Fibonacci Heaps

Priority Queue Operations

- Maintain a set Q (initially empty)
- $\boldsymbol{Insert}(\mathbf{x}, \mathbf{k})$: add x to Q with value $d(x) = k$

  Slight overkill for Dijkstra

  Start by inserting s with value 0, all other nodes with value ∞

- $\boldsymbol{Deletemin}()$: Find an $x \in Q$ minimizing $d(x)$. Delete x from Q and return it.

- $\boldsymbol{Decreasekey}(\boldsymbol{x}, \boldsymbol{k})$: $d(x) \leftarrow \min\{d(x), k\}$

Priority Queue Implementations

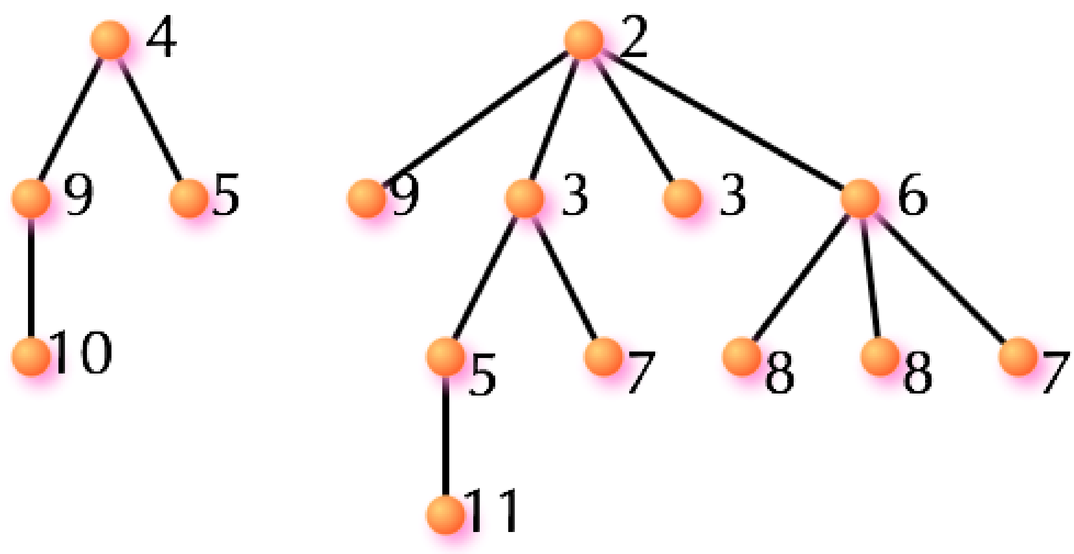
- Next: **Binomial Trees**
  - Insert = Decreasekey = Deletemin = $O(\log n)$ time
- After that: **Fibonacci Trees**
  - Insert = Decreasekey = $O(1)$, Deletemin = $O(\log n)$

Heap-ordered trees

- Represent Q by a **forest** of rooted trees
- One element per node
- *Heap ordered:* $d(x) \geq d(\text{parent}(x))$

  $\Rightarrow$ element with min. key at a root

  $\Rightarrow$ two heap ordered trees can be linked in $O(1)$ time

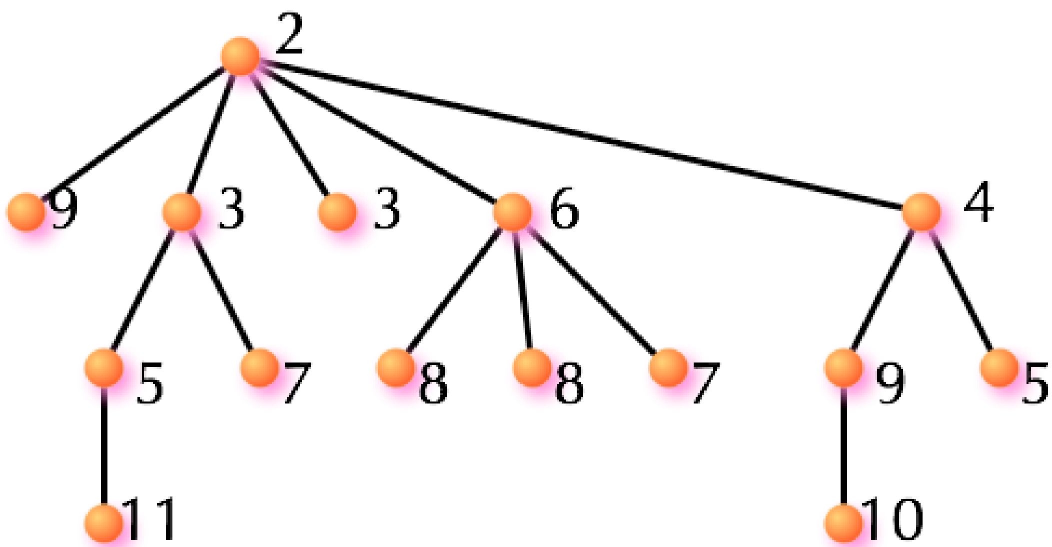


Heap-ordered trees

- Represent Q by a **forest** of rooted trees
- One element per node
- *Heap ordered:* $d(x) \geq d(\text{parent}(x))$

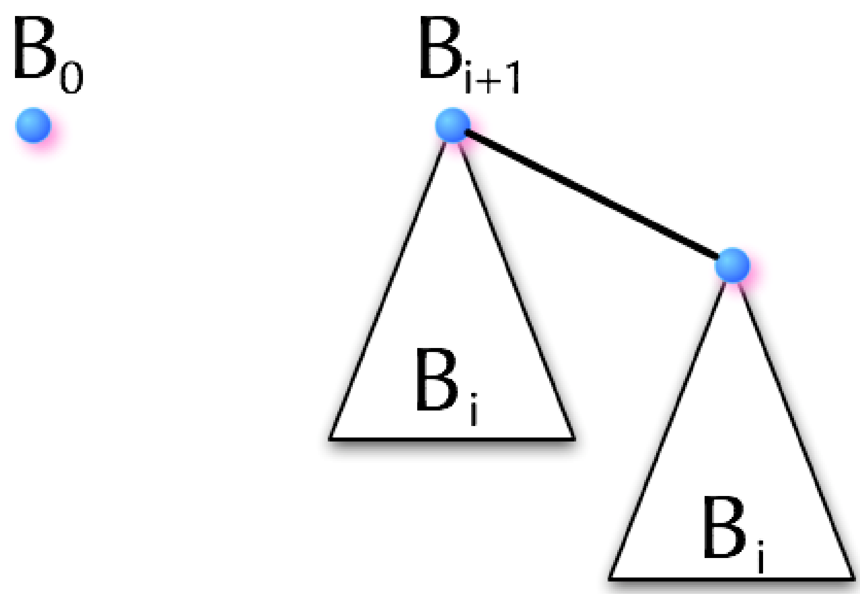
  $\Rightarrow$ element with min. key at a root

