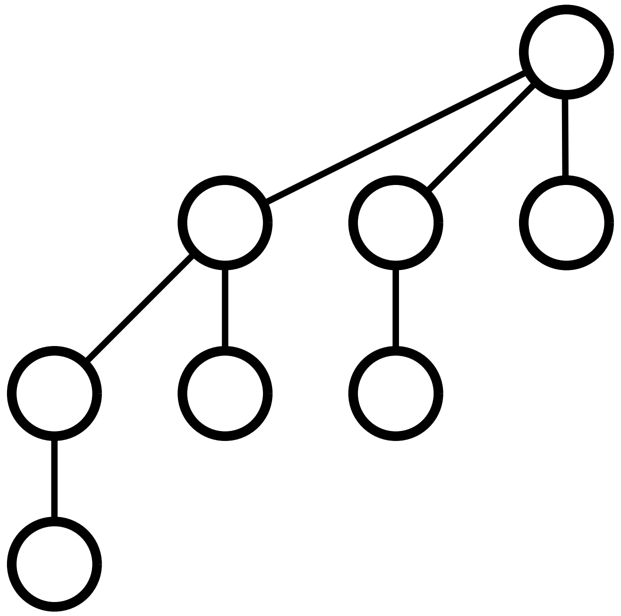
Binomial Heaps

Introduction and Worst-Case Analysis

Amortized Analysis of Insert

Amortized Analysis of Lazy-Union



Priority Queues

Abstract data typ: manage a set of elements with keys (their priority) under the operations *insert, minimum, deleteMinimum*, and optionally *decreaseKey, remove* and *merge*.

Implementations:

	deleteMin	decreaseKey	insert	build
Binary heaps	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Balanced Search Trees	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n \log n)$
Fibonaccis Heaps	$O(\log n)^*$	$O(1)^*$	$O(1)$	$O(n)$

* amortised

Priority Queues

Abstract data typ: manage a set of elements with keys (their priority) under the operations *insert, minimum, deleteMinimum*, and optionally *decreaseKey, remove* and *merge*.

Implementations:

	deleteMin	decreaseKey	insert	build
Binary heaps	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Balanced Search Trees	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n \log n)$
Fibonaccis Heaps	$O(\log n)^*$	$O(1)^*$	$O(1)$	$O(n)$

* amortised

Runtime for Dijkstras Algorithm: $O(m + n \log n)$ using Fibonacci heaps

Mergeable Priority Queues

Abstract data typ: manage a set of elements with keys (their priority) under the operations *insert, minimum, deleteMinimum*, and optionally *decreaseKey, remove* and *merge*.

Implementations:

	deleteMin	decreaseKey	insert	build
Binary heaps	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Balanced Search Trees	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n \log n)$
Fibonaccis Heaps	$O(\log n)^*$	$O(1)^*$	$O(1)$	$O(n)$

* amortised

Goal today: Efficiently mergeable heaps

Mergeable Priority Queues

Abstract data typ: manage a set of elements with keys (their priority) under the operations *insert, minimum, deleteMinimum*, and optionally *decreaseKey, remove* and *merge*.

Implementations:

	deleteMin	decreaseKey	insert	merge
Binary heaps	$O(\log n)$	$O(\log n)$	$O(\log n)$	
Binomial heaps				
Fibonaccis Heaps	$O(\log n)^*$	$O(1)^*$	$O(1)$	

* amortised

Goal today: Efficiently mergeable heaps

Mergeable Priority Queues

Abstract data typ: manage a set of elements with keys (their priority) under the operations *insert, minimum, deleteMinimum*, and optionally *decreaseKey, remove* and *merge*.

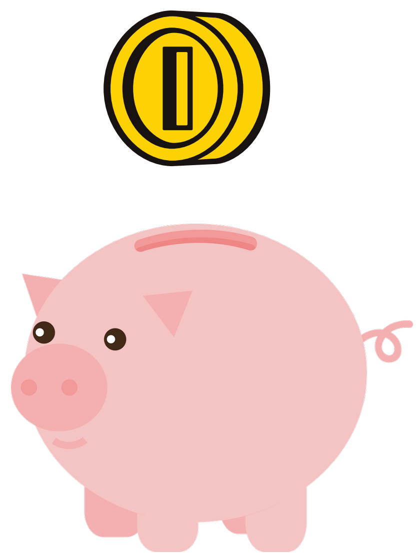
Implementations:

	deleteMin	decreaseKey	insert	merge
Binary heaps	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Binomial heaps	$O(\log n)^*$	$O(\log n)$	$O(1)$	$O(1)$
Fibonaccis Heaps	$O(\log n)^*$	$O(1)^*$	$O(1)$	$O(1)$

* amortised

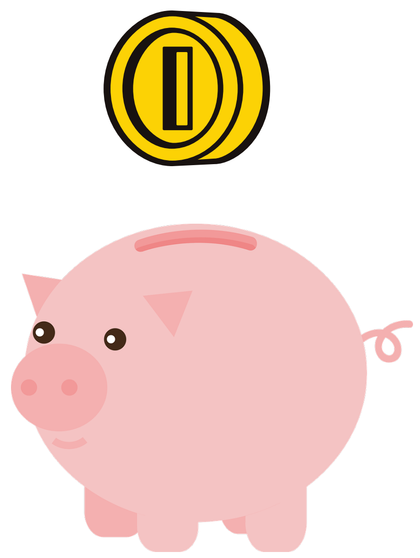
Goal today: Efficiently mergeable heaps

Short Recap: Amortized Analysis



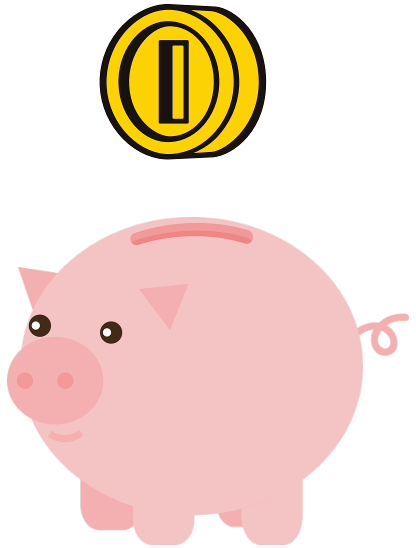
Short Recap: Amortized Analysis

- $D_i =$ data structure after ith operation
- $c_i =$ actual cost of ith operation
- $\hat{c}_i =$ amortized cost of ith operation = coins



Short Recap: Amortized Analysis

- D_i = data structure after ith operation
- c_i = actual cost of ith operation
- $\hat{c}_i$ = amortized cost of ith operation = coins



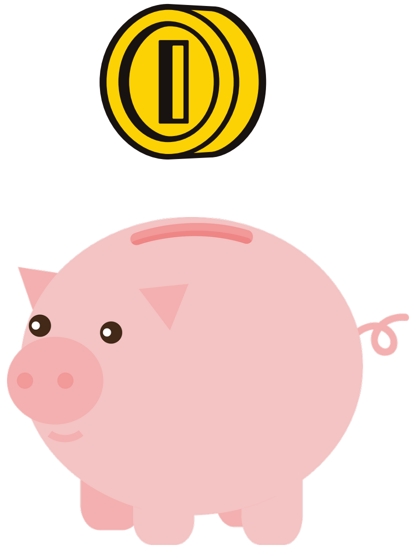
accounting method:

- assign amortized cost ("coins") to every operation
- check whether enough coins are saved to pay for all operations:

for all j : $\sum_{i=1}^{j} \hat{c}_i - \sum_{i=1}^{j} c_i \geq 0$ (coins saved in step i: $\hat{c}_i - c_i$)

Short Recap: Amortized Analysis

- $D_i =$ data structure after ith operation
- $c_i =$ actual cost of ith operation
- $\hat{c}_i =$ amortized cost of ith operation = coins



accounting method:

- assign amortized cost ("coins") to every operation
- check whether enough coins are saved to pay for all operations:

for all j : $\sum_{i=1}^{j} \hat{c}_i - \sum_{i=1}^{j} c_i \geq 0$ (coins saved in step i: $\hat{c}_i - c_i$)

potential method:

- say how many "coins" are saved with data structure D_i: $\Phi(D_i)$ ("potential")
- calculate amortized costs from potential:

$\hat{c}_i = c_i$ + coins saved in step i $= c_i + \Phi(D_i) - \Phi(D_{i-1})$

$= c_i$ + Δ_i (change in potential)

Short Recap: Amortized Analysis (Binary Counter)

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

example: $k = 6$

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1

Short Recap: Amortized Analysis (Binary Counter)

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

example: $k = 6$

flipped bits

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1

Short Recap: Amortized Analysis (Binary Counter)

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0

example: $k = 6$

flipped bits

Short Recap: Amortized Analysis (Binary Counter)

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1

example: $k = 6$

flipped bits

Short Recap: Amortized Analysis (Binary Counter)

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

example: $k = 6$

flipped bits

Short Recap: Amortized Analysis (Binary Counter)

Algorithm. increment a k- bit binary counter

Representation as array. $A[j]$: jth least-significant bit

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	0	0	1	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	0	0	0

example: $k = 6$

flipped bits

costs. number of bits flipped. How many for n increments?

Short Recap: Amortized Analysis (Accounting Method)

actual cost per operation: 1 coin per bit flipped

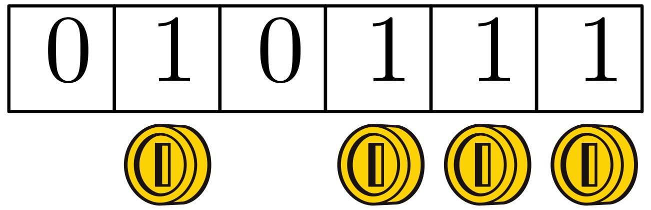
invariant: ???

accounting/amortized cost ($\hat{c}_i$): ???

Short Recap: Amortized Analysis (Accounting Method)

actual cost per operation: 1 coin per bit flipped

invariant: Every 1 of the counter has a coin

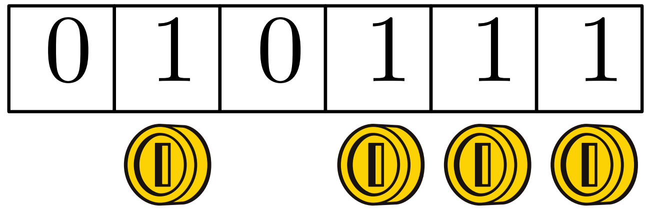


accounting/amortized cost ($\hat{c}_i$): ???

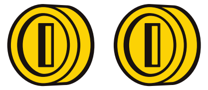
Short Recap: Amortized Analysis (Accounting Method)

actual cost per operation: 1 coin per bit flipped

invariant: Every 1 of the counter has a coin



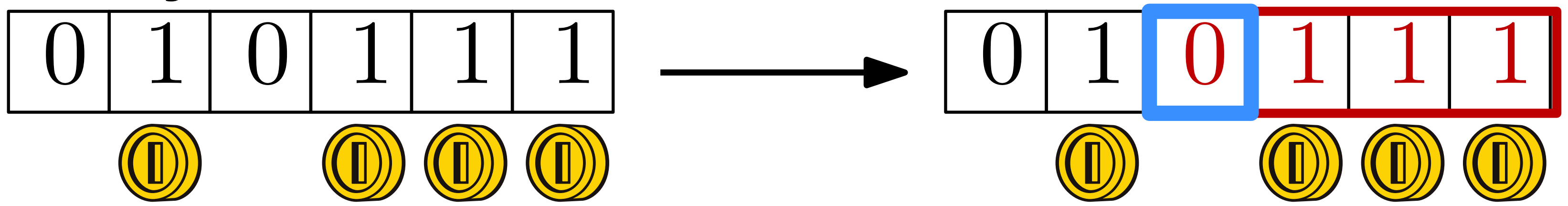
accounting/amortized cost ($\hat{c}_i$):

increment: assign 2 coins ($\hat{c}_i$ = 2) 

Short Recap: Amortized Analysis (Accounting Method)

actual cost per operation: 1 coin per bit flipped

invariant: Every 1 of the counter has a coin



accounting/amortized cost ($\hat{c}_i$):

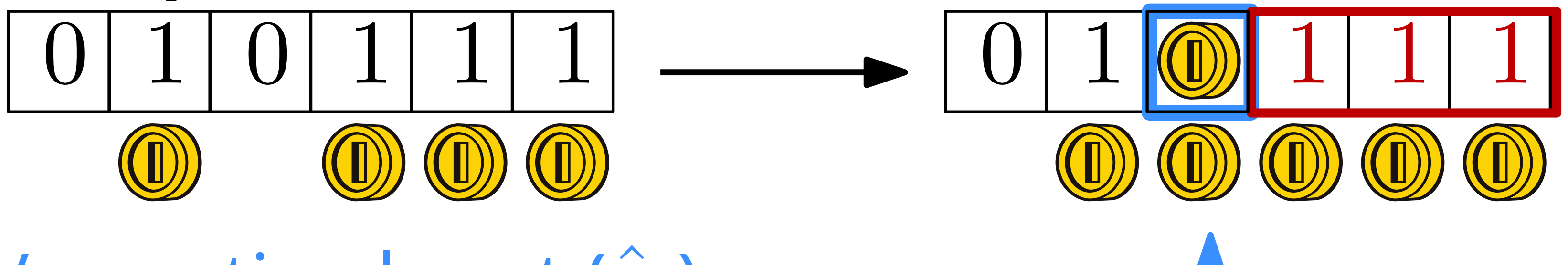
increment: assign 2 coins ($\hat{c}_i$ = 2)

- 1 bit flips from 0 to 1: pay 1 coin to flip 0 to 1 and save 1 coin with new 1

Short Recap: Amortized Analysis (Accounting Method)

actual cost per operation: 1 coin per bit flipped

invariant: Every 1 of the counter has a coin



accounting/amortized cost ($\hat{c}_i$):

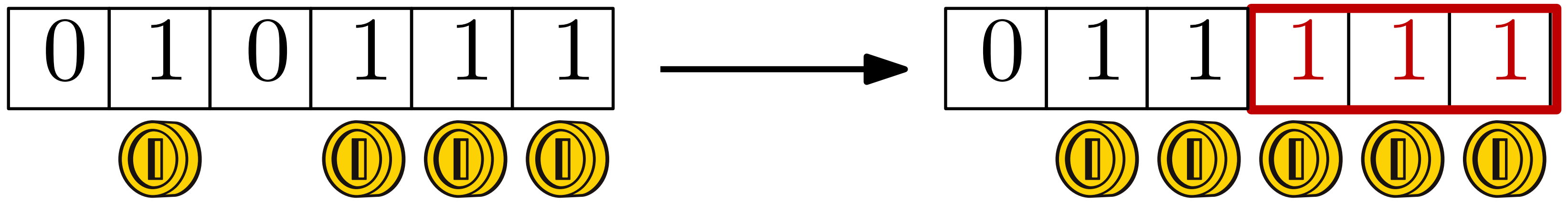
increment: assign 2 coins ($\hat{c}_i$ = 2)

- 1 bit flips from 0 to 1: pay 1 coin to flip 0 to 1 and save 1 coin with new 1

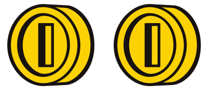
Short Recap: Amortized Analysis (Accounting Method)

actual cost per operation: 1 coin per bit flipped

invariant: Every 1 of the counter has a coin



accounting/amortized cost ($\hat{c}_i$):

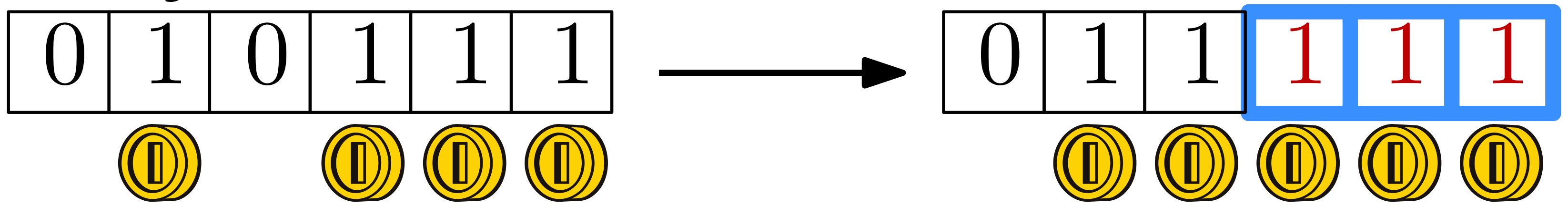
increment: assign 2 coins ($\hat{c}_i$ = 2) 

- 1 bit flips from 0 to 1: pay 1 coin to flip 0 to 1 and save 1 coin with new 1

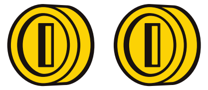
Short Recap: Amortized Analysis (Accounting Method)

actual cost per operation: 1 coin per bit flipped

invariant: Every 1 of the counter has a coin



accounting/amortized cost ($\hat{c}_i$):

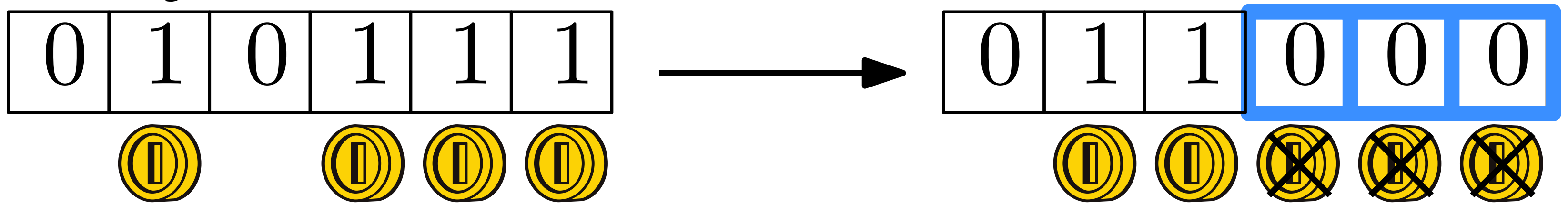
increment: assign 2 coins ($\hat{c}_i$ = 2) 

- 1 bit flips from 0 to 1: pay 1 coin to flip 0 to 1 and save 1 coin with new 1
- to flip a 1 to 0: use saved coin

Short Recap: Amortized Analysis (Accounting Method)

actual cost per operation: 1 coin per bit flipped

invariant: Every 1 of the counter has a coin



accounting/amortized cost ($\hat{c}_i$):

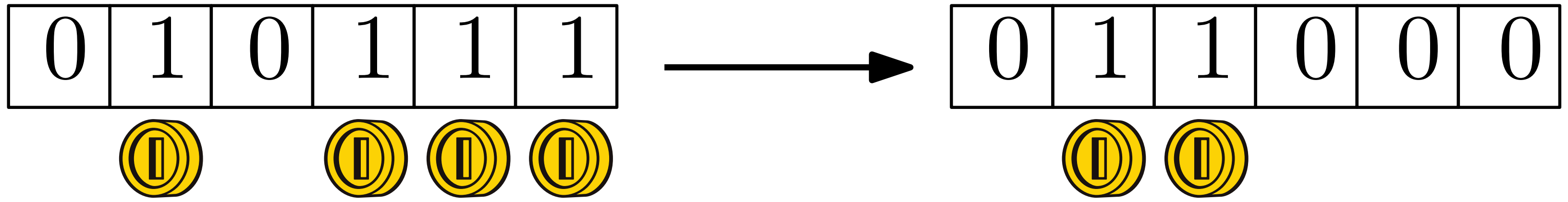
increment: assign 2 coins ($\hat{c}_i$ = 2)

- 1 bit flips from 0 to 1: pay 1 coin to flip 0 to 1 and save 1 coin with new 1
- to flip a 1 to 0: use saved coin

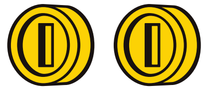
Short Recap: Amortized Analysis (Accounting Method)

actual cost per operation: 1 coin per bit flipped

invariant: Every 1 of the counter has a coin



accounting/amortized cost ($\hat{c}_i$):

increment: assign 2 coins ($\hat{c}_i$ = 2) 

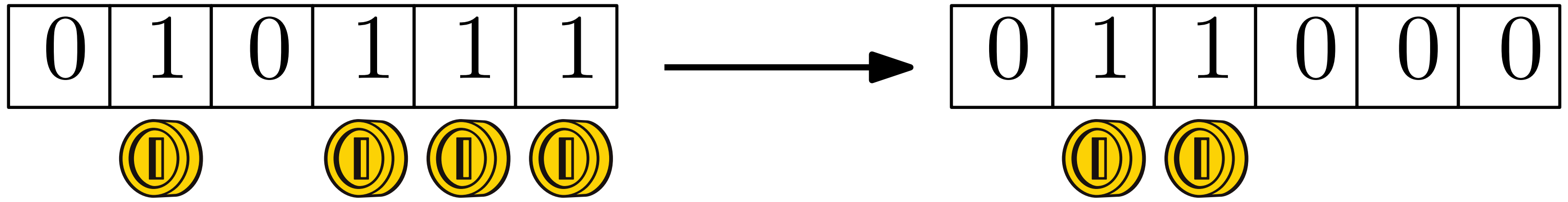
- 1 bit flips from 0 to 1: pay 1 coin to flip 0 to 1 and save 1 coin with new 1
- to flip a 1 to 0: use saved coin

The worst-case running time of a sequence of n increments (starting from 0) is $O(n)$. The amortized running time of one increments is $O(\hat{c}_i) = O(1)$

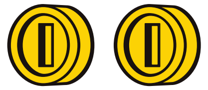
Short Recap: Amortized Analysis (Accounting Method)

actual cost per operation: 1 coin per bit flipped

invariant: Every 1 of the counter has a coin



accounting/amortized cost ($\hat{c}_i$):

increment: assign 2 coins ($\hat{c}_i$ = 2) 

- 1 bit flips from 0 to 1: pay 1 coin to flip 0 to 1 and save 1 coin with new 1
- to flip a 1 to 0: use saved coin

> The worst-case running time of a sequence of n increments (starting from 0) is $O(n)$. The amortized running time of one increments is $O(\hat{c}_i) = O(1)$

- invariant $\Rightarrow$ number of coins in data structure $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$
- amortized cost per operation $\hat{c}_i \leq 2$
- actual total costs $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \leq \sum_{i=1}^{n} 2 = 2n$

Short Recap: Amortized Analysis (Potential Method)

$\Phi(D_i) =$???

Short Recap: Amortized Analysis (Potential Method)

$\Phi(D_i) =$ number of 1-bits

Short Recap: Amortized Analysis (Potential Method)

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$

Short Recap: Amortized Analysis (Potential Method)

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

Short Recap: Amortized Analysis (Potential Method)

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

- c_i = number of bits flipped

Short Recap: Amortized Analysis (Potential Method)

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

0	1	0	1	1	1

- c_i = number of bits flipped

Short Recap: Amortized Analysis (Potential Method)

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

0	1	0	1	1	1

- c_i = number of bits flipped
  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0

Short Recap: Amortized Analysis (Potential Method)

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

0	1	1	1	1	1

- c_i = number of bits flipped

  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0

  = 1 + number of bits flipped from 1 to 0

Short Recap: Amortized Analysis (Potential Method)

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

$\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}$

- c_i = number of bits flipped
  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0
  = 1 + number of bits flipped from 1 to 0

Short Recap: Amortized Analysis (Potential Method)

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

0	1	1	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$

- c_i = number of bits flipped
  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0
  = 1 + number of bits flipped from 1 to 0

Short Recap: Amortized Analysis (Potential Method)

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

0	1	1	0	0	0

- c_i = number of bits flipped
  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0
  = 1 + number of bits flipped from 1 to 0

- change in potential: Δ_i = increase in number of 1s

Short Recap: Amortized Analysis (Potential Method)

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

0	1	1	0	0	0

- c_i = number of bits flipped
  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0
  = 1 + number of bits flipped from 1 to 0

- change in potential: Δ_i = increase in number of 1s
  = number of bits flipped from 0 to 1 - number of bits flipped from 1 to 0

Short Recap: Amortized Analysis (Potential Method)

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

0	1	1	0	0	0

- c_i = number of bits flipped
  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0
  = 1 + number of bits flipped from 1 to 0

- change in potential: Δ_i = increase in number of 1s
  = number of bits flipped from 0 to 1 - number of bits flipped from 1 to 0
  = 1 $-$ number of bits flipped from 1 to 0

Short Recap: Amortized Analysis (Potential Method)

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

0	1	1	0	0	0

- c_i = number of bits flipped
  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0
  = 1 + number of bits flipped from 1 to 0

- change in potential: Δ_i = increase in number of 1s
  = number of bits flipped from 0 to 1 - number of bits flipped from 1 to 0
  = 1 $-$ number of bits flipped from 1 to 0

$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

Short Recap: Amortized Analysis (Potential Method)

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

0	1	1	0	0	0

- c_i = number of bits flipped
  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0
  = 1 + number of bits flipped from 1 to 0

- change in potential: Δ_i = increase in number of 1s
  = number of bits flipped from 0 to 1 - number of bits flipped from 1 to 0
  = 1 $-$ number of bits flipped from 1 to 0

$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
$= (1 +$ number of bits flipped from 1 to 0$) +$
$(1 -$ number of bits flipped from 1 to 0 $)$

Short Recap: Amortized Analysis (Potential Method)

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

amortized cost: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

0	1	1	0	0	0

- c_i = number of bits flipped
  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0
  = 1 + number of bits flipped from 1 to 0

- change in potential: Δ_i = increase in number of 1s
  = number of bits flipped from 0 to 1 - number of bits flipped from 1 to 0
  = 1 $-$ number of bits flipped from 1 to 0

$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

$= (1 +$ number of bits flipped from 1 to 0$) +$

$(1 -$ number of bits flipped from 1 to 0$) \quad = 2$

Short Recap: Amortized Analysis (Potential Method)

$\Phi(D_i) =$ number of 1-bits $= \sum_{j=0}^{k} A[j]$

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i

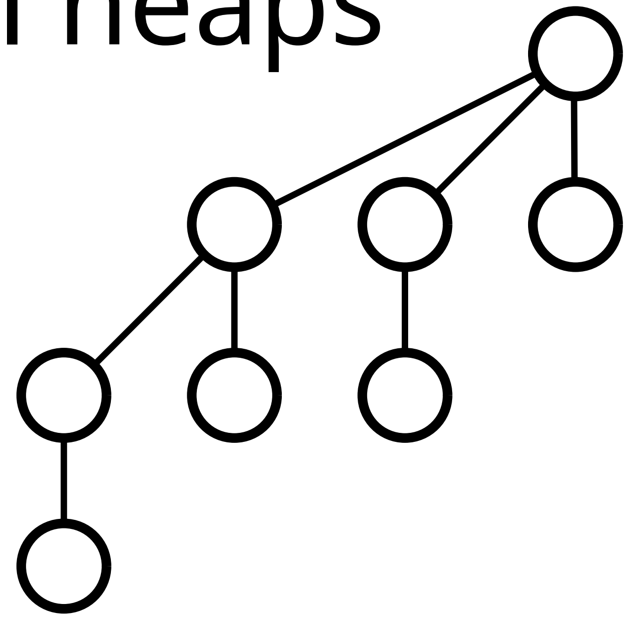
amortized cost: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \; = 2$

