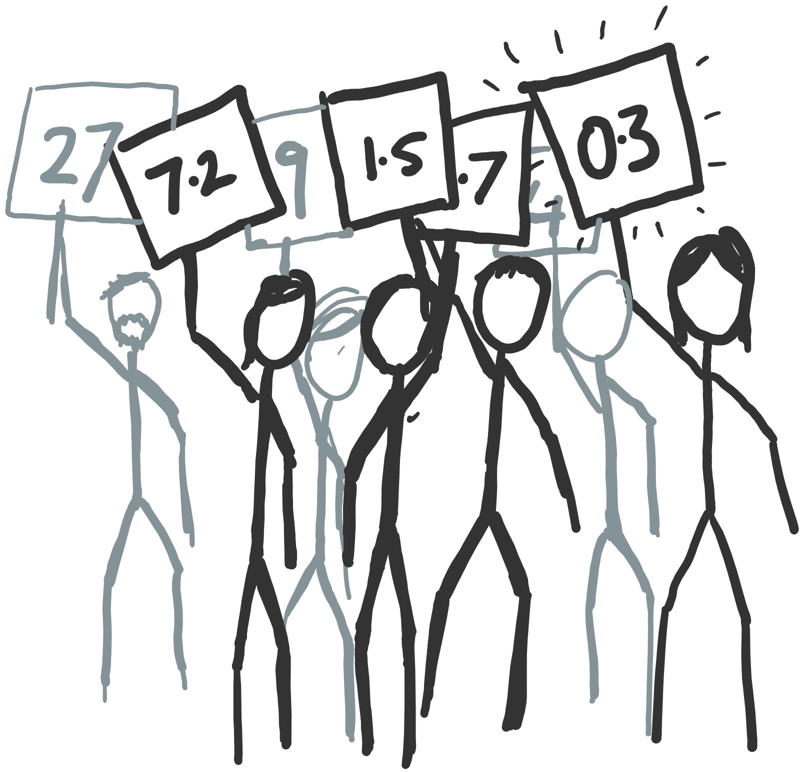
SECTION 7.5

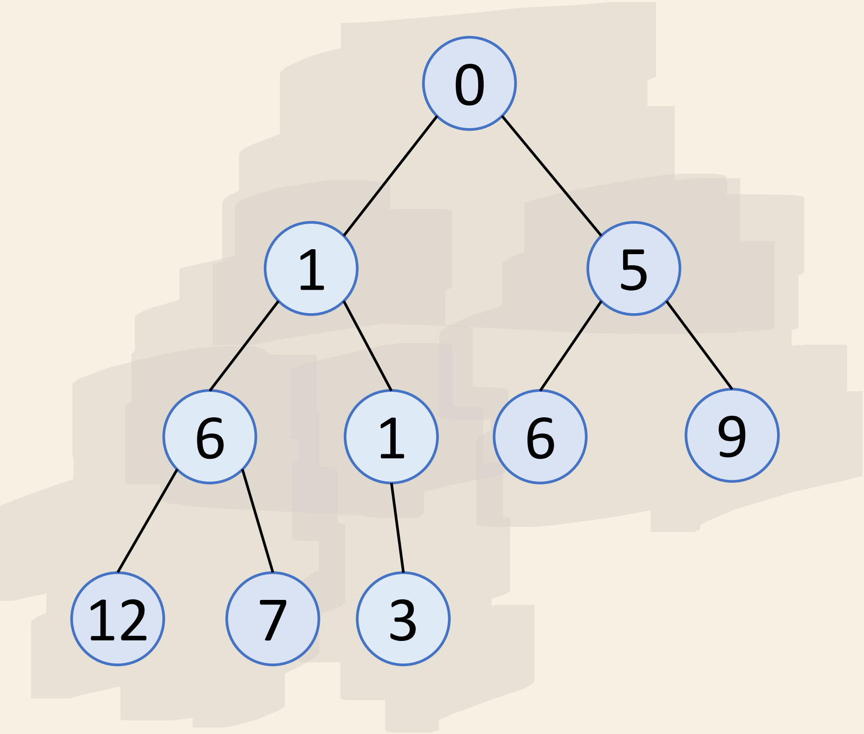
Three priority queues




```
AbstractDataType PriorityQueue
    # Holds a dynamic collection of items
    # Each item has a value v, and a key/priority k

    # Extract the item with the smallest key
    Pair<Key, Value> popmin()

    # Add v to the queue, and give it key k
    push(Value v, Key k)

    # For a value already in the queue, give it a new (lower) key
    decreasekey(Value v, Key k′)
```

```
    # Sometimes we also include methods for
    Pair<Key, Value> peekmin()
    delete(Value v)
    merge_with(PriorityQueue q)
```

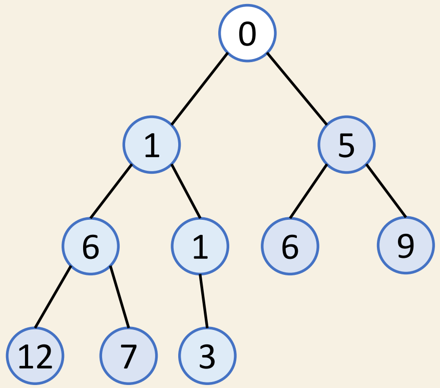
The binary heap



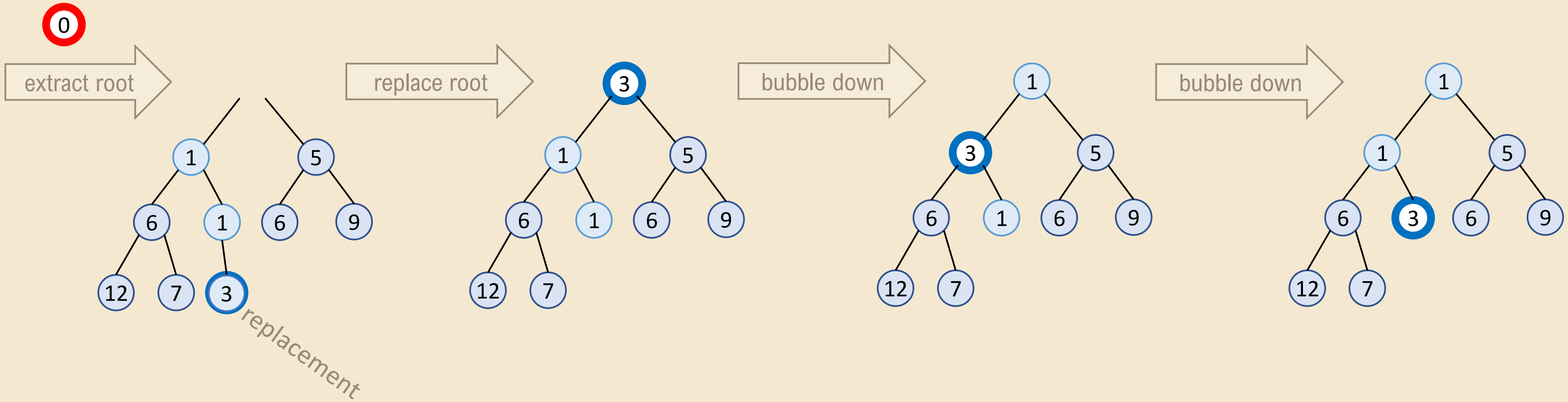
The heap property

every node's key is ≤ those of its children

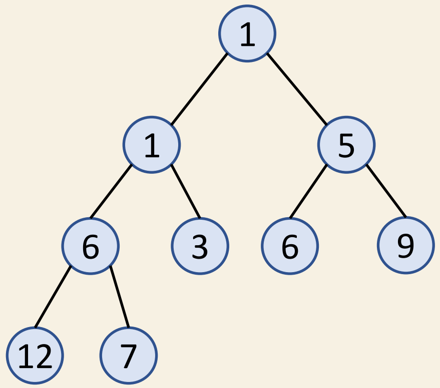
The binary heap



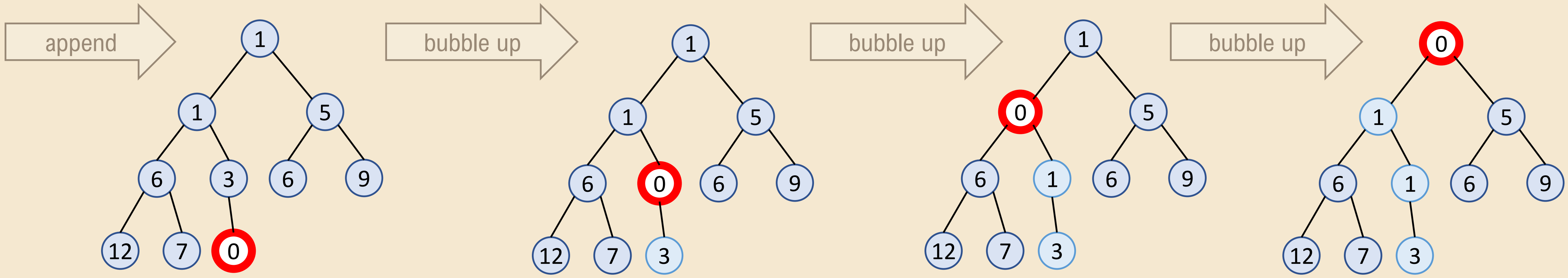
popmin()



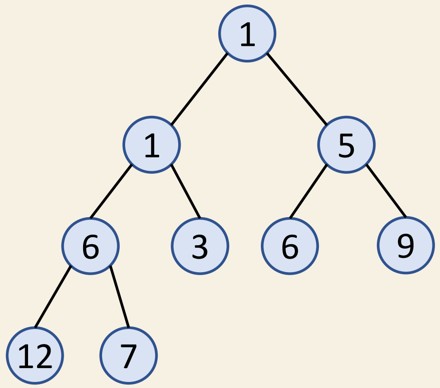
The binary heap



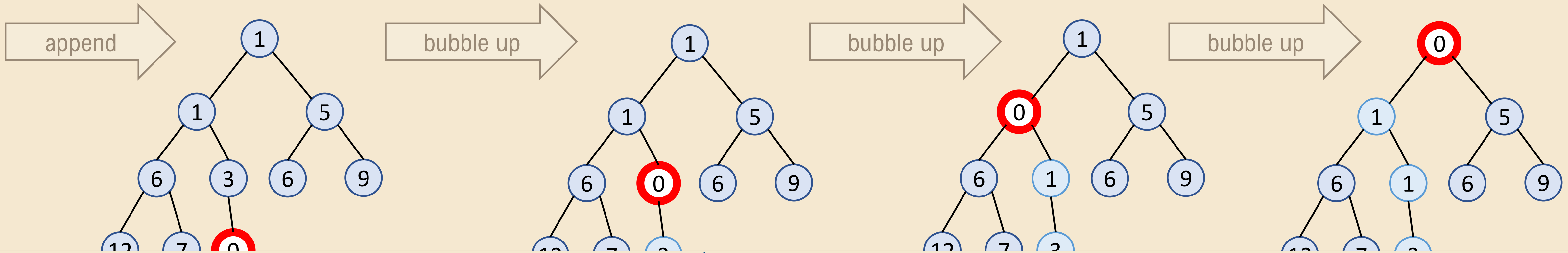
push(*new item*)