  $\Rightarrow$ two heap ordered trees can be linked in $O(1)$ time



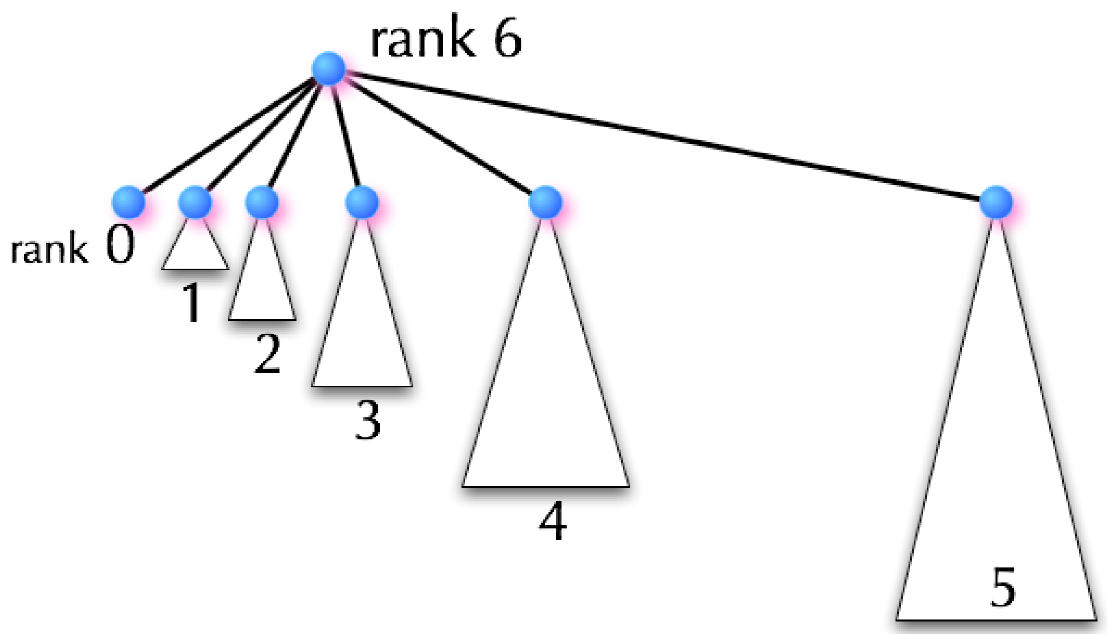
Binomial Trees

- A **binomial tree** is a tree that obeys a special structure
- A binomial tree B_i of "rank 0" must be a **single node**.
- A binomial tree B_{i+1} of "rank $i + 1$" is:
  - A B_i, with another B_i added as the rightmost child of the root.



Binomial Trees

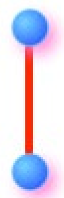
- Equivalent definition:
  - B_{i+1} = one root node with children $B_0, B_1, \ldots, B_i$



- Size of binomial trees: $|B_0| = 1,\ |B_i| = 2|B_{i-1}| \ \Rightarrow\ |B_i| = 2^i$

Binomial Trees

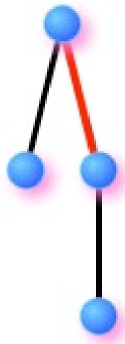
- B_i = binomial tree with rank i
  - B_0 = one node
  - B_{i+1} = make one B_i the rightmost child of another B_i



Rank 1

Binomial Trees

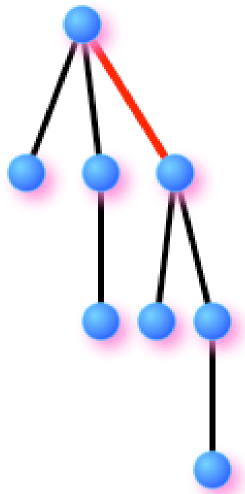
- B_i = binomial tree with rank i
  - B_0 = one node
  - B_{i+1} = make one B_i the rightmost child of another B_i



Rank 2

Binomial Trees

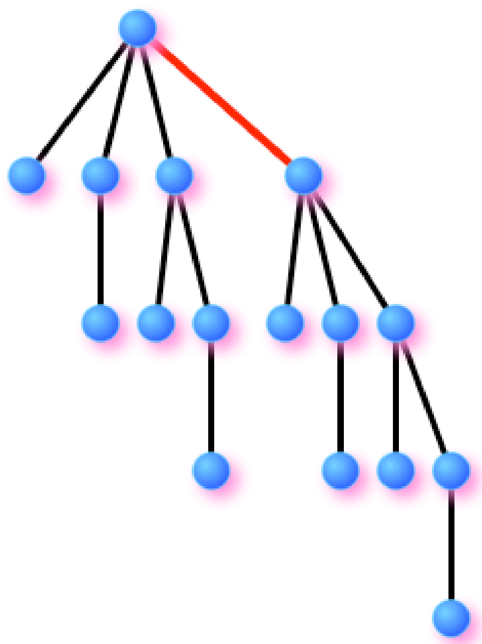
- B_i = binomial tree with rank i
  - B_0 = one node
  - B_{i+1} = make one B_i the rightmost child of another B_i



