

Amortized Analysis Binary Heaps Binomial Heaps

Amortized Analysis



The technique was introduced
by R. Tarjan in 1985.

Amortized Analysis

In a sequence of operations the worst case does not occur often in each operation - some operations may be cheap, some may be expensive.

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

When same operation takes different times, how can we accurately calculate the runtime complexity?

Amortized Analysis

Amortized analysis gives the average performance (over time) of each operation in the worst case.

The amortized cost per operation for a sequence of n operations is the total cost of the operations divided by n .

It requires that the total real cost of the sequence should be bounded by the total of the amortized costs of all the operations.

Amortized Analysis

Three methods are used in amortized analysis

1. Aggregate Method (or brute force)
2. Accounting Method (or the banker's method)
3. Potential Method (or the physicist's method)

We won't use a potential method in this class.

Unbounded Array

Thinking about it abstractly, an unbounded array should be like an array in the sense that we can get and set the value of an arbitrary element via its index.

We should also be able to add a new element to the end of the array, and delete an element from the end of the array.

Unbounded Array

The general implementation strategy:

we maintain an array of a fixed length limit and an internal index size which tracks how many elements are actually used in the array. When we add a new element we increment size, when we remove an element we decrement size.

The tricky issue is how to proceed when we are already at the limit and want to add another element.

Unbounded Array

At that point, we allocate a new array twice as large and copy the elements we already have to the new array.

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

In order to make the analysis as concrete as possible, we will count the number of inserts and the number of copyings. We won't analyze deletions.

insert	old size	new size	copy
1	1	-	-
2	1	2	1
3	2	4	2
4	4	-	-
5	4	8	4
6	8	-	-
7	8	-	-
8	8	-	-
9	8	16	8

Aggregate Method

The table shows that 9 inserts require $1 + 2 + 4 + 8$ copy operations.

Let us generalize the pattern.

Assume we start with the array of size 1 and make $2^n + 1$ inserts.

These inserts will require

$$1 + 2 + 4 + \dots + 2^n = 2^{n+1} - 1$$

copy operations.

Thus, the total work is $(2^n + 1) + (2^{n+1} - 1)$.

Aggregate Method

The total work is $(2^n + 1) + (2^{n+1} - 1) = 3 \cdot 2^n$

Next we compute the average cost per insert.

$$\lim_{n \rightarrow \infty} \frac{3 \times 2^n}{2^n + 1} = 3$$

We say that the amortized cost of insertion is constant, $O(3)$.

Such method of analysis is called an aggregate method.

The Accounting Method

The aggregate method seeks an upper bound on the overall running time of a sequence of operations.

The accounting method seeks a payment for each individual operation.

Intuitively, we maintain a bank account and each operation is charged to it.

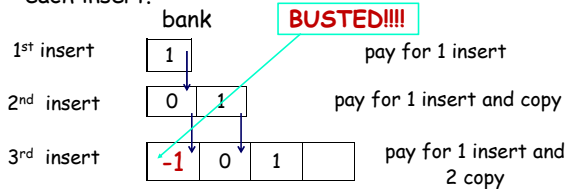
Some operations are charged very little but also generate a surplus. Others, drain the savings.

The balance in the bank account must always remain positive.

Unbounded Array

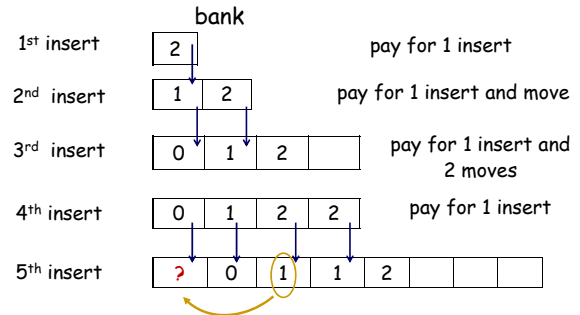
We will assign a dollar token to each operation. Say it costs 1 token to insert an element and 1 token to move it when the table is doubled.

So, we have to assigned at least 2 tokens to each insert.

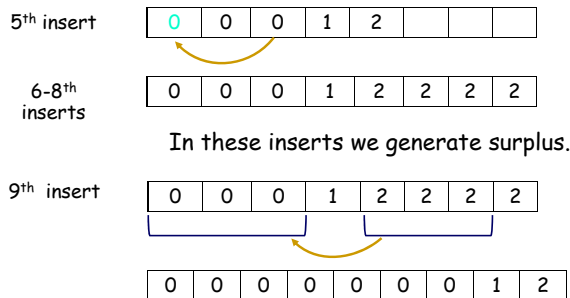


Unbounded Array

Let us assign 3 tokens to each insert.



