Heap and PQ

Class 9

Performance with n Pairs

- a PQ always has n pushes and n pops

	sorted vect	sorted list	unsort vect	unsort list
push	$O(n), \Omega(\lg n)$	$O(n), \Omega(1)$	$\Theta(1)$	$\Theta(1)$
pop	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
n pairs	$O(n^2), \Omega(n \lg n)$	$O(n^2), \Omega(n)$	$\Theta(n^2)$	$\Theta(n^2)$

- which implementation is best?
- sorted list, then sorted vector, then the unsorted versions tie for last
- however, we can do better than any of these if instead we use a heap

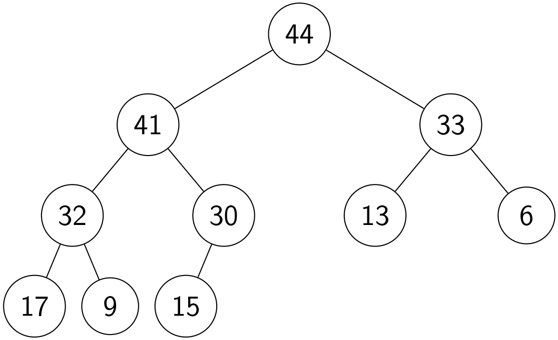
Heap

- a digression to a new data structure
- heap is a very unfortunate name
- heap has two completely different meanings in CS
  1. the area of memory from which memory for dynamically allocated variables is obtained
  2. a specific data structure (in our case, we will assume a binary heap when we say heap)

Heap Data Structure

Heap

A complete binary tree in which the root is either empty or contains a data element of higher value than both children, and both children are recursively heaps.

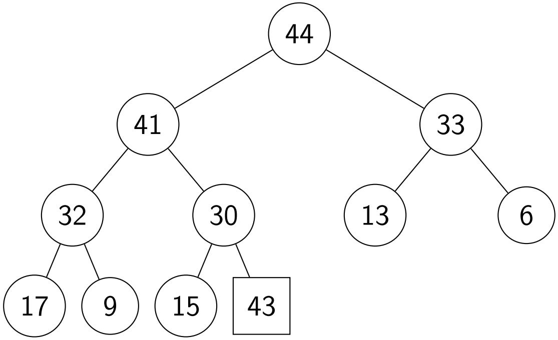


Heap Operations

- a heap has two main operations
  1. push
  2. pop

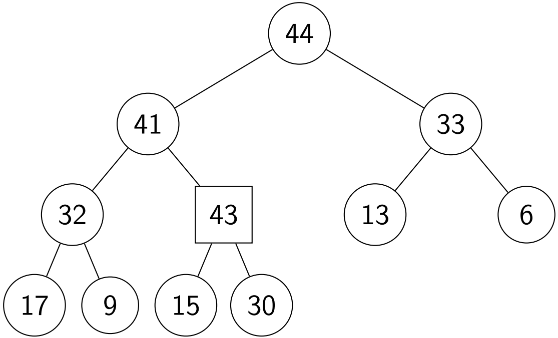
Heap Push

- since the heap must be complete, the inserted item must go to the right (sibling or cousin) of the last element
- or, if the bottom row is filled, open a new row
- insert 43



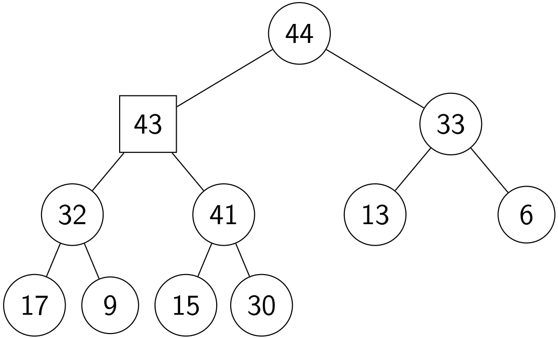
Heap Push

- then, bubble up



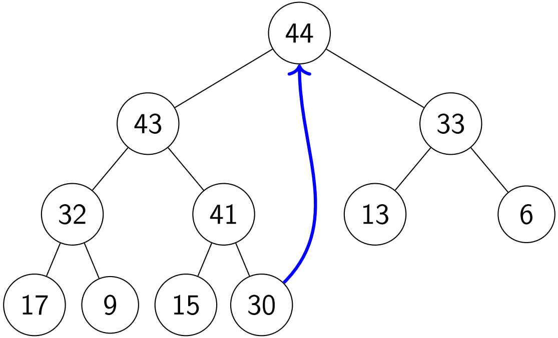
Heap Push Final Arrangement

- bubble up until done



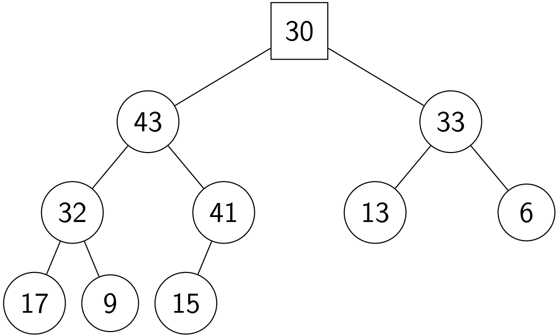
Pop

- clearly the delete(_max) operation returns the root element
- to maintain heap structure, we must get rid of the rightmost element on the last row
- we copy the rightmost last row element to the root