- c_i = number of bits flipped
  = number of bits flipped from 0 to 1 + number of bits flipped from 1 to 0

- change in potential: Δ_i = increase in number of 1s
  = number of bits flipped from 0 to 1 - number of bits flipped from 1 to 0
  = 1 $-$ number of bits flipped from 1 to 0

The worst-case running time of a sequence of n increments (starting from 0) is $O(n)$. The amortized running time of one increments is $O(\hat{c}_i) = O(1)$

Binomial trees and Binomial heaps

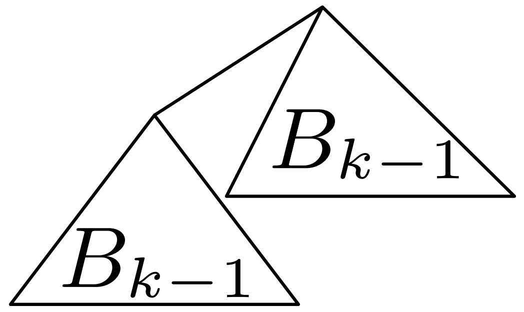


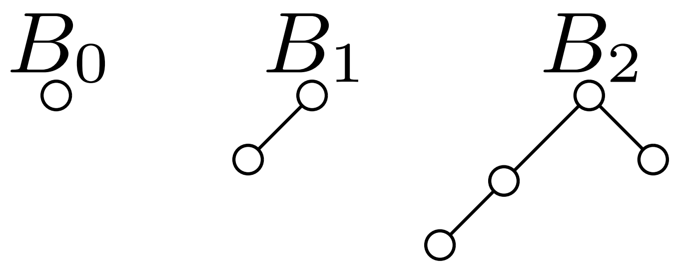
Binomial Trees

Binomial tree: recursively defined

B_0 single node

B_k two B_{k-1} joined by making one a child of the root of the other



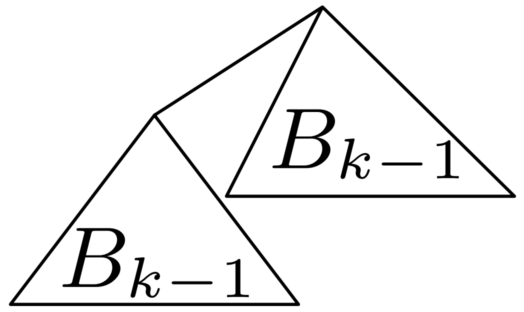


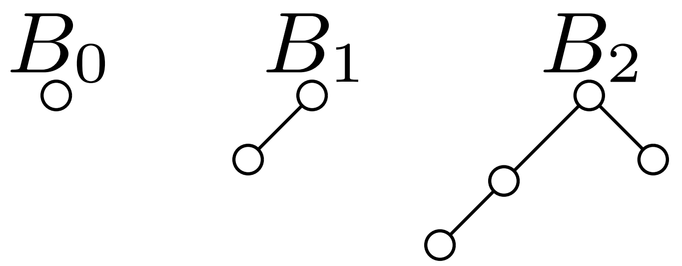
Binomial Trees

Binomial tree: recursively defined

B_0 single node

B_k two B_{k-1} joined by making one a child of the root of the other





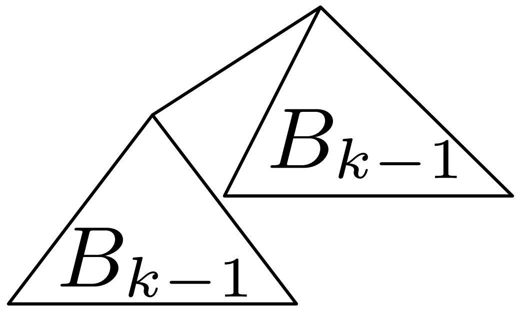
Question: What does B_3 look like?

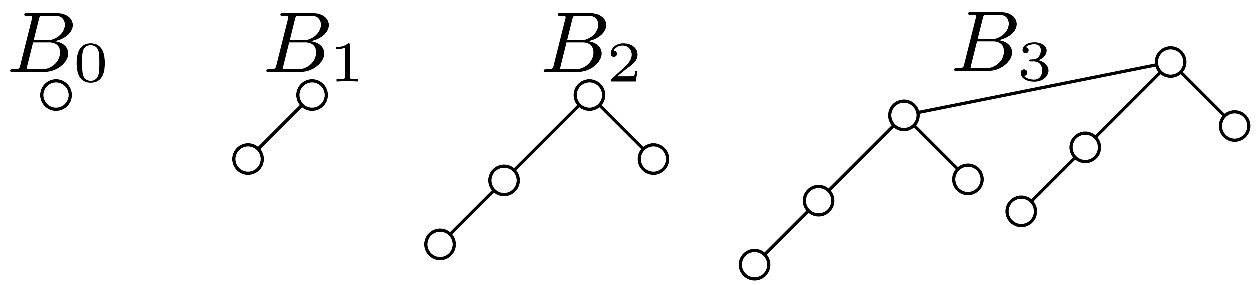
Binomial Trees

Binomial tree: recursively defined

B_0 single node

B_k two B_{k-1} joined by making one a child of the root of the other



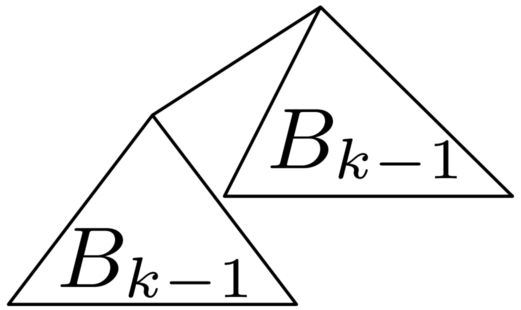


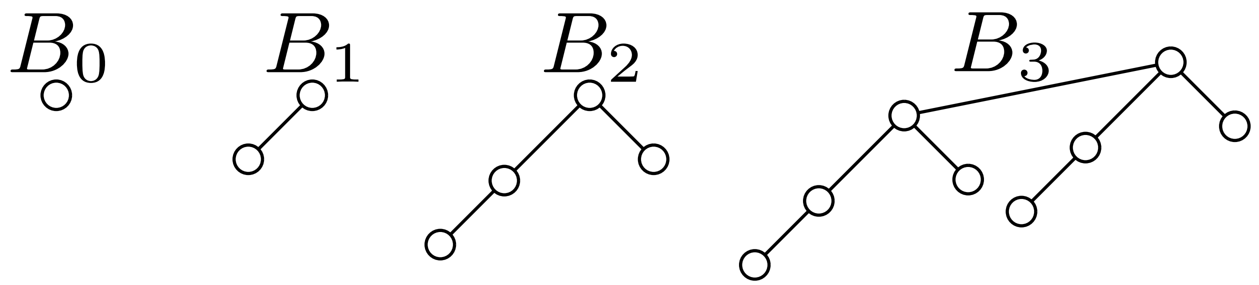
Binomial Trees

Binomial tree: recursively defined

B_0 single node

B_k two B_{k-1} joined by making one a child of the root of the other





Lemma: A binomial tree B_k has

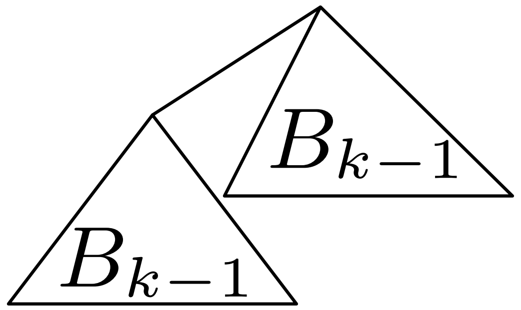
- 2^k nodes
- height k
- on level i for $i = 0, .., k$ exactly $\binom{k}{i}$ nodes
- root of degree k; children of degree $k-1, ..., 0$

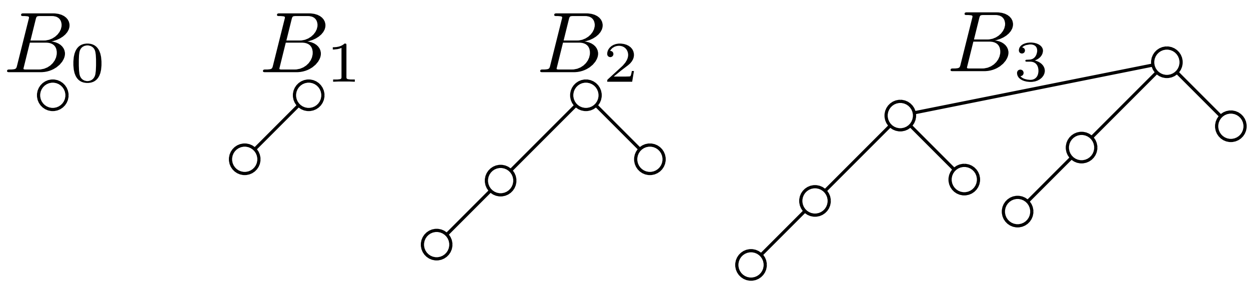
Binomial Trees

Binomial tree: recursively defined

B_0 single node

B_k two B_{k-1} joined by making one a child of the root of the other





Lemma: A binomial tree B_k has

- 2^k nodes
- height k
- on level i for $i = 0, .., k$ exactly $\binom{k}{i}$ nodes
- root of degree k; children of degree $k-1, ..., 0$

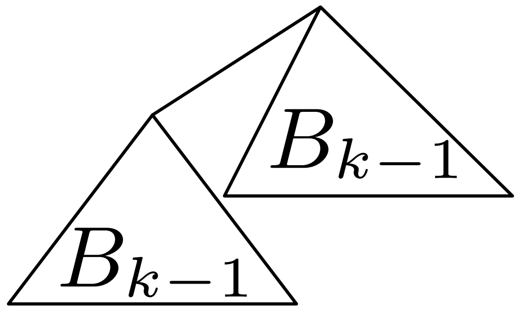
Proof: Induction(k)

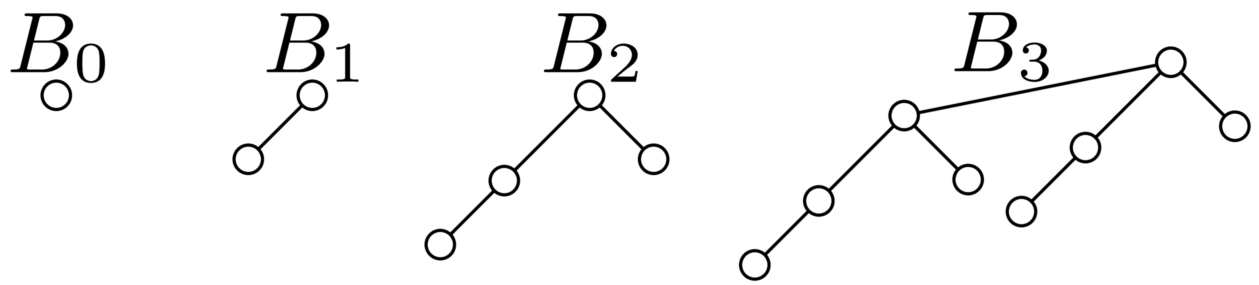
Binomial Trees

Binomial tree: recursively defined

B_0 single node

B_k two B_{k-1} joined by making one a child of the root of the other





Lemma: A binomial tree B_k has

- 2^k nodes
- height k
- on level i for $i = 0, .., k$ exactly $\binom{k}{i}$ nodes
- root of degree k; children of degree $k-1, ..., 0$

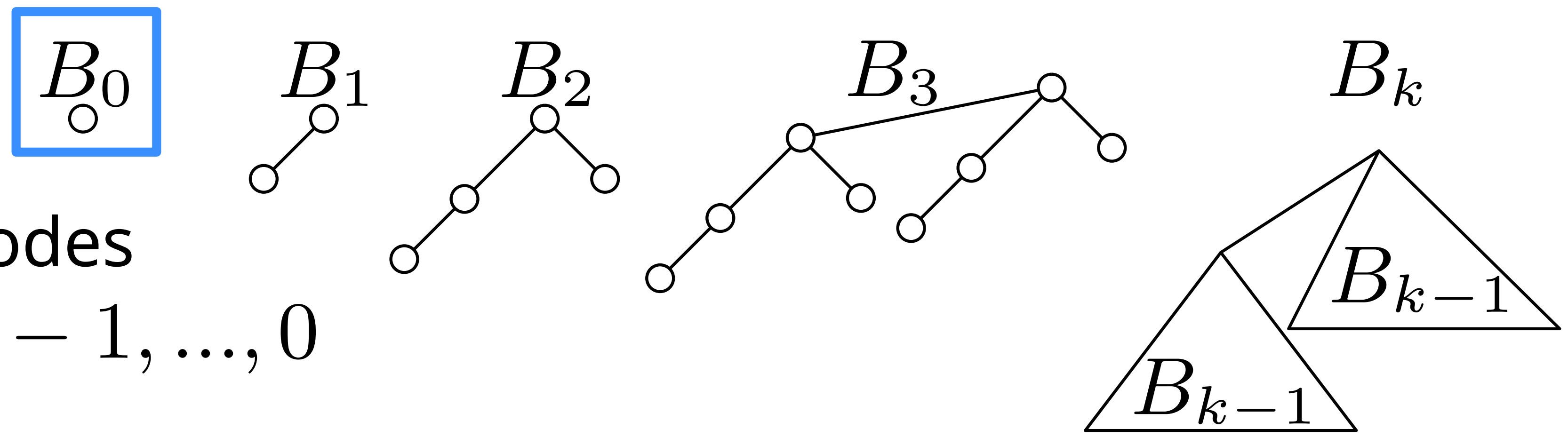
Proof: Induction(k)

Corollary: Maximal degree in a binomial tree on n nodes is $\log n$.

Binomial Trees – Properties

Lemma: A binomial tree B_k has

- 2^k nodes, height k
- on level i for $i = 0, .., k$ exactly $\binom{k}{i}$ nodes
- root of degree k; children of degree $k-1, \ldots, 0$



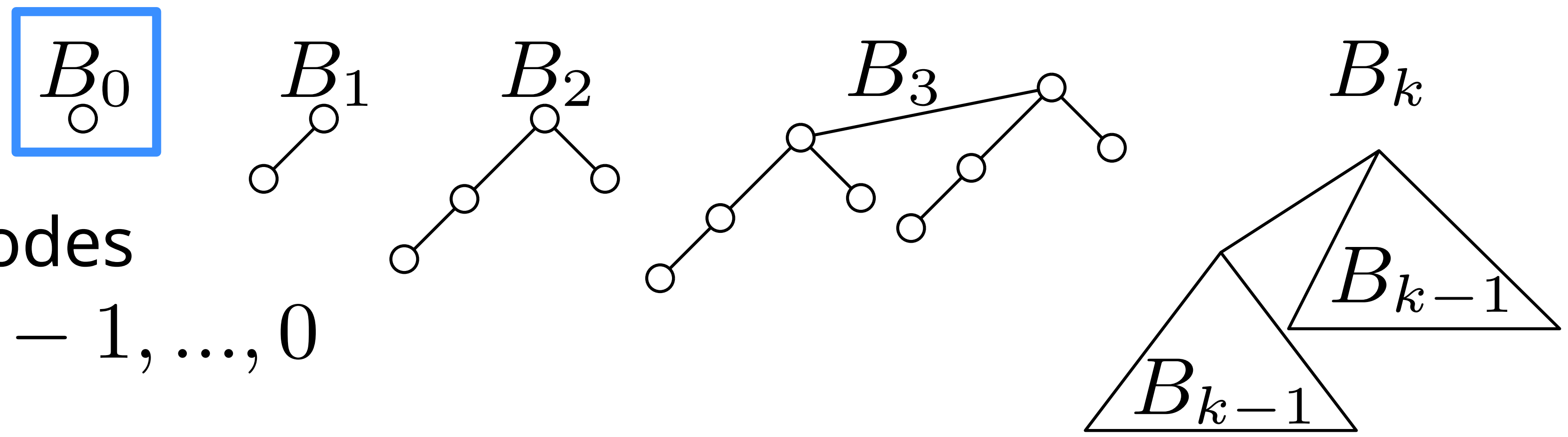
Proof: Induction(k)

base: $k = 0$

Binomial Trees – Properties

Lemma: A binomial tree B_k has

- 2^k nodes, height k
- on level i for $i = 0, .., k$ exactly $\binom{k}{i}$ nodes
- root of degree k; children of degree $k-1, \ldots, 0$



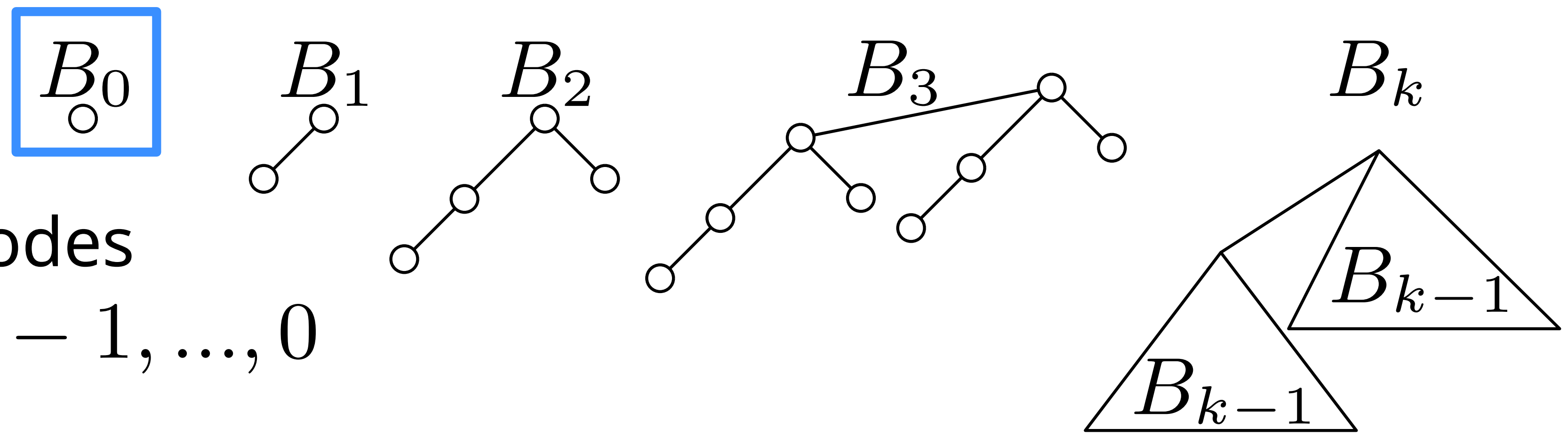
Proof: Induction(k)

base: $k = 0$

- $1 = 2^0$ nodes, height 0

Binomial Trees – Properties

Lemma: A binomial tree B_k has
- 2^k nodes, height k
- on level i for $i = 0, .., k$ exactly $\binom{k}{i}$ nodes
- root of degree k; children of degree $k-1, \ldots, 0$



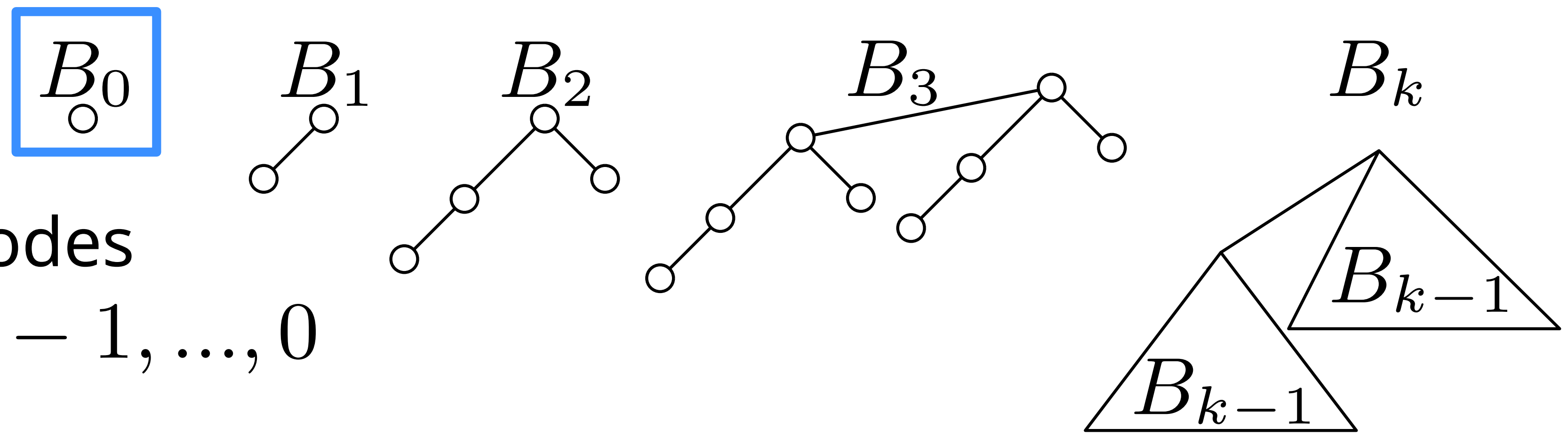
Proof: Induction(k)

base: $k = 0$
- $1 = 2^0$ nodes, height 0
- on level 0 exactly $\binom{0}{0} = 1$ nodes

Binomial Trees – Properties

Lemma: A binomial tree B_k has

- 2^k nodes, height k
- on level i for $i = 0, .., k$ exactly $\binom{k}{i}$ nodes
- root of degree k; children of degree $k-1, ..., 0$



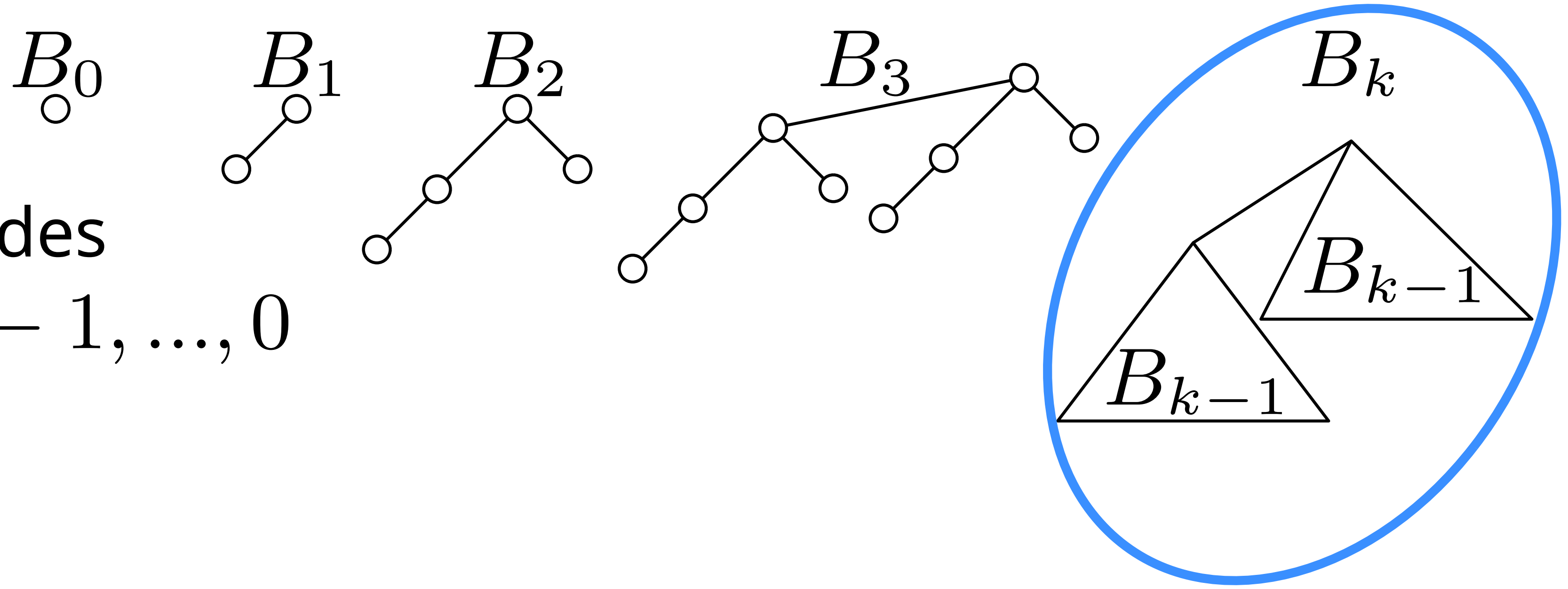
Proof: Induction(k)

base: $k = 0$

- $1 = 2^0$ nodes, height 0
- on level 0 exactly $\binom{0}{0} = 1$ nodes
- root of degree 0: no children

Binomial Trees – Properties

Lemma: A binomial tree B_k has
- 2^k nodes, height k
- on level i for $i = 0, .., k$ exactly $\binom{k}{i}$ nodes
- root of degree k; children of degree $k-1, ..., 0$



