CSCB63 – Design and Analysis of Data Structures

Anya Tafliovich[1]

[1]with huge thanks to Anna Bretscher and Albert Lai

mergeable heaps

Recall the heap data structure:

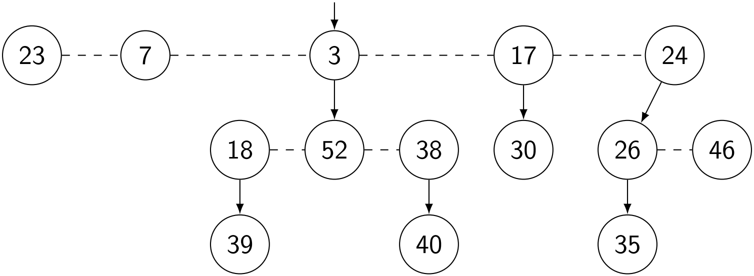
- `insert`(j, p): insert job j with priority p
- `max()` or `min()`: return job with max/min priority
- `extract-max()` or `extract-min()`: remove and return job with max/min priority
- `increase-priority`(j, p'): increase priority of job j to p' (optional)

Does not support:

- `union`(H_1, H_2): merge / union two heaps H_1 and H_2

Fibonacci (min-)heap

- a forest of (min-)heaps:
  - parent priority $\leq$ child priority
  - siblings in circular doubly-linked list; parent points to one arbitrary child
- roots in circular doubly-linked list
- pointer to minimum-priority root



binary heap vs Fibonacci heap

	binary heap worst-case	Fibonacci heap amortised
`insert`	$\Theta(\log n)$	$\Theta(1)$
`extract-min`	$\Theta(\log n)$	$O(\log n)$
`decrease-priority`	$\Theta(\log n)$	$\Theta(1)$
`union`	$\Theta(n)$	$\Theta(1)$

If Prim's algorithm uses a Fibonacci heap:

- if $n = |V|$ and $m = |E|$, then we have
- n calls of `extract-min`: $\mathcal{O}(n \log n)$ total
- and up to m calls of `decrease-priority`: $\mathcal{O}(m)$ total

for a total of: $\mathcal{O}(n \log n + m)$ time

Fibonacci heap: fields

Each node has:

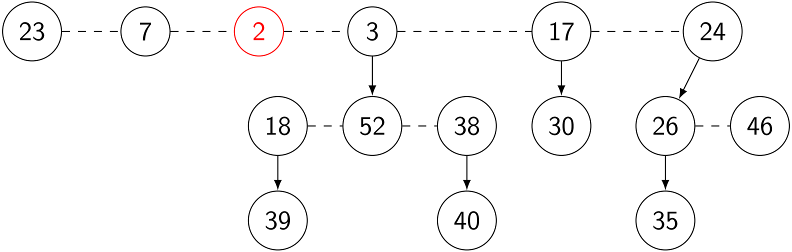
- *key*: priority
- *left*, *right*: for circular list of siblings
- *parent*: pointer to parent
- *child*: pointer to one child
- *degree*: number of children
- *marked*: boolean, important during decrease-priority

The heap has:

- *root_list*: a circular doubly-linked list of roots of the heaps
- *min*: pointer to root node with minimum *key*

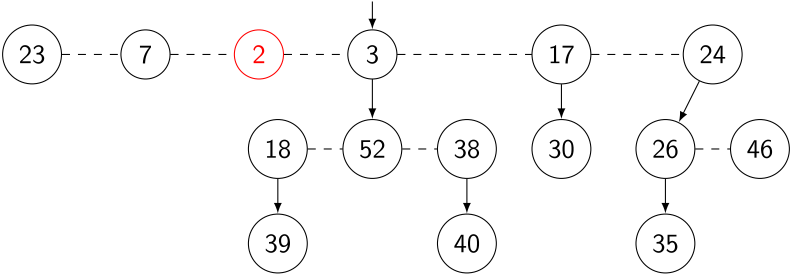
Fibonacci heap: `insert`

```
insert(H, k):
0. new_root := new node(key=k, marked=false)
1. add new_node to H.root_list
2. if k < H.min.key:
3.   H.min = new_root
```



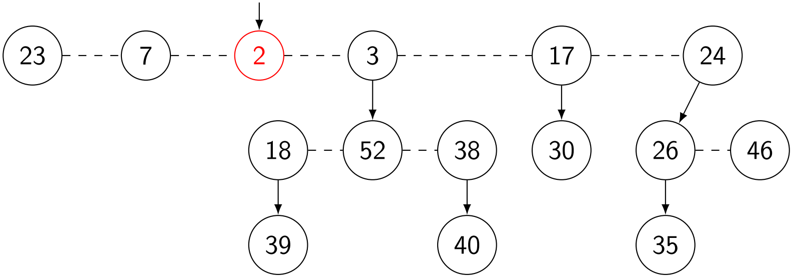
Fibonacci heap: `insert`

```
insert(H, k):
0. new_root := new node(key=k, marked=false)
1. add new_node to H.root_list
2. if k < H.min.key:
3.   H.min = new_root
```



