Priority Queues

30th September 2020

Priority Queues

- Binary heaps
- Leftist heaps
- Binomial heaps
- Fibonacci heaps

Priority queues are important in, among other things, operating systems (process control in multitasking systems), search algorithms (A, A*, D*, etc.), and simulation.

Priority Queues

Priority queues are data structures that hold elements with some kind of priority (*key*) in a queue-like structure, implementing the following operations:

- **`insert()`** – Inserting an element into the queue.
- **`deleteMin()`** – Removing the element with the highest priority.

And maybe also:

- **`buildHeap()`** – Build a queue from a set (>1) of elements.
- **`increaseKey()/DecreaseKey()`** – Change priority.
- **`delete()`** – Removing an element from the queue.
- **`merge()`** – Merge two queues.

Priority Queues

An unsorted linked list can be used. **`insert()`** inserts an element at the head of the list ($O(1)$), and **`deleteMin()`** searches the list for the element with the highest priority and removes it ($O(n)$).

A sorted list can also be used (reversed running times).

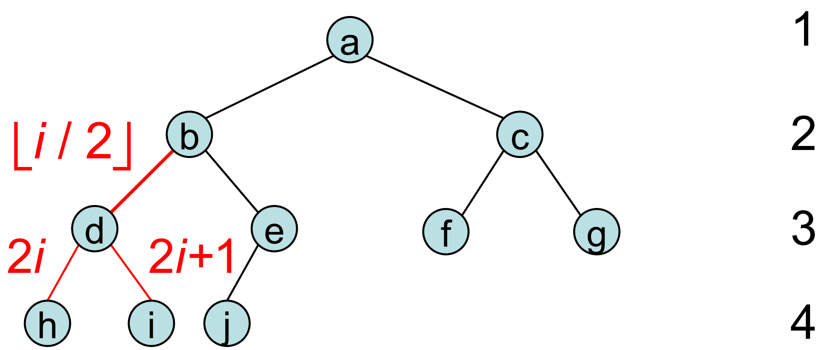
– Not very efficient implementations.

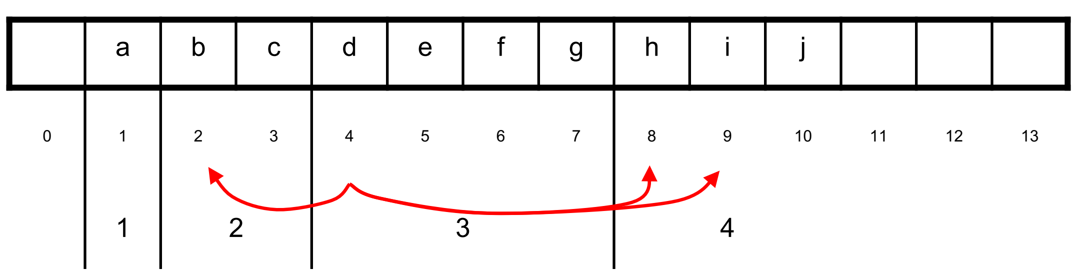
To make an efficient priority queue, it is enough to keeps the elements "almost sorted".

Binary heaps

A *binary heap* is organized as a complete binary tree. (All levels are full, except possibly the last.)

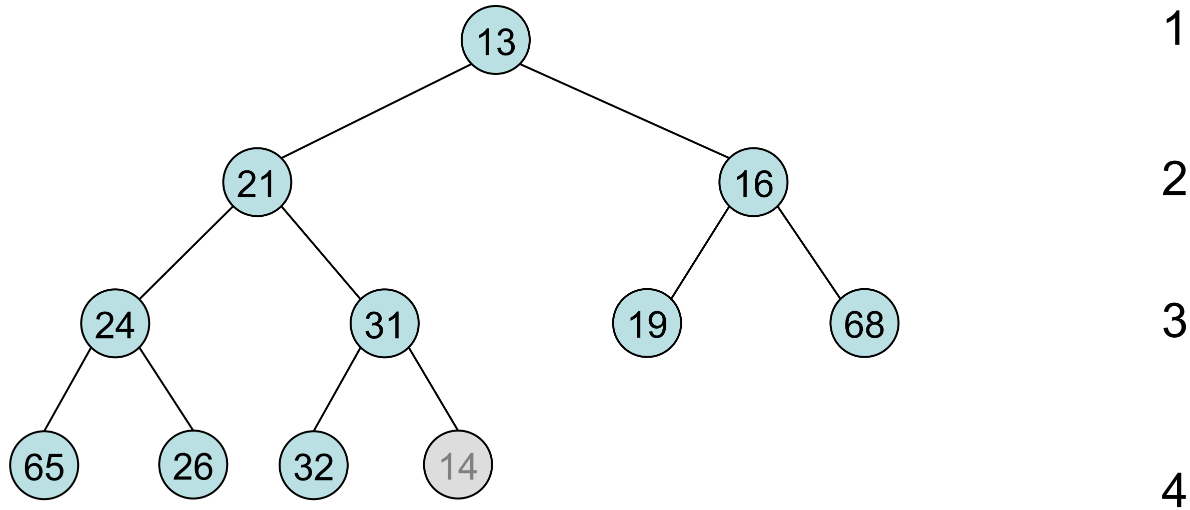
In a *binary heap* the element in the root must have a key less than or equal to the key of its children, in addition each sub-tree must be a binary heap.





Binary heaps

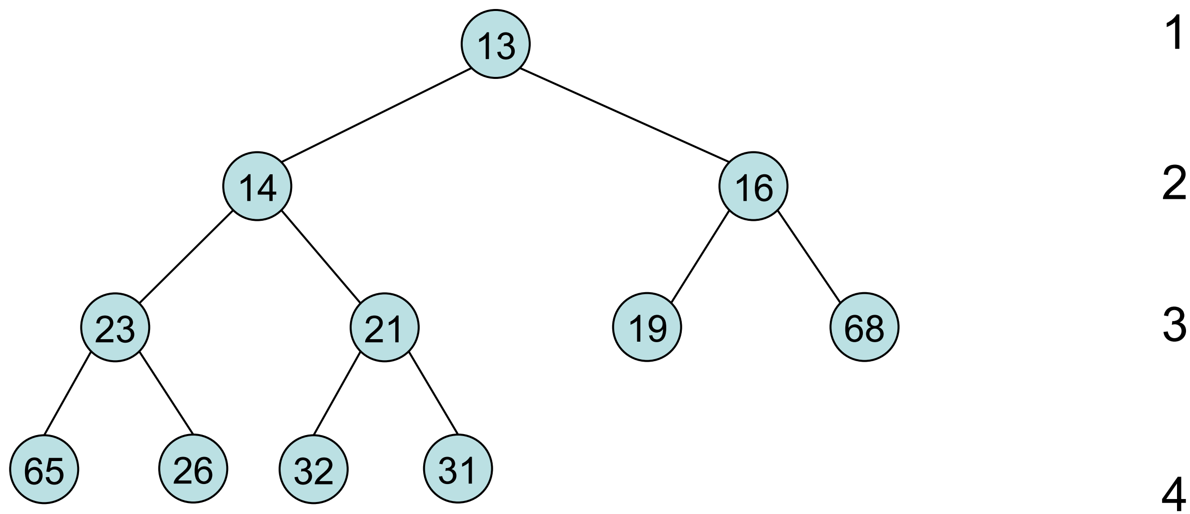
`insert(14)`



	13	21	16	24	31	19	68	65	26	32	14		
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2		3				4					

Binary heaps

`insert(14)`

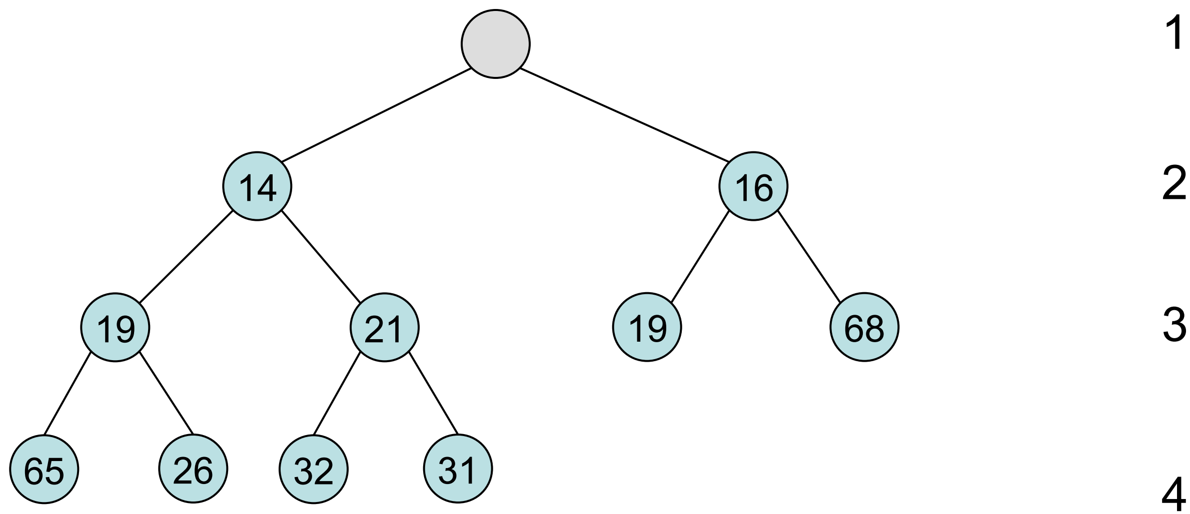


	13	14	16	24	21	19	68	65	26	32	31		
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2		3				4					

"**`percolateUp()`**"

Binary heaps

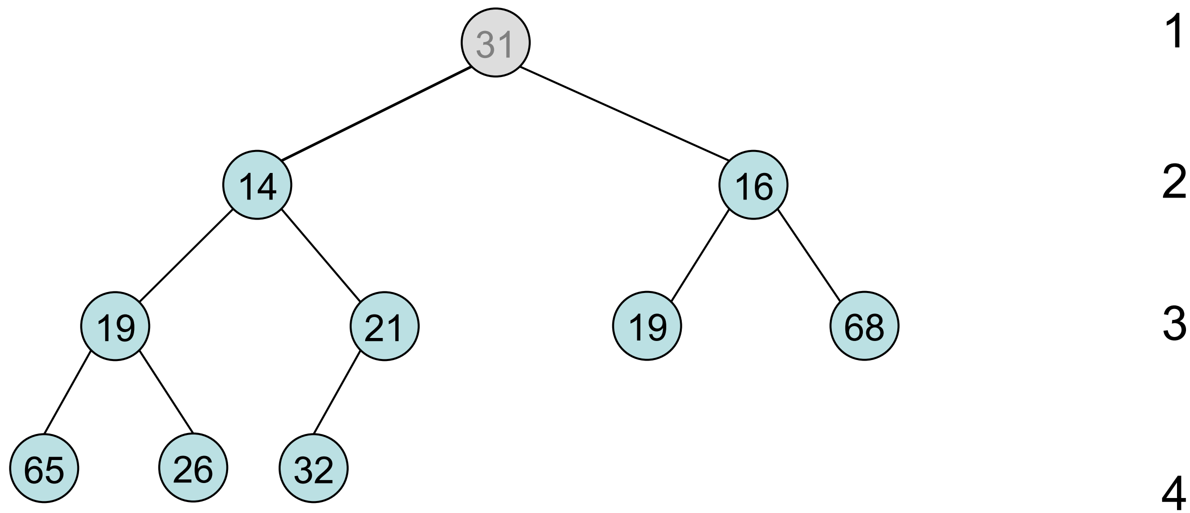
`deleteMin()`



0	1	2	3	4	5	6	7	8	9	10	11	12	13
		14	16	19	21	19	68	65	26	32	31		
	1	2		3				4					

Binary heaps

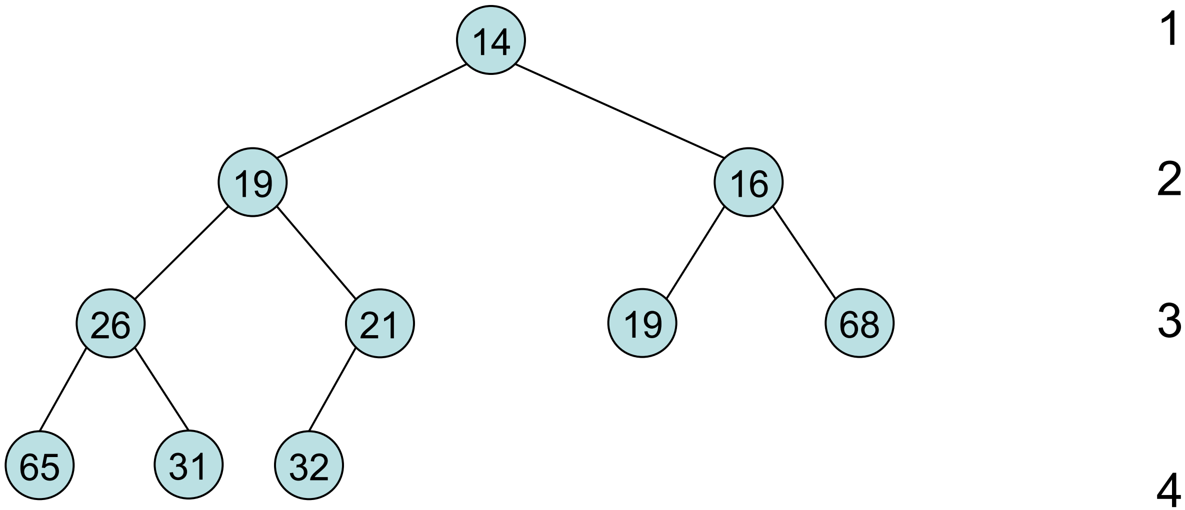
`deleteMin()`



	31	14	16	19	21	19	68	65	26	32			
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2		3				4					

Binary heaps

`deleteMin()`



	14	19	16	19	21	26	68	65	31	32			
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2			3				4				

”`percolateDown()`”

Binary heaps

	worst case	average
`insert()`	$O(\log N)$	$O(1)$
`deleteMin()`	$O(\log N)$	$O(\log N)$

`buildHeap()`	$O(N)$

(Insert elements into the array unsorted, and run **`percolateDown()`** on each root in the resulting heap (the tree), bottom up)

(The sum of the heights of a binary tree with N nodes is $O(N)$.)

`merge()`	$O(N)$

(N = number of elements)

Leftist heaps

To implement an efficient `merge()`, we move away from arrays, and implement so-called *leftist heaps* as pure trees.

The idea is to make the heap (the tree) as skewed as possible, and do all the work on a short (right) branch, leaving the long (left) branch untouched.

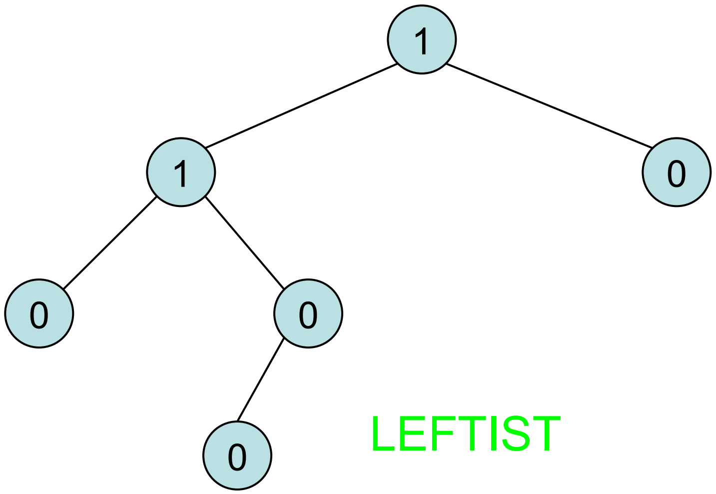
A *leftist heap* is still a binary tree with the heap structure (key in root is lower than key in children), but with an extra skewness requirement.

For all nodes *X* in our tree, we define the *null-path-length*(*X*) as the distance from *X* to a descendant with less than two children (*i.e.* 0 or 1).