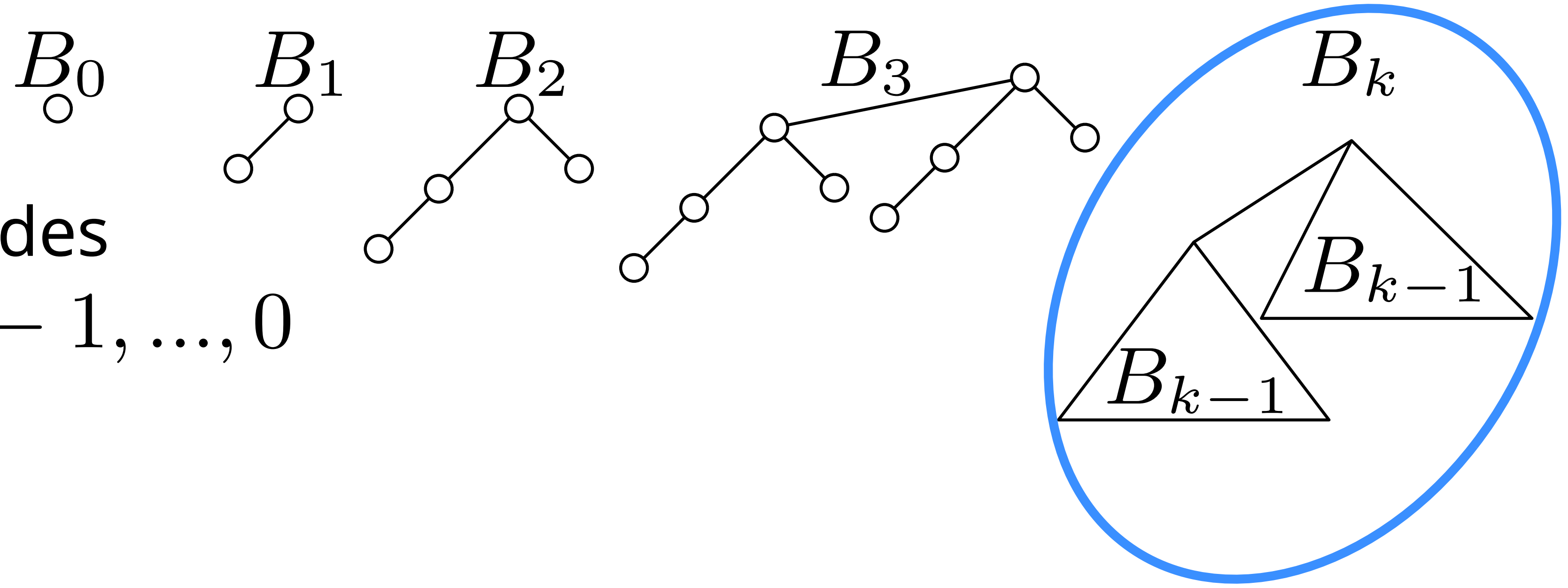
Proof: Induction(k)

step: $k > 0$

Binomial Trees – Properties

Lemma: A binomial tree B_k has

- 2^k nodes, height k
- on level i for $i = 0, .., k$ exactly $\binom{k}{i}$ nodes
- root of degree k; children of degree $k-1, \ldots, 0$



Proof: Induction(k)

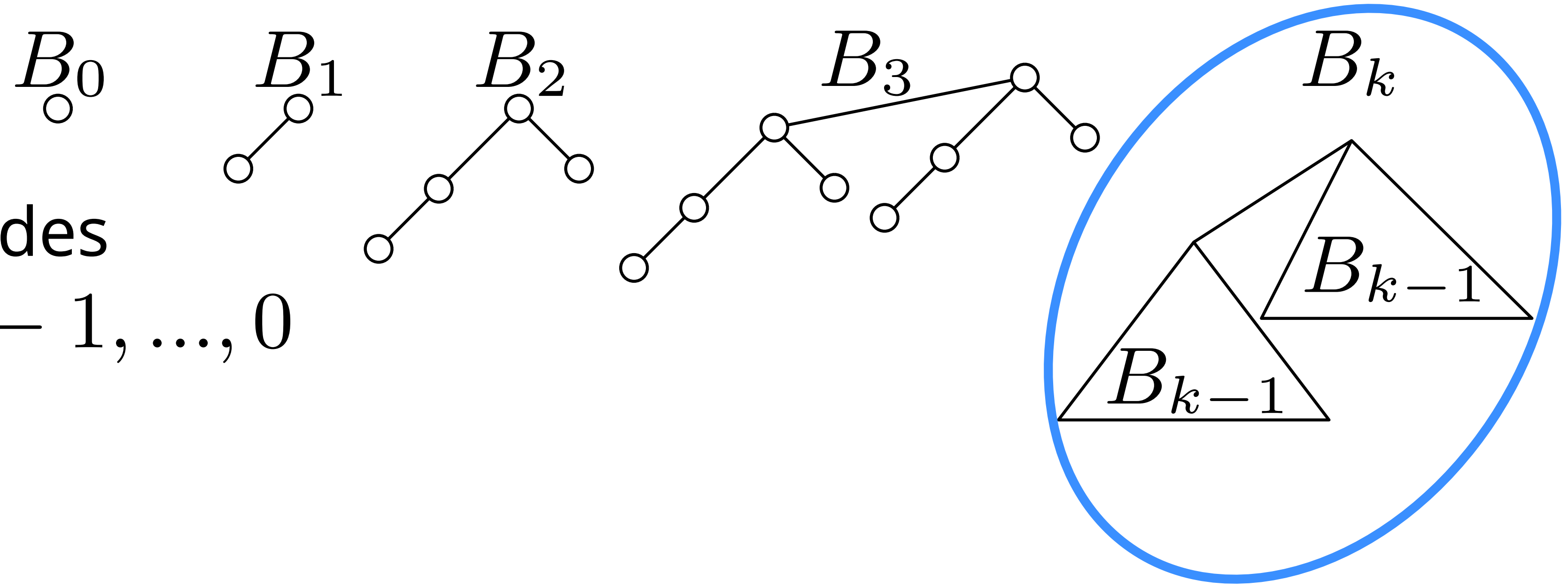
step: $k > 0$

- $2^{k-1} + 2^{k-1} = 2^k$ nodes

Binomial Trees – Properties

Lemma: A binomial tree B_k has

- 2^k nodes, height k
- on level i for $i = 0, .., k$ exactly $\binom{k}{i}$ nodes
- root of degree k; children of degree $k-1, \ldots, 0$



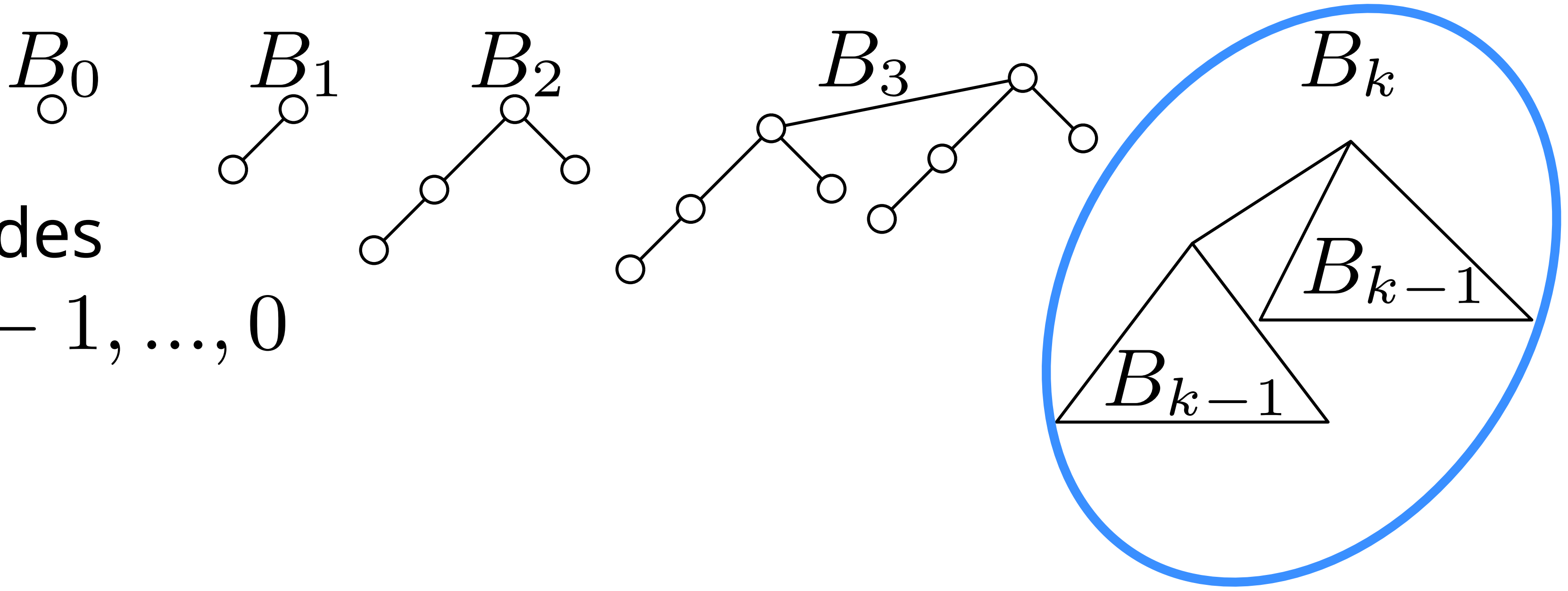
Proof: Induction(k)

step: $k > 0$

- $2^{k-1} + 2^{k-1} = 2^k$ nodes, height $(k-1) + 1 = k$

Binomial Trees – Properties

Lemma: A binomial tree B_k has
- 2^k nodes, height k
- on level i for $i = 0, .., k$ exactly $\binom{k}{i}$ nodes
- root of degree k; children of degree $k-1, \ldots, 0$



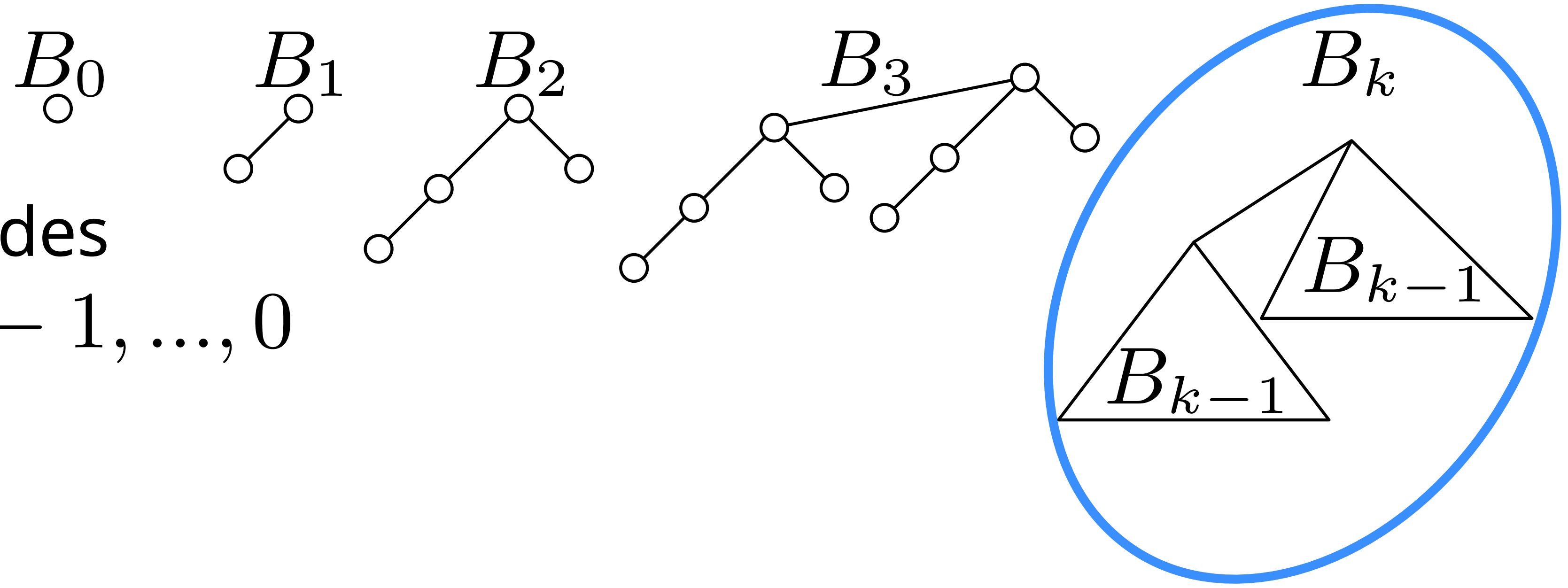
Proof: Induction(k)

step: $k > 0$
- $2^{k-1} + 2^{k-1} = 2^k$ nodes, height $(k-1)+1 = k$
- on level $i = 0$ exactly $\binom{k}{0} = 1$ nodes

Binomial Trees – Properties

Lemma: A binomial tree B_k has

- 2^k nodes, height k
- on level i for $i = 0, .., k$ exactly $\binom{k}{i}$ nodes
- root of degree k; children of degree $k-1, ..., 0$

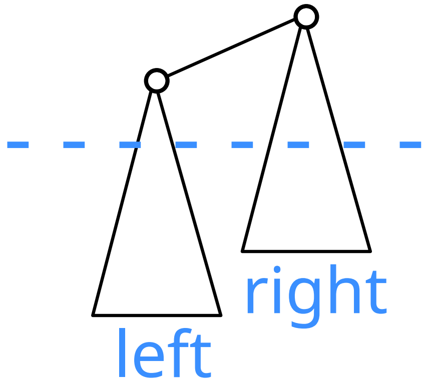


Proof: Induction(k)

step: $k > 0$

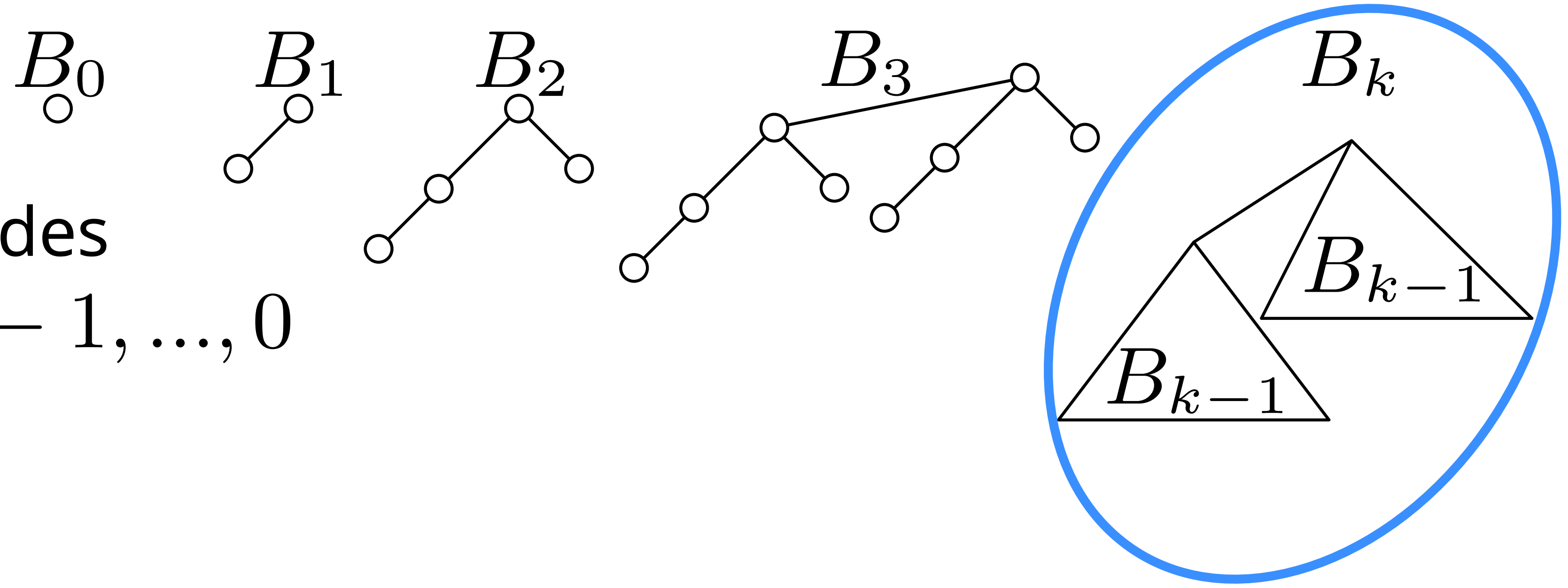
- $2^{k-1} + 2^{k-1} = 2^k$ nodes, height $(k-1)+1 = k$
- on level $i = 0$ exactly $\binom{k}{0} = 1$ nodes

  $i > 0$:



Binomial Trees – Properties

Lemma: A binomial tree B_k has
- 2^k nodes, height k
- on level i for $i = 0, .., k$ exactly $\binom{k}{i}$ nodes
- root of degree k; children of degree $k-1, \ldots, 0$

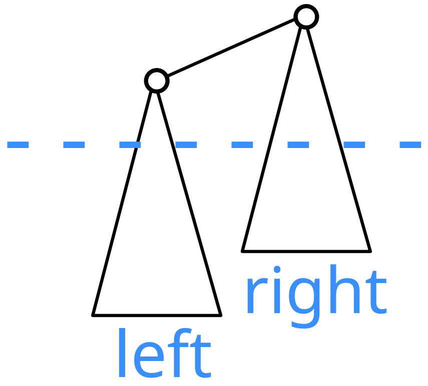


Proof: Induction(k)

step: $k > 0$

- $2^{k-1} + 2^{k-1} = 2^k$ nodes, height $(k-1)+1 = k$
- on level $i = 0$ exactly $\binom{k}{0} = 1$ nodes

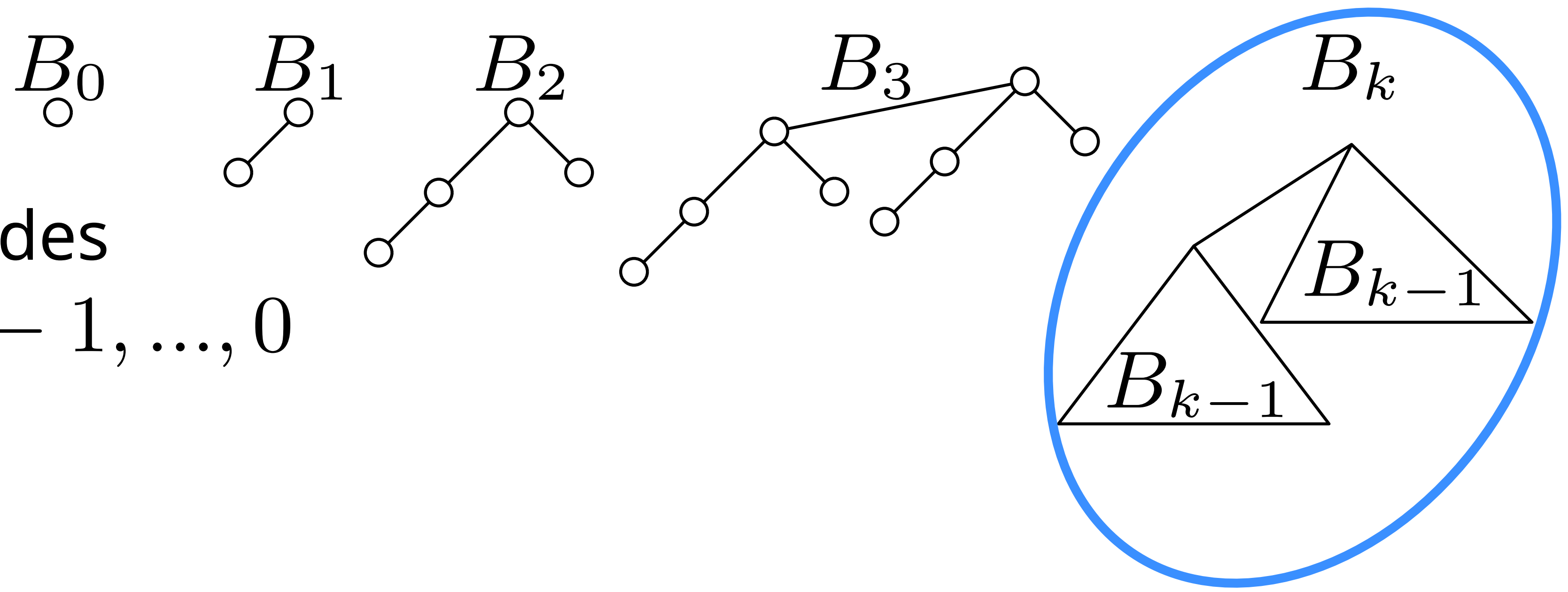
  $i > 0$: $\underset{\text{left}}{\binom{k-1}{i}} + \underset{\text{right}}{\binom{k-1}{i-1}} = \binom{k}{i}$ nodes



Binomial Trees – Properties

Lemma: A binomial tree B_k has

- 2^k nodes, height k
- on level i for $i = 0, .., k$ exactly $\binom{k}{i}$ nodes
- root of degree k; children of degree $k-1, \ldots, 0$

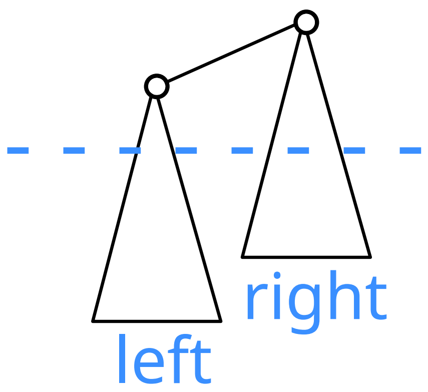


Proof: Induction(k)

step: $k > 0$

- $2^{k-1} + 2^{k-1} = 2^k$ nodes, height $(k-1)+1 = k$
- on level $i = 0$ exactly $\binom{k}{0} = 1$ nodes

  $i > 0$: $\underbrace{\binom{k-1}{i}}_{\text{left}} + \underbrace{\binom{k-1}{i-1}}_{\text{right}} = \binom{k}{i}$ nodes

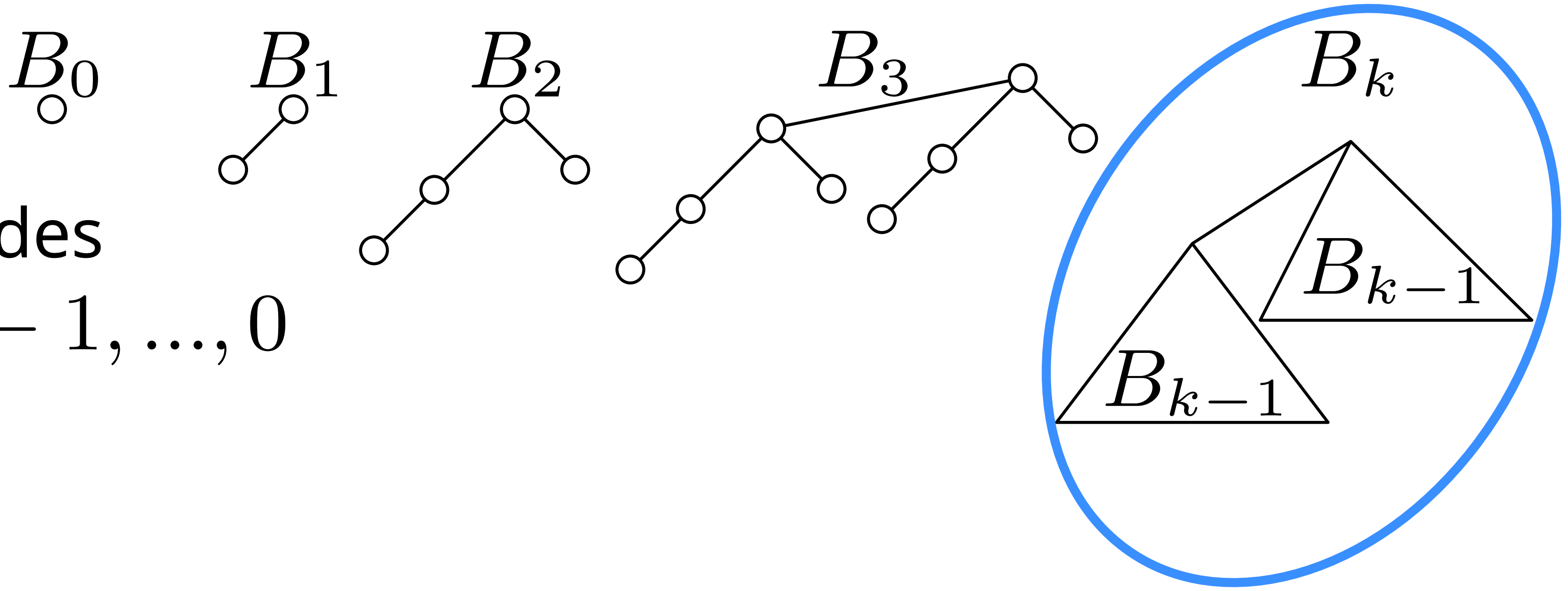


- degree of root: $(k-1)+1 = k$

Binomial Trees – Properties

Lemma: A binomial tree B_k has

- 2^k nodes, height k
- on level i for $i = 0, .., k$ exactly $\binom{k}{i}$ nodes
- root of degree k; children of degree $k-1, ..., 0$

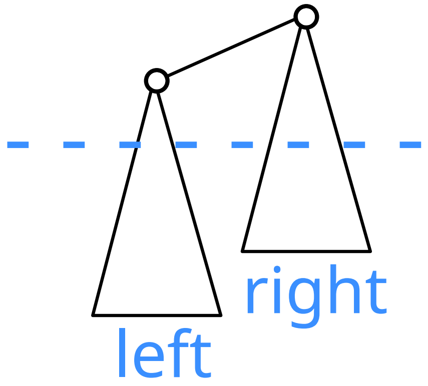


Proof: Induction(k)

step: $k > 0$

- $2^{k-1} + 2^{k-1} = 2^k$ nodes, height $(k-1)+1 = k$
- on level $i = 0$ exactly $\binom{k}{0} = 1$ nodes

  $i > 0$: $\underbrace{\binom{k-1}{i}}_{\text{left}} + \underbrace{\binom{k-1}{i-1}}_{\text{right}} = \binom{k}{i}$ nodes

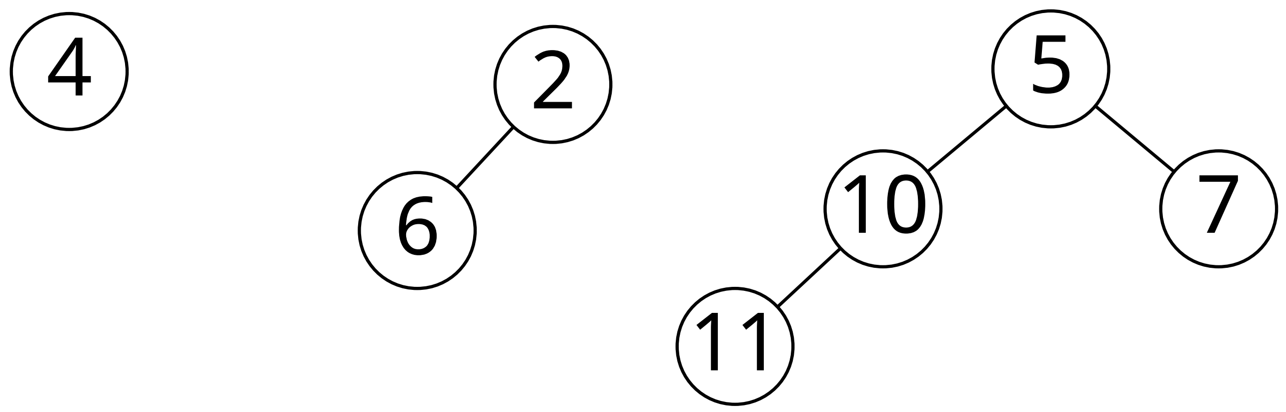


- degree of root: $(k-1)+1 = k$
- degree of children: $\underbrace{k-1}_{\text{left}},\ \underbrace{k-2, \ldots, 0}_{\text{right}}$

Binomial Heaps

Binomial Heap is a set of binomial trees where each node stores a key with the binomial heap property:

- each binomial tree fulfills the MinHeap-property
- for all $k \geq 0$ there is at most one binomial tree B_k



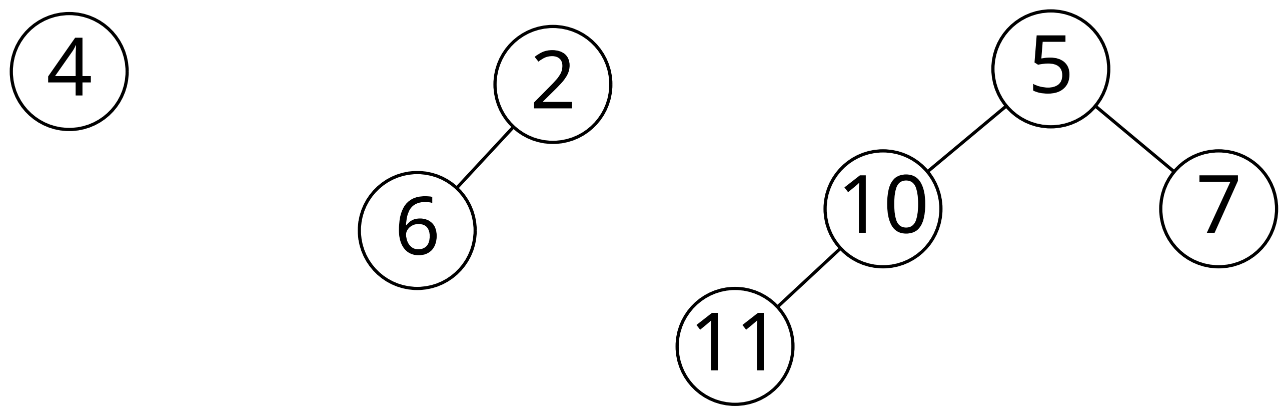
Binomial Heaps

Binomial Heap is a set of binomial trees where each node stores a key with the binomial heap property:

- each binomial tree fulfills the MinHeap-property
- for all $k \geq 0$ there is at most one binomial tree B_k

$\Rightarrow$ a binomial heap on n nodes consists of at most $\lfloor \log n \rfloor + 1$ binomial trees

and these correspond to the binary representation of $n = \sum_{i=0}^{\lfloor \log n \rfloor} b_i 2^i$



here $n = 7$, which is 111 in binary representation.

