Analysis of Algorithms

V. Adamchik

Lecture 3

Fall 2023

CSCI 570

University of Southern California

Heaps

Reading: chapter 3

Amortized Analysis

In a <u>sequence</u> of operations the worst case does not necessarily occur in each operation - some operations may take different times.

Therefore, a traditional worst-case per operation analysis can give overly *pessimistic* bound.

Consider insertions into an array

some operations take $O(n)$, others – $O(1)$

if the current array is full, the cost of insertion is linear;
if it is not full, insertion takes a constant time.

Therefore, amortized analysis is an alternative to the traditional worst-case analysis. Namely, we perform a worst-case analysis on a sequence of operations.

The Aggregate Method

The amortized cost of an operation is given by $\frac{T(n)}{n}$, where $T(n)$ is the upper bound on the total cost of n operations.

Example: unbounded array (with a doubling-up resizing policy)

Insertions:	1, 2, 3, 4, 5, 6, 7, 8, 9, …, 2^n+1
Insertion Cost:	1, 1, 1, 1, 1, 1, 1, 1, 1, …, 1
Copy Cost:	0, 1, 2, 0, 4, 0, 0, 0, 8, … , 2^n

In lecture 2 we computed the average cost per insert: O(1)

It is important to realize that we achieve a great amortized cost just because we have implemented a clever resizing policy!

Review Questions

2. (T/F) Amortized analysis is used to determine the average runtime complexity of an algorithm.
3. (T/F) Compared to the worst-case analysis, amortized analysis provides a more accurate upper bound on the performance of an algorithm.
4. (T/F) The total amortized cost of a sequence of n operations gives a lower bound on the total actual cost of the sequence.
5. (T/F) Amortized constant time for a dynamic array is still guaranteed if we increase the array size by 5%.
6. (T/F) If an operation takes $O(1)$ expected time, then it takes $O(1)$ amortized time.
7. Suppose you have a data structure such that a sequence of n operations has an amortized cost of $O(n \log n)$. What could be the highest actual time of a single operation?

$O(1), O(1), \ldots, O(1), O(n^2) \rightarrow \text{Total} = O(n^2)$

$O(1), O(1), \ldots, O(1), O(n \log n)$

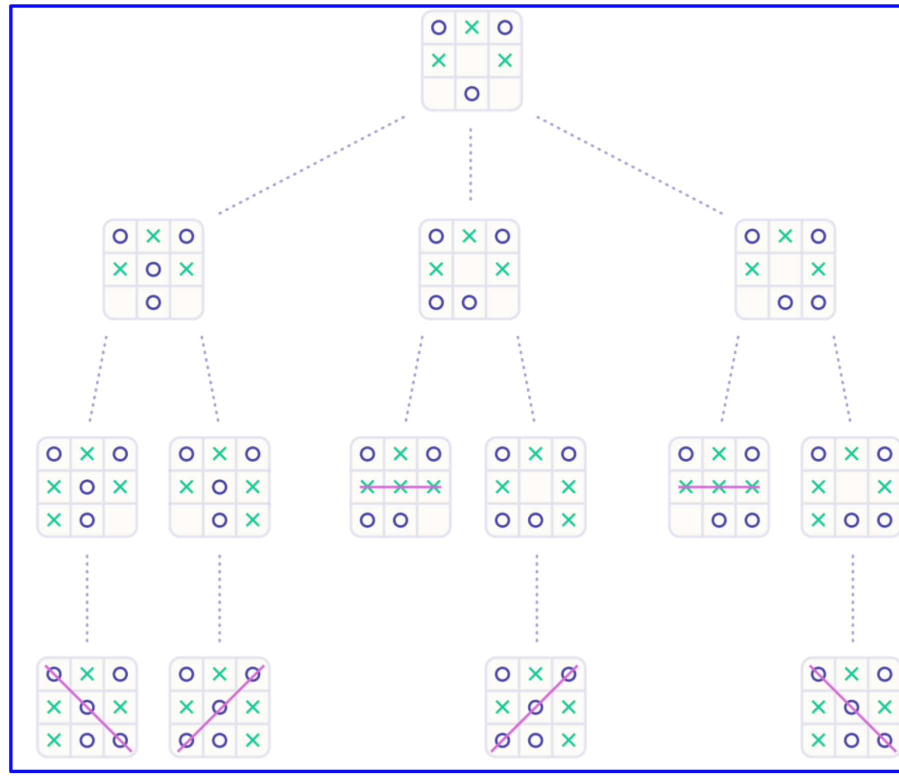
Ans: $O(n \log n)$

Heap and Priority Queue for Solving Optimization Problems

In this lecture, we will discuss a data structure that allows us to quickly access the highest priority element.

1 sort

2 Heap



To win a game you cannot simply run a DFS/BFS, among all possible moves you have to choose the best move!

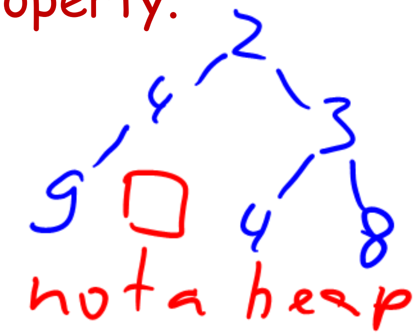
BST

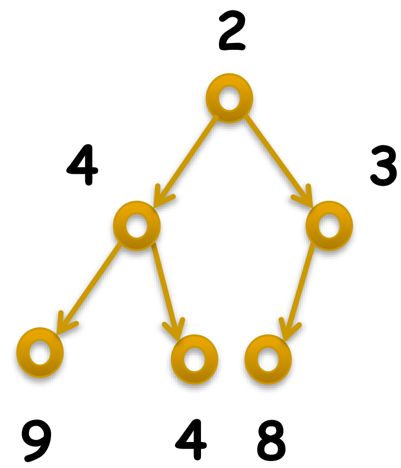
L < P < R

Binary min-Heap

L > P, R > P

A binary heap is a complete binary tree which satisfies the heap ordering property.