There are enough money in the bank account to bail out the busted element.



In short, we run a 'ponzi' scheme...

Binary counter

Given n binary numbers each of $\log n$ bits long

00...000
00...001
00...010
00...011
00...100
...
111...111

Let the cost of incrementing a binary number is the number of bits flipped. What is the amortized cost per increment?

Binary counter

Consider 3-bit numbers

	# of flips
000	1
001	2
010	1
011	3
100	1
101	2
110	1
111	3

Clearly, in the worst-case all bits are flipped. Thus, the cost per increment is $O(\log n)$

Amortized Cost

The aggregate method.

	# of flips
000	1
001	2
010	1
011	3
100	1
101	2
110	1
111	3

Consider the least significant bit.

Each time we increment a number, that bit is changed. Thus, the number of times this bit changes is n .

Consider the next significant bit. It changes $n/2$ times, the next one - $n/4$, and so on.

Amortized Cost

Thus the total cost is given by

$$\begin{aligned} n + n/2 + n/4 + \dots + 2 &= \\ &= n (1 + 1/2 + 1/4 + \dots + 2/n) \leq \\ &\leq n \sum_{k=0}^{\infty} \frac{1}{2^k} = 2n \end{aligned}$$

It follows the amortized cost per increment is a constant $O(2)$.

Amortized Cost

The accounting method.

	1 → 0 flips
000	0
001	1
010	0
011	2
100	0
101	1
110	0
111	3

The key point is that each increment has exactly one 0 → 1 flip.

and different increments have different numbers of 1 → 0 flips.

Amortized Cost

The accounting method.

	bank
000	0
001	1
010	1
011	2
100	1
101	2
110	2
111	3

Every time you flip 0 → 1, pay the actual cost of 1, plus put 1 into a bank.

Now, every time you flip a 1 → 0, use the money in the bank to pay for the flip.

Clearly, by design, our bank account cannot go negative.

Another Binary Counter



Let us assume that the cost of a flip is 2^k to flip k-th bit.

The least significant bit has $k=0$, and the most significant bit has $k=\log n$.

What is the amortized cost per increment? Use the aggregate method.

Solution

Start with the least significant bit.

Binary Heaps

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

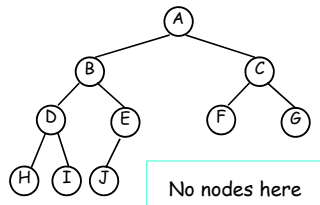
the min-heap property: the value of each node is greater than or equal to the value of its parent

the max-heap property: the value of each node is less than or equal to the value of its parent,



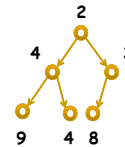
Complete Binary Tree

completely filled, except the bottom level that is filled from left to right



Binary Heap

The word "heap" will always refer to a min-heap, unless otherwise noted.



Not a Heap



Binary Heap Invariants

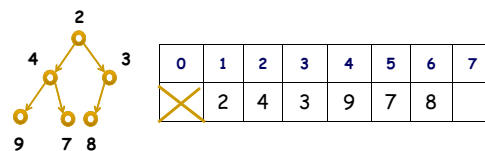
1. Structure Property
2. Ordering Property

Heap Operations

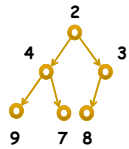
insert
deleteMin
decreaseKey
build
meld

Implementation

A heap tree is uniquely represented by storing it in an array.



Implementation



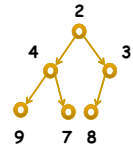
0	1	2	3	4	5	6	7
X	2	4	3	9	7	8	

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

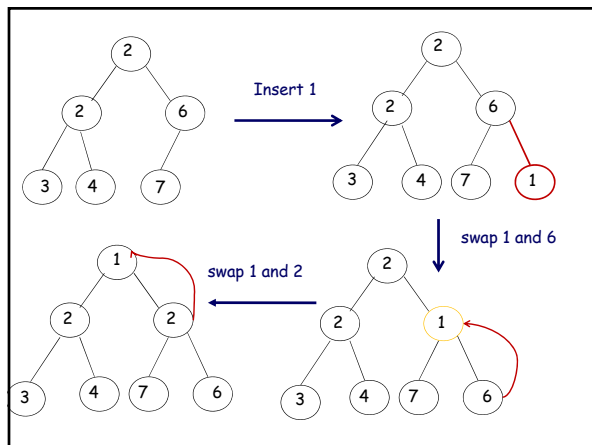
Insert

The new element is initially appended to the end of the heap array.



