Overview

- Heap Operations
- d-Heaps
- Leftist Heaps
- Binomial Heaps
- Fibonacci Heaps

Heaps

A *heap* is an abstract data structure consisting of *items*, each with an associated *key*.

Following operations must be available on a heap:

- `makeheap`(S): create and return a new heap whose items are elements of the set S.
- `insert`(i, H): insert item i into heap H.
- `deletemin`(H): delete from the heap H and return the item with the smallest key (`nil` if H is empty).

In addition to these three operations, some of the following operations can be available.

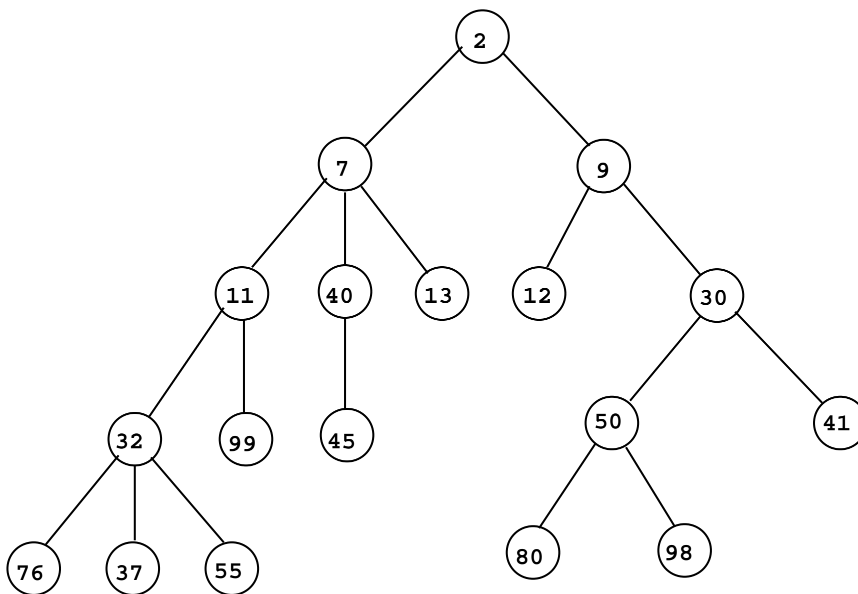
- `findmin`(H): ruturn from heap H the item with the smallest key.
- `delete`(i, H): delete item i from heap H.
- `meld`$(H1, H2)$: return the heap formed by combining distinct heaps $H1$ and $H2$. Heaps $H1$ and $H2$ are destroyed.
- `heapify`(Q): return a heap formed by melding all heaps in the list Q.

Direct access to items in heaps is assumed.

Heap-Ordered Trees

Heaps will be represented by *heap-ordered rooted trees*

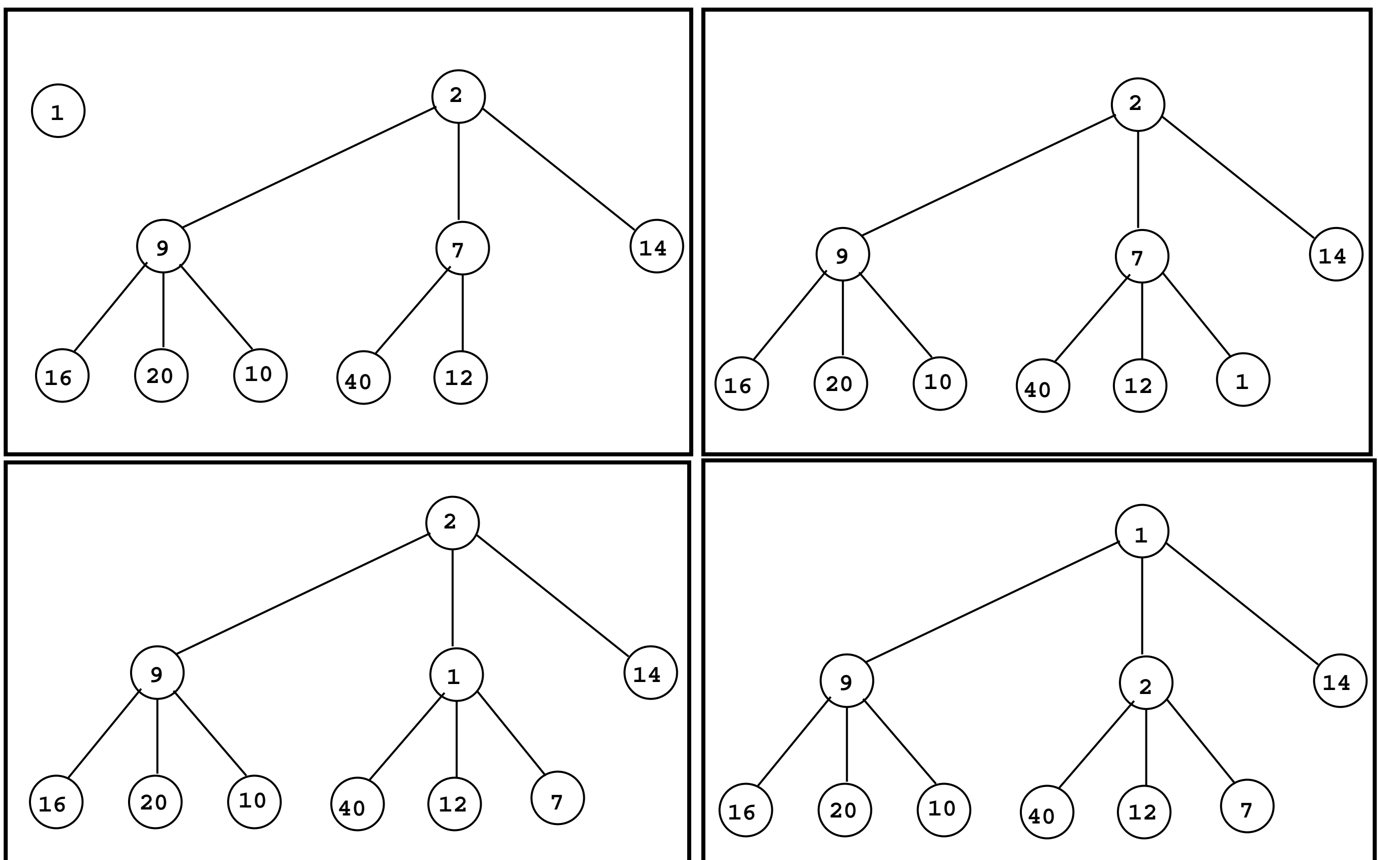
- Each node in the tree contains one item.
- Items are arranged in the heap order: key of a parent is not greater than keys of its sons.
- Otherwise heap-ordered trees can have any structure.
- Heap-ordered trees can be empty.



- *d-heaps*: trees with up to d sons for each node; especially useful when melding is insignificant.
- *leftist heaps*: binary trees with rightmost paths kept short; especially useful when melding is important.