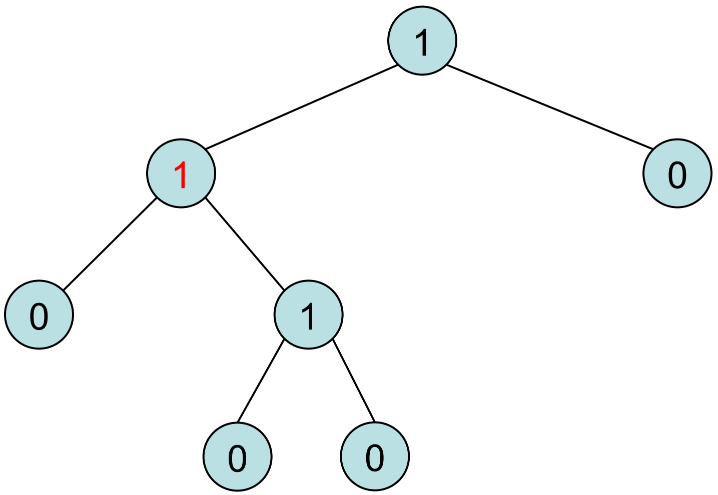
The skewness requirement is that for every node the null path length of its left child be at least as large as the null path length of the right child.

For the empty tree we define the *null-path-length* to be -1, as a special case.

