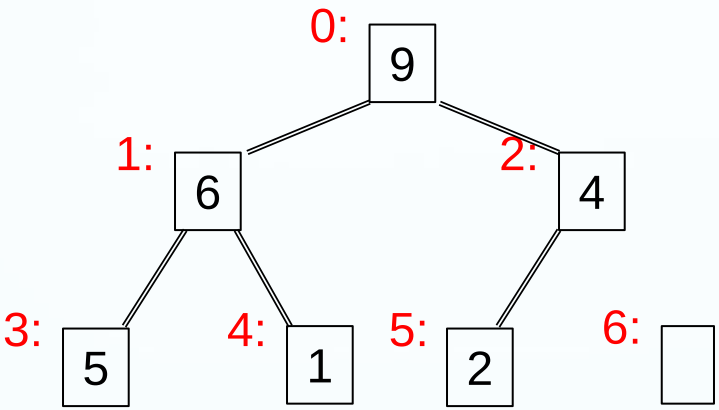


Full Binary Tree Indexed

- Think of a complete binary tree of arbitrary depth
- Enumerate the tree nodes, layer by layer
- Those are the indices of the encoding in which we use only a single list
- Need to explain access, insertion, deletion



Example Tree, Encoding & Access



Tree encoding:

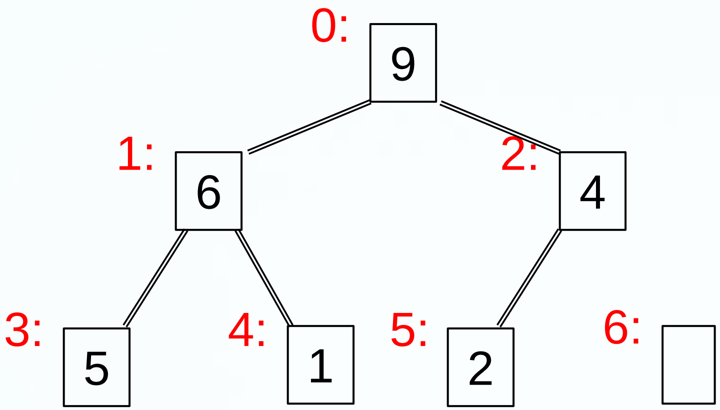
[9, 6, 4, 5, 1, 2]

Access Mappings:

Parent to left child: $k \;{-}{>}\; 2k + 1$

Parent to right child: $k \;{-}{>}\; 2k + 2$

Child to parent: $k \;{-}{>}\; (k - 1)//2$



Tree encoding:
[9, 6, 4, 5, 1, 2]

Mappings:

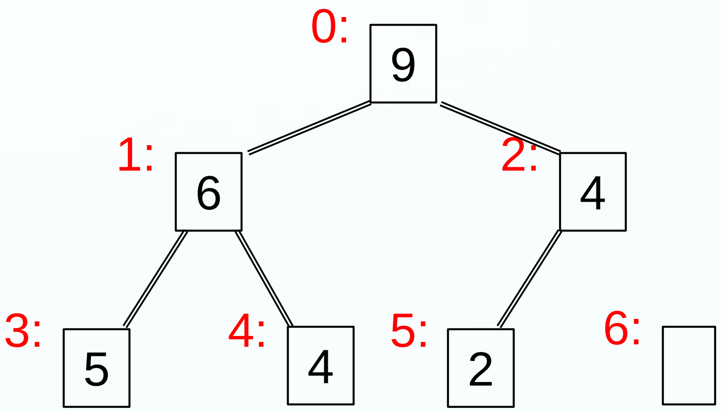
Parent to left child: $k \; {-}{>} \; 2k + 1$

Parent to right child: $k \; {-}{>} \; 2k + 2$

Child to parent: $k \; {-}{>} \; (k - 1)//2$

- At root (labeled 9, index 0), go to left child (labeled 6):
  $0 \; {-}{>} \; 2 * 0 + 1 = 1$
- At node labeled 6, index 1, go to right child (labeled 1):
  $1 \; {-}{>} \; 2 * 1 + 2 = 4$
- At child labeled 2, index 5, go to parent (labeled 4):
  $5 \; {-}{>} \; (5 - 1)//2 = 2$
- At child labeled 4, index 2, goto parent labeled 9):
  $2 \; {-}{>} \; (2 - 1)//2 = 0$

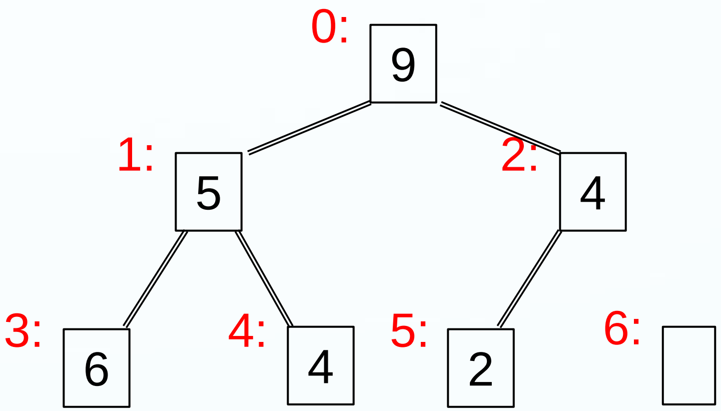
CQ: Is this a Priority Queue?



A. Yes

B. No

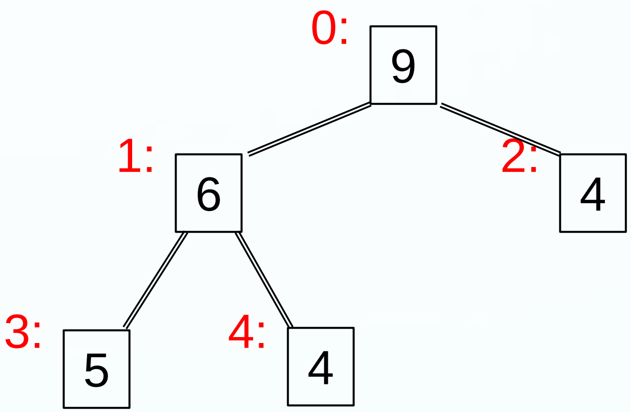
CQ: Is this a Priority Queue?



A. Yes

B. No

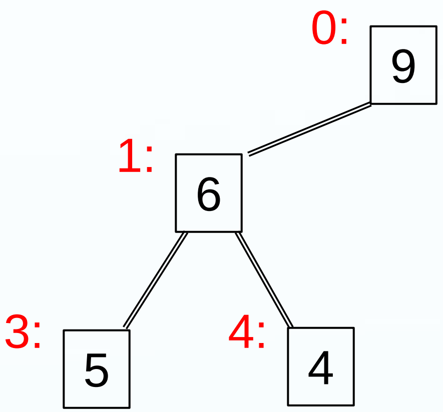
CQ: Is this a Priority Queue?



A. Yes

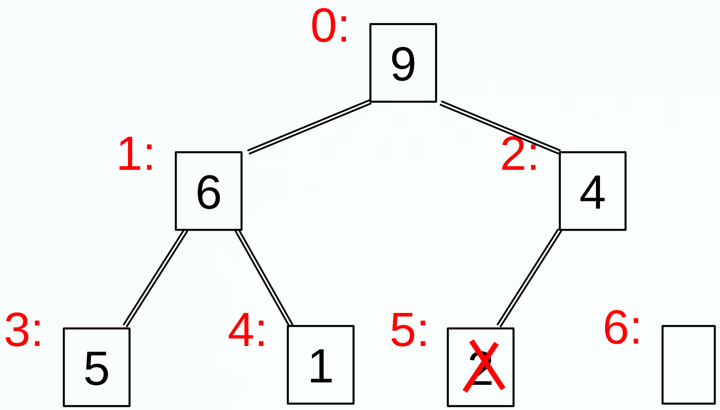
B. No

CQ: Is this a Priority Queue?



