floordrobe, *noun.* A heap of clothing left on the floor of a room. *In computer science:* the most perfect design for an advanced data structure.

Pushing N items is $O(N \log N)$ — but if we're clever we can create a binary heap of N items in $O(N)$.

	popmin	push	decreasekey
binary heap	$O(\log N)$	$O(\log N)$	$O(\log N)$
binomial heap	$O(\log N)$	$\cancel{O(\log N)}$ $O(1)$ amortized	$O(\log N)$

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

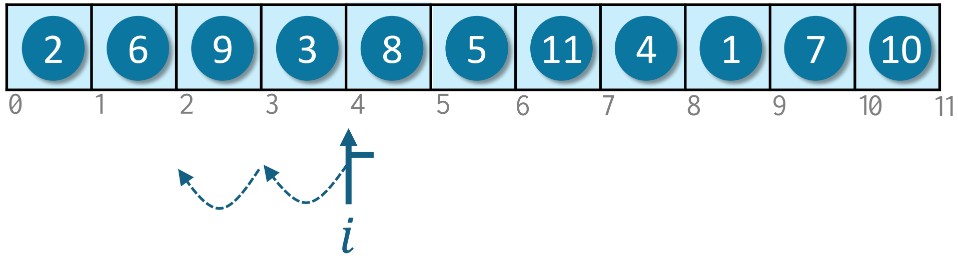
QUESTION1. Can we make both push and `decreasekey` be $O(1)$?

QUESTION2. What's the binomial heap's secret sauce that lets it have $O(1)$ push?

Pushing N items is $O(N \log N)$ — but if we're clever we can create a binary heap of N items in $O(N)$.

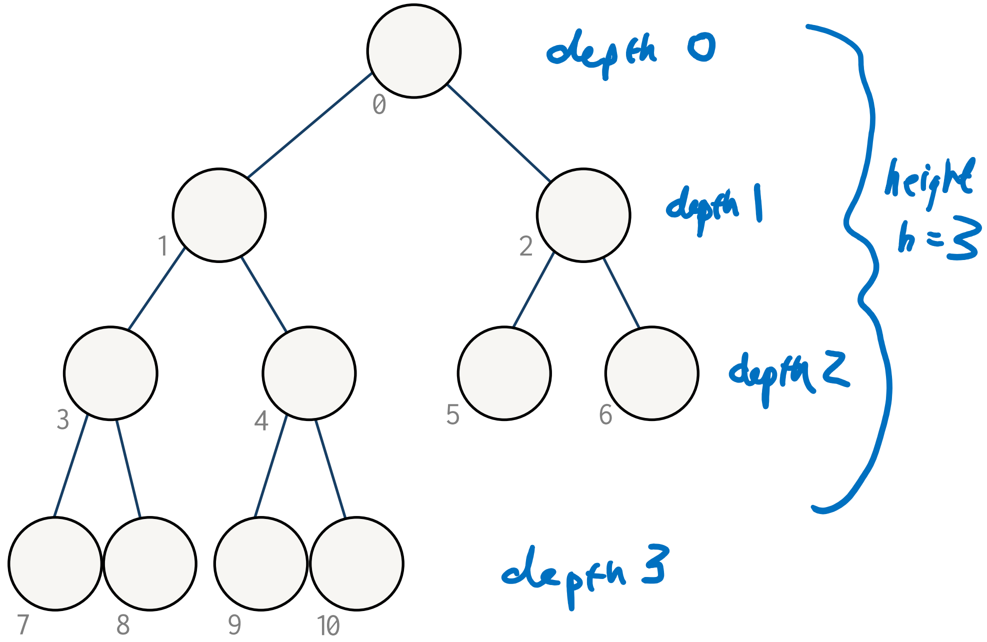
	popmin	push	decreasekey
binary heap	$O(\log N)$	$O(\log N)$	$O(\log N)$
binomial heap	$O(\log N)$	$\cancel{O(\log N)}$ $O(1)$ amortized	$O(\log N)$

```
# Fast binary-heapification
for i in (⌊N/2⌋-1)..0:
  # assert: trees rooted at (i+1)..N are heaps

  re-heapify the tree rooted at x[i]
  by bubbling down
```



- When we reheapify from depth d it takes $h - d$ work to bubble down, and there are $\leq 2^d$ items that need this work.
- There are more items at greater depths, and it's these items that take the least work.
- Total work is $\sum_{d=0}^{h} 2^d (h-d)$

  $\leq 2 \times 2^h = 2N$ [printed notes chapter 2.10]

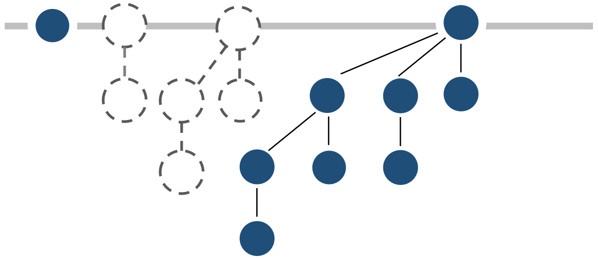


Pushing N items is $O(N \log N)$ — but if we're clever we can create a binary heap of N items in $O(N)$.

	popmin	push	decreasekey
binary heap	$O(\log N)$	$O(\log N)$	$O(\log N)$
binomial heap	$O(\log N)$	$\cancel{O(\log N)}$ $O(1)$ amortized	$O(\log N)$

SECRET SAUCE. **Design your data structure so that most of the time it's sufficient to only touch a small bit of it.**

- The binary heap's fast-heapification achieves this through doing its work in a batch (rather than push by push)
- The binomial heap achieves this by splitting up the heap into semi-isolatable trees



Pushing N items is $O(N \log N)$ — but if we're clever we can create a binary heap of N items in $O(N)$.