1. Structure Property

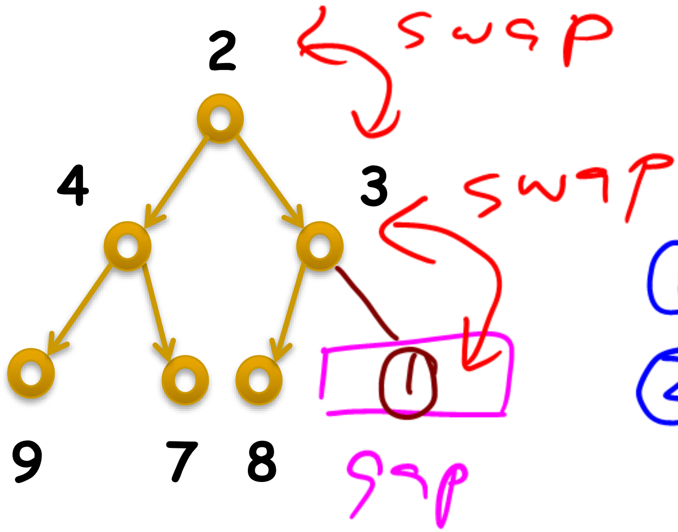
2. Ordering Property

0	1	2	3	4	5	6	7
X	2	4	3	9	4	8	gap

Consider k-th element of the array,

- its left child is located at 2*k index
- its right child is located at 2*k+1 index
- its parent is located at k/2 index

insert (tree reps)



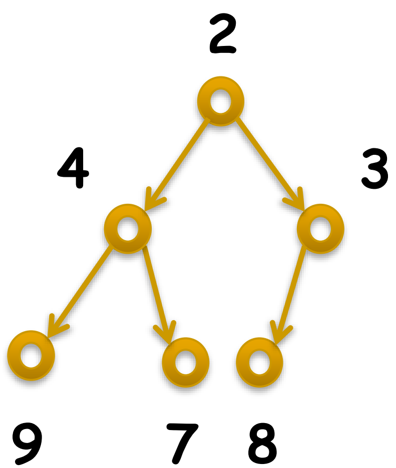
Algorithm:

1. insert to a gap
2. restore the ordering property

Runtime = # of swaps in the worst-case

$O(\log n)$

insert (array reps)



0	1	2	3	4	5	6	7
X	2	4	3	9	7	8	1
			1				3
	1		2				3

Implementation: a single for-loop

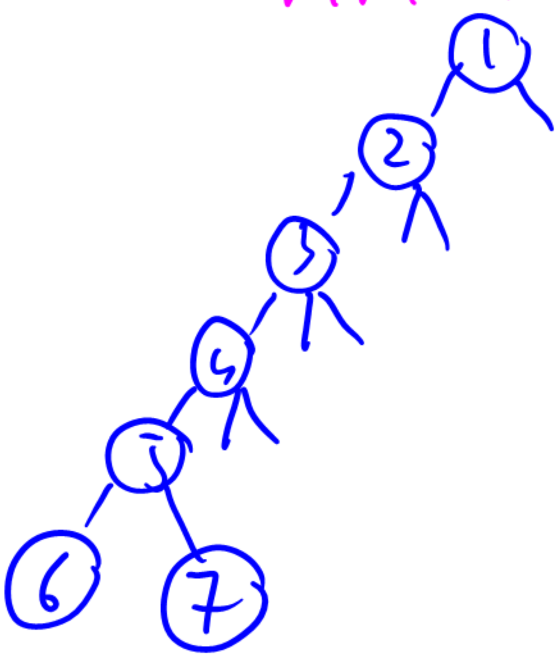
percolation

Discussion Problem 1

$63 = 64 - 1 = 2^6 - 1$

The values 1, 2, 3, . . . , 63 are all inserted (in any order) into an initially empty min-heap. What is the smallest number that could be a leaf node?

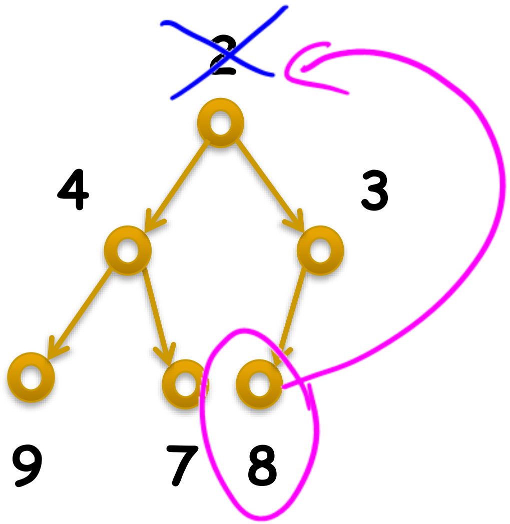
min-heap



Proof by example.