A. Yes

B. No



Tree encoding:
[9, 6, 4, 5, 1, ~~2~~]

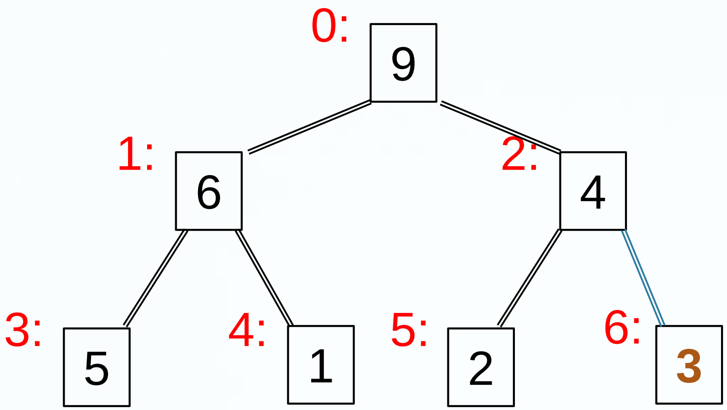
Mappings:

Parent to left child: $k \ -> \ 2k+1$

Parent to right child: $k \ -> \ 2k+2$

Child to parent: $k \ -> \ (k-1)//2$

- If we delete any node other than the last one, it will leave a hole in the tree and spoil the list encoding
- So we better only delete the last element in the list, i.e., the last leaf of the last layer (method pop())

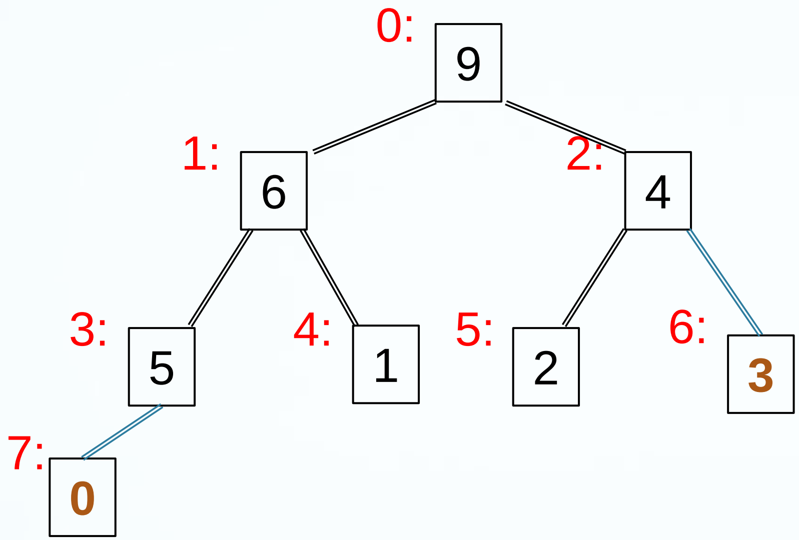


Tree encoding:
[9, 6, 4, 5, 1, 2, **3**]

Mappings:

Parent to left child:	$k \ {-}{>}\ 2k+1$
Parent to right child:	$k \ {-}{>}\ 2k+2$
Child to parent:	$k \ {-}{>}\ (k-1)//2$

- If we delete any node other than the last one, it will leave a hole in the tree and its list encoding
- So we better only delete the last element in the list, i.e., the last leaf of the last layer (function pop())
- Likewise, adding requires adding at the end of the list, i.e., the next leaf position in the last layer

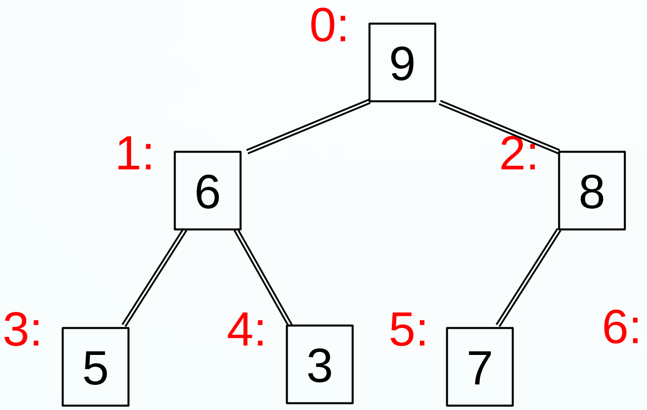


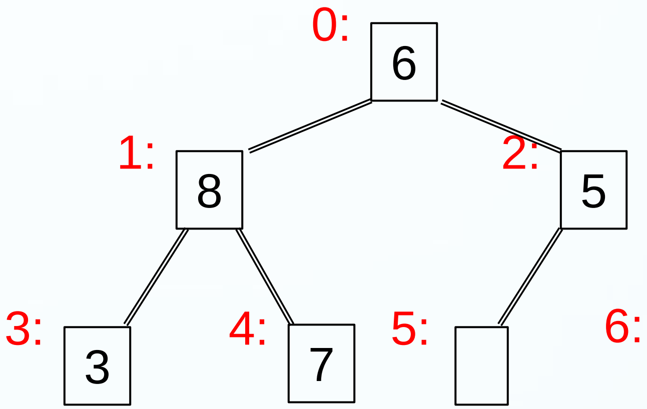
Tree encoding:
[9, 6, 4, 5, 1, 2, **3**, **0**]

Mappings:
Parent to left child: $k -> 2k + 1$
Parent to right child: $k -> 2k + 2$
Child to parent: $k -> (k - 1)//2$

- If we delete any node other than the last one, it will leave a hole in the tree and its list encoding
- So we better only delete the last element in the list, i.e., the last leaf of the last layer (function pop())
- Likewise, adding requires adding at the end of the list, i.e., the next leaf position in the last layer
  - If that layer is already full, we automatically start filling the next layer

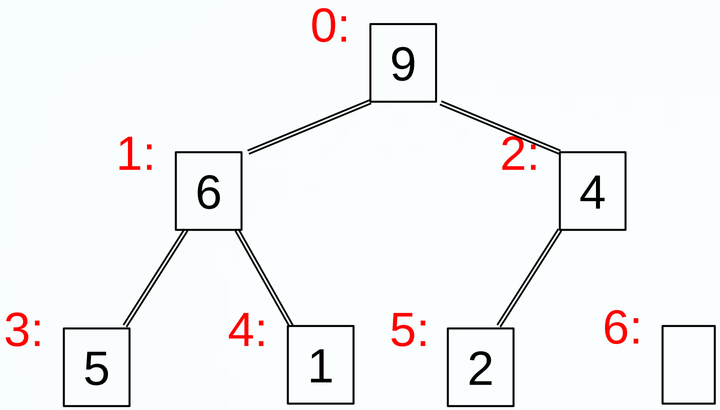
Improper Delete





[~~9~~,6,8,5,3,7]

[6,8,5,3,7]



Tree encoding:

[9, 6, 4, 5, 1, 2]

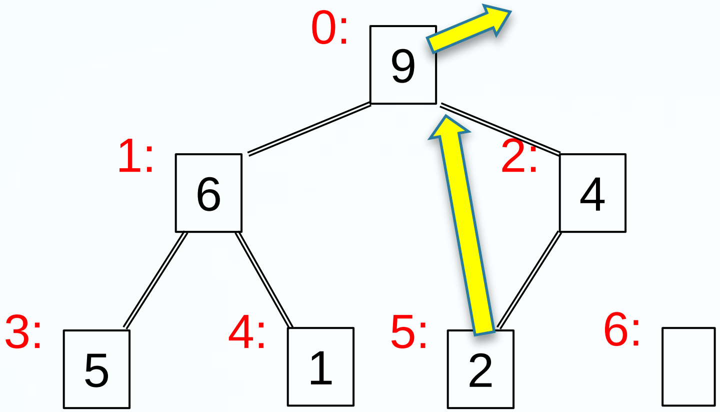
Mappings:

Parent to left child: $k \; -> \; 2k+1$

Parent to right child: $k \; -> \; 2k+2$

Child to parent: $k \; -> \; (k-1)//2$

- But if we want the max element, we must take the root!!
- More than that: only deleting at the end preserves the heap property
- What to do?