Leftist heaps



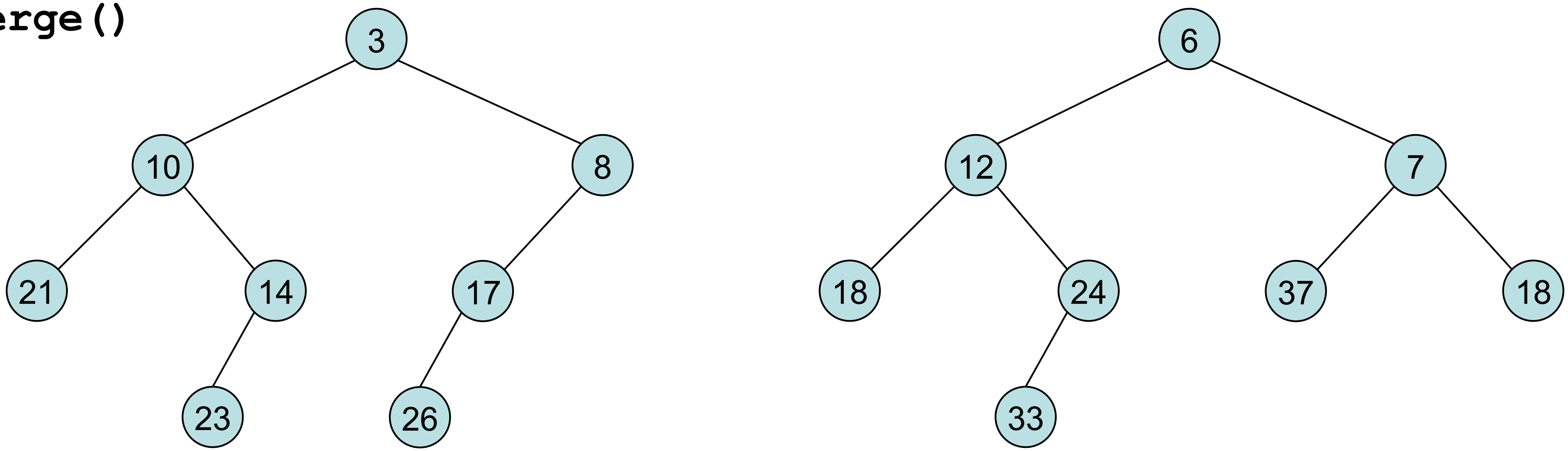
LEFTIST



NOT LEFTIST

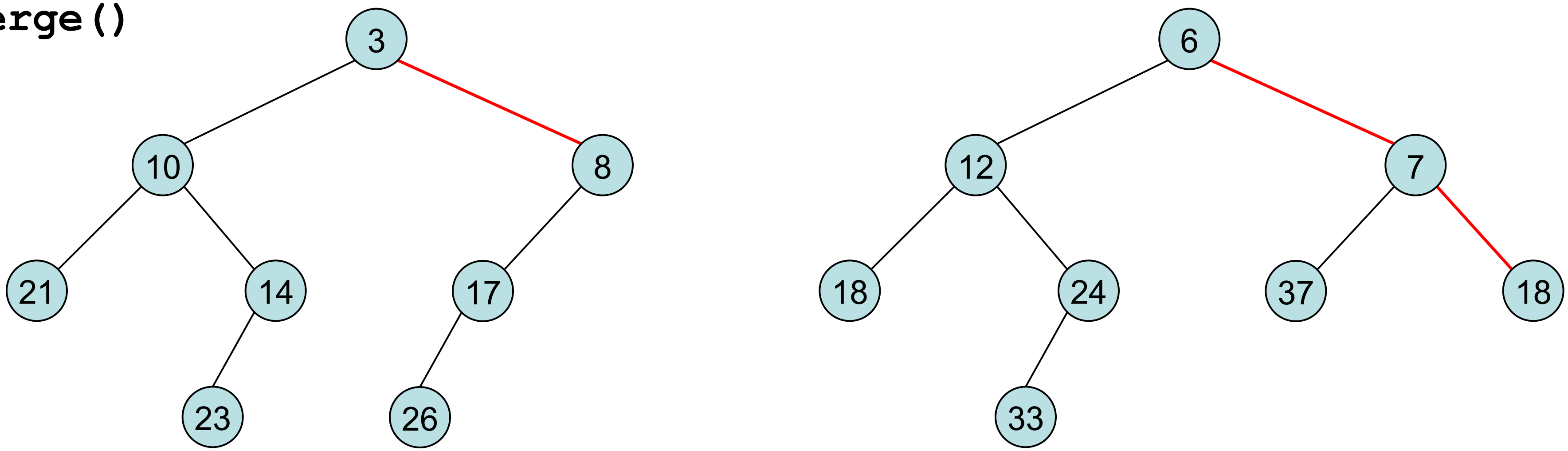
Leftist heaps

`merge()`



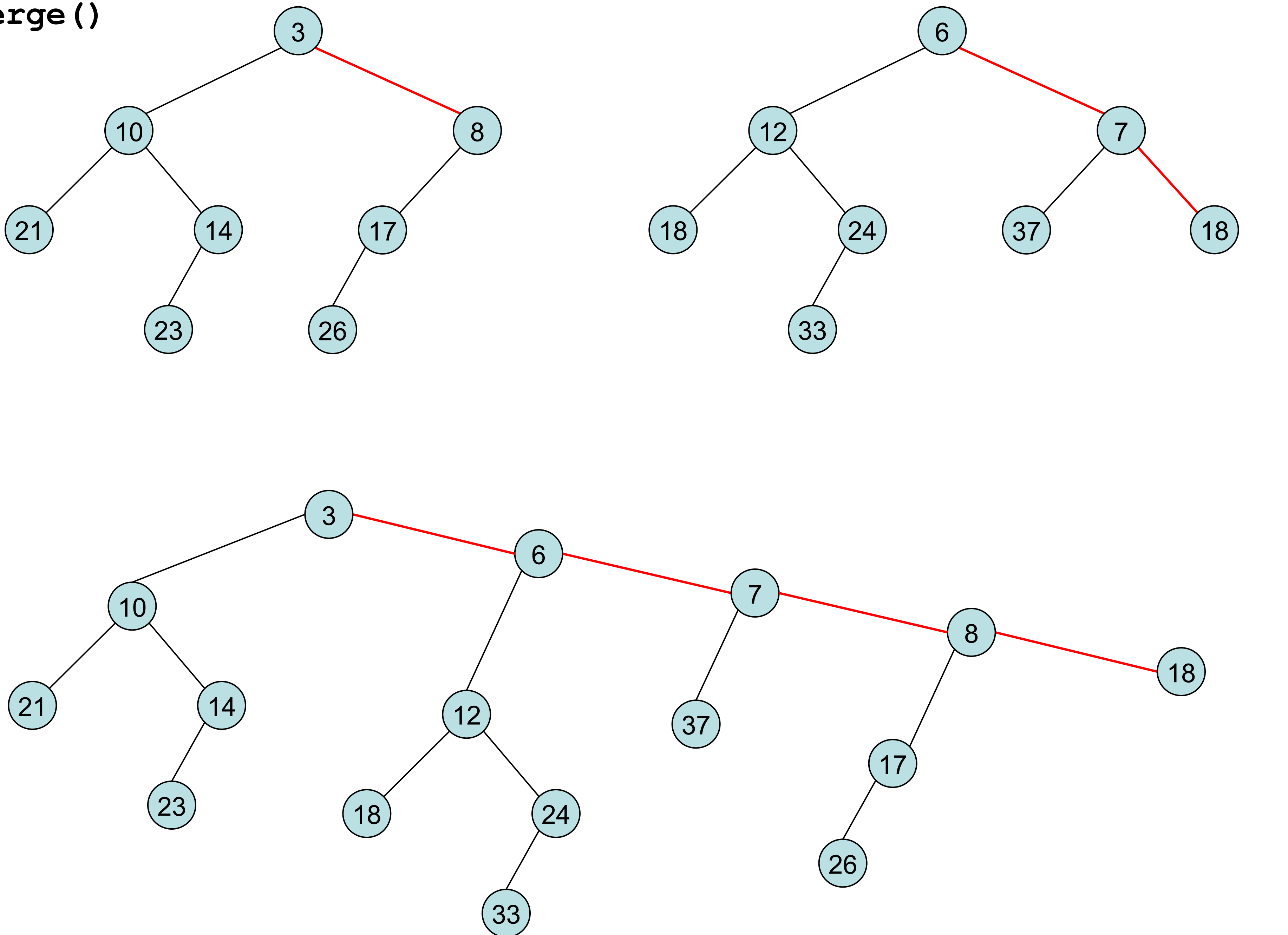
Leftist heaps

`merge()`



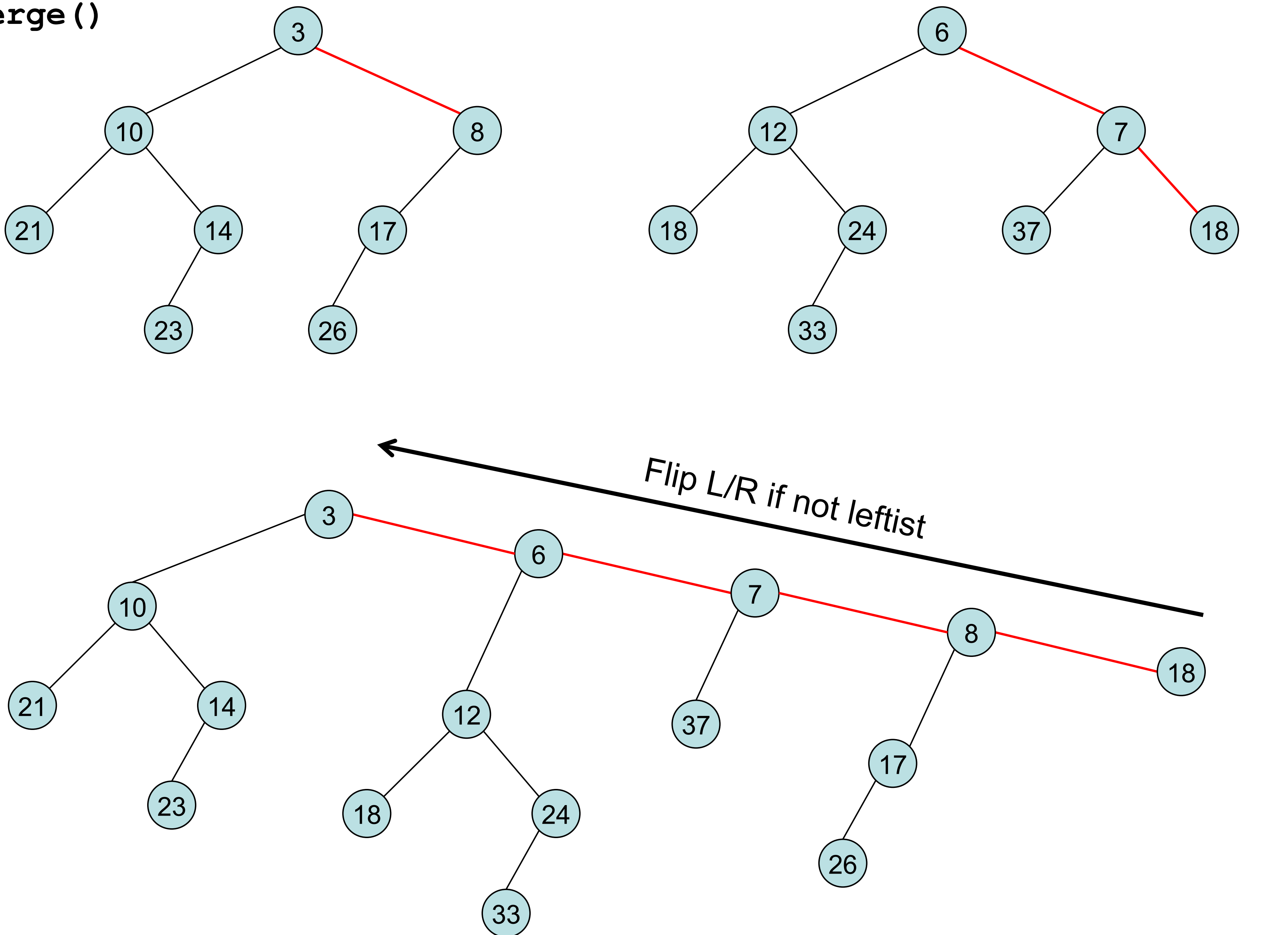
Leftist heaps

`merge()`



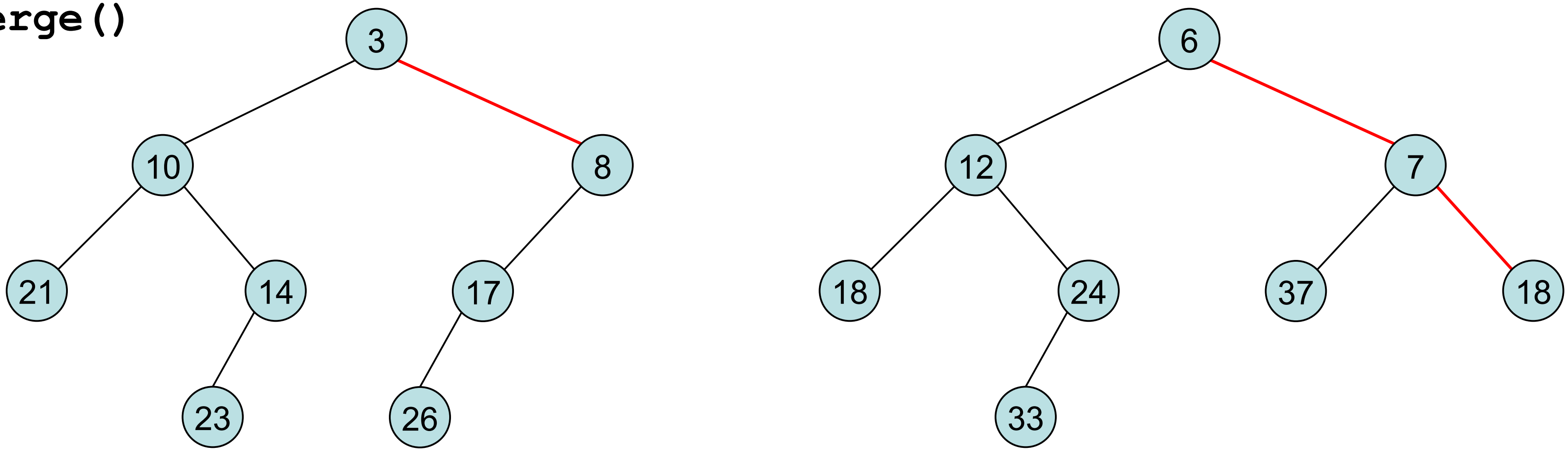
Leftist heaps

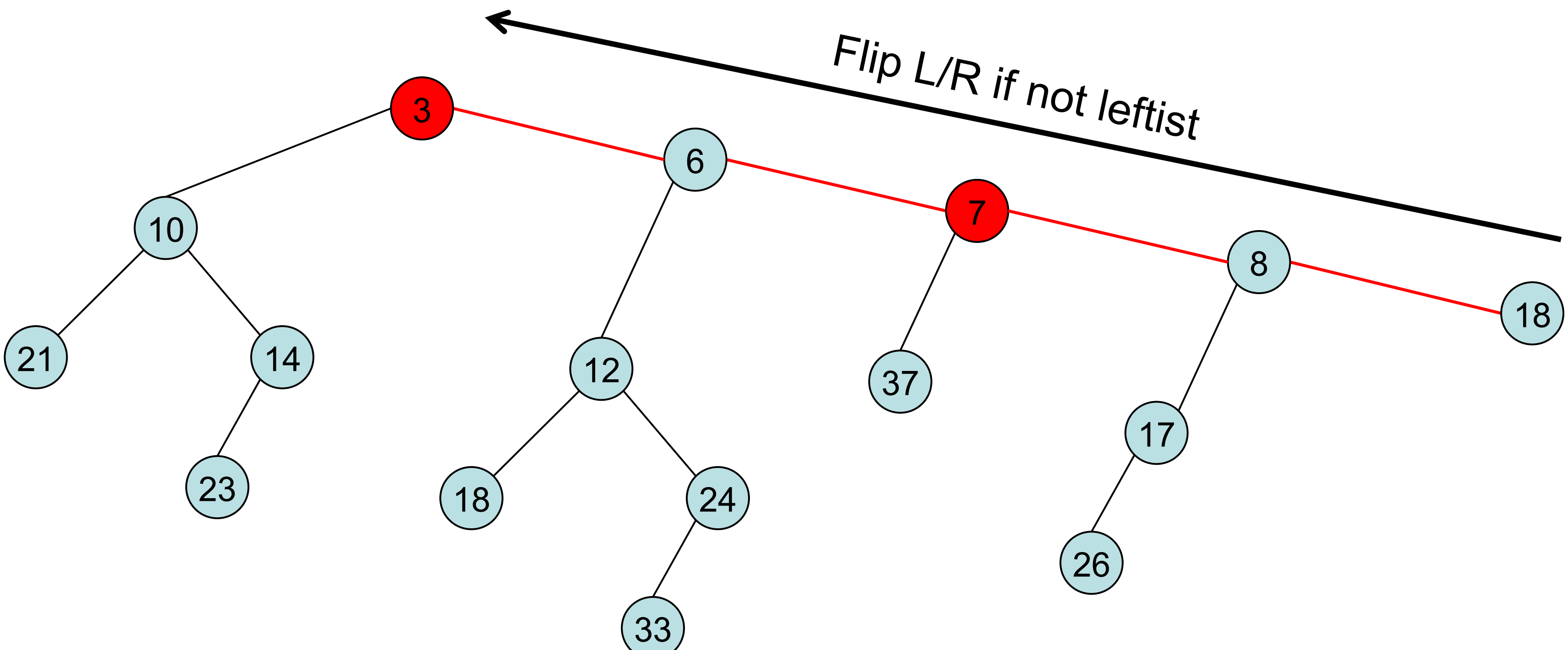
`merge()`



Leftist heaps

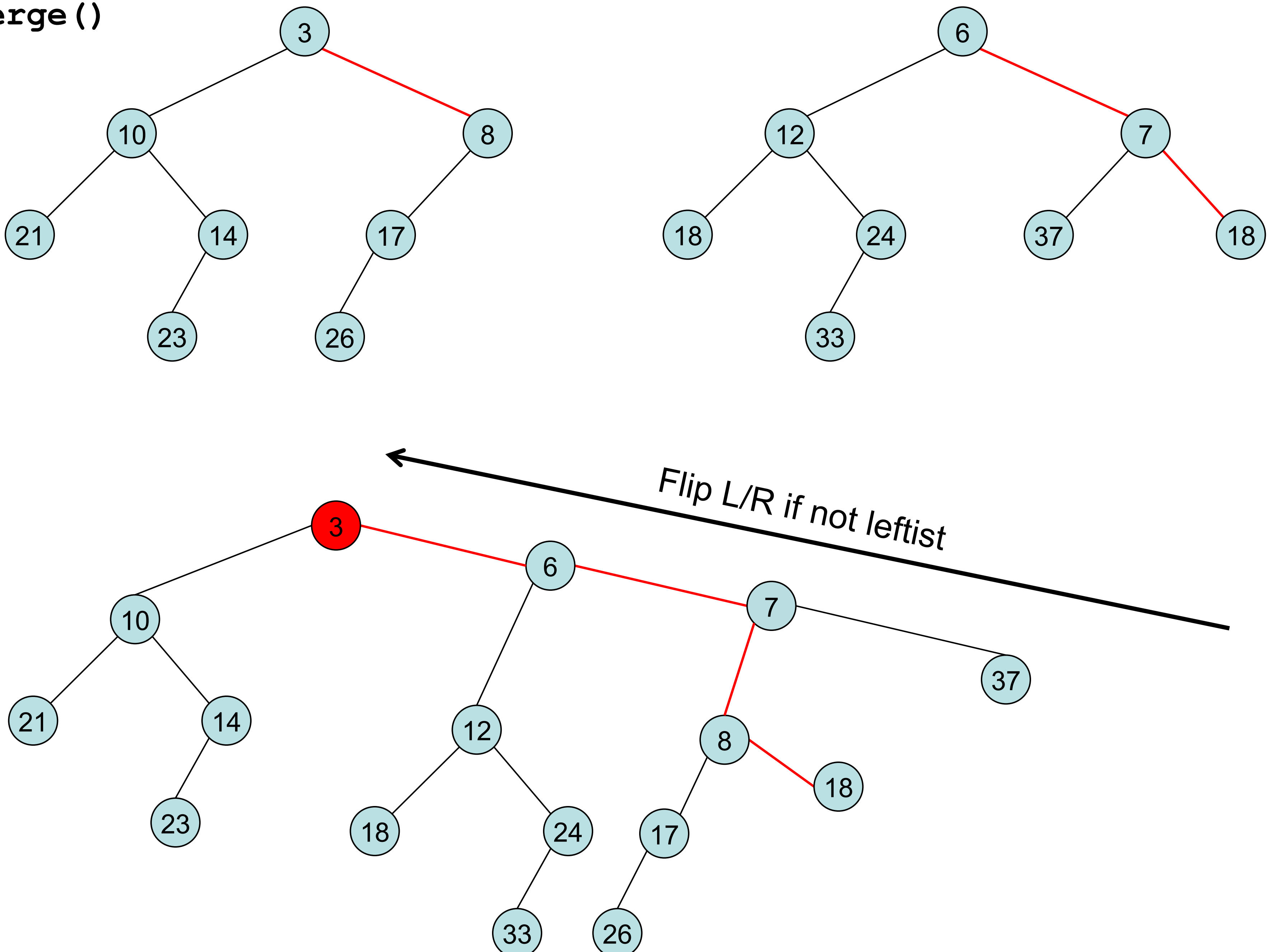
merge()





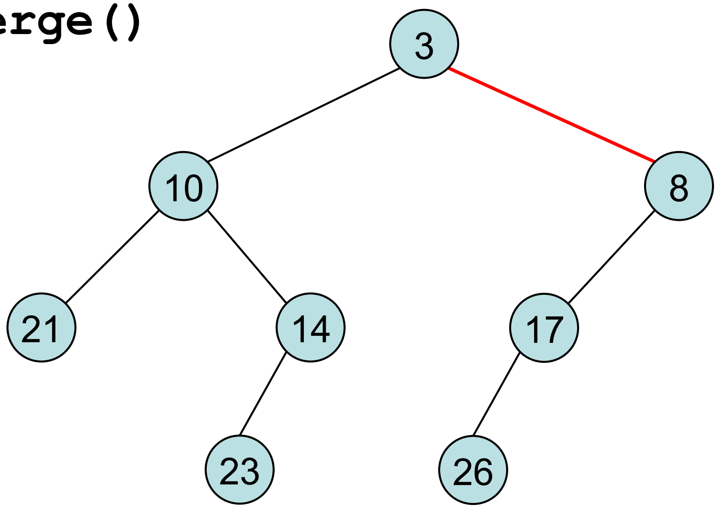
Leftist heaps

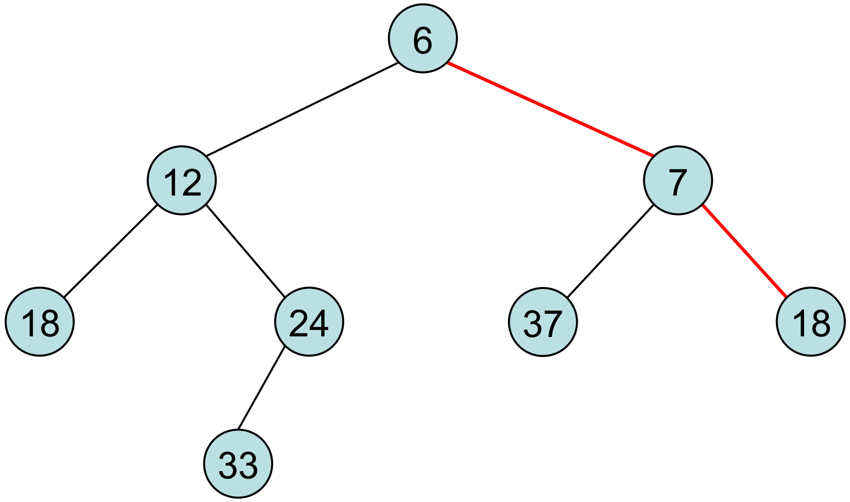
`merge()`

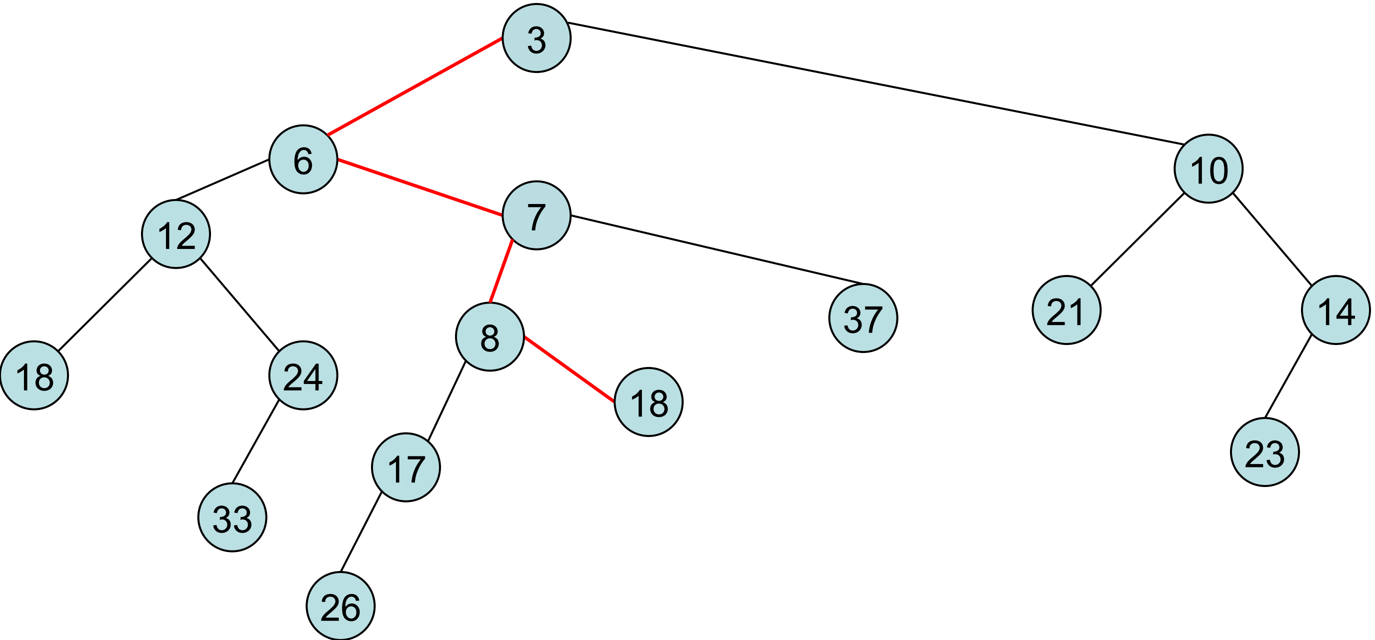


Leftist heaps

`merge()`



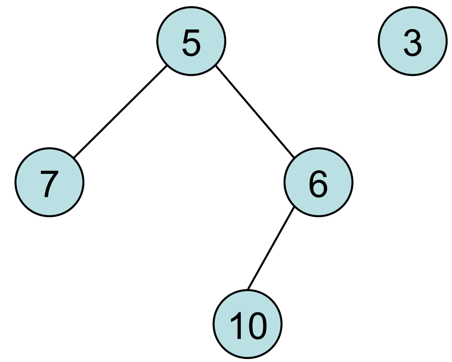




Leftist heaps

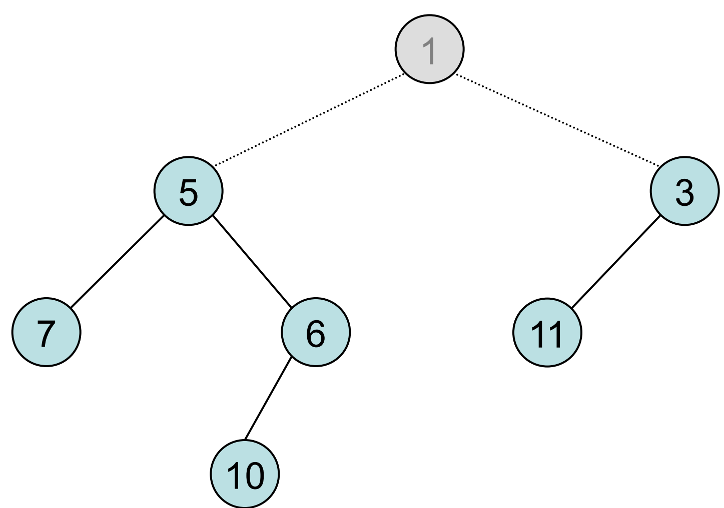
`insert(3)`

`merge()`



`deleteMin()`

`merge()`



Leftist heaps

	worst case
`merge()`	$O(\log N)$
`insert()`	$O(\log N)$
`deleteMin()`	$O(\log N)$
`buildHeap()`	$O(N)$

(N = number of elements)

In a leftist heap with N nodes, the right path is at most $\lfloor \log (N+1) \rfloor$ long.

Binomial heaps

Leftist heaps:

`merge()`, **`insert()`** and **`deleteMin()`** in *O*(log *N*) time w.c.

Binary heaps:

`insert()` in *O*(1) time on average.

Binomial heaps

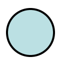
`merge()`, **`insert()`** og **`deleteMin()`** in *O*(log *N*) time w.c.

`insert()` *O*(1) time on average

Binomial heaps are collections of trees (sometimes called a forest), each tree a heap.