	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	$O(\log N)$

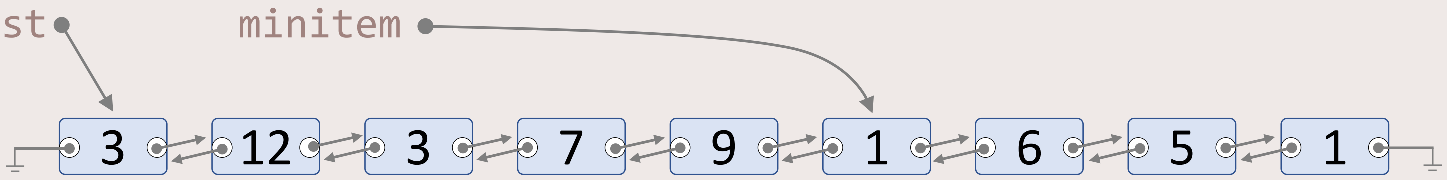
Dijkstra's algorithm makes $O(E)$ calls to push / `decreaskey`, and only $O(V)$ calls to `popmin`.

QUESTION1. Can we make both push and decreasekey be $O(1)$?

~~QUESTION2. What's the binomial heap's secret sauce that lets it have $O(1)$ push?~~

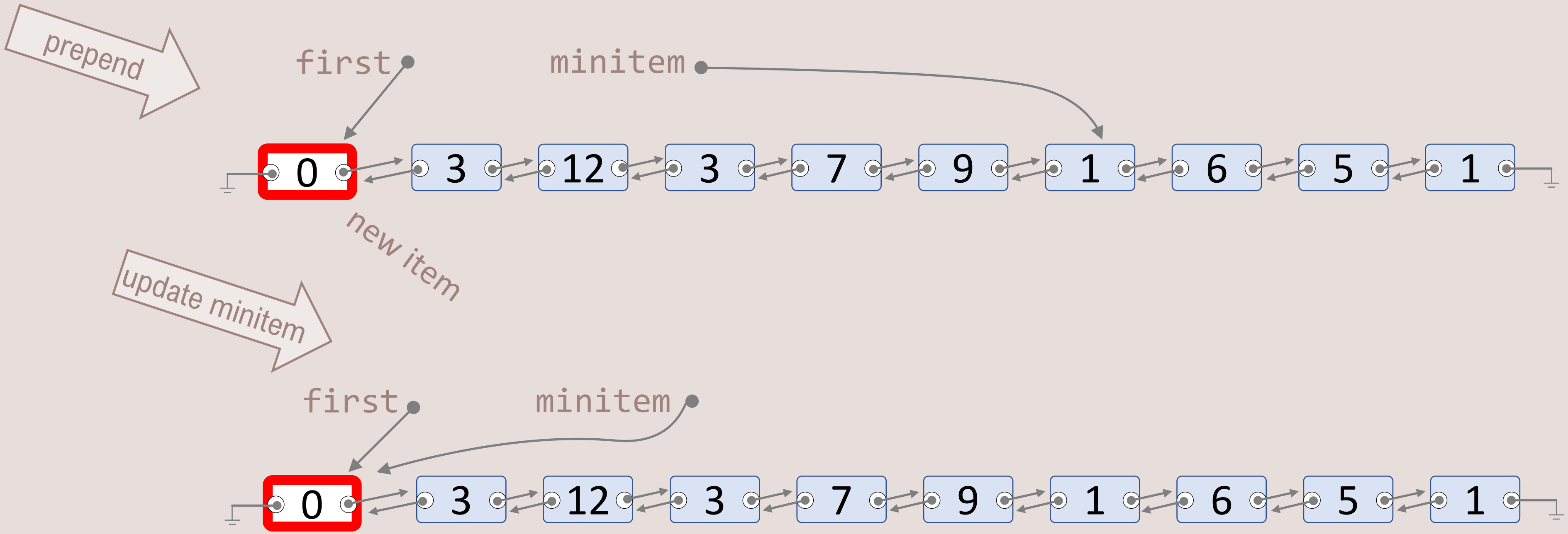
Floordrobe.

