

# Priority Queues

A priority queue has to perform these two operations fast!

1. Insert an element into the set

2. Find the smallest element in the set

Insert

Find Min

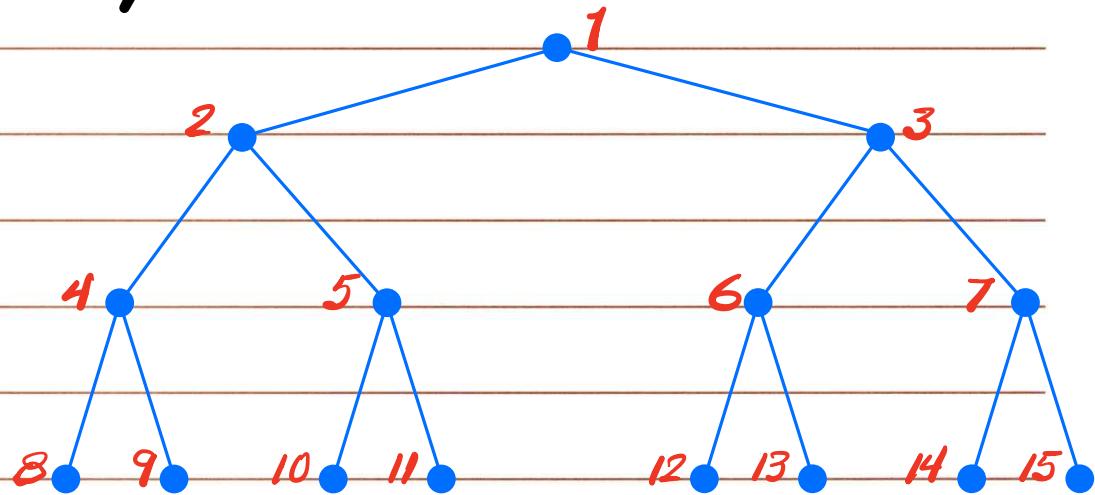
Array Implementation       $O(1)$        $O(n)$

Sorted array       $O(n)$        $O(1)$

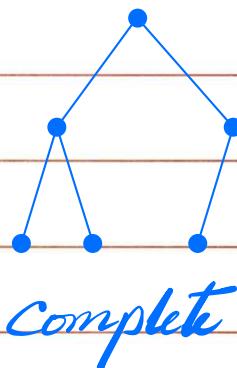
Linked List       $O(1)$        $O(n)$

Sorted linked list       $O(n)$        $O(1)$

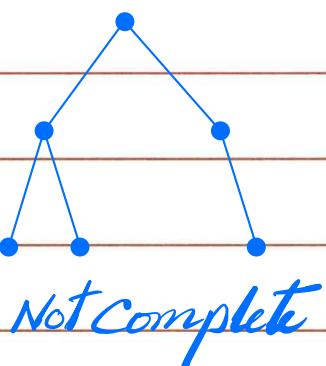
Def. A binary tree of depth  $k$  which has exactly  $2^k - 1$  nodes is called a full binary tree.



Def. A binary tree with  $n$  nodes and of depth  $k$  is complete iff its nodes correspond to the nodes which are numbered 1 to  $n$  in the full binary tree of depth  $k$ .



complete



Not complete

Traversing a complete binary tree stored as an array.