The binary heap



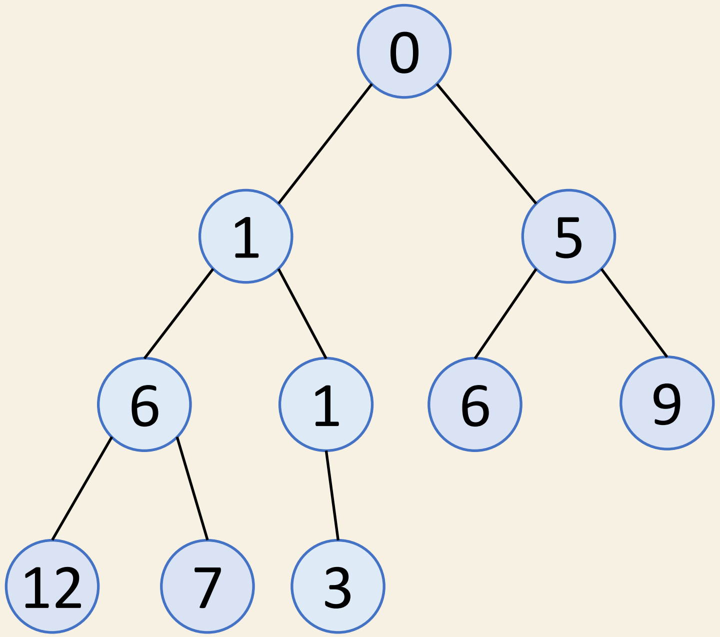
push(*new item*)



decreasekey(*item, new key*)

similar

The binary heap



SHAPE LEMMA

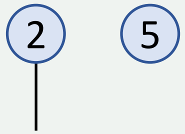
The height is $O(\log N)$
where N is the number of items in the heap

COMPLEXITY ANALYSIS

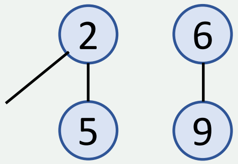
All operations are $O(\log N)$,

Binomial trees

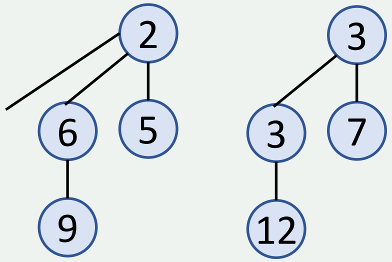
a tree of degree 0



two trees of degree 0
merge to give a tree of degree 1

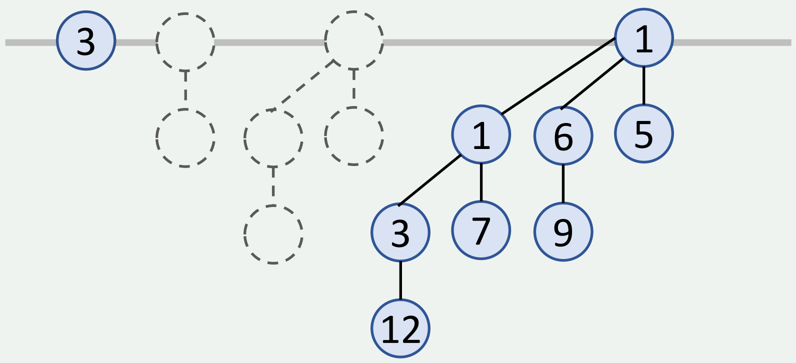


two trees of degree 1
merge to give a tree of degree 2



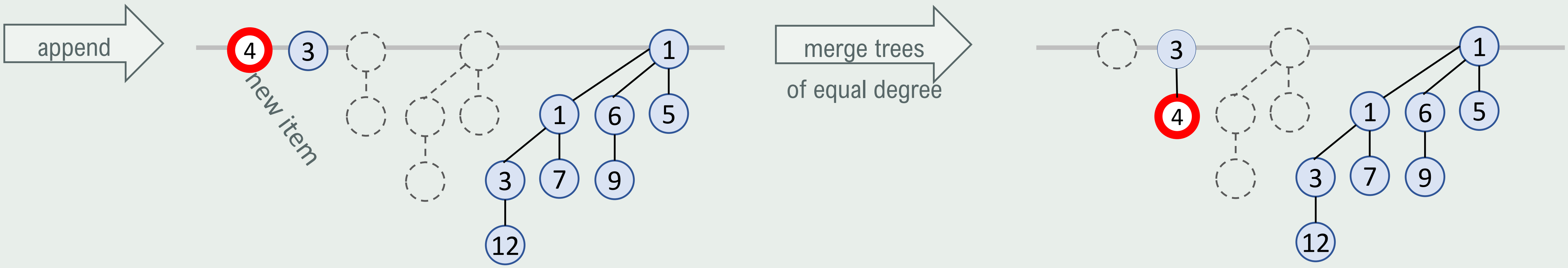
two trees of degree 2
merge to give a tree of degree 3

The binomial heap

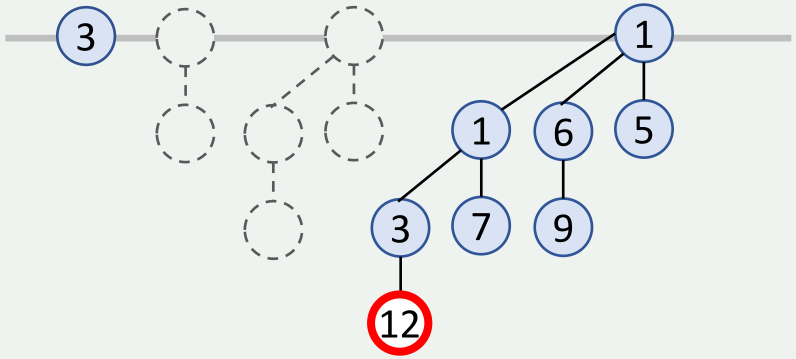


- a list of binomial trees, at most one of each degree
- each tree is a heap

push(*new item*)

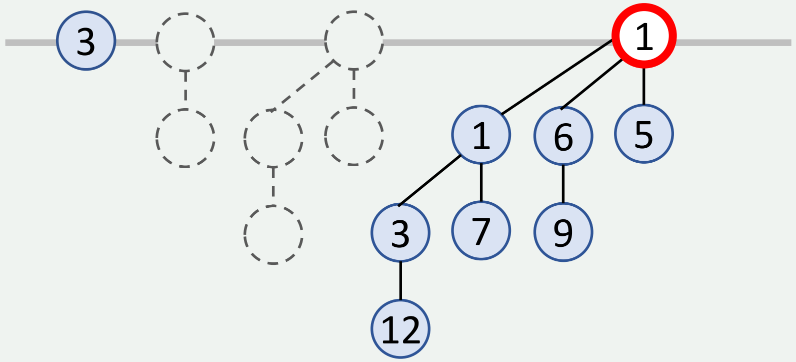


The binomial heap

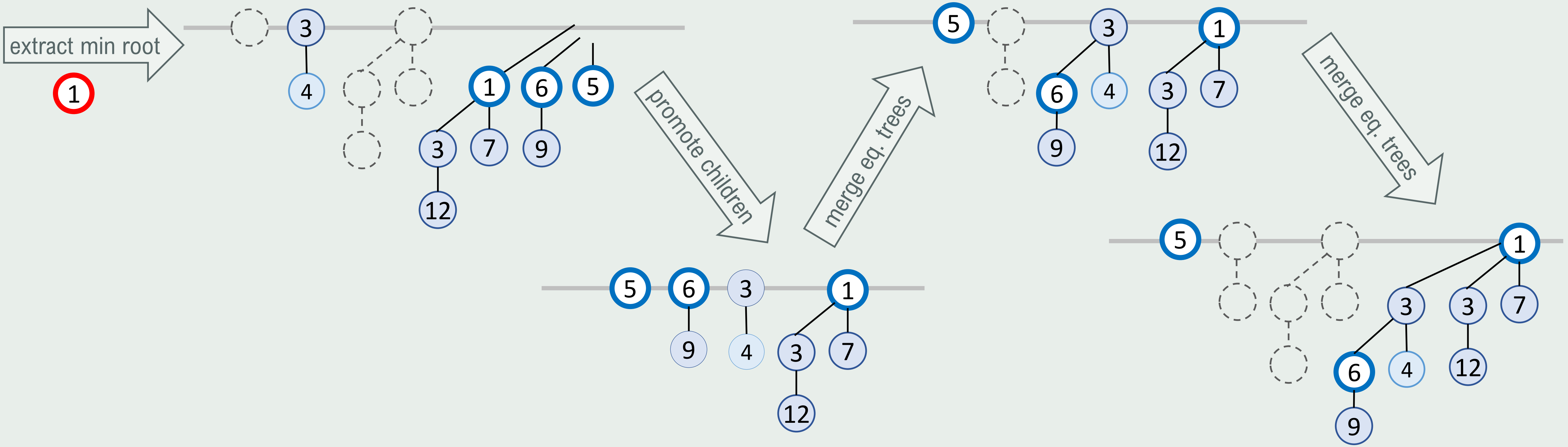


decreasekey(*item, new key*)