Binomial heaps

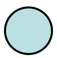
Binomial trees



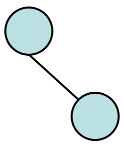
B_0

Binomial heaps

Binomial trees



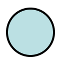
B_0



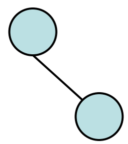
B_1

Binomial heaps

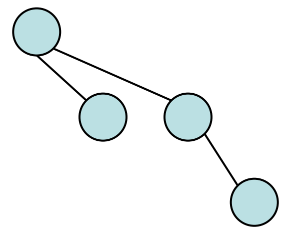
Binomial trees



B_0



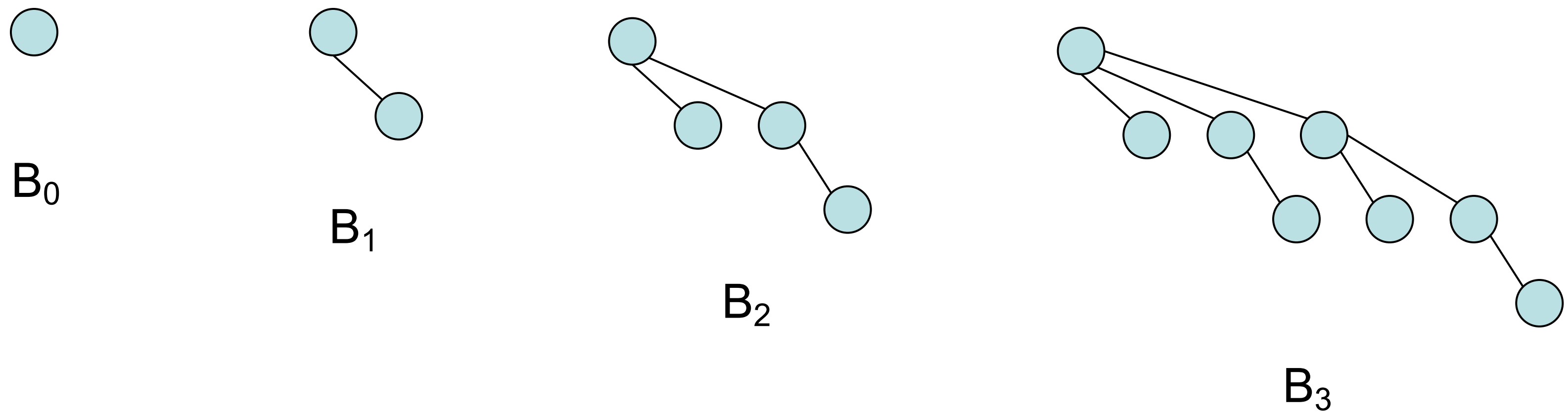
B_1



B_2

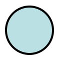
Binomial heaps

Binomial trees

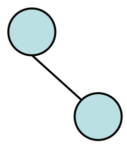


Binomial heaps

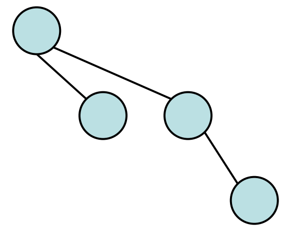
Binomial trees



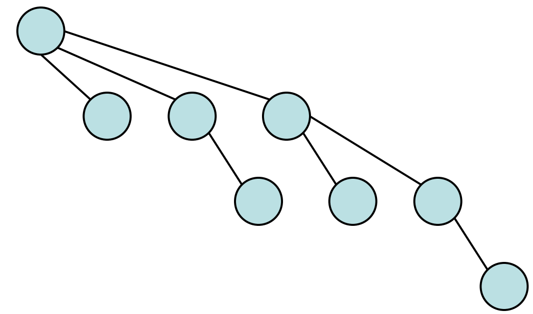
B_0



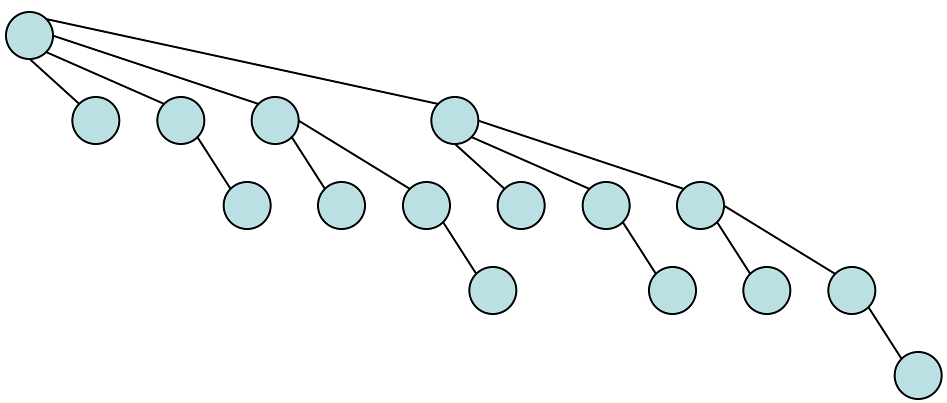
B_1



B_2



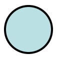
B_3



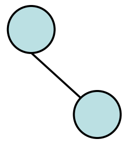
B_4

Binomial heaps

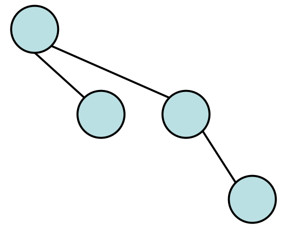
Binomial trees



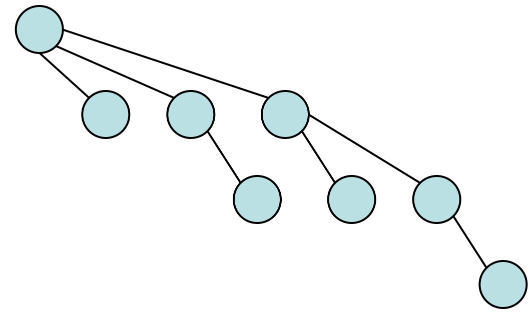
B_0



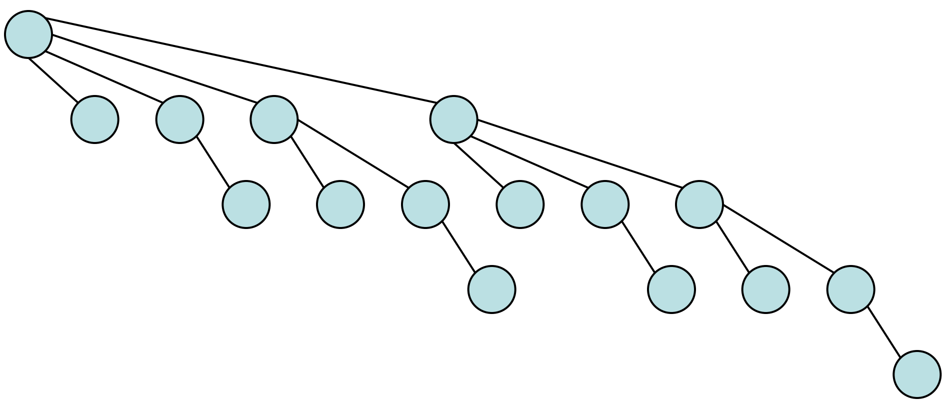
B_1



B_2



B_3



B_4

$B_i = 2 \times B_{i-1}$, root of one tree connected as a child of the root of the other tree.

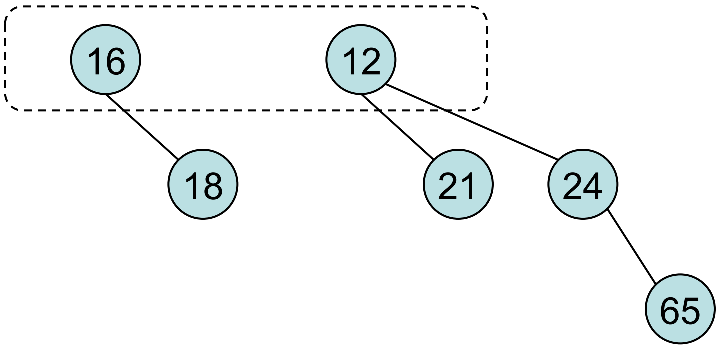
A tree of height k has:

2^k nodes in total,

$\binom{k}{d}$ nodes on level d.

Binomial heaps

Binomial heap

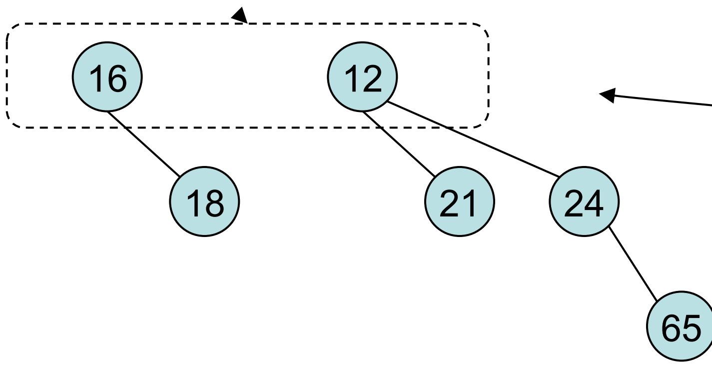


Maximum one tree of each size:

6 elements: 6 binary = 011 (0+2+4) $\cancel{B_0}$ B_1 B_2

Binomial heaps

Binomial heap



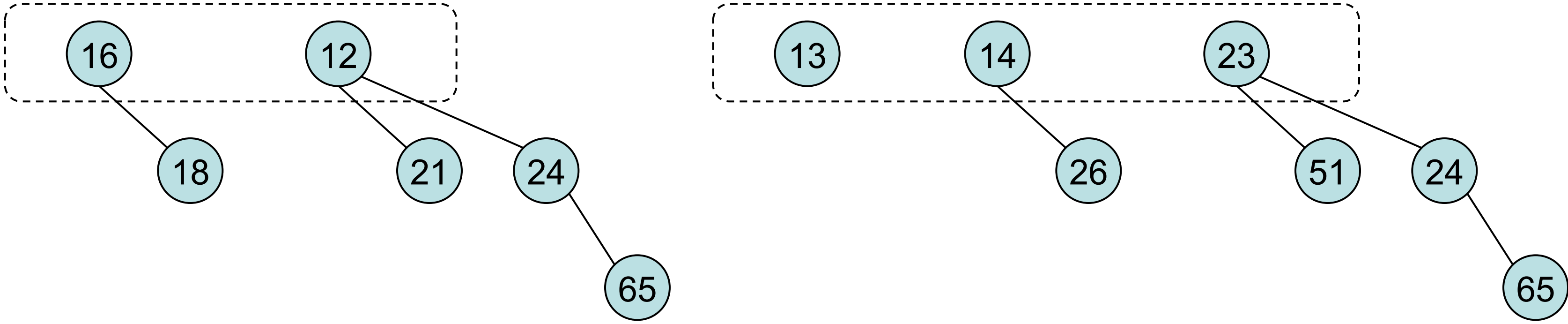
Maximum one tree of each size:

6 elements: 6 binary = 011 (0+2+4) $\cancel{B_0}$ B_1 B_2

The length of the root list in a heap of N elements is $O(\log N)$.
(Doubly linked, circular list.)

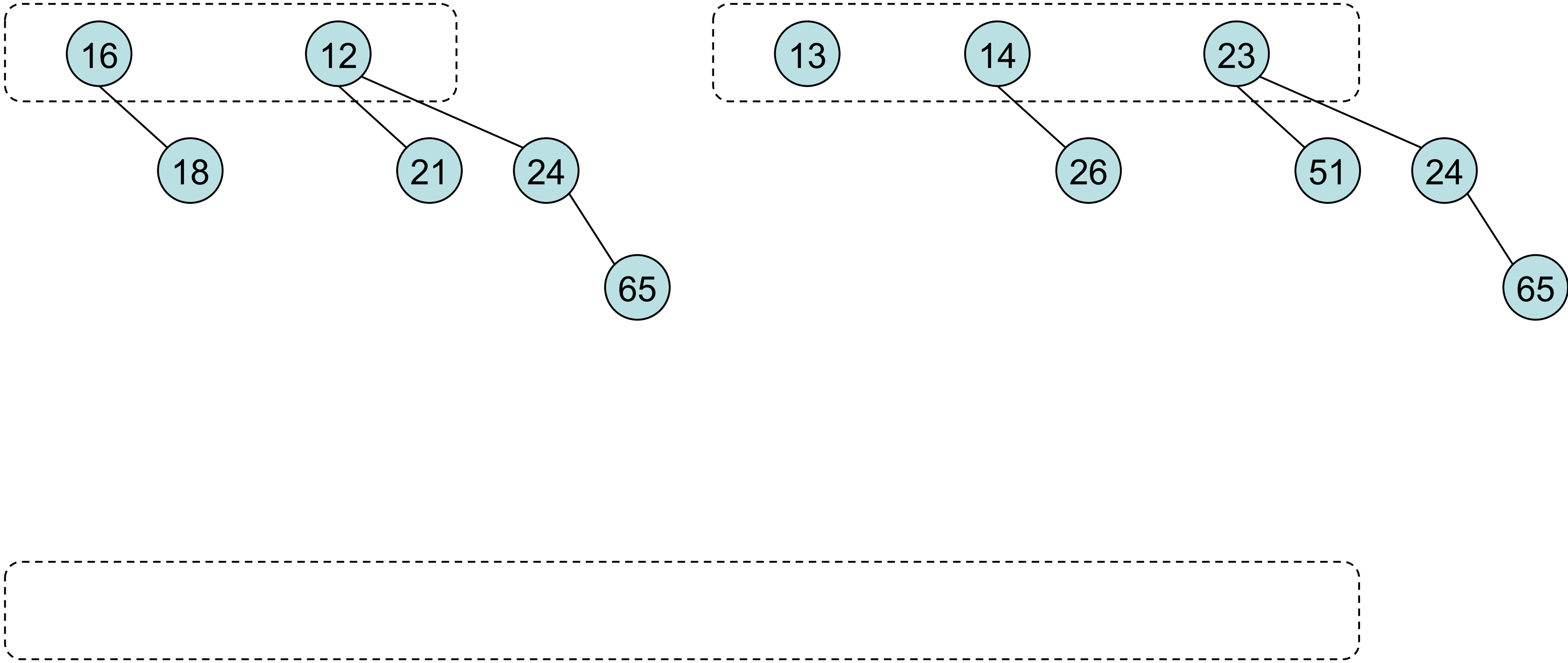
Binomial heaps

`merge()`



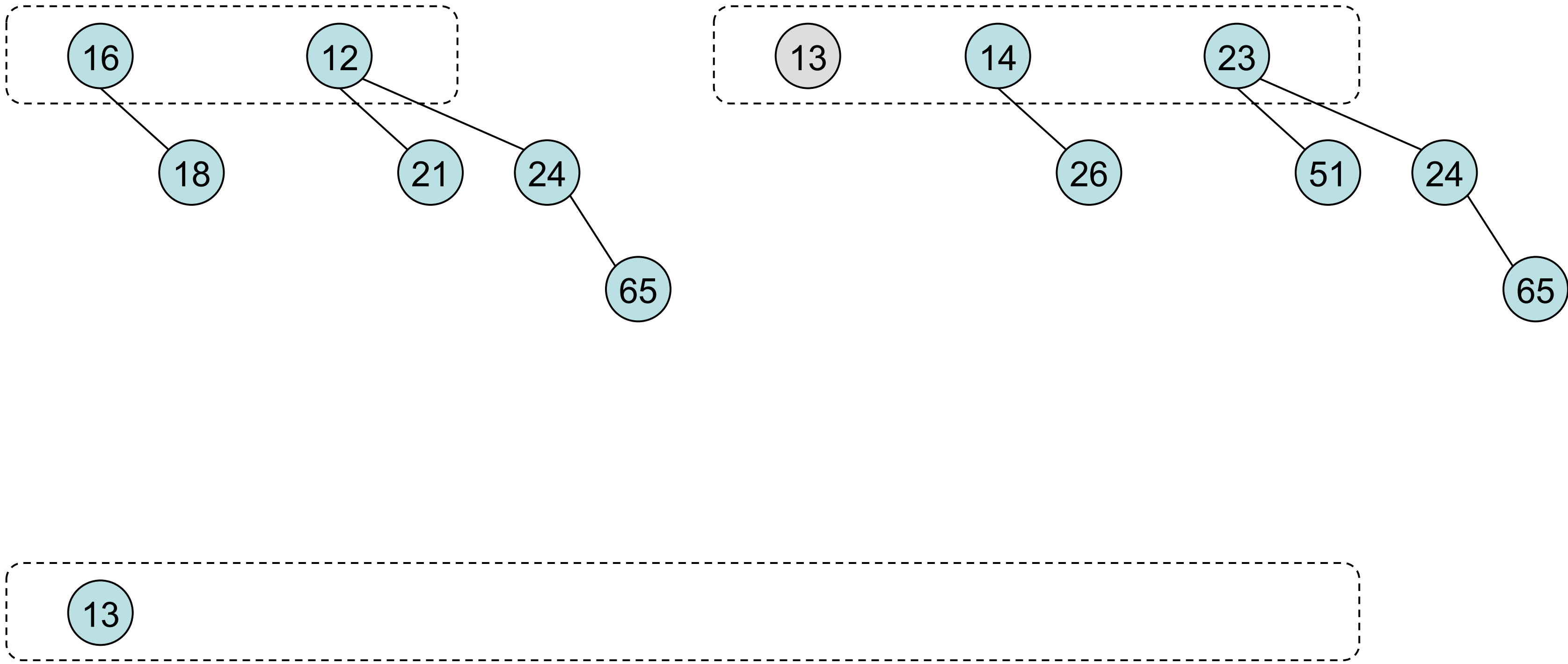
Binomial heaps

merge()