Tree encoding:
[9, 6, 4, 5, 1, 2]

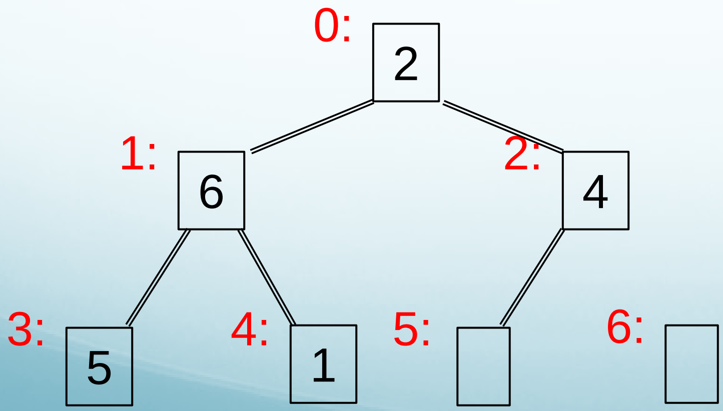
Mappings:

Parent to left child: $k -> 2k + 1$

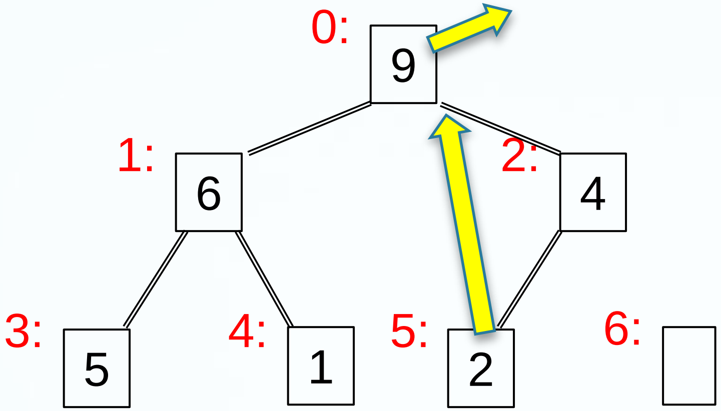
Parent to right child: $k -> 2k + 2$

Child to parent: $k -> (k - 1)//2$

- But if we want the max element, we must take the root!! And we need to keep it a heap
- Take the root and plug the hole with the “last leaf”



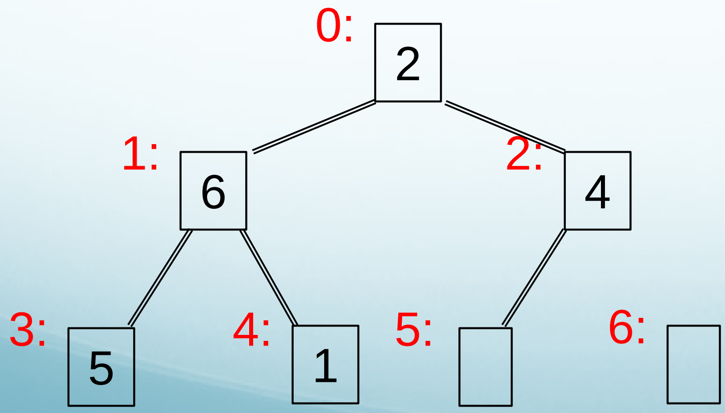
Tree encoding:
[9, 6, 4, 5, 1, 2]
➡ [2, 6, 4, 5, 1]



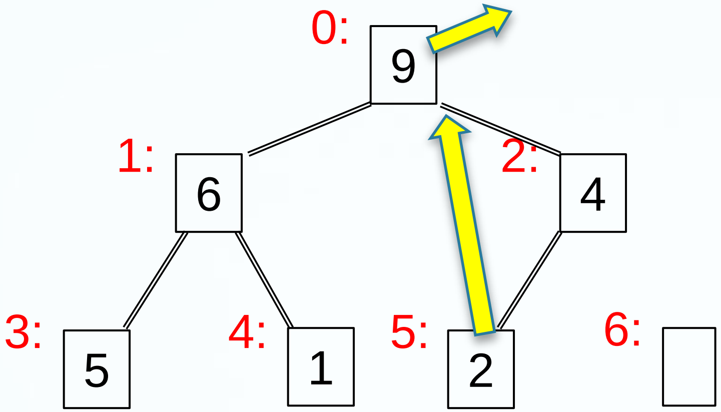
Tree encoding:
[9, 6, 4, 5, 1, 2]

Mappings:
Parent to left child: $k \text{ −> } 2k + 1$
Parent to right child: $k \text{ −> } 2k + 2$
Child to parent: $k \text{ −> } (k - 1)//2$

- Take the root and plug the hole with the “last leaf”
- Too bad that destroys the heap property ☹



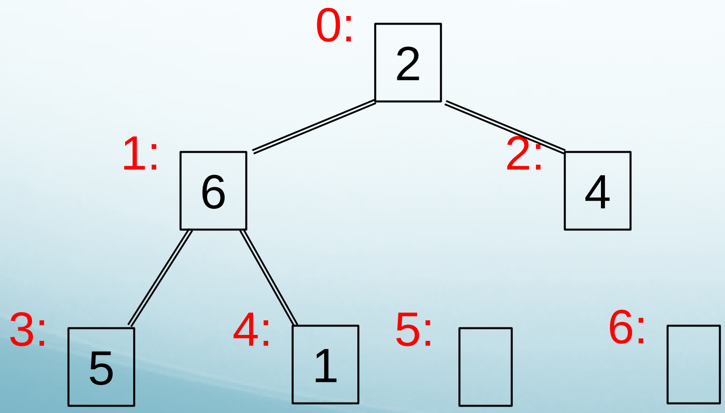
Tree encoding:
[9, 6, 4, 5, 1, 2]
→ [2, 6, 4, 5, 1]



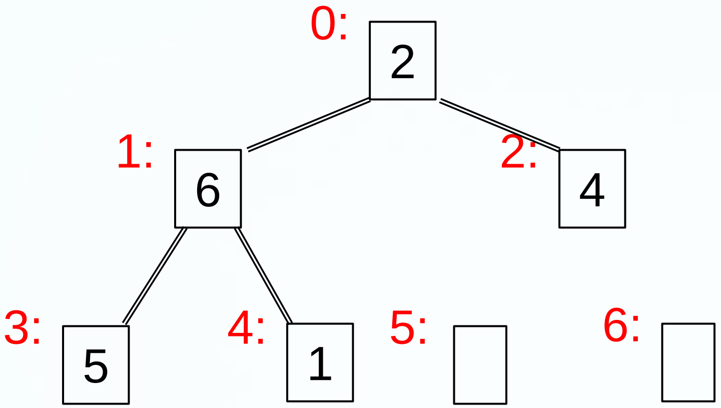
Tree encoding:
[9, 6, 4, 5, 1, 2]

Mappings:
Parent to left child: $k -> 2k + 1$
Parent to right child: $k -> 2k + 2$
Child to parent: $k -> (k - 1)//2$

- Take the root and plug the hole with the “last leaf”
- If it destroys the heap property – we better fix it...



Tree encoding:
[9, 6, 4, 5, 1, 2]
→ [2, 6, 4, 5, 1]



Mappings:

Parent to left child: $k \rightarrow 2k+1$

Parent to right child: $k \rightarrow 2k+2$

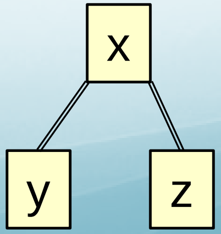
Child to parent: $k \rightarrow (k-1)//2$

Tree encoding:

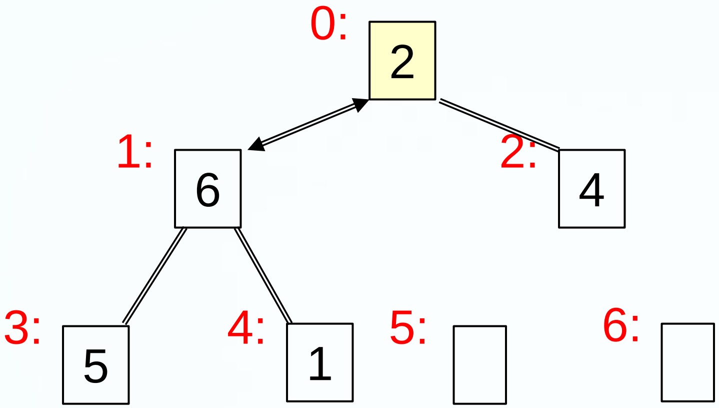
[9, 6, 4, 5, 1, 2]

➡ [2, 6, 4, 5, 1]

- Take the root and plug the hole with the “last leaf”
- Restoring the heap property – recursively:
  - Compare inserted node x with its (up to) 2 children y and z
  - If x ≥ y and x ≥ z, or x is a leaf, then stop
  - Otherwise swap x and the larger of its children
  - Repeat at the new position of x



x ≥ y
x ≥ z



Tree encoding:

[9, 6, 4, 5, 1, 2]