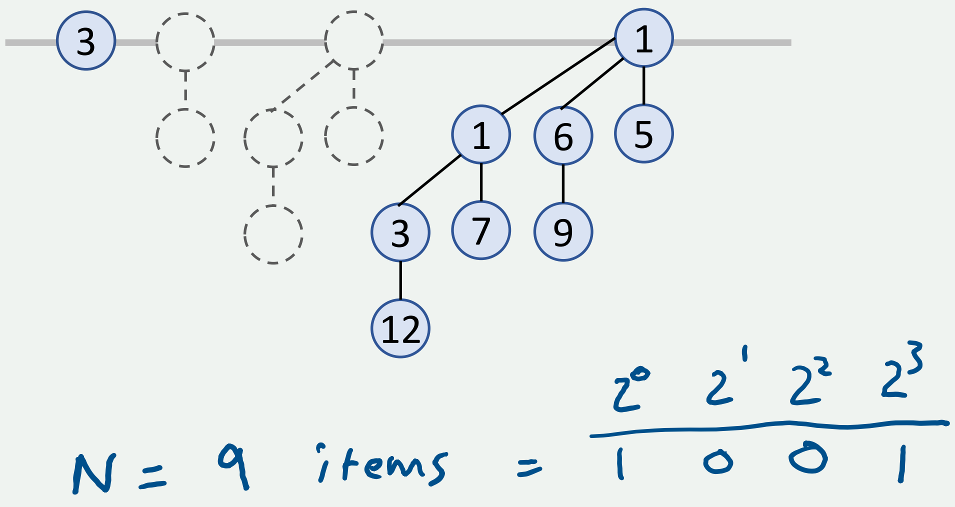
The binomial heap

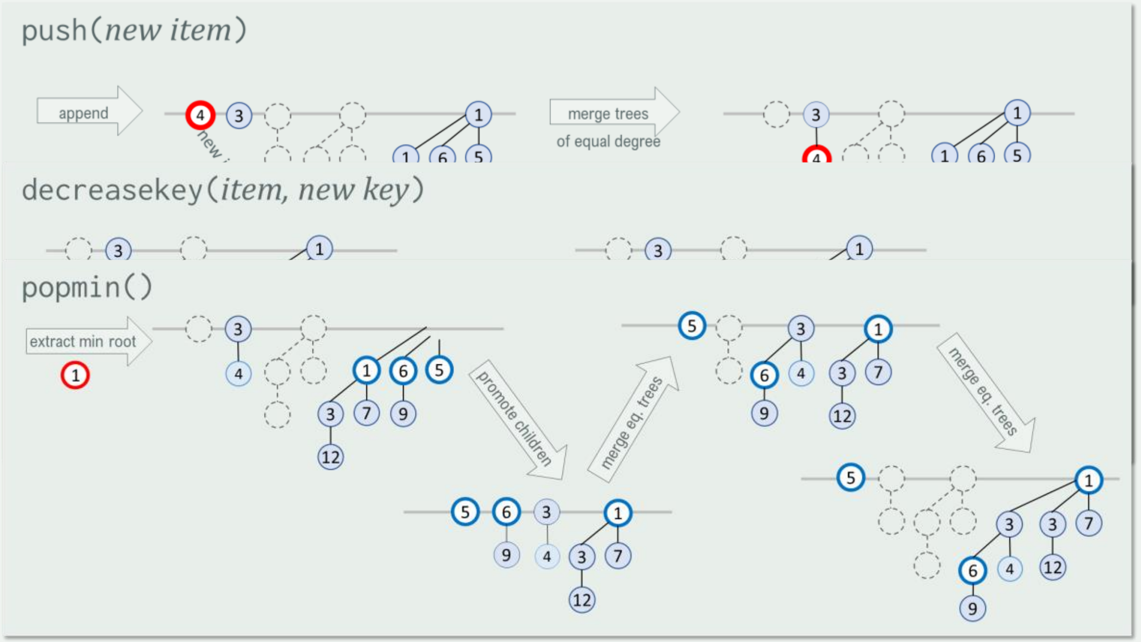


popmin()



The binomial heap





SHAPE THEOREM

- A binomial tree of degree k has 2^k items
- In a binomial heap with N items, the binary digits of N tell us which binomial trees are present

Also, in a binomial tree of degree k,

- the root has degree k
- its k children are binomial trees
- the height is k

COMPLEXITY ANALYSIS

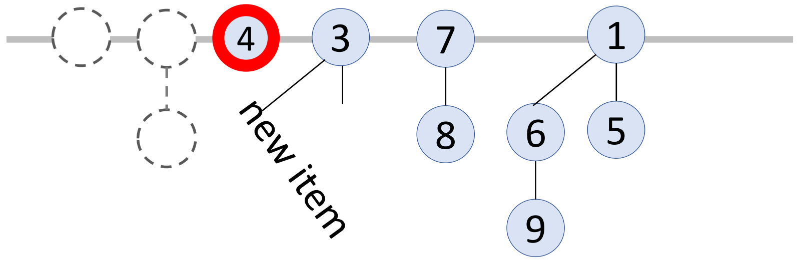
- push() is $O(\log N)$
  we have to merge $O(\log N)$ trees
- decreasekey() is $O(\log N)$
  in the worst case we have to bubble up from the bottom of the largest tree
- popmin() is $O(\log N)$
  scan $O(\log N)$ trees; promote $O(\log N)$ children; do $O(\log N)$ merges to recover the heap

	popmin	push	decreasekey
binary heap	$O(\log N)$	$O(\log N)$	$O(\log N)$
binomial heap	$O(\log N)$	$O(\log N)$	$O(\log N)$

And what about aggregate costs?

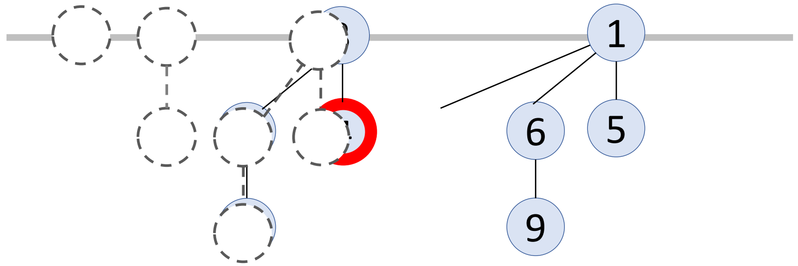
	popmin	push	decreasekey
binary heap	$O(\log N)$	$O(\log N)$	$O(\log N)$
binomial heap	$O(\log N)$	$O(\log N)$	$O(\log N)$

And what about aggregate costs?



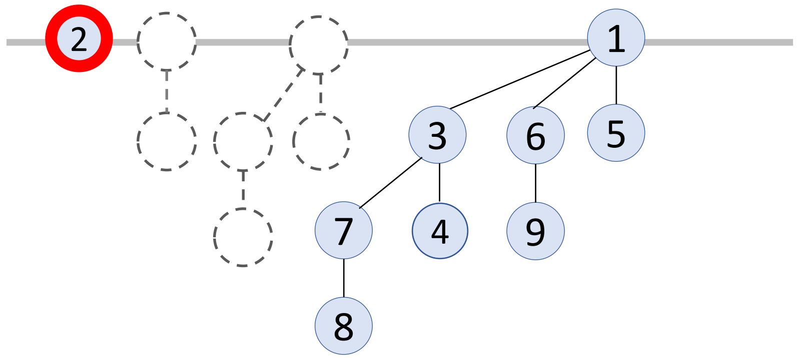
	popmin	push	decreasekey
binary heap	$O(\log N)$	$O(\log N)$	$O(\log N)$
binomial heap	$O(\log N)$	$O(\log N)$	$O(\log N)$

And what about aggregate costs?



	popmin	push	decreasekey
binary heap	$O(\log N)$	$O(\log N)$	$O(\log N)$
binomial heap	$O(\log N)$	~~$O(\log N)$~~ $O(1)$ amortized [Ex. sheet 6 q. 2, 4]	$O(\log N)$

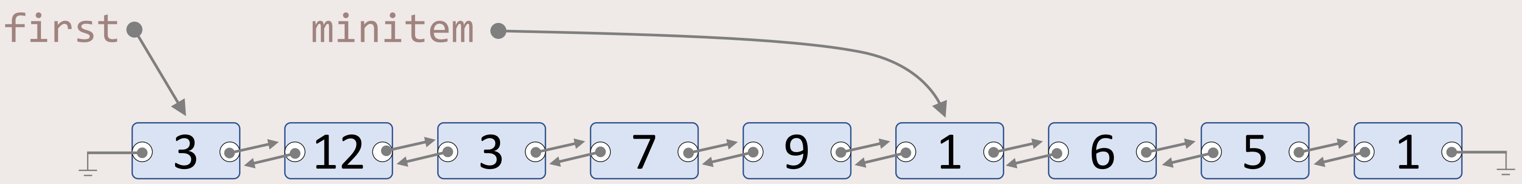
And what about aggregate costs?



Dijsktra's algorithm makes $O(E)$ calls to push/decreasekey, and only $O(V)$ calls to `popmin`.

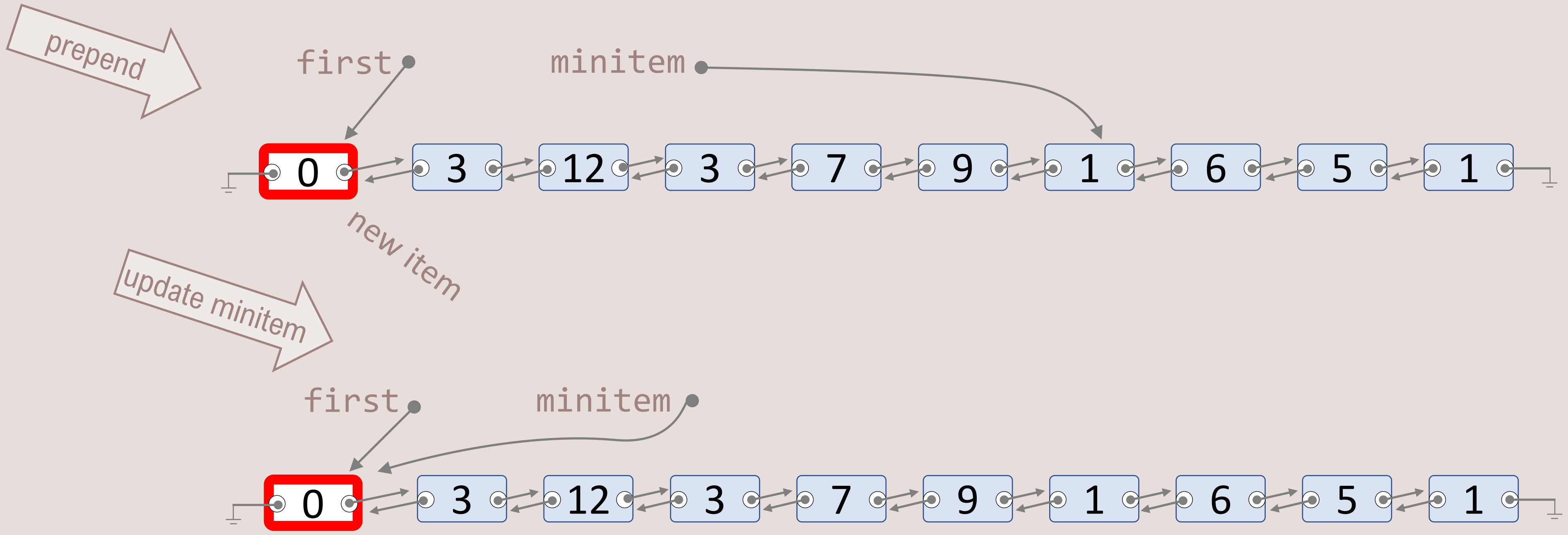
QUESTION. Can we make both push and decreasekey be $O(1)$?

