

CSCB63 – Design and Analysis of Data Structures

Anya Tafliovich¹

¹with huge thanks to Anna Bretscher and Albert Lai

meargeable heaps

Recall the heap data structure:

- $\text{insert}(j, p)$: insert job j with priority p
- $\text{max}()$: return job with max priority
- $\text{extract-max}()$: remove and return job with max priority
- optionally, $\text{increase-priority}(j, p')$: increase priority of job j to p'

meargeable heaps

Recall the heap data structure:

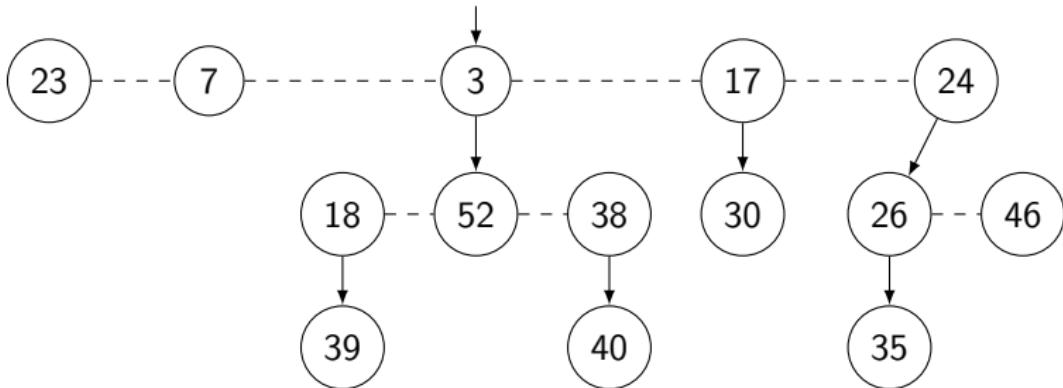
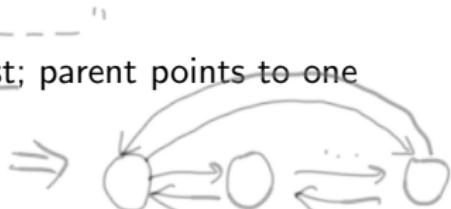
- $\text{insert}(j, p)$: insert job j with priority p
- $\text{max}()$: return job with max priority
- $\text{extract-max}()$: remove and return job with max priority
- optionally, $\text{increase-priority}(j, p')$: increase priority of job j to p'

Does not support:

- $\text{union}(H_1, H_2)$: 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



pointer from parent to one child

binary heap vs Fibonacci heap

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

binary heap vs Fibonacci heap

	binary heap worst-case	Fibonacci heap amortized
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:
- and up to m calls of decrease-priority:

for a total of:

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*

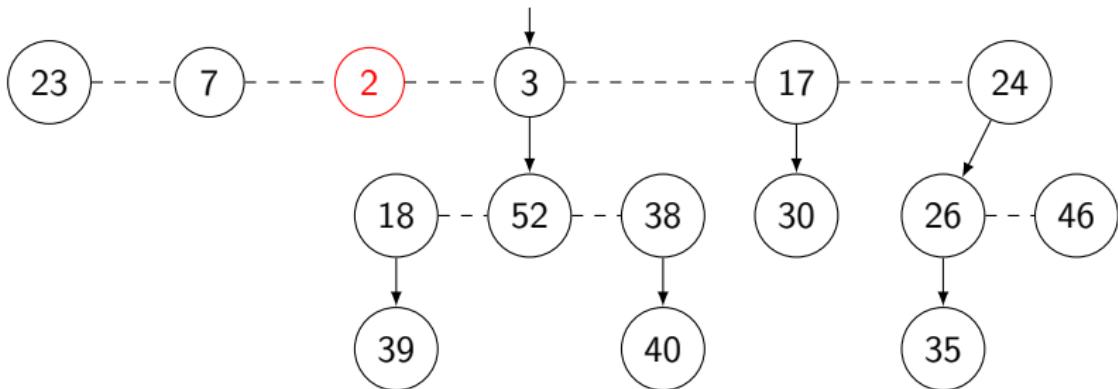
Fibonacci heap: insert

```
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



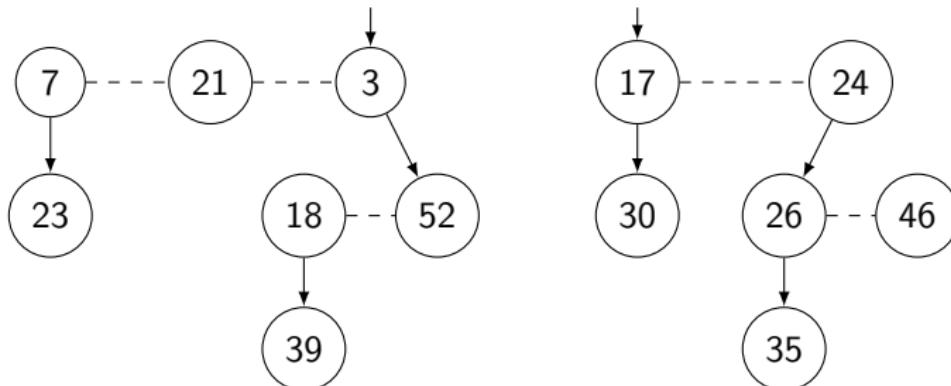
Fibonacci heap: union