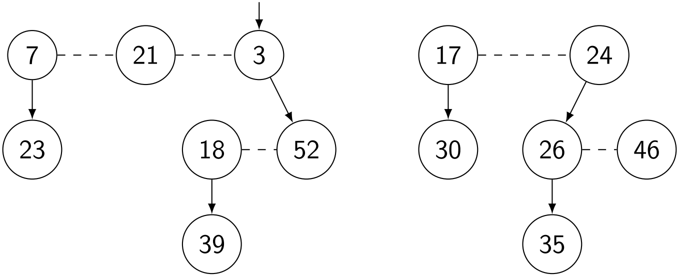
Fibonacci heap: `insert`

```
insert(H, k):
0. new_root := new node(key=k, marked=false)
1. add new_node to H.root_list
2. if k < H.min.key:
3.   H.min = new_root
```



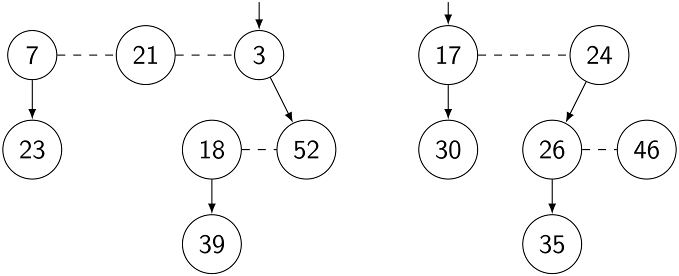
Fibonacci heap: union

```
union(H, H_1, H_2):
0.  H.root_list := H_1.root_list + H_2.root_list
1.  if H_1.min.key <= H_2.min.key:
2.    H.min := H_1.min
3.  else:
4.    H.min := H_2.min
```



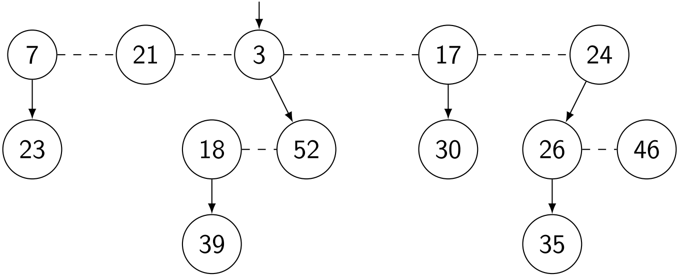
Fibonacci heap: `union`

```
union(H, H_1, H_2):
  H.root_list := H_1.root_list + H_2.root_list
  if H_1.min.key <= H_2.min.key:
    H.min := H_1.min
  else:
    H.min := H_2.min
```



Fibonacci heap: union

```
union(H, H_1, H_2):
0.  H.root_list := H_1.root_list + H_2.root_list
1.  if H_1.min.key <= H_2.min.key:
2.    H.min := H_1.min
3.  else:
4.    H.min := H_2.min
```

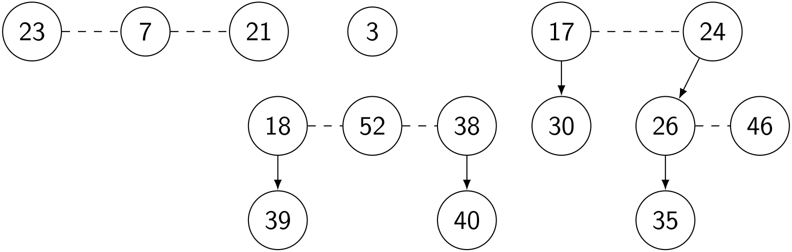


Fibonacci heap: `insert` and `union`

- Complexity of `insert`: $\mathcal{O}(1)$
- Complexity of `union`: $\mathcal{O}(1)$
- "Real work" is in `extract-min` and `decrease-priority`

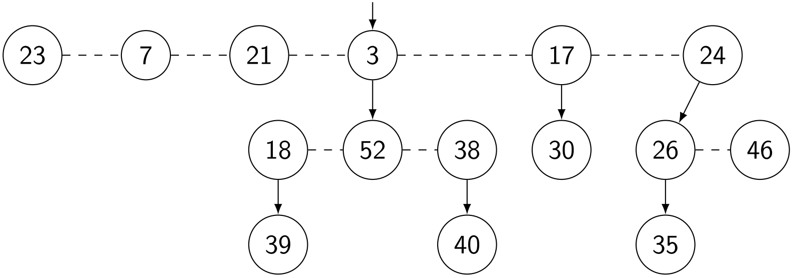
Fibonacci heap: `extract-min`

```
extract-min(H):
0. remove H.min from H.root_list
1. add each child of H.min to H.root_list
2. H.min := any former child of H.min  // can be wrong!
3. consolidate(H)                      // real work here
```



