

Guest lecture: Sarah Matzko

I heap

II binary heap

III Binomial queues

IV fibonacci heap

— main operations:

insert — enqueue

deletemin — dequeue

What is a heap — heap order

— smallest on top

— parents < child
(node key values)

— AKA priority queue

↓ (FIFO list)

in order of priority

priority queue = in order of priority,
not FIFO

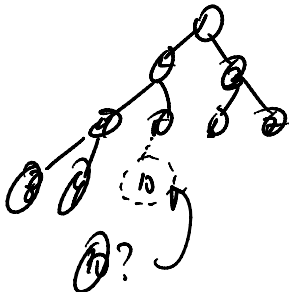
←

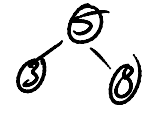
Binary heap

- not much harder than binary trees
- structure: binary tree (but not binary search tree)


complete binary tree

top-to-bottom,
left to right insertion
order

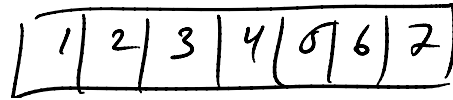
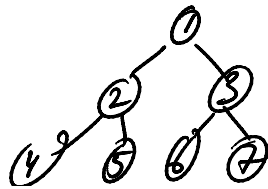


if  } order matters.
in BST

not in binary heap

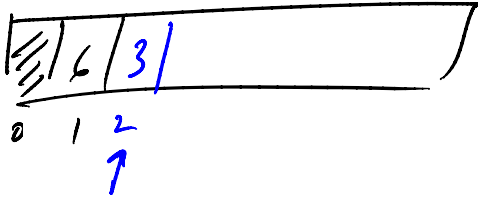
 except in
in min heap
min is at top

- can represent binary heaps (complete binary trees) by arrays (logical structure)



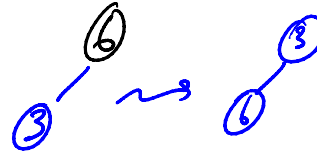
- heap order (obvious property)

- insert & determine in binary heap
- insert 1 6 3 1 4 7 2 5 8



violates heap order

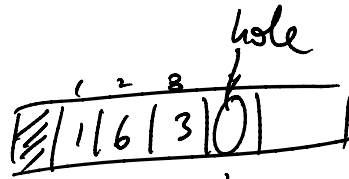
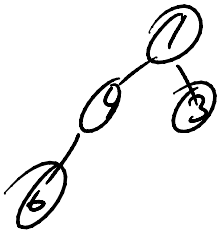
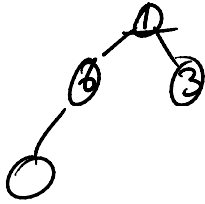
- need to percolate up:
 - a) compare node to parent ($\text{hole}/2$)
 - b) if node < parent, swap
 - c) update "hole" to parent's location



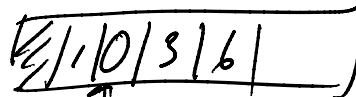
percolate up: check parent
if (parent < child)
swap elements

how do we find parent
at index x ?
 $x/2$

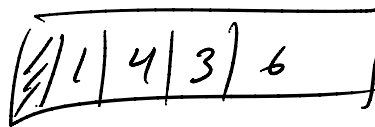
6 3 1 4 7 2 5 8

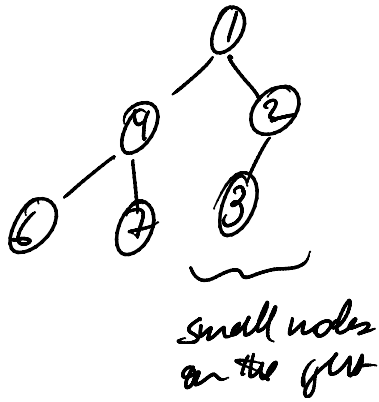


percolate down to pos 2



percolate hole to pos 1
no swap, instead keep





- looks funny, but valid
in terms of heap order
(parents are smaller than
children in all cases)

delete Min:

1. get the minimum value (root)
2. last element is now 'holeless'
3. hole at root

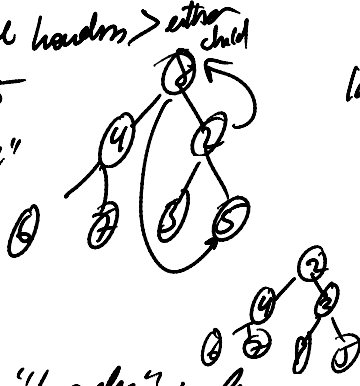
4. while $hander > \text{either child}$

a. promote 5

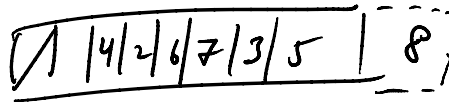
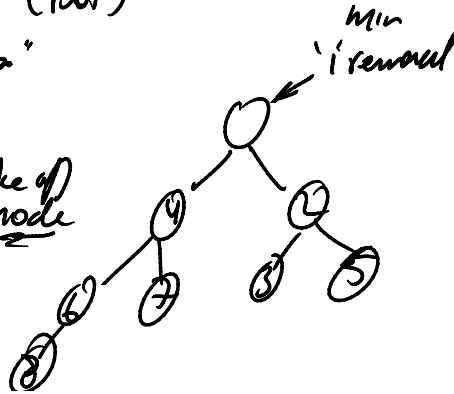
b. move "hole"
to smaller
child's
index

index

5. place "holeless" value



take of
last node



put in
for (root) pos

- note: children of element x : $2x$, $2x+1$
- pendents p : move hole from end of list
- promote down: move hole from root (front of list) down the list (tree)

- let's start:

init hole = 1;

init smaller = index of smaller child

hole = smaller

- when do we stop? we size of array
(e.g. vector, size())

Note: half the
tree nodes are
leaves —

stop when

hole 'hole' \geq size/2

(we're at a leaf)

Performance

insert:

average times:
only pendents
up 1 level
 $\approx O(1)$
[amortized]
 $\Theta(1)$
not worst
case $O(\lg n)$

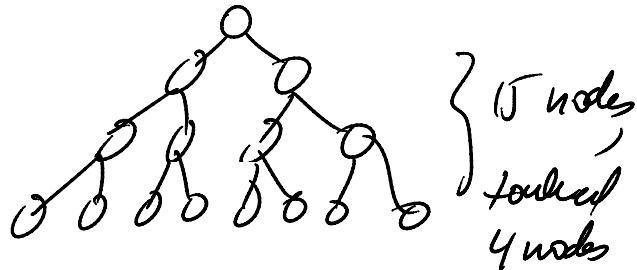
deleteMin: may have to percolate down
to leaf — what's the
height of the tree?

$$2^4 = 16$$

$$\log_2 16 = 4$$

for a tree with
 n elements,

$$\text{height} \approx \log_2 n \Rightarrow \text{deleteMin} \in O(\lg n)$$



Binomial queue — also a heap

— performance is better than binary heap

$\lg N$ insert, delete min, merge



— binomial queue: a forest
a list of trees.

not good in bin heap

— binomial tree in the forest has exactly 2^{height} nodes

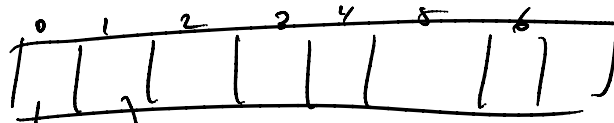
— easiest way to tell a tree's height is to count its children (tree can have any # children)

— only 1 tree of each height is in the queue

— each array slot has tree of height = to its index

Grandma's game:

5 5 1 2 3 4 2 6



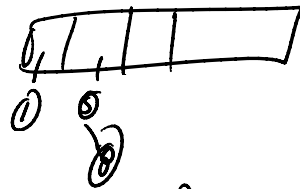
- insert 5:
merge 5

result:

5 has 1 child
⇒ put in spot 1



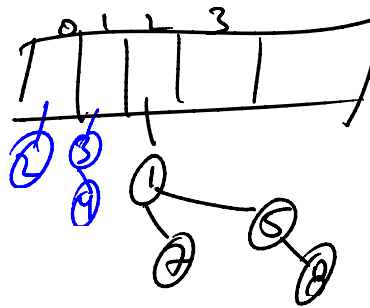
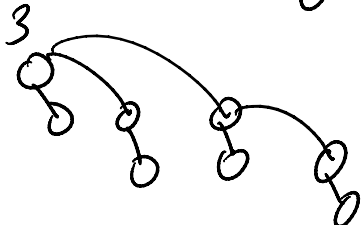
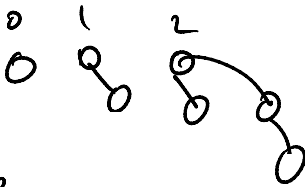
insert 1: ops to spot 0



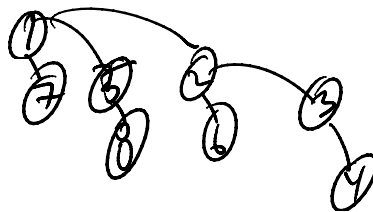
insert 7: merge with 5

note: $2^x + 2^x = 2^{x+1}$
 $= 2(2^x) = 2^{x+1} \checkmark$

in general:

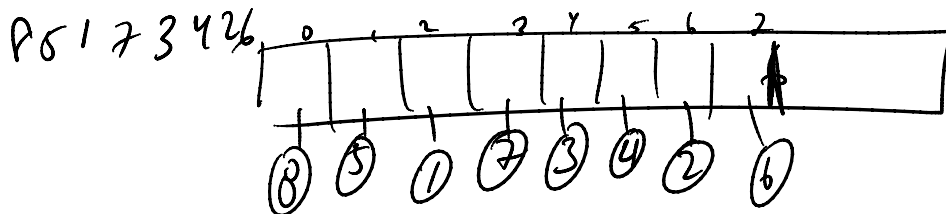


insert 6:



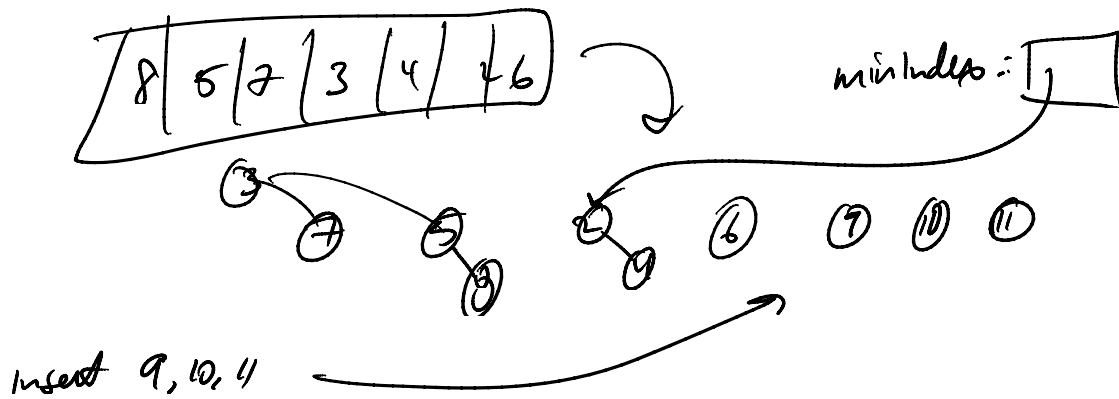
- Fibonacci heap:

- improves on binomial queue insertion
- performance improvement comes from lazy insertions & deletions
- insert is always $O(1)$
- deletion doesn't destroy cause disturbing of the tree
- only organized to be a binomial queue when new min is needed (i.e. during deletion)

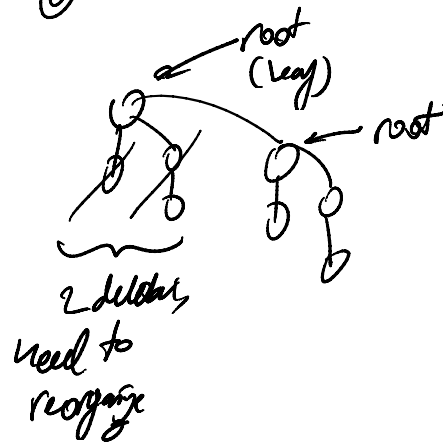
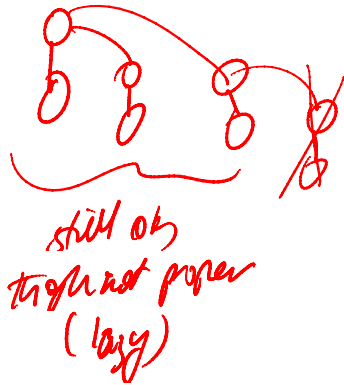
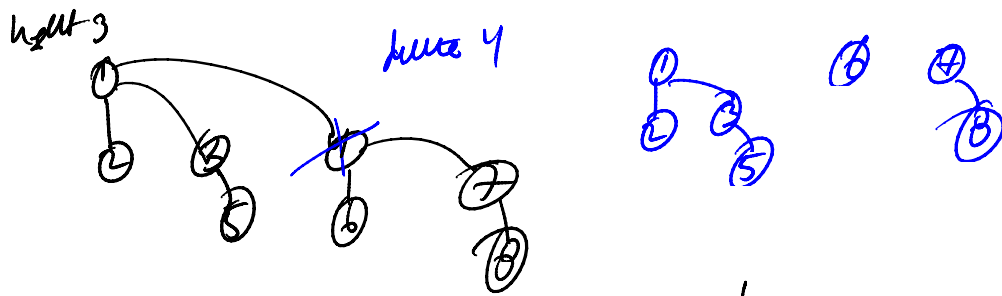


how to
keep track of
min?
minIndex = 2

so far so good, but: when delete min
delete 0, how we search through list?
No, recognize tree: merge trees



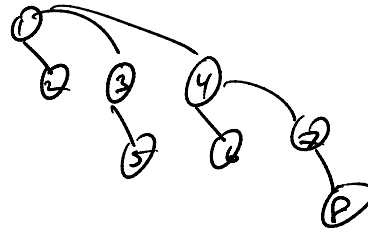
delete min performance: worst case: $O(n)$ (touch every single node)
if tree is balanced tree structure,
 $O(\lg n)$ [amortized?]



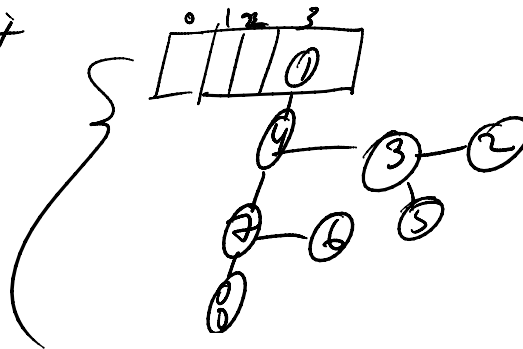
- how to represent?

fibnode:

- data
- first child link
- next sibling link
- height
- boolean: lost_a_child (used to determine whether to disband tree)
- parent link (like a backpointer)



a linked list of children basically



why is it called Fibonacci heap?

given height

3	=	3
4	=	5
5	=	8
6	=	13
7	=	21

deleting nodes w/out disbanding.