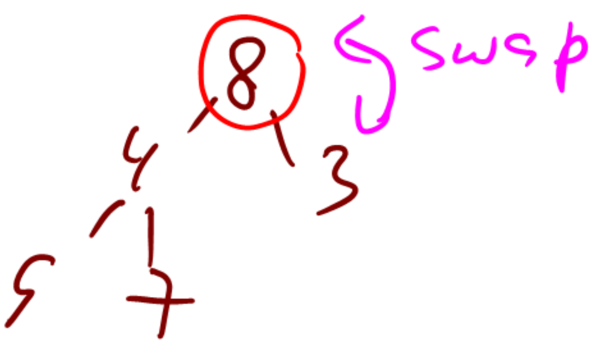
deleteMin (tree reps)

remove the root



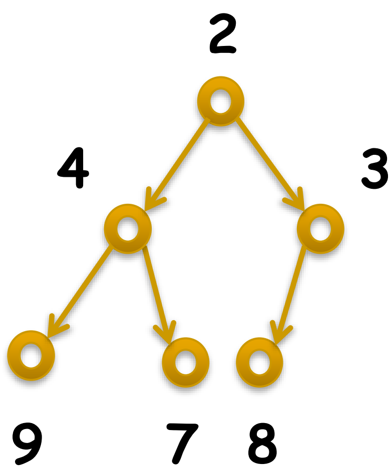
① preseve the structural property: by moving the last item to the root

② percolate down



Runtime: $O(\log n)$

deleteMin (array reps)



0	1	2	3	4	5	6	7
X	2	4	3	9	7	8	
X	8	4	3	9	7		
	3		8				

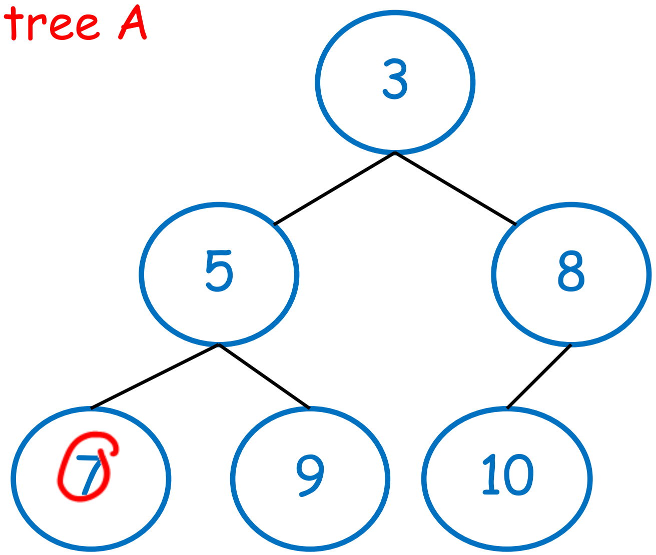
Implementation: a single for-loop

Discussion Problem 2

Suppose you have two binary min-heaps, A and B, with a total of *n* elements between them. You want to discover if A and B have a key in common. Give a solution to this problem that takes time O(n log n). Do not use the fact that heaps are implemented as arrays, use only API operations: insert and deleteMin.

Runtime: $O(n \log n)$

tree A



Compare $3 < 6$

A.delete

$5 < 6$

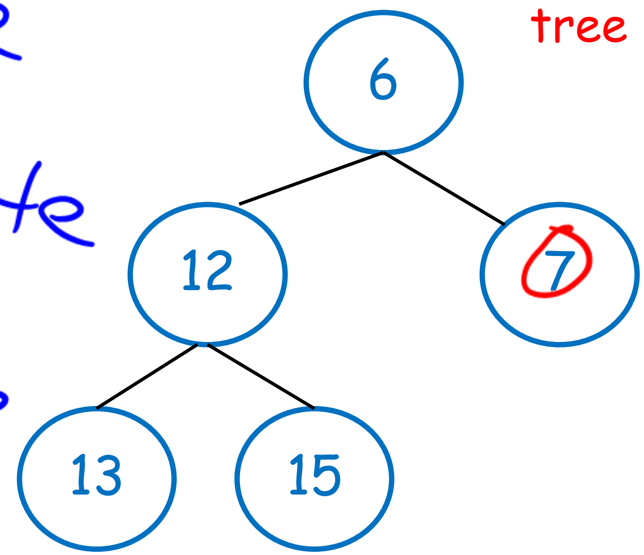
A.delete

$7 > 6$

B.delete

$7 = 7$

tree B

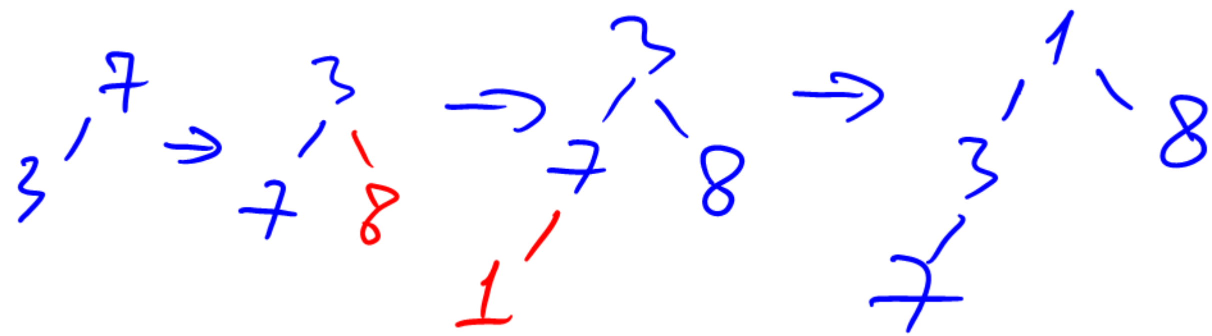


Build a BST. $O(n^2)$

Build a Heap by Insertion

Given an array - turn it into a heap.