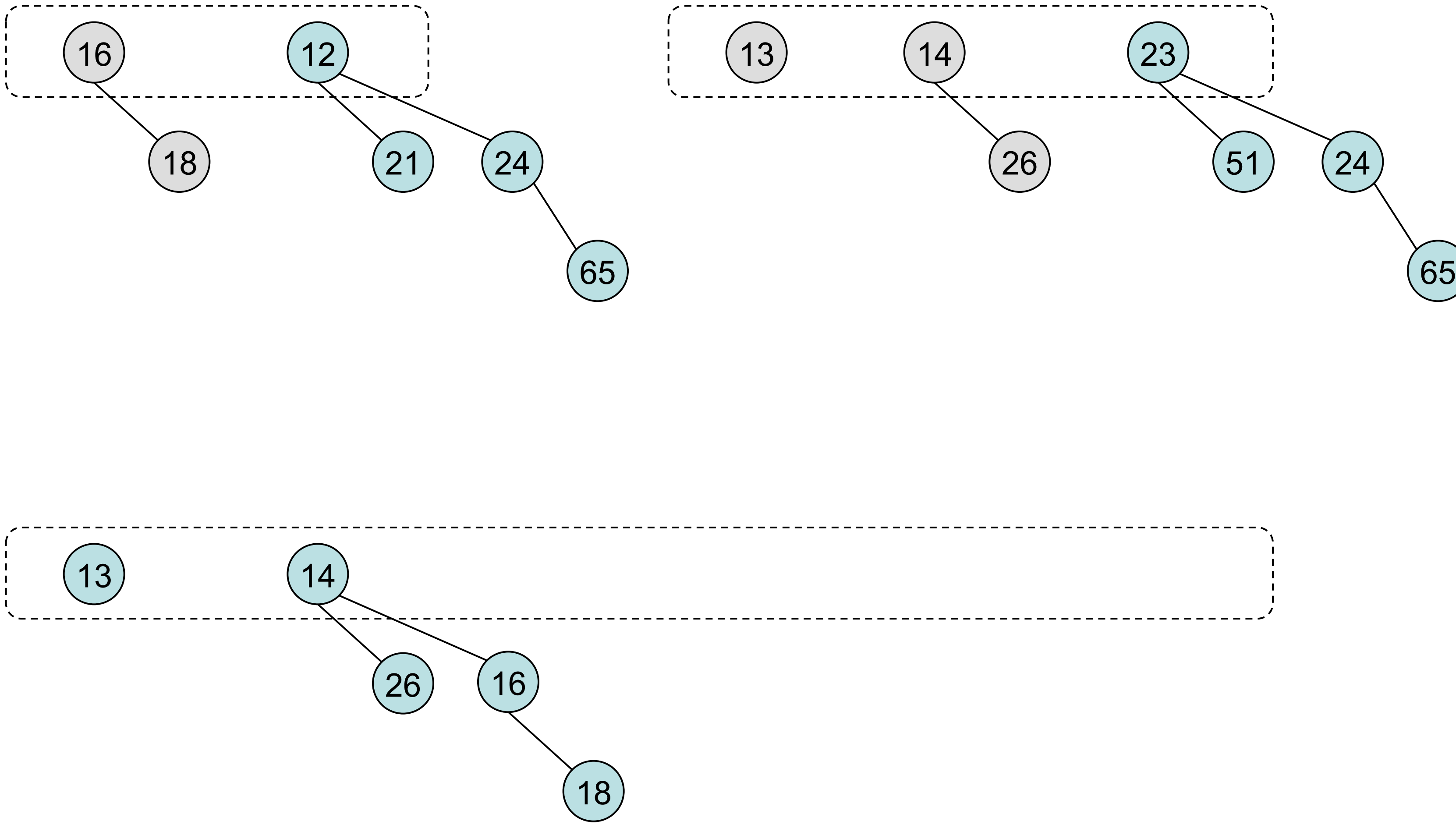
Binomial heaps

merge()



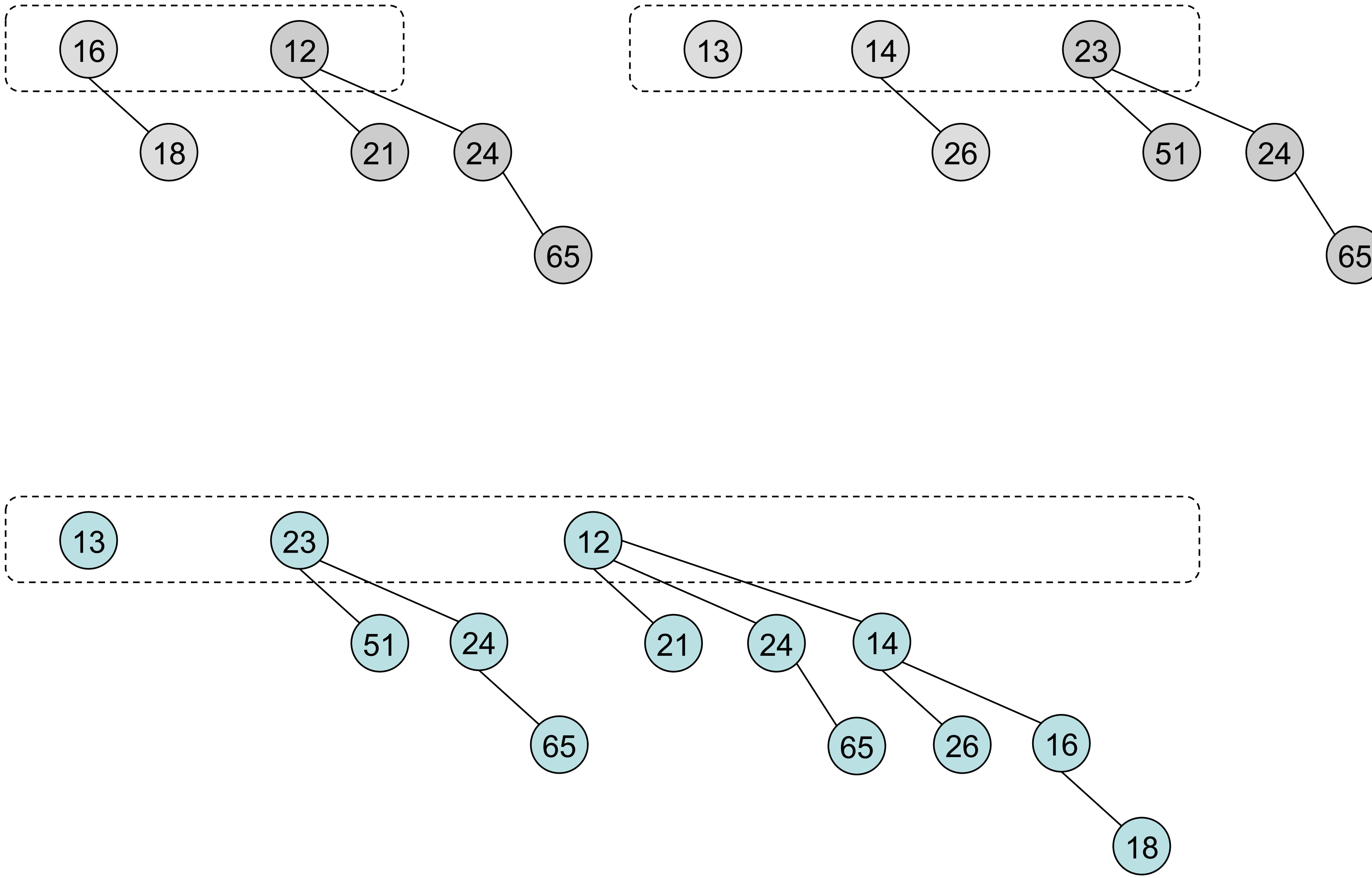
Binomial heaps

`merge()`



Binomial heaps

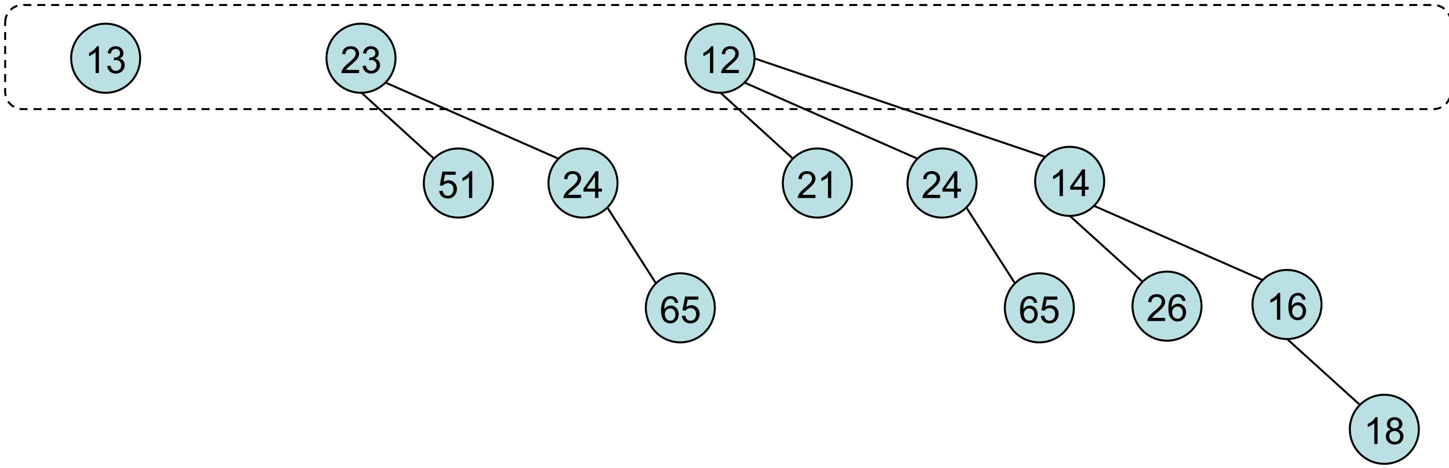
`merge()`



The trees (the root list) is kept sorted on height.

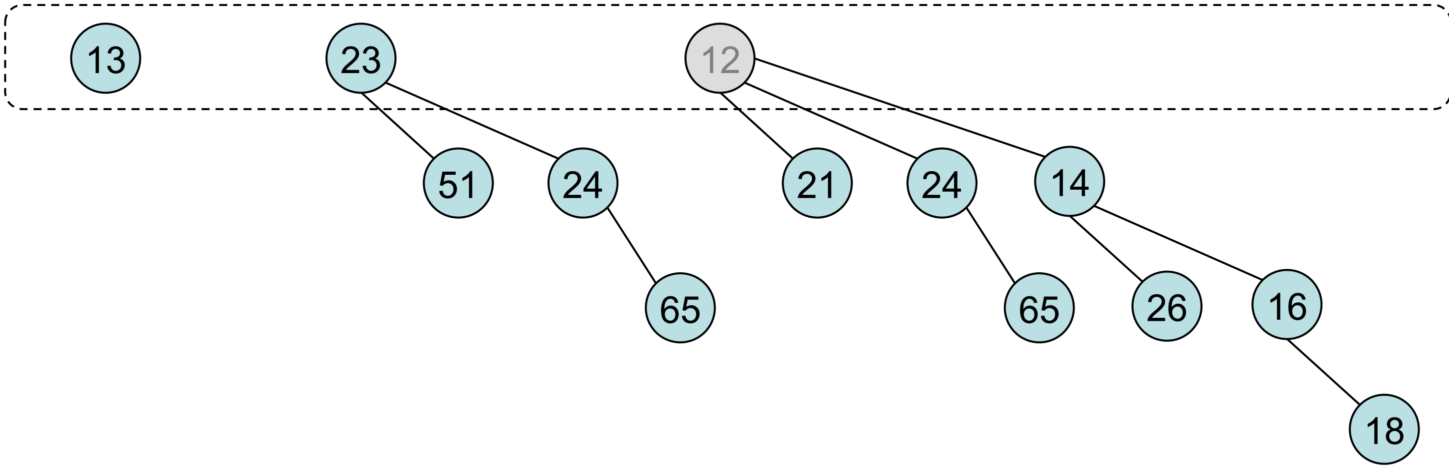
Binomial heaps

`deleteMin()`

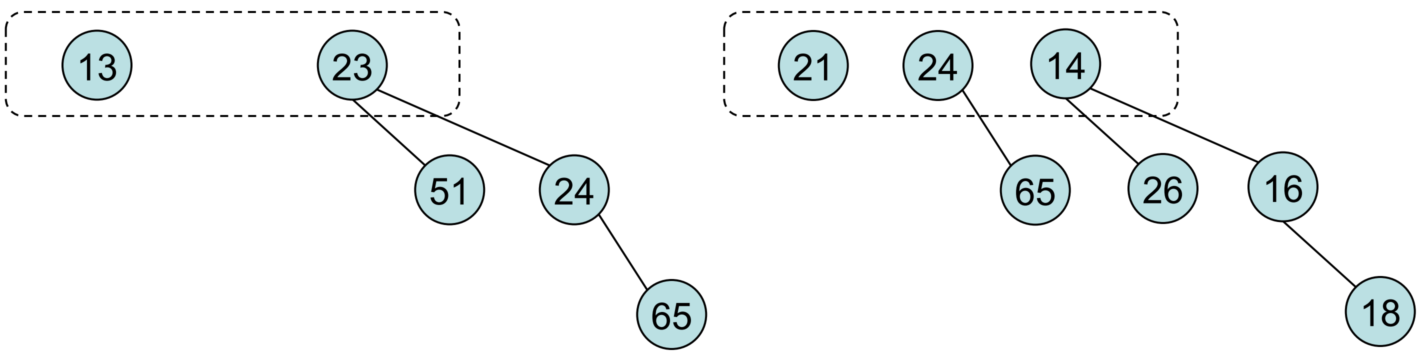


Binomial heaps

`deleteMin()`



`merge()`



Binomial heaps

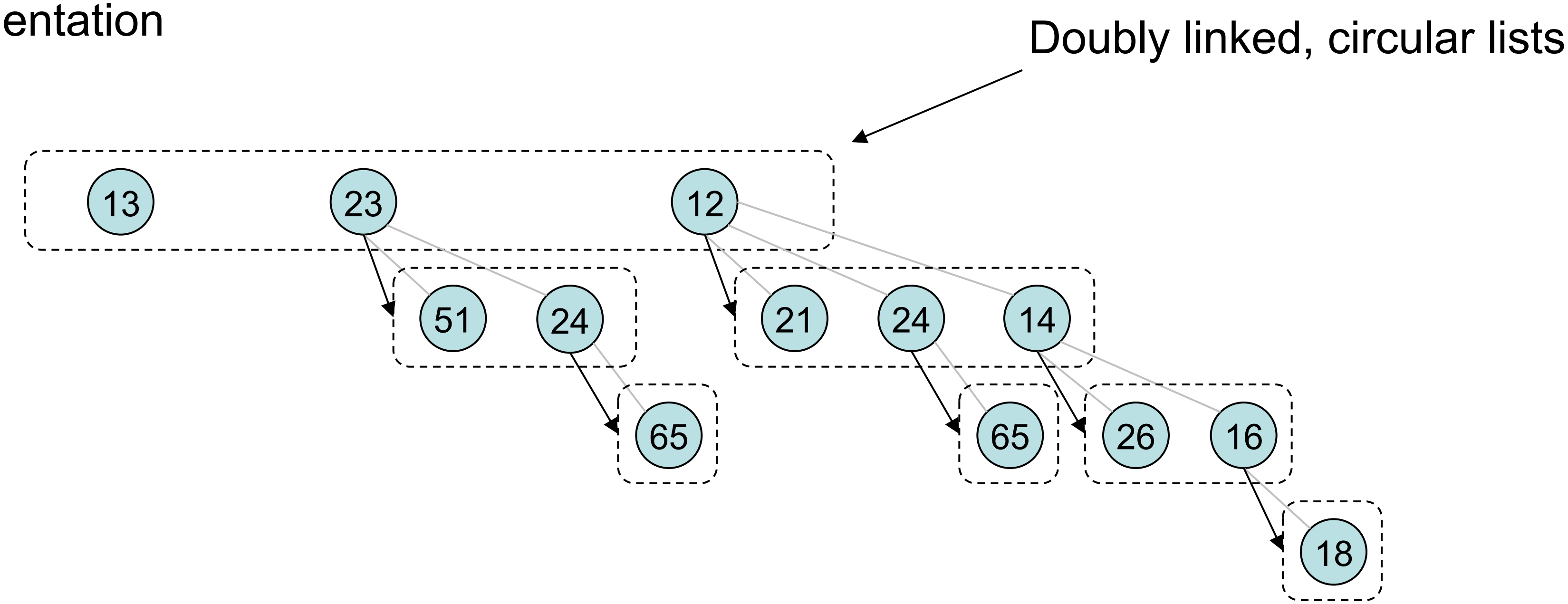
	worst case	average case
`merge()`	$O(\log N)$	$O(\log N)$
`insert()`	$O(\log N)$	$O(1)$
`deleteMin()`	$O(\log N)$	$O(\log N)$
`buildHeap()`	$O(N)$	$O(N)$

(Run N **`insert()`** on an initially empty heap.)