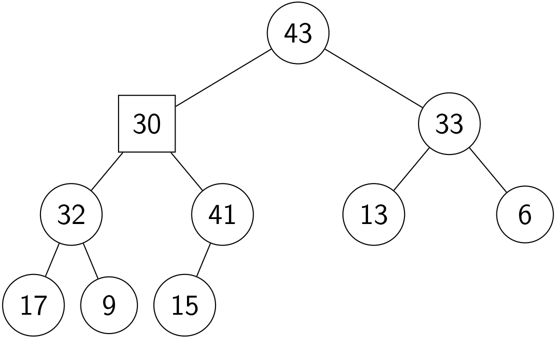
Pop

- then call percolate down



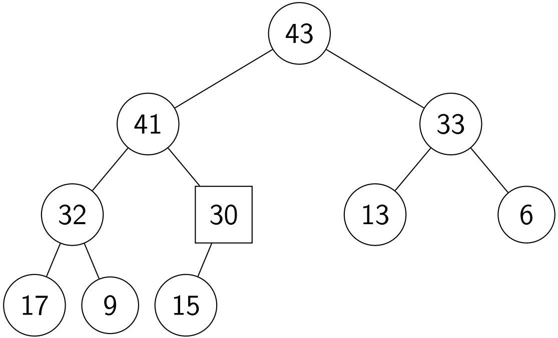
Pop

- continue percolate down



Pop

- continue percolate down



Heap vs PQ

- we have been discussing the heap data structure
- but clearly, a heap can be used to implement a PQ
- analysis of the heap

	heap
push	
pop	
n pairs	

Heap vs PQ

- we have been discussing the heap data structure
- but clearly, a heap can be used to implement a PQ
- analysis of the heap

	heap
push	$O(\lg n), \Omega(1)$
pop	$O(\lg n), \Omega(1)$
n pairs	$O(n \lg n), \Omega(n)$

- a huge improvement over $O(n^2)$ for a sorted list or vector

Implementation

- but this heap thing is just an ADT
- it's an idea, a pretty abstraction

Implementation

- but this heap thing is just an ADT
- it's an idea, a pretty abstraction
- but how do we implement it in code?
- what physical data structure do we use?

Implementation

- but this heap thing is just an ADT
- it's an idea, a pretty abstraction
- but how do we implement it in code?
- what physical data structure do we use?

- a heap is a tree
- we will talk about trees in general later
- we normally implement trees with nodes and pointers to nodes

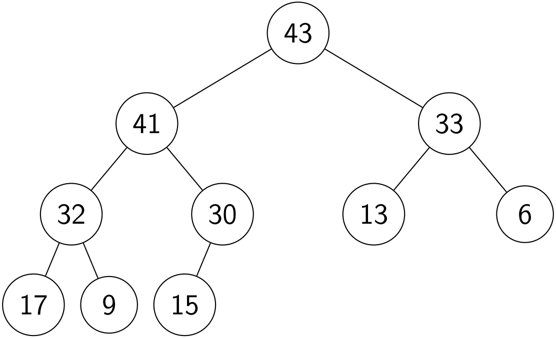
Implementation

- but this heap thing is just an ADT
- it's an idea, a pretty abstraction
- but how do we implement it in code?
- what physical data structure do we use?

- a heap is a tree
- we will talk about trees in general later
- we normally implement trees with nodes and pointers to nodes

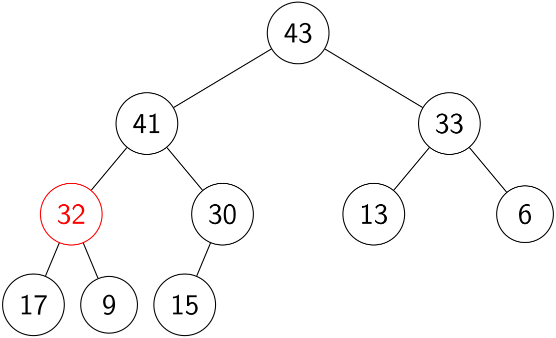
- we could implement a heap this way
- but instead, amazingly, we can implement it on a vector

Heap Implementation



43	41	33	32	30	13	6	17	9	15
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Who's Your Daddy?