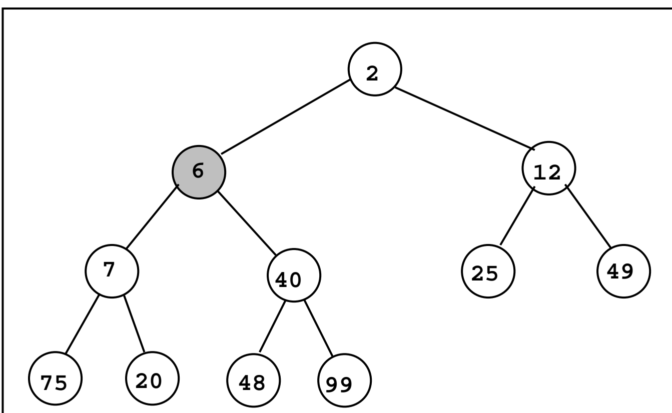
Insertion in Heap-Ordered Trees

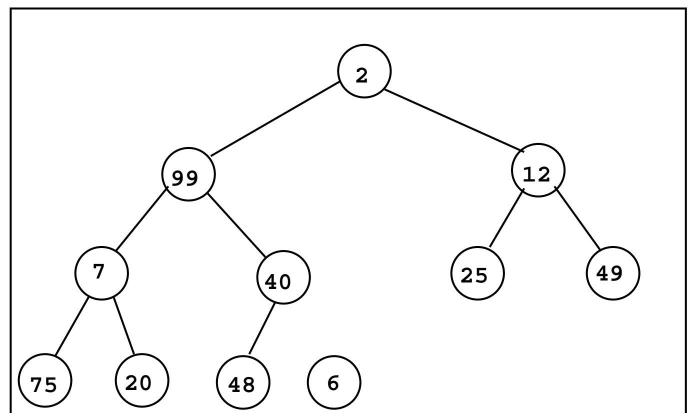
- add new node x as a son of a node y in H. We will later discuss how y is selected.
- if `key(x)` $\geq$ `key(p(x))` then **STOP**.
- if `key(x)` $<$ `key(p(x))` then *sift-up*: swap the pointers to the two items.
- repeat this process as long as possible or until the root is reached.

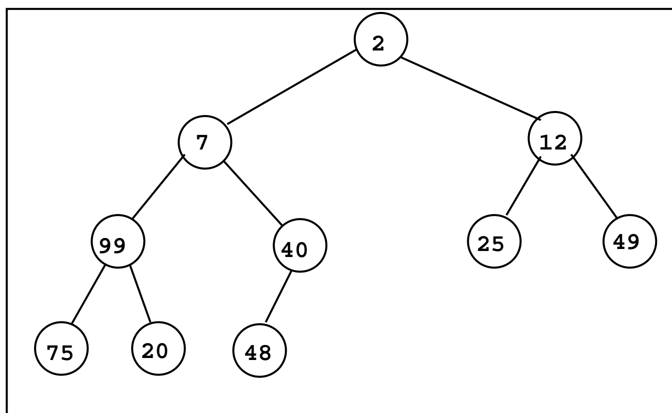


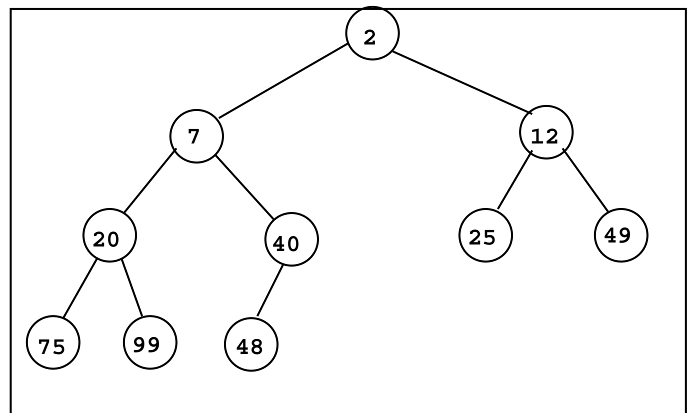
Deletion in Heap-Ordered Trees

- find a leaf node y in H.
- if $x = y$, then delete node x. **STOP**.
- replace the pointer in x by the pointer in y.
- Delete node y.
- if $key(x) < key(p(x))$, sift-up is performed. Repeat as long as necessary.
- if $key(x)$ is greater than one of its children, *sift-down* is performed: the pointer in x is swapped with the pointer of its child with the smallest key. After the swap, new father has the key not greater than any of its sons. Repeat as long as necessary.









Complexity of sift-up and sift-down

- h = height of the heap-ordered tree (length of the longest path from the root).
- d = maximum number of sons of any node.

- `sift-up`: Can be done in $O(1)$ time.
- `sift-down`: Finding the son with smallest key requires $O(d)$ time. Given the son with smallest key, the swap can be done in $O(1)$ time. $O(d)$ in total.

- `insert` requires $O(h)$ sift-ups in the worst case. Its worst-case time complexity is therefore $O(h)$.
- `delete` requires $O(h)$ sift-ups or $O(h)$ sift-downs in the worst case. Its worst-case time complexity is therefore $O(\max\{h, dh\}) = O(dh)$.
- `deletemin` requires $O(h)$ sift-downs in the worst case. Its worst-case time complexity is therefore $O(dh)$.

Heap Operations on d-heaps

d-heaps are heaps with each node having up to d sons. They are created and kept in breadth-first fashion. The height of d-trees is $O(log_d n)$.

- When inserting, new node is added to the first (in breadth-first fashion) node with less than d sons.
- When deleting, node to be deleted is swapped with the last (in breadth-first fashion) node.

- If the number of deletions is small, d can be increased to reduce insertion times.

- d-heaps are so regular that no explicit pointers are needed. Number the n nodes of a d-tree in breadth-first order.
- parent of i-th node has number $\lceil (i-1)/d \rceil$.
- children of i-th node have numbers which are integers in the interval $[d(i-1)+2..\min\{di+1, n\}]$.

Creation of d-heaps

- n insertions $O(n \log_d n)$.
- build a d-tree disregarding key values, followed by sift-downs (beginning with the last item).
  - number of nodes at height i, $i \geq 0$ is at most (by induction)
  $$\frac{n}{d^i}$$
  - number of sift-downs needed for a node x of height i needed to make the tree rooted at x heap-ordered is $O(di)$.
  - overall complexity is $O(n + \sum_{i=1}^{h} \frac{n}{d^i} di)$.