insert 7, 3, 8, 1, 4, 9, 4, 10, 2, 0 into an initially empty heap.



and so on

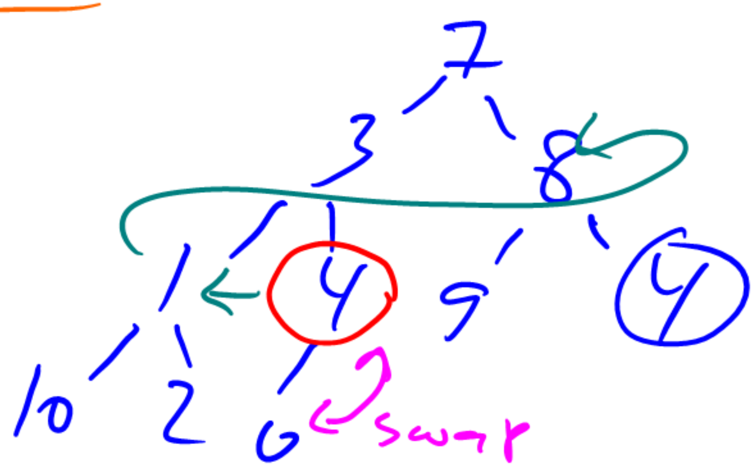
Runtime: $O(n \cdot \log n)$

Total work: $\log 2 + \log 3 + \log 4 + \ldots + \log n =$

$= \log(n!) = O(n \log n)$

Build a Heap in O(n)

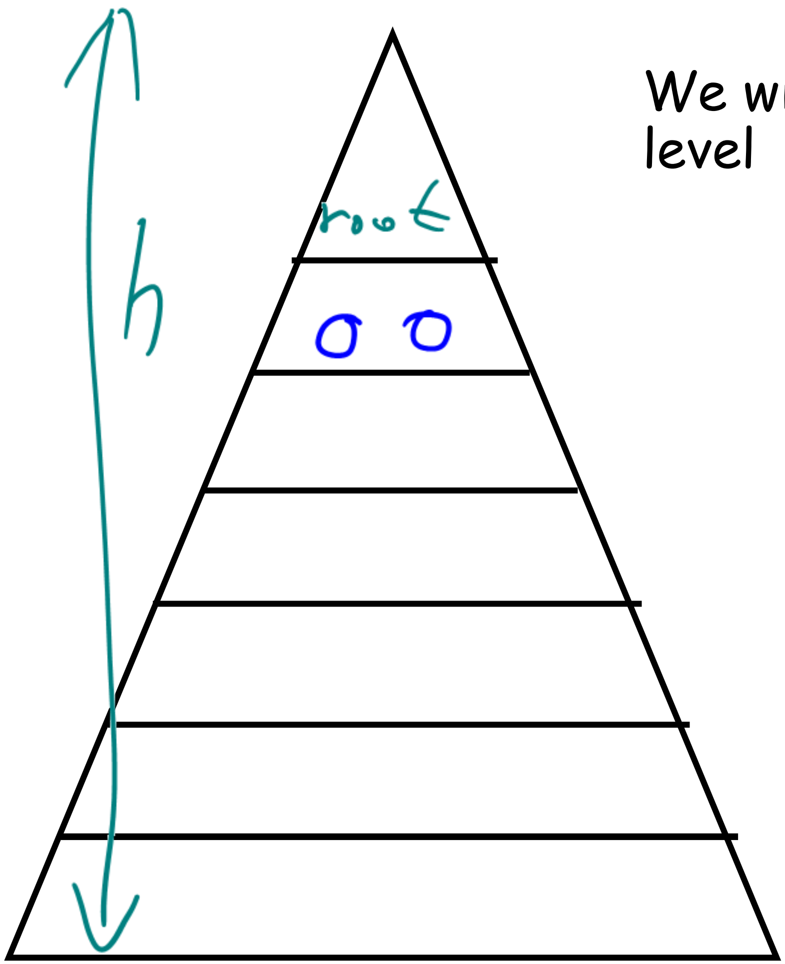
Heapify: 7, 3, 8, 1, 4, 9, 4, 10, 2, 0



Algorithm;

1. start at index n/2
2. compare with its children, and swap with the smallest child if it exists.
3. Go to the left sibling
4. Repeat #2

Complexity of heapify



We will count the max number of swaps at each level

height	# of nodes	# of swaps
0 root	1	h
1	2	h-1
2	4	h-2
...	...	...
h-1	2^{h-1}	1

Complexity of heapify

Runtime = Total Cost = $1 \times h + 2 \times (h-1) + 4 \times (h-2) + \ldots + 2^{h-1} \times 1 =$

$= \sum_{k=0}^{h-1} 2^k \times (h-k) =$ Wolfram Alpha

$= O(n)$

Break, 5 mins

Discussion Problem 3

How would you sort using a binary heap?

What is it runtime complexity?

Sorting using BST: $O(n^2)$

① make a BST: $O(n^2)$

② traversal, in-order, $O(n)$

Heapsort Algorithm:

① make a heap: $O(n)$

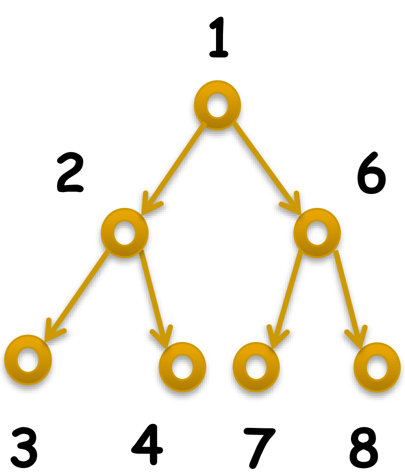
② delete : $O(n \log n)$

HEAPSORT

Run delMin n-times

in-place
nonstable

$O(n \log n)$



0	1	2	3	4	5	6	7
	1	2	6	3	4	7	8

	2	3	6	8	4	7	1

	3	4	6	8	7	2	1

	4	8	6	7	3	2	1

Discussion Problem 4

How would you merge two binary min-heaps?