Linked-list priority queue

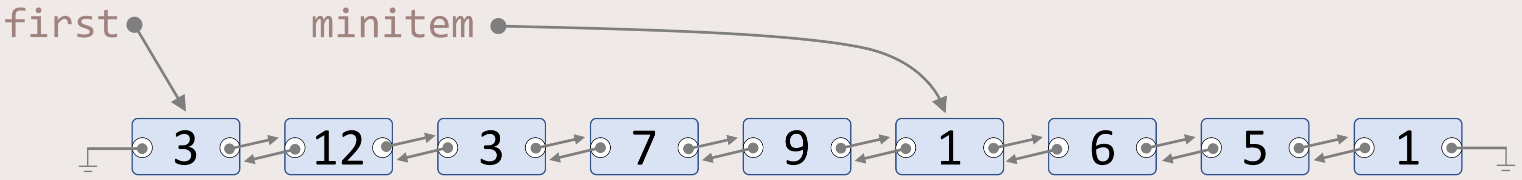


push is O(1)

push(*new item*)

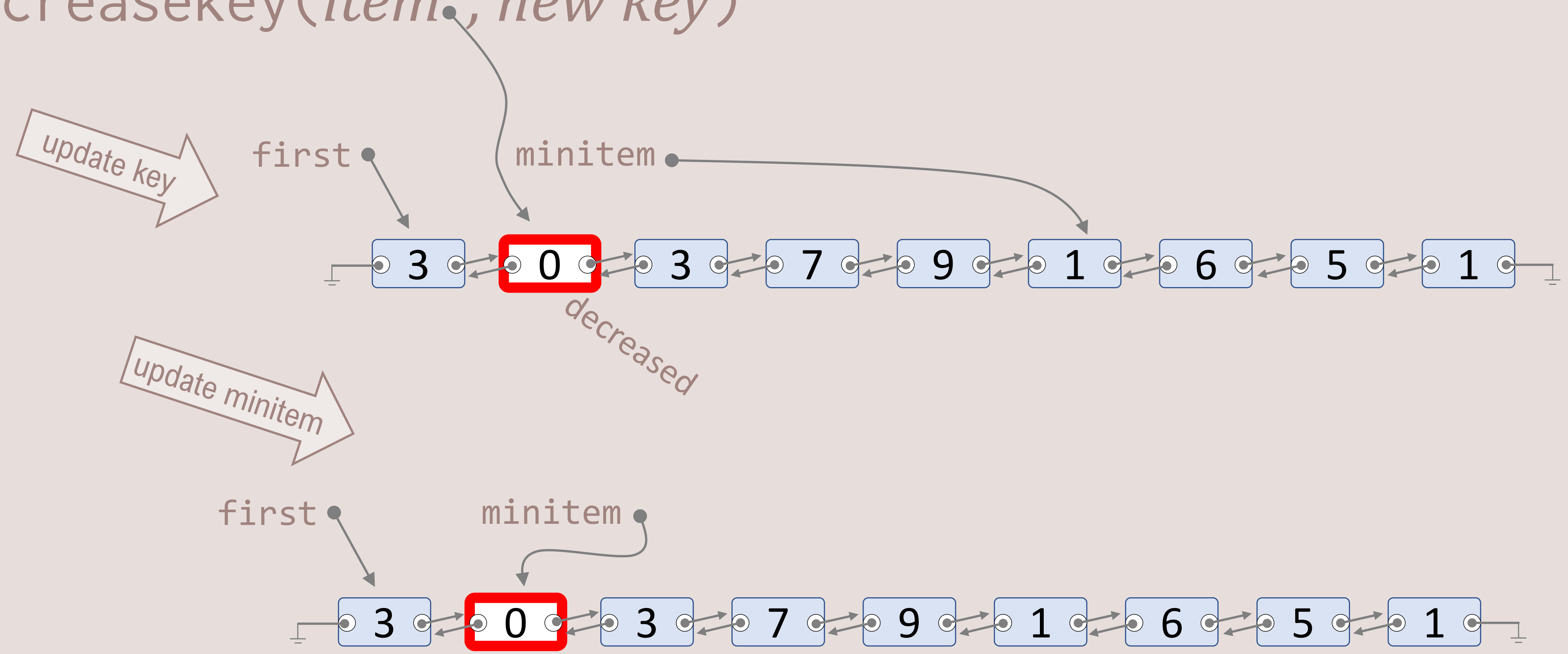


Linked-list priority queue

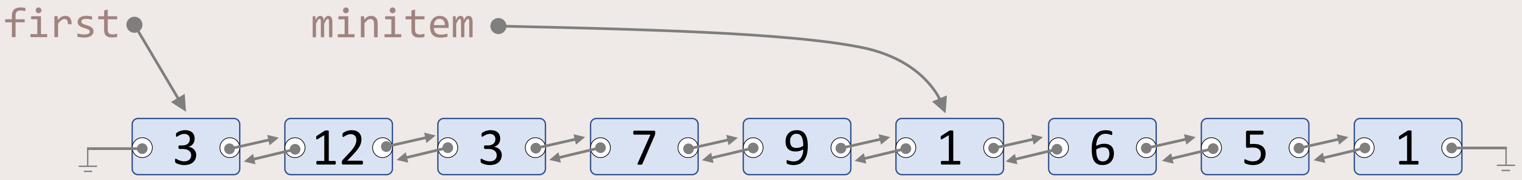


decreasekey is O(1)

decreasekey(*item*, *new key*)