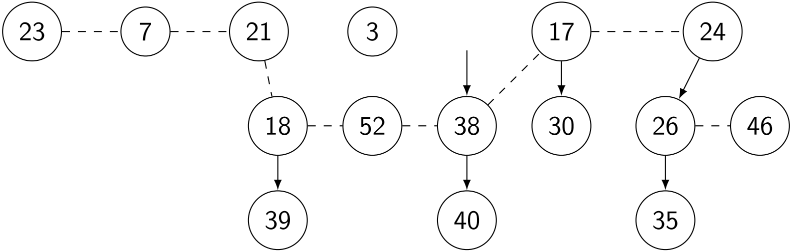
Fibonacci heap: `extract-min`

```
extract-min(H):
0. remove H.min from H.root_list
1. add each child of H.min to H.root_list
2. H.min := any former child of H.min  // can be wrong!
3. consolidate(H)                      // real work here
```



Fibonacci heap: `extract-min`

```
extract-min(H):
0. remove H.min from H.root_list
1. add each child of H.min to H.root_list
2. H.min := any former child of H.min  // can be wrong!
3. consolidate(H)                      // real work here
```



consolidate: idea

Want:

- end with root list with nodes of unique degree

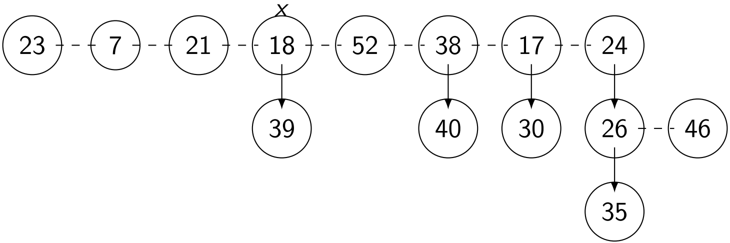
Idea:

- repeat until all nodes in root list have unique degree:
  - walk through root list
  - remember degree of each node so far
  - if see a node x with degree same as that of already seen y,
  - $u := x$ or y, whoever's key is larger
  - $v := x$ or y, whoever's key is smaller
  - add u to children of v
  - remove u from root list
- update *min*

How to remember degrees of nodes?

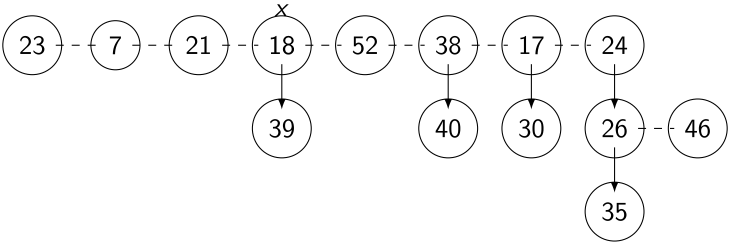
- maintain array A of pointers
- $A[i]$ is root node with degree i

