Analysis of Algorithms

V. Adamchik

Lecture 3

Spring 2023

CSCI 570

University of Southern California

Heaps

Reading: chapter 3

Amortized Analysis

In a sequence 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.

An example: 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!

The Accounting Method

The accounting method (or the banker's method) computes the individual cost of each operation. This approach is only practical if we guess that the amortized cost is constant.

We assign different charges to each operation; some operations may charge more or less than they actually cost. The amount we charge an operation is called its amortized cost.

Example: amortized queue (two stacks A and B)

Enqueue (A.push) requires 3 tokens (it generate a surplus of 2 tokens)
Dequeue (B.dequeue) requires 1 token

Consider n enqueue operations, followed by a single dequeue.
In order to perform dequeue we need to do: A.pop and B.push n times.
This requires 2n tokens, which are generate by n enqueues.

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?

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

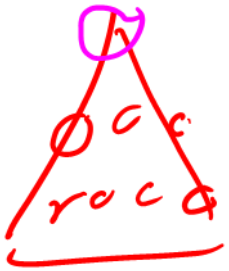
$1, 1, 1, \ldots, 1, n^2 \rightarrow T(n) = 2n - 1 + n^2 = O(n^2)$

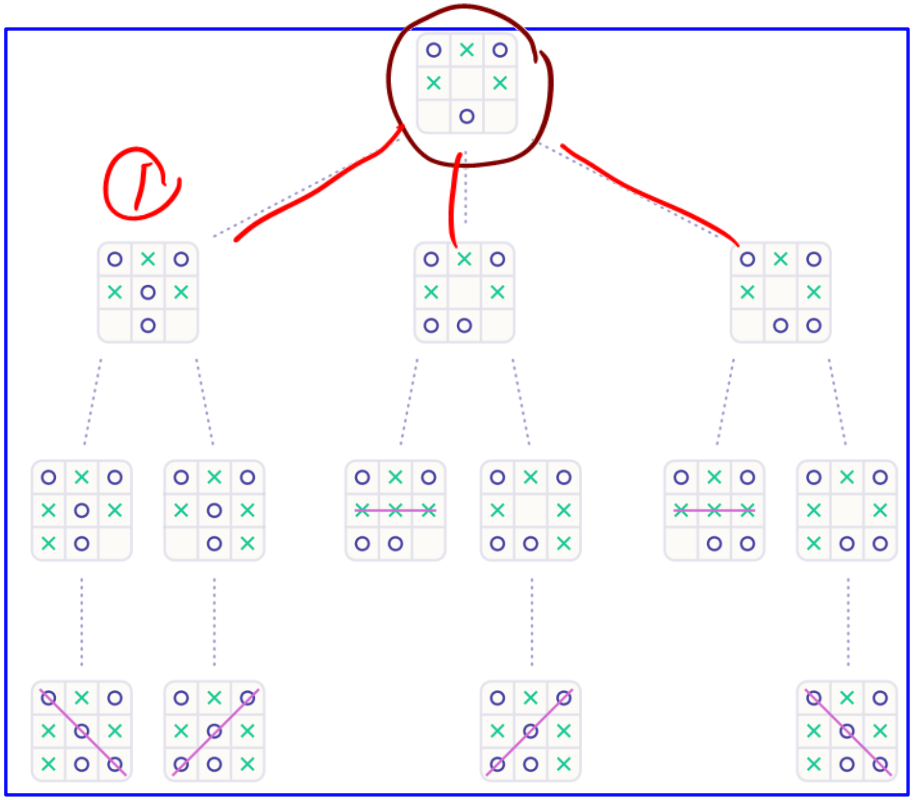
$n-1$

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.