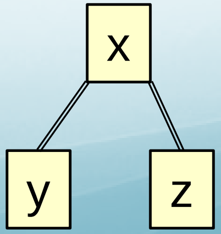
➡ [2, 6, 4, 5, 1]

Step 1:

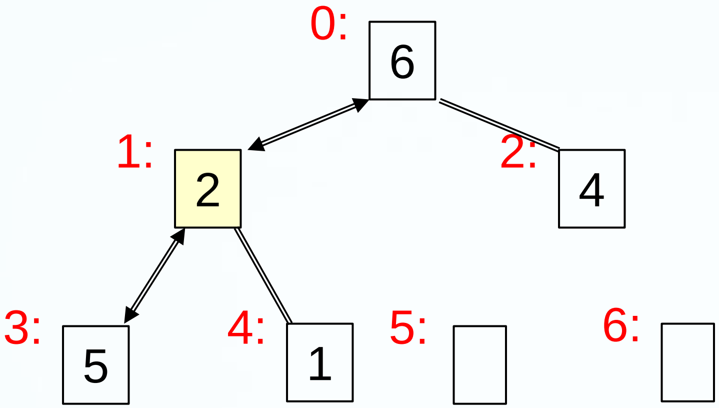
[2, 6, 4, 5, 1]

➡ [6, 2, 4, 5, 1]

- Take the root and plug the hole with the “last leaf”
- Restoring the heap property – recursively:
  - Compare inserted node x with its (up to) 2 children y and z
  - If x ≥ y and x ≥ z, or x is a leaf, then stop
  - Otherwise swap x and the larger of its children
  - Repeat recursively



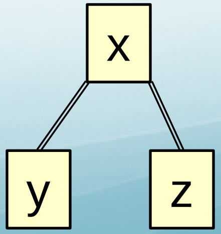
x ≥ y
x ≥ z



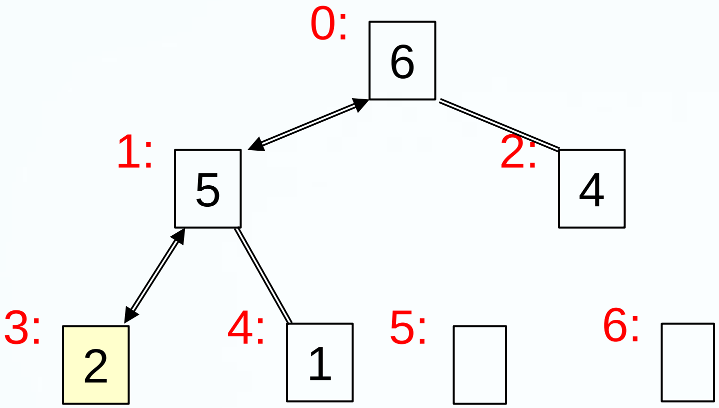
Tree encoding:
[9, 6, 4, 5, 1, 2]
➡ [2, 6, 4, 5, 1]

Steps:
[2, 6, 4, 5, 1]
➡ [6, 2, 4, 5, 1]
➡ [6, 5, 4, 2, 1]

- Take the root and plug the hole with the "last leaf"
- Restoring the heap property – recursively:
  - Compare inserted node x with its (up to) 2 children y and z
  - If x ≥ y and x ≥ z, or x is a leaf, then stop
  - Otherwise swap x and the larger of its children
  - Repeat recursively



x ≥ y
x ≥ z



Tree encoding:

[9, 6, 4, 5, 1, 2]

→ [2, 6, 4, 5, 1]

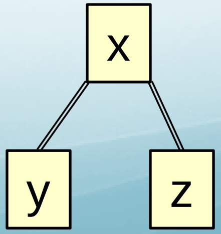
Steps:

[2, 6, 4, 5, 1]

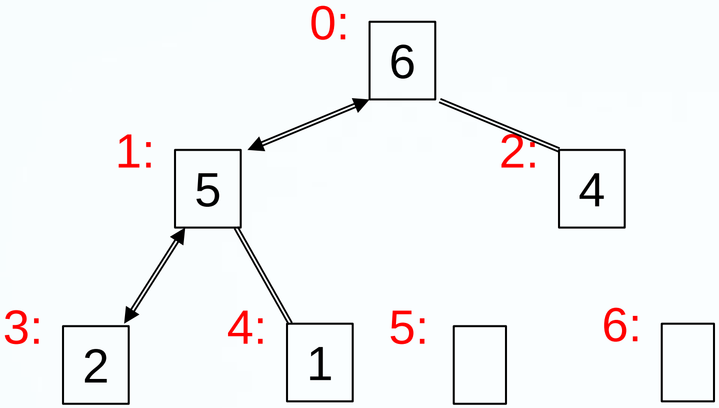
→ [6, 2, 4, 5, 1]

→ [6, 5, 4, 2, 1]

- Take the root and plug the hole with the “last leaf”
- Restoring the heap property – recursively:
  - Compare inserted node x with its (up to) 2 children y and z
  - If x ≥ y and x ≥ z, or x is a leaf, then stop
  - Otherwise swap x and the larger of its children
  - Repeat recursively



x ≥ y
x ≥ z



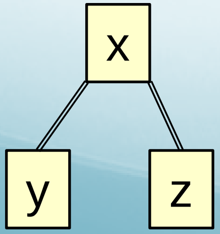
Steps:

[9, 6, 4, 5, 1, 2]

➡ [2, 6, 4, 5, 1]

➡ [6, 2, 4, 5, 1]

➡ [6, 5, 4, 2, 1]



$x \geq y$
$x \geq z$

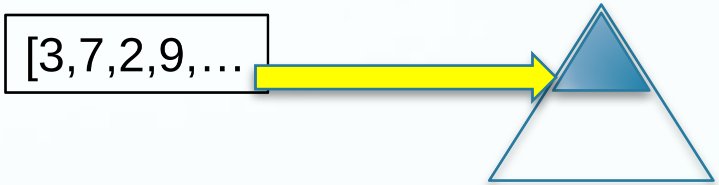
```
def queueDelete(H):
    if len(H) == 0: return  # queue is empty
    max = H[0]
    H[0] = H[-1]
    del H[-1]
    k, n = 0, len(H)
    while k < n:
        kL, kR = 2*k+1, 2*k+2
        if kL >= n:     # at a leaf
            return max
        if kR >= n:     # only left child exists
            if H[kL] > H[k]:
                H[k], H[kL] = H[kL], H[k]
                k = kL  # swap -- continue the loop
            else: return max   # one child and done
        # both children exist, identify the larger one
        if H[kL] > H[kR]:
            kM = kL
        else:
            kM = kR
            if H[k] >= H[kM]:
                return max
        H[k], H[kM] = H[kM], H[k]
        k = kM
    # end of while loop
    return max
```

Application: Sorting

- Motivation for sorting data:
  - We can answer questions like min/max very efficiently
  - We can search very efficiently
    - What if we need to search many many times
  - Many algorithms require their *input* to be sorted
- Heap sort uses the heap-based priority queue to implement a sorting algorithm efficiently

Recall the Sorting Phases

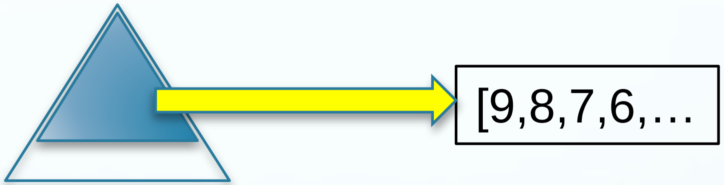
1. Put all items in the list into a priority queue



[3,7,2,9,…

2. Extract items by decreasing magnitude

We get a sorted list that way

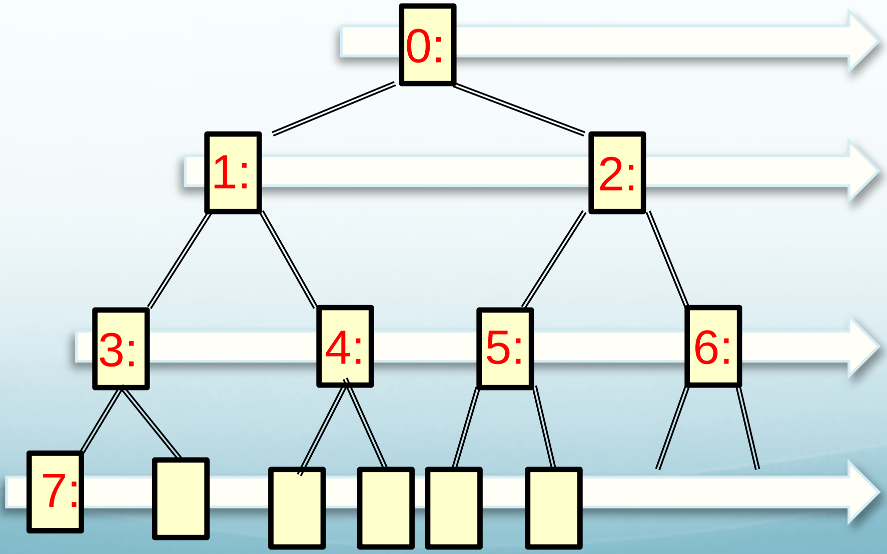


[9,8,7,6,…

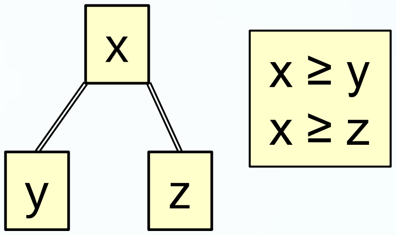
But we still need to enter items into the queue!