What is it runtime complexity? O(n)

O(n)

① offline algorithms
all data is available: you can preprocess the data

heapify

② online algorithms
(streaming data)

Runtime O(n log n) by insertion

Discussion Problem 5

Devise a heap-based algorithm that finds *k* largest elements out of *n* elements. Assume that *n* > *k*. What is its runtime complexity?

① offline (all data is available)

Algorithm:

a) build a max-heap

b) run delete k times

$O(n + k\log n)$

② online (streaming data)

Algorithm:

a) wait for k items, then build a heap, min-heap

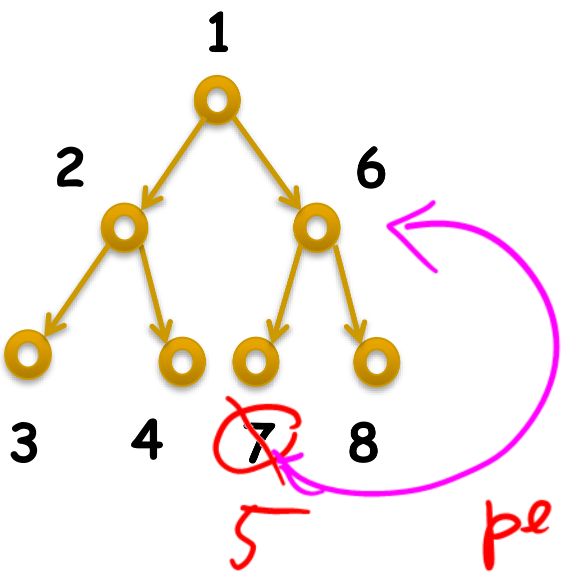
b) wait for (k+1)st item

c) new item ≤ root, ignore it; o.w. run deleteMin, then insert

$O(k + (n-k)\log k)$

decreaseKey

Runtime: O(log n)
worst-case

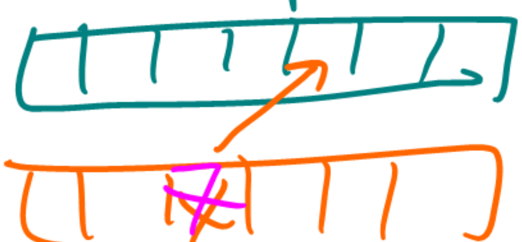


swap

percolate up

how do you find 7 in O(1) time?

heap



index (data, index)

data

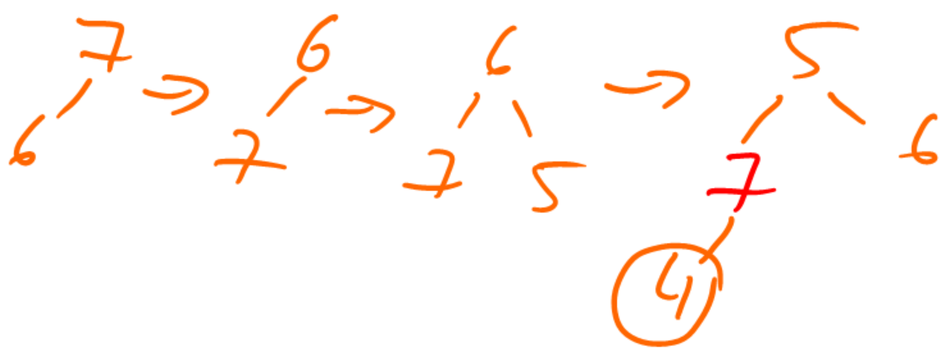
hashtable

A new kind of heaps

We want to create a heap with a better <u>amortized</u> complexity of insertion. This example will demonstrate that binary heaps do not provide a better upper bound for the worst-case complexity.

0 1 1 2 2 2

Insert 7, 6, 5, 4, 3, 2, 1 into an empty binary min-heap.