consolidate: example



$A[0] = nil$
$A[1] = nil$
$A[2] = nil$
$A[3] = nil$

`consolidate`: example



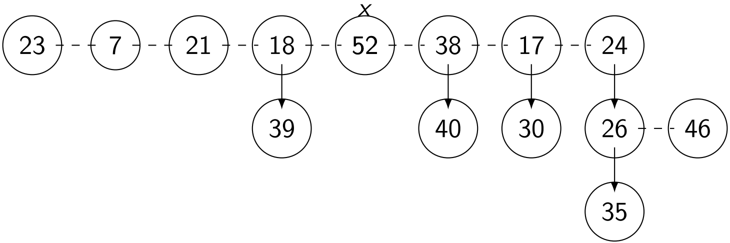
$A[0] = nil$

$A[1] = 18$

$A[2] = nil$

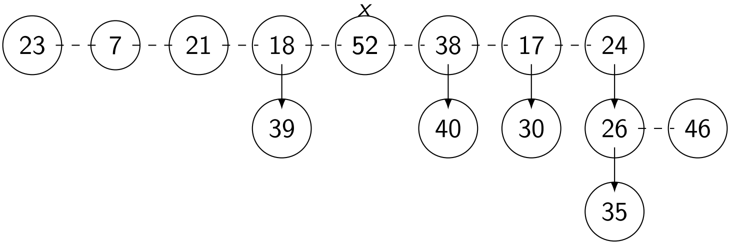
$A[3] = nil$

`consolidate`: example



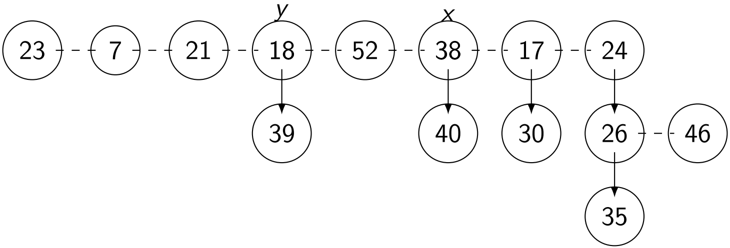
$A[0] = nil$
$A[1] = 18$
$A[2] = nil$
$A[3] = nil$

`consolidate`: example



$A[0] = 52$
$A[1] = 18$
$A[2] = nil$
$A[3] = nil$

`consolidate`: example



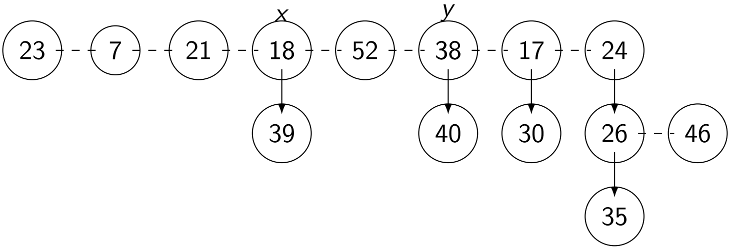
$A[0] = 52$

$A[1] = 18$

$A[2] = nil$

$A[3] = nil$

`consolidate`: example



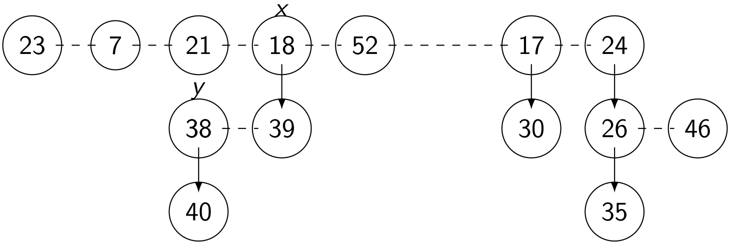
$A[0] =$ 52

$A[1] =$ 18

$A[2] = \mathit{nil}$

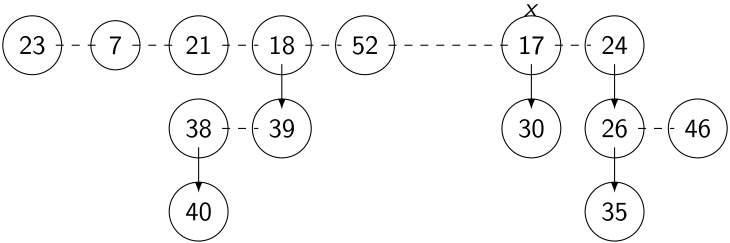
$A[3] = \mathit{nil}$

consolidate: example



$A[0] = 52$
$A[1] = nil$
$A[2] = 18$
$A[3] = nil$

`consolidate`: example



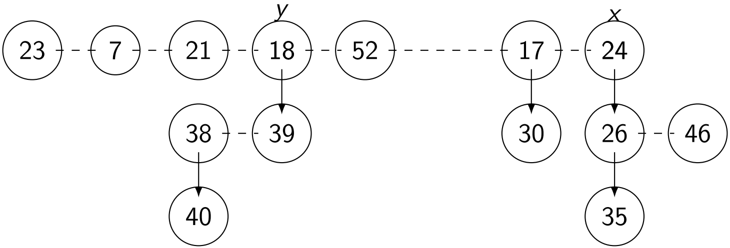
$A[0] =$ 52

$A[1] =$ 17

$A[2] =$ 18

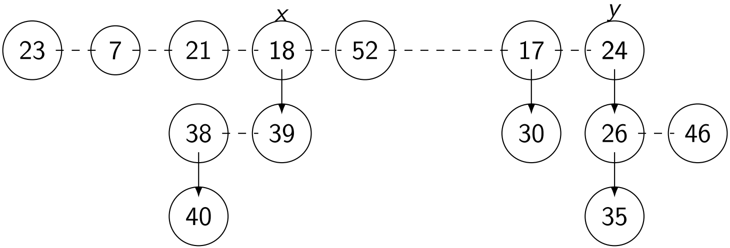
$A[3] = nil$

consolidate: example



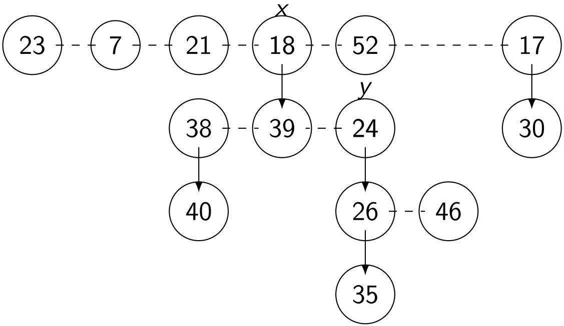
$A[0] = 52$
$A[1] = 17$
$A[2] = 18$
$A[3] = nil$