$\text{Parent}(i)$  is at  $\lfloor i/2 \rfloor$  if  $i \neq 1$

if  $i=1$ ,  $i$  is the root

$\text{Lchild}(i)$  is at  $2i$  if  $2i \leq n$

otherwise, it has no left child

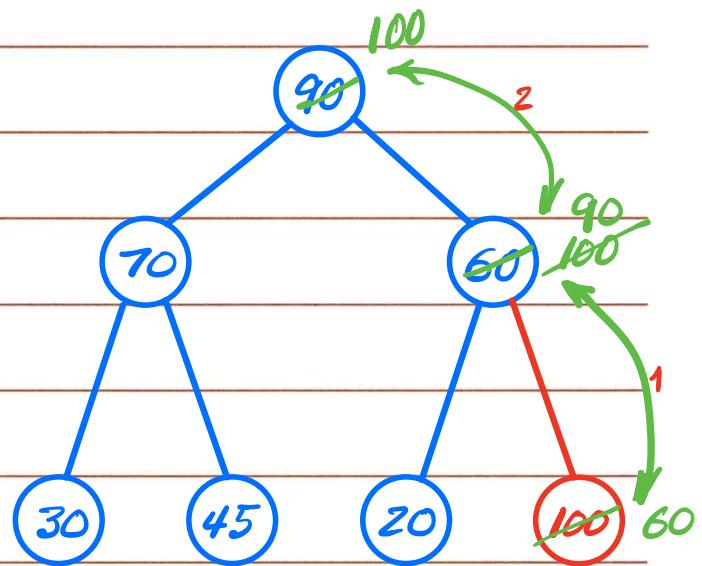
$\text{Rchild}(i)$  is at  $2i+1$  if  $2i+1 \leq n$

otherwise, it has no right child

Def. A binary heap is a complete binary tree with the property that the value of the key at each node is at least as large as the key values at its children (Maxheap)

Find-Max

Takes  $O(1)$



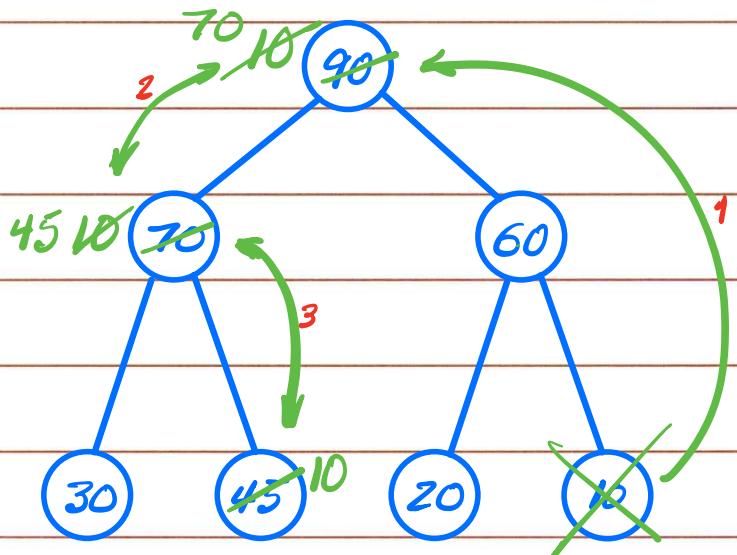
Insert

ex. insert 100

Takes  $O(\lg n)$

Extract-Max

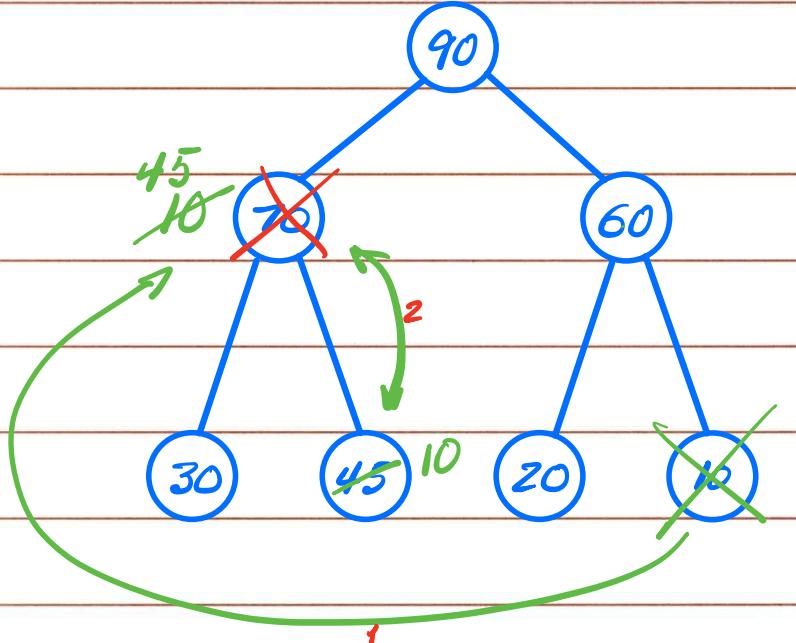
Takes  $O(\lg n)$



## Delete

ex. Delete 70

Takes O(lgn)