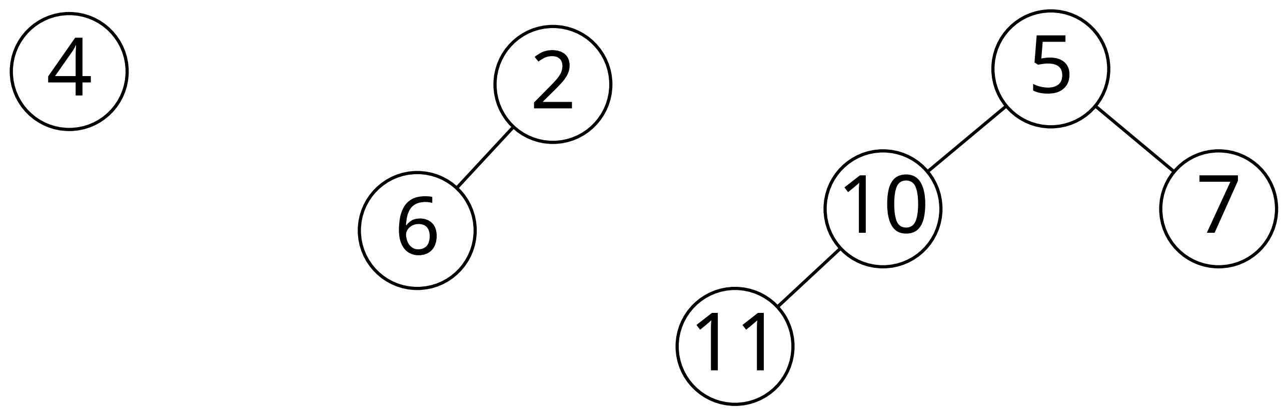
Binomial Heaps

Binomial Heap is a set of binomial trees where each node stores a key with the binomial heap property:

- each binomial tree fulfills the MinHeap-property
- for all $k \geq 0$ there is at most one binomial tree B_k

$\Rightarrow$ a binomial heap on n nodes consists of at most $\lfloor \log n \rfloor + 1$ binomial trees

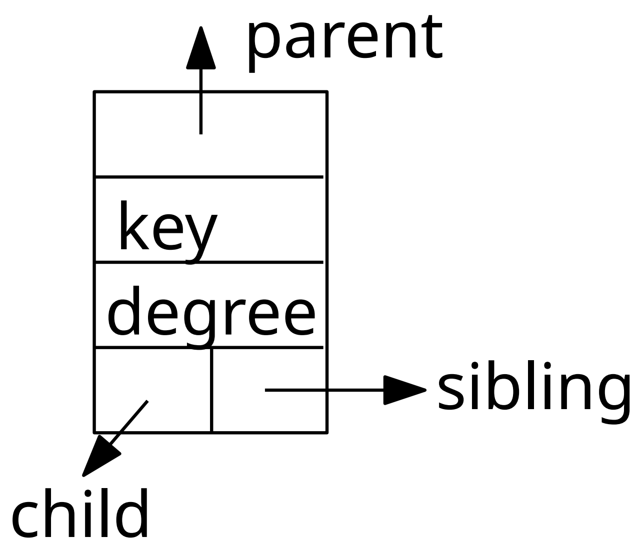
and these correspond to the binary representation of $n = \sum_{i=0}^{\lfloor \log n \rfloor} b_i 2^i$



here $n = 7$, which is 111 in binary representation.

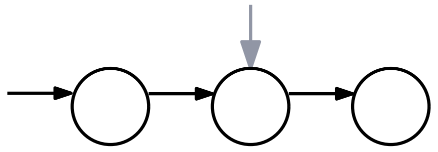
What does a binomial heap on $n = 6$ or $n = 8$ nodes look like?

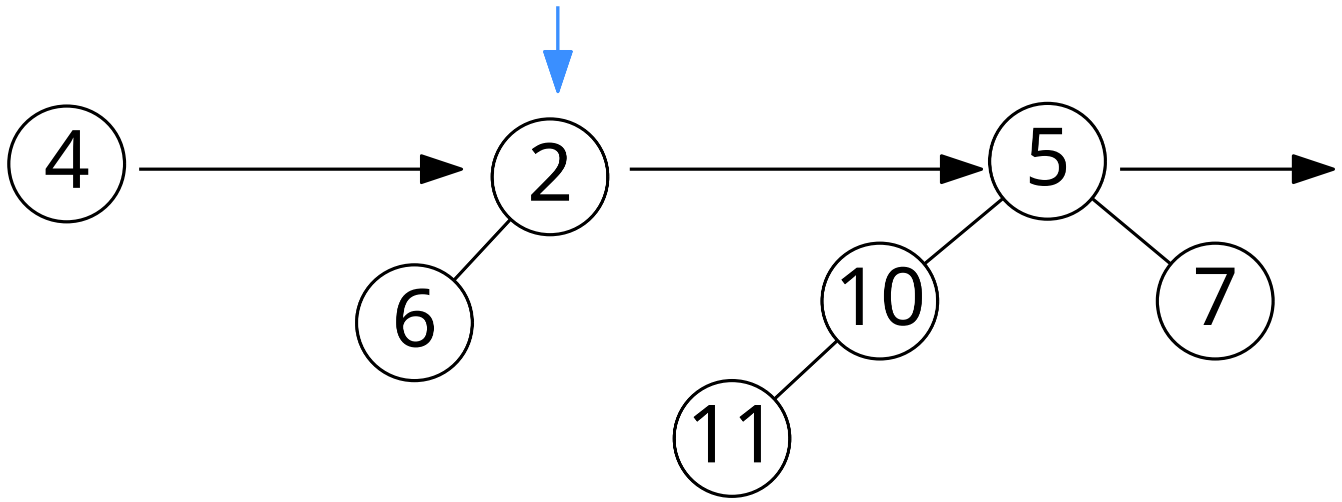
Binomial Heaps



Implementation:

- each node x stores key$[x]$, degree$[x]$, and three pointers to its parent, leftmost child, right sibling
- root of binomial heaps are stored in a linked list (of heaps of increasing size)





Operations

make-0: generate empty heap (i.e. empty list)

Operations

make-0: generate empty heap (i.e. empty list) head$[H] \rightarrow$ NIL

Operations

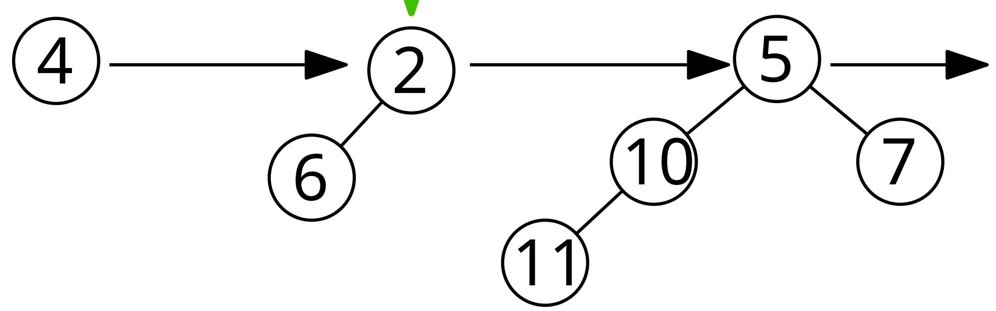
make-0: generate empty heap (i.e. empty list) head$[H] \rightarrow$ NIL

min: iterate through list of roots (or store extra pointer)

Operations

make-0: generate empty heap (i.e. empty list) head$[H] \rightarrow$ NIL

min: iterate through list of roots (or store extra pointer) min$[H]$



Operations

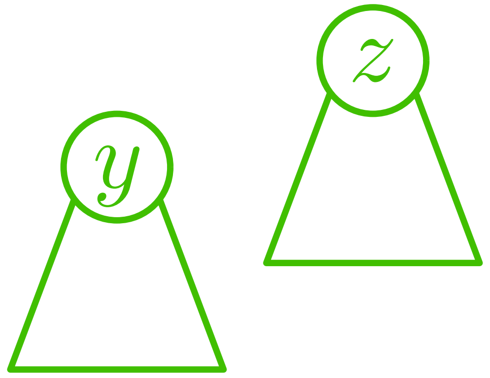
make-0: generate empty heap (i.e. empty list)

min: iterate through list of roots (or store extra pointer)

union(H_1, H_2):

- merge the lists of roots (by increasing degree/size)
- iterate through merged list and link binomial trees of equal k

link(y, z) (assuming key[z] $\leq$ key[y])



Operations

make-0: generate empty heap (i.e. empty list)

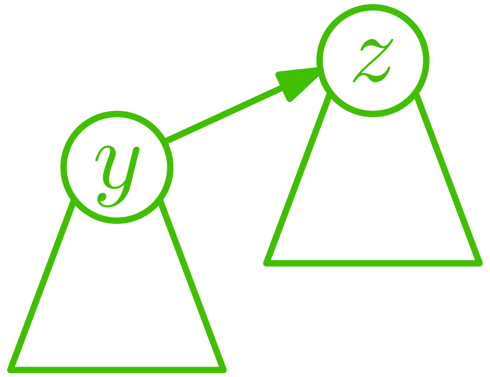
min: iterate through list of roots (or store extra pointer)

union(H_1, H_2):

- merge the lists of roots (by increasing degree/size)
- iterate through merged list and link binomial trees of equal k

link(y, z) (assuming key[z] $\leq$ key[y])

parent[y] $= z$



Operations

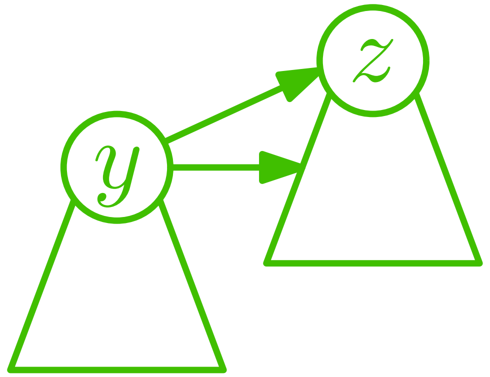
make-0: generate empty heap (i.e. empty list)

min: iterate through list of roots (or store extra pointer)

union(H_1, H_2):

- merge the lists of roots (by increasing degree/size)
- iterate through merged list and link binomial trees of equal k

link(y, z) (assuming key[z] $\leq$ key[y])
  parent[y] $= z$
  sibling[y] $=$ child[z]



Operations

make-0: generate empty heap (i.e. empty list)

min: iterate through list of roots (or store extra pointer)

union(H_1, H_2):

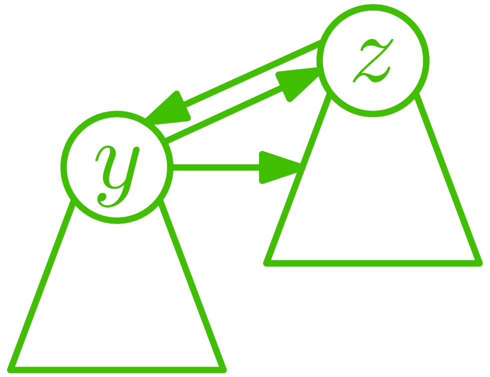
- merge the lists of roots (by increasing degree/size)
- iterate through merged list and link binomial trees of equal k

link(y, z) (assuming key[z] $\leq$ key[y])

parent[y] $= z$

sibling[y] $=$ child[z]

child[z] $= y$



Operations

make-0: generate empty heap (i.e. empty list)

min: iterate through list of roots (or store extra pointer)

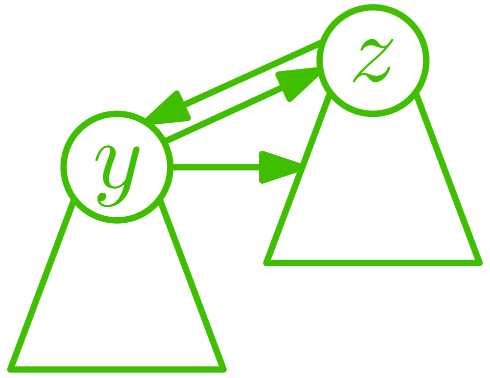
union(H_1, H_2):

- merge the lists of roots (by increasing degree/size)
- iterate through merged list and link binomial trees of equal k

link(y, z) (assuming key[z] $\leq$ key[y])

parent[y] $= z$
sibling[y] $=$ child[z]
child[z] $= y$

degree[z] = degree[z] + 1



Operations

make-0: generate empty heap (i.e. empty list)

min: iterate through list of roots (or store extra pointer)

union(H_1, H_2):

- merge the lists of roots (by increasing degree/size)
- iterate through merged list and link binomial trees of equal k

insert(H, a):

- make a 1-binomial heap (make-1)
- union with existing heap

Operations

make-0: generate empty heap (i.e. empty list)

min: iterate through list of roots (or store extra pointer)

union(H_1, H_2):

- merge the lists of roots (by increasing degree/size)
- iterate through merged list and link binomial trees of equal k

insert(H, a):

- make a 1-binomial heap (make-1) How to make a 1-binomial heap?
- union with existing heap

Operations

make-0: generate empty heap (i.e. empty list)

min: iterate through list of roots (or store extra pointer)

union(H_1, H_2):

- merge the lists of roots (by increasing degree/size)
- iterate through merged list and link binomial trees of equal k

insert(H, a):

- make a 1-binomial heap (make-1)
- union with existing heap

How to make a 1-binomial heap?

parent[x] = NIL
child[x] = NIL
sibling[x] = NIL
key[x] = a
degree[x] = 0
Head[H'] = x

Operations

make-0: generate empty heap (i.e. empty list)

min: iterate through list of roots (or store extra pointer)

union(H_1, H_2):

- merge the lists of roots (by increasing degree/size)
- iterate through merged list and link binomial trees of equal k

insert(H, a):

- make a 1-binomial heap (make-1)
- union with existing heap

delete-min:

- find min root x and delete it from root list
- union with the list of children of x (in opposite direction)

Operations

make-0: generate empty heap (i.e. empty list)

min: iterate through list of roots (or store extra pointer)

union(H_1, H_2):

- merge the lists of roots (by increasing degree/size)
- iterate through merged list and link binomial trees of equal k

insert(H, a):

- make a 1-binomial heap (make-1)
- union with existing heap

delete-min:

- find min root x and delete it from root list
- union with the list of children of x (in opposite direction)

decKey: Q: how to realize decreaseKey(x,k)?

Operations

make-0: generate empty heap (i.e. empty list)

min: iterate through list of roots (or store extra pointer)

union(H_1, H_2):

- merge the lists of roots (by increasing degree/size)
- iterate through merged list and link binomial trees of equal k

insert(H, a):

- make a 1-binomial heap (make-1)
- union with existing heap

delete-min:

- find min root x and delete it from root list
- union with the list of children of x (in opposite direction)

decKey: siftUp/bubbleUp in corresponding tree like in ordinary heap

Operations

make-0: generate empty heap (i.e. empty list)

min: iterate through list of roots (or store extra pointer)

union(H_1, H_2):

- merge the lists of roots (by increasing degree/size)
- iterate through merged list and link binomial trees of equal k

insert(H, a):

- make a 1-binomial heap (make-1)
- union with existing heap

delete-min:

- find min root x and delete it from root list
- union with the list of children of x (in opposite direction)

decKey: siftUp/bubbleUp in corresponding tree like in ordinary heap

delete(x): Q: how to realize delete(x)?

Operations

make-0: generate empty heap (i.e. empty list)

min: iterate through list of roots (or store extra pointer)

union(H_1, H_2):

- merge the lists of roots (by increasing degree/size)
- iterate through merged list and link binomial trees of equal k

insert(H, a):

- make a 1-binomial heap (make-1)
- union with existing heap

delete-min:

- find min root x and delete it from root list
- union with the list of children of x (in opposite direction)

decKey: siftUp/bubbleUp in corresponding tree like in ordinary heap

delete(x): decKey($x, -\infty$); delete-min

Operations – Examples

make-0

Operations – Examples

make-0 head $\rightarrow\!\!\blacktriangleright\!\!|$ $\emptyset$

Operations – Examples

make-0	head $\rightarrow\!\blacktriangleright\!|$	$\emptyset$
insert(5)	???	???

Operations – Examples

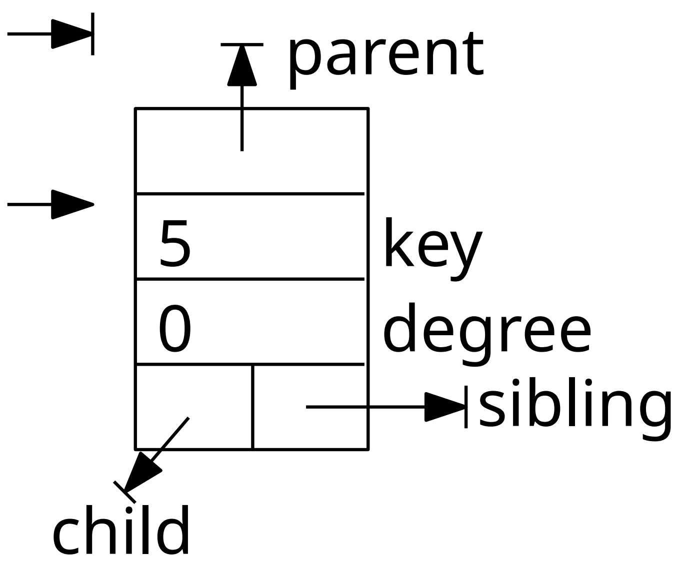
make-0 head $\rightarrow\!|$ $\emptyset$

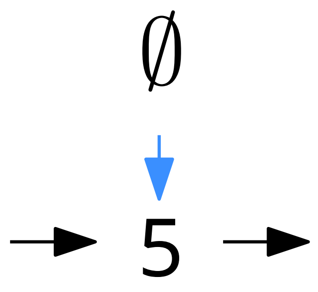
insert(5) $\rightarrow 5 \rightarrow$ (make-1)

Operations – Examples

make-0 head

insert(5) head



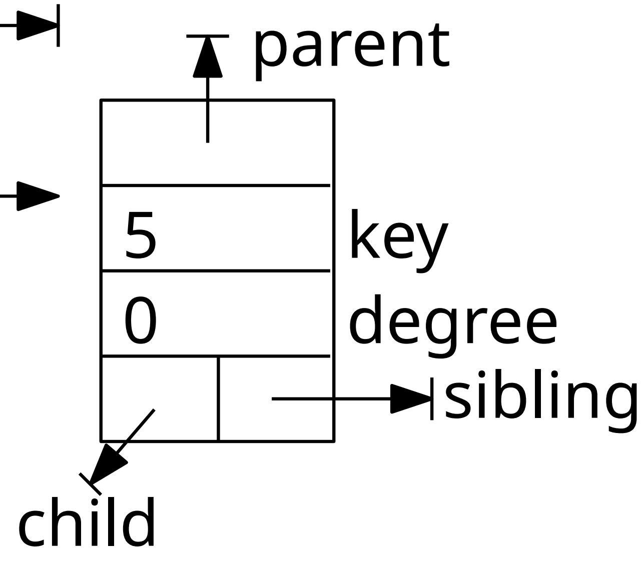


(make-1)

Operations – Examples

make-0 head → $\emptyset$

insert(5) head →



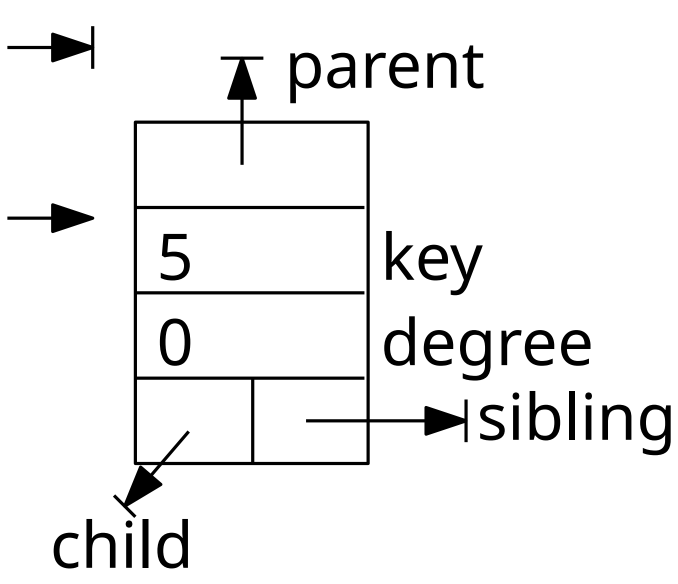
→ 5 → (make-1)

insert(7) ??? ???