$n + \sum_{i=1}^{h} \frac{n}{d^i} di =$
$n + n \sum_{i=1}^{h} \frac{i}{d^{i-1}} \leq$
$n + n \sum_{i=1}^{h} \frac{i}{2^{i-1}} =$
$n + 2n \sum_{i=1}^{h} \frac{i}{2^i} < n + 4n = 5n$

$$O(n + \sum_{i=1}^{h} \frac{n}{d^i} di) = O(n)$$

Complexity of Operations on d-Heaps

	d-heap	leftist	binomial	Fibonacci
makeheap	$O(n)$			
insert	$O(\log_d n)$			
deletemin	$O(d \log_d n)$			
findmin	$O(1)$			
delete	$O(d \log_d n)$			
decrease	$O(\log_d n)$			
meld	$O(n)$			

Melding of Heaps

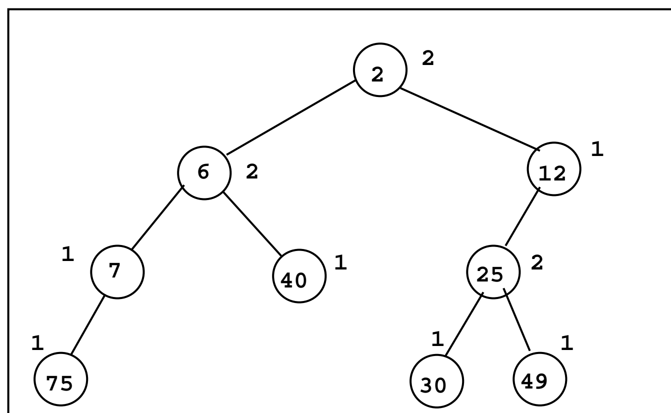
- d-heaps are not well-suited for the melding operation. One possibility is to insert items from one heap $H1$ into the other heap $H2$. This will require $O(n \log_d n)$ time.
- H could be constructed from scratch in $O(n)$ time using `makeheap`.
- It can be done faster by merging along two paths in the two heaps. For example along the rightmost paths in $H1$ and $H2$.
- Merging of two d-heaps along paths results in a heap which is not necessarily a d-heap.
- It is better to use binary trees where each node has two pointers to its left and right son.
- Efficient merging along rightmost paths requires that they are kept as short as possible.

Leftist Heaps

- *rank* of a node x in a binary tree is the minimum number of nodes on a path from x to a node with at most one son.

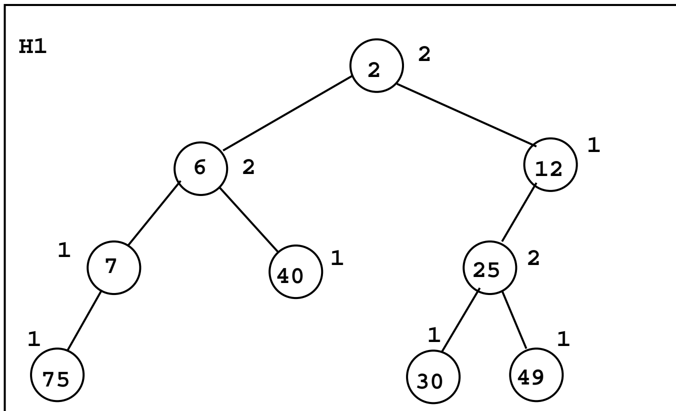
  $$rnk(x) = \begin{cases} 1 & \text{if } x \text{ has at most one node} \\ 1 + \min\{rnk(L(x)), rnk(R(x))\} & \text{otherwise} \end{cases}$$

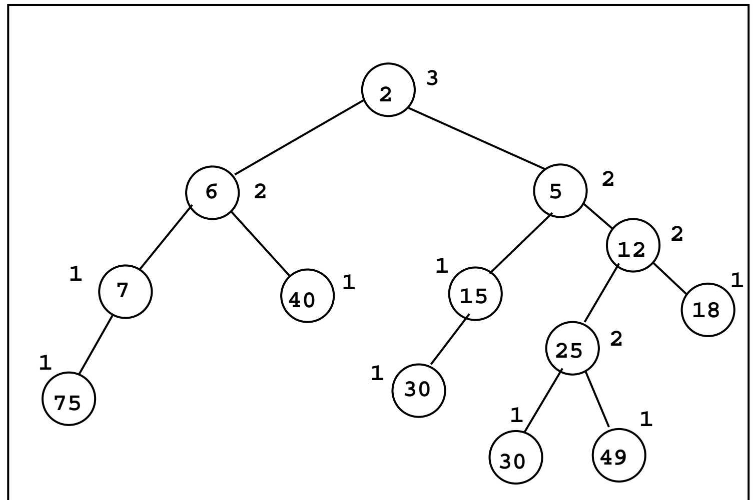
- A binary tree is *leftist* if the rank of the left son of every node is greater than or equal to the rank of the corresponding right son.
- The rightmost path in the leftist tree is a shortest path from the root to a leaf node. Its length is bounded by $\log n$ (by induction).
- A *leftist heap* is a leftist tree with items arranged in heap order.

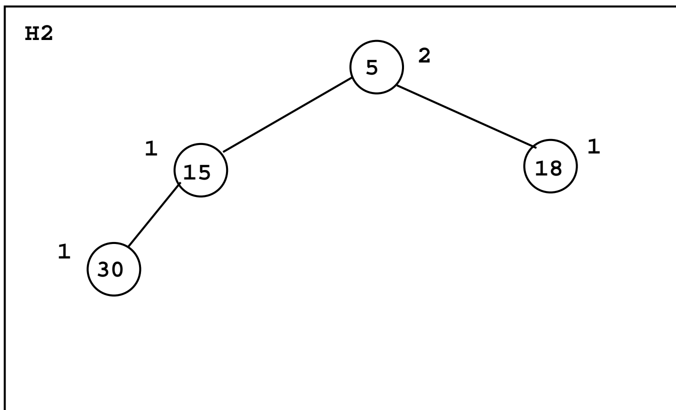


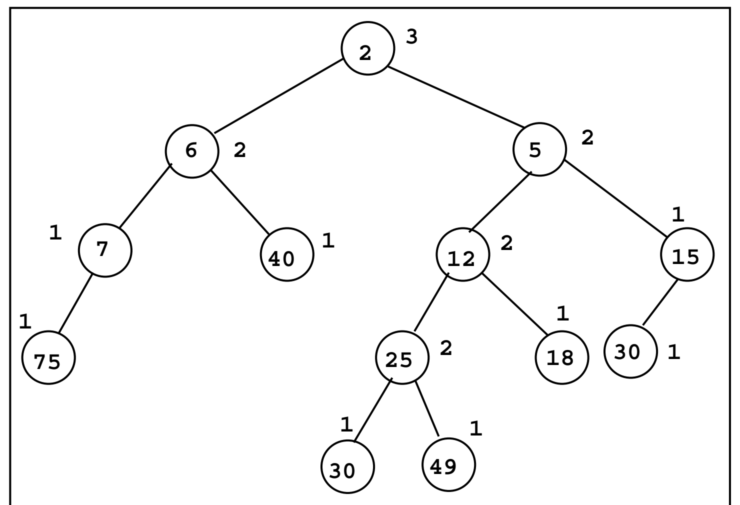
Melding Two Leftist Heaps

- Merge the two right paths, arranging nodes in nondecreasing order by key. The merged path is the rightmost.
- Recompute the ranks of nodes.
- Make the tree leftist by swapping left and right children as necassary.
- All this can be done in one sweep by a recursive procedure in $O(\log n)$ time.









`insert` on Leftist Heaps

Make the item to be inserted into a one-item leftist heap, and meld it with the existing heap; $O(\log n)$.

`deletemin` on Leftist Heaps

Remove the root and meld the two subtrees. Note that these subtrees are leftist heaps; $O(\log n)$ time.

`heapify` on Leftist Heaps

- Meld heaps pairwise and place the resulting heap at the rear of the queue.
- After the first scan the number of heaps is $\lfloor \frac{k}{2} \rfloor$.
- Continue scanning until the queue contains only one heap. There is $\lfloor \log k \rfloor$ scans.