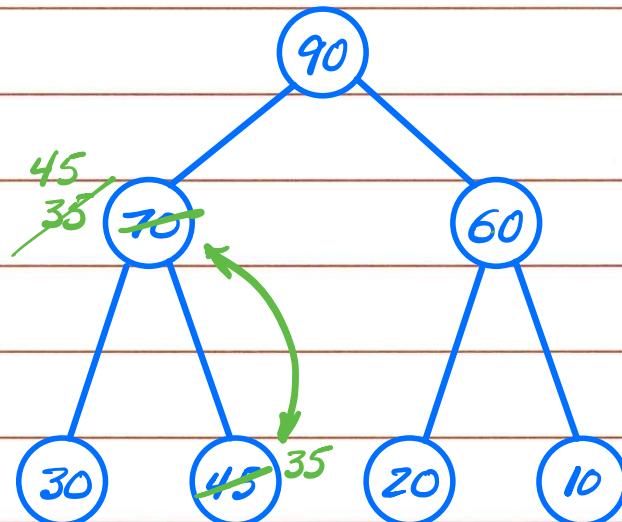


## Decrease-key

ex. Decrease-key

70 to 35

Takes O(lgn)



## Construction

Can be done in  $O(n \lg n)$  time using n insert operations.

Can we do better?

## Bottom up construction

# of nodes

1



# of swaps  
 $\lg n$

$n/8$

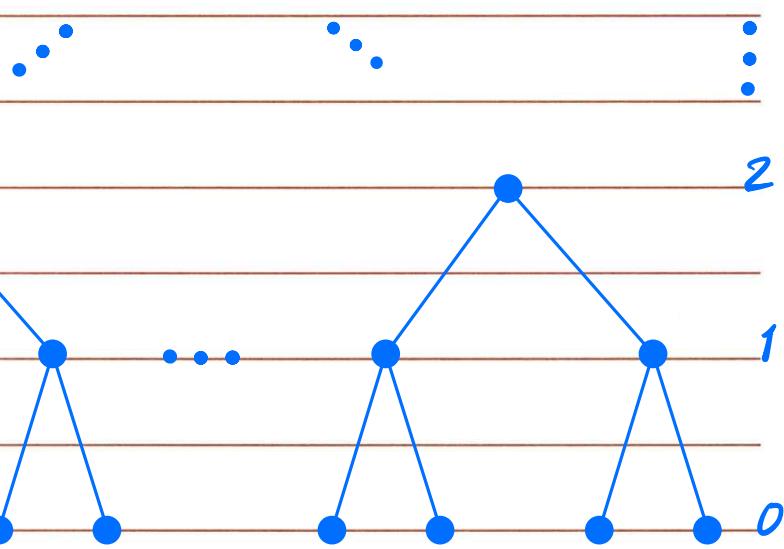
$n/4$

$n/2$

2

1

0



$T = \text{Maximum number of swaps needed}$

$$T = n/4 * 1 + n/8 * 2 + n/16 * 3 + \dots$$

$$T_{1/2} = n/8 * 1 + n/16 * 2 + n/32 * 3 + \dots$$

$$T - T_{1/2} = n/4 * 1 + \underbrace{n/8 * 1 + n/16 * 1}_{n/2} + \dots$$

$$T_{1/2} = n/2$$

$$T = n$$

Bottom up construction takes  $O(n)$

Q: What is the best run time to merge two binary heaps of size  $n$ ?

A:  $O(n)$  using linear time construction

Finding the top k elements in an array.

Input: An unsorted array of length n.