Rank 3

Binomial Trees

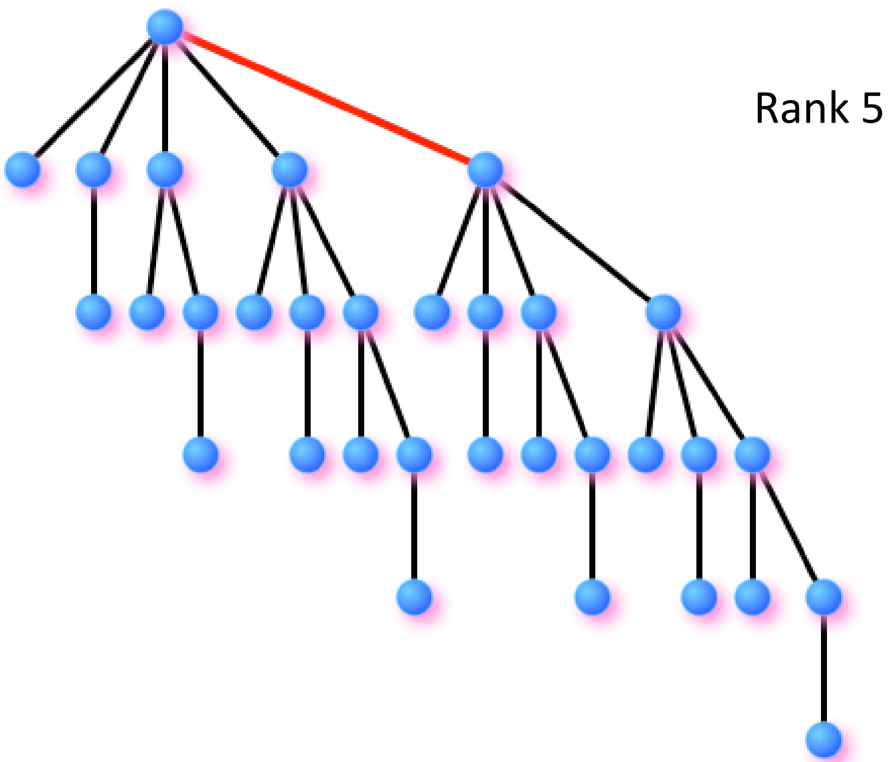
- B_i = binomial tree with rank i
  - B_0 = one node
  - B_{i+1} = make one B_i the rightmost child of another B_i



Rank 4

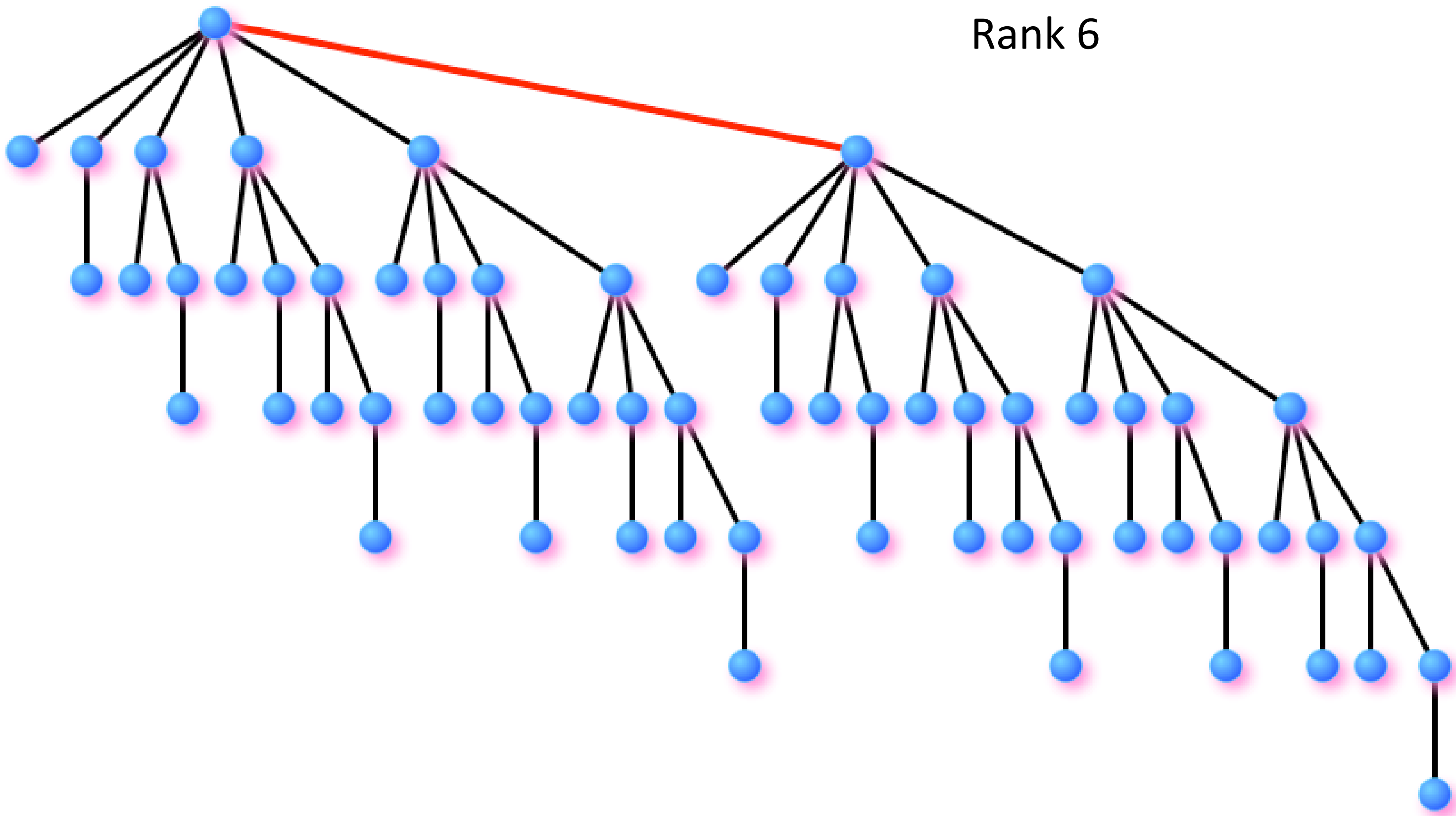
Binomial Trees

- B_i = binomial tree with rank i
  - B_0 = one node
  - B_{i+1} = make one B_i the rightmost child of another B_i



Binomial Trees

- B_i = binomial tree with rank i
  - B_0 = one node
  - B_{i+1} = make one B_i the rightmost child of another B_i



Binomial *Heaps*

- Structure of a binomial heap:

  (1) A list of heap-ordered binomial trees

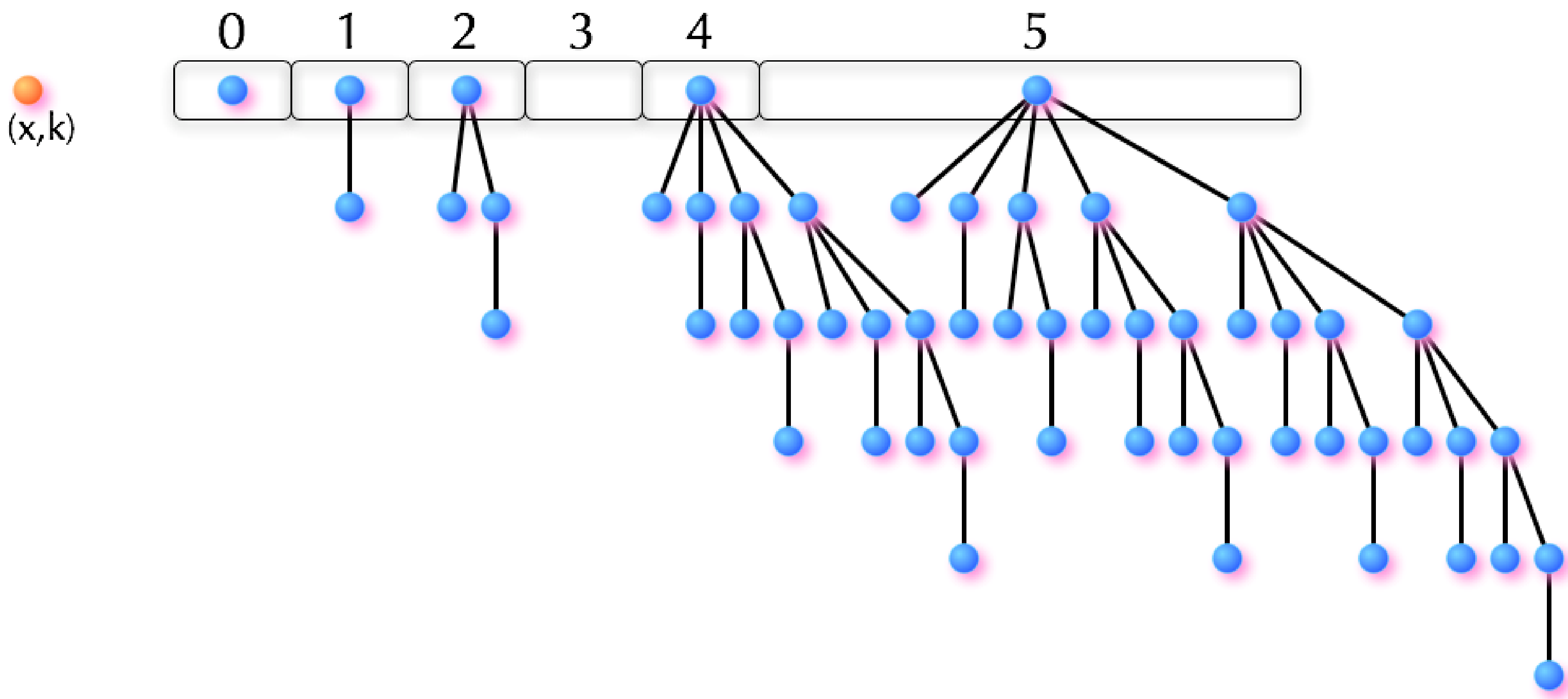
  (2) At most one tree of each rank

- $Insert(X, k)$:
  - Create a new node X (a B_0 tree)
  - Add X to the list of trees (1)
  - **Now there might be two rank-0 trees**
  - Do some work to restore property (2)