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

Binary min-Heap

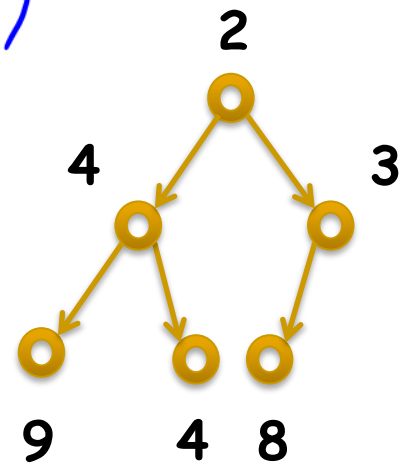
BST

L < P < R

Heap: L < P, P > R

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

not a heap



1. Structure Property

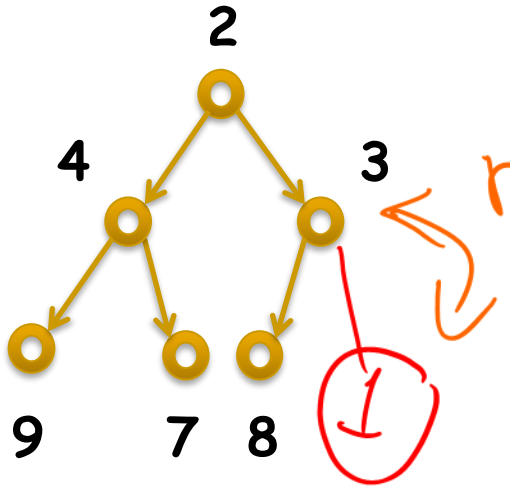
2. Ordering Property

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

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)



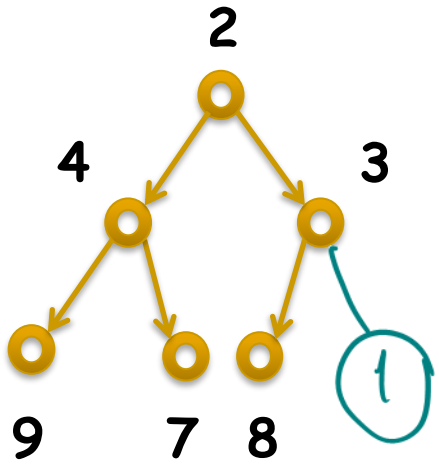
percolate up

Algorithm:

1. insert it to a gap

Runtime Complexity! O(log n)

insert (array reps)



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

implementation: a single for-loop

Discussion Problem 1

Break.

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?

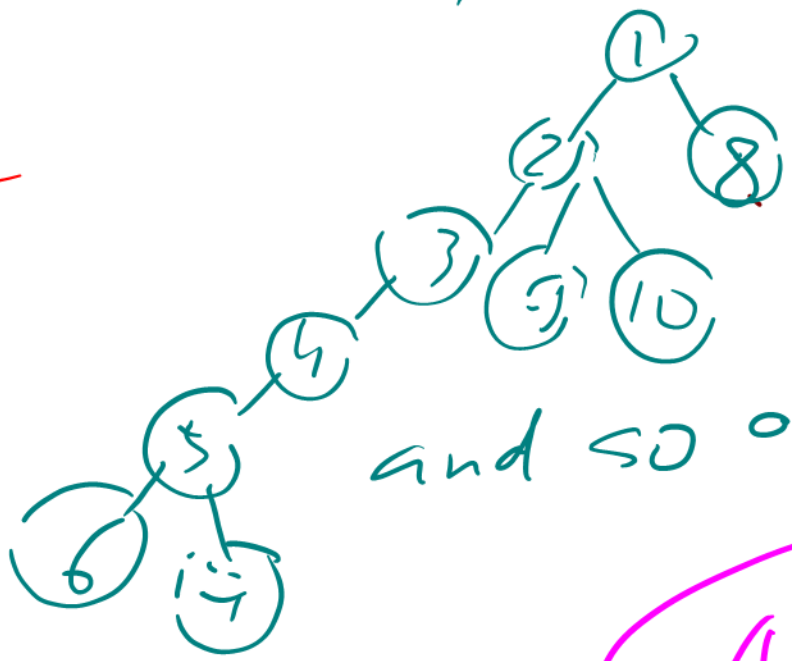
Proof by example

1) 4

2) ~~32~~

3) 6

4) ~~—~~



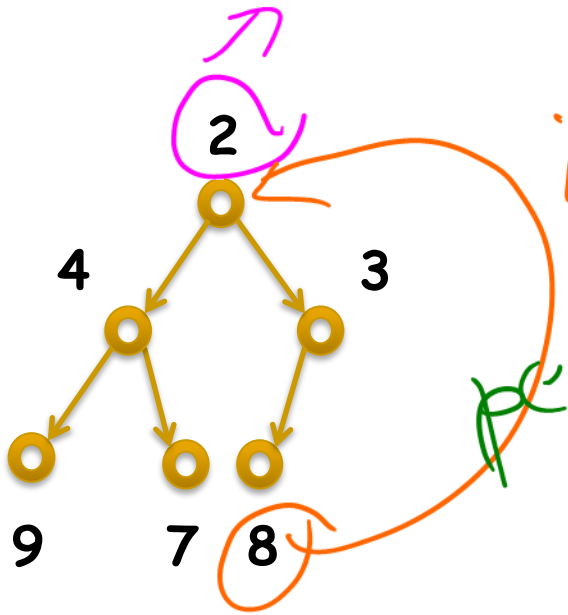
and so on

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

Ans. : 6

deleteMin (tree reps)