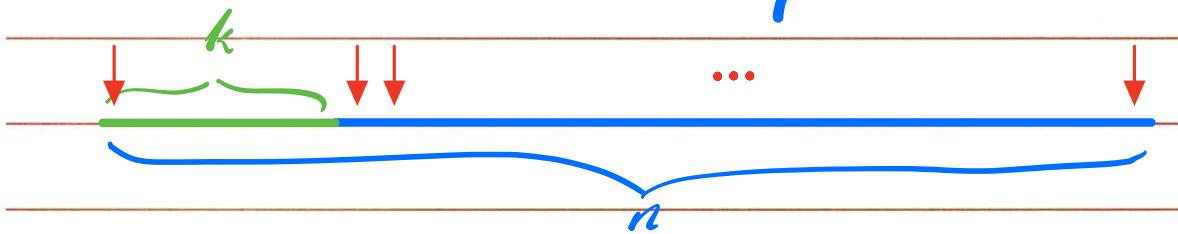
Output: Top k values in the array ( $k < n$ )

Constraints:

- You cannot use any additional memory
- Your algorithm should run in time  $O(n \lg k)$

Solution:

- Construct a Minheap of size  $k$ .



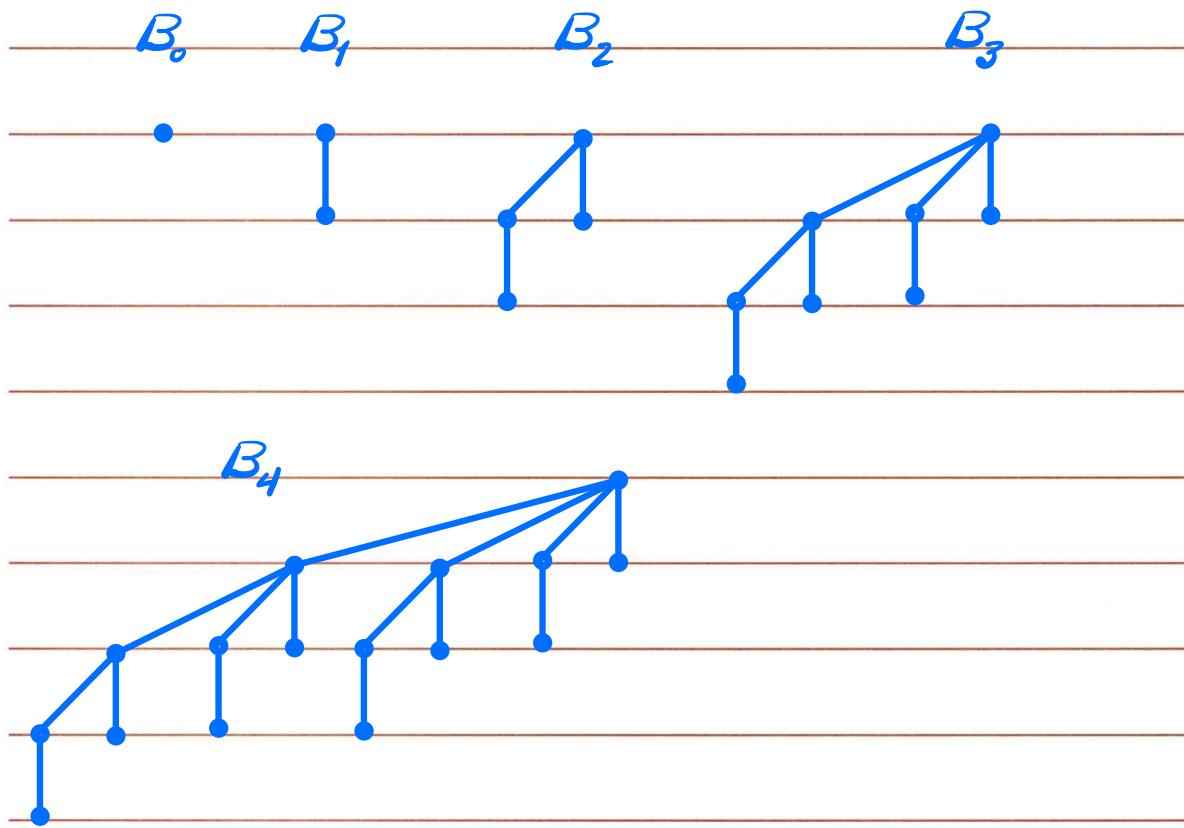
- More through the rest of the  $(n-k)$  elements.  
If we encounter an element larger than the element at the top of our minheap, swap them.