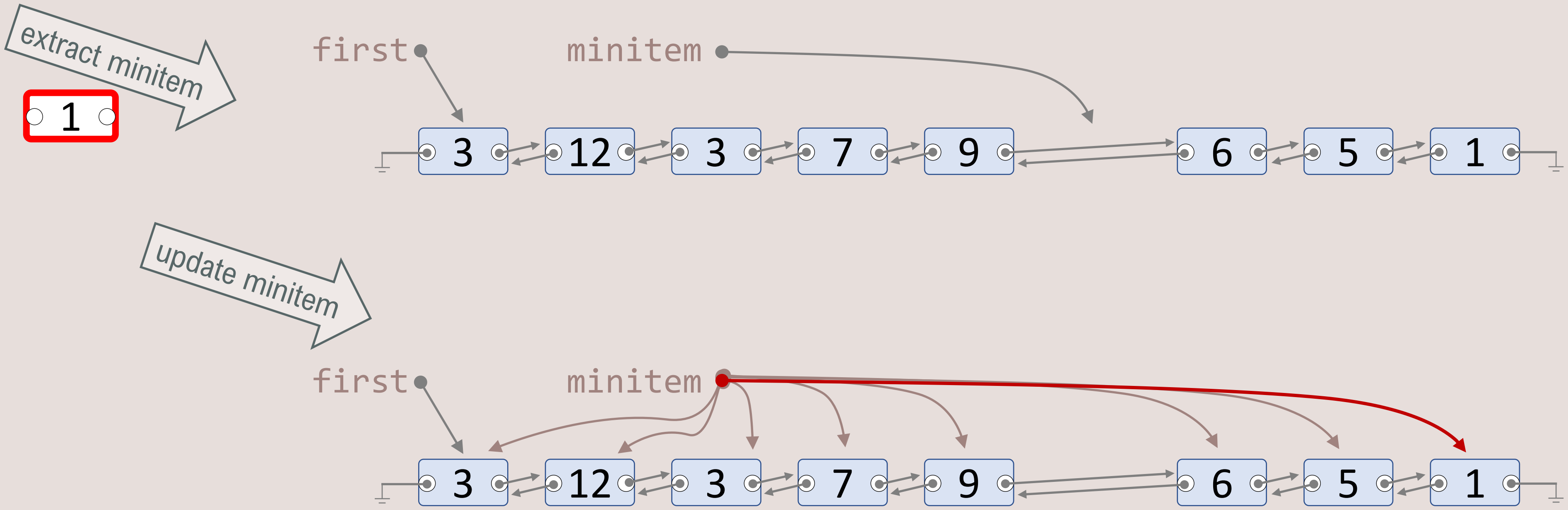


Linked-list priority queue



popmin is O(N)

popmin()



	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

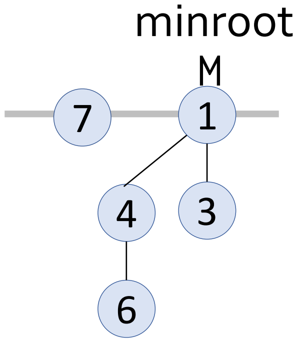
batch-push is $O(N)$

Design strategy for the Fibonacci heap:

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

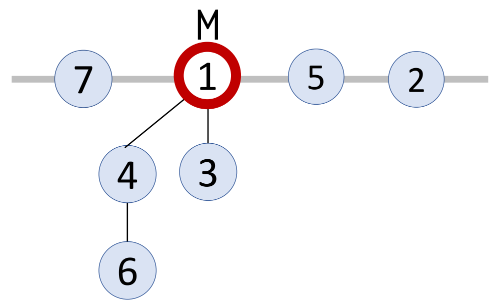
SECTION 7.6

The Fibonacci Heap

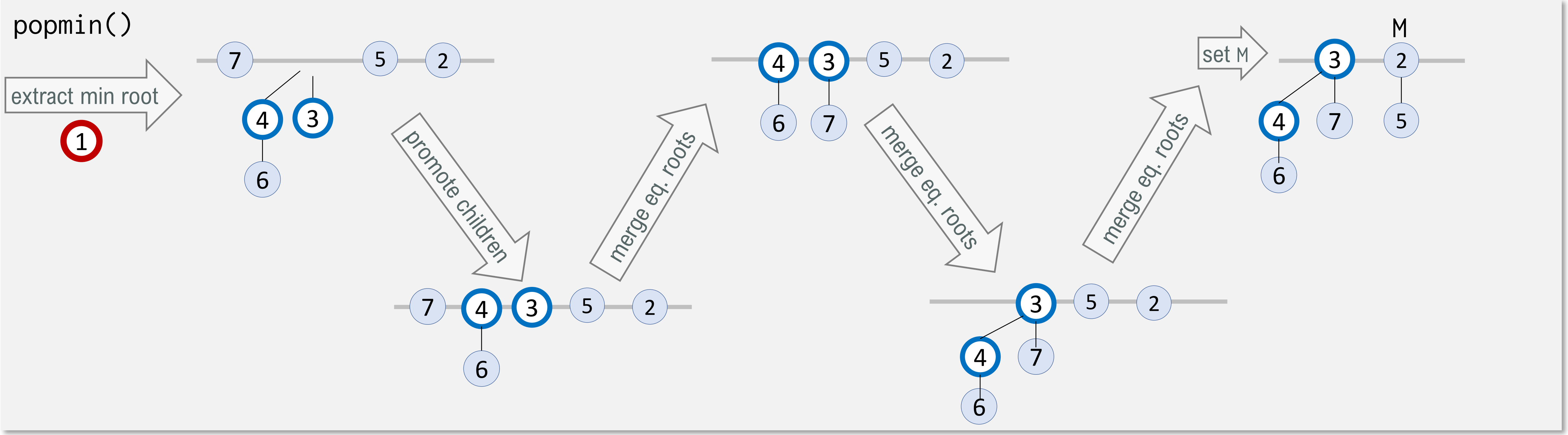


- store a list of trees, each a heap
- trees can have any shape
- keep track of the minroot