```
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):
```

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: $\Theta(1)$
- Complexity of union: $\Theta(1)$
- “Real work” is in extract-min and decrease-priority

Fibonacci heap: extract-min

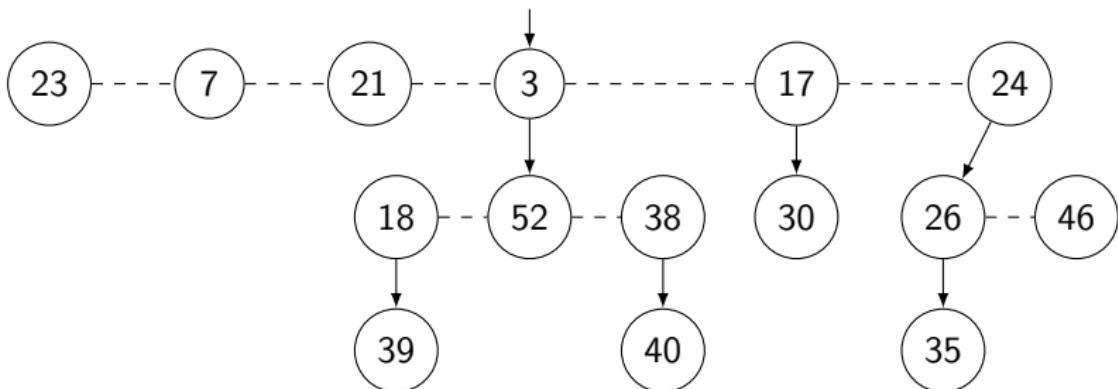
extract-min(H):

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



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
 - $v.\text{child} := u$
 - remove u from root list
- update \min

How to remember degrees of nodes?

consolidate: idea

Want:

- end with root list with nodes of unique degree

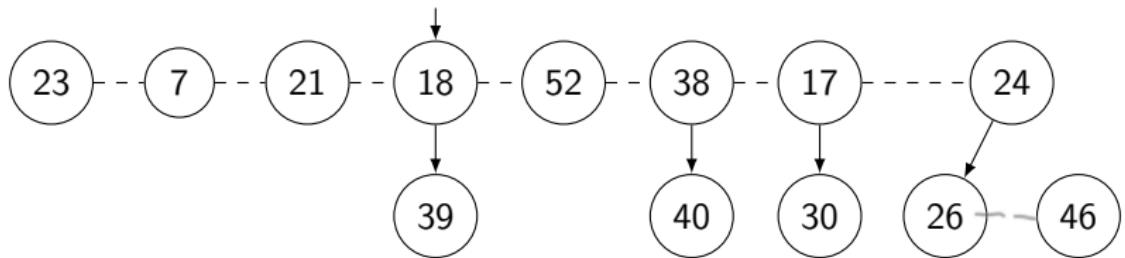
Idea:

- 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
 - $v.child := u$
 - 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



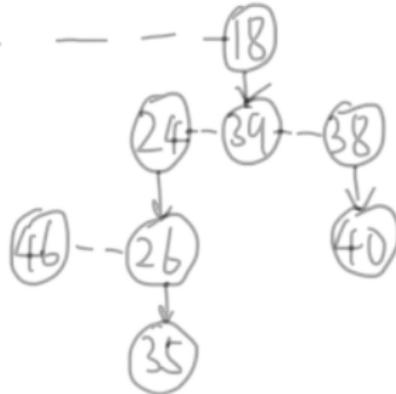
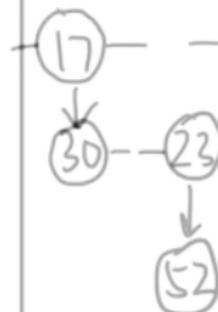
$A[0]$

$A[1]$

$A[2]$

$A[3]$

$A[0] = \text{nil}$
 $A[1] = \text{nil}$
 $A[2] = \text{nil}$
 $A[3] = \text{nil}$
 $A[4] = \text{nil}$



consolidate: algorithm