Insertion in a Binomial Heap

$Insert(x, k)$: create new B_0 tree (one node) for x

Now more than one B_0 tree

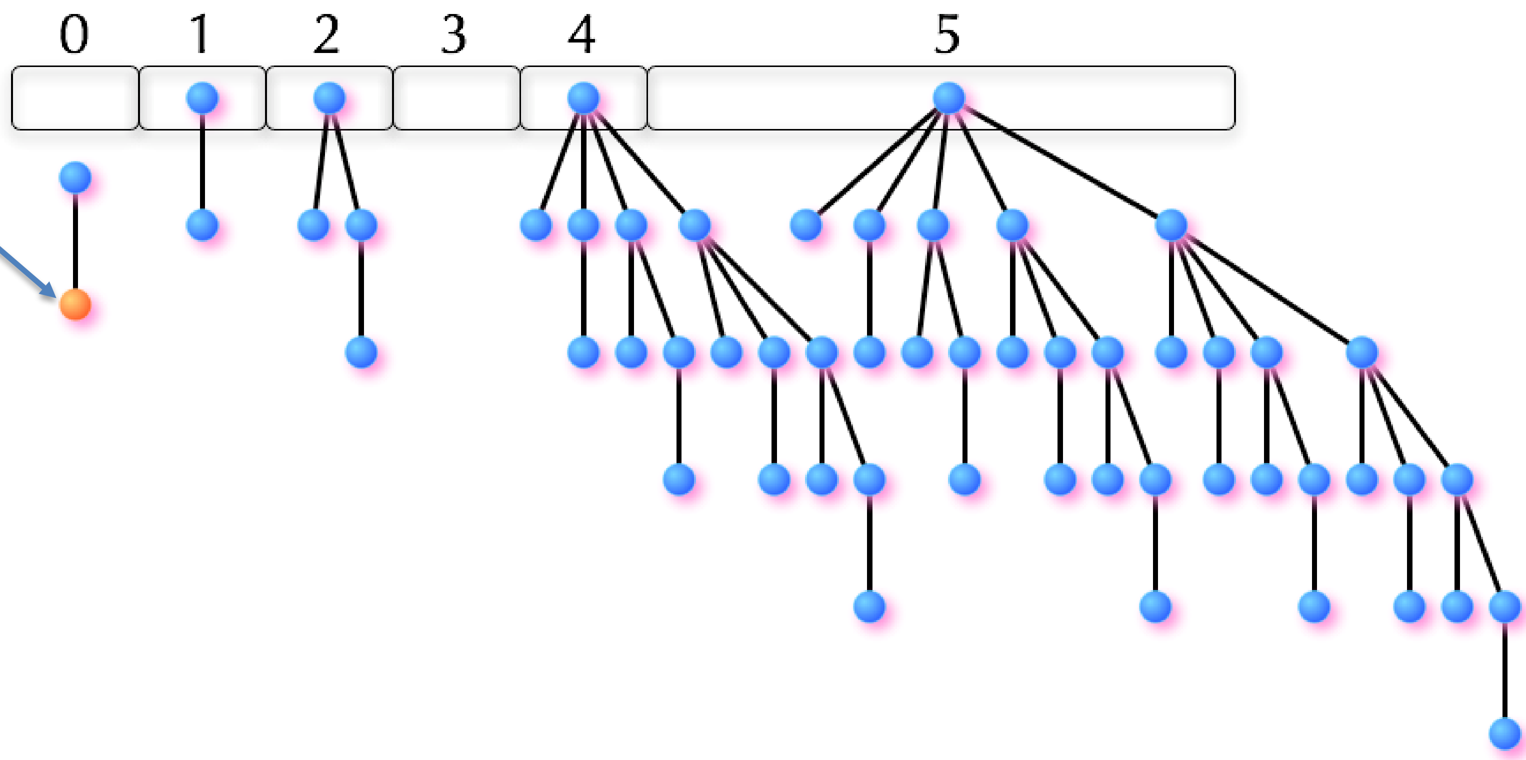


Insertion in a Binomial Heap

$Insert(x, k)$:

Link B_0 trees. Now there's too many B_1 trees.

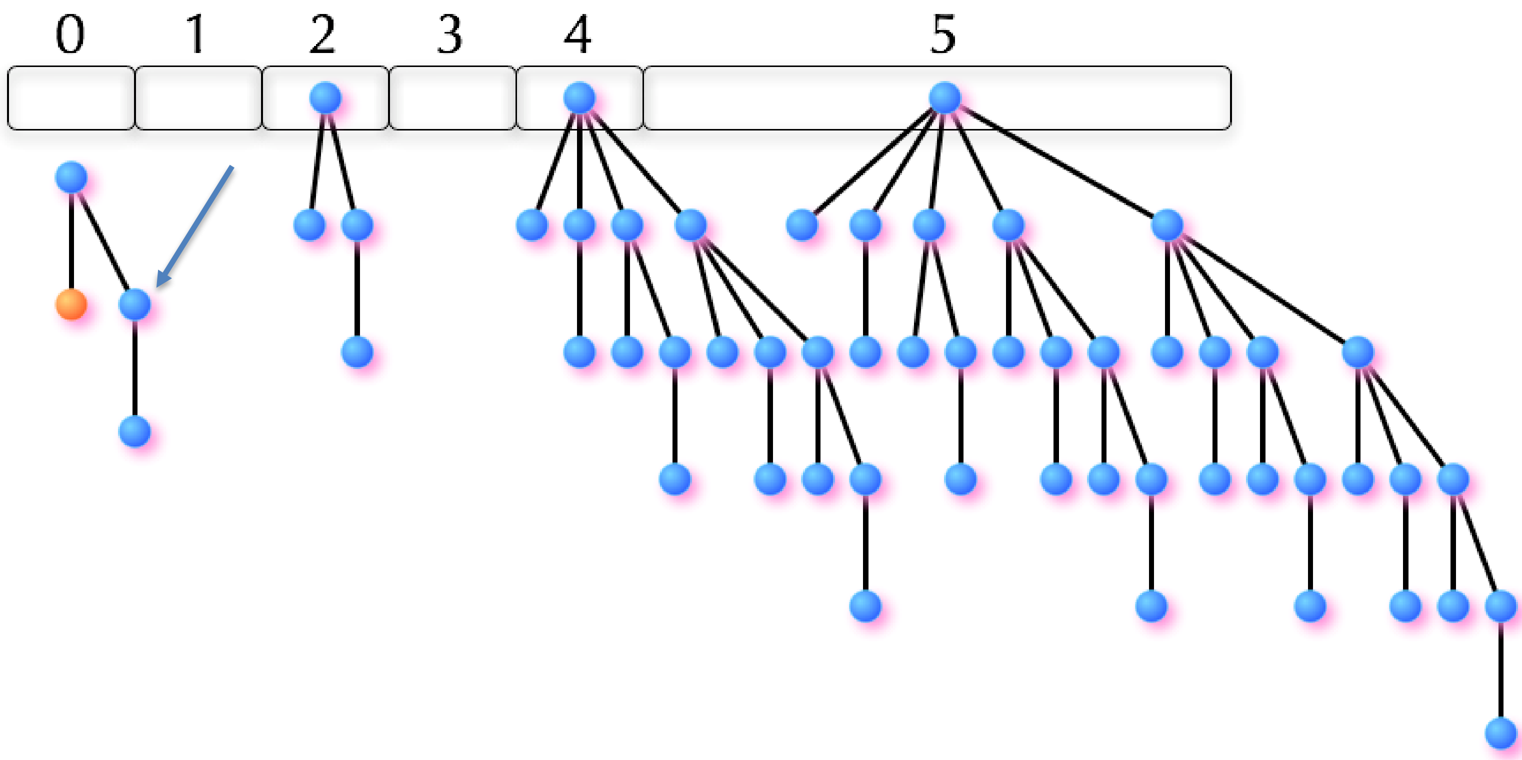
Make rightmost child



Insertion in a Binomial Heap

Insert(x,k):