The total number of swaps

$$\sum_{k=0}^{\log n - 1} k \cdot 2^k = O(n \log n)$$

$$AC(\text{per insert}) = \frac{O(n \log n)}{n} = \log n$$

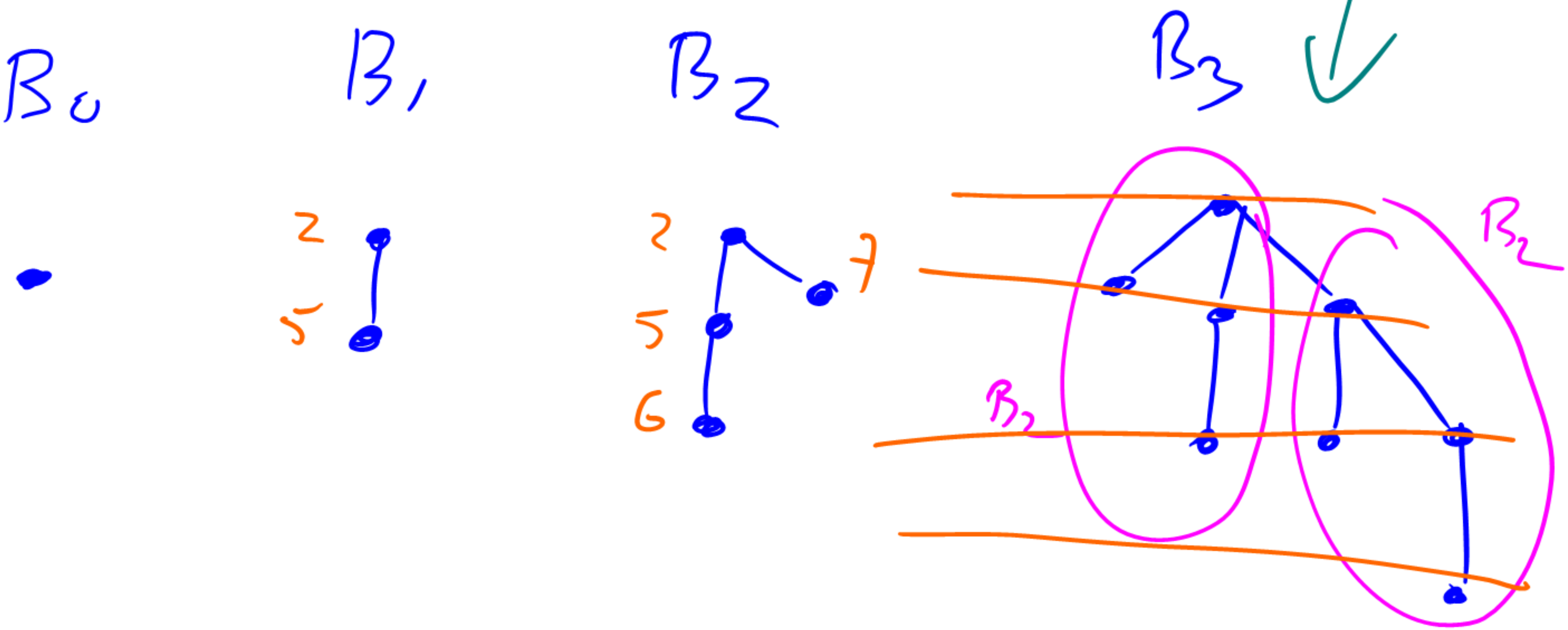
$$(1+x)^3 = 1 + 3x + 3x^2 + 1x^3$$

?

Binomial Trees B_k

The binomial tree B_k is defined as

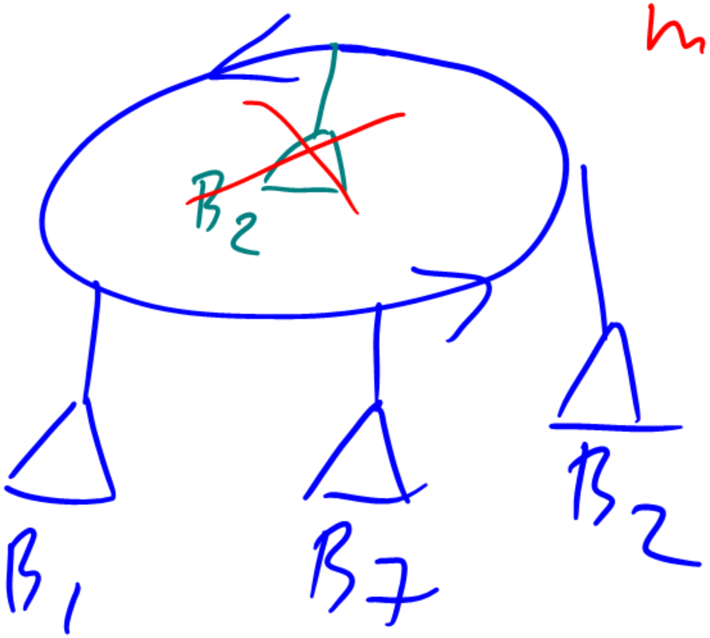
1. B_0 is a single node
2. B_k is formed by joining two B_{k-1} trees



Binomial Heaps

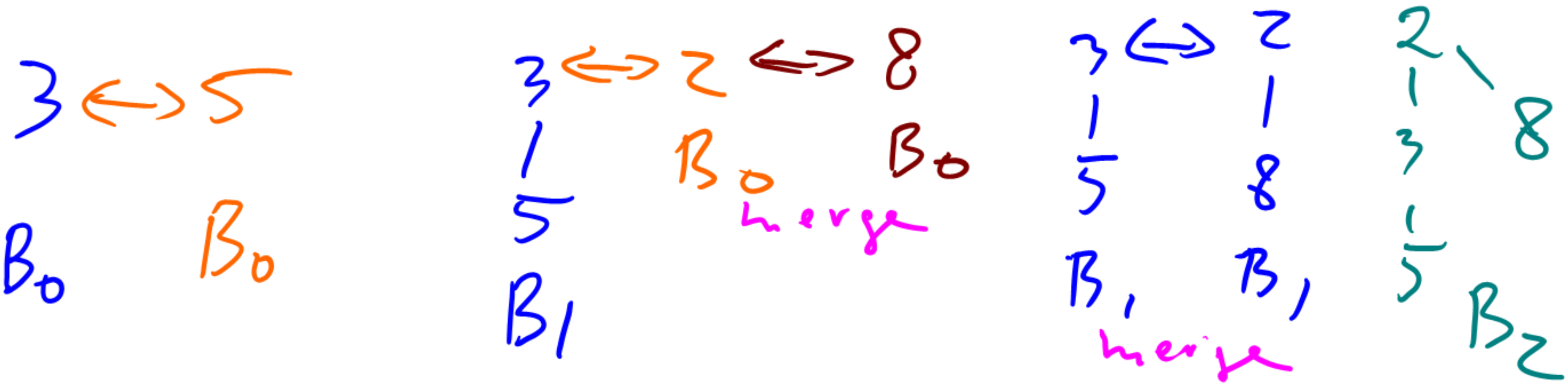
A binomial heap is a collection (a linked list or a queue) of at most Celling(log n) binomial trees (of unique rank) in increasing order of size where each tree has a heap ordering property.