```
consolidate(H):
```

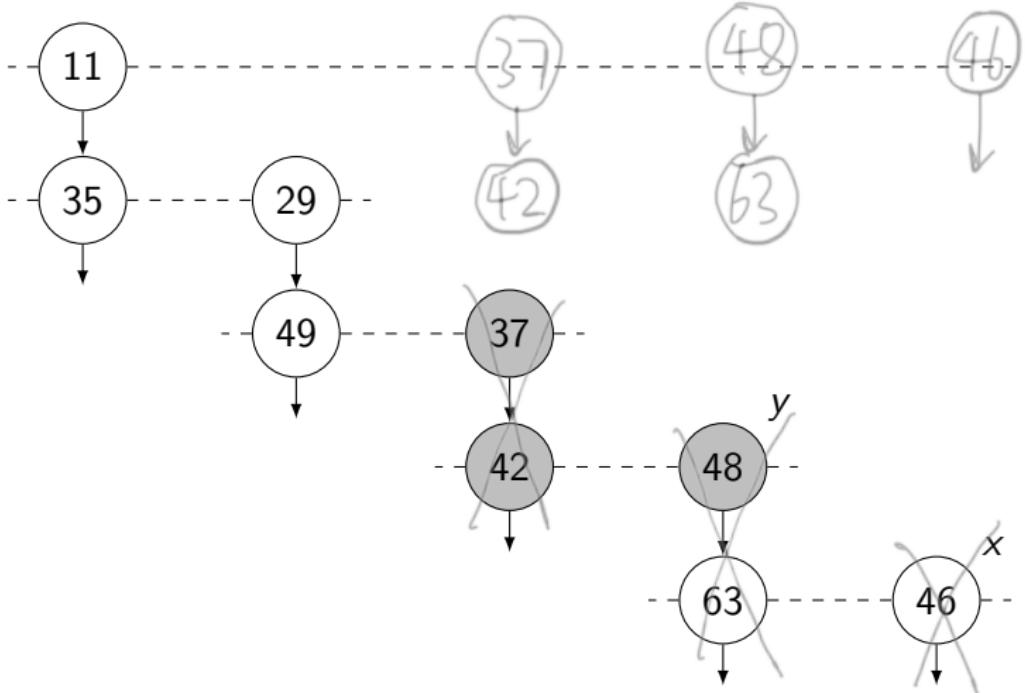
0. for each node n in H.root_list:
1. x := n
2. while A[x.degree] != null:
3. y := A[x.degree]
4. A[x.degree] := null
5. if x.key > y.key:
6. x, y := y, x
7. remove y from H.root_list
8. make y child of x // x.degree increases
9. y.marked := false // used later
10. A[x.degree] := x
11. update H.min

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: algorithm

```
decrease-priority(H, x, k):
0. if k >= x.key: return
1. x.key := k
2. y := x.parent
3. if y != null and y.key > x.key:
4.     cut(H, x, y)
5.     while y.parent != null:
6.         if not y.marked:
7.             y.marked := true
8.             break
9.         else:
10.            cut(H, y, y.parent)
11.            y := y.parent
12. if x.key < H.min.key:
13.     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 amortized 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

- $\text{insert}(j, p)$: $O(1)$

- $\text{max}()$: $O(1)$

- $\text{extract-min}()$:

- remove node from root list: $O(1)$

- insert children into root list: $O(D(H))$

- consolidate(H):

- how many times can a root become a child of another root?

- \therefore max number of merges:

- find new min: (update) $O(D(H))$

- total:

$$O(t(H) + D(H))$$

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 : $O(1)$
- if heap not ordered, cut n : $O(1)$
- if cascading cuts:
 $O(\# \text{cuts})$
- only cut marked nodes during cascading cuts: $\# \text{cuts} \leq m(H)$
- so decrease-priority is $O(m(H))$

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: $O(t(H) + D(H))$
- decrease-priority: $O(m(H))$
- define potential function:

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

complexity: insert

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

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

- How does insert change potential:

$$\Delta \Phi = \Phi(H_i) - \Phi(H_{i-1}) = t(H_i) + 2m(H_i) - t(H_{i-1}) - 2m(H_{i-1})$$

$= 1$

increase 1 tree

↓

↖ did not mark nodes

- Then amortized complexity is:

$$a_i = t_i + \Delta \Phi$$

$$= 1 + 1 = 2 \in O(1)$$

$a_i = t_i + \Phi(H)$, where $t_i = x+1$

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

- at least $x - 1$ nodes become unmarked

- at most 1 node becomes marked

- then:

mark the parent

$$m(H_i) - m(H_{i-1}) \leq -\min\{x-1, x\} + \max\{0, 1\}$$

$$\leq -(x-1) + 1 = -x + 2$$

$$(-x+1+0 / -x+1+1 / -x+0 / -x+1)$$

decrease-priority node
unmarked before

$x-1/x$ unmarked

0/1 marked

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:

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

Amortized cost:

$$a_i = t_i + \Delta\Phi = x + 1 - x + 4 = 5 \in O(1)$$

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 $0, 1, 2, \dots, d(H_i), d(H_i)+1$
- then: $t(H_i) \leq d(H_i) + 1$

a child of the
original min-node
may be
marked before

Then

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

complexity: extract-min

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

$$d_i = t_i + \Delta\phi$$

$$\begin{aligned} & \text{\# of trees before extract-min} \\ & + \text{\# of child trees in original min-node} \\ & = d(H_i) \end{aligned}$$

$$\leq t(H_{i-1}) + d(H_i) + d(H_i) + 1 - t(H_{i-1})$$

$$= 2d(H_i) + 1 \in O(d(H_i))$$

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 $N(d) = fib(d + 2)$ — hence the name “Fibonacci heap”!
$$\therefore n \geq \phi^d$$
$$\therefore d \leq \log_\phi n$$
$$= fib(d) + fib(d+1)$$

Fibonacci heap: complexity

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