Linked-list priority queue

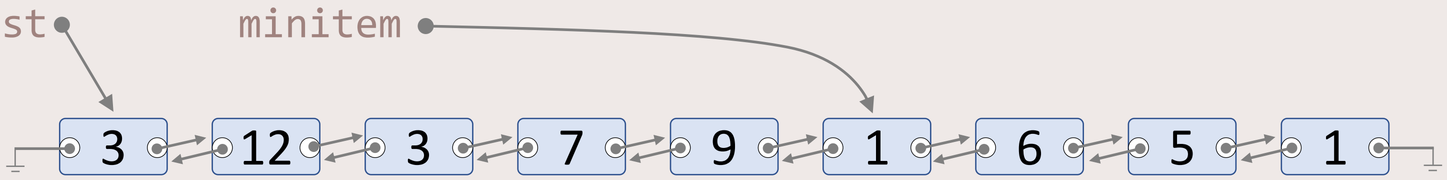


push is O(1)

push(*new item*)

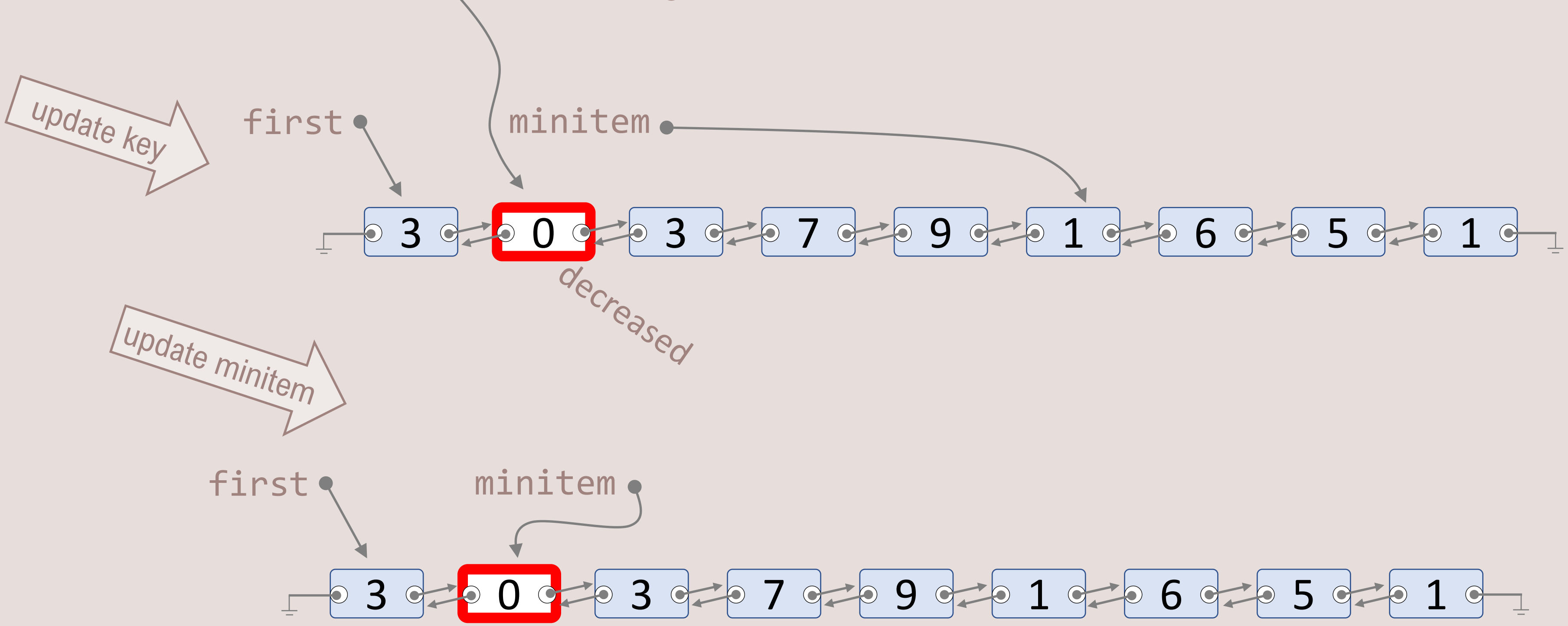


Linked-list priority queue

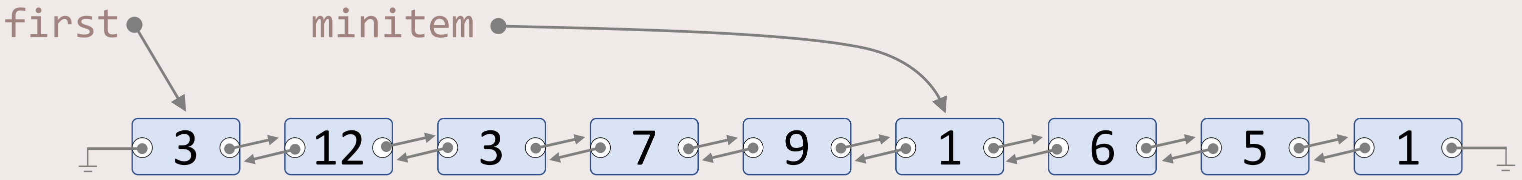


decreasekey is O(1)

decreasekey(*item*, *new key*)

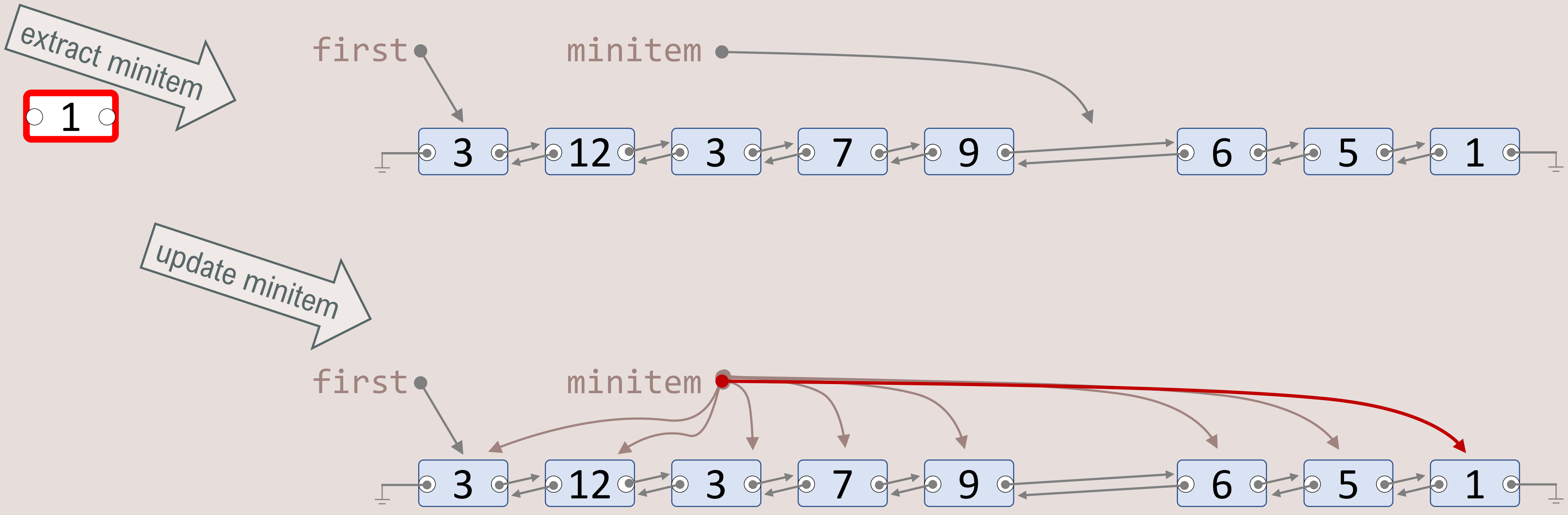


Linked-list priority queue



popmin is O(N)

popmin()



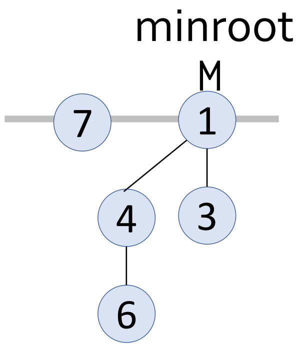
but N pushes are only $O(N)$ (see heapsort, §2.10)

	popmin	push	decreasekey
binary heap	$O(\log N)$	$O(\log N)$	$O(\log N)$
binomial heap	$O(\log N)$	$O(1)$ amort	$O(\log N)$
linked list	$O(N)$	$O(1)$	$O(1)$
Fibonacci heap	$O(\log N)$ amort	$O(1)$ amort	$O(1)$ amort

- Be lazy
- Do cleanup in batches
- Give your data enough structure that you only need to touch a little bit of it

SECTION 7.6

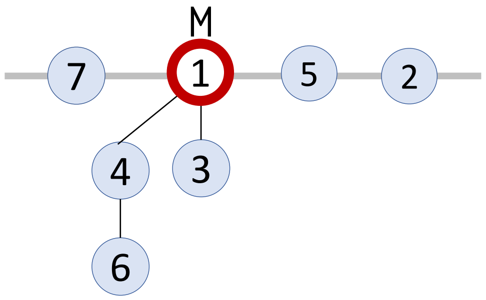
The Fibonacci Heap



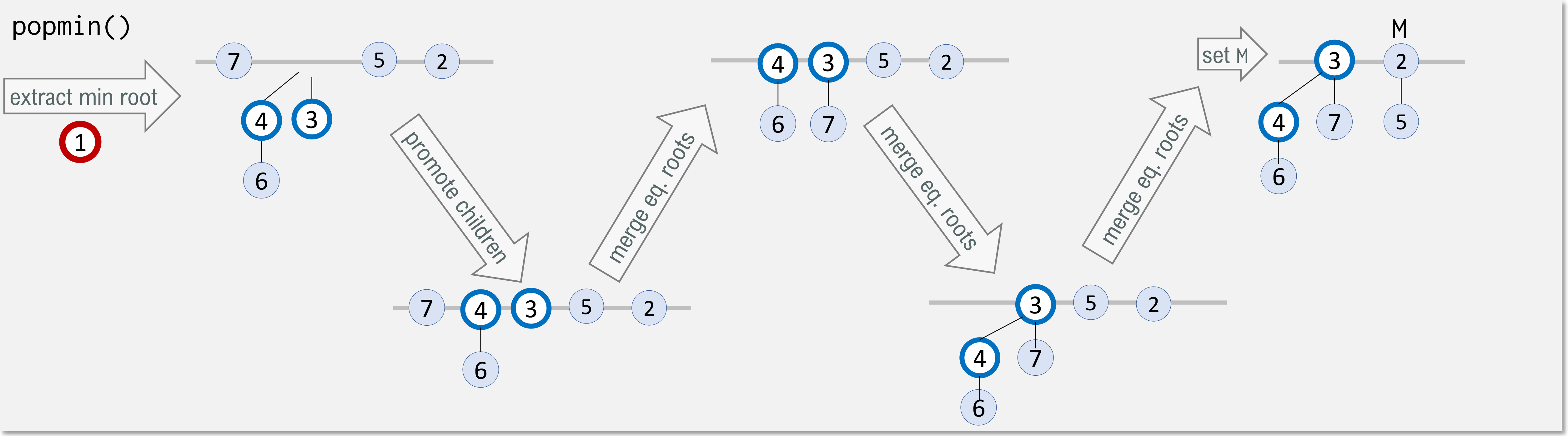
- store a list of trees, each a heap
- trees can have any shape
- keep track of the minroot