Operations – Examples

make-0 head → $\emptyset$

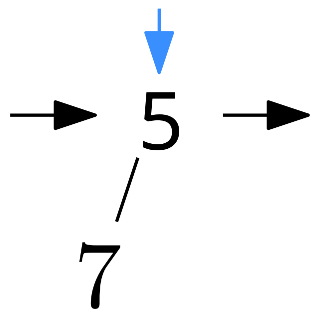
insert(5) head →



→ 5 →

(make-1)

insert(7)

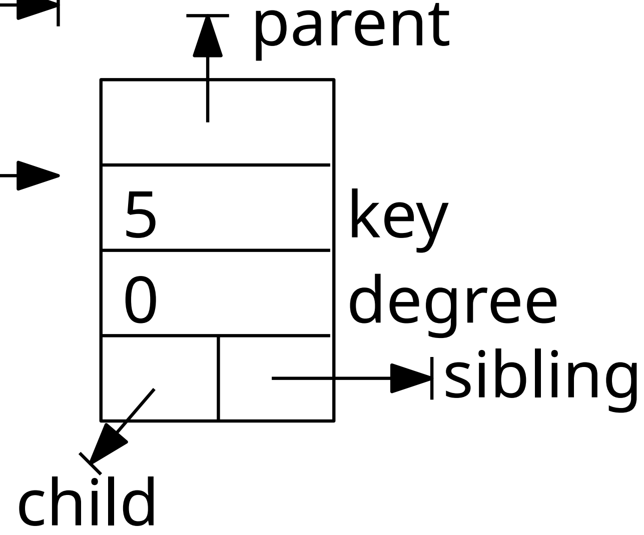


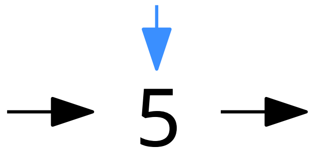
(make-1 +
union $\rightarrow$ link)

Operations – Examples

make-0 head $\rightarrow\!\mid$ $\emptyset$

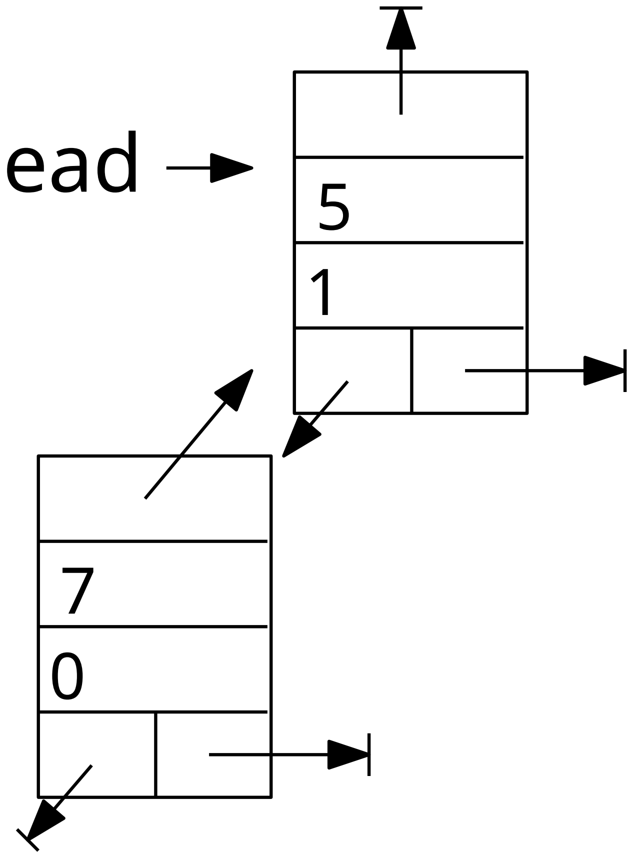
insert(5) head $\rightarrow$

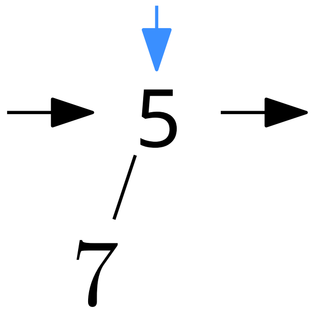




(make-1)

insert(7) head $\rightarrow$



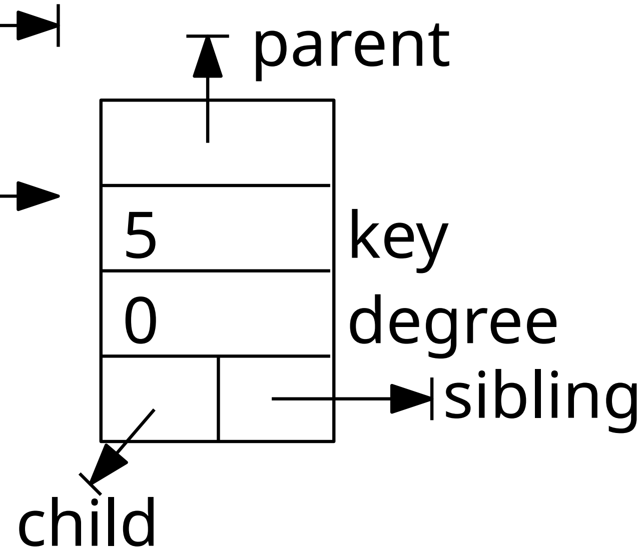


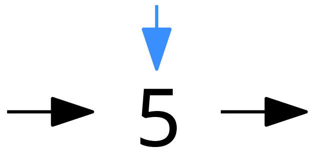
(make-1 +
union $\rightarrow$ link)

Operations – Examples

make-0 head →| $\emptyset$

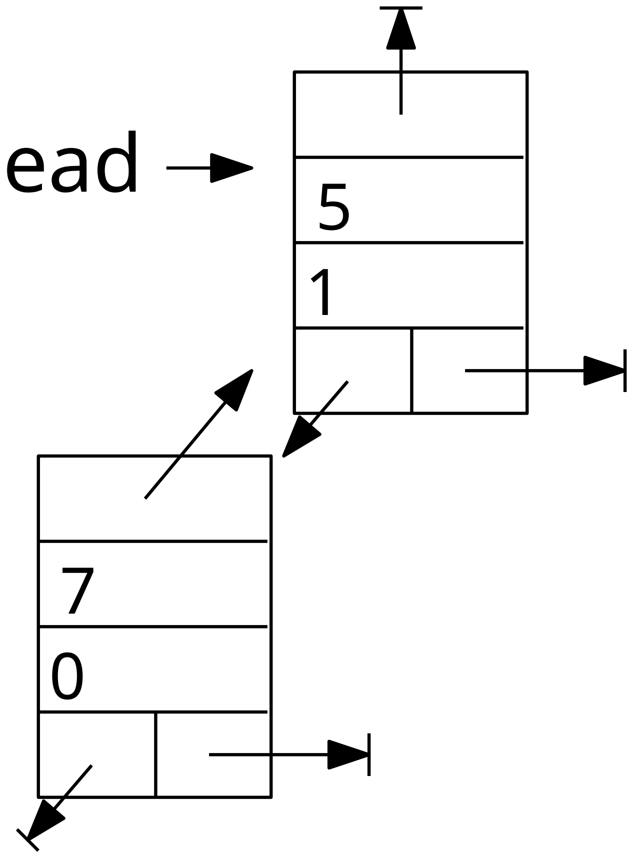
insert(5) head →

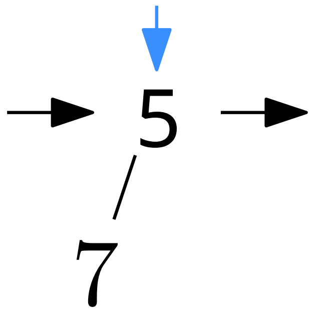




(make-1)

insert(7) head →



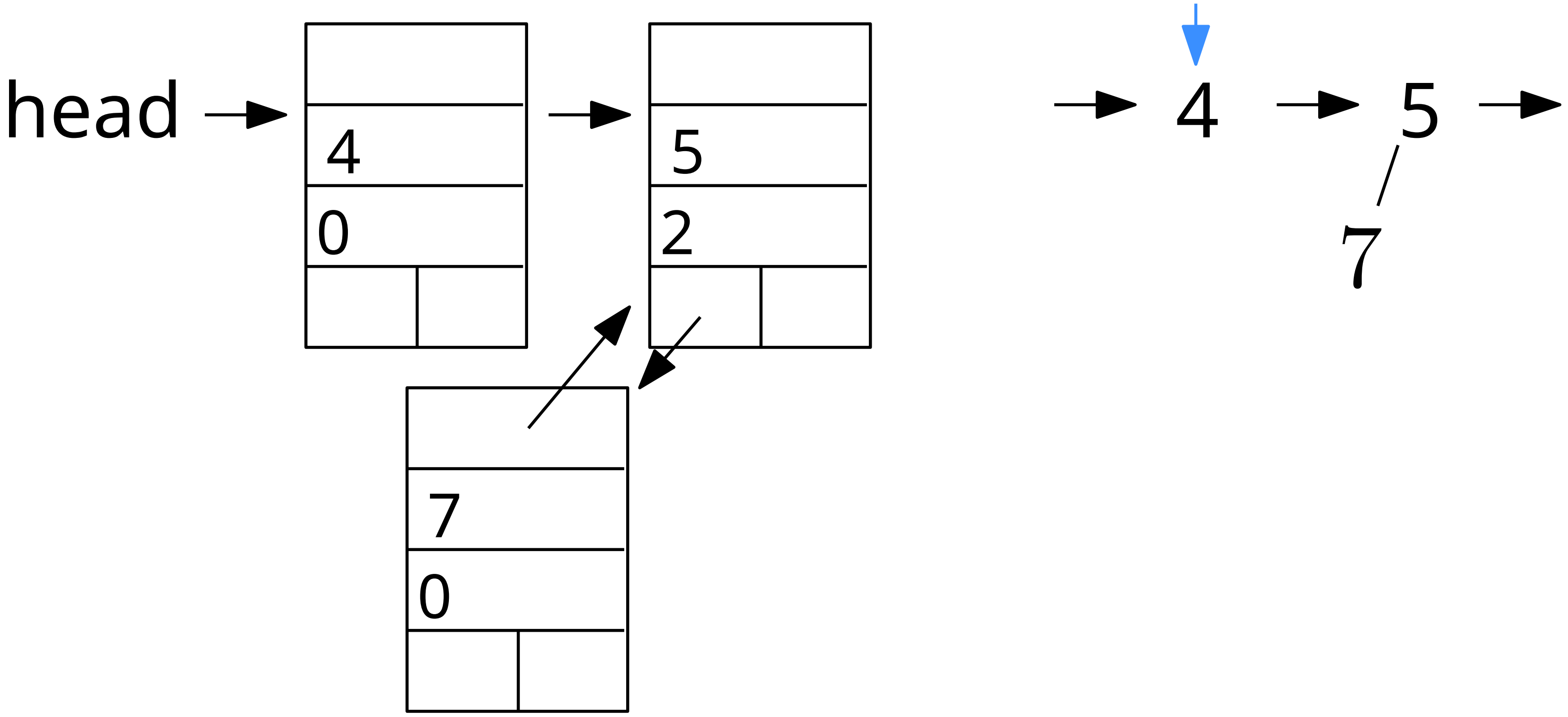


(make-1 + union → link)

insert(4) ???

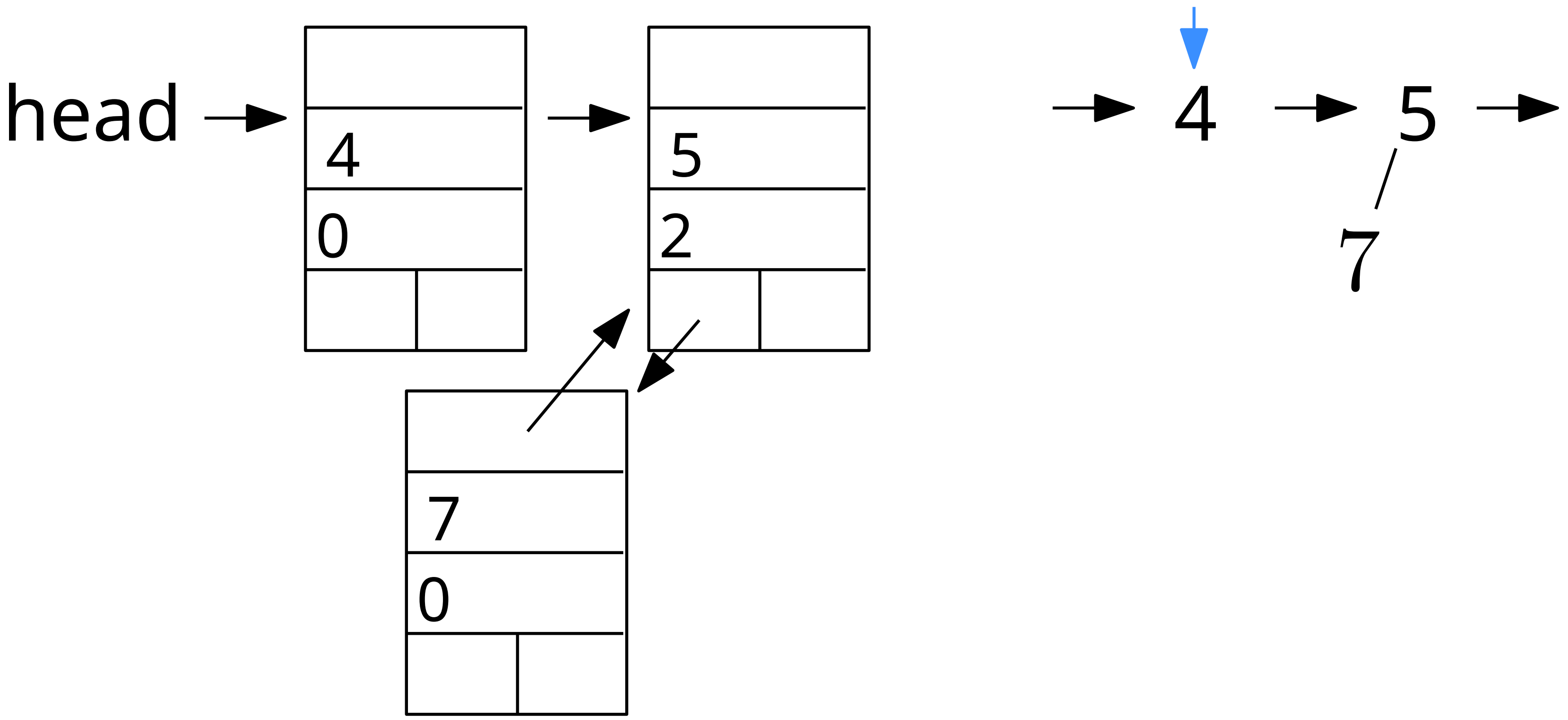
Operations – Examples

insert(4)



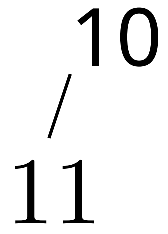
Operations – Examples

insert(4)



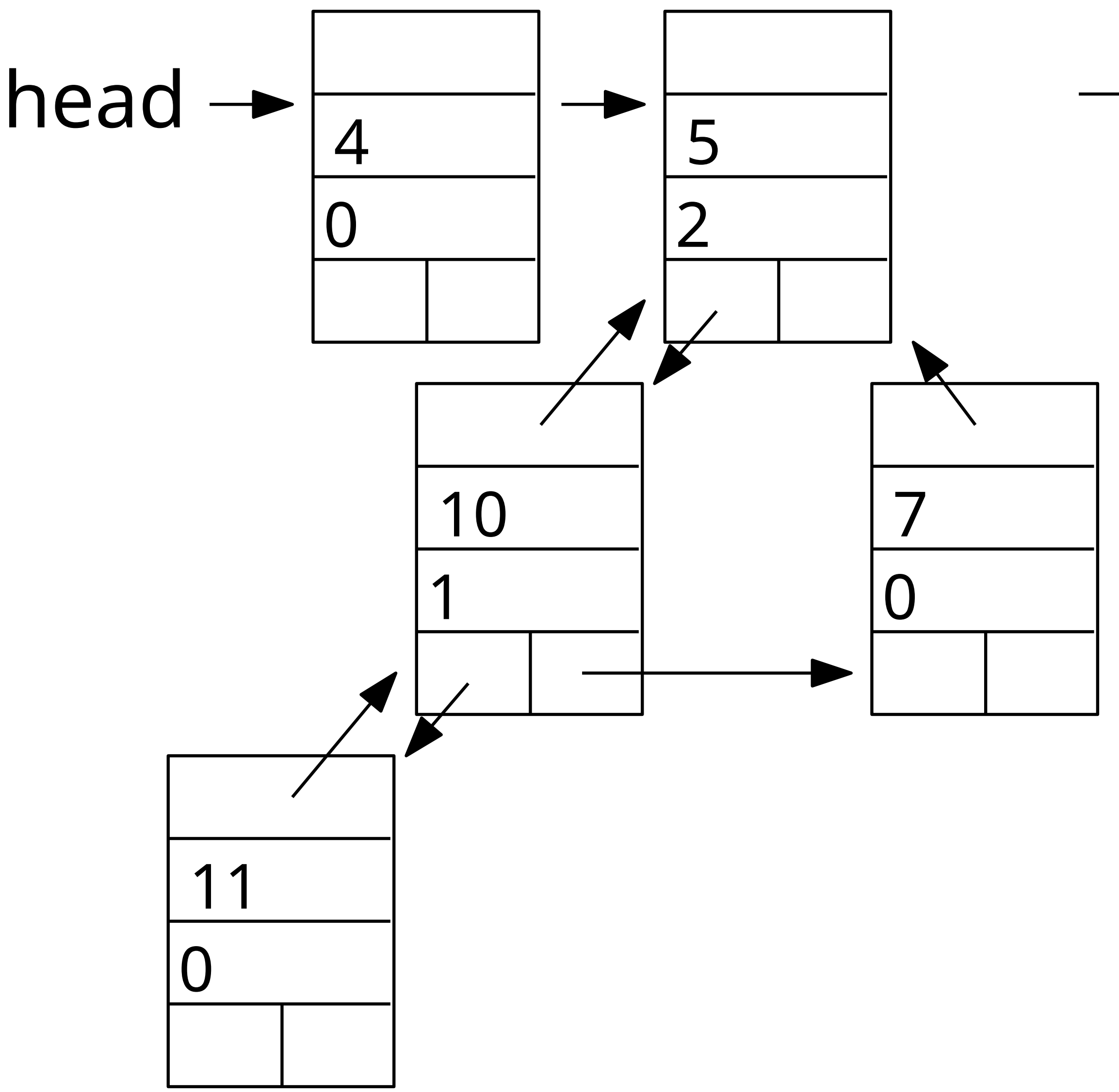
union(H, H')

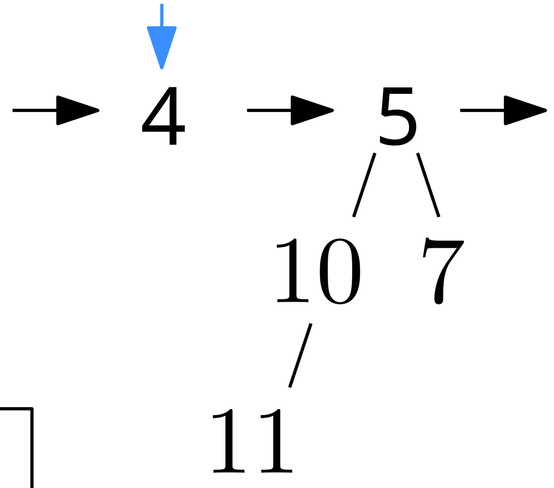
H' :



Operations – Examples

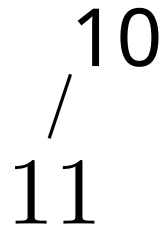
insert(4)



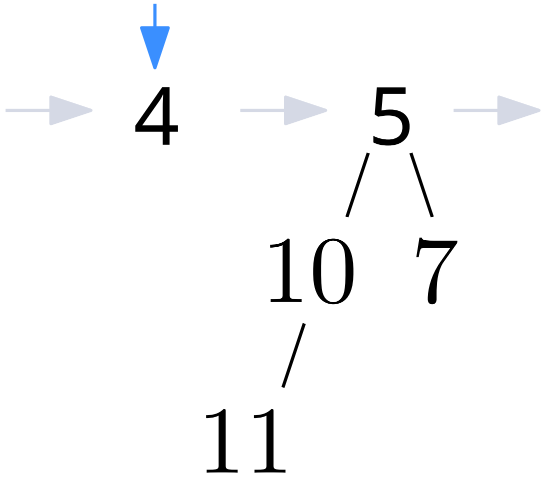


union(H, H')

H' :

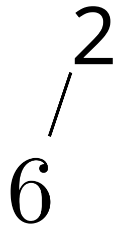


Operations – Examples



union(H, H')

H'' :

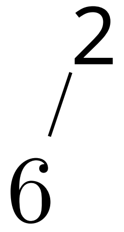


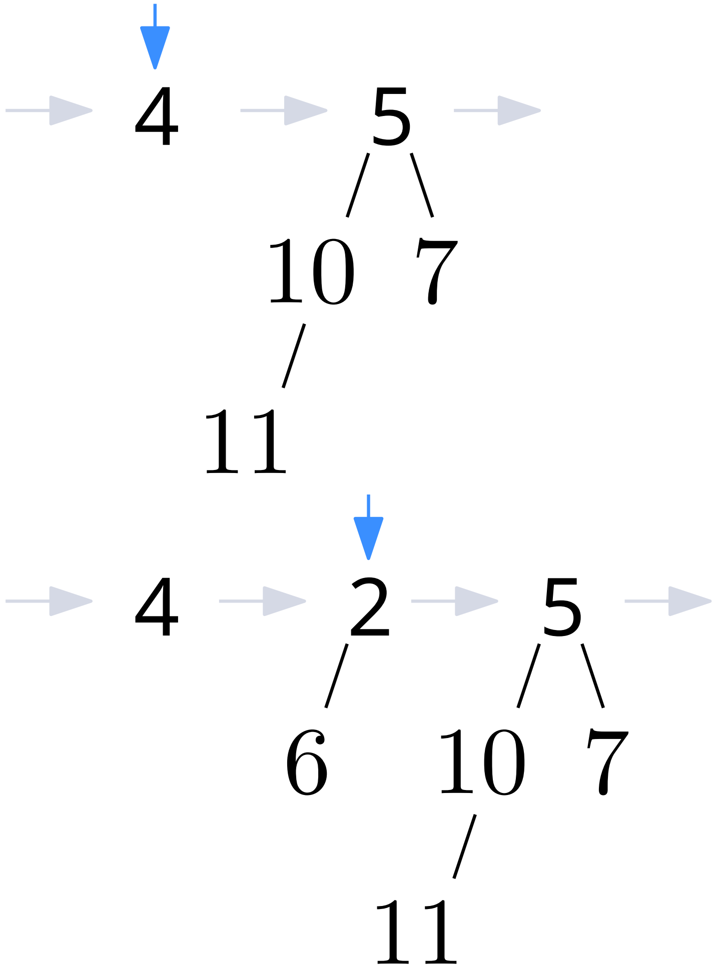
???

Operations – Examples

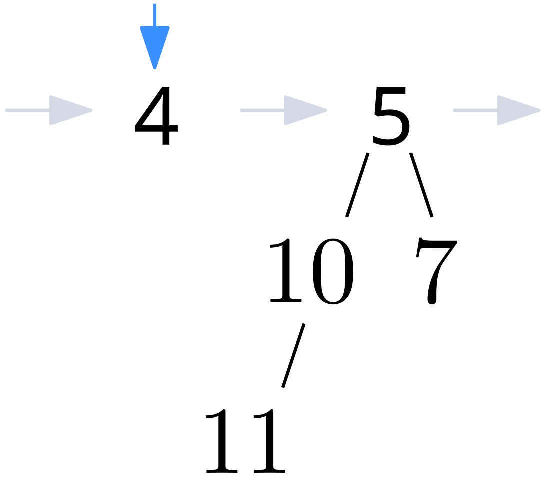
union(H, H')

H'' :

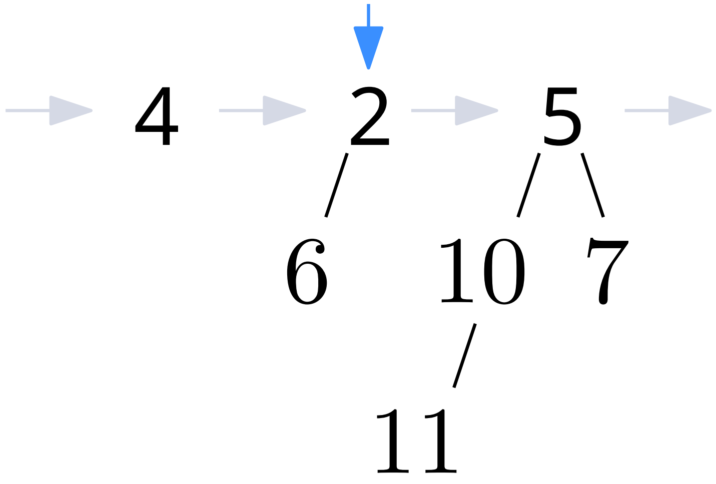




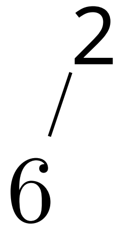
Operations – Examples



union(H, H')



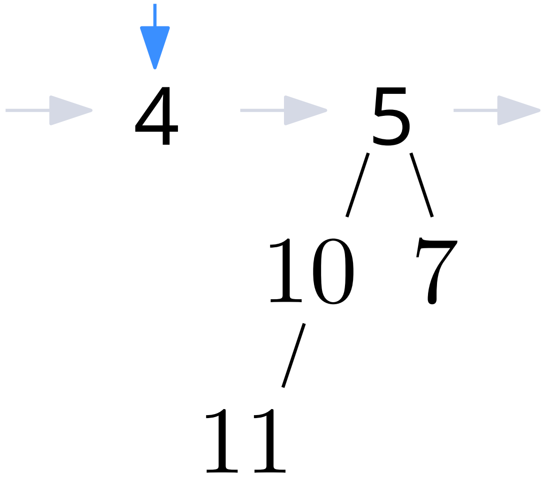
H'' :



deleteMin

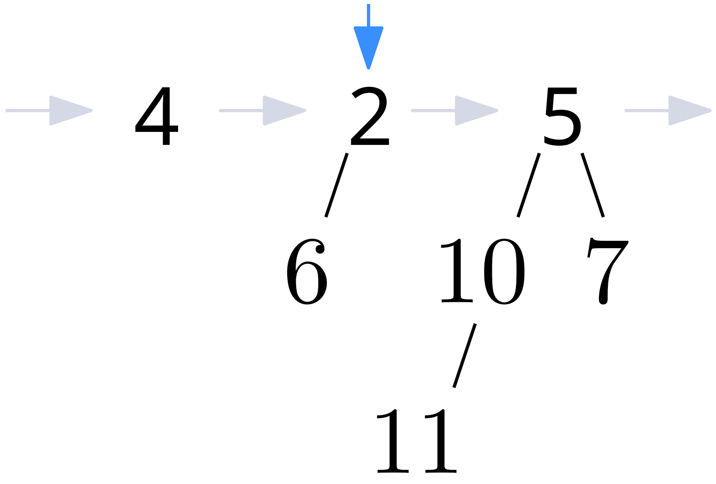
???