(N = number of elements)

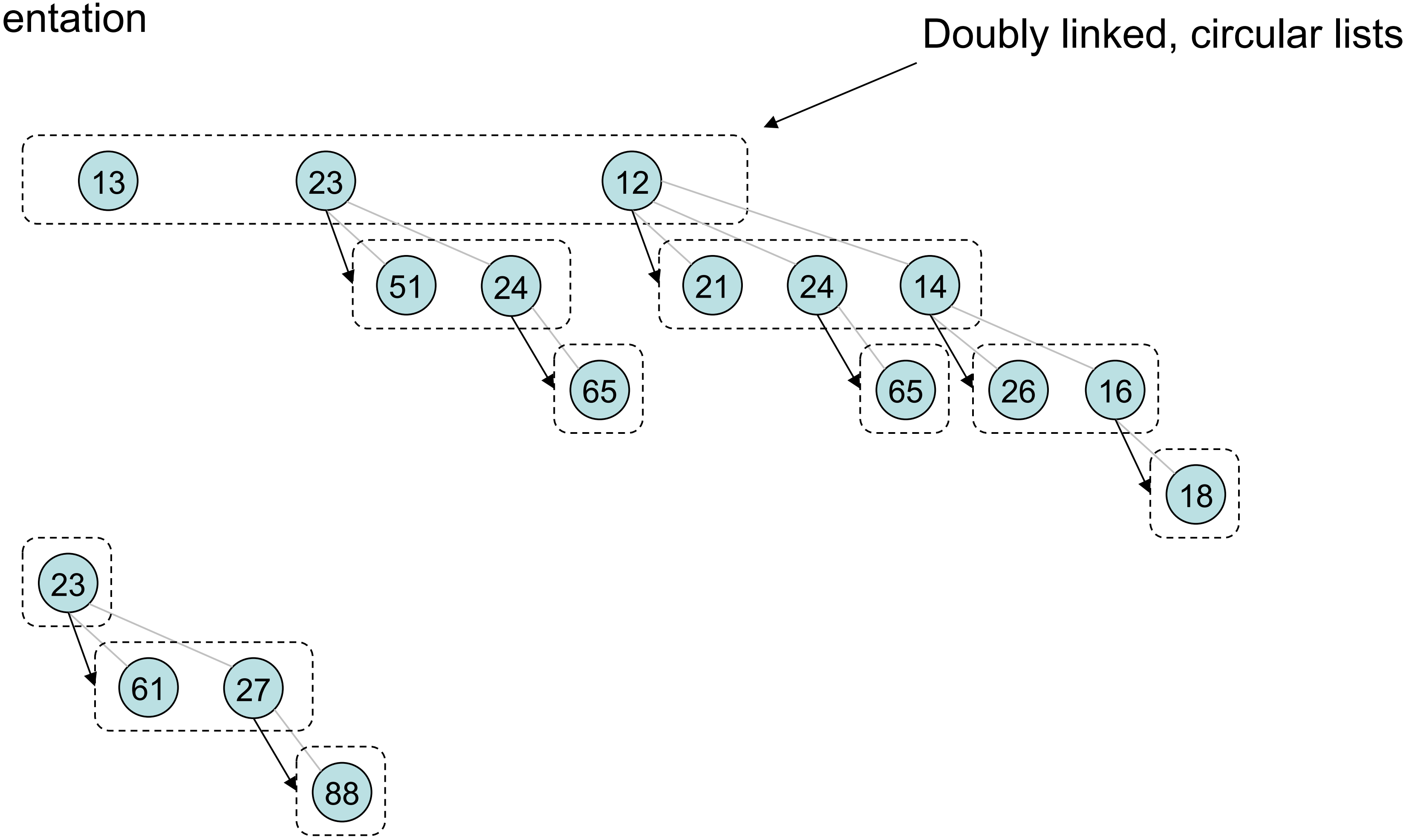
Binomial heaps

Implementation



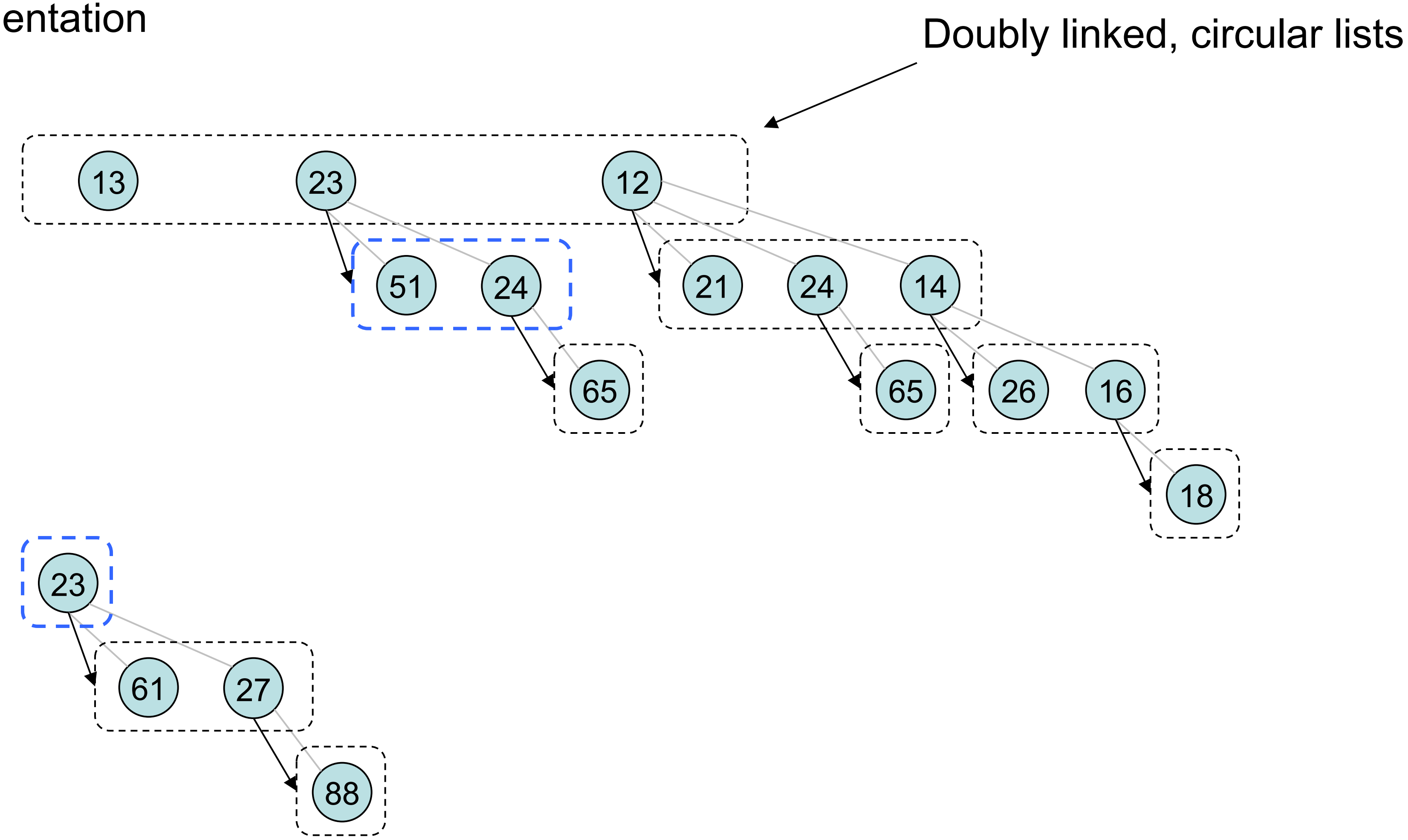
Binomial heaps

Implementation



Binomial heaps

Implementation



Fibonacci heaps

Very elegant, and in theory efficient, way to implement heaps: Most operations have $O(1)$ amortized running time. (Fredman & Tarjan ’87)

`insert()`, **`decreaseKey()`** and **`merge()`**	$O(1)$ amortized time
`deleteMin()`	$O(\log N)$ amortized time

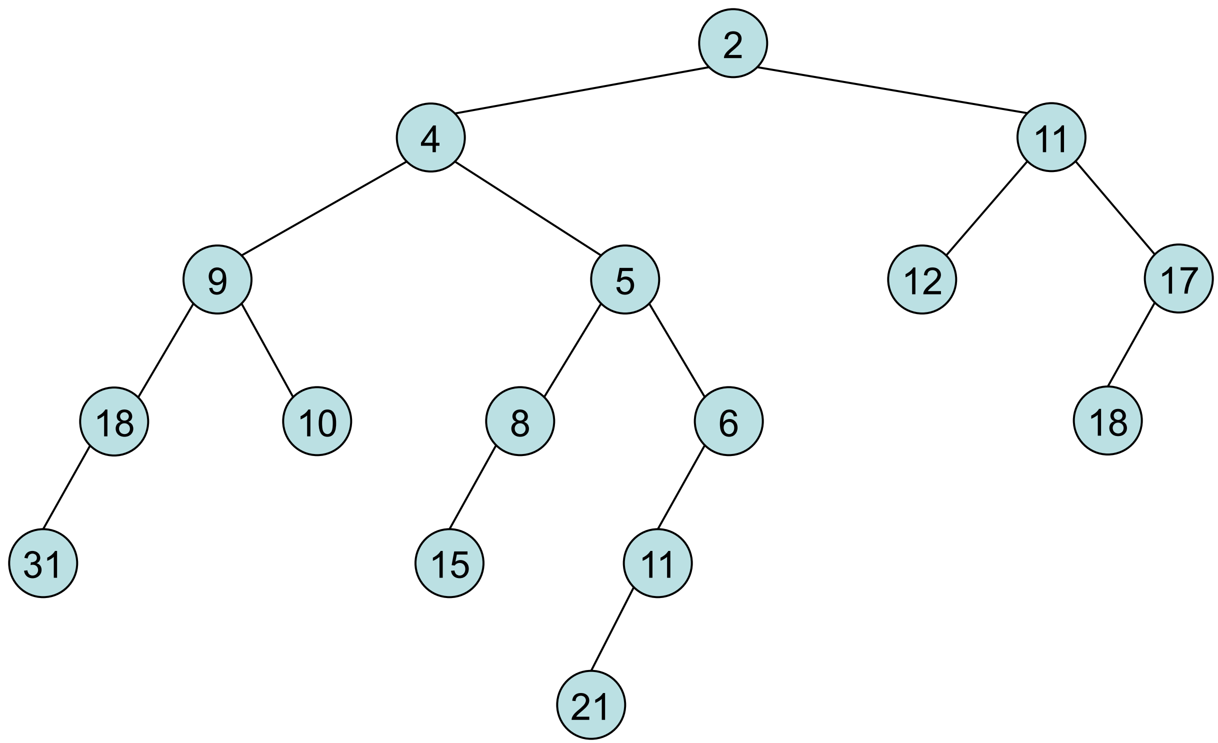
Combines elements from leftist heaps and binomial heaps.

A bit complicated to implement, and certain hidden constants are a bit high.

Best suited when there are few **`deleteMin()`** compared to the other operations. The data structure was developed for a shortest path algorithm (with many **`decreaseKey()`** operations), also used in spanning tree algorithms.

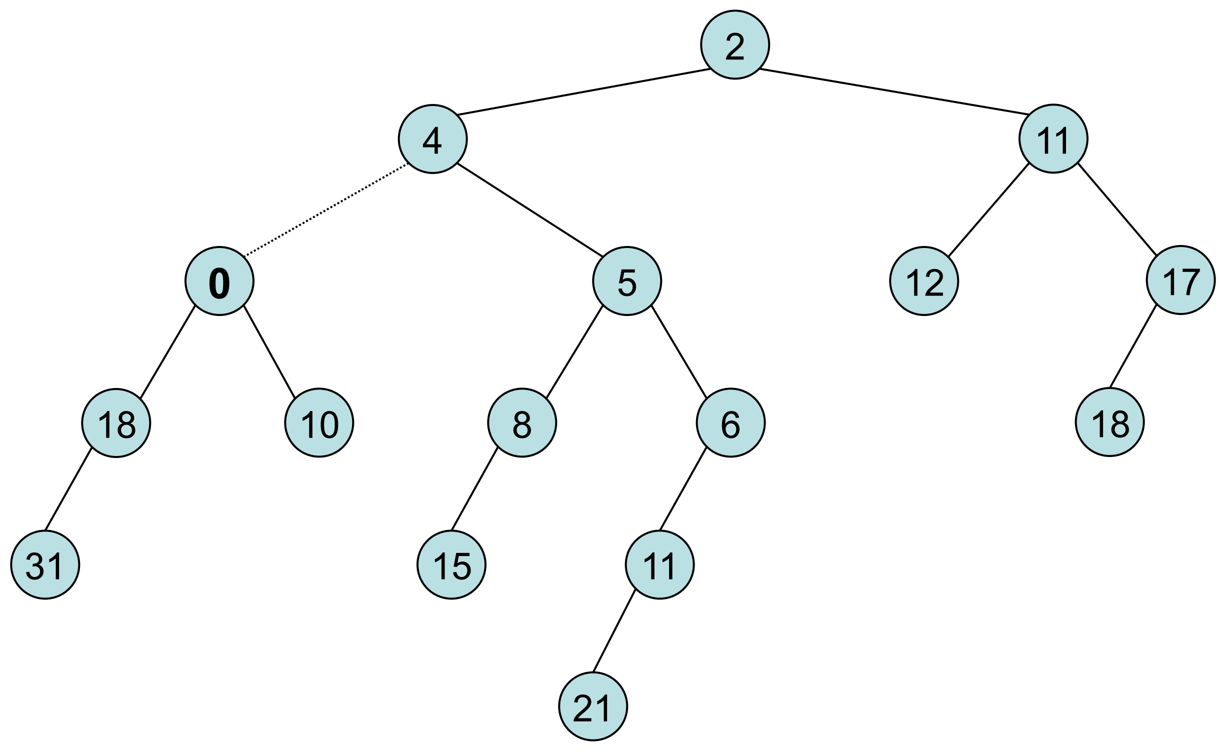
Fibonacci heaps

We include a smart **`decreaseKey()`** method from leftist heaps.



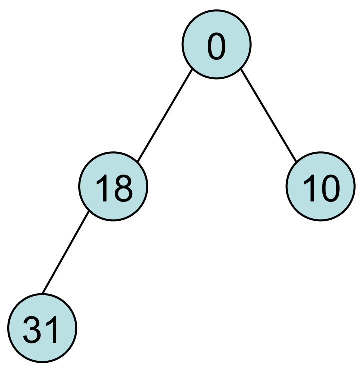
Fibonacci heaps

We include a smart **`decreaseKey()`** method from leftist heaps.

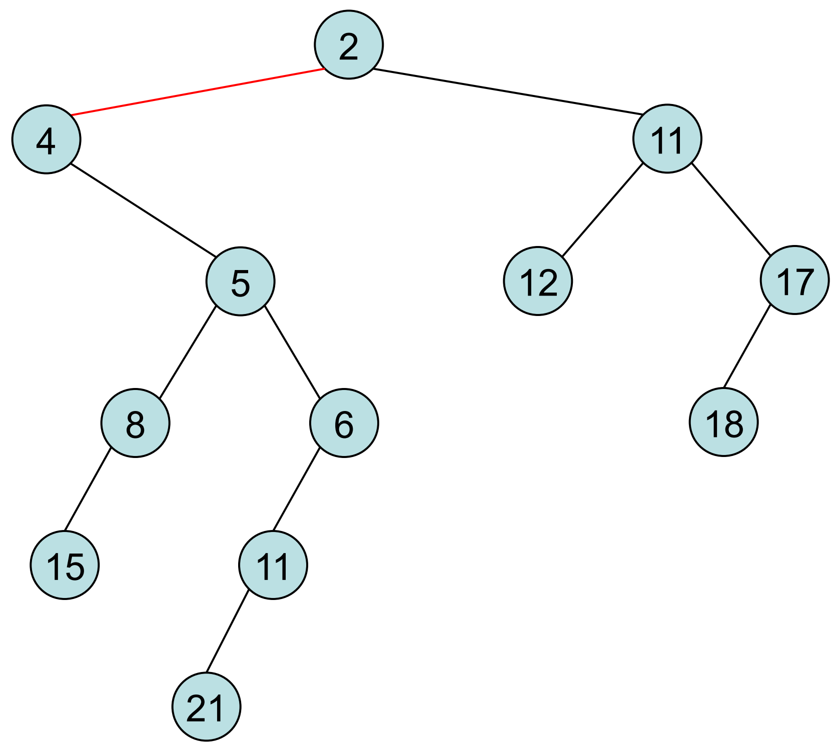


Fibonacci heaps

We include a smart **`decreaseKey()`** method from leftist heaps.