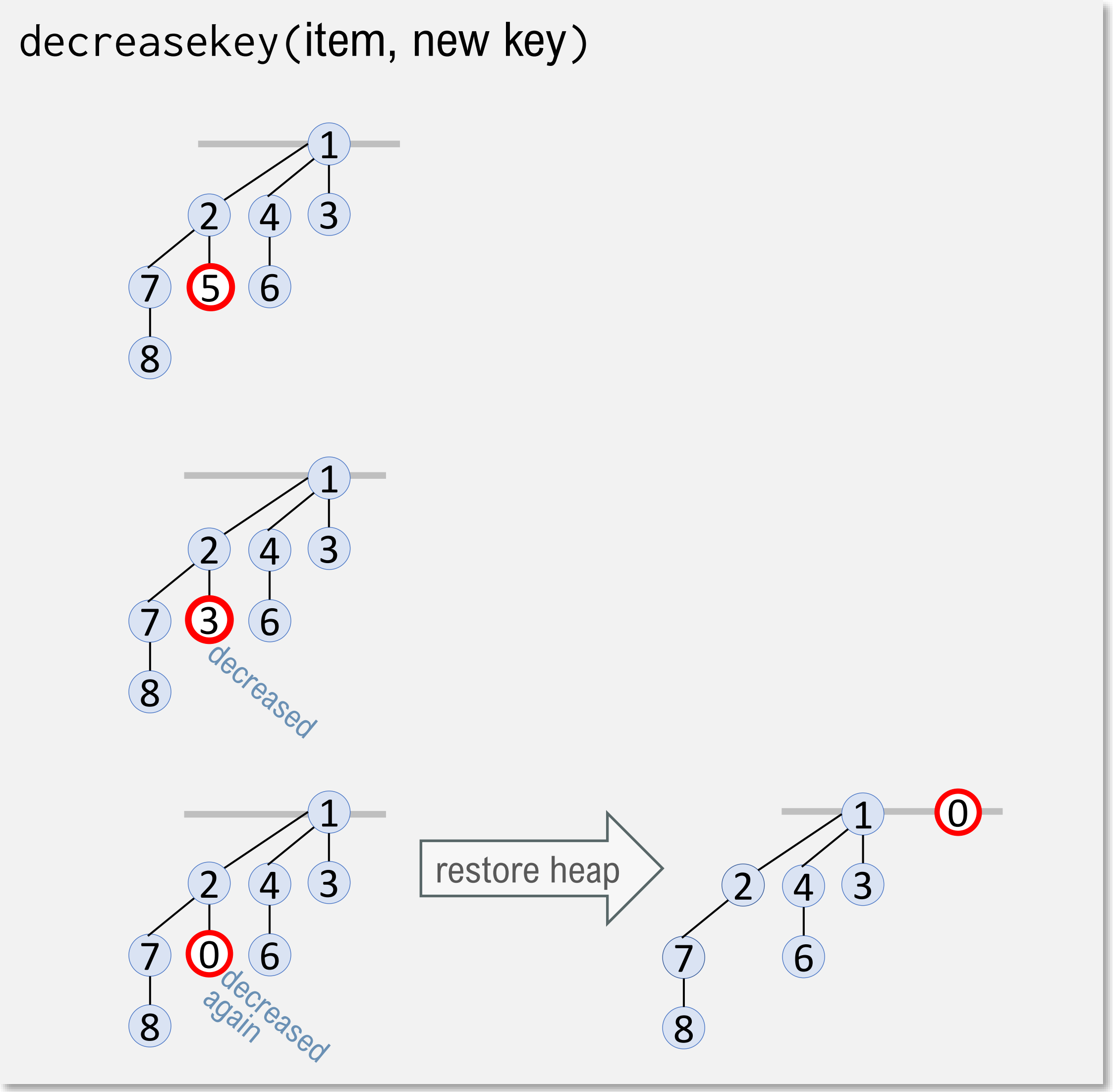
```
# Maintain a list of heaps (i.e. store a pointer to the root of each heap)
roots = []

# Maintain a pointer to the smallest root
minroot = None

def push(Value v, Key k):
    create a new heap h consisting of a single item (v,k)
    add h to the list of roots
    update minroot if minroot is None or k < minroot.key
```

push(new item)




```
12  def popmin():
13      take note of minroot.value and minroot.key
14      delete the minroot node, and promote its children to be roots
15      # cleanup the roots
16      while there are two roots with the same degree:
17          merge those two roots, by making the larger root a child of the smaller
18      update minroot to point to the root with the smallest key
19      return the value and key we noted in line 13
```

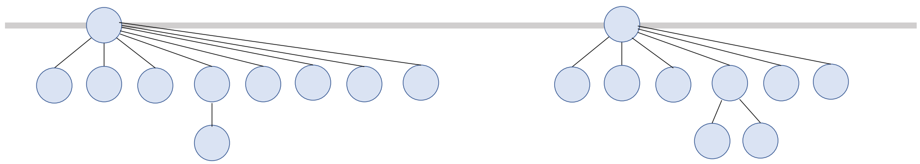




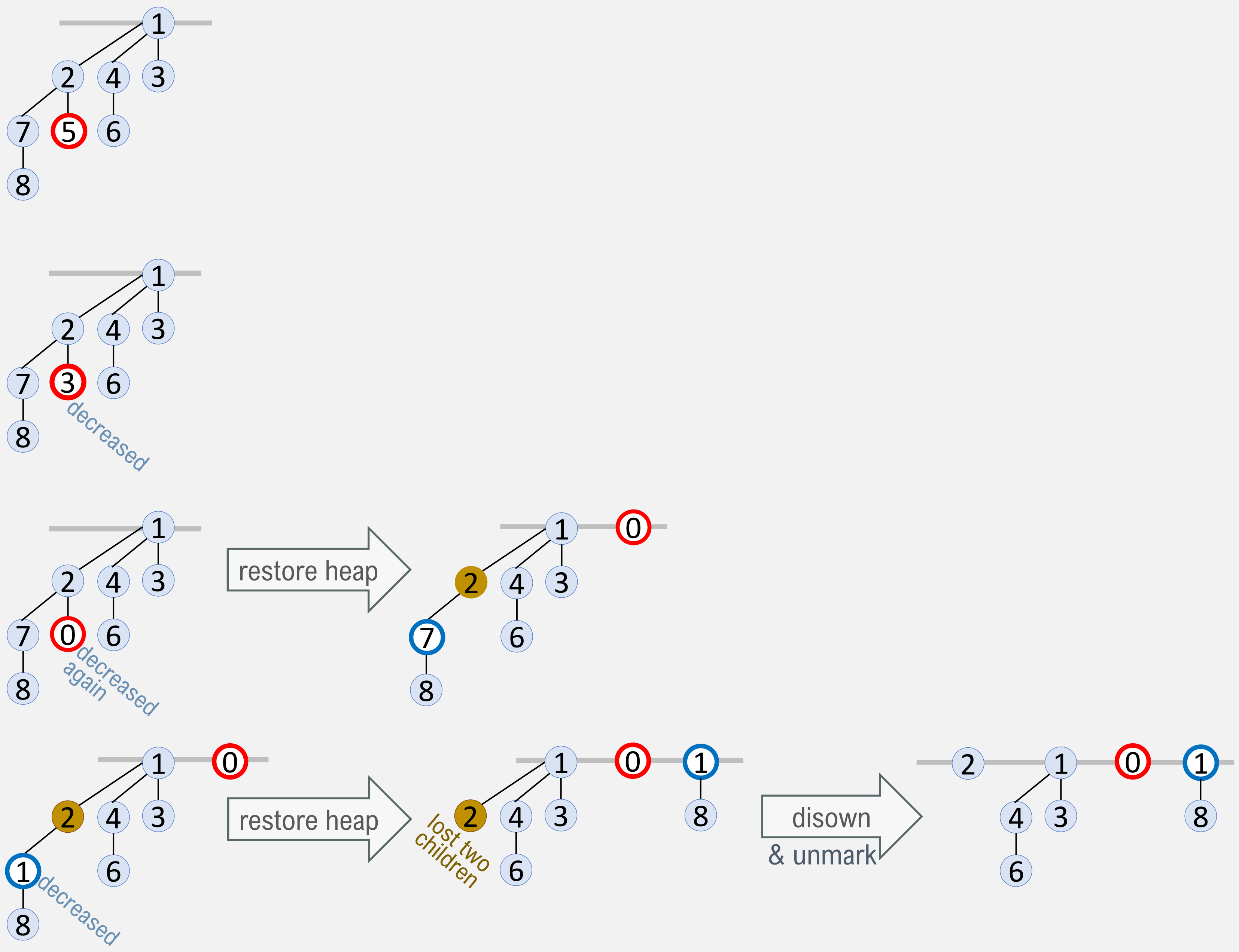
LAZY STRATEGY

Dump heap-violating nodes into the root list, to be cleaned up by the next `popmin()`

... **but** we might end up with a heap with wide shallow trees, which will make `popmin()` slow



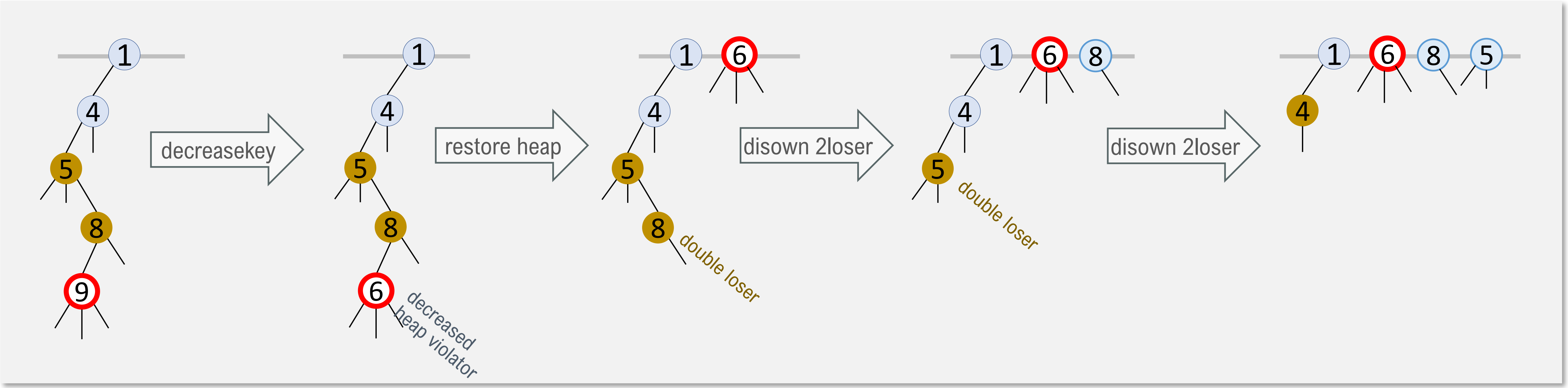
decreasekey(item, new key)