- Let n_i^j denote the number of items in the i-th heap after j-th scan.
- $\sum_{i=1}^{\lfloor \frac{k}{2^j} \rfloor} n_i^j = n$ for all $j = 1, 2, ..., \lfloor \log k \rfloor$.
- $\prod_{l=1}^{m} x_l \leq (\frac{1}{m} \sum_{l=1}^{m} x_l)^m$ for non-negative numbers $x_1, x_2, ..., x_m$.
- Overall complexity $O(\sum_{j=1}^{\lfloor \log k \rfloor} \sum_{i=1}^{\lfloor \frac{k}{2^j} \rfloor} \log n_i^j)$.

$$\sum_{j=1}^{\lfloor \log k \rfloor} \sum_{i=1}^{\lfloor \frac{k}{2^j} \rfloor} \log n_i^j = \sum_{j=1}^{\lfloor \log k \rfloor} \log(\prod_{i=1}^{\lfloor \frac{k}{2^j} \rfloor} n_i^j) \leq$$

$$\sum_{j=1}^{\lfloor \log k \rfloor} \log((\frac{1}{\lfloor \frac{k}{2^j} \rfloor} \sum_{i=1}^{\lfloor \frac{k}{2^j} \rfloor} n_i^j)^{\lfloor \frac{k}{2^j} \rfloor}) = \sum_{j=1}^{\lfloor \log k \rfloor} \log((\frac{n}{\lfloor \frac{k}{2^j} \rfloor})^{\lfloor \frac{k}{2^j} \rfloor}) \leq$$

$$\sum_{j=1}^{\lfloor \log k \rfloor} \frac{k}{2^j} \log(\frac{n2^j}{k}) = \sum_{j=1}^{\lfloor \log k \rfloor} \frac{k}{2^j}(\log \frac{n}{k} + j) =$$

$$k \log \frac{n}{k} \sum_{j=1}^{\lfloor \log k \rfloor} \frac{1}{2^j} + k \sum_{j=1}^{\lfloor \log k \rfloor} \frac{j}{2^j} \leq k \log \frac{n}{k} + 2k$$

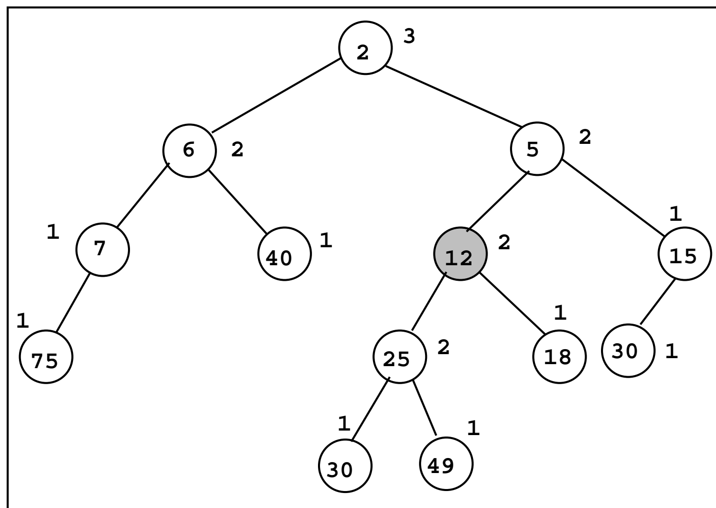
$O(k \max\{1, \log \frac{n}{k}\}) = O(k \max\{1, \log \frac{n}{k}\})$, where k, $k \leq n$, is the number of heaps to be heapified.

`makeheap` on Leftist Heaps

Construct one-item leftist heaps and use `heapify`. $O(n)$ time since $k = n$.

`delete` on Leftist Heaps

- Remove the item that has to be deleted. The leftist heap breaks into three heaps. Two of them are leftist heaps.
- Third (containing the root) can be changed into leftist heap in $O(\log n)$ time if parent pointers are available.
- Use `meld` twice; $O(\log n)$.
- Need of parent pointers implies that other operations must be modified.