`consolidate`: example



$A[0] = 52$
$A[1] = 17$
$A[2] = 18$
$A[3] = nil$

`consolidate`: example



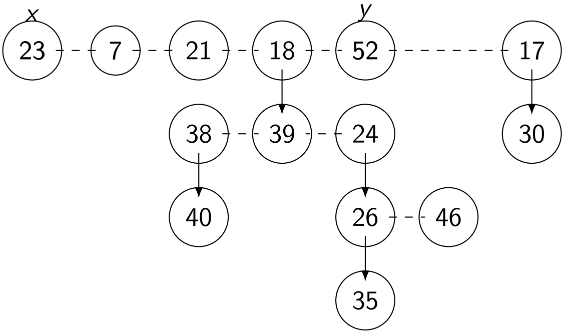
$A[0] = 52$

$A[1] = 17$

$A[2] = nil$

$A[3] = 18$

`consolidate`: example



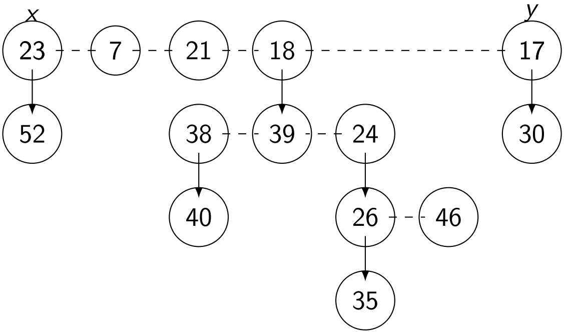
$A[0] =$ 52

$A[1] =$ 17

$A[2] = nil$

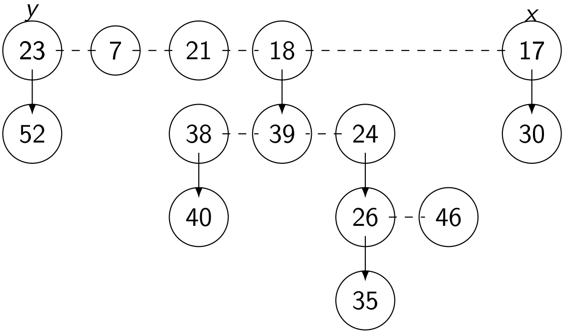
$A[3] =$ 18

`consolidate`: example



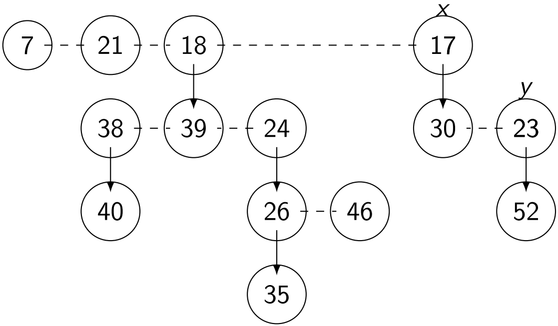
$A[0] = nil$
$A[1] = 17$
$A[2] = nil$
$A[3] = 18$