remove

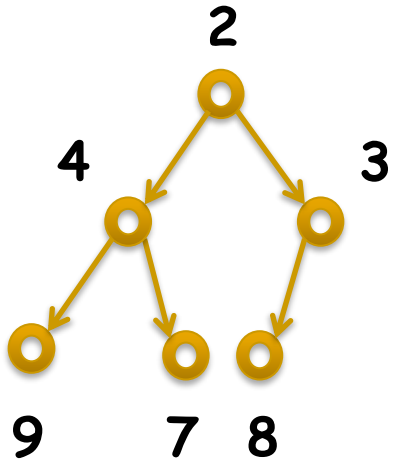


it doesn't hold a heap prop. !

percolation down

Runtime: $O(\log n)$

deleteMin (array reps)

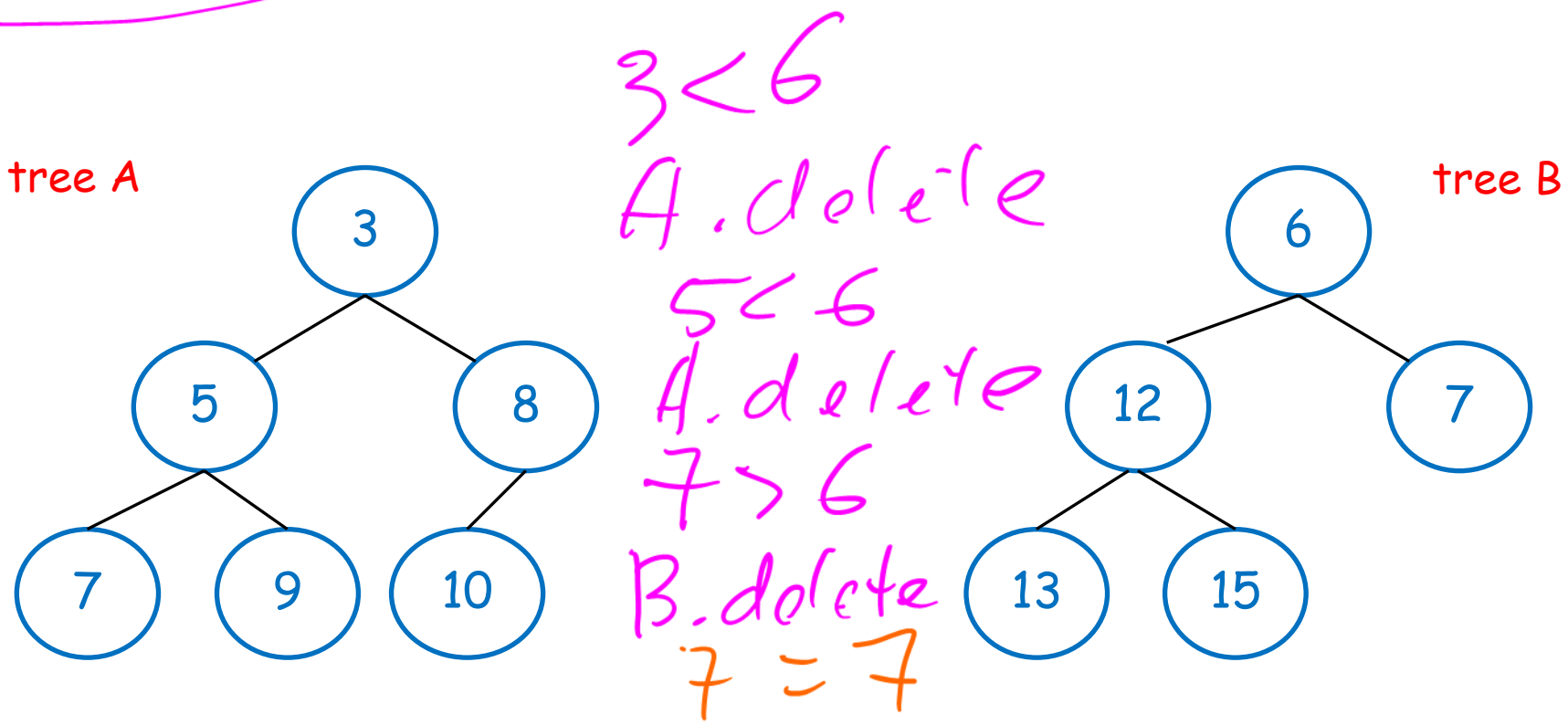


0	1	2	3	4	5	6	7
X	2	4	3	9	7	8	
X	8	4	3	9	7		
X	3	4	8	9	7		

implementation: a single for-loop

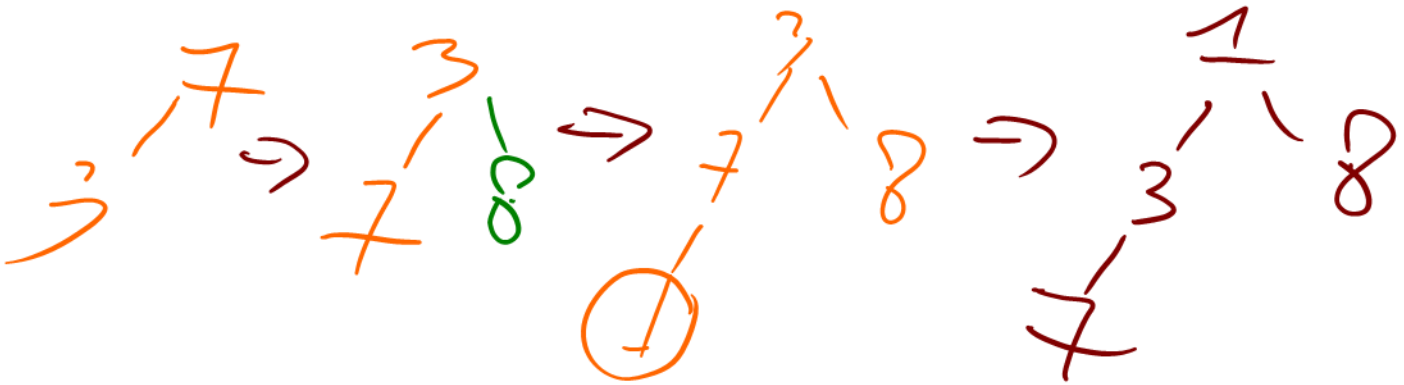
Discussion Problem 2

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.



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.



ch. 2

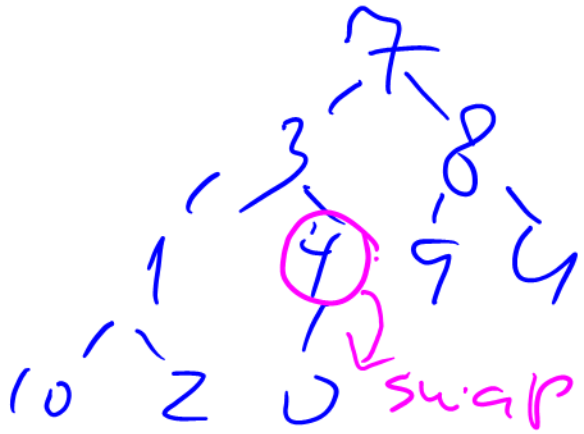
Runtime: $O(n \cdot \log n)$

Better analysis:

$\log 2 + \log 3 + \log 4 + \dots + \log n = \log(n!) = \Theta(n \log n)$

Build a Heap in O(n)