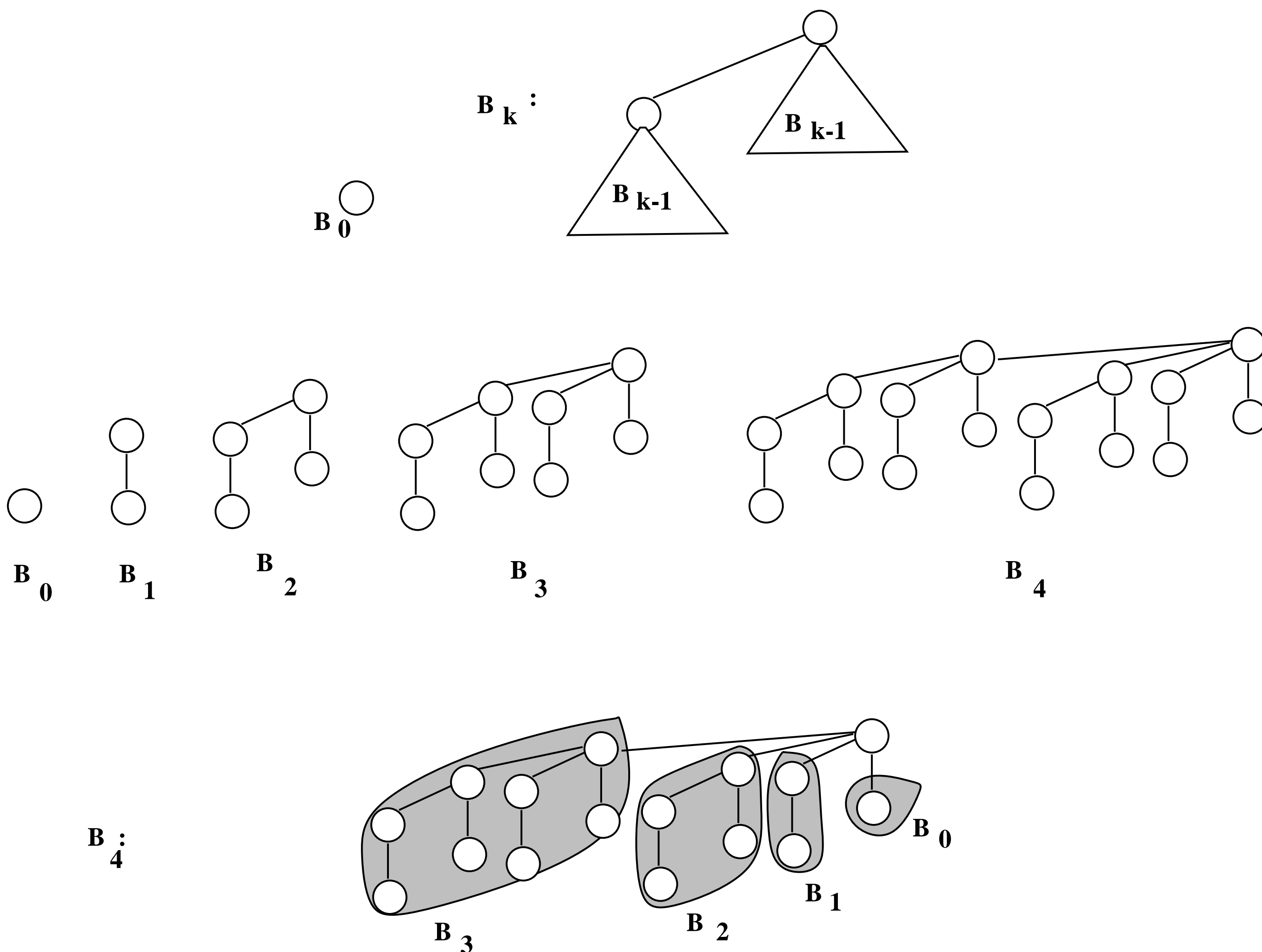
Lazy Deletion

- Mark the item as deleted (without removing it).
- Perform actual deletion during subsequent `deletemin` or `findmin`.
- Modification of `findmin` is necessary: Traverse the leftist heap in preorder. Backtrack as soon as undeleted node is encountered. Heapify $k+1$ subheaps of undeleted nodes. $O(k \max\{1, \log(\frac{n}{k+1})\})$ time needed where k is the number of deleted nodes encountered during the preorder traversal.
- Modification of `deletemin` is necessary: Mark the root as deleted. Traverse the leftist heap in preorder. Backtrack as soon as undeleted node is encountered. Heapify subheaps of undeleted nodes. $O(k \max\{1, \log(\frac{n}{k+1})\})$ time needed.
- Marking of deleted nodes permits also lazy meld. To meld two heaps make them children of a dummy deleted node with very low key (swap subheaps if necessary to preserve leftness). $O(1)$ time needed.

Complexity of Operations on Leftist Heaps

	d-heap	leftist	leftist (lazy)	binomial	Fibonacci
makeheap	$O(n)$	$O(n)$	$O(n)$		
insert	$O(\log_d n)$	$O(\log n)$	$O(1)$		
deletemin	$O(d \log_d n)$	$O(\log n)$	$O(k \max\{1, \log \frac{n}{k+1}\})$		
findmin	$O(1)$	$O(1)$	$O(k \max\{1, \log \frac{n}{k+1}\})$		
delete	$O(d \log_d n)$	$O(\log n)$	$O(1)$		
decrease	$O(\log_d n)$	$O(\log n)$	$O(\log n)$		
meld	$O(n)$	$O(\log n)$	$O(1)$		

Binomial Trees

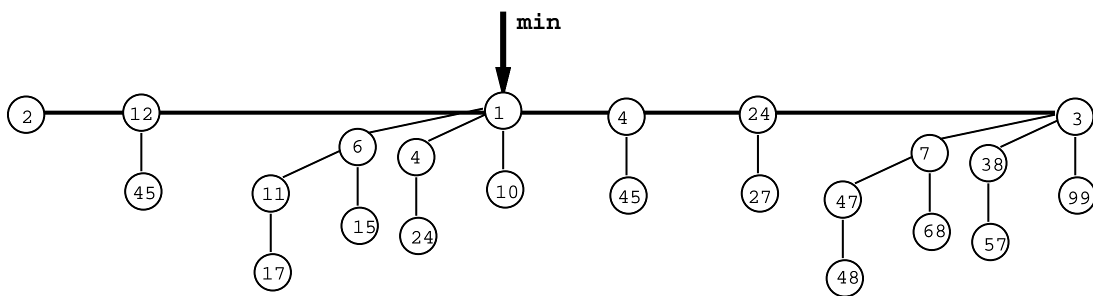


- B_k has $n = 2^k$ nodes,
- B_k has height k,
- there are $\binom{k}{i}$ nodes at depth i, $i = 0, 1, ..., k$.
- the root of B_k has degree k. Its k children are numbered from left to right by their degrees: $k-1, k-2, ..., 1, 0$.
- maximum degree in B_k is $\log n$ hvor $n = 2^k$ (number of nodes).

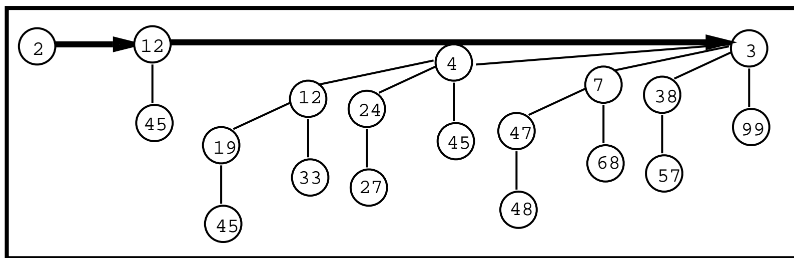
Binomial Heaps

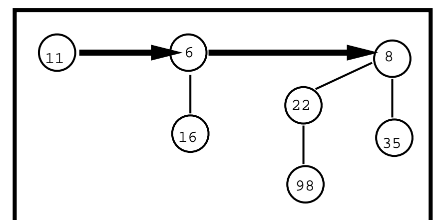
A *binomial heap* is a set of binomial trees which satisfy:

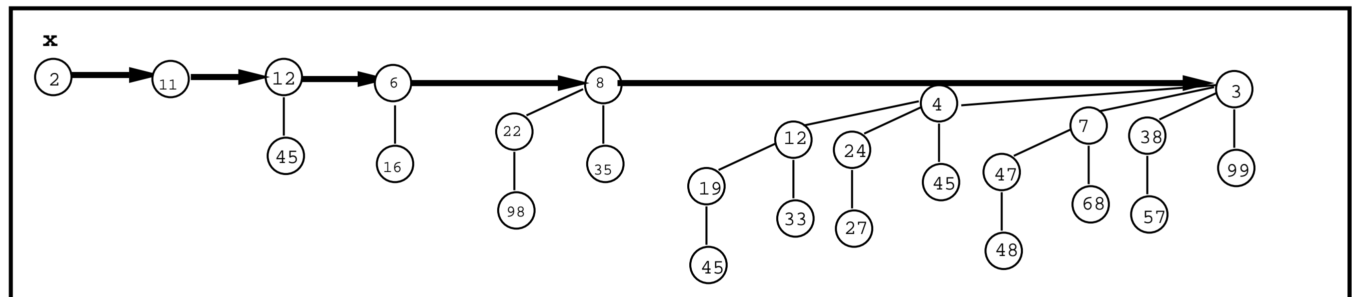
- Each binomial tree is heap-ordered.
- There is at most one binomial tree with a given degree.