`consolidate`: example



$A[0] = nil$
$A[1] = 17$
$A[2] = nil$
$A[3] = 18$

consolidate: example



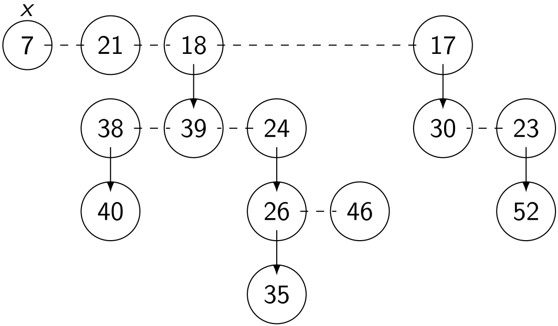
$A[0] = nil$

$A[1] = nil$

$A[2] =$ 17

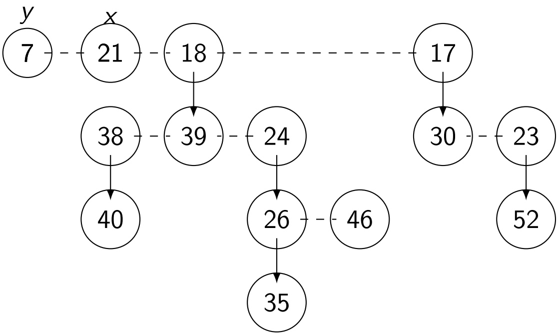
$A[3] =$ 18

consolidate: example



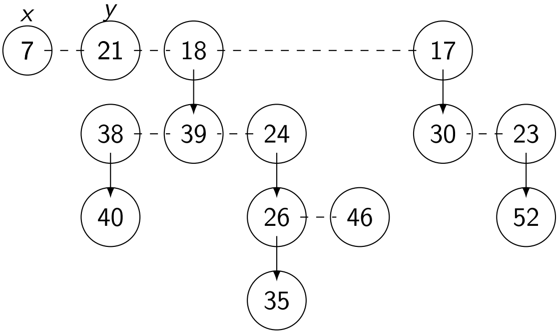
$A[0] = 7$
$A[1] = nil$
$A[2] = 17$
$A[3] = 18$

`consolidate`: example



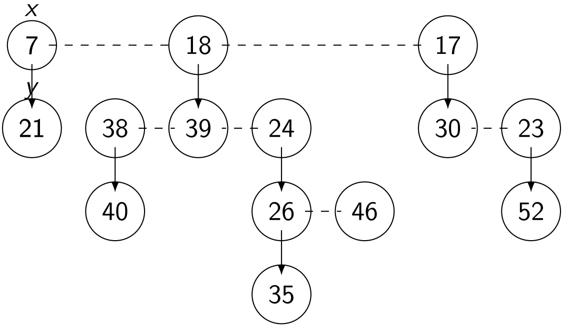
$A[0] =$ 7
$A[1] = nil$
$A[2] =$ 17
$A[3] =$ 18

`consolidate`: example



$A[0] = 7$
$A[1] = nil$
$A[2] = 17$
$A[3] = 18$

`consolidate`: example



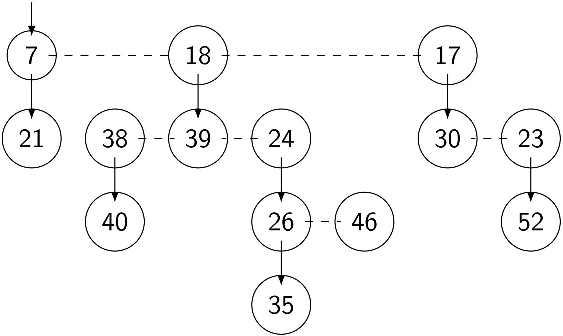
$A[0] = nil$

$A[1] = 7$

$A[2] = 17$

$A[3] = 18$

consolidate: example



$A[0] = nil$

$A[1] = 7$

$A[2] = 17$

$A[3] = 18$

consolidate: algorithm

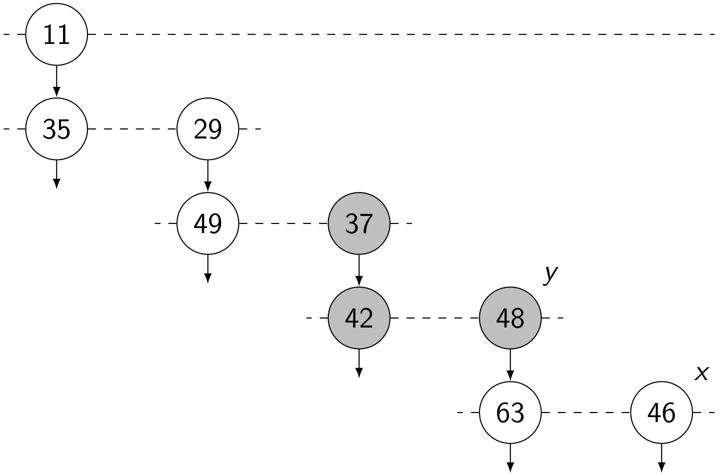
```
consolidate(H):
for each node n in H.root_list:
  x := n
  while A[x.degree] != null:
     y := A[x.degree]
     A[x.degree] := null
     if x.key > y.key:
       x, y := y, x
     remove y from H.root_list
     make y child of x      // x.degree increases
     y.marked := false      // used later
  A[x.degree] := x
  update H.min
```

`decrease-priority`: idea

- this is where we use the `marked` field
- `marked` is *true* if this node lost a child since being removed from root list
- cut child from parent: move child to root list and unmark it
- cascading cuts from some child node:
    - keep going up to root
    - if see an unmarked child, mark it and stop
    - if see a marked child, cut it and keep going