Priority Queue Insertion

- The only place to expand the queue without destroying the tree encoding is at the end: at the next available slot in the last layer
  - If the layer is full, this starts the next layer
- This step is automatic if we just append to the list

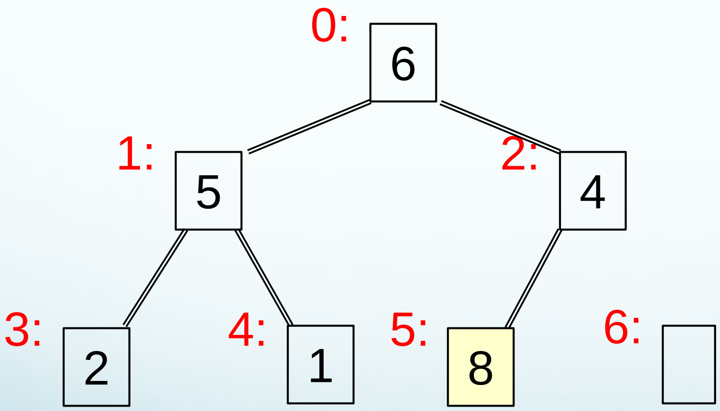


Heap Insertion

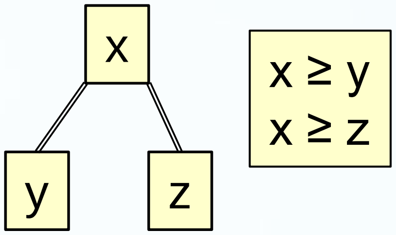


x ≥ y
x ≥ z

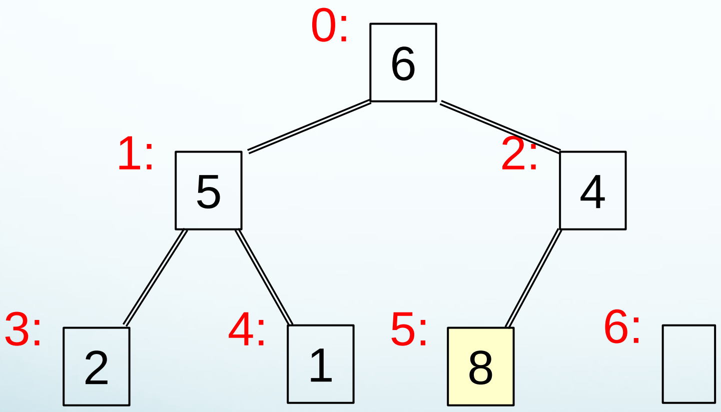
- What about the heap property?
  - It has to be maintained and restored if necessary!

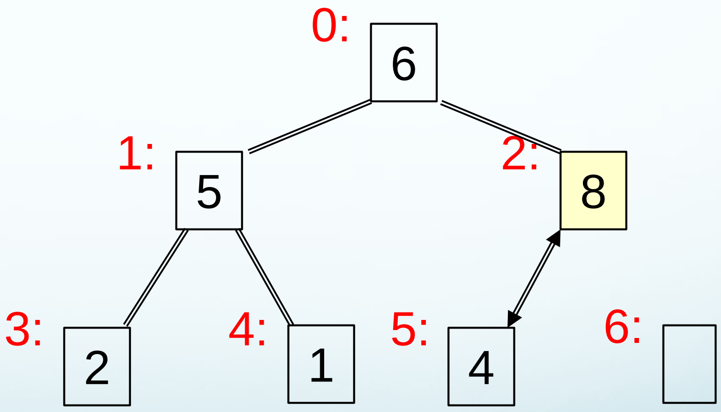


Heap Insertion

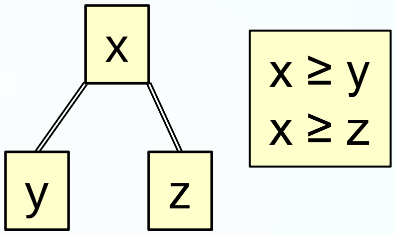


- Restoring heap property
  - Compare new insertion with parent:
    - If parent is greater, we are finished
    - If parent is smaller, exchange

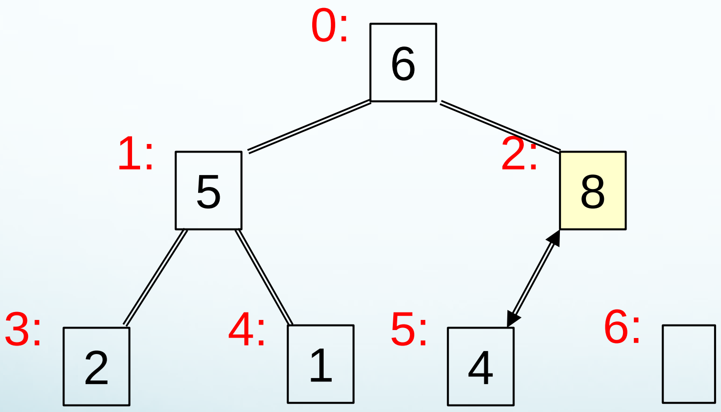


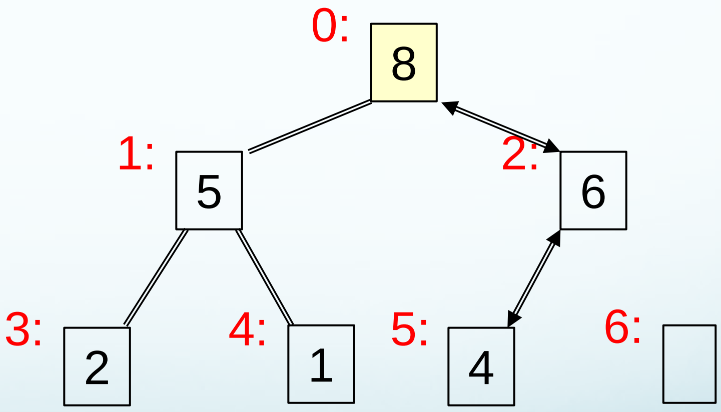


Heap Insertion



- Restoring heap property
  - Compare new insertion with parent:
    - If parent is greater, we are finished
    - If parent is smaller, exchange