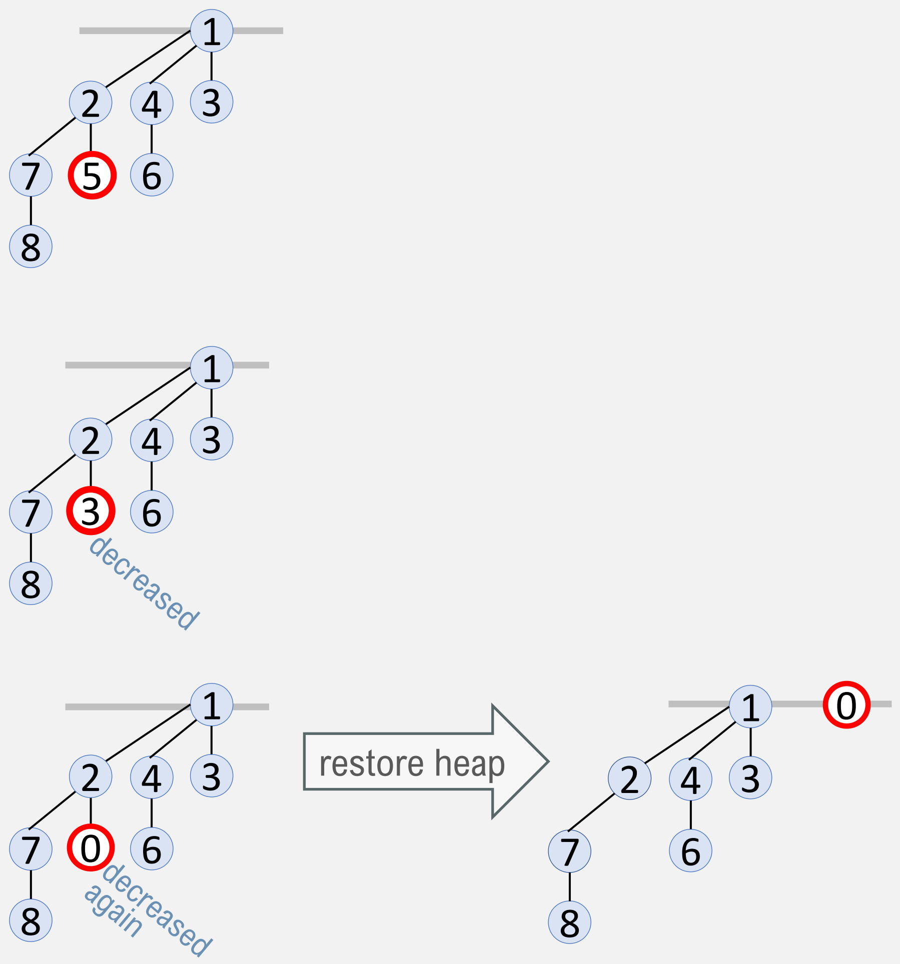
```
# Maintain a list of heaps (i.e. store a pointer to the root of each heap)
roots = []

# 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
```




```
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
```