min-heap



Discussion Problem 6

Given a sequence of numbers: 3, 5, 2, 8, 1, 5, 2, 7.
Draw a binomial heap by inserting the above numbers reading them from left to right



Discussion Problem 7

How many binomial trees does a binomial heap with 25 elements contain?
What are the ranks of those trees?

$$25_2 = (16 + 8 + 1)_2 = 1\ 1\ 0\ 0\ 1$$

$B_4 \quad B_3 \qquad\qquad B_0$

Insertion

What is its worst-case runtime complexity? $O(\log n)$

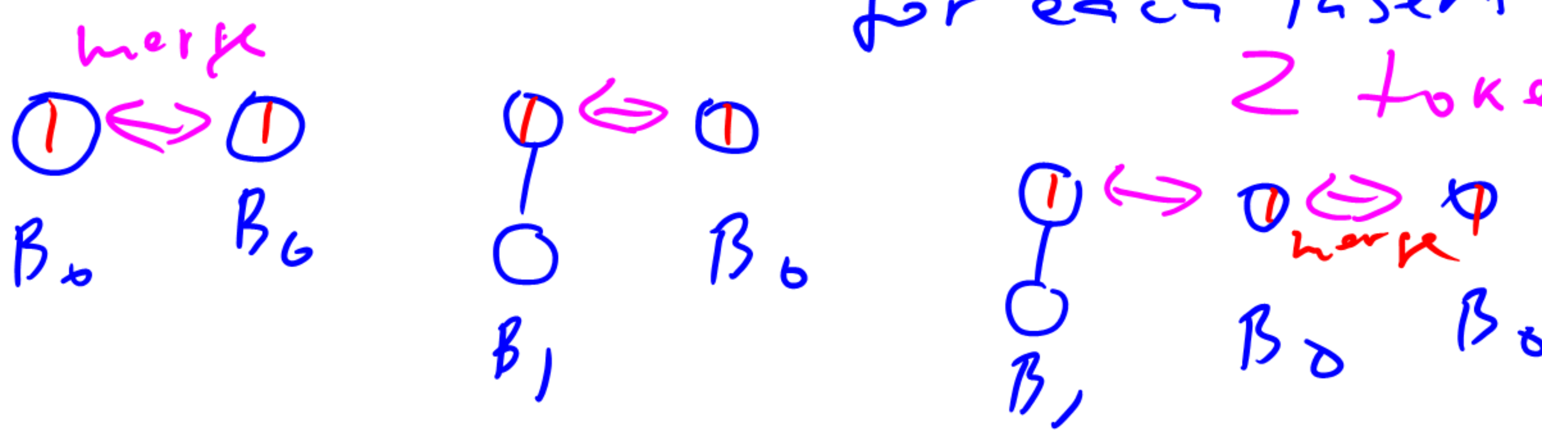
$$15_2 = 1111$$
$$\begin{array}{r} 1111 \\ +\;1 \\ \hline 10000 \end{array}$$

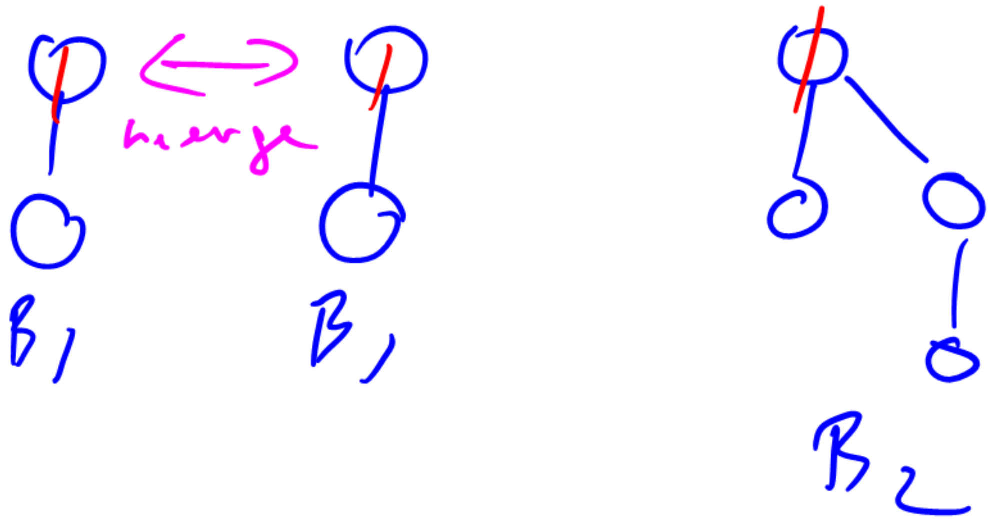
What is its amortized runtime complexity? lecture 2

Use an accounting method.

how many tokens for each insert?