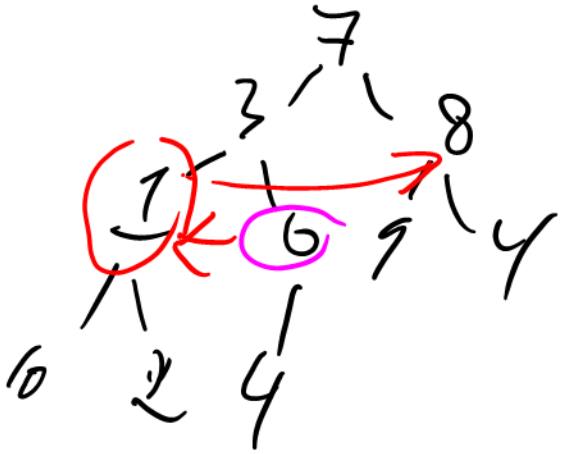
Heapify: 7, 3, 8, 1, 4, 9, 4, 10, 2, 0



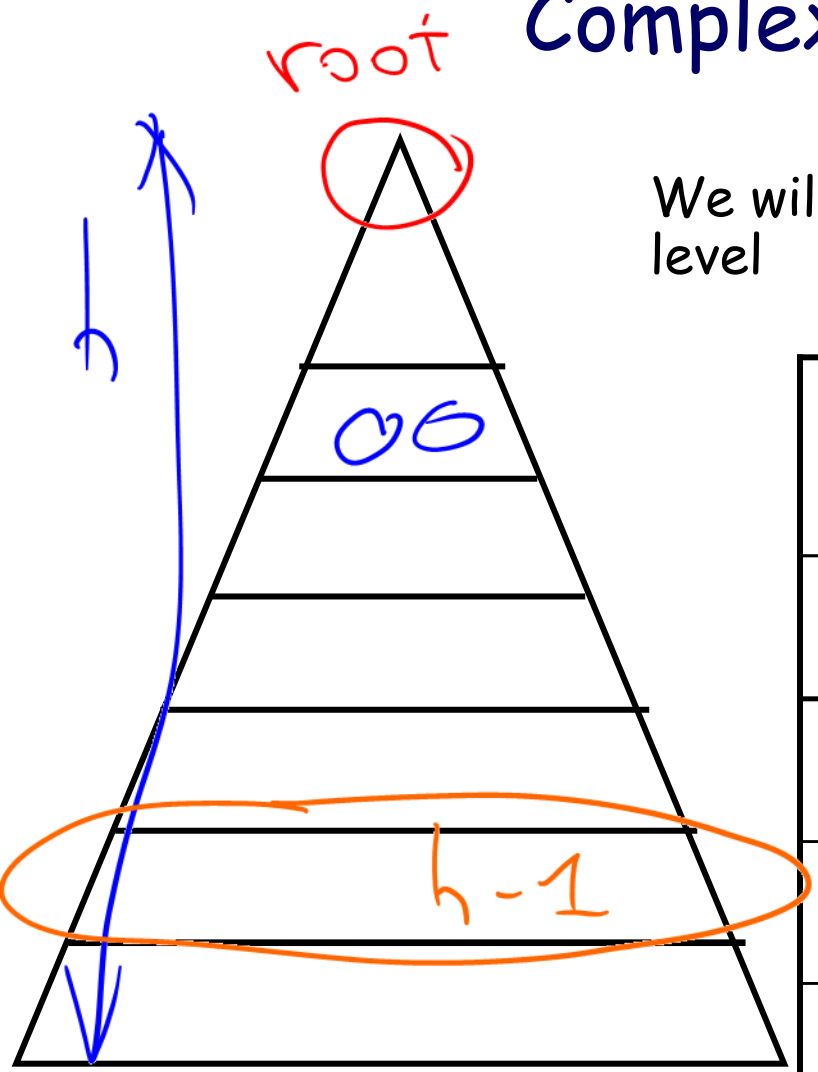
not a heap

Algorithm:

1. start at index $\frac{h}{2}$
2. move to the root and swap



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		
…	…	…
h-1	2^{h-1}	1

Discussion Problem 3

How would you sort using a binary heap?

What is it runtime complexity?

heap sort

Algorithm: (input = array of data)

① heabify, O(n)

② in a loop: run deleteMin

Runtime: O(n log n)

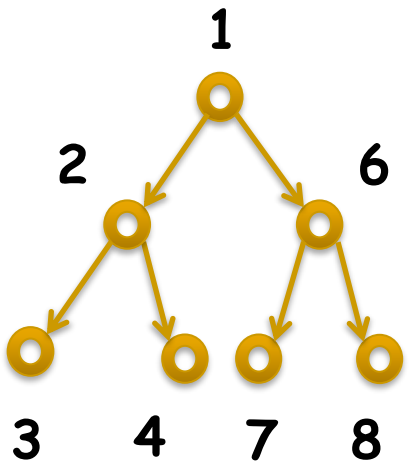
BST: O(n²)

HEAPSORT

Run delMin n-times

$O(n \log n)$

in-place
nonstable



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

B, A

How would you merge two binary min-heaps?

What is it runtime complexity?