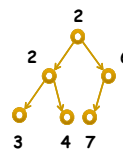
This will preserve the structure property.

Then we percolate it up by swapping positions with the parent, if it's necessary.



Implementation

Insert 1.



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

Insert



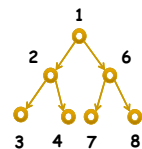
What is the worst-case complexity of `insert()`?

deleteMin

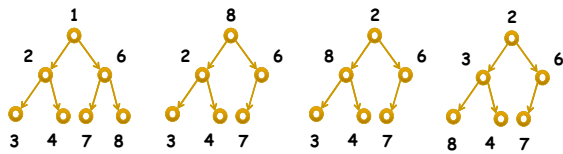
The minimum element can be found at the root of the heap, which is the first element of the array.

Clearly, we cannot delete it.

We move the last element of the heap to the root and then restore the heap property by percolating down.

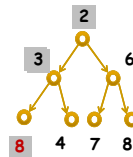


deleteMin



Implementation

delMin



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

deleteMin



What is the worst-case complexity of `deleteMin()`?

Sorting

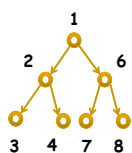


How would you sort using a heap?

HEAPSORT

Run delMin n-times
 $O(n \log n)$

in-place
nonstable

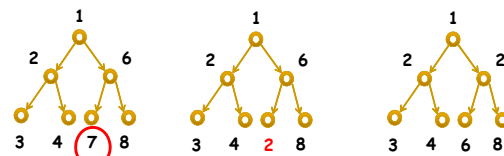


0	1	2	3	4	5	6	7
	1	2	6	3	4	7	8
	2	3	6	8	4	7	1
	3	4	6	8	7	2	1
	4	8	6	7	3	2	1

decreaseKey

We change the key (value) of one of the heap elements.

To restore a heap property we need to **percolate up** this item.



Building a heap

Given an array - turn it into a heap.

There are two algorithms:

- 1) by insertion $O(n \log n)$
- 2) heapify $O(n)$

Heapify

We insert all the elements into an array in any order.

Next, starting at position $n/2$ and working toward position 1, we push each element down the heap by swapping it with its smallest child.



Heapify

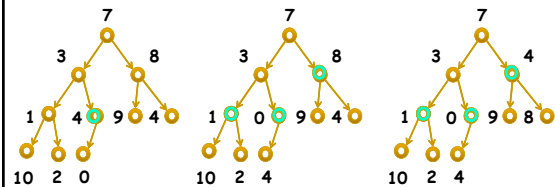
Apply buildHeap operation to the following set of data. Show all intermediate steps.