Operations – Examples

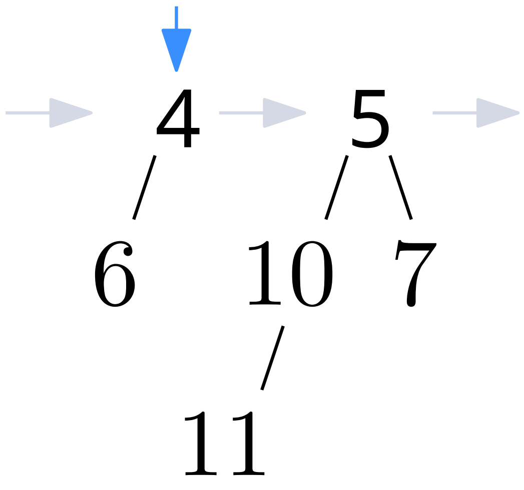


union(H, H')

H'' : 2 / 6



deleteMin



Operations – Runtime

Runtimes: ???

make-0: generate empty heap (i.e. empty list)

min: iterate through list of roots (or store extra pointer)

union(H_1, H_2):

- merge the lists of roots
- iterate through merged list and link binomial trees of equal k

insert(H, a):

- make a 1-binomial heap (make-1)
- union with existing heap

delete-min:

- find min root x and delete it from root list
- union with the list of children of x (in opposite direction)

decKey: siftUp/bubbleUp in corresponding tree like in ordinary heap

delete(x): decKey($x, -\infty$); delete-min

Operations – Runtime

Runtimes: ???

make-0: generate empty heap (i.e. empty list) $O(1)$

min: iterate through list of roots (or store extra pointer)

union(H_1, H_2):

- merge the lists of roots
- iterate through merged list and link binomial trees of equal k

insert(H, a):

- make a 1-binomial heap (make-1)
- union with existing heap

delete-min:

- find min root x and delete it from root list
- union with the list of children of x (in opposite direction)

decKey: siftUp/bubbleUp in corresponding tree like in ordinary heap

delete(x): decKey($x, -\infty$); delete-min

Operations – Runtime

Runtimes: ???

make-0: generate empty heap (i.e. empty list) $O(1)$

min: iterate through list of roots (or store extra pointer) $O(\log n)$

union(H_1, H_2):

- merge the lists of roots
- iterate through merged list and link binomial trees of equal k

insert(H, a):

- make a 1-binomial heap (make-1)
- union with existing heap

delete-min:

- find min root x and delete it from root list
- union with the list of children of x (in opposite direction)

decKey: siftUp/bubbleUp in corresponding tree like in ordinary heap

delete(x): decKey($x, -\infty$); delete-min

Operations – Runtime

Runtimes: ???

make-0: generate empty heap (i.e. empty list) $O(1)$

min: iterate through list of roots (or store extra pointer) $O(\log n)$

union(H_1, H_2): $O(\log n)$

- merge the lists of roots
- iterate through merged list and link binomial trees of equal k

insert(H, a):

- make a 1-binomial heap (make-1)
- union with existing heap

delete-min:

- find min root x and delete it from root list
- union with the list of children of x (in opposite direction)

decKey: siftUp/bubbleUp in corresponding tree like in ordinary heap

delete(x): decKey($x, -\infty$); delete-min

Operations – Runtime

Runtimes: ???

make-0: generate empty heap (i.e. empty list) $O(1)$

min: iterate through list of roots (or store extra pointer) $O(\log n)$

union(H_1, H_2): $O(\log n)$

- merge the lists of roots
- iterate through merged list and link binomial trees of equal k

insert(H, a): $O(\log n)$

- make a 1-binomial heap (make-1)
- union with existing heap

delete-min:

- find min root x and delete it from root list
- union with the list of children of x (in opposite direction)

decKey: siftUp/bubbleUp in corresponding tree like in ordinary heap

delete(x): decKey($x, -\infty$); delete-min

Operations – Runtime

Runtimes: ???

make-0: generate empty heap (i.e. empty list) $O(1)$

min: iterate through list of roots (or store extra pointer) $O(\log n)$

union(H_1, H_2): $O(\log n)$

- merge the lists of roots
- iterate through merged list and link binomial trees of equal k

insert(H, a): $O(\log n)$

- make a 1-binomial heap (make-1)
- union with existing heap

delete-min: $O(\log n)$

- find min root x and delete it from root list
- union with the list of children of x (in opposite direction)

decKey: siftUp/bubbleUp in corresponding tree like in ordinary heap

delete(x): decKey($x, -\infty$); delete-min

Operations – Runtime

Runtimes: ???

make-0: generate empty heap (i.e. empty list) $O(1)$

min: iterate through list of roots (or store extra pointer) $O(\log n)$

union(H_1, H_2): $O(\log n)$

- merge the lists of roots
- iterate through merged list and link binomial trees of equal k

insert(H, a): $O(\log n)$

- make a 1-binomial heap (make-1)
- union with existing heap

delete-min: $O(\log n)$

- find min root x and delete it from root list
- union with the list of children of x (in opposite direction)

decKey: siftUp/bubbleUp in corresponding tree like in ordinary heap $O(\log n)$

delete(x): decKey($x, -\infty$); delete-min

Operations – Runtime

Runtimes: ???

make-0: generate empty heap (i.e. empty list) $O(1)$

min: iterate through list of roots (or store extra pointer) $O(\log n)$

union(H_1, H_2): $O(\log n)$

- merge the lists of roots
- iterate through merged list and link binomial trees of equal k

insert(H, a): $O(\log n)$

- make a 1-binomial heap (make-1)
- union with existing heap

delete-min: $O(\log n)$

- find min root x and delete it from root list
- union with the list of children of x (in opposite direction)

decKey: siftUp/bubbleUp in corresponding tree like in ordinary heap $O(\log n)$

delete(x): decKey($x, -\infty$); delete-min $O(\log n)$

Runtimes

	worst-case	
make		
min		
insert		
delete-min		
union		

Runtimes

	worst-case	
make	$O(1)$	
min	$O(1)$	
insert	$O(\log n)$	
delete-min	$O(\log n)$	
union	$O(\log n)$	

Runtimes

	worst-case	
make	$O(1)$	
min	$O(1)$	
insert	$O(\log n)$	
delete-min	$O(\log n)$	
union	$O(\log n)$	

Which operation has a lower amortized runtime?

Runtimes

	worst-case	amortised
make	$O(1)$	$O(1)$
min	$O(1)$	$O(1)$
insert	$O(\log n)$	$O(1)$
delete-min	$O(\log n)$	$O(\log n)$
union	$O(\log n)$	$O(\log n)$

Amortised Analysis with accounting or potential method

Binomial Heap – Accounting Method

idea: save 1 coin per tree

Binomial Heap – Accounting Method

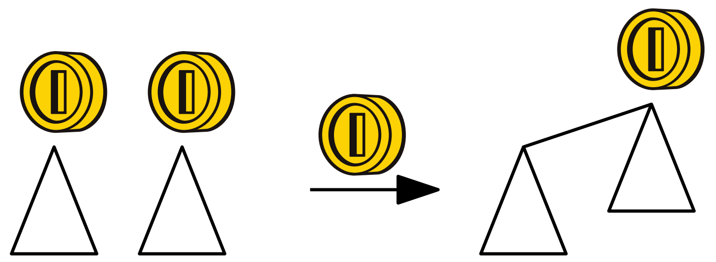
idea: save 1 coin per tree

make-0: $\hat{c} = 1$ for creating empty heap

Binomial Heap – Accounting Method

idea: save 1 coin per tree

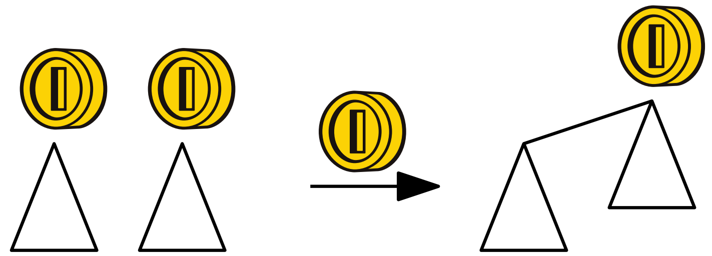
make-0: $\hat{c} = 1$ for creating empty heap
link: $\hat{c} = 0$ (pay for link with existing coin)



Binomial Heap – Accounting Method

idea: save 1 coin per tree

make-0: $\hat{c} = 1$ for creating empty heap
link: $\hat{c} = 0$ (pay for link with existing coin)

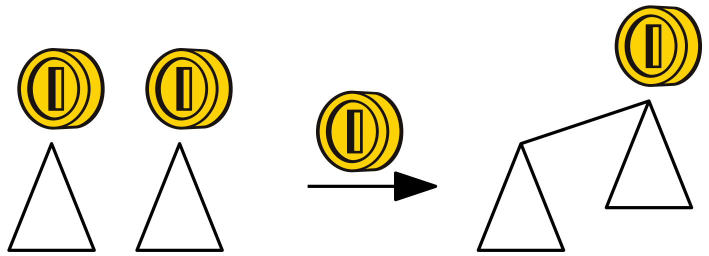


insert: $\hat{c} = 3$ (2 for make-1 + 1 for calling union + 0 for linking)

Binomial Heap – Accounting Method

idea: save 1 coin per tree

make-0: $\hat{c} = 1$ for creating empty heap
link: $\hat{c} = 0$ (pay for link with existing coin)



insert: $\hat{c} = 3$ (2 for make-1 + 1 for calling union + 0 for linking)

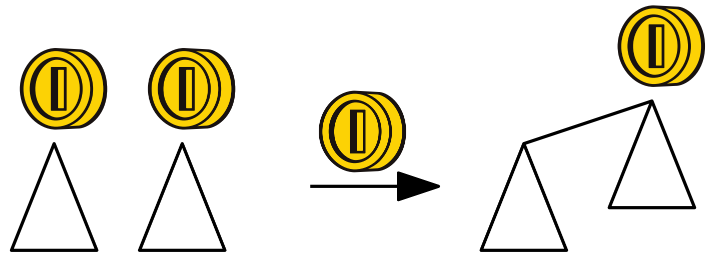
union:

merge lists: $\Theta(m)$, where $m = $ #roots$= O(\log n)$

Binomial Heap – Accounting Method

idea: save 1 coin per tree

make-0: $\hat{c} = 1$ for creating empty heap
link: $\hat{c} = 0$ (pay for link with existing coin)



insert: $\hat{c} = 3$ (2 for make-1 + 1 for calling union + 0 for linking)

union:

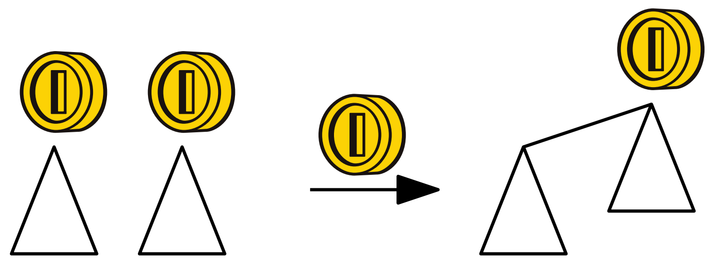
merge lists: $\Theta(m)$, where $m =$ #roots$= O(\log n)$

link trees of same degree: pay with coins

Binomial Heap – Accounting Method

idea: save 1 coin per tree

make-0: $\hat{c} = 1$ for creating empty heap
link: $\hat{c} = 0$ (pay for link with existing coin)

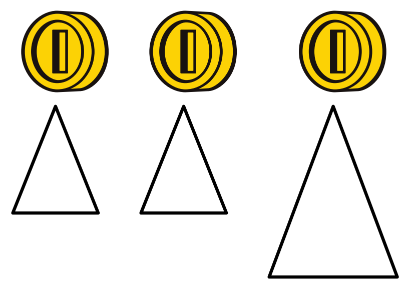


insert: $\hat{c} = 3$ (2 for make-1 + 1 for calling union + 0 for linking)

union:

merge lists: $\Theta(m)$, where $m =$ #roots$= O(\log n)$

link trees of same degree: pay with coins

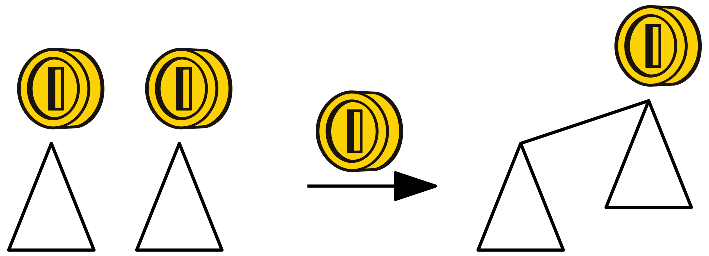


Binomial Heap – Accounting Method

idea: save 1 coin per tree

make-0: $\hat{c} = 1$ for creating empty heap

link: $\hat{c} = 0$ (pay for link with existing coin)

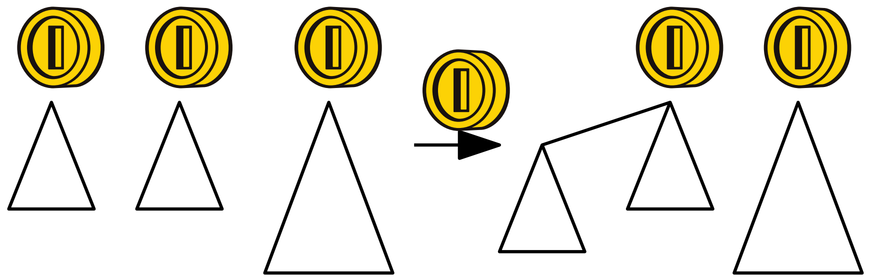


insert: $\hat{c} = 3$ (2 for make-1 + 1 for calling union + 0 for linking)

union:

merge lists: $\Theta(m)$, where $m =$ #roots$= O(\log n)$

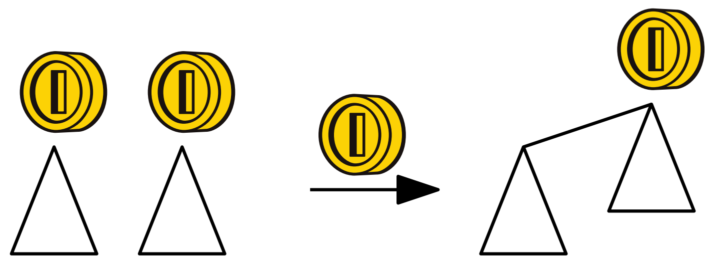
link trees of same degree: pay with coins



Binomial Heap – Accounting Method

idea: save 1 coin per tree

make-0: $\hat{c} = 1$ for creating empty heap
link: $\hat{c} = 0$ (pay for link with existing coin)

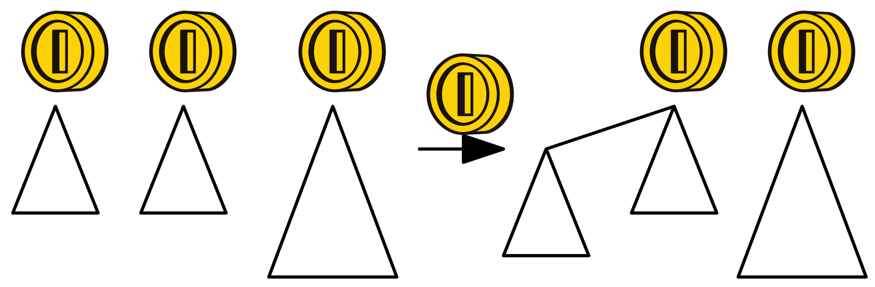


insert: $\hat{c} = 3$ (2 for make-1 + 1 for calling union + 0 for linking)

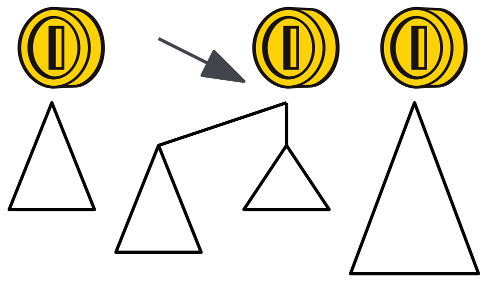
union:

merge lists: $\Theta(m)$, where $m = \#\text{roots} = O(\log n)$

link trees of same degree: pay with coins



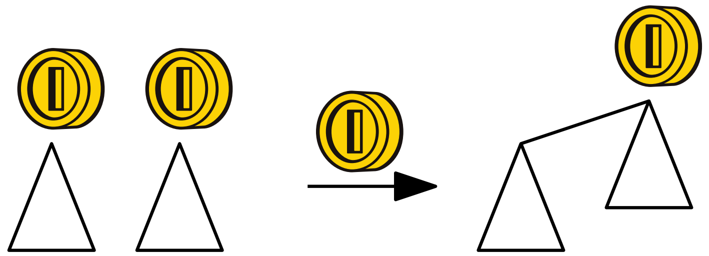
deleteMin:



Binomial Heap – Accounting Method

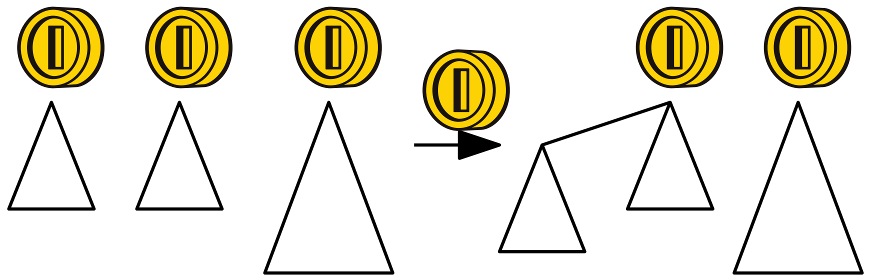
idea: save 1 coin per tree

make-0: $\hat{c} = 1$ for creating empty heap
link: $\hat{c} = 0$ (pay for link with existing coin)

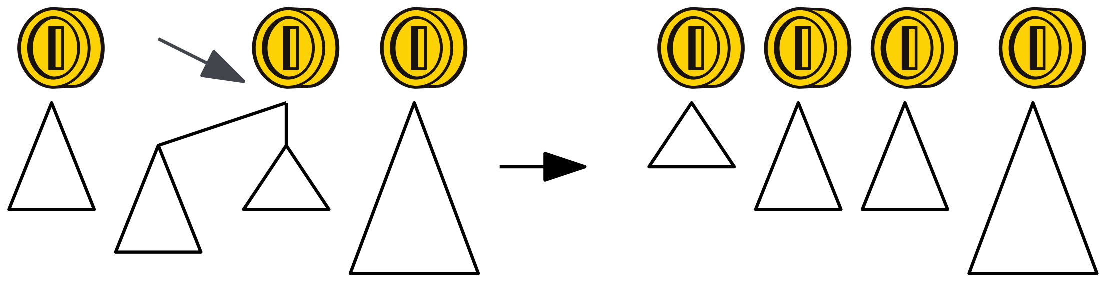


insert: $\hat{c} = 3$ (2 for make-1 + 1 for calling union + 0 for linking)

union:
merge lists: $\Theta(m)$, where $m =$ #roots$= O(\log n)$
link trees of same degree: pay with coins



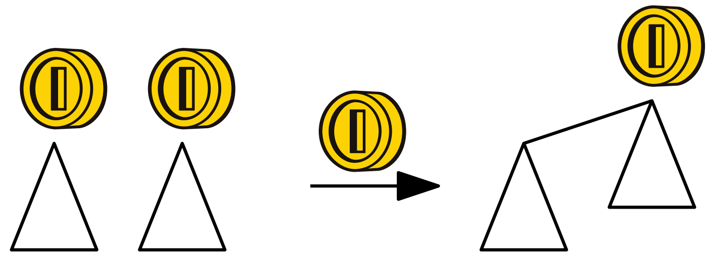
deleteMin: $\leq \log n$ children of min-node (need a coin each)



Binomial Heap – Accounting Method

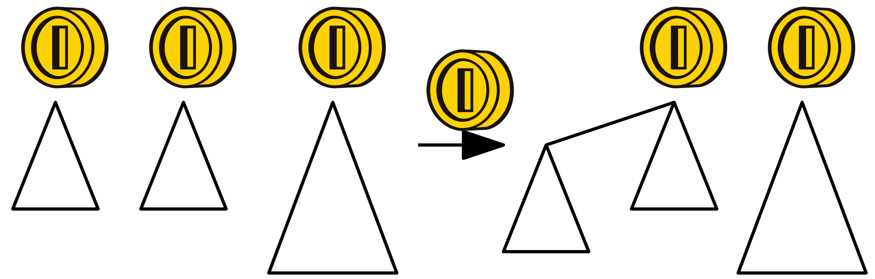
idea: save 1 coin per tree

make-0: $\hat{c} = 1$ for creating empty heap
link: $\hat{c} = 0$ (pay for link with existing coin)

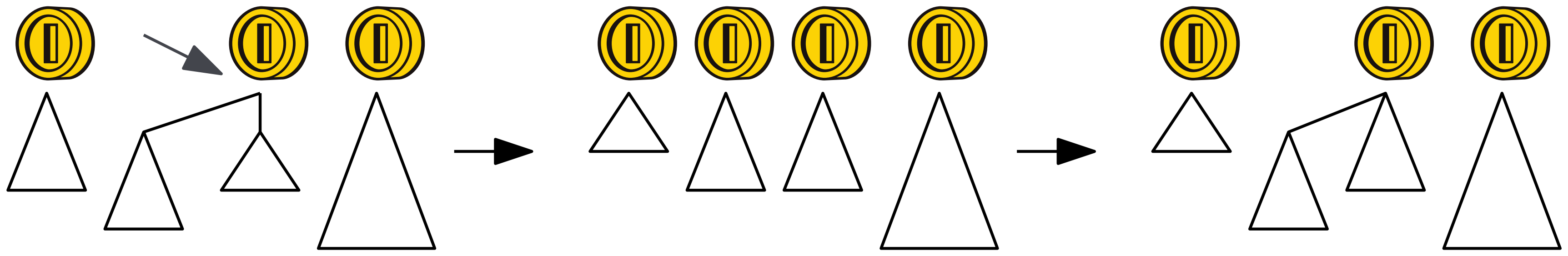


insert: $\hat{c} = 3$ (2 for make-1 + 1 for calling union + 0 for linking)

union:
merge lists: $\Theta(m)$, where $m = $ #roots$= O(\log n)$
link trees of same degree: pay with coins



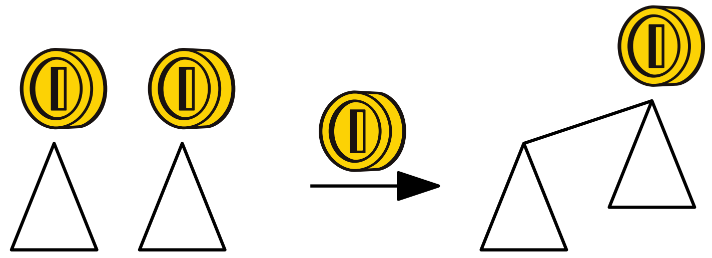
deleteMin: $\leq \log n$ children of min-node (need a coin each)
union with remaining roots in $O(\log n)$ time



Binomial Heap – Accounting Method

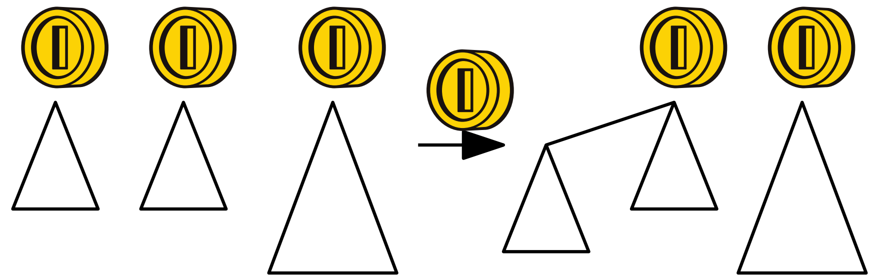
idea: save 1 coin per tree

make-0: $\hat{c} = 1$ for creating empty heap
link: $\hat{c} = 0$ (pay for link with existing coin)



insert: $\hat{c} = 3$ (2 for make-1 + 1 for calling union + 0 for linking)

union:
merge lists: $\Theta(m)$, where $m = $ #roots$= O(\log n)$
link trees of same degree: pay with coins



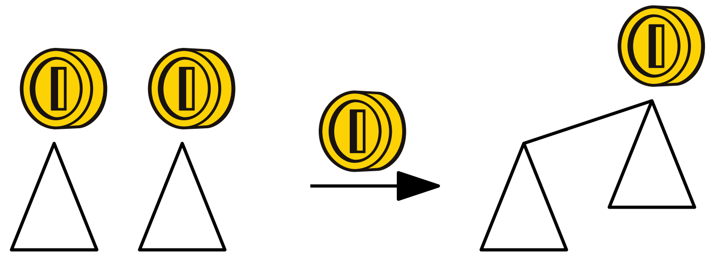
deleteMin: $\leq \log n$ children of min-node (need a coin each)
union with remaining roots in $O(\log n)$ time

decreaseKey: does not change number of trees,
$\hat{c} = $ actual cost $= O(\log n)$

Binomial Heap – Accounting Method

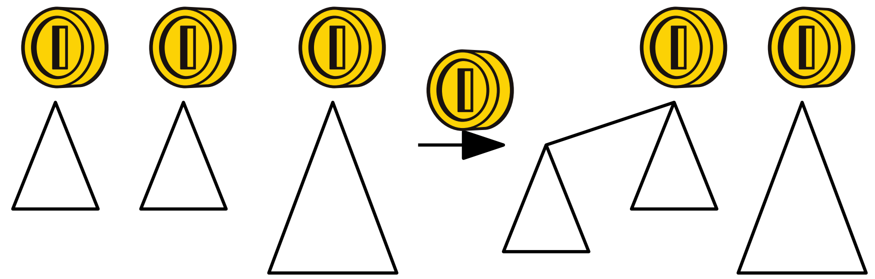
idea: save 1 coin per tree

make-0: $\hat{c} = 1$ for creating empty heap
link: $\hat{c} = 0$ (pay for link with existing coin)



insert: $\hat{c} = 3$ (2 for make-1 + 1 for calling union + 0 for linking)

union:
merge lists: $\Theta(m)$, where $m = \#$roots$= O(\log n)$
link trees of same degree: pay with coins



deleteMin: $\leq \log n$ children of min-node (need a coin each)
union with remaining roots in $O(\log n)$ time

decreaseKey: does not change number of trees,
$\hat{c} =$ actual cost $= O(\log n)$

all amortized costs as stated
+ always enough coins to pay
for actual costs

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. *(here:* $c = 1$ *works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make-0:

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make-0: $c_i = 1$, $\Delta_i = 0 \to \hat{c}_i = c_i = 1$

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make-0: $c_i = 1, \Delta_i = 0 \to \hat{c}_i = c_i = 1$

link:

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#$trees in D_i , where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make-0: $c_i = 1$, $\Delta_i = 0 \rightarrow \hat{c}_i = c_i = 1$

link: $c_i = 1$, $\Delta_i = -1 \cdot c$

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make-0: $c_i = 1, \Delta_i = 0 \rightarrow \hat{c}_i = c_i = 1$

link: $c_i = 1, \Delta_i = -1 \cdot c \quad \rightarrow \hat{c}_i = 1 - c \leq 0$

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make-0: $c_i = 1$, $\Delta_i = 0 \rightarrow \hat{c}_i = c_i = 1$

link: $c_i = 1$, $\Delta_i = -1 \cdot c \quad \rightarrow \hat{c}_i = 1 - c \leq 0$

make-1:

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make-0: $c_i = 1, \Delta_i = 0 \rightarrow \hat{c}_i = c_i = 1$

link: $c_i = 1, \Delta_i = -1 \cdot c \rightarrow \hat{c}_i = 1 - c \leq 0$

make-1: $c_i = 1, \Delta_i = 1 \cdot c$

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make-0: $c_i = 1$, $\Delta_i = 0 \rightarrow \hat{c}_i = c_i = 1$

link: $c_i = 1$, $\Delta_i = -1 \cdot c \quad \rightarrow \hat{c}_i = 1 - c \leq 0$

make-1: $c_i = 1$, $\Delta_i = \quad 1 \cdot c \quad \rightarrow \hat{c}_i = 1 + c = \Theta(1)$

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

$\Phi(D_0) = 0, \Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make-0: $c_i = 1, \Delta_i = 0 \rightarrow \hat{c}_i = c_i = 1$

link: $c_i = 1, \Delta_i = -1 \cdot c \quad \rightarrow \hat{c}_i = 1 - c \leq 0$

make-1: $c_i = 1, \Delta_i = \quad 1 \cdot c \quad \rightarrow \hat{c}_i = 1 + c = \Theta(1)$

insert:

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make-0: $c_i = 1, \Delta_i = 0 \rightarrow \hat{c}_i = c_i = 1$

link: $c_i = 1, \Delta_i = -1 \cdot c \rightarrow \hat{c}_i = 1 - c \leq 0$

make-1: $c_i = 1, \Delta_i = 1 \cdot c \rightarrow \hat{c}_i = 1 + c = \Theta(1)$

insert: a number $k > 0$ trees are “iterated through” and $k - 1$ linked

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#$trees in D_i , where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make-0: $c_i = 1$, $\Delta_i = 0 \rightarrow \hat{c}_i = c_i = 1$

link: $c_i = 1$, $\Delta_i = -1 \cdot c \rightarrow \hat{c}_i = 1 - c \leq 0$

make-1: $c_i = 1$, $\Delta_i = 1 \cdot c \rightarrow \hat{c}_i = 1 + c = \Theta(1)$

insert: a number $k > 0$ trees are "iterated through" and $k - 1$ linked

$c_i = 1 + k$

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. (*here:* $c = 1$ *works*)