① heapify, O(n)

② API, B.insert(A.deleteMin)

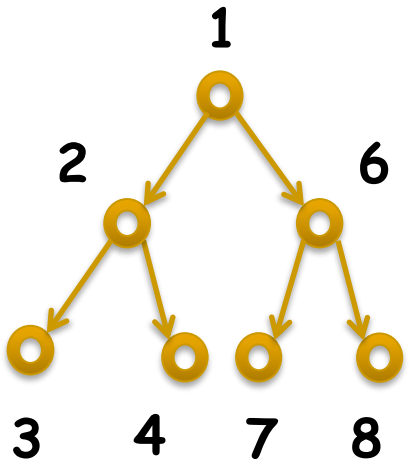
Discussion Problem 5

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)
   a) build a max-heap
   b) deleteMax (k times)

② Online Algorithm (stream of data)
   a) build a min-heap
   b) wait for (k+1) item
   c) item ≤ root, then ignore it
      item > root, then deleteMin & insert

decreaseKey



A new kind of heaps

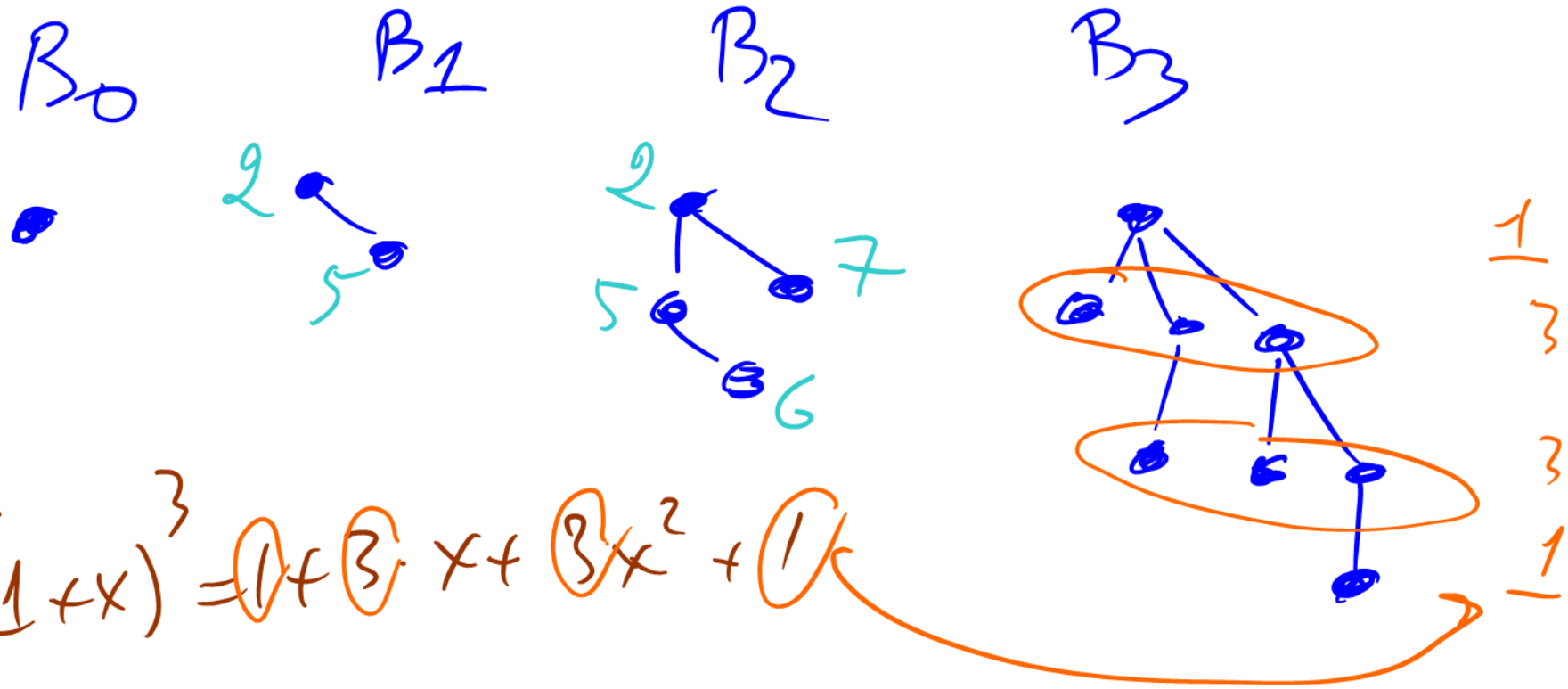
We want to create a heap with a better amortized complexity of insertion. This example will demonstrate that binary heaps do not provide a better upper bound for the worst-case complexity.