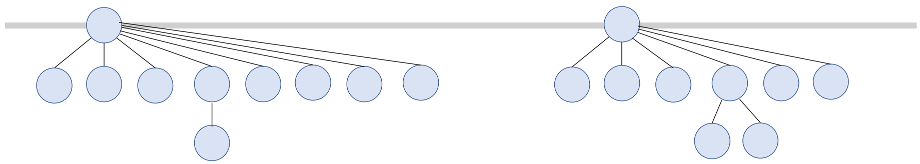
`decreasekey(item, new key)`



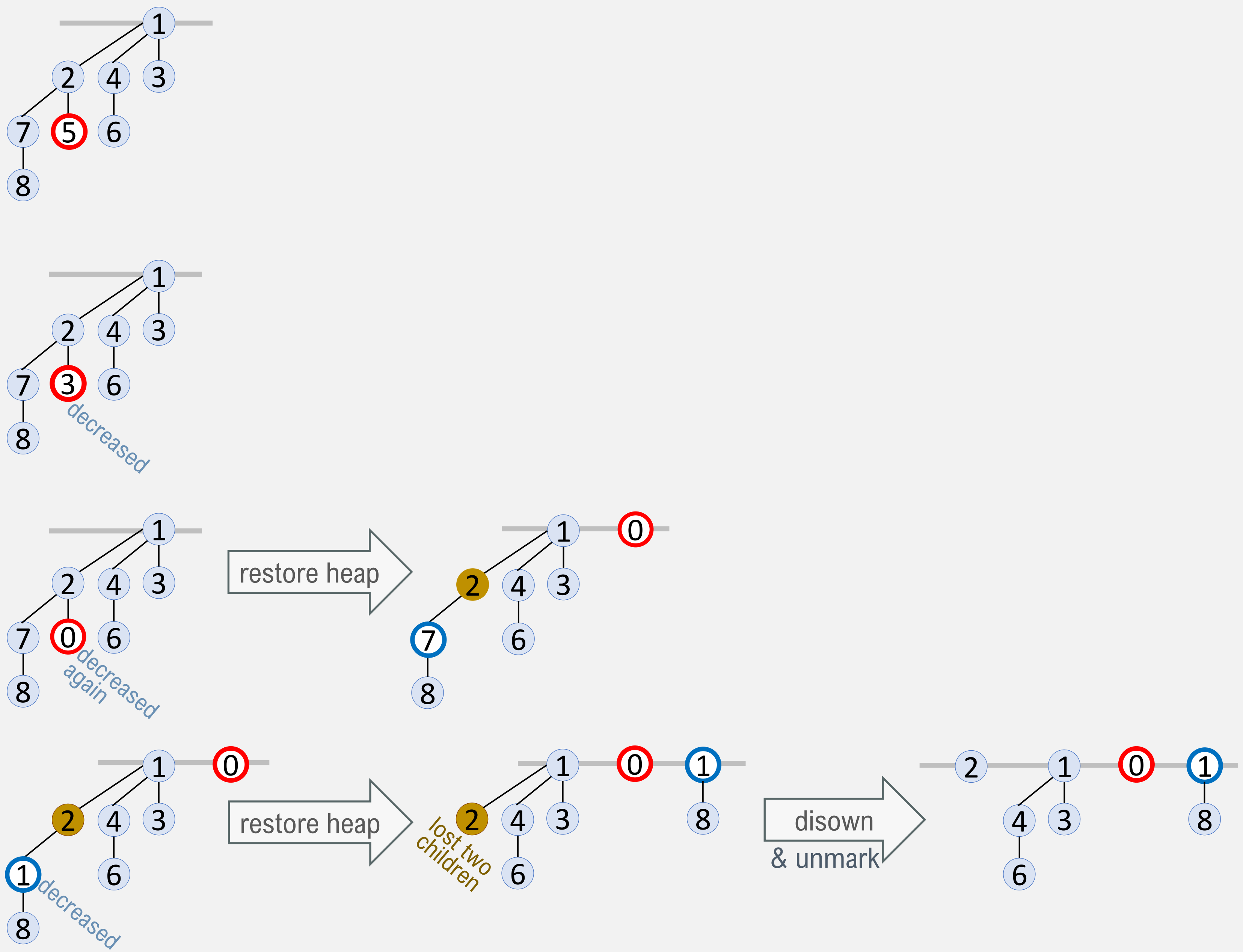
LAZY STRATEGY

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



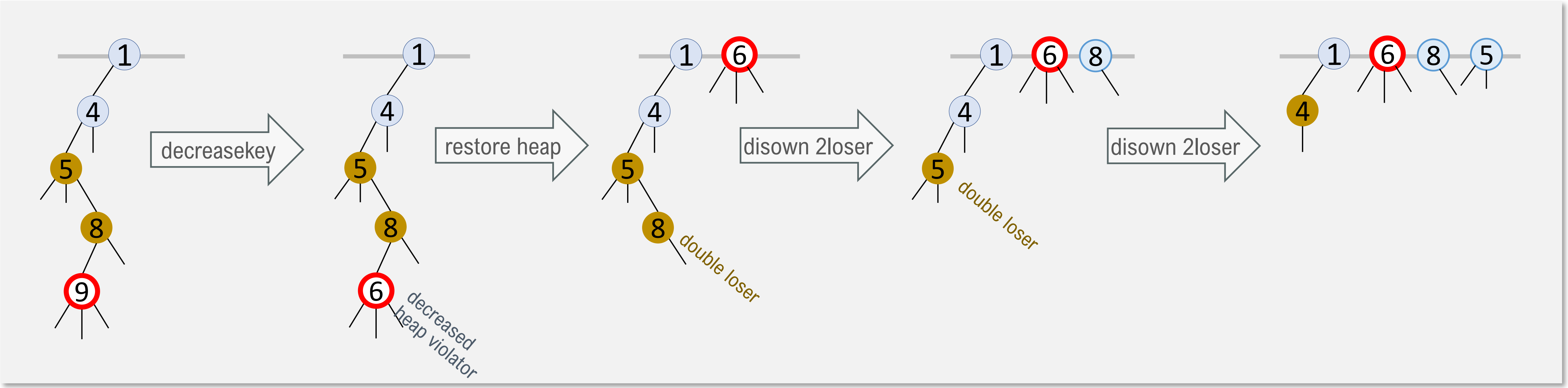
decreasekey(item, new key)