Rule 1. Lose one child, and you're marked a LOSER

Rule 2. Lose two children, and you're dumped into the root list

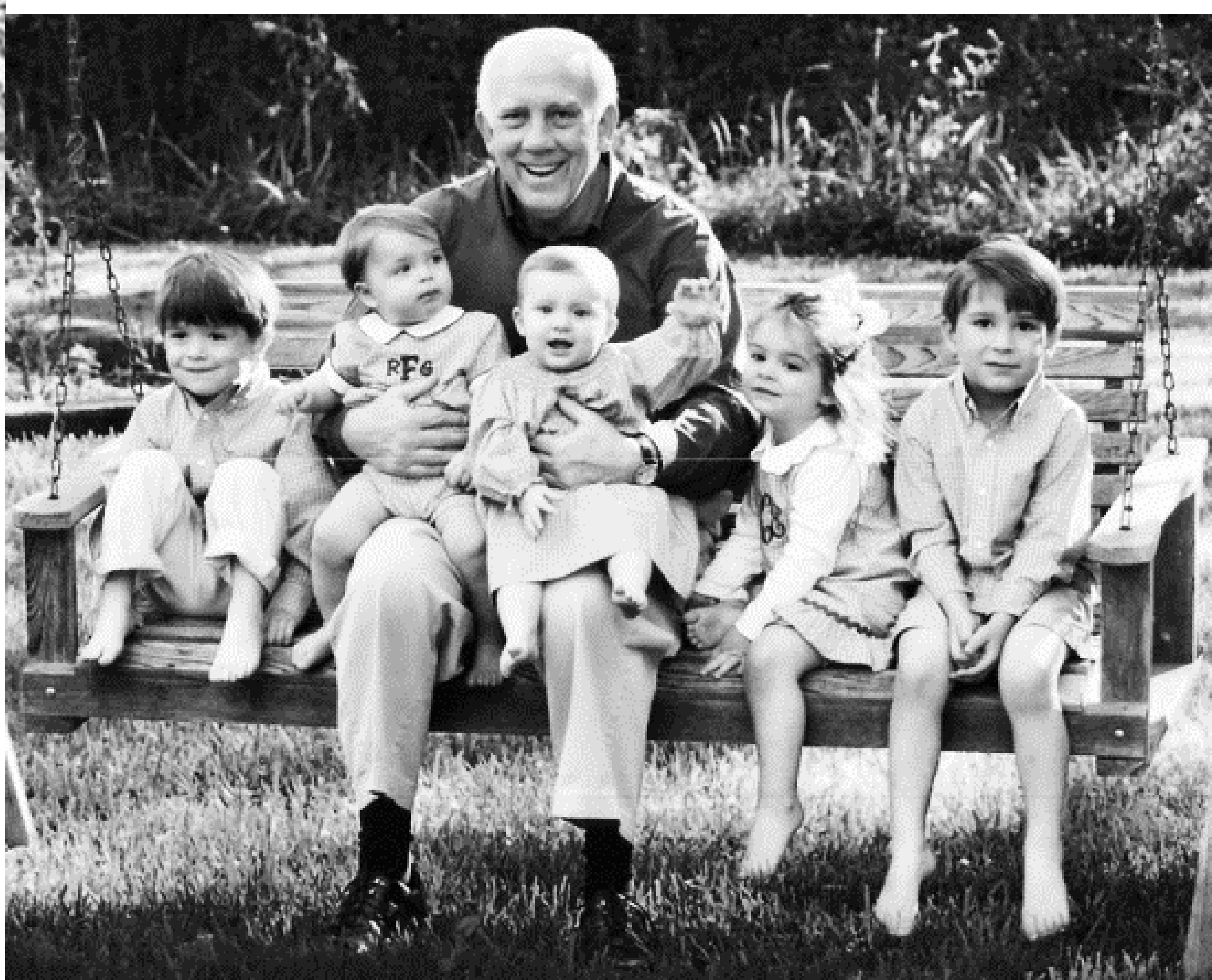
```
# Every node will store a flag, n.loser = True / False

def decreasekey(v, k′):
    let n be the node where this value is stored
    n.key = k′
    if n violates the heap condition:
        repeat:
            p = n.parent
            remove n from p.children
            insert n into the list of roots, updating minroot if necessary
            n.loser = False
            n = p
        until p.loser == False
        if p is not a root:
            p.loser = True

# Modify popmin so that when we promote minroot's children, we erase any loser flags
```



Sometimes it pays
to let mess build up

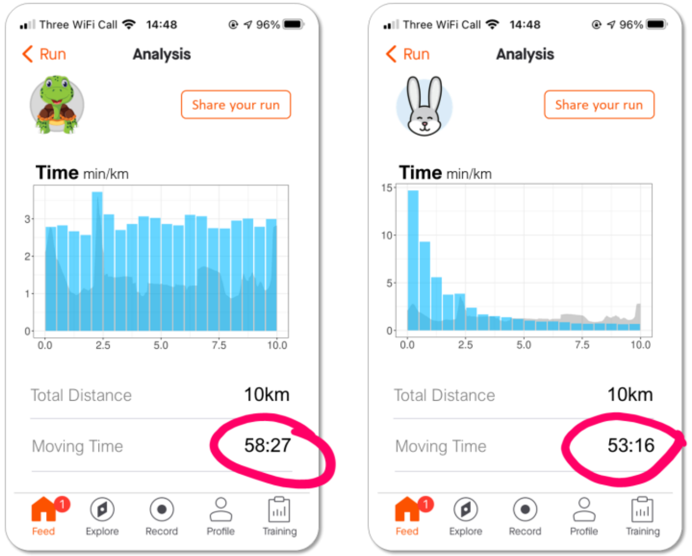
Your parents want
lots of grandchildren*

* and they'll disown you if
you don't have enough



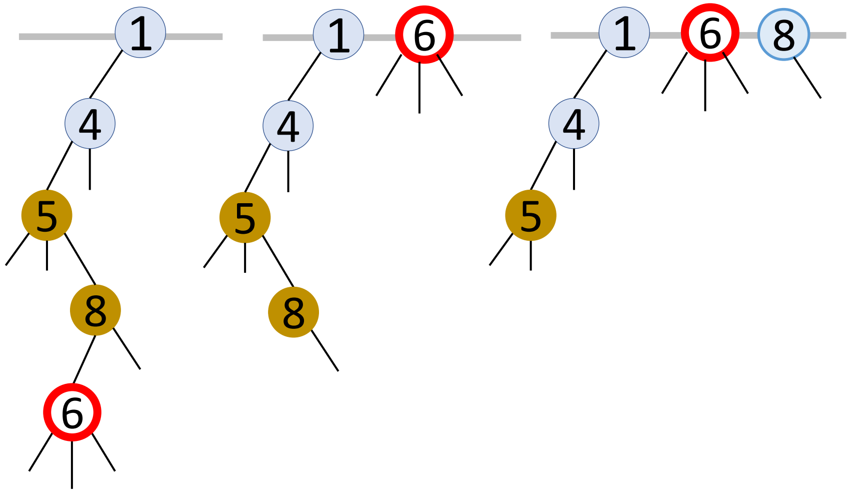
SECTION 7.8

Amortized analysis of the Fibonacci Heap

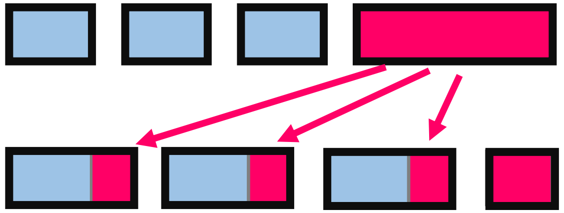


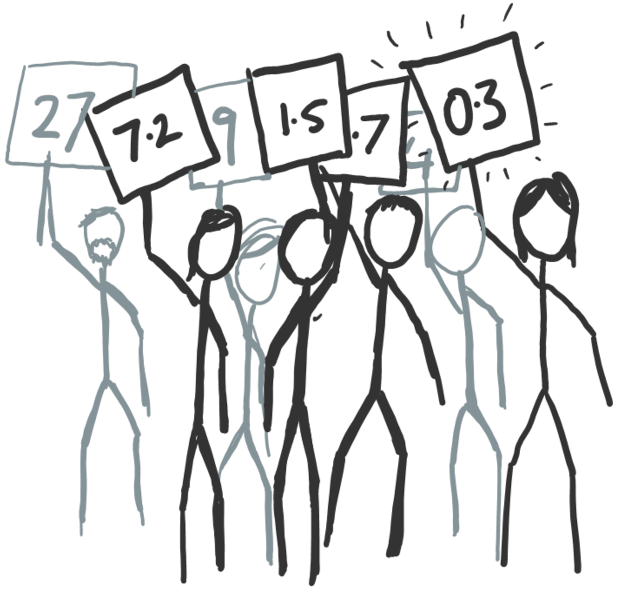
Three WiFi Call 14:48 96%
Run
Analysis
Share your run
Time min/km
Total Distance
10km
Moving Time
58:27
Feed
Explore
Record
Profile
Training
Three WiFi Call 14:48 96%
Run
Analysis
Share your run
Time min/km
Total Distance
10km
Moving Time
53:16
Feed
Explore
Record
Profile
Training