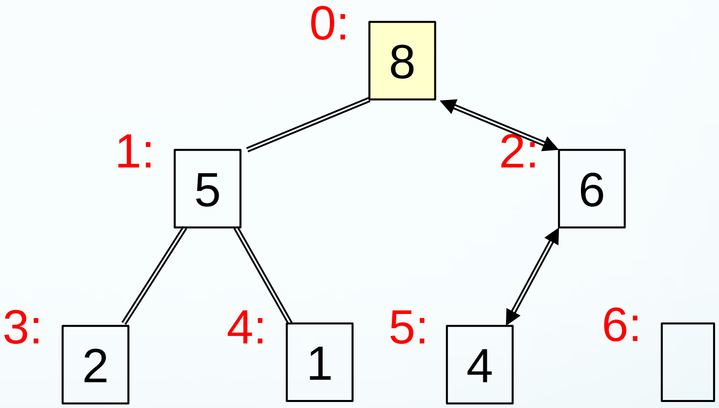




Heap Insertion

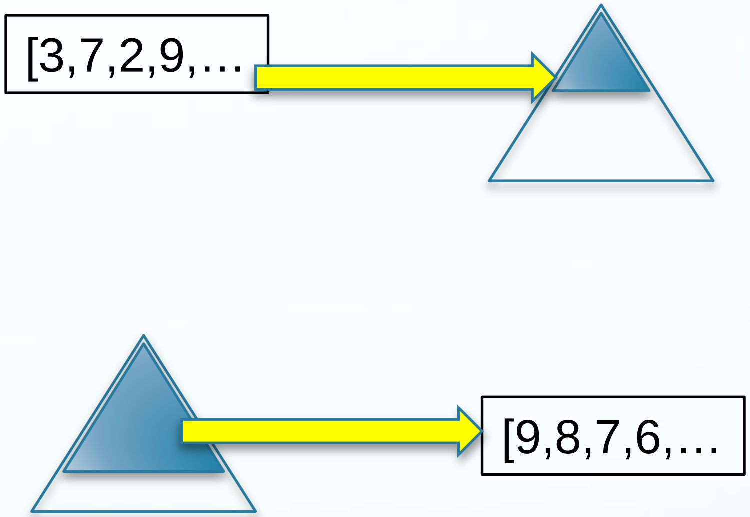
- Resulting code

```
def queueInsert(H, x):
    H.append(x)
    k = len(H)-1
    kP = (k-1)//2
    while k > 0:
        kP = (k-1)//2
        if H[k] <= H[kP]:
            return
        H[k],H[kP] = H[kP],H[k]
        k = kP
    return
```



Putting it all together: Heapsort

```
def heapSort(L):
    L1 = []
    for x in L:
        queueInsert(L1,x)
    L2 = []
    while len(L1) != 0:
        x = queueDelete(L1)
        L2.append(x)
    return L2
```



[3,7,2,9,…

[9,8,7,6,…

Complexity: Swap Path

- Deletion:
  - Take out L[0], paste in L[-1]
  - Compare new L[0] with its children, swap if needed
  - Every swap is a step on a path from tree root to leaf
  - Complexity proportional to tree height h
- Insertion
  - Add at list end
  - Compare with parent, swap if necessary
  - Every swap is a step on the path from leaf to root
  - Complexity proportional to tree height h

Complexity: Tree Stats

- With n the number of items in the list, how high is the tree?

n	height
1:	1
2-3:	2
4-7:	3
8-15:	4
2^k-(2^{k+1}-1):	k+1

- So if the list has n elements, it encodes a tree of height $O(\log(n))$
- Therefore, insertion and deletion of one element takes $O(\log(n))$ steps

Heapsort Complexity

- Two phases, each doing n-times $O(\log(n))$ work
- Total work therefore $O(n \log(n))$

Can we sort faster?

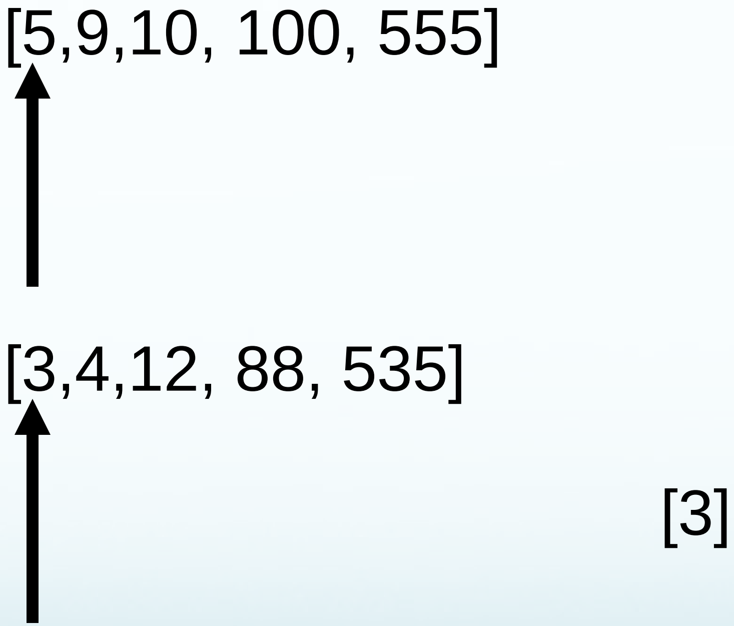
- No:
  - In general, sorting in $O(n \log n)$ time is the best we can do
- But heap sort is not the only $O(n \log n)$ time sorting algorithm
- Consider merge sort

Merge Sort

Observation 1: We can merge two sorted lists in linear time

- What is in the input?
  - Both the lists, n = total amount of elements
- Why is the complexity linear?
  - We must examine each element in each of the lists
  - Its linear in the total amount of elements
    - O(len(list1) + len(list2)) = O(n)

Observation 1: We can merge two sorted lists in linear time

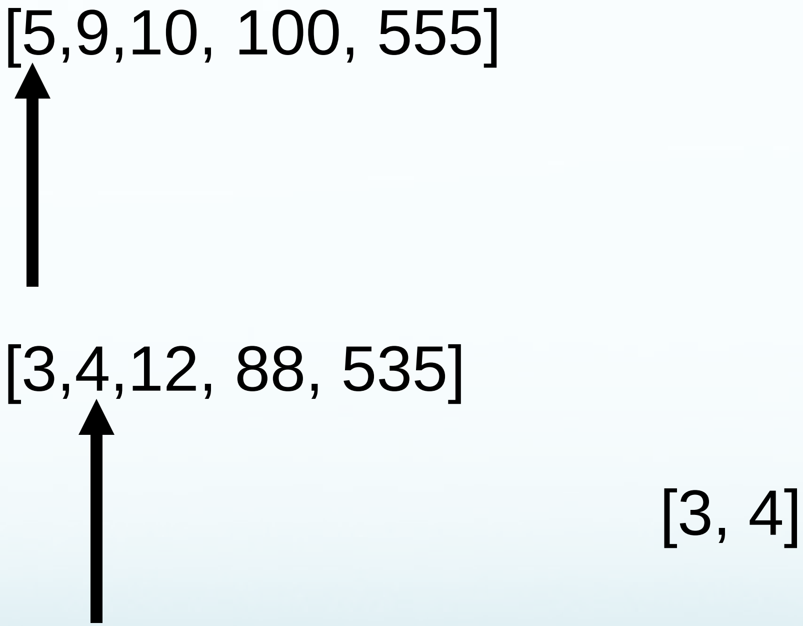


[5,9,10, 100, 555]

[3,4,12, 88, 535]

[3]

Observation 1: We can merge two sorted lists in linear time

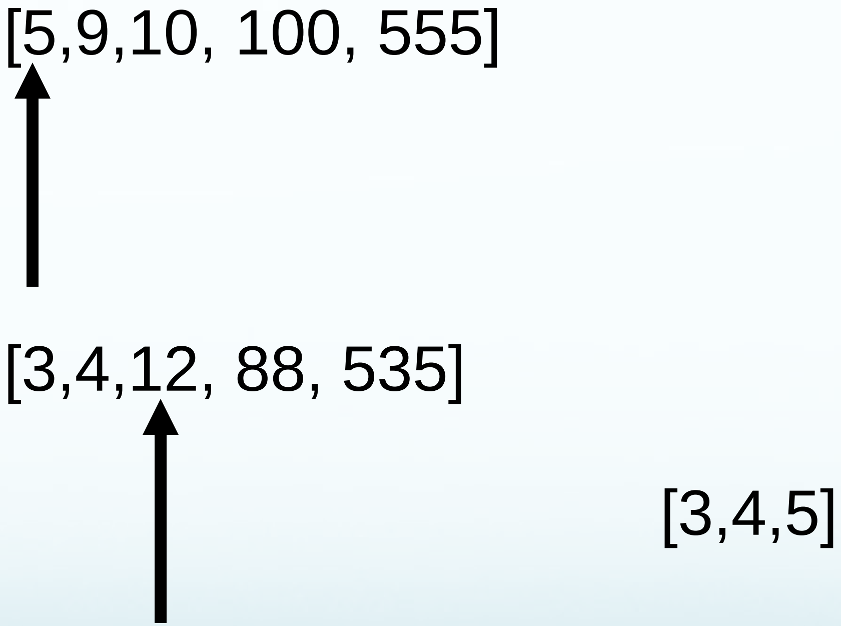


[5,9,10, 100, 555]