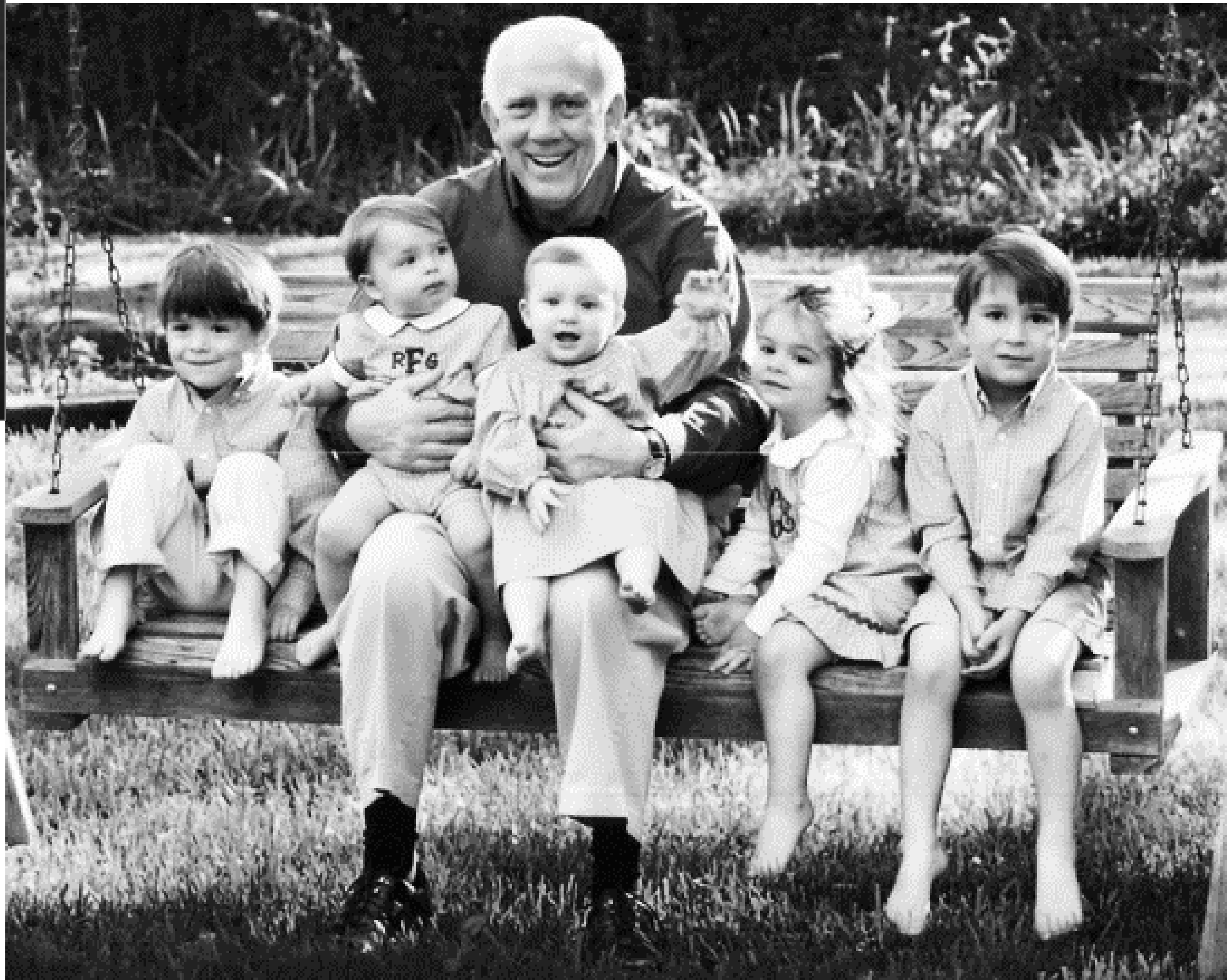
Rule 1. Lose one child, and you're marked a LOSER

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

Take-away: this is an elegant use of potential functions to account for *two separate* unbounded-cost operations.

FIBONACCI HEAP COMPLEXITY ANALYSIS

COMPLEXITY ANALYSIS

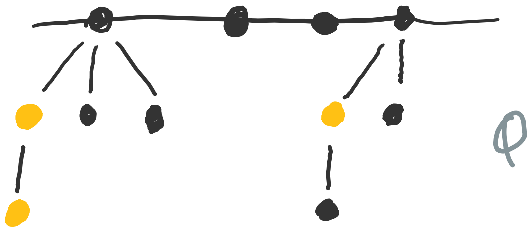
In a Fibonacci heap with N items,
using the potential function

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

- `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 $\leq \log_\phi N$



ϕ is golden ratio
≈ 1.618

BINOMIAL HEAP COMPLEXITY ANALYSIS

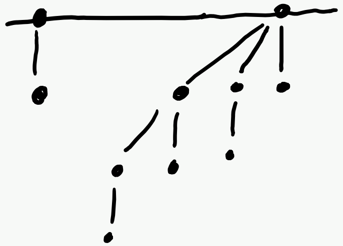
COMPLEXITY ANALYSIS

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 $\leq \log_2 N$



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

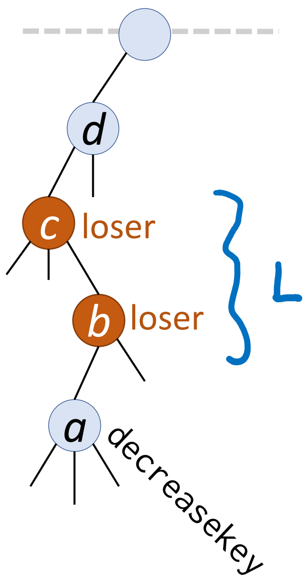

```
def push(Value v, Key k):
    create a new heap h consisting of a single item (v,k)
    add h to the list of roots
    update minroot if minroot is None or k < minroot.key
```

am. cost

$c = O(1)$ $\Delta\Phi = 1$ $c' = c + \Delta\Phi = O(1)$

Φ = num.roots + 2 × 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. $c = O(1)$ $\Delta\Phi = 0$ $c + \Delta\Phi = O(1)$.