This involves extract-min and insert ( $O(\lg k)$ )

$$\text{Overall complexity} = O(k) + O((n-k)\lg k) = O(n\lg k)$$

Def. A binomial tree  $B_k$  is an ordered tree defined recursively

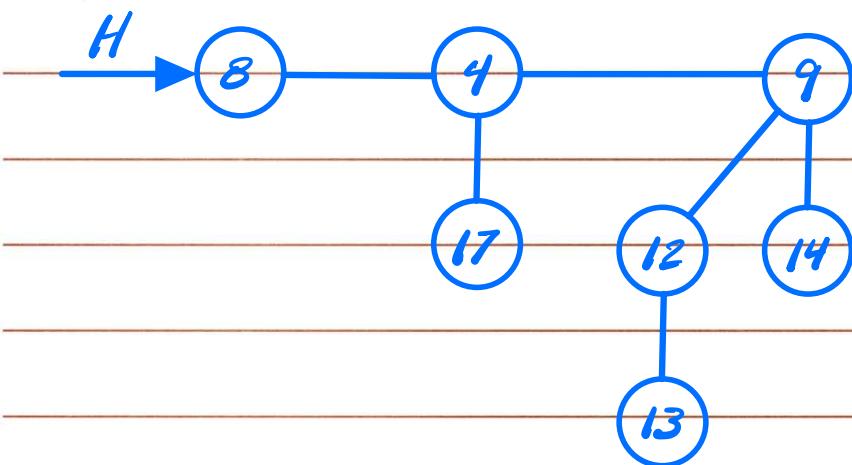
- Binomial tree  $B_0$  consists of one node
- Binomial tree  $B_k$  consists of  $\geq 2$  binomial trees  $B_{k-1}$  that are linked together such that root of one is the leftmost child of the root of the other.



Def. A binomial heap  $H$  is a set of binomial trees that satisfies the following properties:

1- Each binomial tree in  $H$  obeys the min-heap property.

2- For any non-negative integer  $k$ , there is at most one binomial tree in  $H$  whose root has degree  $k$ .