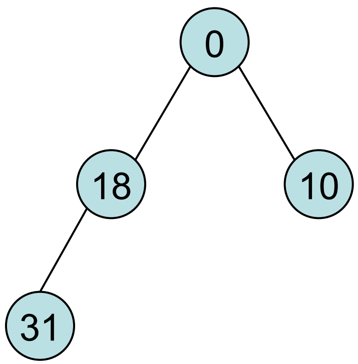
Leftist



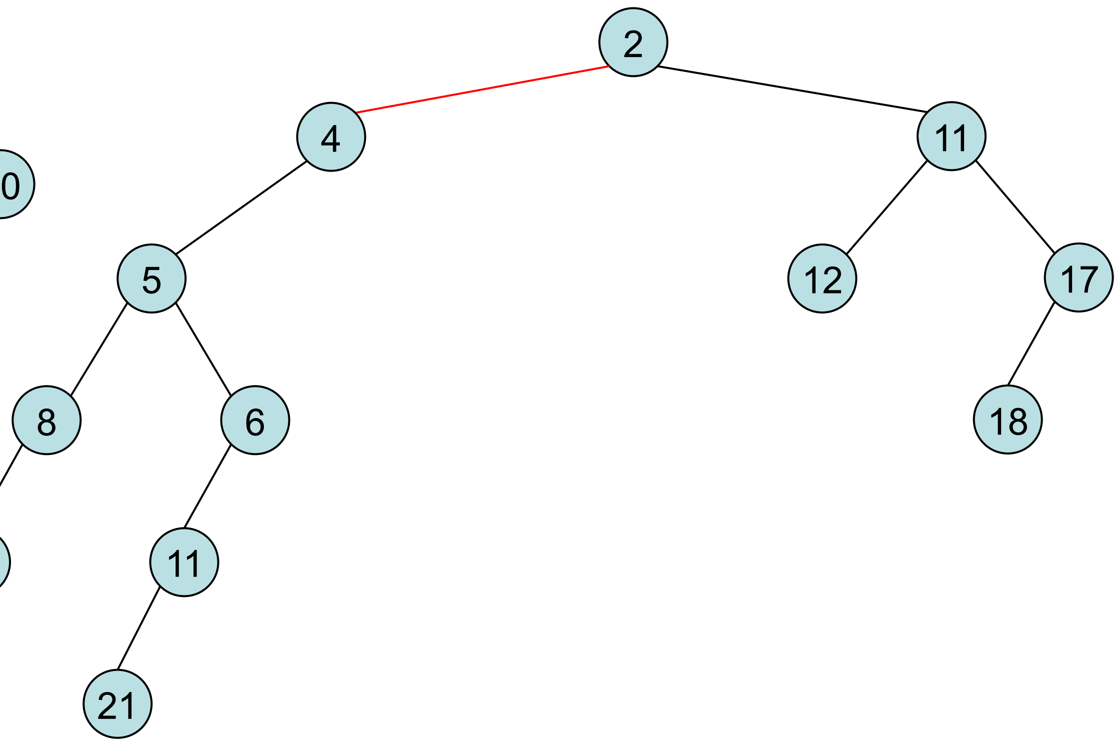
Ikke leftist

Fibonacci heaps

We include a smart **`decreaseKey()`** method from leftist heaps.



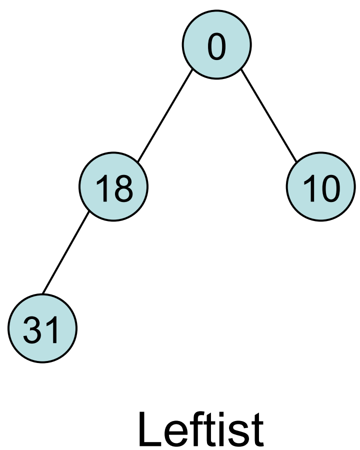
Leftist

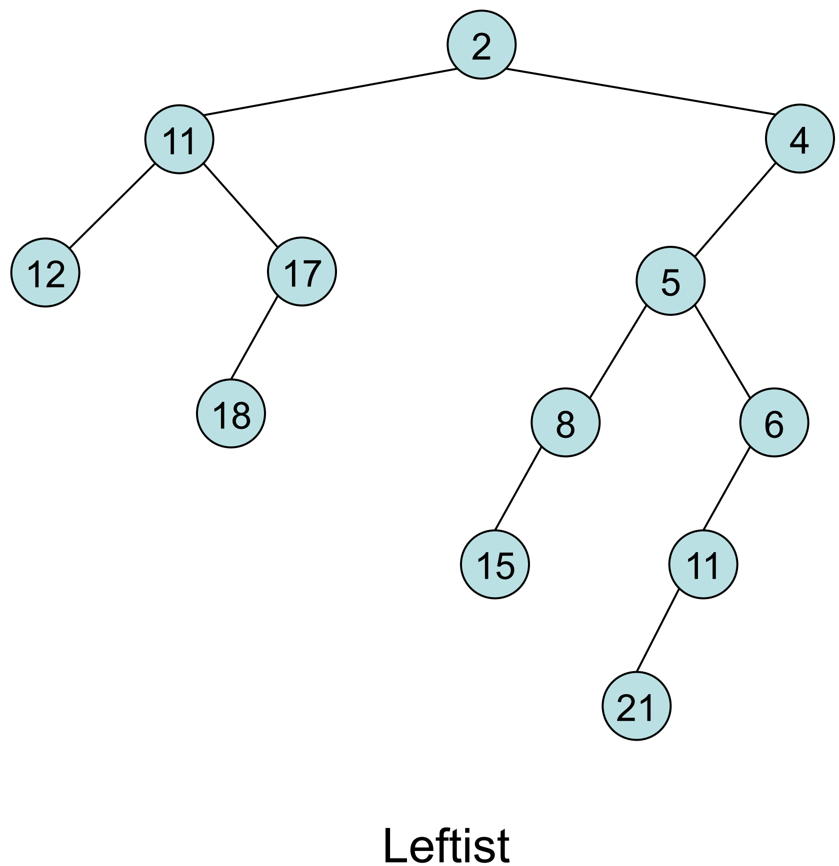


Ikke leftist

Fibonacci heaps

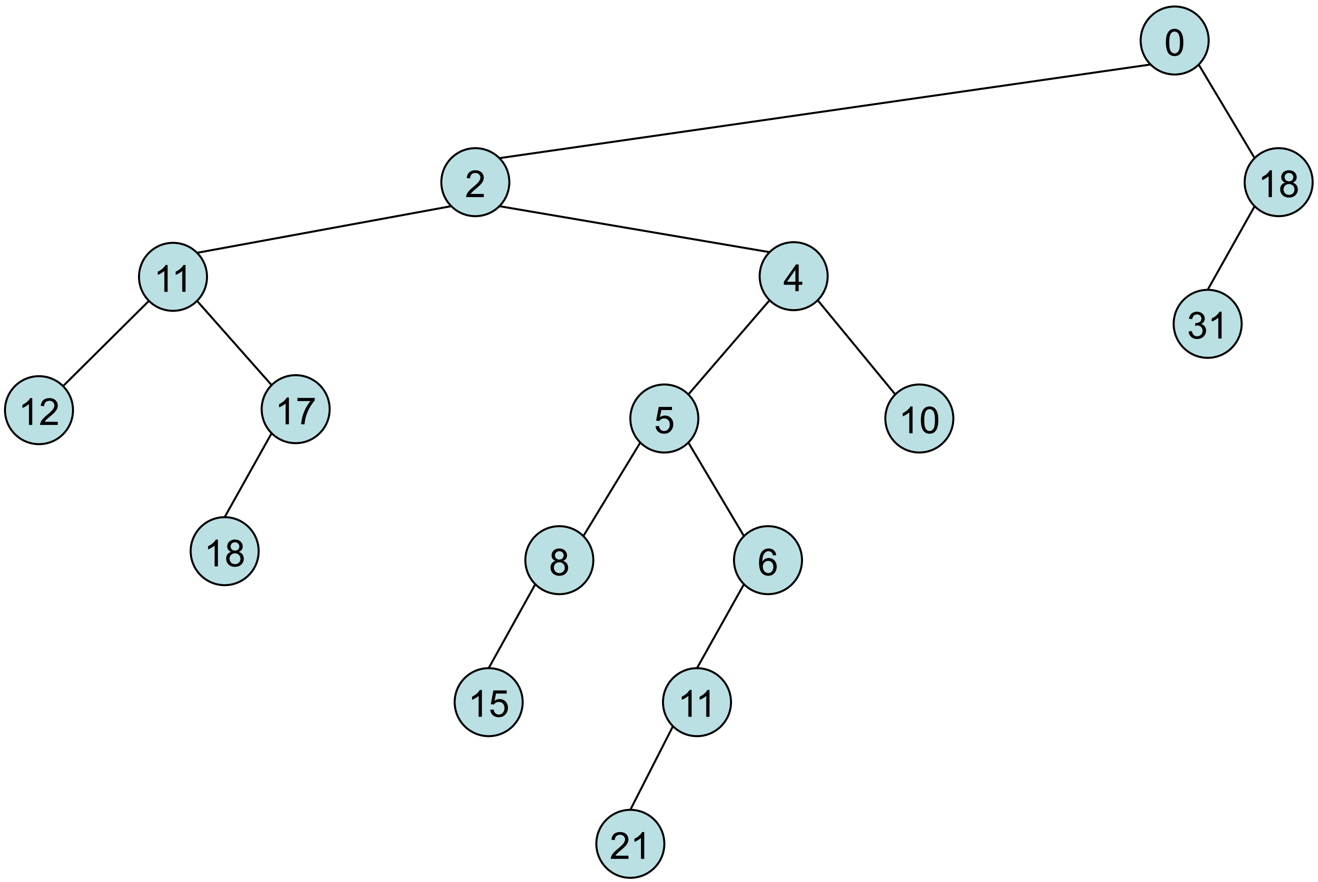
We include a smart `decreaseKey()` method from leftist heaps.





Fibonacci heaps

We include a smart **`decreaseKey()`** method from leftist heaps.

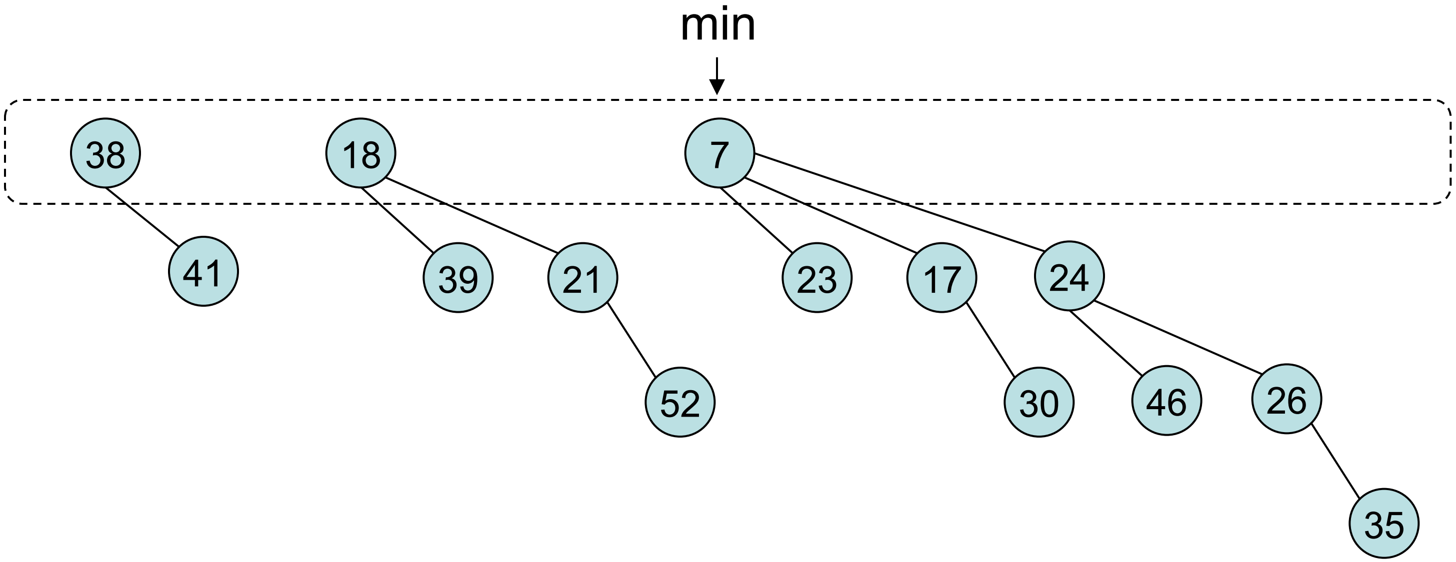


Fibonacci heaps

We include a smart **`decreaseKey()`** method from leftist heaps.

The method must be modified a bit, as we wish to use trees that are binomial trees, or partial binomial trees.

- Nodes are marked the first time child is removed.
- The second time a node gets a child removed, it is cut off, and becomes the root of a separate tree

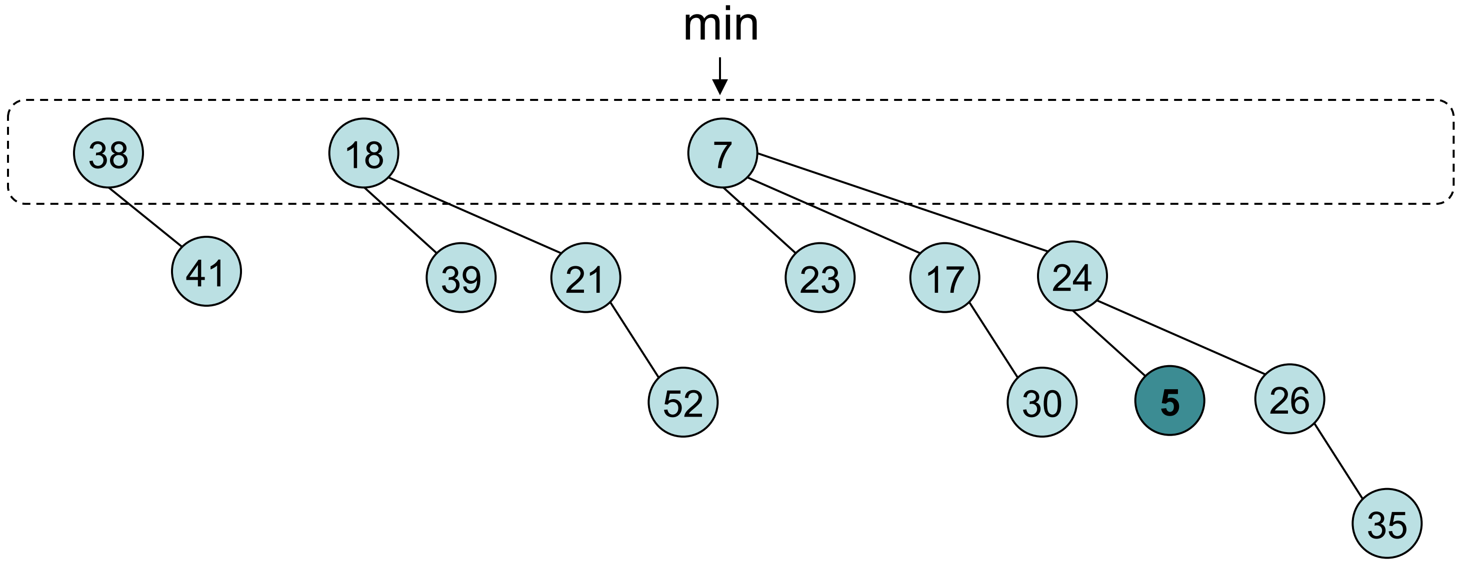


Fibonacci heaps

We include a smart **`decreaseKey()`** method from leftist heaps.

The method must be modified a bit, as we wish to use trees that are binomial trees, or partial binomial trees.