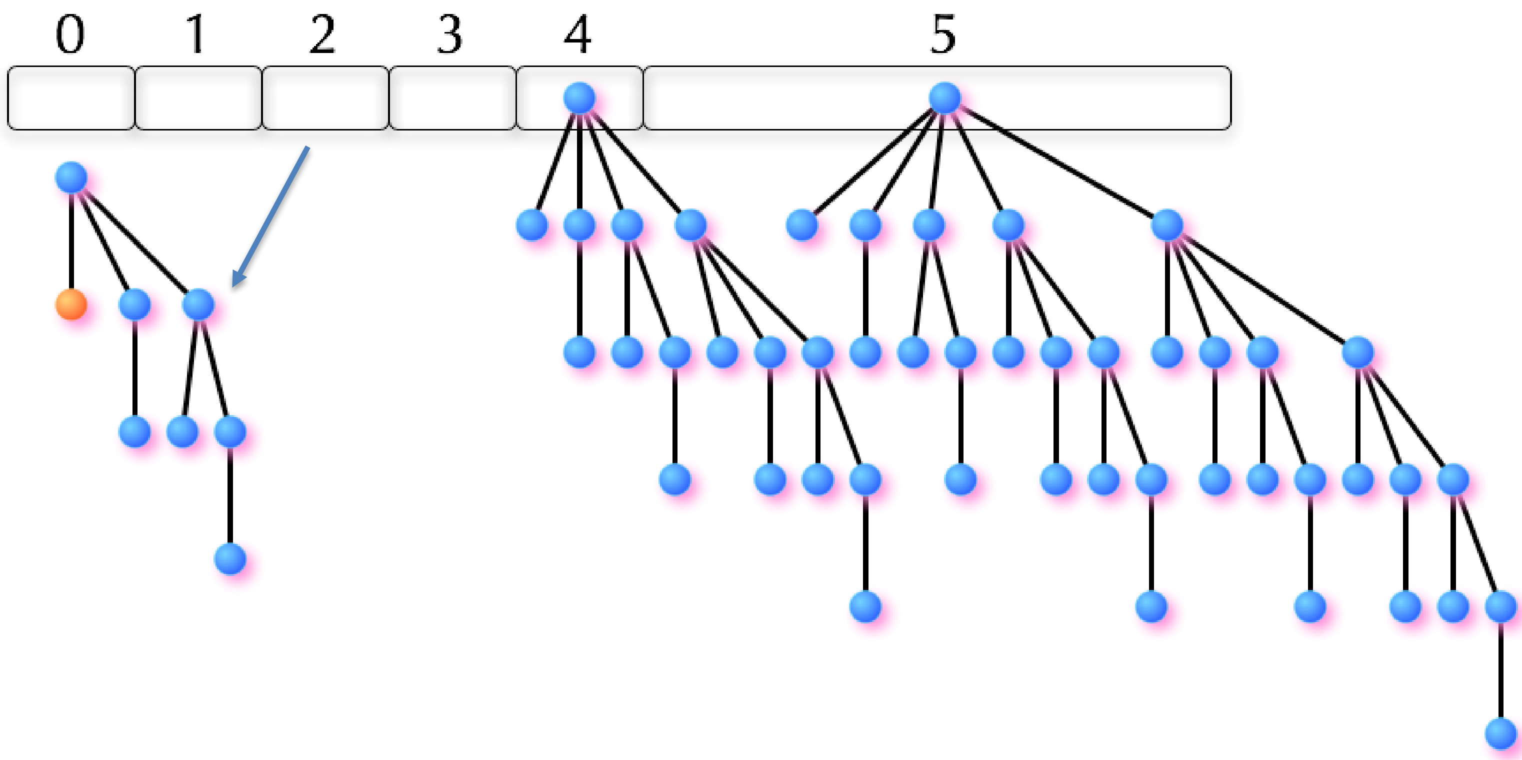
Link B_1 trees. Now too many B_2 trees.



Insertion in a Binomial Heap

Insert(x,k):

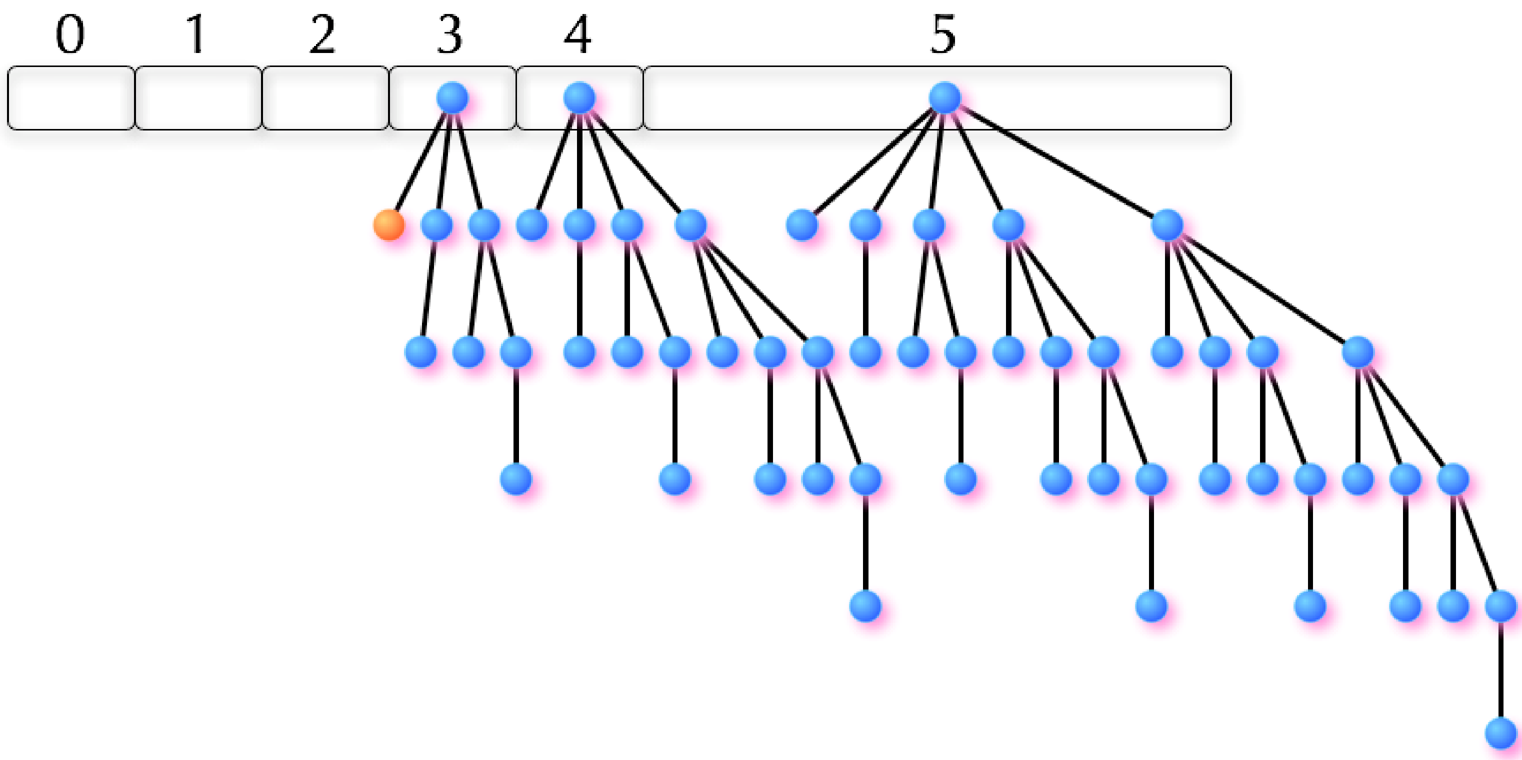
Link B_2 trees.