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

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



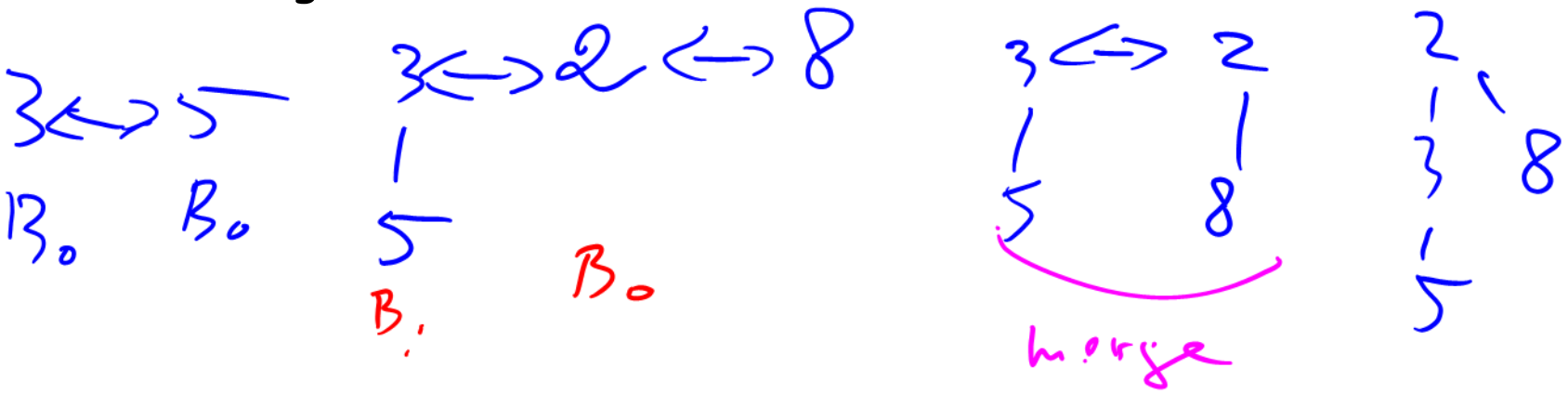
Binomial Heaps

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.

Discussion Problem 6

Break.

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?

$25_2 = (16 + 8 + 1)_2 = \underset{B_4}{1}\underset{B_3}{1}00\underset{B_0}{1}$

$N_2 = \#\text{bits} = O(\log N)$

Insertion

What is its worst-case runtime complexity?

$15_2 = 1111$

$+\quad\quad 1$

10000

$O(\log n)$

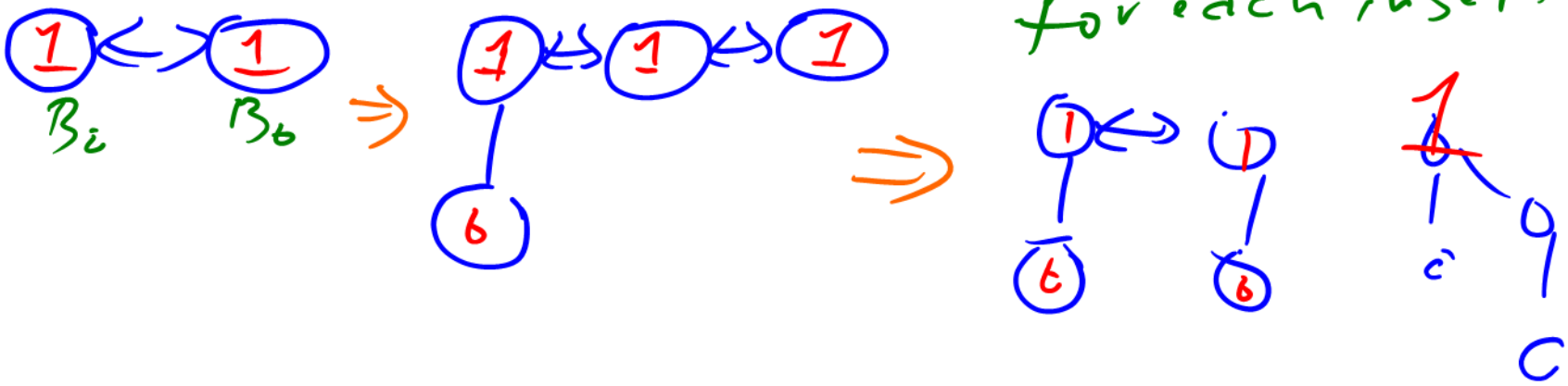
$O(2)$

What is its amortized runtime complexity?

Use an accounting method.

how many tokens?