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

Heapify

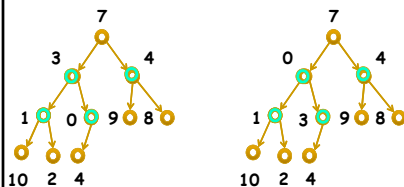
swap 4 and 0

swap 8 and 4



Heapify

swap 3 and 0

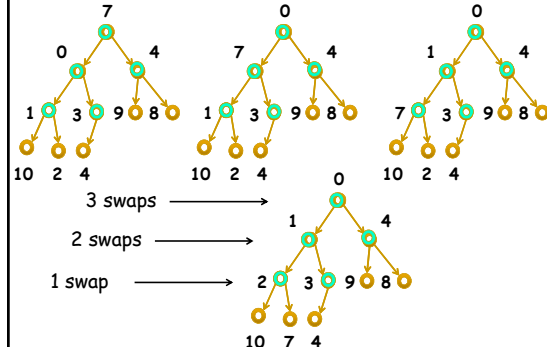



Heapify

swap 7 and 0

swap 7 and 1

swap 7 and 2





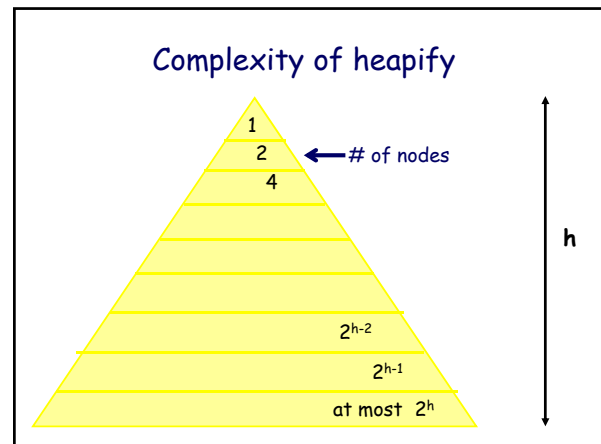
What is the worst-case complexity of **heapify**?

Quick analysis:

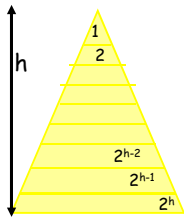
during the algorithm execution at most $n/2$ heap elements percolate down the heap.

Thus, $O(n \log n)$ in the worst-case.

A good upper bound, but it's not asymptotically tight.



Complexity of heapify



We will count the max number of swaps at each level

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

What is total work ?

Complexity of heapify

To compute the total work we multiply the number of swaps by the number of nodes on each level:

$$T(n) = \sum_{k=1}^h k 2^{h-k}$$

where n is the total number of nodes

Note, $2^h = O(n)$

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

Complexity of heapify

$$\sum_{k=1}^h k 2^{h-k} = 2^h \sum_{k=1}^h \frac{k}{2^k} \leq 2^h \sum_{k=1}^{\infty} \frac{k}{2^k} = 2^h 2 = O(n)$$

Here

$$x = \sum_{k=1}^{\infty} \frac{k}{2^k} = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \dots$$

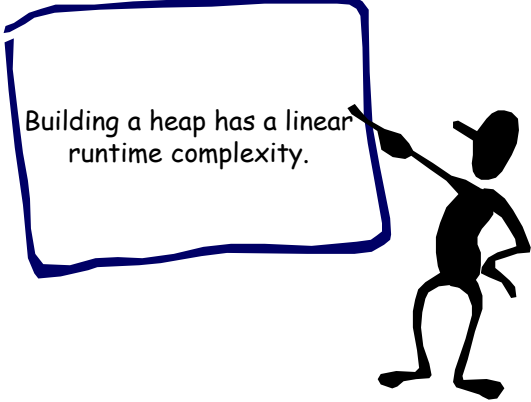
Compute

$$\frac{x}{2} = \frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \dots$$

Subtract

$$x - \frac{x}{2} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots = 1$$

It follows, $x = 2$.



Building a heap has a linear runtime complexity.



Devise the algorithm of **merging** two heaps into one.
What is its running time?

Priority Queues

A **priority queue** is a data structure which supports two basic operations: insert a new item according its priority, and remove the item with the highest priority.

It's implemented as a heap.

Binary Heaps

	Complexity
findMin	$\Theta(1)$
deleteMin	$\Theta(\log n)$
insert	$\Theta(\log n)$
decreaseKey	$\Theta(\log n)$
build	$\Theta(n)$
merge (meld)	$\Theta(n)$

Efficient searching is not supported

BINOMIAL HEAPS

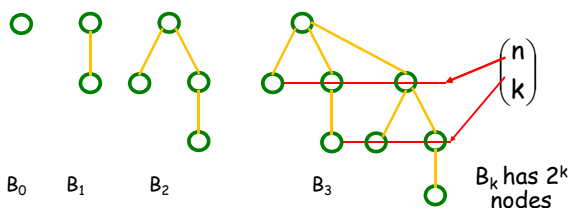


We describe a different kind of heap that has a slight improvement over the binary heap. This data structure was introduced by Vuillemin in 1978.

Binomial Trees

The binomial tree of rank k , B_k , is defined recursively as follows

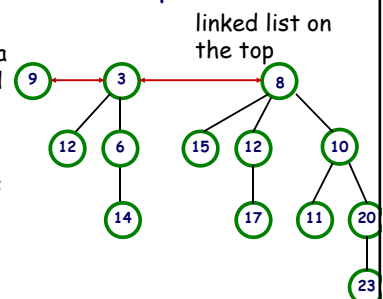
1. B_0 is a single node
2. B_k is formed by joining two B_{k-1} trees



Binomial Heaps

A binomial heap is a collection (a linked list) of at most $\lceil \log n \rceil$ binomial trees in increasing order of size where each tree has a heap ordering property.