[3,4,12, 88, 535]

[3, 4]

Observation 1: We can merge two sorted lists in linear time

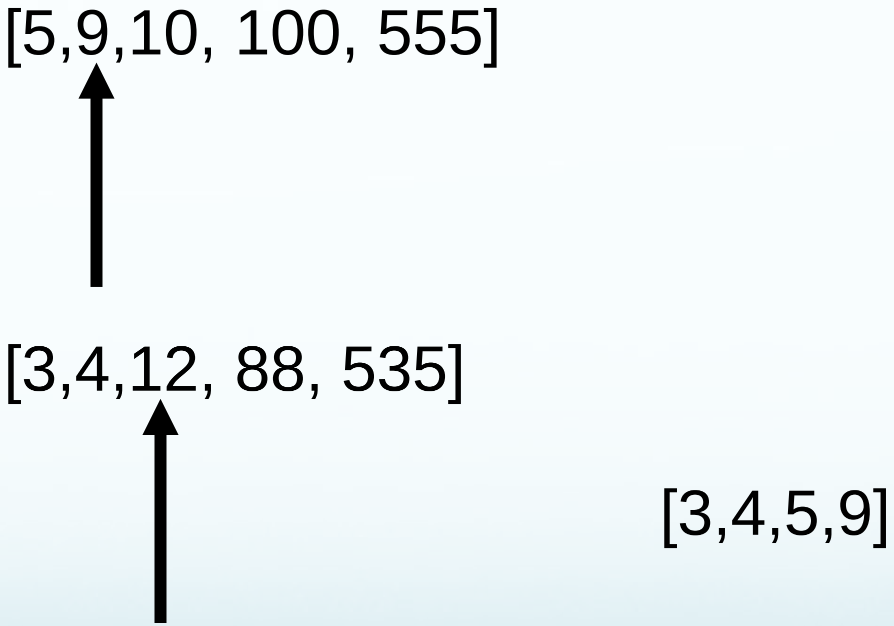


[5,9,10, 100, 555]

[3,4,12, 88, 535]

[3,4,5]

Observation 1: We can merge two sorted lists in linear time

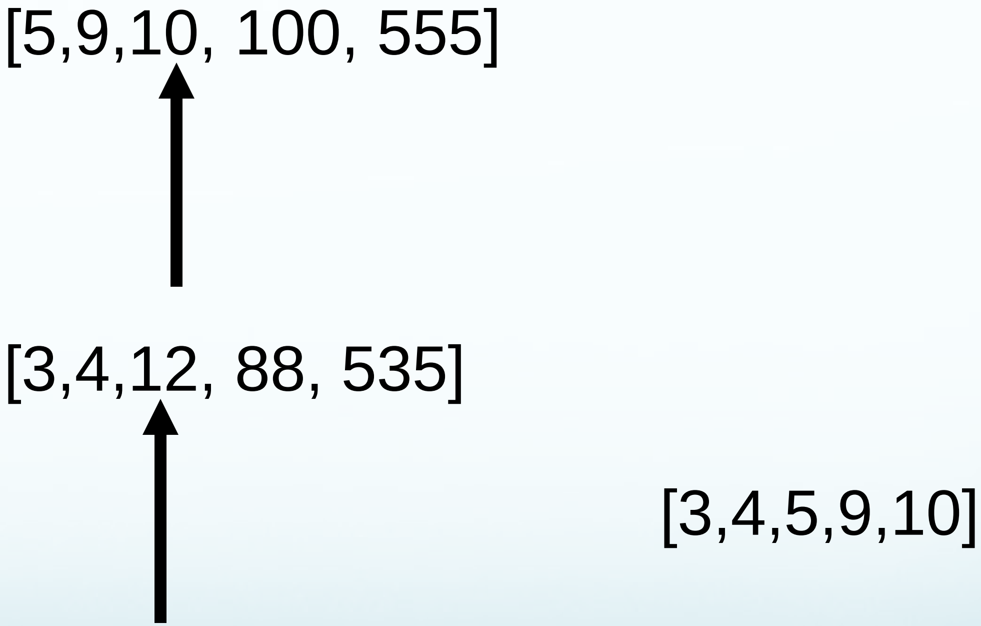


[5,9,10, 100, 555]

[3,4,12, 88, 535]

[3,4,5,9]

Observation 1: We can merge two sorted lists in linear time

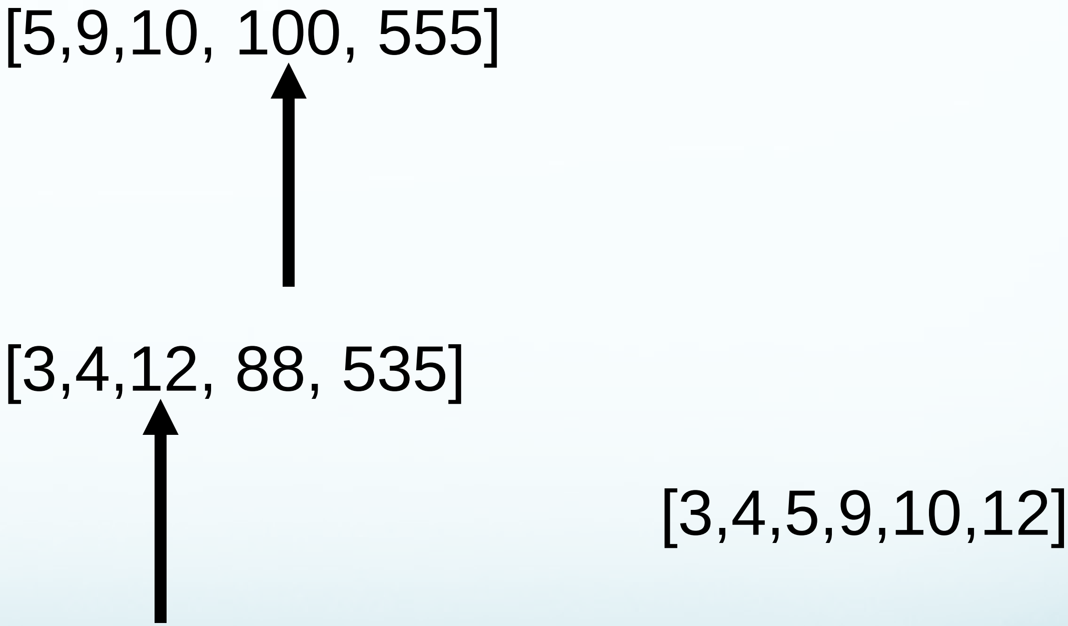


[5,9,10, 100, 555]

[3,4,12, 88, 535]

[3,4,5,9,10]

Observation 1: We can merge two sorted lists in linear time

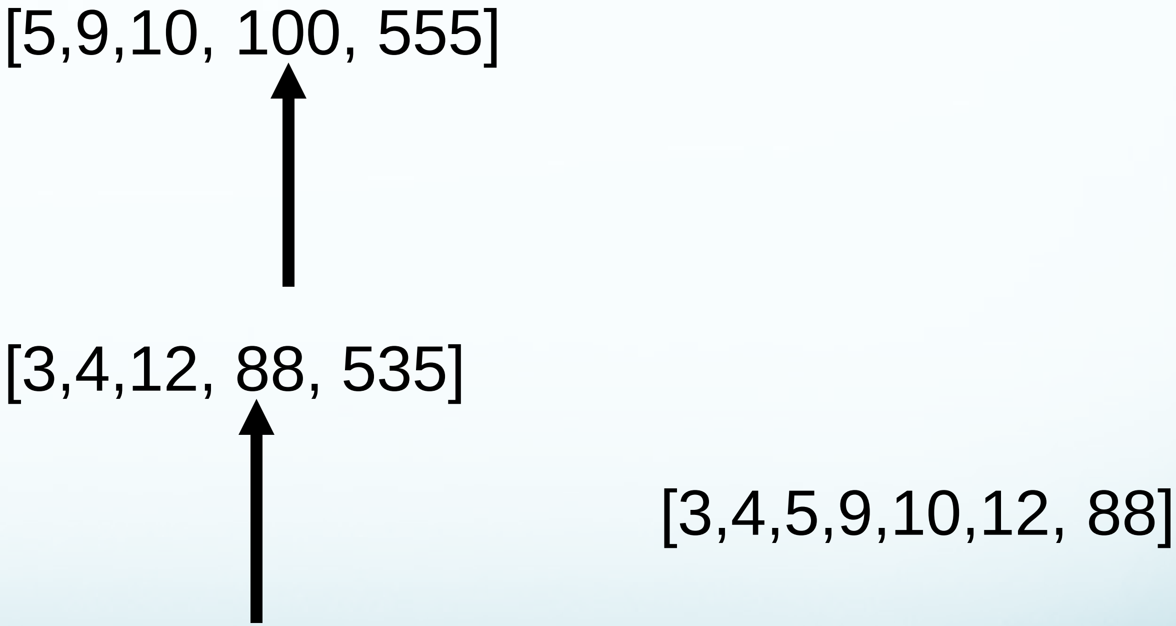


[5,9,10, 100, 555]