Φ

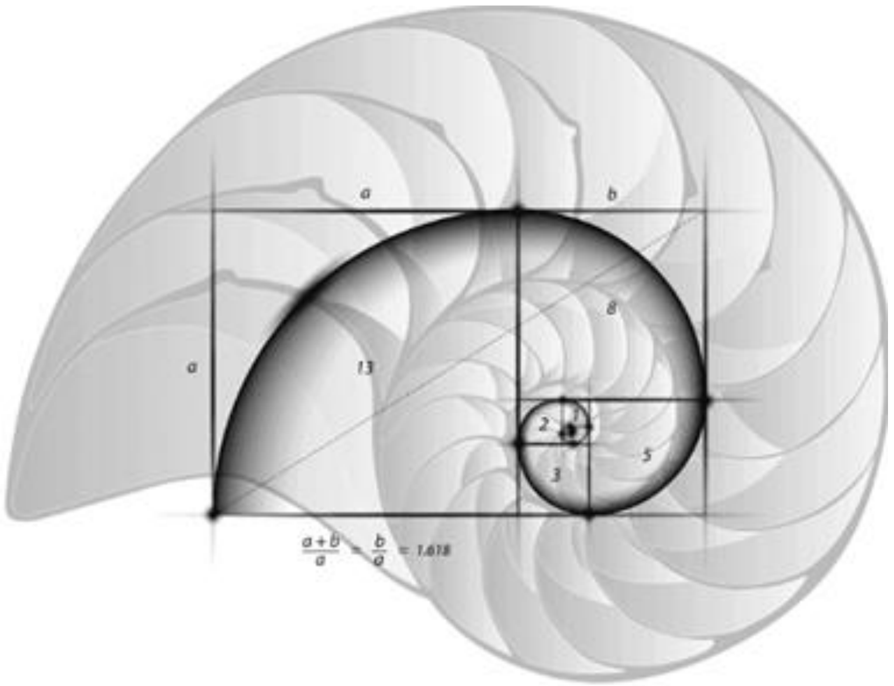


1
4
5
8
6
1
6
4
5
8
1
6
8
4
5





27
7·2
1·5
·7
0·3



FIBONACCI HEAP
COMPLEXITY ANALYSIS

COMPLEXITY ANALYSIS

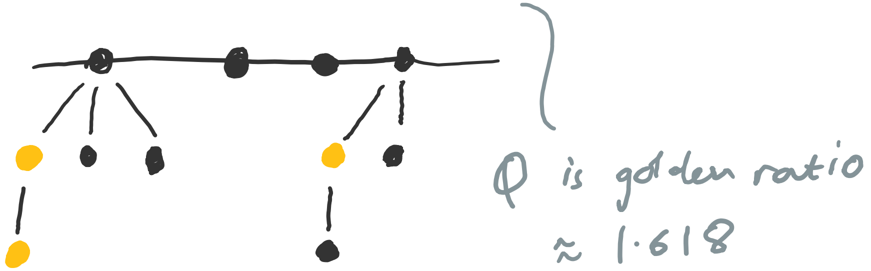
In a Fibonacci heap with N items,
using the potential function

$$\Phi = \text{num.roots} + 2 \times \text{num.losers},$$

- `push()` has amortized cost $O(1)$
- `decreasekey()` has amortized cost $O(1)$
- `popmin()` has amortized cost $O(\log N)$

SHAPE THEOREM

Every node has degree $\leq \log_\phi N$



BINOMIAL HEAP
COMPLEXITY ANALYSIS

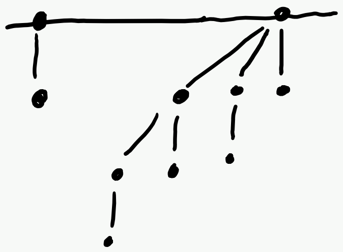
COMPLEXITY ANALYSIS

In a binomial heap with N items

- `push()` is $O(\log N)$
- `decreasekey()` is $O(\log N)$
- `popmin()` is $O(\log N)$

SHAPE THEOREM

The largest tree has degree $\leq \log_2 N$



$\Phi = \text{num.roots} + 2 \times \text{num.losers}$


```
def push(Value v, Key k):
    create a new heap h consisting of a single item (v,k)
    add h to the list of roots
    update minroot if minroot is None or k < minroot.key
```

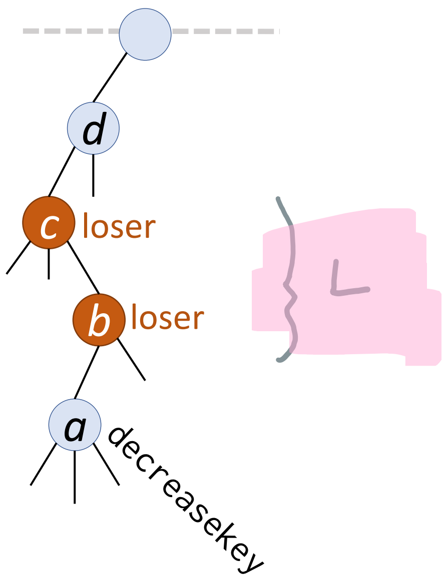
$c = O(1)$

$\Delta\Phi = 1$

$\text{am. cost} = c + \Delta\Phi = O(1)$

$\Phi = \text{num.roots} + 2 \times \text{num.losers}$


```
def decreasekey(v, k′):
    let n be the node where this value is stored
    n.key = k′
    if n violates the heap condition:
        repeat:
            p = n.parent
            remove n from p.children
            insert n into the list of roots, updating minroot if necessary
            n.loser = False
            n = p
        until p.loser == False
        if p is not a root:
            p.loser = True
```



CASE I: no heap violation

$c = O(1)$ $\Delta\Phi = 0$ $\Rightarrow$ $c + \Delta\Phi = O(1)$

CASE II: heap violation

1. move a to rootlist
   $c = O(1)$ $\Delta\Phi = 1$ or $\Delta\Phi = -1$ if a was loser $\Rightarrow$ $c + \Delta\Phi = O(1)$

2. Move up L losers also
   $c = O(L)$ $\Delta\Phi = +L - 2L = -L$ $\Rightarrow$ $c + \Delta\Phi = O(1)$

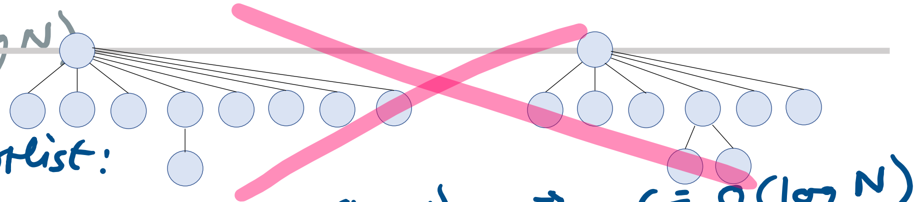
3. Mark d as a loser
   $c = O(1)$ $\Delta\Phi = 2$ unless d is root, $\Delta\Phi = 0$ $\Rightarrow$ $c + \Delta\Phi = O(1)$

in both cases, total amortized cost is $O(1)$

```
12 def popmin():
13     take note of minroot.value and minroot.key
14     delete the minroot node, and promote its children to be roots
15     # cleanup the roots
16     while there are two roots with the same degree:
17         merge those two roots, by making the larger root a child of the smaller
18     update minroot to point to the root with the smallest key
19     return the value and key we noted in line 13
```

1. cut out minroot, promote its children: $c = O(\#\text{children})$, $\Delta\Phi \le -1 + \#\text{children}$ $\Rightarrow c + \Delta\Phi = O(\log N)$

2. cleanup: we'll see that $c + \Delta\Phi = O(\log N)$

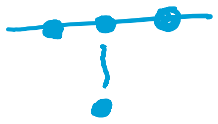


3. fix minroot, by scanning the cleaned-up rootlist; there's at most one tree of each degree; max degree $= O(\log N)$ $\Rightarrow$ $c = O(\log N)$

} total amortized cost is $O(\log N)$

```
20 def cleanup(roots):
21     root_array = [None, None, ....]
22     for each tree t in roots:
23         x = t
24         while root_array[x.degree] is not None:
25             u = root_array[x.degree]
26             root_array[x.degree] = None
27             x = merge(x, u)
28         root_array[x.degree] = u
29     roots = list of non-None values from root_array
```

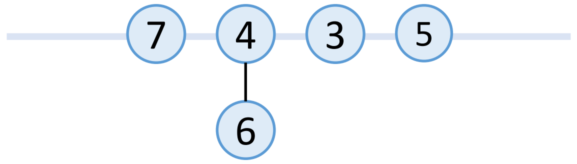
num.roots dec. by 1

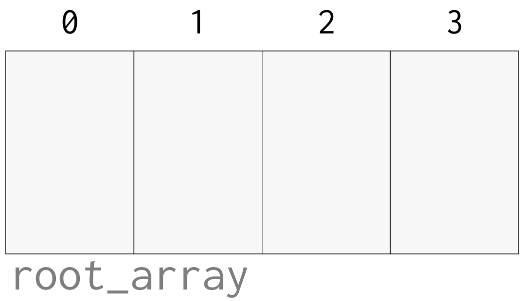


num roots inc. by # children

num. losers decreases, maybe

```
def cleanup(roots):
    root_array = [None, None, ....]
    for each tree t in roots:
        x = t
        while root_array[x.degree] is not None:
            u = root_array[x.degree]
            root_array[x.degree] = None
            x = merge(x, u)
        root_array[x.degree] = u
    roots = list of non-None values from root_array
```





At the end of cleanup, we want to have $\leq$1 tree of any given degree.

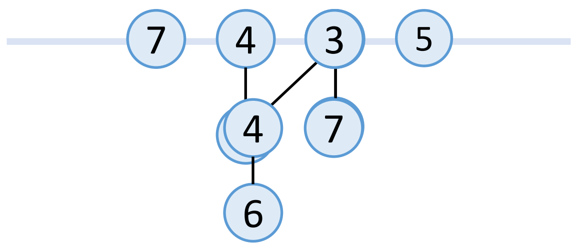
SHAPE THEOREM
Every node has degree $\leq \log_\phi N$

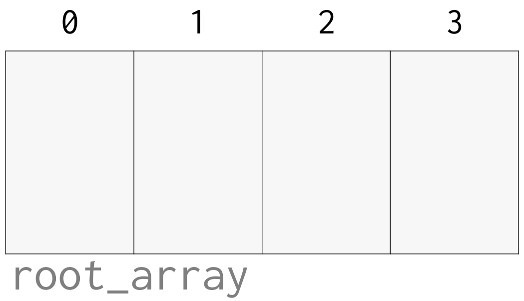
To fit them these trees, we'll need an array of size $\leq \log_\phi N + 1$

```
def cleanup(roots):
    root_array = [None, None, ....]
    for each tree t in roots:
        x = t
        while root_array[x.degree] is not None:
            u = root_array[x.degree]
            root_array[x.degree] = None
            x = merge(x, u)
        root_array[x.degree] = u
    roots = list of non-None values from root_array
```

← empty array of size $\lfloor \log_\phi N \rfloor + 1$

for each t in roots:





updated roots:

Suppose we start with x trees, do M merges, and end up with y trees.

$c = O(x + M + \log N) = O(y + 2M + \log N) = O(2M + 2\log N) = O(M + \log N)$

- x: process each tree
- M: do the merge
- $\log N$: update root list
- $y = x - M$, since each merge decreases # trees
- $y \leq \log_\phi N + 1$

$\Delta\Phi = -M$

num. roots decreases every merge

am. cost is

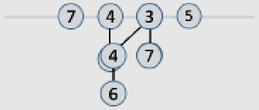
$c + \Delta\Phi = O(M + \log N) - M$

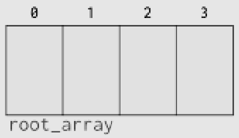
$= O(\log N)$

```
def cleanup(roots):
    root_array = [None, None, ....]
    for each tree t in roots:
        x = t
        while root_array[x.degree] is not None:
            u = root_array[x.degree]
            root_array[x.degree] = None
            x = merge(x, u)
        root_array[x.degree] = u
    roots = list of non-None values from root_array
```

$\Phi = \text{num.roots} + 2 \times \text{num.losers}$ pays in advance for these "uncontrolled" iterations

for each t in roots:





updated roots:

Suppose we start with x trees, do M merges, and end up with y trees.