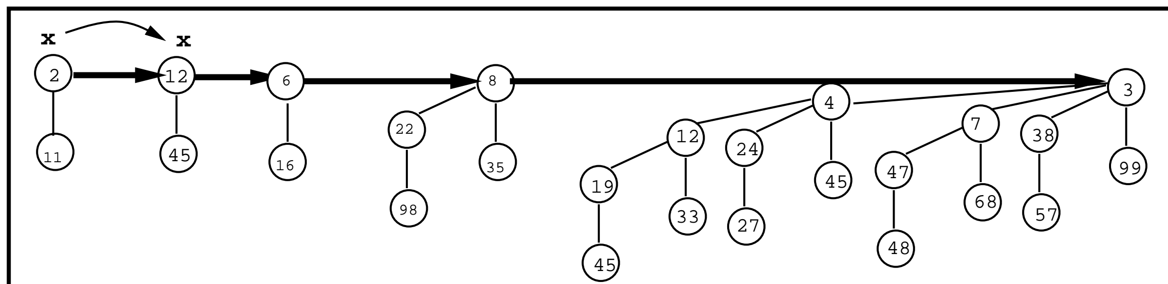


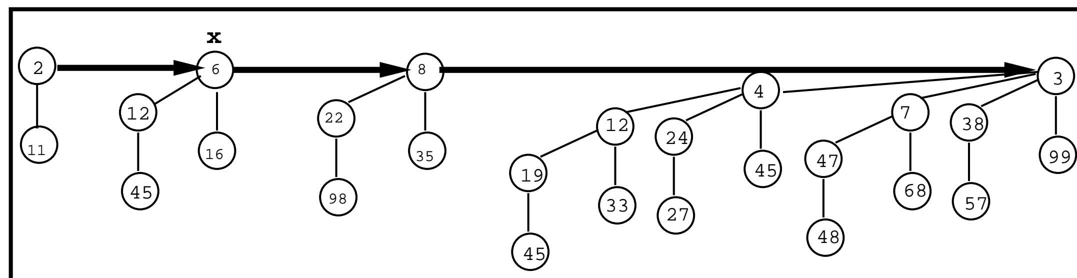
Melding Two Binomial Heaps

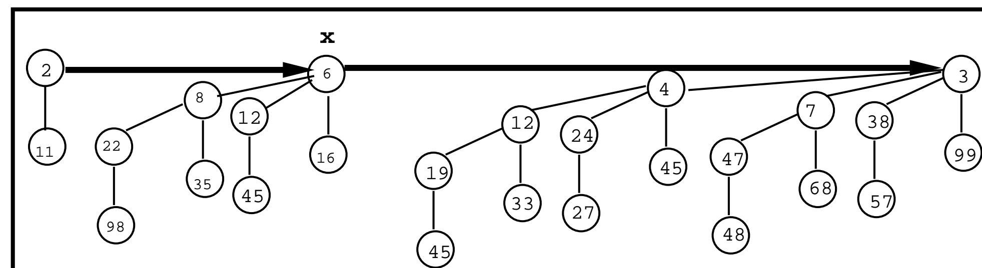










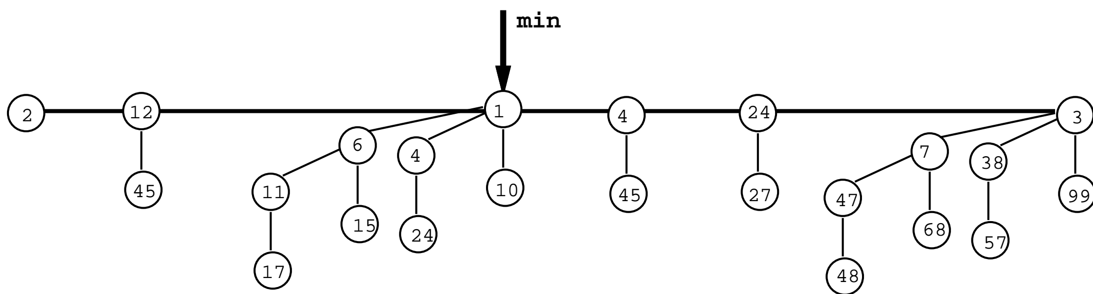


Complexity of Operations on Binomial Heaps

	d-heap	leftist	leftist (lazy)	binomial	Fibonacci
makeheap	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
insert	$O(\log_d n)$	$O(\log n)$	$O(1)$	$O(logn)$	
deletemin	$O(d \log_d n)$	$O(\log n)$	$O(k \max\{1, \log \frac{n}{k+1}\})$	$O(logn)$	
findmin	$O(1)$	$O(1)$	$O(k \max\{1, \log \frac{n}{k+1}\})$	$O(logn)$	
delete	$O(d \log_d n)$	$O(\log n)$	$O(1)$	$O(logn)$	
decrease	$O(\log_d n)$	$O(\log n)$	$O(\log n)$	$O(logn)$	
meld	$O(n)$	$O(\log n)$	$O(1)$	$O(logn)$	

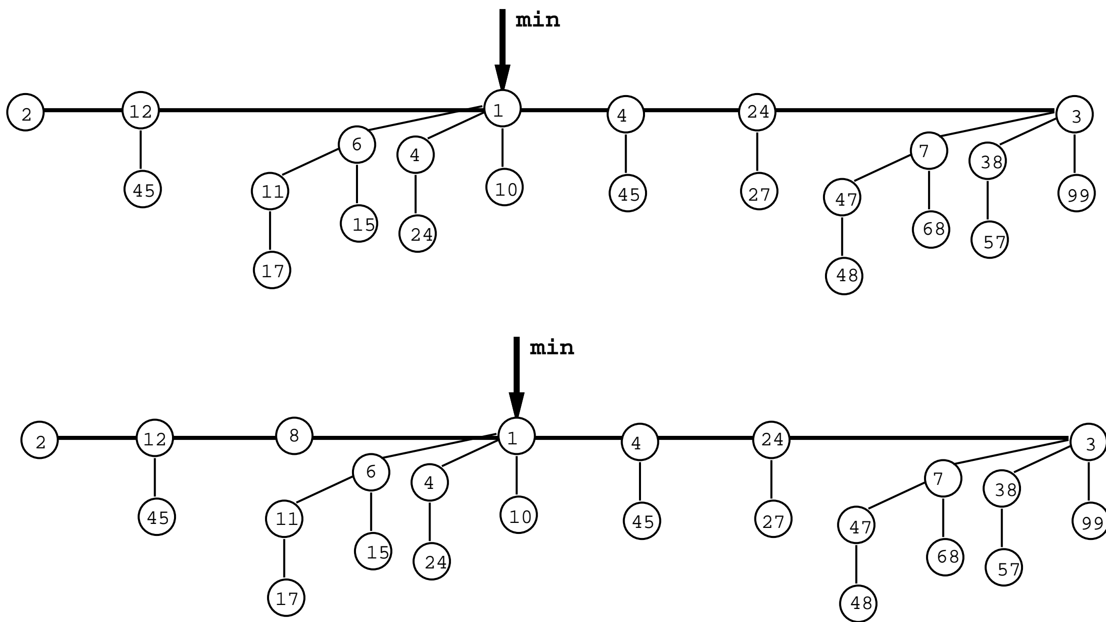
Fibonacci Heaps

A *Fibonacci heap* is a set of heap-ordered trees.