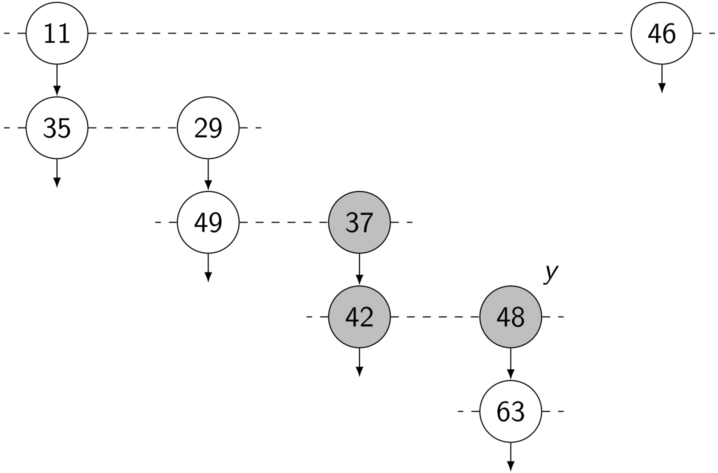
decrease-priority: example

decrease-priority$(x, 46)$. $y.key > x.key$, will promote x.



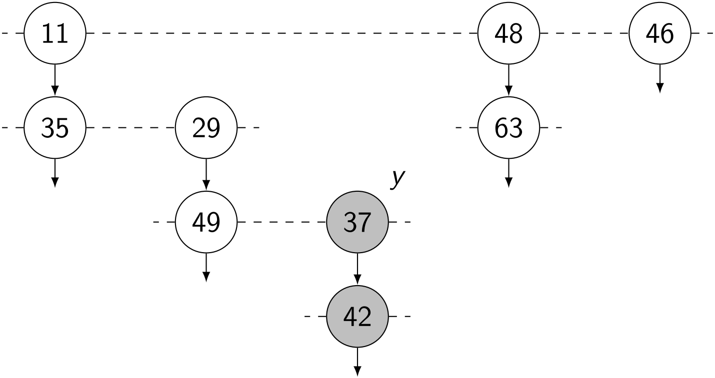
decrease-priority: example

y lost a child while already marked, will be promoted.



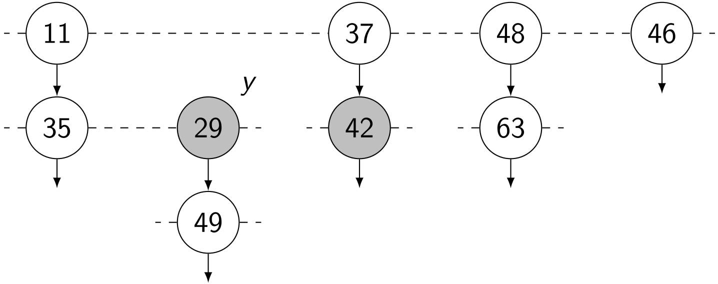
decrease-priority: example

2nd y lost a child while already marked, will be promoted.



decrease-priority: example

3rd y lost a child while unmarked. Mark it now. Exit.



decrease-priority: algorithm

```
decrease-priority(H, x, k):
if k >= x.key: return
x.key := k
y := x.parent
if y != null and y.key > x.key:
  cut(H, x, y)
  while y.parent != null:
    if not y.marked:
      y.marked := true
      break
    else:
      cut(H, y, y.parent)
      y := y.parent
if x.key < H.min.key:
  H.min := x
```

decrease-priority: cut

```
cut(H, x, y):
0. remove x from children of y
1. add x to H.root_list
2. x.marked := false
3. if x.key < H.min.key:
4.   H.min := x
```

complexity of Fibonacci heap operations

- Look at actual worst case time first
- Then define our potential function
- Then find amortised complexity of operations

complexity: actual costs

- define
  - $t(H)$: number of trees in heap (nodes in the root list)
  - $d(H)$: degree of node with maximum degree in heap
- `insert(`j`,` p`)`: $\mathcal{O}(1)$
- `min()`: $\mathcal{O}(1)$
- `extract-min()`:
  - remove node from root list: $\mathcal{O}(1)$
  - insert children into root list: $\mathcal{O}(d(H))$
  - `consolidate(H)`:
    - how many times can a root become a child of another root? once
    - $\therefore$ max number of merges: $\mathcal{O}(t(H))$
    - find new min: $\mathcal{O}(d(H))$
    - total: $\mathcal{O}(t(H) + d(H))$

complexity: actual costs

- define
  - $t(H)$: number of trees in heap (nodes in the root list)
  - $d(H)$: degree of node with maximum degree in heap
  - $m(H)$: number of marked nodes in heap