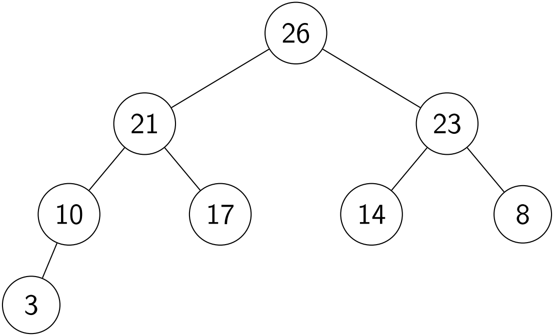
43	41	33	32	30	13	6	17	9	15
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Navigating

- for any element i
  - i's parent is $\frac{i-1}{2}$
  - i's children are
    - $2i+1$
    - $2i+2$

PQ Example

- starting with the heap below, do the following operations
- show the results
  - schematically for a PQ
  - physically, including the indexes considered
- push (24); push (19); pop();



Building a Heap

- suppose we have n elements
- we wish to build a heap with them

- one at a time, n times, we do a push
- analyze this

Building a Heap

- suppose we have n elements
- we wish to build a heap with them

- one at a time, n times, we do a push
- analyze this

- each push takes $O(\lg n), \Omega(1)$ operations
- there are n pushes
- for a total of $O(n \lg n), \Omega(n)$ operations

Heapify

- we can do better

```
void heapify(array)
{
  size_t size = array.size();
  for (size_t index = (size / 2) - 1; index < size; index--)
  {
    percolate_down(index);
  }
}
```

- analyze this

Heapify

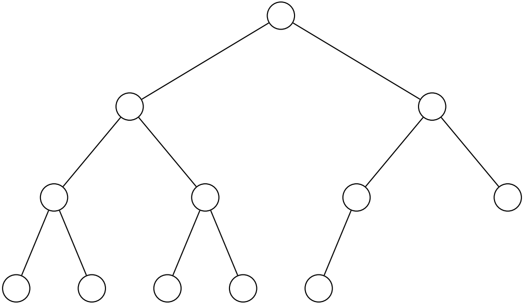
- we can do better

```
void heapify(array)
{
  size_t size = array.size();
  for (size_t index = (size / 2) - 1; index < size; index--)
  {
    percolate_down(index);
  }
}
```

- analyze this
  - a for loop ($\Theta(n)$) containing
  - percolate_down ($O(\lg n), \Omega(1)$)
- for an overall analysis of $O(n \lg n), \Omega(n)$ BUT! ...

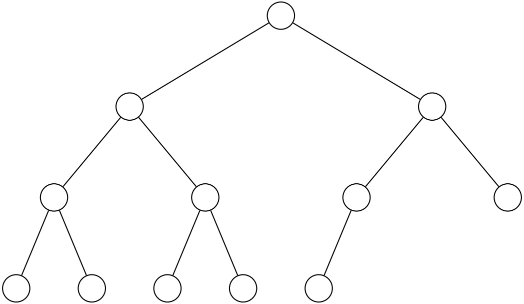
Heapify Analysis

- how many leaf nodes are in a heap (for $n > 2$)?



Heapify Analysis

- how many leaf nodes are in a heap (for $n > 2$)?



$$\left\lceil \frac{n}{2} \right\rceil$$

Analysis of Heapify

- in a full binary tree there are $n = 2^h$ nodes
- half of them are leaves, half are interior nodes
- we count comparisons
- the actual number of comparisons in heapify is

$$\begin{aligned} &2\frac{n}{2} \text{ for the level above the leaves} \\ &+2\frac{n}{4} \text{ for the level above that} \\ &+2\frac{n}{8} \text{ for the level above that} \\ &\vdots \end{aligned}$$

- for $h-1$ levels

$$\sum_{i=1}^{h-1} \frac{2n}{2^i} \in \Theta(n)$$

A Digression for an Application

- once we perform heapify, what do we know about the element in vector position 0?

26	21	23	10	17	14	8	3
0	1	2	3	4	5	6	7

An Application

- swap positions 0 and $n - 1$

26	21	23	10	17	14	8	3

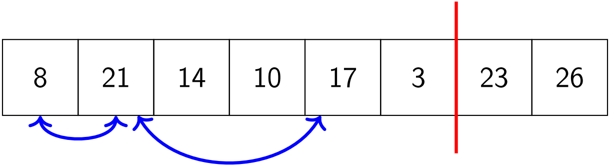
An Application

- "wall off" position $n - 1$
- call percolate down on 0

3	21	23	10	17	14	8	26

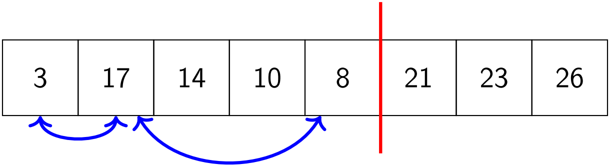
An Application

- swap positions 0 and $n-2$
- "wall off" position $n-2$
- call percolate down on 0



An Application

- swap positions 0 and $n - 3$
- "wall off" position $n - 3$
- call percolate down on 0



Heapsort

```
heapify();
for (size_t i = n - 1; i != 0; i--)
{
  swap(array.at(0), array.at(i))
  pretend the array is one smaller
  percolate_down(0)
}
```

Heapsort Analysis

- heapify is $O(n)$ (from earlier)
- a $\Theta(n)$ loop containing:
  - percolate_down, which is $O(\lg n)$

$$\begin{aligned} T(n) &\leq n + n \lg n \\ &\in O(n \lg n) \end{aligned}$$

- what is big-Omega?