[3,4,12, 88, 535]

[3,4,5,9,10,12]

Observation 1: We can merge two sorted lists in linear time



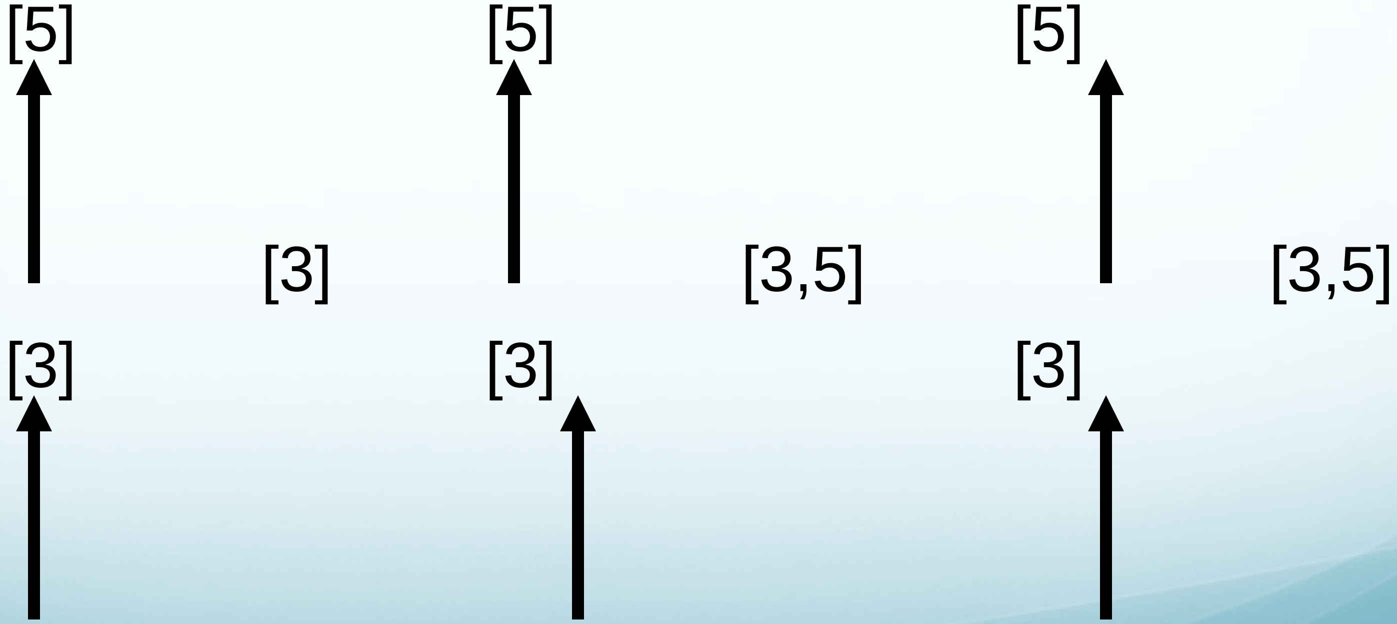
[5,9,10, 100, 555]

[3,4,12, 88, 535]

[3,4,5,9,10,12, 88]

Observation 2

- Notice that merging two lists of length one ends up producing a sorted list of length two



[5] [5] [5]

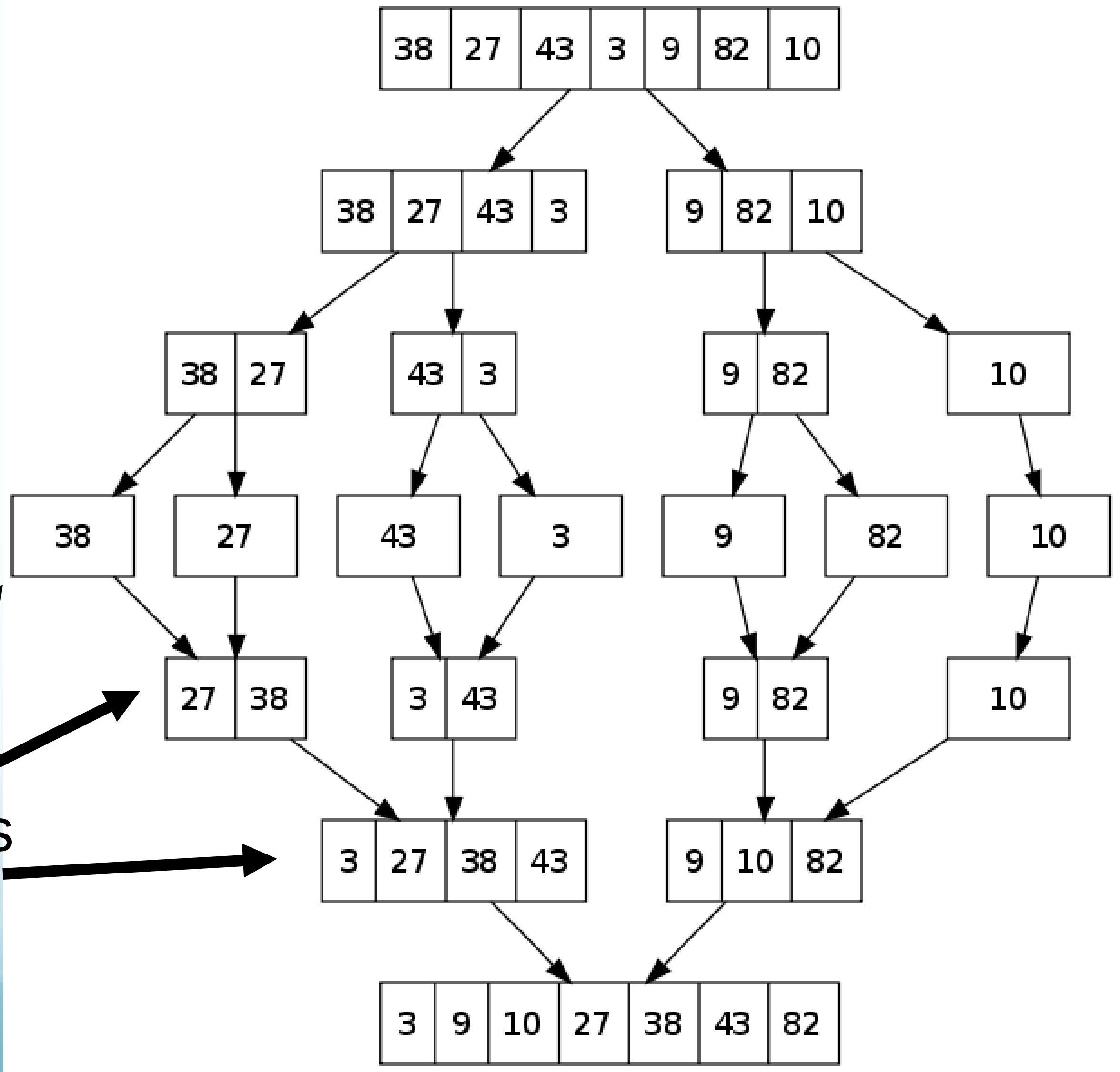
[3] [3,5] [3,5]

[3] [3] [3]

Lets build the intuition for Merge-Sort

- We know we can merge sorted lists in linear time
- We know that merging two lists of length one results in a sorted list of length two
- Lets split our unsorted list into a bunch of lists of length one and merge them into progressively bigger lists!
  - We split a list into two smaller lists of equal parts
  - Keep splitting until we have lists of length one
  - Lists of length 1 are already sorted

Visual Representation



n elements merged

log(n)

Putting it all together

- We know that there are log(n) splits
  - At each “level” we split each list in two
- We know that we need to merge a total of n elements at each “level”
  - n * log(n)   thus O(n log n)