Insertion in a Binomial Heap

Insert(x,k):

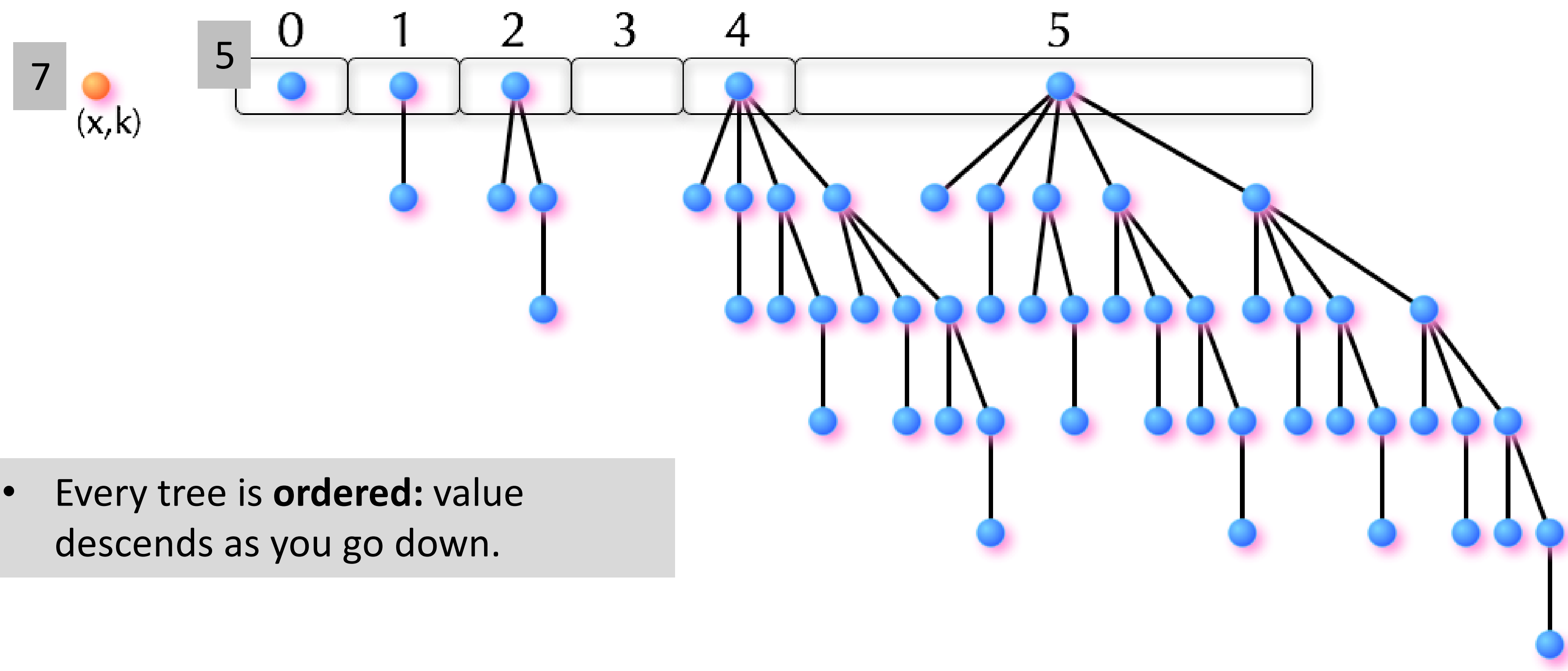
Link B_2 trees. At most 1 tree of each rank so we're done.



Insertion: Which Goes Under?