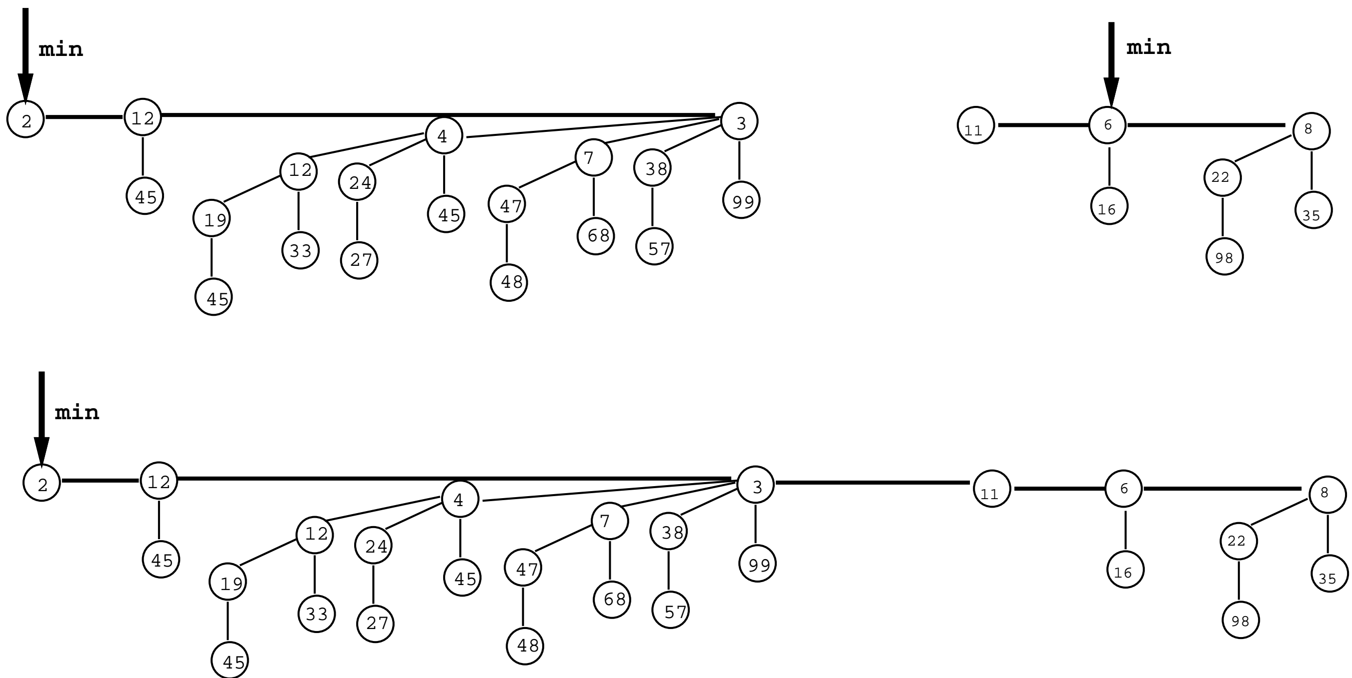
- A node x becomes *marked* if it lost a child since the last time x was made the child of another node.
- $t(H)$: number of trees in the root list.
- $m(H)$: number of marked nodes.
- The potential of a Fibonacci heap: $\Phi(H) = t(H) + 2m(H)$.

Inserting into a Fibonacci Heap



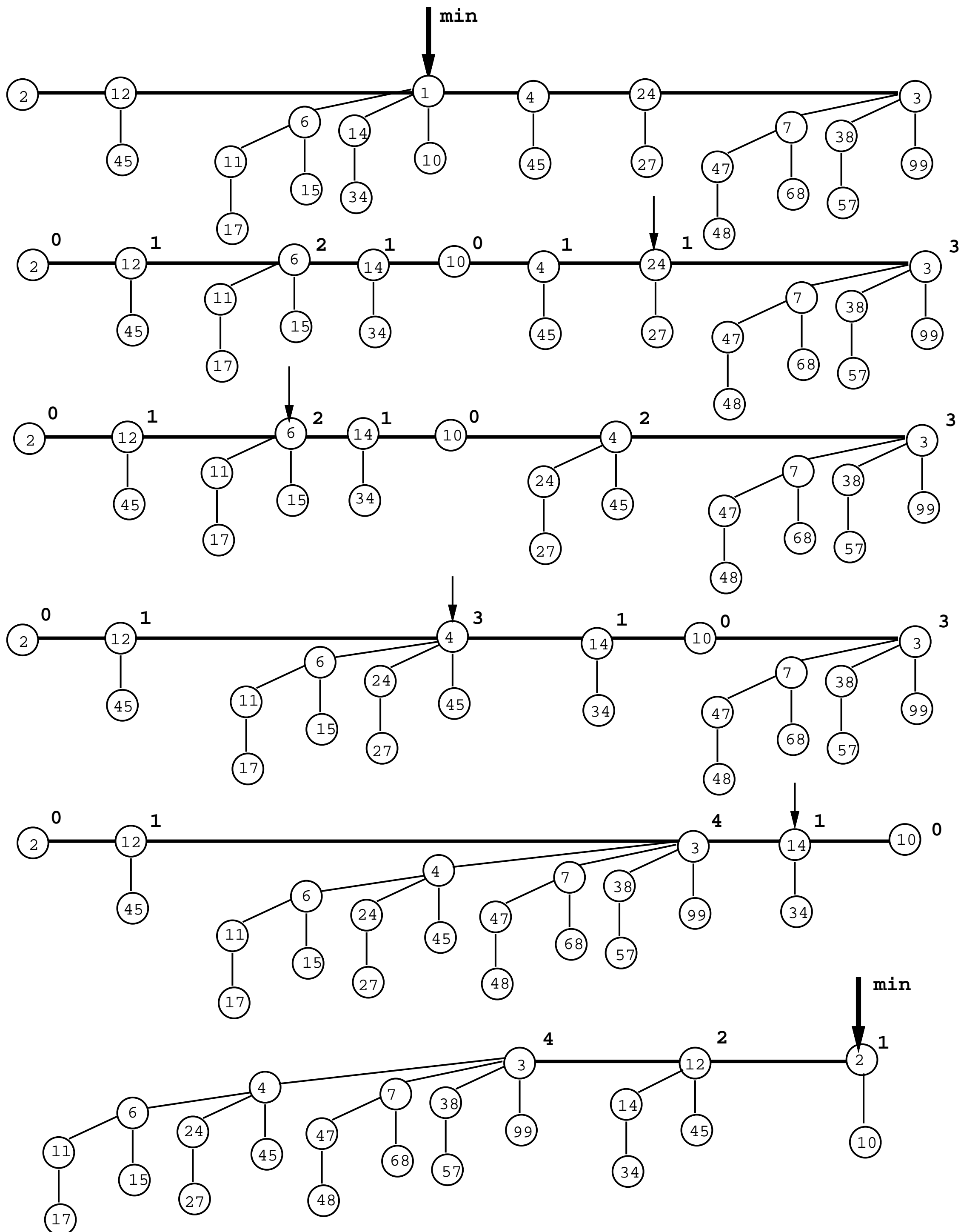
- actual cost: $O(1)$.
- potential change: $(t(H)+1)+2m(H)-t(H)-2m(H) = 1$
- amortized cost: $O(1)$

Melding Two Fibonacci Heaps



- actual cost: $O(1)$.
- potential change: $\Phi(H) - \Phi(H_1) - \Phi(H_2) = t(H) - t(H_1) - t(H_2) + 2m(H) - 2m(H_1) - 2m(H_2) = 0$
- amortized cost: $O(1)$