- Nodes are marked the first time child is removed.
- The second time a node gets a child removed, it is cut off, and becomes the root of a separate tree

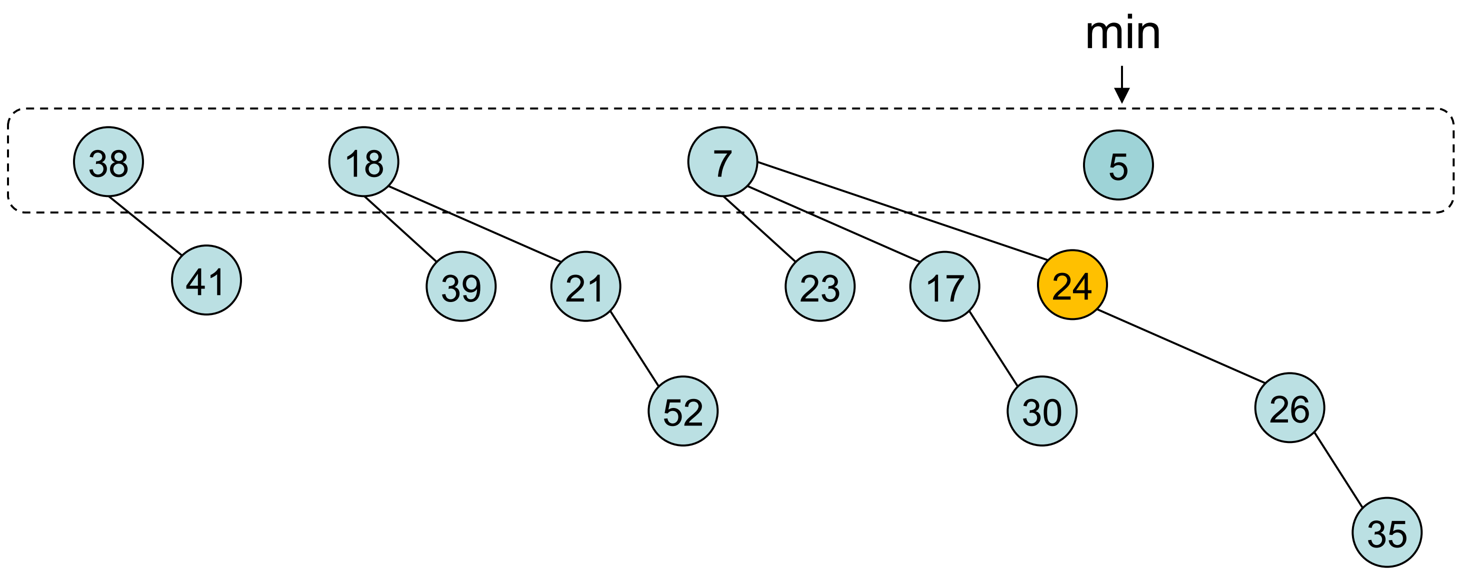


Fibonacci heaps

We include a smart **`decreaseKey()`** method from leftist heaps.

The method must be modified a bit, as we wish to use trees that are binomial trees, or partial binomial trees.

- Nodes are marked the first time child is removed.
- The second time a node gets a child removed, it is cut off, and becomes the root of a separate tree

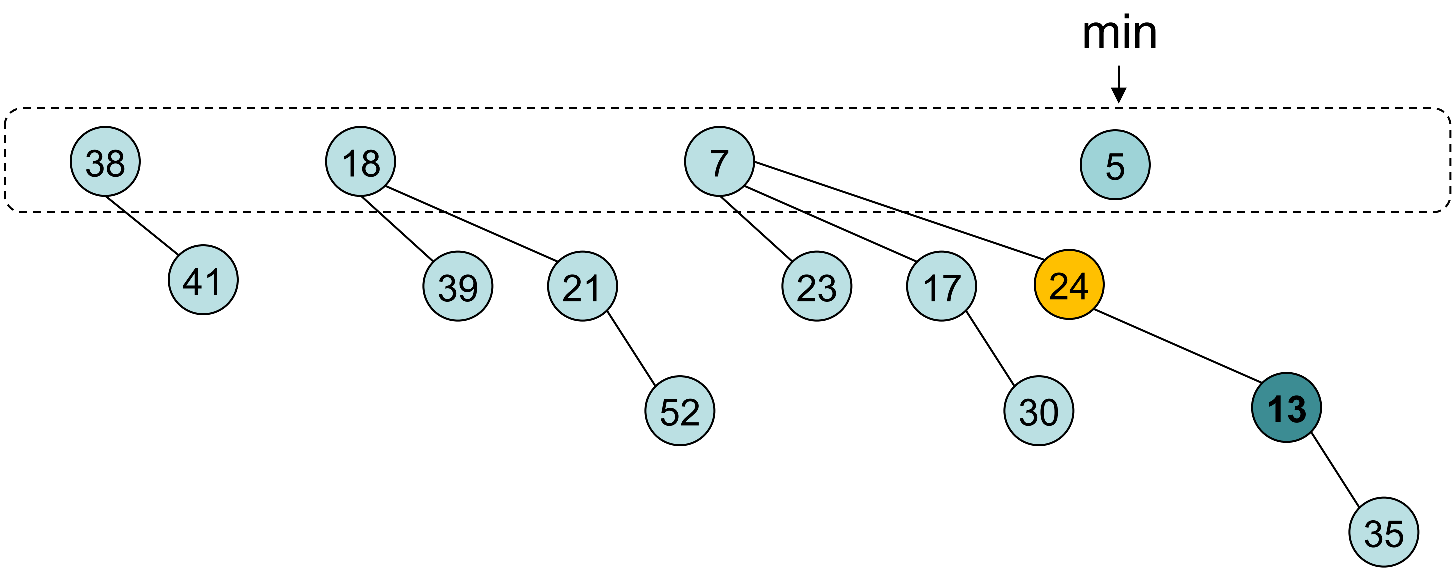


Fibonacci heaps

We include a smart **`decreaseKey()`** method from leftist heaps.

The method must be modified a bit, as we wish to use trees that are binomial trees, or partial binomial trees.

- Nodes are marked the first time child is removed.
- The second time a node gets a child removed, it is cut off, and becomes the root of a separate tree

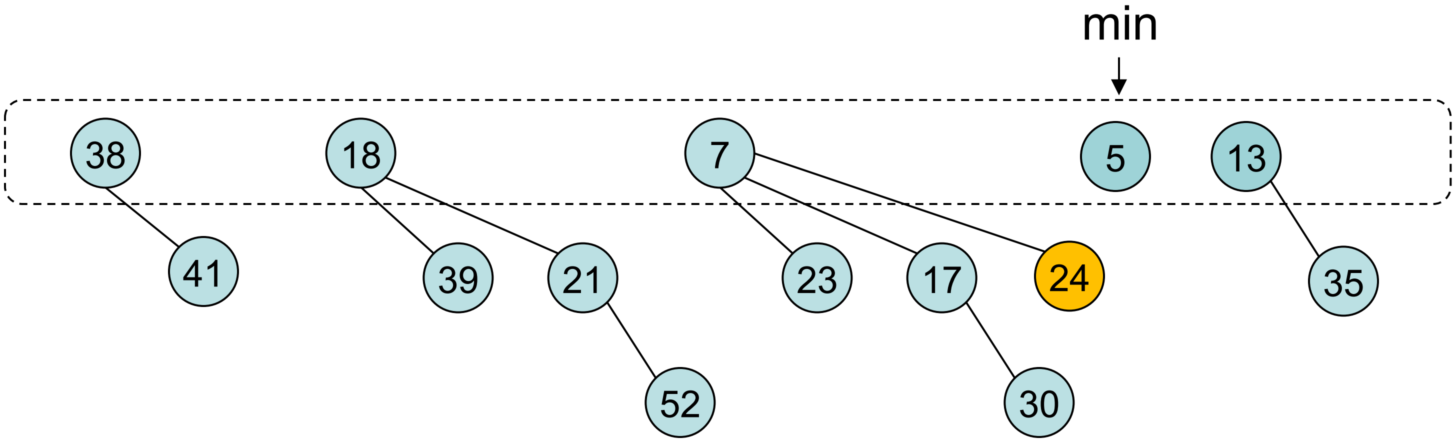


Fibonacci heaps

We include a smart **`decreaseKey()`** method from leftist heaps.

The method must be modified a bit, as we wish to use trees that are binomial trees, or partial binomial trees.

- Nodes are marked the first time child is removed.
- The second time a node gets a child removed, it is cut off, and becomes the root of a separate tree

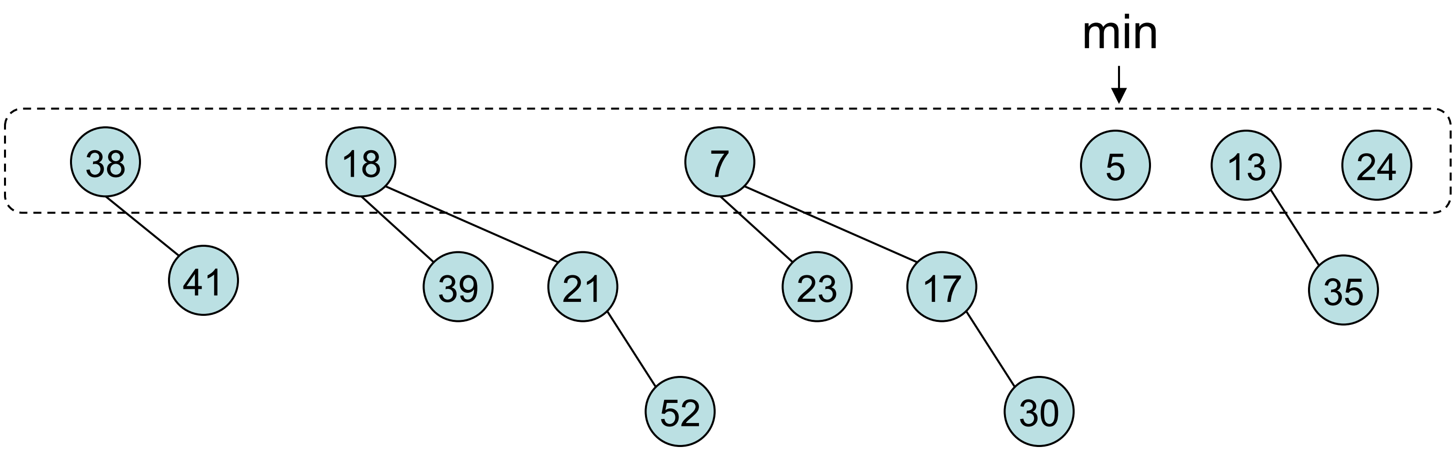


Fibonacci heaps

We include a smart **`decreaseKey()`** method from leftist heaps.

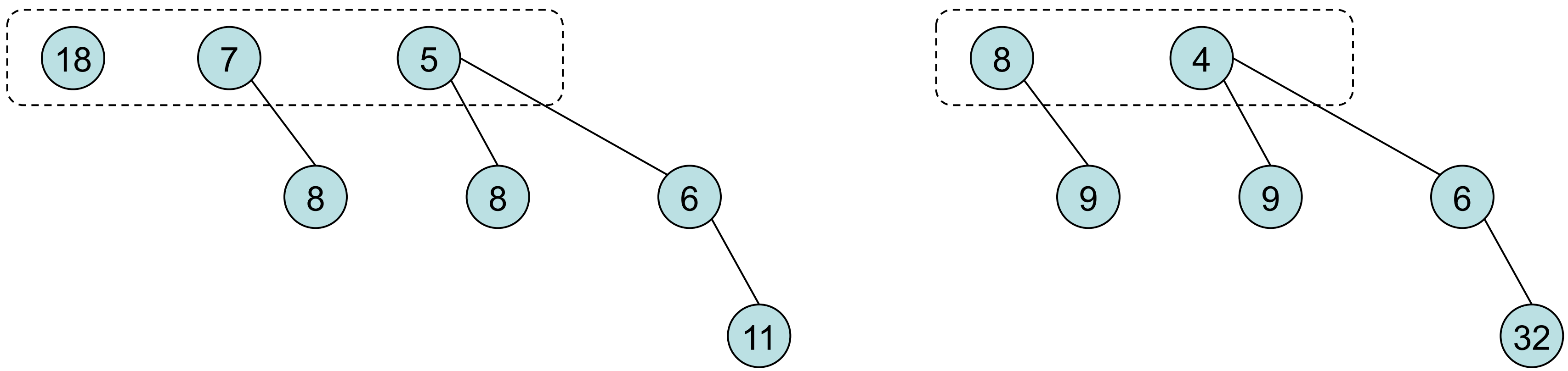
The method must be modified a bit, as we wish to use trees that are binomial trees, or partial binomial trees.

- Nodes are marked the first time child is removed.
- The second time a node gets a child removed, it is cut off, and becomes the root of a separate tree



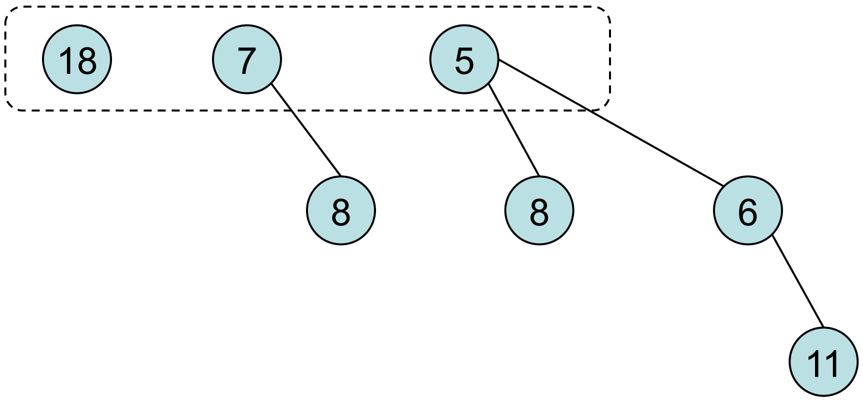
Fibonacci heaps

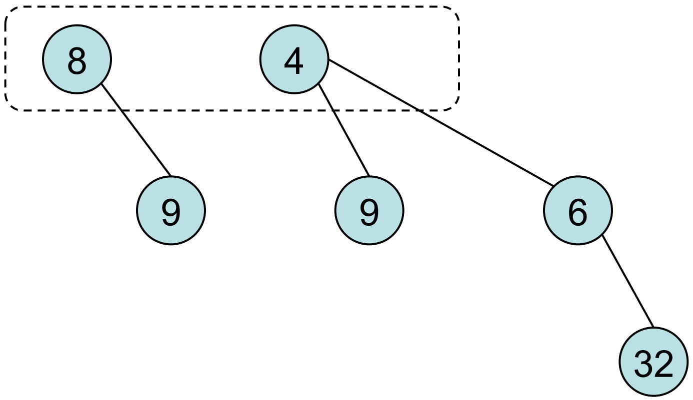
We use *lazy merging* / *lazy binomial queue*.

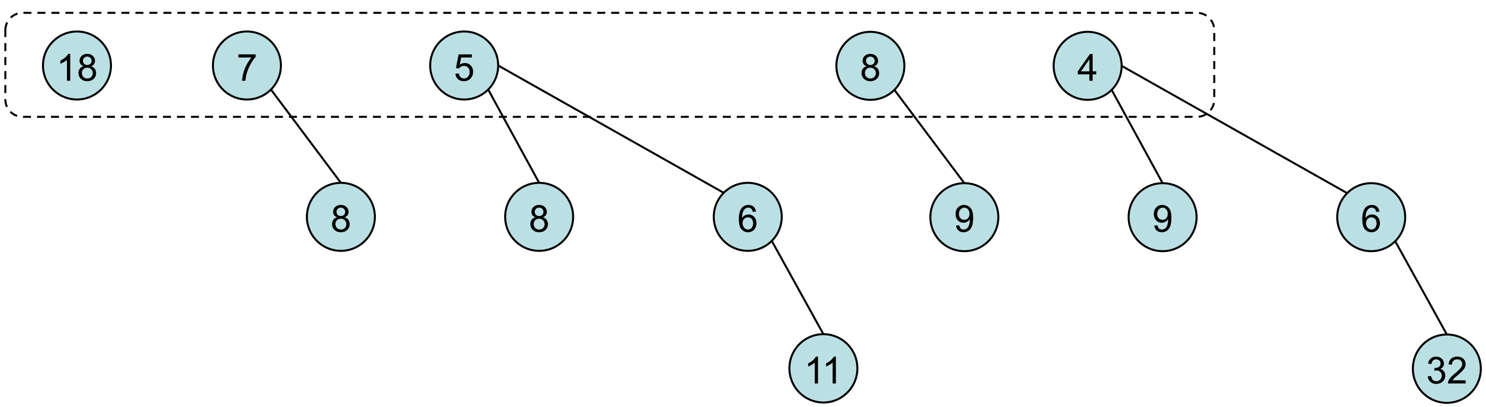


Fibonacci heaps

We use *lazy merging* / *lazy binomial queue*.







Fibonacci heaps

The problem with our **`decreaseKey()`**-method and *lazy merging* is that we have to clean up afterwards. This is done in by the **`deleteMin()`**-method which becomes expensive ($O(\log N)$ amortized time):

All trees are examined, we start with the smallest, and merge two and two, so that we get at most one tree of each size.

Each root has a number of children – this is used as the size of the tree. (Recall how we construct binomial trees, and that they may be partial as a result of **`decreaseKey()`**'s)

The trees are put in lists, one per size, and we begin merging, starting with the smallest.

Fibonacci heaps

	Amortized time
`insert()`	$O(1)$
`decreaseKey()`	$O(1)$
`merge()`	$O(1)$
`deleteMin()`	$O(\log N)$
`buildHeap()`	$O(N)$

(Run N **`insert()`** on an initially empty heap.)

(N = number of elements)