- `decrease-priority(n, p)`:
  - set new priority of n: $\mathcal{O}(1)$
  - if heap not ordered, cut n: $\mathcal{O}(1)$
  - if cascading cuts: $\mathcal{O}(\#cuts)$
  - only cut <u>marked</u> nodes during cascading cuts: $\#cuts \leq m(H)$
  - so `decrease-priority` is $\mathcal{O}(m(H))$

complexity: observations

Observations AKA potential function magic:

- `extract-min` moves nodes from root list down
- `decrease-priority` cuts nodes / moves them up to root list
- `extract-min`: $\mathcal{O}(t(H) + d(H))$
- `decrease-priority`: $\mathcal{O}(m(H))$
- define potential function:

$$\Phi(H) = t(H) + 2 * m(H)$$

- Initially: $\Phi(H_0) = t(H_0) + 2 * m(H_0) = 0$

complexity: `insert`

- Potential function: $\Phi(H) = t(H) + 2 * m(H)$

- How does `insert` change potential:

$$\begin{aligned}\Delta(\Phi) =& \Phi(H_i) - \Phi(H_{i-1}) \\ =& t(H_i) + 2 * m(H_i) - t(H_{i-1}) - 2 * m(H_{i-1}) \\ =& 1\end{aligned}$$

- Then amortised complexity is:

$$a_i = t_i + \Delta(\Phi) = \mathcal{O}(1)$$

complexity: `decrease-priority`

- Potential function: $\Phi(H) = t(H) + 2 * m(H)$

- How does `decrease-priority` change potential?
- if we make x cuts
- for each cut, a node added to root list:

  $$t(H_i) = t(H_{i-1}) + x$$

- every cut unmarks a marked node
- $x - 1$ or x nodes become unmarked
- at most 1 node becomes marked
- then:
  $$m(H_i) \leq m(H_{i-1}) - (x - 1) + 1 = m(H_{i-1}) - x + 2$$

complexity: `decrease-priority`

Have:

- $\Phi(H) = t(H) + 2 * m(H)$
- $t(H_i) = t(H_{i-1}) + x$
- $m(H_i) \leq m(H_{i-1}) - x + 2$

Then:

$$\begin{aligned}
&\Delta(\Phi) \\
=&\Phi(H_i) - \Phi(H_{i-1}) \\
=&t(H_i) + 2 * m(H_i) - t(H_{i-1}) - 2 * m(H_{i-1}) \\
\leq&t(H_{i-1}) + x + 2 * (m(H_{i-1}) - x + 2) - t(H_{i-1}) - 2 * m(H_{i-1}) \\
=&4 - x
\end{aligned}$$

Amortised cost:

$a_i = t_i + \Phi(H_i) - \Phi(H_{i-1}) = (x + 1) + 4 - x = 5 \in \mathcal{O}(1)$

complexity: `extract-min`

- Potential function: $\Phi(H) = t(H) + 2 * m(H)$

- How does `extract-min` change potential?
- no nodes become marked, some may become unmarked
  - $m(H_i) \leq m(H_{i-1})$
- after `extract-min` (after `consolidate`), all nodes in root list have different degree
- then: $t(H_i) \leq d(H_i) + 1$

Then

$$\begin{aligned}\Delta(\Phi) =& \Phi(H_i) - \Phi(H_{i-1}) \\ =& t(H_i) + 2 * m(H_i) - t(H_{i-1}) - 2 * m(H_{i-1}) \\ \leq& (d(H_i) + 1) + 2 * m(H_i) - t(H_{i-1}) - 2 * m(H_i) \\ =& d(H_i) + 1 - t(H_{i-1})\end{aligned}$$

complexity: `extract-min`

- recall actual time: $t(H_{i-1}) + d(H_i)$
- change in $\Delta(\Phi) \leq d(H_i) + 1 - t(H_{i-1})$
- Then

$$\begin{aligned} a_i =& t_i + \Delta(\Phi) \\ \leq& t(H_{i-1}) + d(H_i) + d(H_i) + 1 - t(H_{i-1}) \\ =& 2 * d(H_i) + 1 \\ \in& \mathcal{O}(d(H_i)) \end{aligned}$$

complexity: `extract-min`

- $a_i \in \mathcal{O}(d(H_i))$
- Last piece of the puzzle: a bound on $d(H)$
- What is the maximum degree of a root node in a heap of size n?
- What is the minimum number of nodes $N(d)$ in a heap with root nodes of degree d?
- ...
- In tutorial show $N(d) = \mathit{fib}(d+2)$ — hence the name "Fibonacci heap"!
  $\therefore n \geq \phi^d$
  $\therefore d \leq \log_\phi n$

Fibonacci heap: complexity

- `insert`: amortised $\mathcal{O}(1)$
- `extract-min`: amortised $\mathcal{O}(\log n)$
- `decrease-priority`: amortised cost $\mathcal{O}(1)$