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make-0: $c_i = 1, \Delta_i = 0 \to \hat{c}_i = c_i = 1$

link: $c_i = 1, \Delta_i = -1 \cdot c \to \hat{c}_i = 1 - c \leq 0$

make-1: $c_i = 1, \Delta_i = 1 \cdot c \to \hat{c}_i = 1 + c = \Theta(1)$

insert: a number $k > 0$ trees are "iterated through" and $k - 1$ linked

$$c_i = 1 + k \quad \Delta_i = \Phi(D_i) - \Phi(D_{i-1}) = c(\underset{\text{1 new}}{1} - \underset{k-1 \text{ less}}{(k-1)}) = c - c(k-1)$$

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make-0: $c_i = 1, \Delta_i = 0 \rightarrow \hat{c}_i = c_i = 1$

link: $c_i = 1, \Delta_i = -1 \cdot c \quad \rightarrow \hat{c}_i = 1 - c \leq 0$

make-1: $c_i = 1, \Delta_i = \quad 1 \cdot c \quad \rightarrow \hat{c}_i = 1 + c = \Theta(1)$

insert: a number $k > 0$ trees are "iterated through" and $k - 1$ linked

$$c_i = 1 + k \quad \Delta_i = \Phi(D_i) - \Phi(D_{i-1}) = c(\underset{\text{1 new}}{1} - (\underset{k-1 \text{ less}}{k-1})) = c - c(k-1)$$

$$\hat{c}_i = c_i + \Delta_i = 1 + k \; + c - c(k-1)$$

$$= 2 + c \; - (c-1)(k-1) \leq 2 + c = \Theta(1)$$

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. *(here:* $c = 1$ *works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

union:

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

union: $c_i = k + \ell$, where $k = \#\text{roots in merged list}$, $\ell = \#\text{links}$

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. (*here:* $c = 1$ *works*)

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

union: $c_i = k + \ell$, where $k = \#$roots in merged list, $\ell = \#$links

$k > \ell \geq 0$ and $k, \ell = O(\log n)$

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

union: $c_i = k + \ell$, where $k = \#$roots in merged list, $\ell = \#$links

$k > \ell \geq 0$ and $k, \ell = O(\log n)$

$\Delta_i = -c\,\ell$ (because of the ℓ links)

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#$trees in D_i , where $c \geq 1$ is a constant. (*here:* $c = 1$ *works*)

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

union: $c_i = k + \ell$, where $k = \#$roots in merged list, $\ell = \#$links

$k > \ell \geq 0$ and $k, \ell = O(\log n)$

$\Delta_i = -c\,\ell$ (because of the ℓ links) $\rightarrow \hat{c}_i = k + \ell - c\,\ell = O(\log n)$

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot$ #trees in D_i , where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

union: $c_i = k + \ell$, where $k =$ #roots in merged list, $\ell =$ #links

$k > \ell \geq 0$ and $k, \ell = O(\log n)$

$\Delta_i = -c\,\ell$ (because of the ℓ links) $\rightarrow \hat{c}_i = k + \ell - c\,\ell = O(\log n)$

deleteMin:

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#$trees in D_i , where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

union: $c_i = k + \ell$, where $k = \#$roots in merged list, $\ell = \#$links

$k > \ell \geq 0$ and $k, \ell = O(\log n)$

$\Delta_i = -c\,\ell$ (because of the ℓ links) $\rightarrow \hat{c}_i = k + \ell - c\,\ell = O(\log n)$

deleteMin: $c_i = \underbrace{1}_{\text{find and remove min}} + \underbrace{(r + k) + \ell}_{\text{union}} + \underbrace{\log n}_{\text{update min pointer}}$, where $r =$ degree of min

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

union: $c_i = k + \ell$, where $k = \#$roots in merged list, $\ell = \#$links

$k > \ell \geq 0$ and $k, \ell = O(\log n)$

$\Delta_i = -c\,\ell$ (because of the ℓ links) $\rightarrow \hat{c}_i = k + \ell - c\,\ell = O(\log n)$

deleteMin: $c_i = 1 + (r + k) + \ell + \log n$, where $r =$ degree of min

$\Delta_i = c \cdot (r - 1) - c \cdot \ell = c \cdot (r - \ell - 1)$

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 1$ is a constant. *(here: $c = 1$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

union: $c_i = k + \ell$, where $k = \#\text{roots in merged list}$, $\ell = \#\text{links}$

$k > \ell \geq 0$ and $k, \ell = O(\log n)$

$\Delta_i = -c\,\ell$ (because of the ℓ links) $\rightarrow \hat{c}_i = k + \ell - c\,\ell = O(\log n)$

deleteMin: $c_i = 1 + (r + k) + \ell + \log n$, where $r = $ degree of min

$\Delta_i = c \cdot (r - 1) - c \cdot \ell = c \cdot (r - \ell - 1)$

$\hat{c}_i = c_i + \Delta_i = O(\log n)$, since $k, \ell, r \leq \log n$

Binomial Heap – Potential Method

$\Phi(D_i) = c \cdot \#$trees in D_i , where $c \geq 1$ is a constant. (*here:* $c = 1$ *works*)

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

union: $c_i = k + \ell$, where $k = \#$roots in merged list, $\ell = \#$links

$k > \ell \geq 0$ and $k, \ell = O(\log n)$

$\Delta_i = -c\,\ell$ (because of the ℓ links) $\rightarrow \hat{c}_i = k + \ell - c\,\ell = O(\log n)$

deleteMin: $c_i = 1 + (r + k) + \ell + \log n$, where $r =$ degree of min

$\Delta_i = c \cdot (r - 1) - c \cdot \ell = c \cdot (r - \ell - 1)$

$\hat{c}_i = c_i + \Delta_i = O(\log n)$, since $k, \ell, r \leq \log n$

$\rightarrow$ all amortized costs as claimed

Runtimes

	worst-case	amortised
make	$O(1)$	$O(1)$
min	$O(1)$	$O(1)$
insert	$O(\log n)$	$O(1)$
delete-min	$O(\log n)$	$O(\log n)$
union	$O(\log n)$	$O(\log n)$

Amortised Analysis with accounting or potential method

Lazy Union:
only concatenate lists and link only for a delete-min

Runtimes

	worst-case	amortised	amortised lazy union
make	$O(1)$	$O(1)$	$O(1)$
min	$O(1)$	$O(1)$	$O(1)$
insert	$O(\log n)$	$O(1)$	$O(1)$
delete-min	$O(\log n)$	$O(\log n)$	$O(\log n)$
union	$O(\log n)$	$O(\log n)$	$O(1)$

Amortised Analysis with accounting or potential method

Lazy Union:

only concatenate lists and link only for a delete-min

Lazy Union

store: roots in doubly-linked list and maintain min-pointer

operations:
make-0 and link: as before

Lazy Union

store: roots in doubly-linked list and maintain min-pointer

operations:

make-0 and link: as before

union: only concatenate lists (!) + update min-pointer

Lazy Union

store: roots in doubly-linked list and maintain min-pointer

operations:
make-0 and link: as before
union: only concatenate lists (!) + update min-pointer
insert: make-1 + union

Lazy Union

store: roots in doubly-linked list and maintain min-pointer

operations:

make-0 and link: as before

union: only concatenate lists (!) + update min-pointer

insert: make-1 + union

example:

insert(5) $\overset{\downarrow}{5}$

Lazy Union

store: roots in doubly-linked list and maintain min-pointer

operations:

make-0 and link: as before

union: only concatenate lists (!) + update min-pointer

insert: make-1 + union

example:

insert(5) 5

insert(7) $7 \rightarrow 5$

Lazy Union

store: roots in doubly-linked list and maintain min-pointer

operations:

make-0 and link: as before

union: only concatenate lists (!) + update min-pointer

insert: make-1 + union

example:

insert(5) $\quad 5$

insert(7) $\quad 7 \to 5$

insert(4) $\quad 4 \to 5 \to 7$

Lazy Union

store: roots in doubly-linked list and maintain min-pointer

operations:

make-0 and link: as before

union: only concatenate lists (!) + update min-pointer

insert: make-1 + union

example:

insert(5) $\quad 5$

insert(7) $\quad 7 \to 5$

insert(4) $\quad 4 \to 5 \to 7$

union(H, $\begin{smallmatrix} & 10 \\ / & \\ 11 & \end{smallmatrix}$)

Lazy Union

store: roots in doubly-linked list and maintain min-pointer

operations:
make-0 and link: as before
union: only concatenate lists (!) + update min-pointer
insert: make-1 + union

example:

insert(5) $\quad 5$

insert(7) $\quad 7 \to 5$

insert(4) $\quad 4 \to 5 \to 7$

union(H, $\begin{matrix} 10 \\ / \\ 11 \end{matrix}$) $\quad \begin{matrix} 10 & \to 4 \to 5 \to 7 \\ / & \\ 11 & \end{matrix}$

Lazy Union

store: roots in doubly-linked list and maintain min-pointer

operations:

make-0 and link: as before

union: only concatenate lists (!) + update min-pointer

insert: make-1 + union

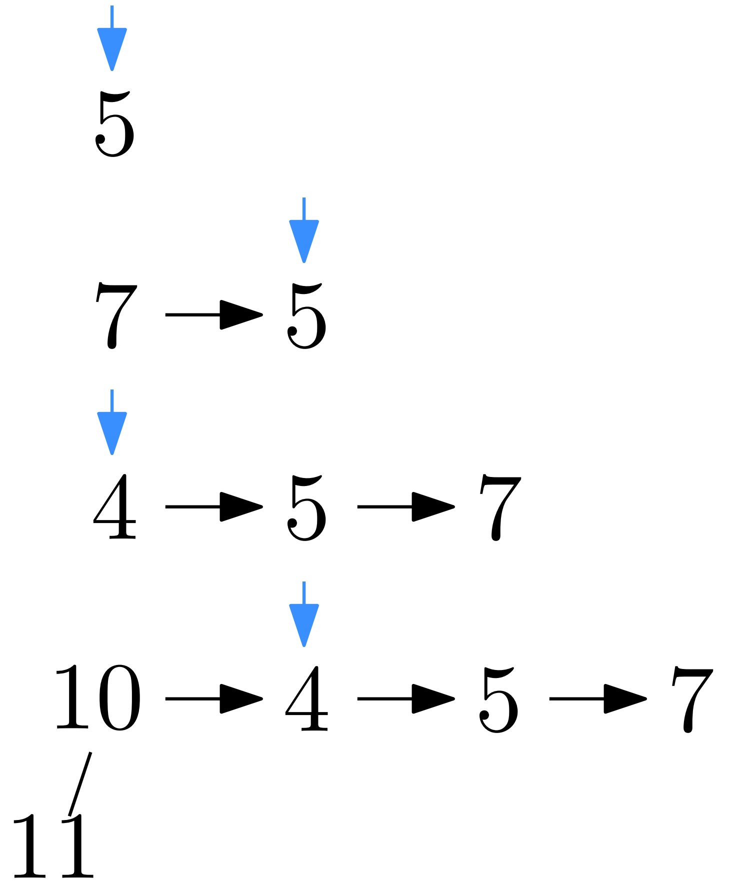
example:

insert(5)

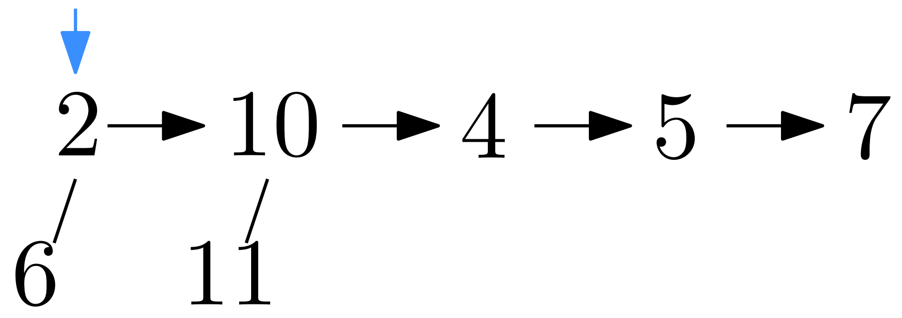
insert(7)

insert(4)

union(H, $\begin{array}{c}10\\ /\\ 11\end{array}$)



union(H, $\begin{array}{c}2\\ /\\ 6\end{array}$)



Lazy Union

store: roots in doubly-linked list and maintain min-pointer

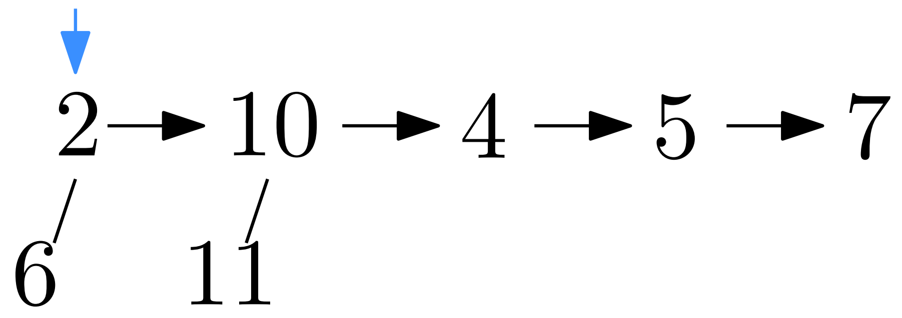
operations:

make-0 and link: as before

union: only concatenate lists (!) + update min-pointer

insert: make-1 + union

deleteMin: (*this is where the work is done*)



Lazy Union

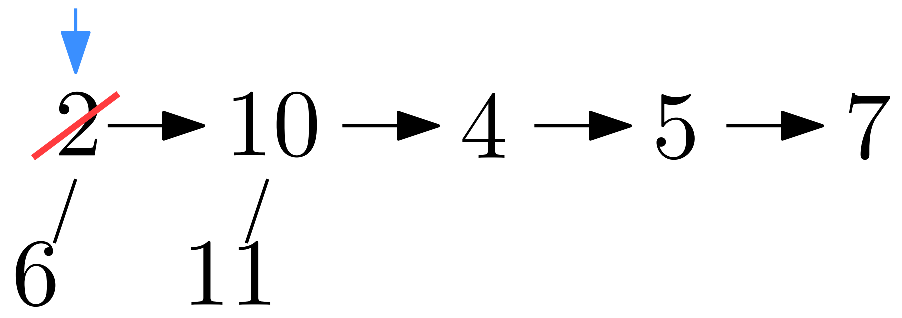
store: roots in doubly-linked list and maintain min-pointer

operations:

make-0 and link: as before

union: only concatenate lists (!) + update min-pointer

insert: make-1 + union



deleteMin: (*this is where the work is done*)

remove min-node

Lazy Union

store: roots in doubly-linked list and maintain min-pointer

operations:

make-0 and link: as before

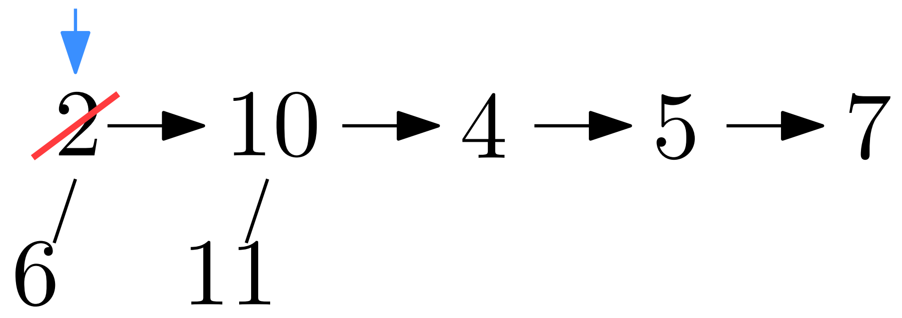
union: only concatenate lists (!) + update min-pointer

insert: make-1 + union

deleteMin: (*this is where the work is done*)

remove min-node

create array A of size $\lfloor \log_2 n \rfloor + 1$



Lazy Union

store: roots in doubly-linked list and maintain min-pointer

operations:

make-0 and link: as before

union: only concatenate lists (!) + update min-pointer

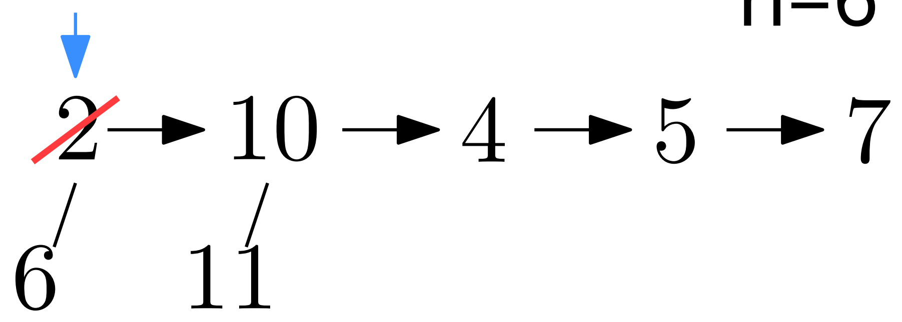
insert: make-1 + union

deleteMin: (*this is where the work is done*)

remove min-node

create array A of size $\lfloor \log_2 n \rfloor + 1$

n=6



Lazy Union

store: roots in doubly-linked list and maintain min-pointer

operations:

make-0 and link: as before

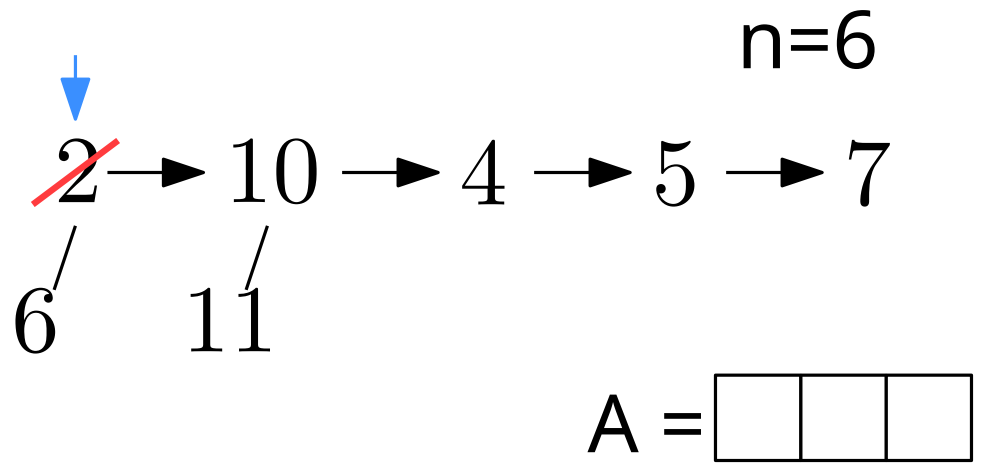
union: only concatenate lists (!) + update min-pointer

insert: make-1 + union

deleteMin: (*this is where the work is done*)

remove min-node

create array A of size $\lfloor \log_2 n \rfloor + 1$



Lazy Union

store: roots in doubly-linked list and maintain min-pointer

operations:

make-0 and link: as before

union: only concatenate lists (!) + update min-pointer

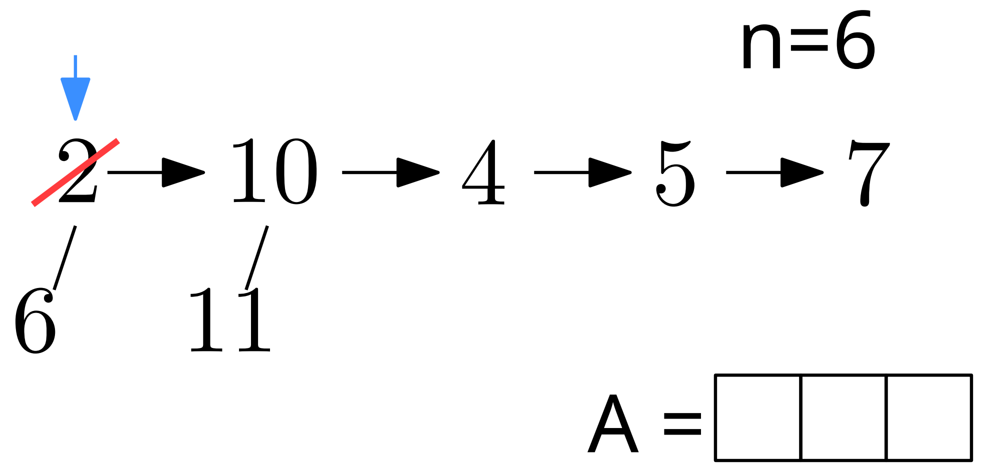
insert: make-1 + union

deleteMin: (*this is where the work is done*)

remove min-node

create array A of size $\lfloor \log_2 n \rfloor + 1$

insert trees of degree i into A[i]. If A[i] is non-empty: link + insert into A[i+1]



Lazy Union

store: roots in doubly-linked list and maintain min-pointer

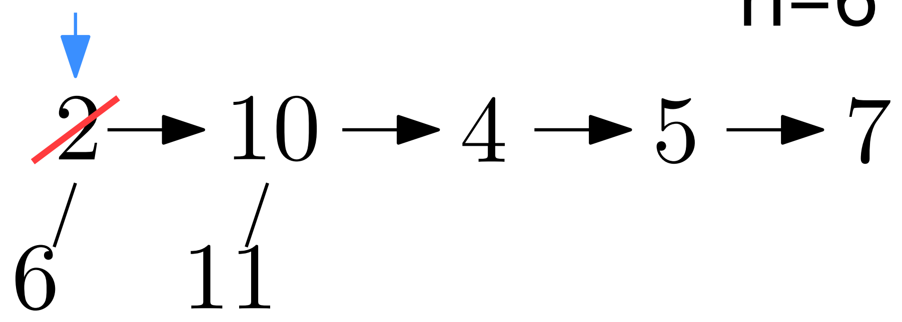
operations:

make-0 and link: as before

union: only concatenate lists (!) + update min-pointer

insert: make-1 + union

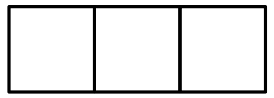
n=6



deleteMin: (*this is where the work is done*)

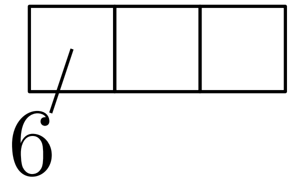
remove min-node

create array A of size $\lfloor \log_2 n \rfloor + 1$

A = 

insert trees of degree i into A[i]. If A[i] is non-empty: link + insert into A[i+1]

consolidating:



Lazy Union

store: roots in doubly-linked list and maintain min-pointer

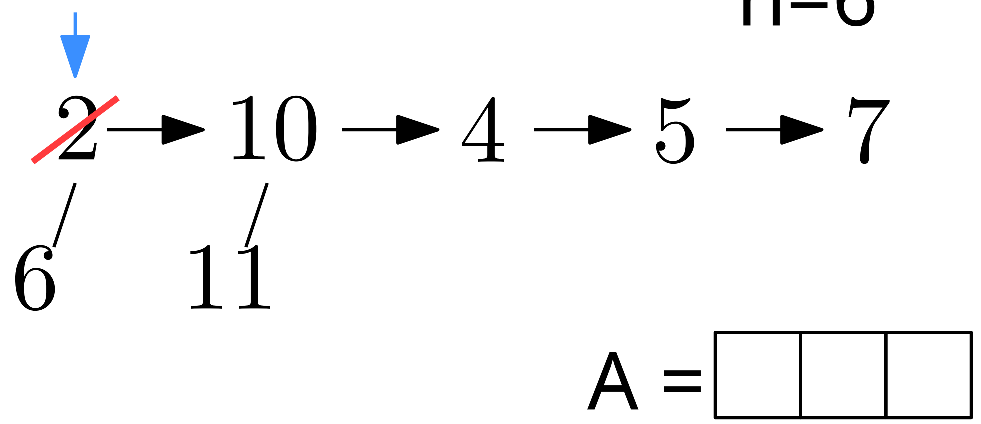
operations:

make-0 and link: as before

union: only concatenate lists (!) + update min-pointer

insert: make-1 + union

n=6



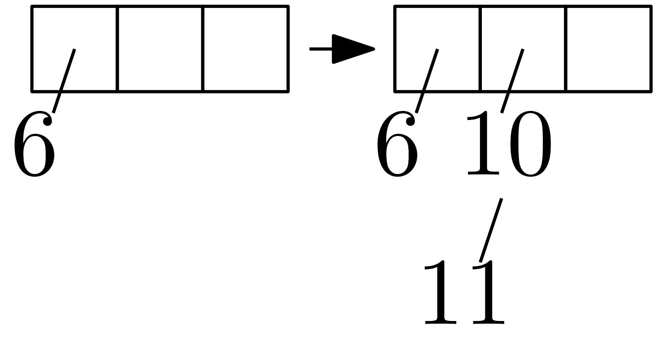
deleteMin: (*this is where the work is done*)

remove min-node

create array A of size $\lfloor \log_2 n \rfloor + 1$

insert trees of degree i into A[i]. If A[i] is non-empty: link + insert into A[i+1]

consolidating:



Lazy Union

store: roots in doubly-linked list and maintain min-pointer

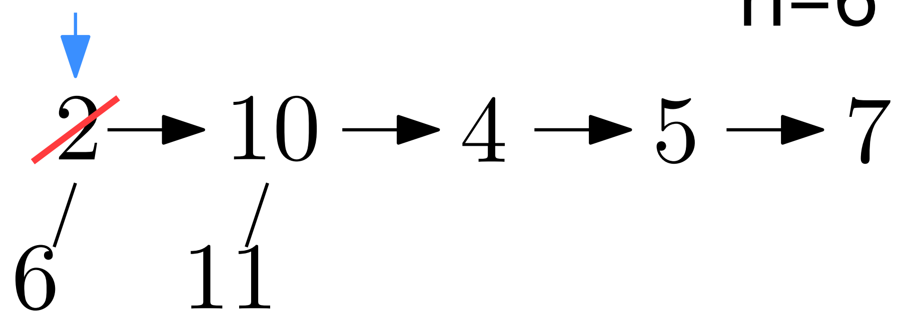
operations:

make-0 and link: as before

union: only concatenate lists (!) + update min-pointer

insert: make-1 + union

n=6



deleteMin: (*this is where the work is done*)