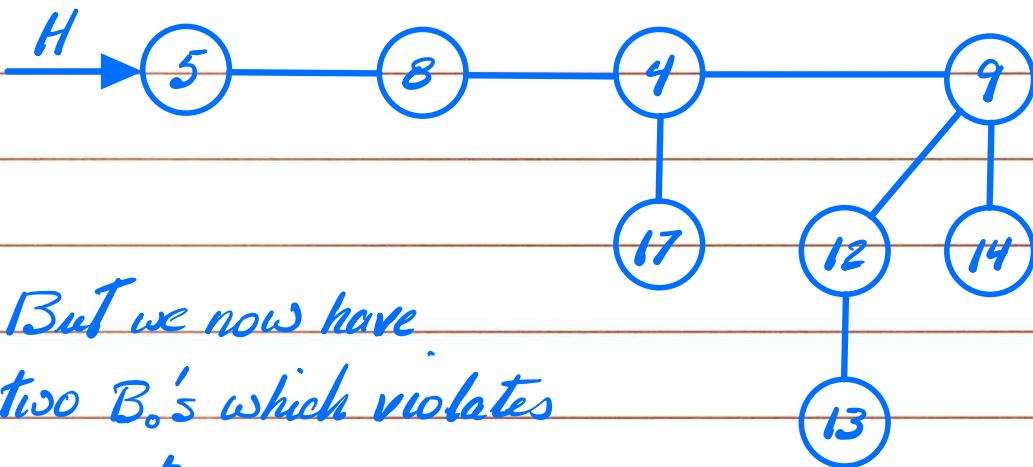


Find-min takes  $O(\lg n)$

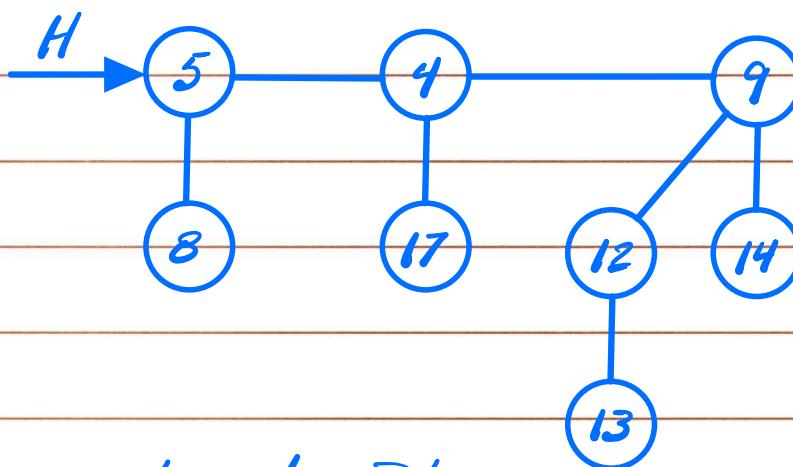
Insert

ex. insert

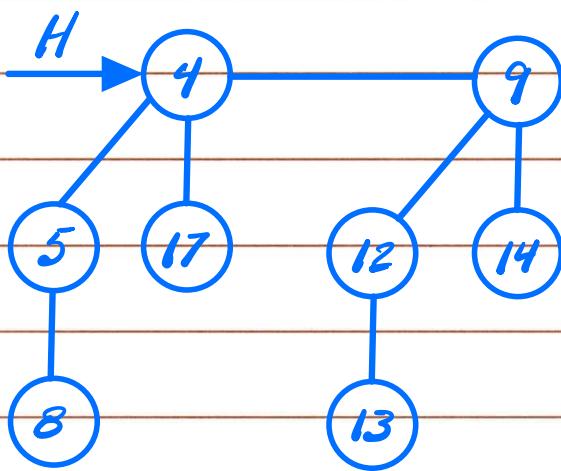
5



But we now have  
two B.'s which violates  
property #2.



Now we have two B.'s...

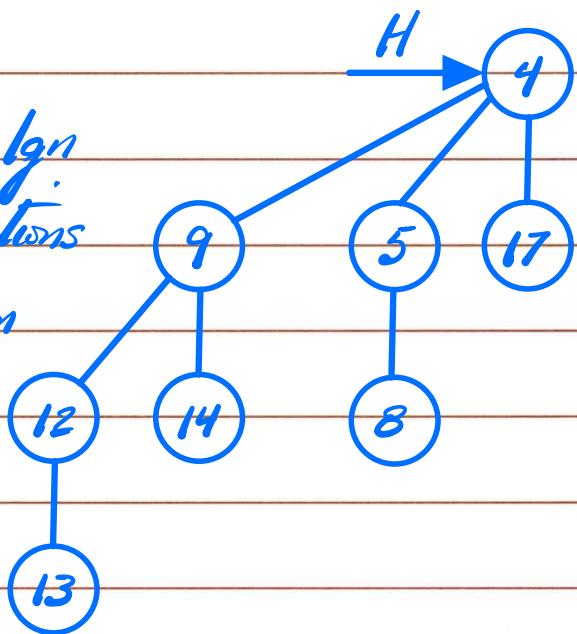


Now we have two  $B_2$ 's...

- Insert requires at most  $\lg n$  binomial tree merge operations

- Each tree merge operation takes  $O(1)$

- So the worst case complexity of the insert operation in a binomial heap is  $O(\lg n)$



## Amortized Cost Analysis

Ex. 1 Starting from an empty stack  
for  $i=1$  to  $n$

Push or Pop  
endfor

Push and Pop take  $O(1)$  each  
Worst case runtime complexity =  $O(n)$

Ex. 2 Starting from an empty stack  
for  $i=1$  to  $n$

Push or Pop or Multipop  
endfor

Push and Pop take  $O(1)$  each (worst case)  
Multipop takes  $O(n)$  (worst case)

Worst case runtime complexity =  $O(n^2)$

But is  $O(n^2)$  a tight upper bound?