In a binomial heap there is at most one binomial tree of any given rank.



Binomial Heaps and Binary Expansion

To store 11 items we need B_3 , B_1 and B_0 binomial trees. It follows from the binary expansion:

$$11_{10} = 1011_2$$

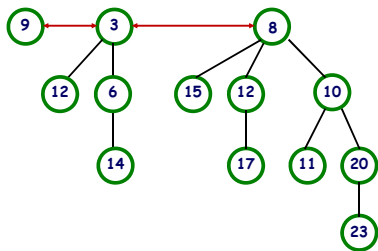


How many binomial trees does a binomial heap with 25 elements contain?
What are the ranks of those trees?



How can we find the minimum element in a binomial heap?

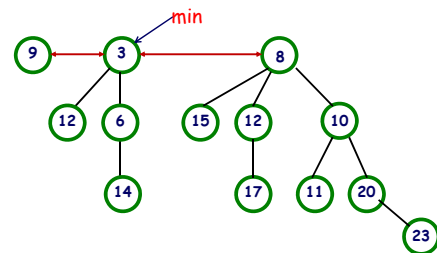
How long does it take?



FindMin

Look at the root of each binomial tree for the minimum - $O(\log n)$

Store the pointer to the minimum root - $O(1)$

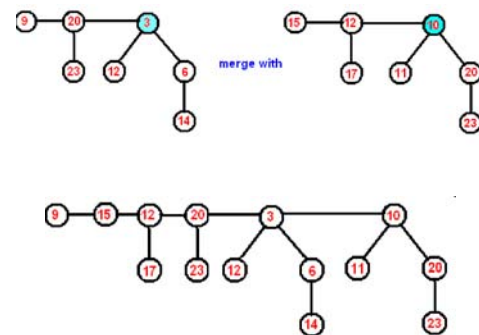


Merging Binomial Heaps

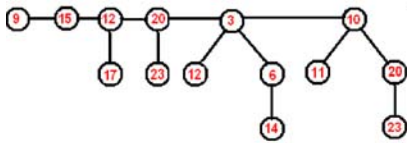
Devise an algorithm for merging two binomial heaps and discuss its complexity.

Note, binary heaps are complete binary trees, and two complete binary trees cannot easily be linked to one another.

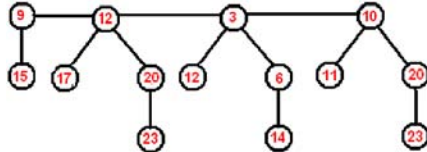
Merging Binomial Heaps



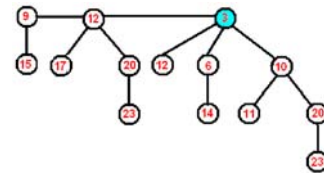
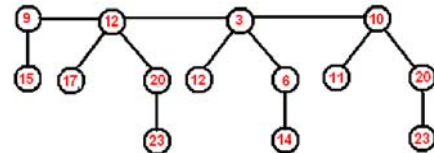
Merging Binomial Trees



Merging trees of the same size requires $O(1)$ time.
Make the smaller root the child of the larger root.



Merging Binomial Trees



Merging Binomial Heaps: binary addition

Merge $B_0B_1B_2B_4$ with B_1B_4 to get $B_0B_3B_5$

We do a binary addition:

```

10111
 10010
-----
101001

```

We need to merge at most $\log n$ binomial trees, and each merge takes $O(1)$ time, so the whole operation takes $O(\log n)$ time

Worst-case complexity of merge.

Think of a binomial heap $B_0B_1B_2B_3\dots B_{\log n}$

First merge - $B_1B_2B_3\dots B_{\log n}$

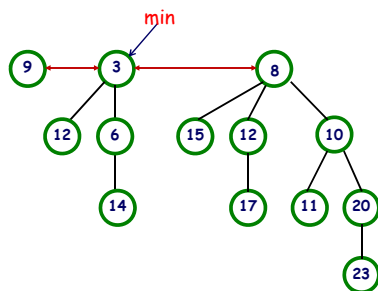
Second merge - $B_2B_3\dots B_{\log n}$

Third merge - $B_3B_4\dots B_{\log n}$ and so on...

each merge takes $O(1)$ time

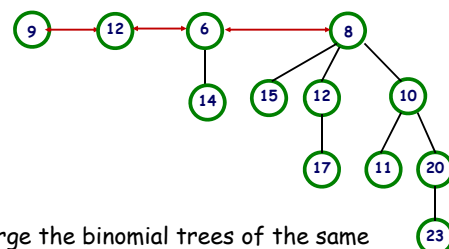
so the whole operation takes $O(\log n)$ time

Devise an algorithm for `deleteMin()` and discuss its complexity.