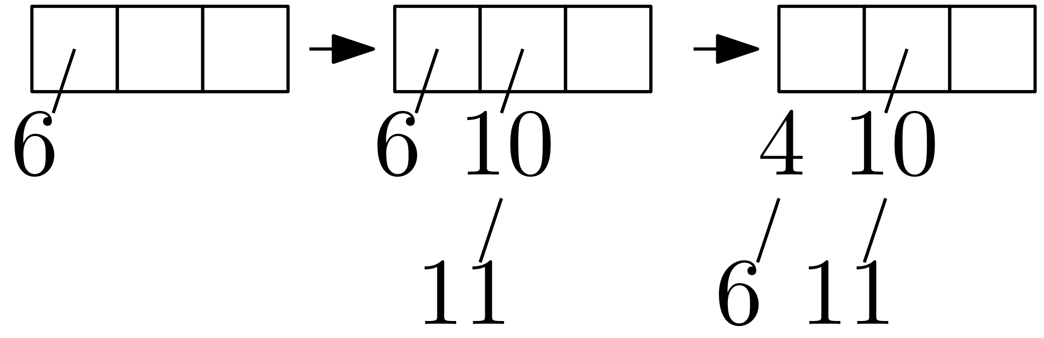
remove min-node

create array A of size $\lfloor \log_2 n \rfloor + 1$

A =

insert trees of degree i into A[i]. If A[i] is non-empty: link + insert into A[i+1]

consolidating:



Lazy Union

store: roots in doubly-linked list and maintain min-pointer

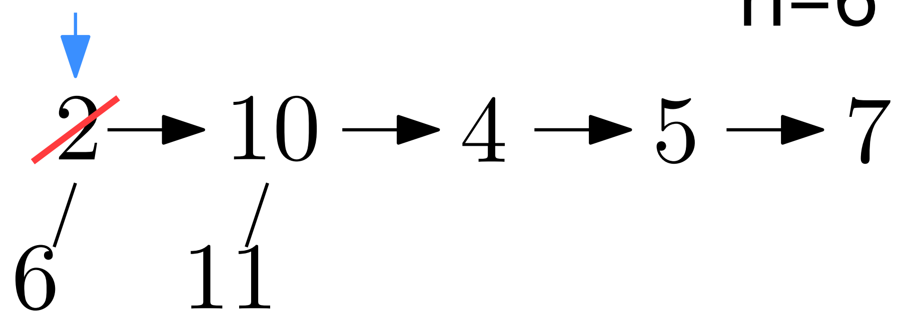
operations:

make-0 and link: as before

union: only concatenate lists (!) + update min-pointer

insert: make-1 + union

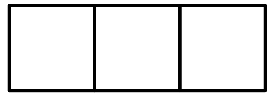
n=6



deleteMin: (*this is where the work is done*)

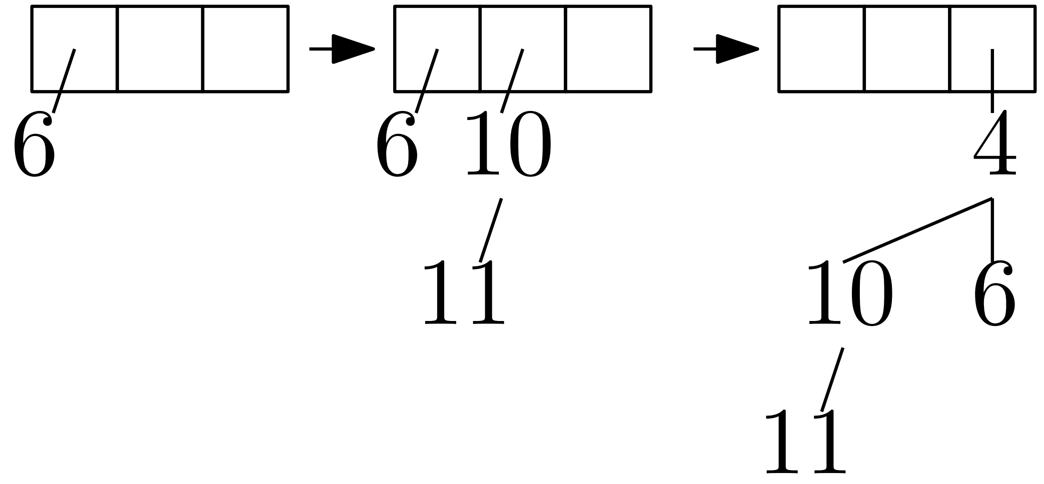
remove min-node

create array A of size $\lfloor \log_2 n \rfloor + 1$

A = 

insert trees of degree i into A[i]. If A[i] is non-empty: link + insert into A[i+1]

consolidating:



Lazy Union

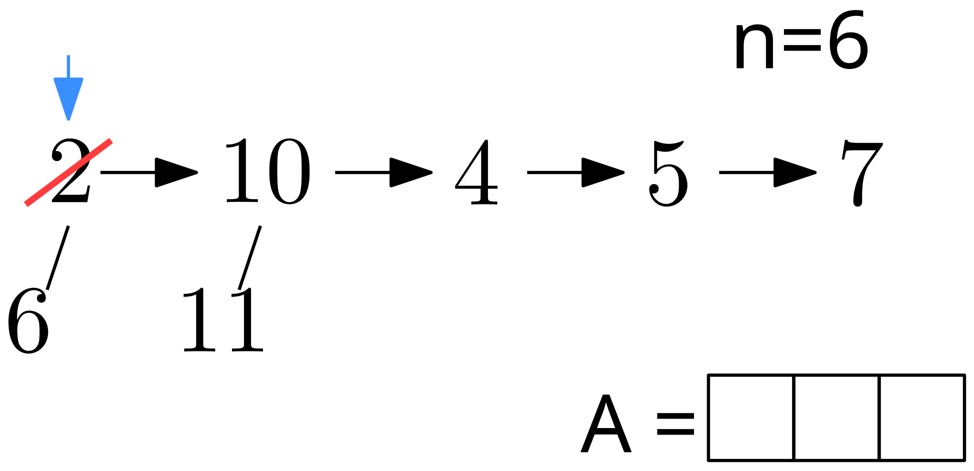
store: roots in doubly-linked list and maintain min-pointer

operations:

make-0 and link: as before

union: only concatenate lists (!) + update min-pointer

insert: make-1 + union



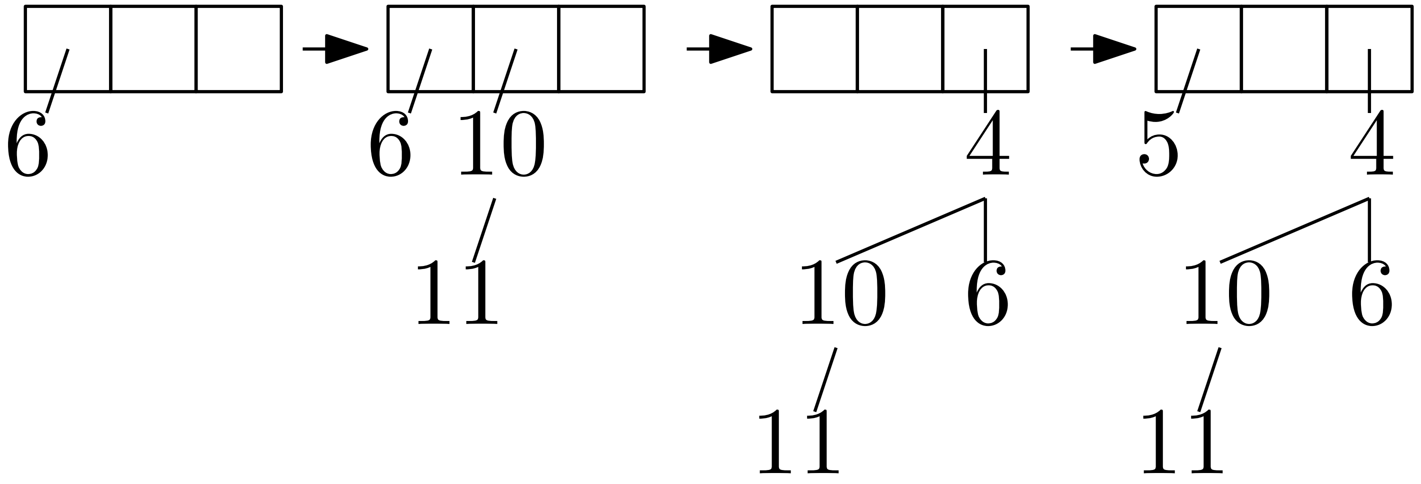
deleteMin: (*this is where the work is done*)

remove min-node

create array A of size $\lfloor \log_2 n \rfloor + 1$

insert trees of degree i into A[i]. If A[i] is non-empty: link + insert into A[i+1]

consolidating:



Lazy Union

store: roots in doubly-linked list and maintain min-pointer

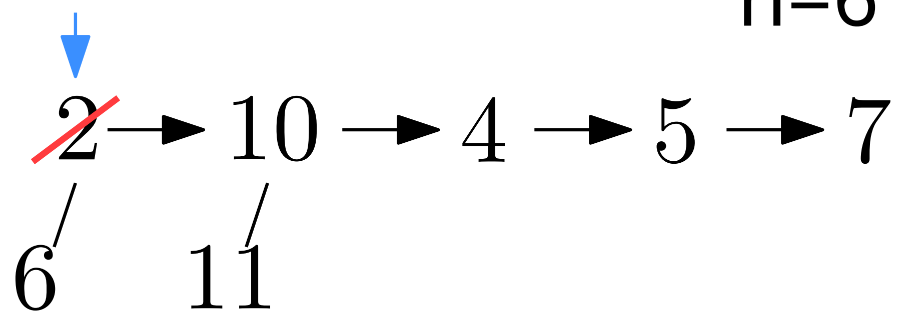
operations:

make-0 and link: as before

union: only concatenate lists (!) + update min-pointer

insert: make-1 + union

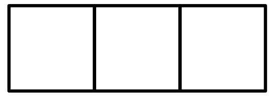
n=6



deleteMin: (*this is where the work is done*)

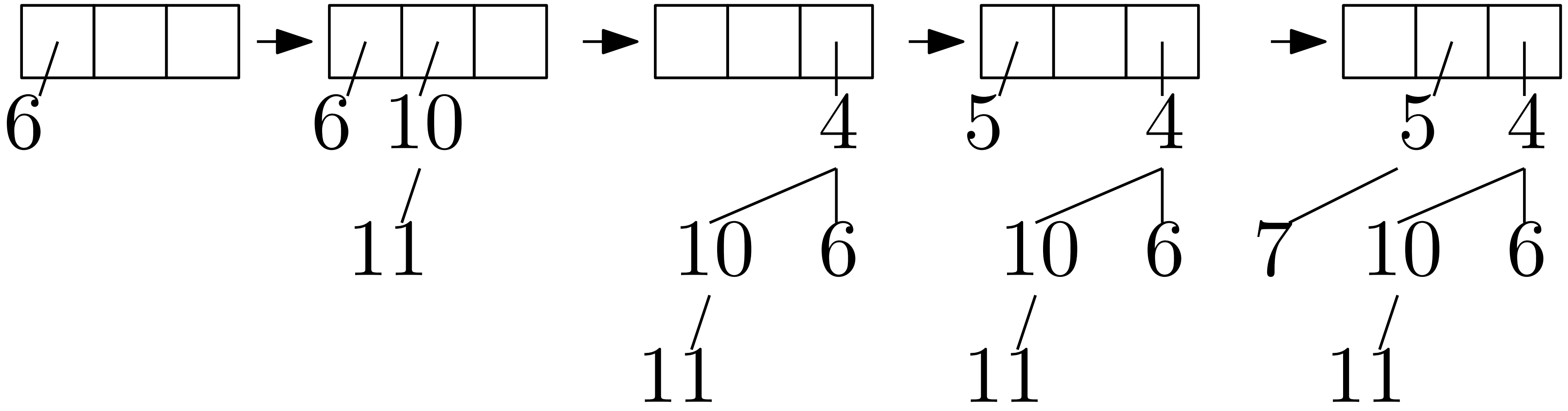
remove min-node

create array A of size $\lfloor \log_2 n \rfloor + 1$

A = 

insert trees of degree i into A[i]. If A[i] is non-empty: link + insert into A[i+1]

consolidating:



Lazy Union – Accounting Method

idea: save 2 coins per tree

Lazy Union – Accounting Method

idea: save 2 coins per tree

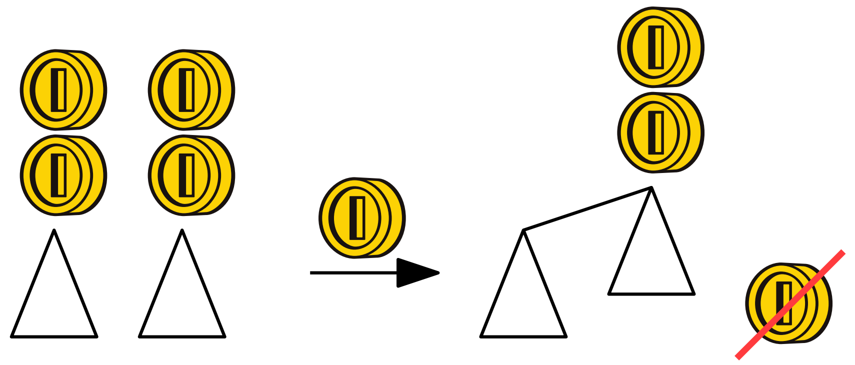
make-0: $\hat{c} = 1$ for creating empty heap

Lazy Union – Accounting Method

idea: save 2 coins per tree

make-0: $\hat{c} = 1$ for creating empty heap

link: $\hat{c} = 1 + 2 - 4 = -1$



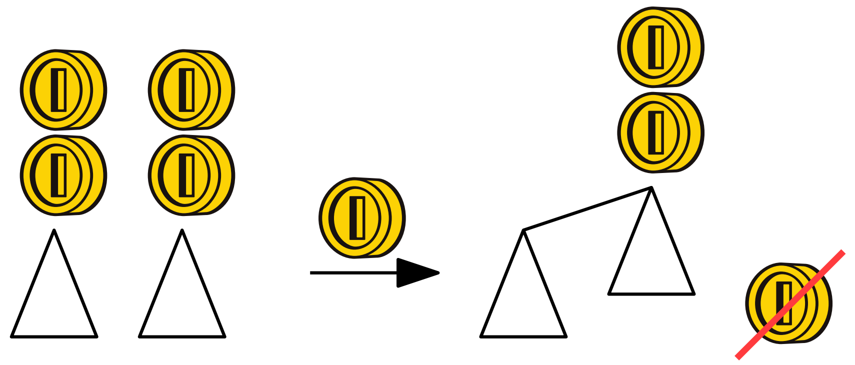
Lazy Union – Accounting Method

idea: save 2 coins per tree

make-0: $\hat{c} = 1$ for creating empty heap

link: $\hat{c} = 1 + 2 - 4 = -1$

union: $\hat{c} = 1$ (lazy)

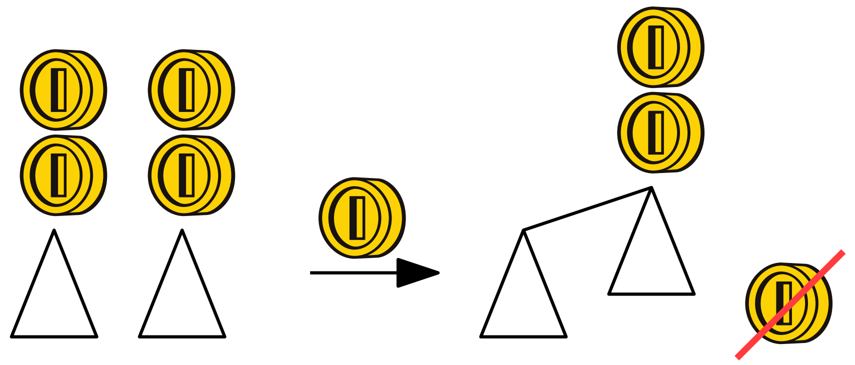


Lazy Union – Accounting Method

idea: save 2 coins per tree

make-0: $\hat{c} = 1$ for creating empty heap
link: $\hat{c} = 1 + 2 - 4 = -1$

union: $\hat{c} = 1$ (lazy)
insert: $\hat{c} = 4$ (1+2 for make-1 + 1 for union)



Lazy Union – Accounting Method

idea: save 2 coins per tree

make-0: $\hat{c} = 1$ for creating empty heap
link: $\hat{c} = 1 + 2 - 4 = -1$

union: $\hat{c} = 1$ (lazy)
insert: $\hat{c} = 4$ (1+2 for make-1 + 1 for union)

deleteMin:

Lazy Union – Accounting Method

idea: save 2 coins per tree

make-0: $\hat{c} = 1$ for creating empty heap
link: $\hat{c} = 1 + 2 - 4 = -1$

union: $\hat{c} = 1$ (lazy)
insert: $\hat{c} = 4$ (1+2 for make-1 + 1 for union)

deleteMin:

pay at most $2 \log n$ coins for children of min

Lazy Union – Accounting Method

idea: save 2 coins per tree

make-0: $\hat{c} = 1$ for creating empty heap
link: $\hat{c} = 1 + 2 - 4 = -1$

union: $\hat{c} = 1$ (lazy)
insert: $\hat{c} = 4$ (1+2 for make-1 + 1 for union)

deleteMin:

pay at most $2 \log n$ coins for children of min

actual cost: $t + \ell + O(\log n)$, where $t = \#$trees to start (after removing min)
and $\ell = \#$links

Lazy Union – Accounting Method

idea: save 2 coins per tree

make-0: $\hat{c} = 1$ for creating empty heap
link: $\hat{c} = 1 + 2 - 4 = -1$

union: $\hat{c} = 1$ (lazy)
insert: $\hat{c} = 4$ (1+2 for make-1 + 1 for union)

deleteMin:

pay at most $2 \log n$ coins for children of min

actual cost: $t + \ell + O(\log n)$, where $t = \#$trees to start (after removing min)
and $\ell = \#$links

ℓ links free up 2ℓ coins (ℓ of these pay for link itself, see above)

$$t + \ell = (t - \ell) + 2\ell$$

Lazy Union – Accounting Method

idea: save 2 coins per tree

make-0: $\hat{c} = 1$ for creating empty heap
link: $\hat{c} = 1 + 2 - 4 = -1$

union: $\hat{c} = 1$ (lazy)
insert: $\hat{c} = 4$ (1+2 for make-1 + 1 for union)

deleteMin:

pay at most $2 \log n$ coins for children of min

actual cost: $t + \ell + O(\log n)$, where $t =$ #trees to start (after removing min)
and $\ell =$ #links

ℓ links free up 2ℓ coins (ℓ of these pay for link itself, see above)

afterwards: #trees $\leq \log n + 1 \rightarrow t - \ell \leq \log n + 1$

$t + \ell = (t - \ell) + 2\ell$

Lazy Union – Accounting Method

idea: save 2 coins per tree

make-0: $\hat{c} = 1$ for creating empty heap
link: $\hat{c} = 1 + 2 - 4 = -1$

union: $\hat{c} = 1$ (lazy)
insert: $\hat{c} = 4$ (1+2 for make-1 + 1 for union)

deleteMin:

pay at most $2 \log n$ coins for children of min

actual cost: $t + \ell + O(\log n)$, where $t =$ #trees to start (after removing min)
and $\ell =$ #links

ℓ links free up 2ℓ coins (ℓ of these pay for link itself, see above)

afterwards: #trees $\leq \log n + 1 \rightarrow t - \ell \leq \log n + 1$

$t + \ell = (t - \ell) + 2\ell \leq 2\ell + \log n + 1$

Lazy Union – Accounting Method

idea: save 2 coins per tree

make-0: $\hat{c} = 1$ for creating empty heap
link: $\hat{c} = 1 + 2 - 4 = -1$

union: $\hat{c} = 1$ (lazy)
insert: $\hat{c} = 4$ (1+2 for make-1 + 1 for union)

deleteMin:

pay at most $2 \log n$ coins for children of min

actual cost: $t + \ell + O(\log n)$, where $t = \#$trees to start (after removing min)
and $\ell = \#$links

ℓ links free up 2ℓ coins (ℓ of these pay for link itself, see above)

afterwards: $\#$trees $\leq \log n + 1 \rightarrow t - \ell \leq \log n + 1$

$t + \ell = (t - \ell) + 2\ell \leq 2\ell + \log n + 1 \;\; \rightarrow \hat{c}_i = O(\log n)$ coins suffice

Lazy Union – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$

Lazy Union – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 2$ is a constant. *(here:* $c = 2$ *works)*

Lazy Union – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 2$ is a constant. *(here: $c = 2$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

Lazy Union – Potential Method

$\Phi(D_i) = c \cdot \#$trees in D_i , where $c \geq 2$ is a constant. *(here: $c = 2$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

Lazy Union – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 2$ is a constant. *(here: $c = 2$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make, link, insert: as for regular union

Lazy Union – Potential Method

$\Phi(D_i) = c \cdot \#$trees in D_i , where $c \geq 2$ is a constant. *(here: $c = 2$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make, link, insert: as for regular union

union:

Lazy Union – Potential Method

$\Phi(D_i) = c \cdot \#$trees in D_i , where $c \geq 2$ is a constant. *(here: $c = 2$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make, link, insert: as for regular union

union: $c_i = 1, \Delta_i = 0 \rightarrow \hat{c}_i = 1$

Lazy Union – Potential Method

$\Phi(D_i) = c \cdot \#$trees in D_i , where $c \geq 2$ is a constant. *(here: $c = 2$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make, link, insert: as for regular union

union: $c_i = 1, \Delta_i = 0 \rightarrow \hat{c}_i = 1$

deleteMin:

Lazy Union – Potential Method

$\Phi(D_i) = c \cdot$ #trees in D_i , where $c \geq 2$ is a constant. (*here:* $c = 2$ *works*)

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make, link, insert: as for regular union

union: $c_i = 1, \Delta_i = 0 \rightarrow \hat{c}_i = 1$

deleteMin: $c_i = t + \ell + O(\log n)$, where $t =$ #trees to start (after removing min)

and $\ell =$ #links

add to array
+ link

everything else

Lazy Union – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 2$ is a constant. *(here: $c = 2$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make, link, insert: as for regular union

union: $c_i = 1, \Delta_i = 0 \to \hat{c}_i = 1$

deleteMin: $c_i = t + \ell + O(\log n)$, where $t = \#$trees to start (after removing min)

and $\ell = \#$links

$$\Delta_i \leq c \log n - c \cdot \ell$$

new trees: children of min

trees removed by linking

Lazy Union – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 2$ is a constant. *(here: $c = 2$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make, link, insert: as for regular union

union: $c_i = 1, \Delta_i = 0 \to \hat{c}_i = 1$

deleteMin: $c_i = t + \ell + O(\log n)$, where $t = \#$trees to start (after removing min)
and $\ell = \#$links

$$\Delta_i \leq c \log n - c \cdot \ell$$

$$\hat{c}_i = c_i + \Delta_i = t + \ell - c \cdot \ell + O(\log n)$$

Lazy Union – Potential Method

$\Phi(D_i) = c \cdot \#$trees in D_i , where $c \geq 2$ is a constant. *(here: $c = 2$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make, link, insert: as for regular union

union: $c_i = 1, \Delta_i = 0 \rightarrow \hat{c}_i = 1$

deleteMin: $c_i = t + \ell + O(\log n)$, where $t = \#$trees to start (after removing min)
and $\ell = \#$links

$$\Delta_i \leq c \log n - c \cdot \ell$$

$$\begin{aligned} \hat{c}_i = c_i + \Delta_i &= t + \ell - c \cdot \ell + O(\log n) \\ &\leq t - \ell + O(\log n) \end{aligned}$$

Lazy Union – Potential Method

$\Phi(D_i) = c \cdot \#$trees in D_i , where $c \geq 2$ is a constant. *(here: $c = 2$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

make, link, insert: as for regular union

union: $c_i = 1, \Delta_i = 0 \rightarrow \hat{c}_i = 1$

deleteMin: $c_i = t + \ell + O(\log n)$, where $t = \#$trees to start (after removing min)
and $\ell = \#$links

$$\Delta_i \leq c \log n - c \cdot \ell$$

$$\begin{aligned} \hat{c}_i = c_i + \Delta_i &= t + \ell - c \cdot \ell + O(\log n) \\ &\leq t - \ell + O(\log n) = O(\log n), \end{aligned}$$

since $t - \ell = \#$trees at end $= O(\log n)$

Lazy Union – Potential Method

$\Phi(D_i) = c \cdot \#\text{trees in } D_i$, where $c \geq 2$ is a constant. *(here: $c = 2$ works)*

$\Phi(D_0) = 0$, $\Phi(D_i) \geq 0 = \Phi(D_0)$ ✓

$\hat{c}_i = c_i + \Delta_i$, where $\Delta_i = \Phi(D_i) - \Phi(D_{i-1})$

all amortized costs as claimed

make, link, insert: as for regular union

union: $c_i = 1, \Delta_i = 0 \rightarrow \hat{c}_i = 1$

deleteMin: $c_i = t + \ell + O(\log n)$, where $t = \#$trees to start (after removing min) and $\ell = \#$links

$$\Delta_i \leq c \log n - c \cdot \ell$$

$$\hat{c}_i = c_i + \Delta_i = t + \ell - c \cdot \ell + O(\log n)$$
$$\leq t - \ell + O(\log n) = O(\log n),$$

since $t - \ell = \#\text{trees at end} = O(\log n)$

Runtimes

	worst-case	amortised	amortised lazy union
make	$O(1)$	$O(1)$	$O(1)$
min	$O(1)$	$O(1)$	$O(1)$
insert	$O(\log n)$	$O(1)$	$O(1)$
delete-min	$O(\log n)$	$O(\log n)$	$O(\log n)$
union	$O(\log n)$	$O(\log n)$	$O(1)$

Amortised Analysis with accounting or potential method

Lazy Union:
only concatenate lists and link only for a delete-min

Runtimes

	worst-case	amortised	amortised lazy union
make	$O(1)$	$O(1)$	$O(1)$
min	$O(1)$	$O(1)$	$O(1)$
insert	$O(\log n)$	$O(1)$	$O(1)$
delete-min	$O(\log n)$	$O(\log n)$	$O(\log n)$
union	$O(\log n)$	$O(\log n)$	$O(1)$

Amortised Analysis with accounting or potential method

Lazy Union:

only concatenate lists and link only for a delete-min

but decreaseKey still costs $O(\log n)$ time!

Runtimes

	worst-case	amortised	amortised lazy union
make	$O(1)$	$O(1)$	$O(1)$
min	$O(1)$	$O(1)$	$O(1)$
insert	$O(\log n)$	$O(1)$	$O(1)$
delete-min	$O(\log n)$	$O(\log n)$	$O(\log n)$
union	$O(\log n)$	$O(\log n)$	$O(1)$

Amortised Analysis with accounting or potential method

Lazy Union:

only concatenate lists and link only for a delete-min

but decreaseKey still costs $O(\log n)$ time! $\longrightarrow$ Fibonacci Heaps !