2 tokens




merge
B_1
B_1
B_2

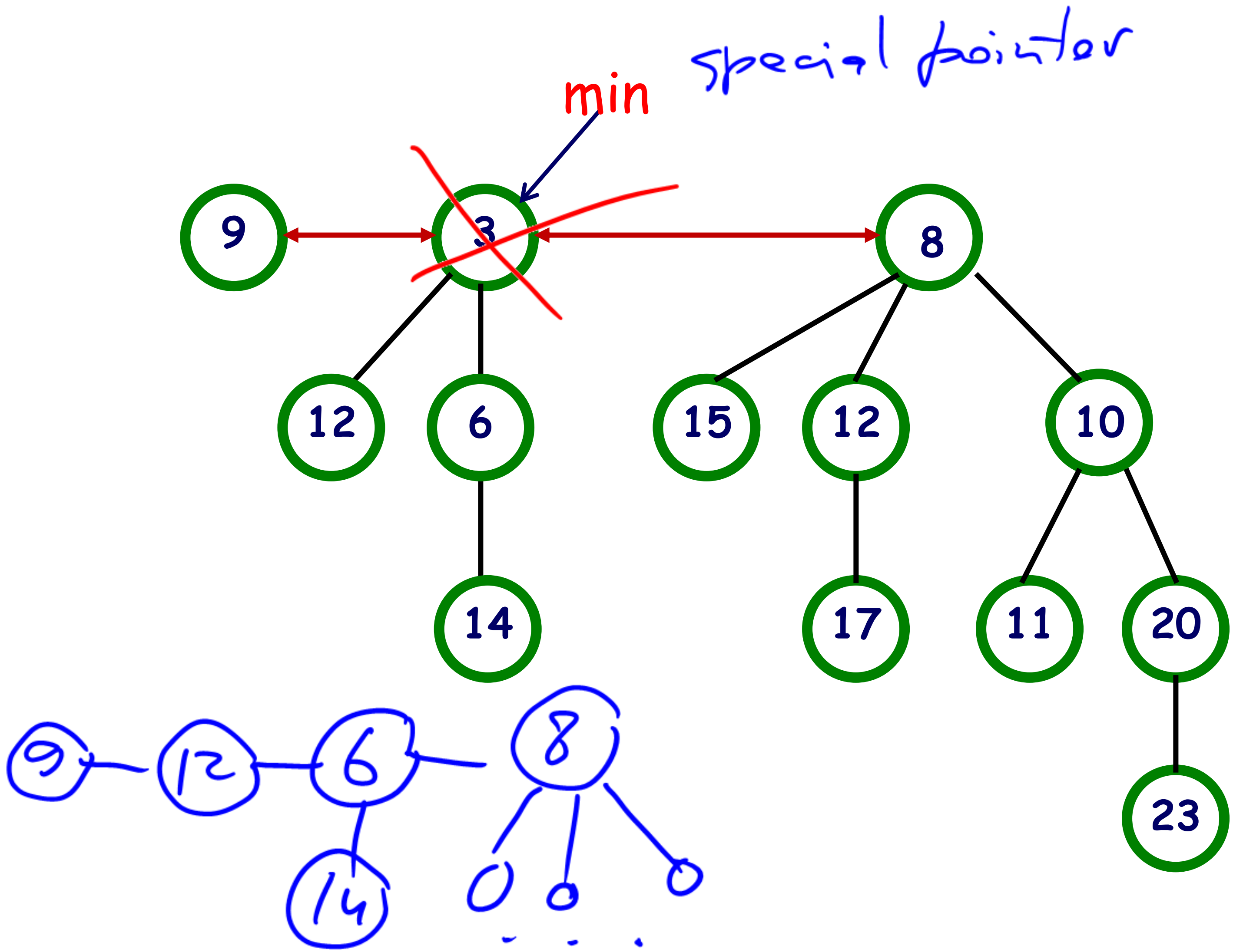
Building :
Binomial vs Binary Heaps

The cost of inserting n elements into a binary heap, one after the other, is $\Theta(n \log n)$ in the worst-case. This is an online algprithm.

If n is known in advance (an offile algorithm), we run heapify, so a binary heap can be constructed in time $\Theta(n)$.

The cost of inserting n elements into a binomial heap, one after the other, is $\Theta(n)$ (amortized cost), even if n is not known in advance.

deleteMin()



deleteMin()

Algorithm:

① delete the min, $O(1)$

② move subtrees to the top (LL) level

③ traverse all and merge trees of the same rank

Runtime: $O(\log n)$

④ update the min-pointer, $O(\log n)$

Discussion Problem 8

Devise an algorithm for merging two binomial heaps and discuss its complexity. Merge $B_0B_1B_2B_4$ with B_1B_4.

merging two binary heap — $O(n)$

binomial — $O(\log n)$

heaps

Algorithm for binomial heaps

① merge two LL, $O(1)$

② traverse and merge binomial trees of the same rank, $O(\log n)$

$$
\begin{array}{r}
10111 \\
+\ 10010 \\
\hline
101001
\end{array}
$$

$B_5\,B_3\,B_0$

Heaps

"lazy"

	Binary	Binomial	Fibonacci
findMin	$\Theta(1)$	$\Theta(1)$	
deleteMin	$\Theta(\log n)$	$\Theta(\log n)$	
insert	$\Theta(\log n)$	$\Theta(1)$ (ac)	
decreaseKey	$\Theta(\log n)$	$\Theta(\log n)$	$O(1)$ ac
merge	$\Theta(n)$	$\Theta(\log n)$	

ac – amortized cost.

FIBONACCI HEAPS

Idea: relaxed (*lazy*) binomial heaps

Goal: decreaseKey in $O(1)$ ac.

The algorithm is outside of the scope of this course.

Heaps

•

	Binary	Binomial	Fibonacci
findMin	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
deleteMin	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$ (ac)
insert	$\Theta(\log n)$	$\Theta(1)$ (ac)	$\Theta(1)$
decreaseKey	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$ (ac)
merge	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$ (ac)