$c = O(x + M + \log N)$ (process each tree; do the merge; update root list) $= O(y + 2M + \log N)$ ($y = x - M$, since each merge decreases # trees) $= O(2M + 2\log N)$ ($y \le \log_\varphi N + 1$) $= O(M + \log N)$

$\Delta\Phi = -M$ (num. roots decreases every merge)

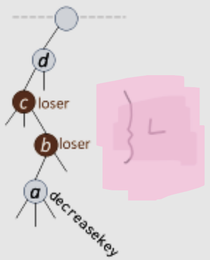
am. cost is $c + \Delta\Phi = O(M + \log N) - M = O(\log N)$

```
def cleanup(roots):
    root_array = [None, None, ....]      ← empty array of size ⌊log_φ N⌋+1
    for each tree t in roots:
        x = t
        while root_array[x.degree] is not None:
            u = root_array[x.degree]
            root_array[x.degree] = None
            x = merge(x, u)
        root_array[x.degree] = u
    roots = list of non-None values from root_array
```

popmin

had to do M merges

```
def decreasekey(v, k'):
    let n be the node where this value is stored
    n.key = k'
    if n violates the heap condition:
        repeat:
            p = n.parent
            remove n from p.children
            insert n into the list of roots, updating minroot if necessary
            n.loser = False
            n = p
        until p.loser == False
        if p is not a root:
            p.loser = True
```



CASE I: no heap violation

$c = O(1)$ $\quad \Delta\Phi = 0$ $\quad \Rightarrow c + \Delta\Phi = O(1)$

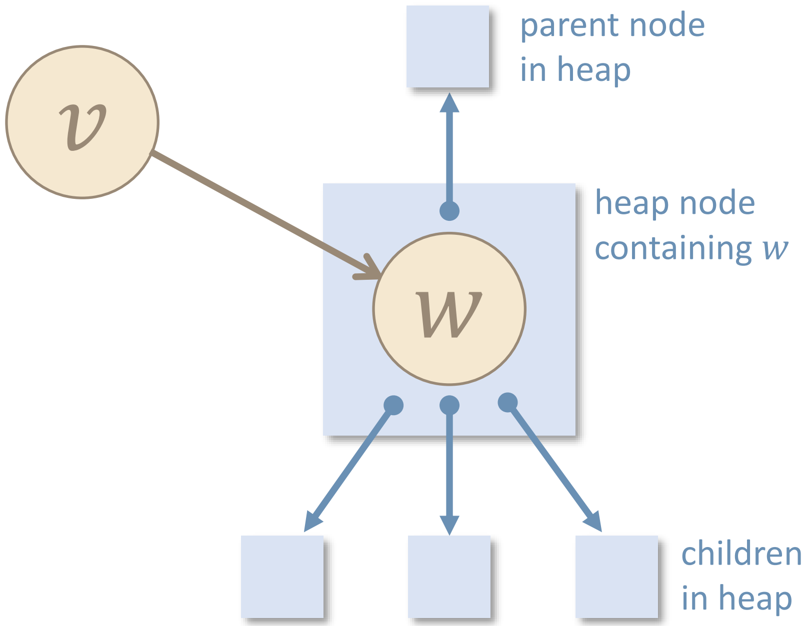
CASE II: heap violation

1. move a to rootlist $c = O(1)$ $\quad \Delta\Phi = 1$ or $\Delta\Phi = -1$ if a was loser $\quad \Rightarrow c + \Delta\Phi = O(1)$
2. Move up L losers also $c = O(L)$ $\quad \Delta\Phi = +L - 2L = -L$ $\quad \Rightarrow c + \Delta\Phi = O(1)$
3. Mark d as a loser $c = O(1)$ $\quad \Delta\Phi = 2$ unless d is root, $\Delta\Phi = 0$ $\quad \Rightarrow c + \Delta\Phi = O(1)$

in both cases, total amortized cost is $O(1)$

decreasekey

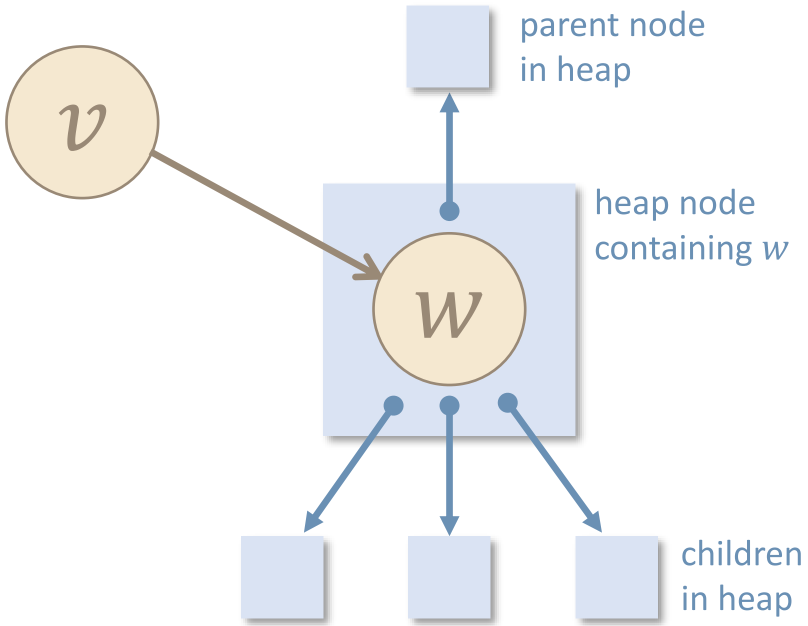
had to move L nodes to root



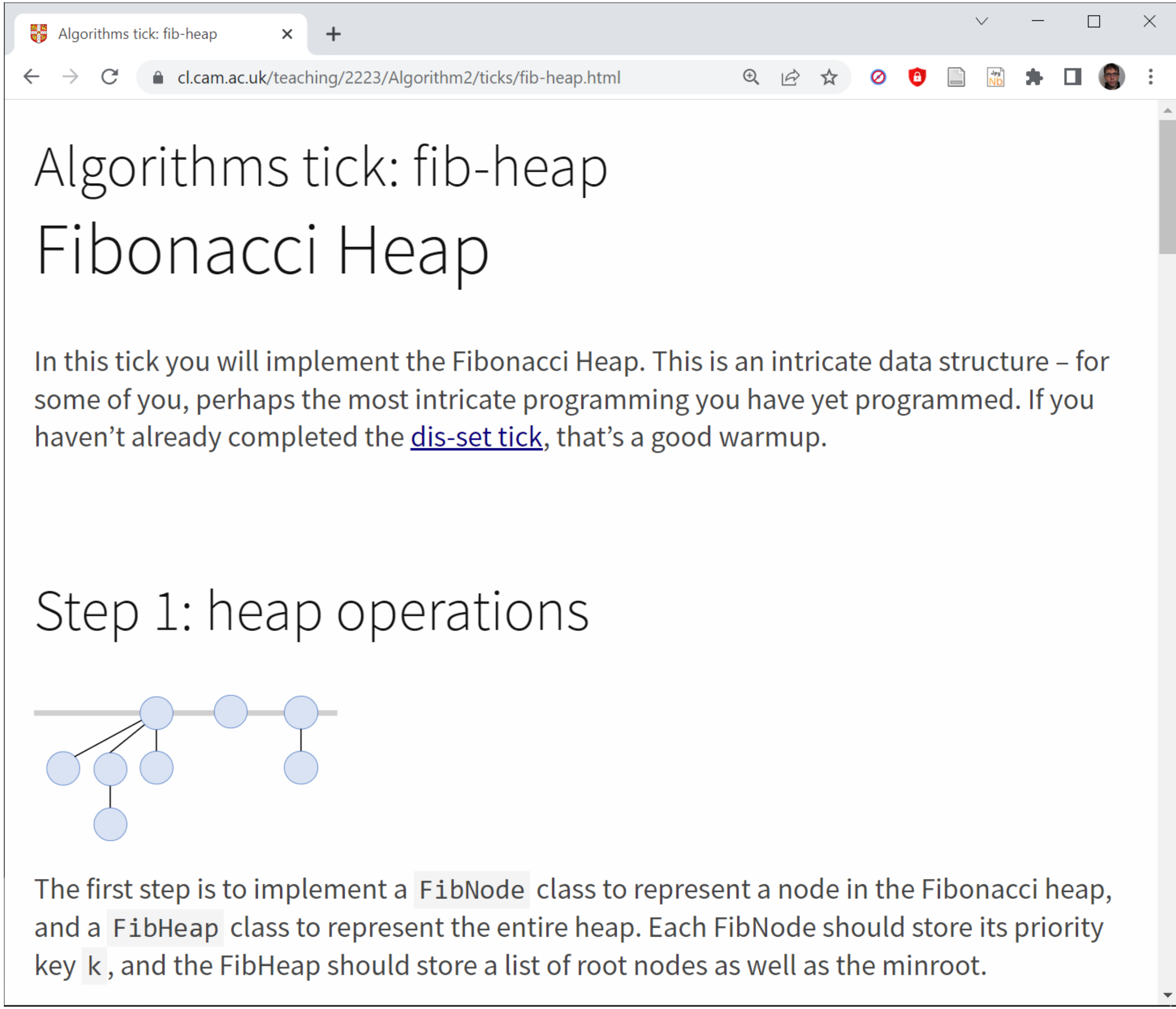

```
def dijkstra(g, s):
    ...
    toexplore = PriorityQueue()
    toexplore.push(s, key=0)

    while not toexplore.is_empty():
        v = toexplore.popmin()
        for (w,edgecost) in v.neighbours:
            dist_w = v.distance + edgecost
            ...
            toexplore.decreasekey(w, key=dist_w)
```

QUESTION. **How can decreasekey be $O(\log N)$?**

Doesn't it take $O(N)$ in the first place, to find the heap node that we want to decrease?




```
def dijkstra(g, s):
    ...
    toexplore = PriorityQueue()
    toexplore.push(s, key=0)

    while not toexplore.is_empty():
        v = toexplore.popmin()
        for (w,edgecost) in v.neighbours:
            dist_w = v.distance + edgecost
            ...
            toexplore.decreasekey(w, key=dist_w)
```


Algorithms tick: fib-heap

cl.cam.ac.uk/teaching/2223/Algorithm2/ticks/fib-heap.html

Algorithms tick: fib-heap

Fibonacci Heap

In this tick you will implement the Fibonacci Heap. This is an intricate data structure – for some of you, perhaps the most intricate programming you have yet programmed. If you haven't already completed the dis-set tick, that's a good warmup.

Step 1: heap operations

The first step is to implement a `FibNode` class to represent a node in the Fibonacci heap, and a `FibHeap` class to represent the entire heap. Each FibNode should store its priority key `k`, and the FibHeap should store a list of root nodes as well as the minroot.