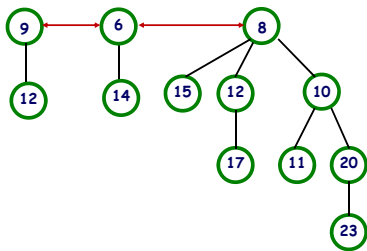
1. Find the binomial tree that contains the min.
2. Delete the root and move subtrees to top list.

Deleting the root of B_k yields B_0, B_1, \dots, B_{k-1} .



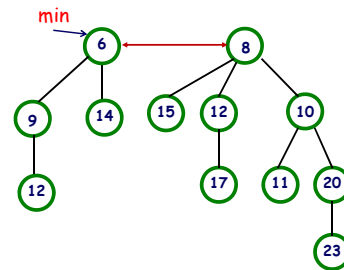
3. Merge the binomial trees of the same rank.

3. Merge the binomial trees of the same rank.



3. Merge the binomial trees of the same rank.

4. Set a pointer to the new min.



Worst-case complexity of delMin?

Same as merging two heaps!

It takes $O(\log n)$ time

Insertion

How do we insert into a binomial heap?

Essentially this is a union of two binomial heaps: one of size 1 and other of the size n .

Therefore it takes $O(\log n)$ in the worst case.

...but not all inserts are so slow...



Insertion

What is the **amortized cost** of insertion?

Use the accounting method.

Each insertion requires only two tokens. One to create a single binomial tree. The other to pay for the future merge.

This is as in the binary counter.

Insertion: 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.

If n is known in advance, 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.

DEMO

<https://www.cs.usfca.edu/~galles/visualization/BinomialQueue.html>

FIBONACCI HEAPS

Fredman and Tarjan, 1987



Idea: relaxed binomial heaps

Goal: decreaseKey in $O(1)$

FIBONACCI HEAPS

We want to incorporate the decreaseKey operation into the binomial heap. How can we do this?

One idea to try is simply this. To decrease the key of a node, we simply change its key value. If its new key is less than that of its parent, we swap the node with its parent. We repeat this till we get to the top of the binomial tree, and stop.

Its running time is $O(\log n)$.

But we're striving for $O(1)$ running time.

FIBONACCI HEAPS

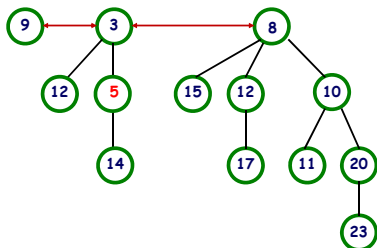
Better idea:

take the node you want to decrease, and change its key, and disconnect it and its entire subtree from where it is, and attach it to the tree root list.

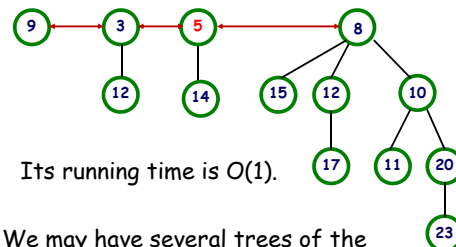
This is clearly $O(1)$ time.

decreaseKey: example

Suppose we want to change 6 to 5.



decreaseKey: example



Its running time is $O(1)$.

We may have several trees of the same rank. We will fix the heap when deleteMin is called.

decreaseKey

The problem is that we can no longer prove the bound on the time of deleteMin. That heap may contain more than $\log(n)$ binomial trees.

There's a clever way to fix this devised by Fredman and Tarjan, the complexity of decreaseKey will be $O(1)$ amortized. This is the Fibonacci heap algorithm.

The algorithm is outside of the scope of this course.

Insertion



How we insert into the Fibonacci heap?

Just add a single node to the top level.

Do not merge binomial trees!

We may have several trees of the same rank. We will fix the heap when deleteMin is called.

Clearly, lazy insertion runs in $O(1)$ time in the worst-case.

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(meld)	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$ (ac)

ac - amortized cost.