2 tokens for each insert



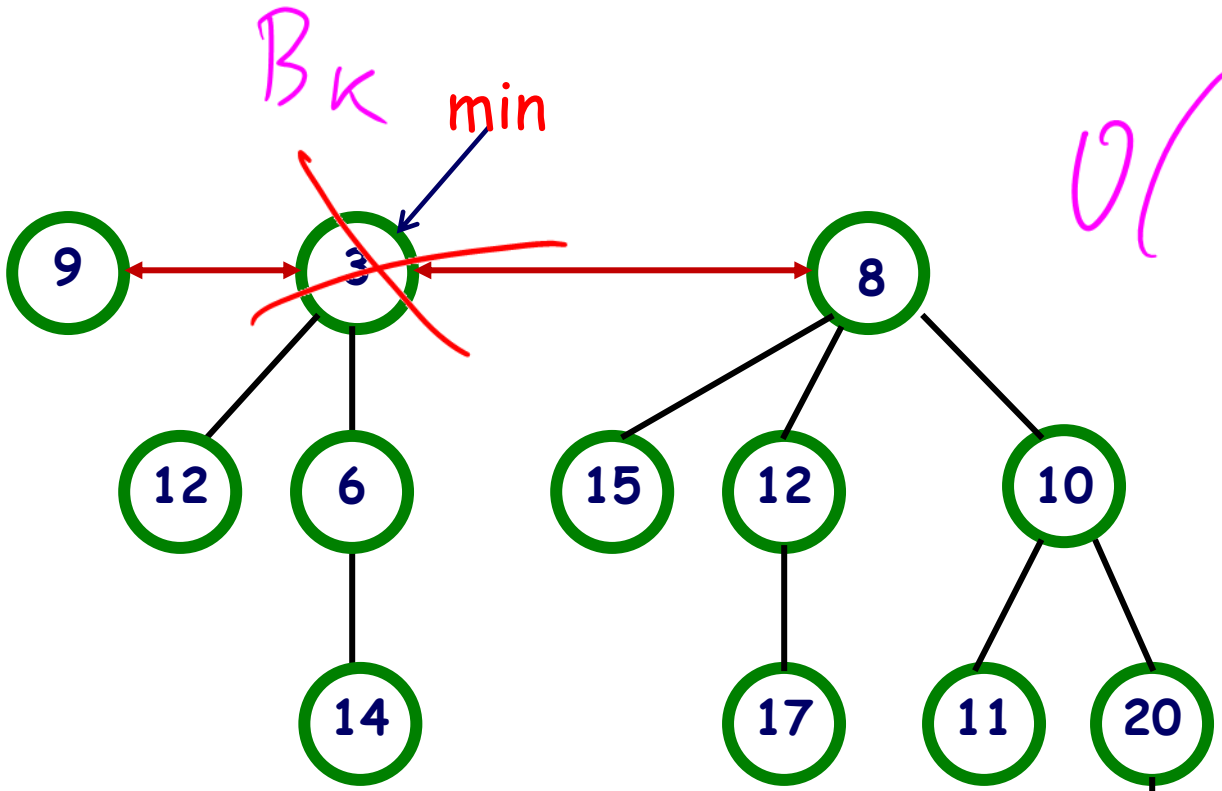
Building : Binomial vs Binary Heaps

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.

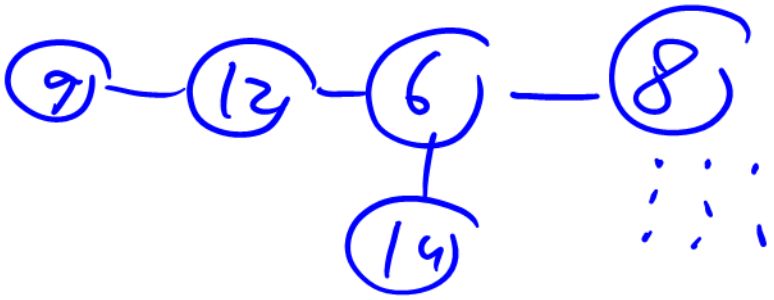
If n is known in advance (an offile algorithm), we run heapify, so a binary heap can be constructed in time $\Theta(n)$.

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()



O(log n)



Bk has k children

deleteMin()

Algorithm:

1) delete the min, $O(1)$ $O(\log n)$
2) move subtrees to the top level
3) traverse a collection (LL) and merge binomial trees of the same rank $O(\log n)$
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.

Runtime complexity: $O(\log n)$

Binary heap: $O(n)$

$$
\begin{array}{r}
 1\,0\,1\,1\,1 \\
+\;1\,0\,0\,1\,0 \\
\hline
1\,0\,1\,0\,0\,1
\end{array}
$$

B_5 B_3 B_0

Heaps

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