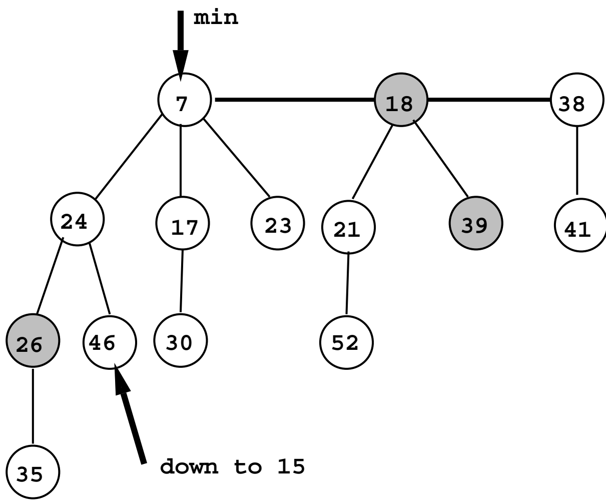
`deletemin` in Fibonacci Heaps

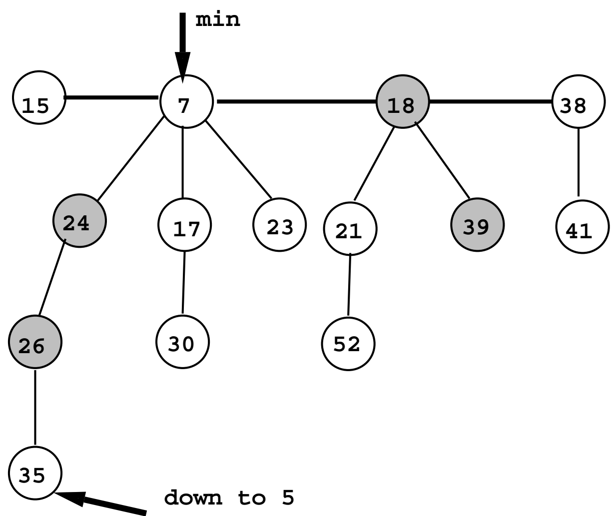


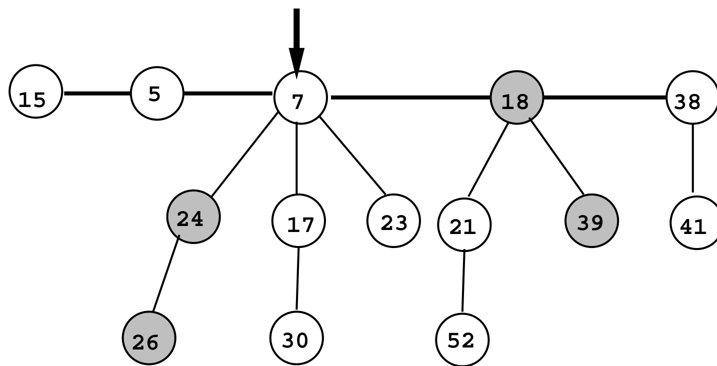
Amortized Cost of `deletemin`

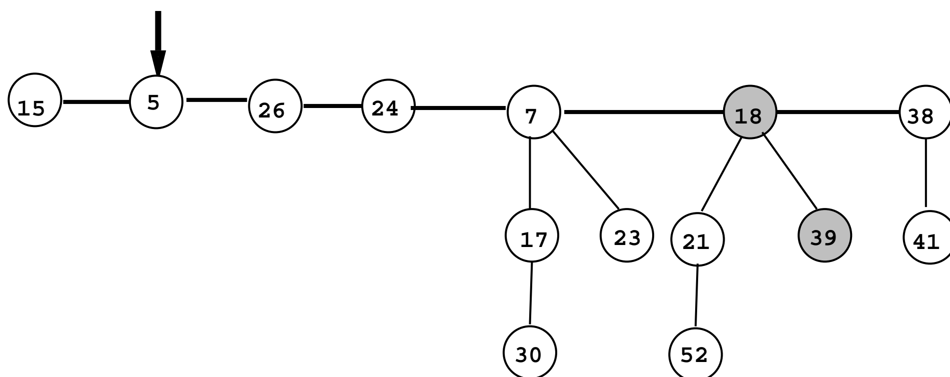
- $D(n)$: maximum degree of a node in a Fibonacci heap with n items.
- actual cost is $O(t(H)) + O(D(n))$ since:
  - Removal of smallest item + addition of its children to the root list: $O(D(n))$.
  - Consolidation requires one scan of the root list. Whenever needed, melding requiring $O(1)$ time is carried out. Size of the root list: $t(H) + D(n) - 1$. Cost of consolidation: $O(t(H)) + O(D(n))$.
  - Identification of new minimum: $O(D(n))$.
- potential change: $D(n)+1+2m(H)-t(H)-2m(H) = D(n) + 1 - t(H)$.
- amortized cost: $O(t(H))+O(D(n))+D(n)+1-t(H) = O(t(H))+O(D(n))-t(H) = O(D(n))$ if potential units are sufficiently large to ensure that $t(H)$ dominates the constant hidden in $O(t(H))$.

`decrease` in Fibonacci Heaps









Amortized Cost of `decrease`

- each cascading cut requires $O(1)$ time (disregarding recursive call)
- actual cost: $O(c)$ where c is the number of cascading cuts.
- potential change: Number of new trees in the root tree is increased by c. Each recursive call of cascading cut procedure (except the last one) clears a marked node. Last cascading cut can mark a node. The number of marked nodes is reduced to $m(H)-(c-1)+1 = m(H)-c+2$.

  $$(t(H)+c)+2(m(H)-c+2)-t(H)-2m(H) = 4-c$$

- amortized cost: $O(c)+4-c = O(1)$ if potential units are sufficiently large to dominate the constant hidden in $O(c)$

Maximum Degree in Fibonacci Heaps

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{otherwise} \end{cases}$$

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

- x any node in a Fibonacci heap.
- $size(x)$: number of nodes in the tree rooted at x.
- k: degree of x.
- $n \geq size(x) \geq F_{k+2} \geq \phi^k$ where $\phi = (1 + \sqrt{5})/2$.
- $\log_\phi n \geq k$

Complexity of Operations on Fibonacci Heaps

	d-heap	leftist	leftist (lazy)	binomial	Fibonacci
makeheap	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
insert	$O(\log_d n)$	$O(\log n)$	$O(1)$	$O(logn)$	$O(1)$
deletemin	$O(d \log_d n)$	$O(\log n)$	$O(k \max\{1, \log \frac{n}{k+1}\})$	$O(logn)$	$O(logn)$
findmin	$O(1)$	$O(1)$	$O(k \max\{1, \log \frac{n}{k+1}\})$	$O(logn)$	$O(1)$
delete	$O(d \log_d n)$	$O(\log n)$	$O(1)$	$O(logn)$	$O(logn)$
decrease	$O(\log_d n)$	$O(\log n)$	$O(\log n)$	$O(logn)$	$O(1)$
meld	$O(n)$	$O(\log n)$	$O(1)$	$O(logn)$	$O(1)$