## Aggregate Analysis

- Specify the set of operations involved.
- Show that a sequence of  $n$  operations (for all  $n$ ) takes worst case time  $T(n)$  total.
- In the worst case, the amortized cost (average cost) per operation will be  $T(n)/n$

Back to Ex. 2

Observation 1 : Multipop takes  $O(n)$  time if there are  $n$  elements pushed on the stack.

I.e. The total number of pops cannot be greater than the total number of pushes.

Observation 2 : We can only do  $O(n)$  pushes

So the sequence of  $n$  Push, Pop or Multipop operations will take  $O(n)$  time in the worst case

$\therefore$  Amortized cost of each operation =  $\frac{O(n)}{n} = O(1)$

Ex. 2 Starting from an empty stack  
for  $i=1$  to  $n$

Push or Pop or Multipop  
endfor

Push takes  $O(1)$  amortized

Pop "  $O(1)$  "

Multipop "  $O(1)$  "

worst case runtime complexity =  $\Theta(n)$

## Accounting Method

- Assign different charges to different operations.
- If the charge for an operation exceeds its actual cost, the excess is stored as credit.
- The credit can later help pay for operations whose actual cost is higher than their amortized cost.
- Total credit at any time =  
$$\frac{\text{Total amortized cost} - \text{Total actual cost}}{\text{total charges}}$$
- Total credit can never be negative.

Ex. 2 Starting from an empty stack  
for  $i = 1$  to  $n$

Push or Pop or Multipop  
endfor

try #1: assign a charge of 1 to each operation

<u>Op.</u>	<u>charge</u>	<u>Actual Cost</u>	<u>Total Credit</u>
Push	1	1	0
Push	1	1	0
Multipop	1	2	-1

try #1: assign charges as follows:

Push	2	$\rightarrow O(1)$
Pop	0	$\rightarrow O(1)$
Multipop	0	$\rightarrow O(1)$

<u>Op.</u>	<u>charge</u>	<u>Actual Cost</u>	<u>Total Credit</u>
Push	2	1	1
Push	2	1	2
Push	2	1	3
Multipop	0	3	0

Ex. 2 Starting from an empty stack  
for  $i = 1$  to  $n$

Push or Pop or Multipop  
endfor

Push takes  $O(1)$  amortized

Pop "  $O(1)$  "

Multipop "  $O(1)$  "

worst case runtime complexity =  $\Theta(n)$

## Fibonacci Heaps

- Fibonacci heaps are loosely based on binomial heaps.
- A Fibonacci heap is a collection of min-heap trees similar to binomial heaps. However, trees in a Fibonacci heap are not constrained to be binomial trees.
- Unlike binomial heaps, trees in Fibonacci heaps are not ordered.

See Fibonacci heap animation at:

[www.cs.usfca.edu/~galles/JavascriptVisual/FibonacciHeap.html](http://www.cs.usfca.edu/~galles/JavascriptVisual/FibonacciHeap.html)

	Binary Heap	Binomial Heap	Fibonacci Heap
Find-Min	$O(1)$	$O(\lg n)$	$O(1)$
Insert	$O(\lg n)$	$O(\lg n)$	$O(1)$
Extract-Min	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
Delete	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
Decrease-key	$O(\lg n)$	$O(\lg n)$	$O(1)$
Merge	$O(n)$	$O(\lg n)$	$O(1)$
Construct	$O(n)$	$O(n)$	$O(n)$

Worst Case Costs

Amortized costs

3. 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?

5. Suppose you have two 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  $O(n \log n)$  and explain why it is correct. Give a brief explanation for why your algorithm has the required running time. For this problem, do not use the fact that heaps are implemented as arrays; treat them as abstract data types.

























