Case II: heap violation.

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

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

3. mark d as loser. $c = O(1)$ $\Delta\Phi = 2$ or $\Delta\Phi = 0$ if d was a root $c + \Delta\Phi = O(1)$

tot. is $O(1)$

in each cases, $O(1)$ am. cost.

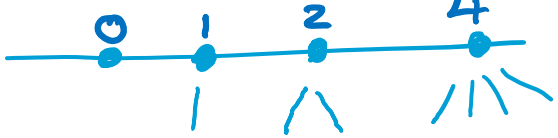
Φ = num.roots + 2 × num.losers

degree $\leq \log_\phi N$


```
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
```

at most one tree of any degree.

after cleanup:



1. cut out minroot, promote its children $c = O(\#\text{children})$ $\Delta\Phi \leq \#\text{children} - 1$ $c + \Delta\Phi = O(\log N)$ by Shape Theorem

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

3. fix minroot. by scanning all roots. $c = O(\log N)$ $\Delta\Phi = 0$ $c + \Delta\Phi = O(\log N)$

largest possible root-degree is $\lfloor \log_\phi N \rfloor$

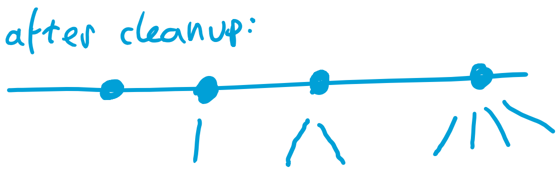
The total for these three steps is $O(\log N)$ amortized cost.

trees after cleanup is $\leq 1 + \lfloor \log_\phi N \rfloor$

num.roots dec. by 1

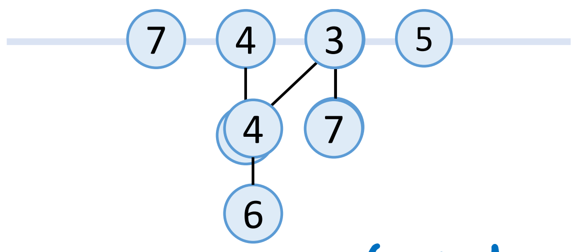
num roots inc. by # children
num. losers decreases, maybe

```
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
```

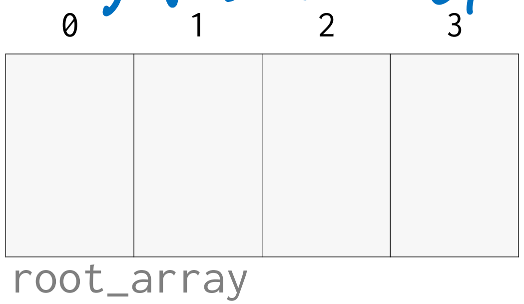



```
20 def cleanup(roots):
21     root_array = [None, None, ....]
22     for each tree t in roots:
23         x = t
24         while root_array[x.degree] is not None:
25             u = root_array[x.degree]
26             root_array[x.degree] = None
27             x = merge(x, u)
28         root_array[x.degree]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

for each t in roots:



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



updated roots:

Suppose I started with x trees, do M merges, end up with y trees, so $y = x - M \Leftrightarrow x = y + M$.

$$c = O(\underbrace{x}_{\text{iteration over roots.}} + \underbrace{M}_{\text{merging}} + \underbrace{\log N}_{\text{copy over}}) \underbrace{=}_{\text{since } x = y+M} O(y + 2M + \log N) \underbrace{=}_{\text{since } y \le \lfloor \log_\phi N \rfloor + 1} O(2M + 2\log N) = O(M + \log N)$$

since constants don't matter for O.

$\Delta\Phi = -M$ since # roots has decreased by M

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

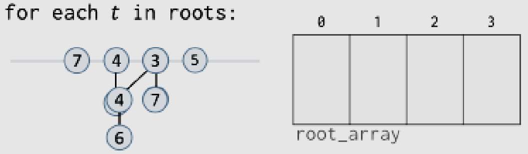
```
def cleanup(roots):
    root_array = [None, None, ....]
    for each tree t in roots:
        x = t
        while root_array[x.degree] is not None:
            u = root_array[x.degree]
            root_array[x.degree] = None
            x = merge(x, u)
        root_array[x.degree] = u
    roots =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

After cleanup, we have at most one tree of each degree; max degree $\le \lfloor \log_\phi N \rfloor$, so we may have up to $1 + \lfloor \log_\phi N \rfloor$ trees.

Φ = num.roots + 2 × 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

$= O(2M + 2\log N)$ — $y \le \log_\phi N + 1$

$= O(M + \log N)$

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

```
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 $\lfloor \log_\phi N \rfloor + 1$

am. cost is

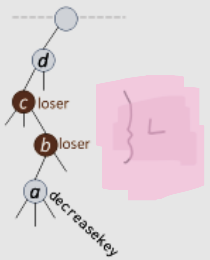
$c + \Delta\Phi = O(M + \log N) - M$

$= O(\log N)$

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

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

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 $O(1)$

decreasekey

had to move L nodes to root