$Insert(x, k)$: create new B_0 tree (one node) for x

Now more than one B_0 tree

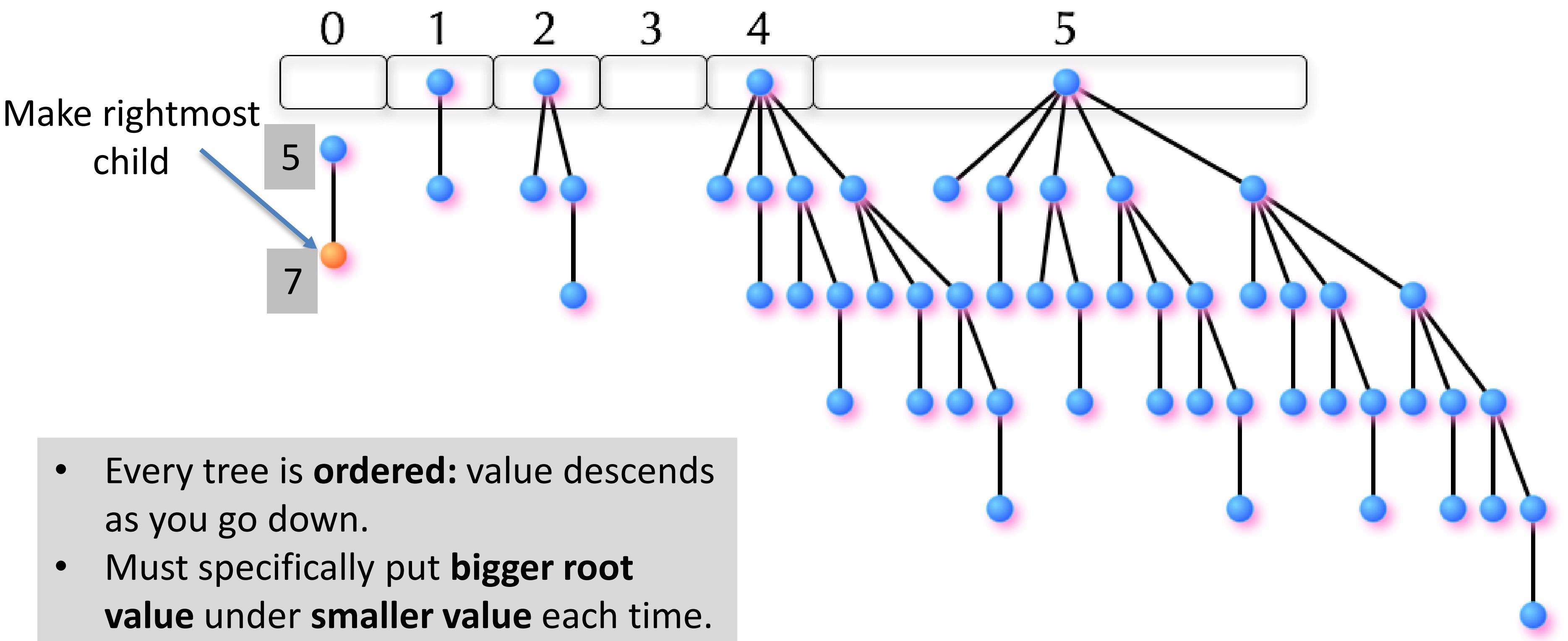


- Every tree is **ordered:** value descends as you go down.

Insertion: Which Goes Under?

$Insert(x, k)$:

Link B_0 trees. Now there's too many B_1 trees.



- Every tree is **ordered:** value descends as you go down.
- Must specifically put **bigger root value** under **smaller value** each time.

Binomial *Heaps*

- Structure of a binomial heap:

  (1) A list of heap-ordered binomial trees

  (2) At most one tree of each rank

- Insert(X,k) :
  - Create a new node X (a B_0 tree)
  - Repeatedly **link** trees until property is restored

Claim:

Insert takes $O(\log n)$ operations in the worst case.

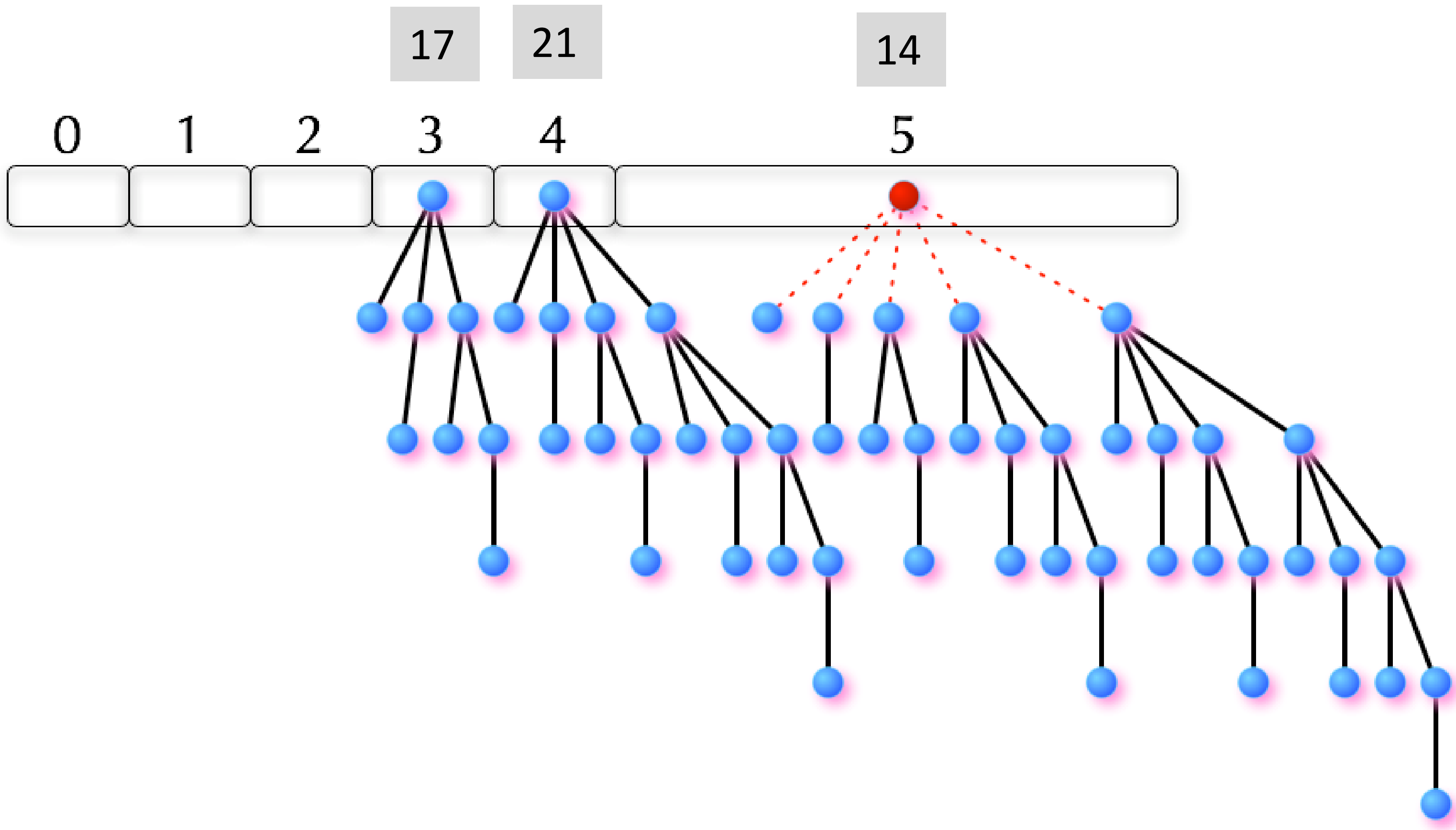
- Initially: n elements and $O(\log n) + 1$ inserted tree in the collection
- One operation links two trees → one fewer in the collection
- Can only perform $O(\log n)$ linkings per insert

Binomial *Heaps*

- Structure of a binomial heap:
  (1) A list of heap-ordered binomial trees
  (2) At most one tree of each rank
- Insert(X,k): $O(\log n)$ operations (# of trees)
- Deletemin():
  - Scan **roots** of trees in collection to find the one with min key
  - Remove and return min root **r**
  - Insert children of **r** to the collection

Deletion in a Binomial Heap

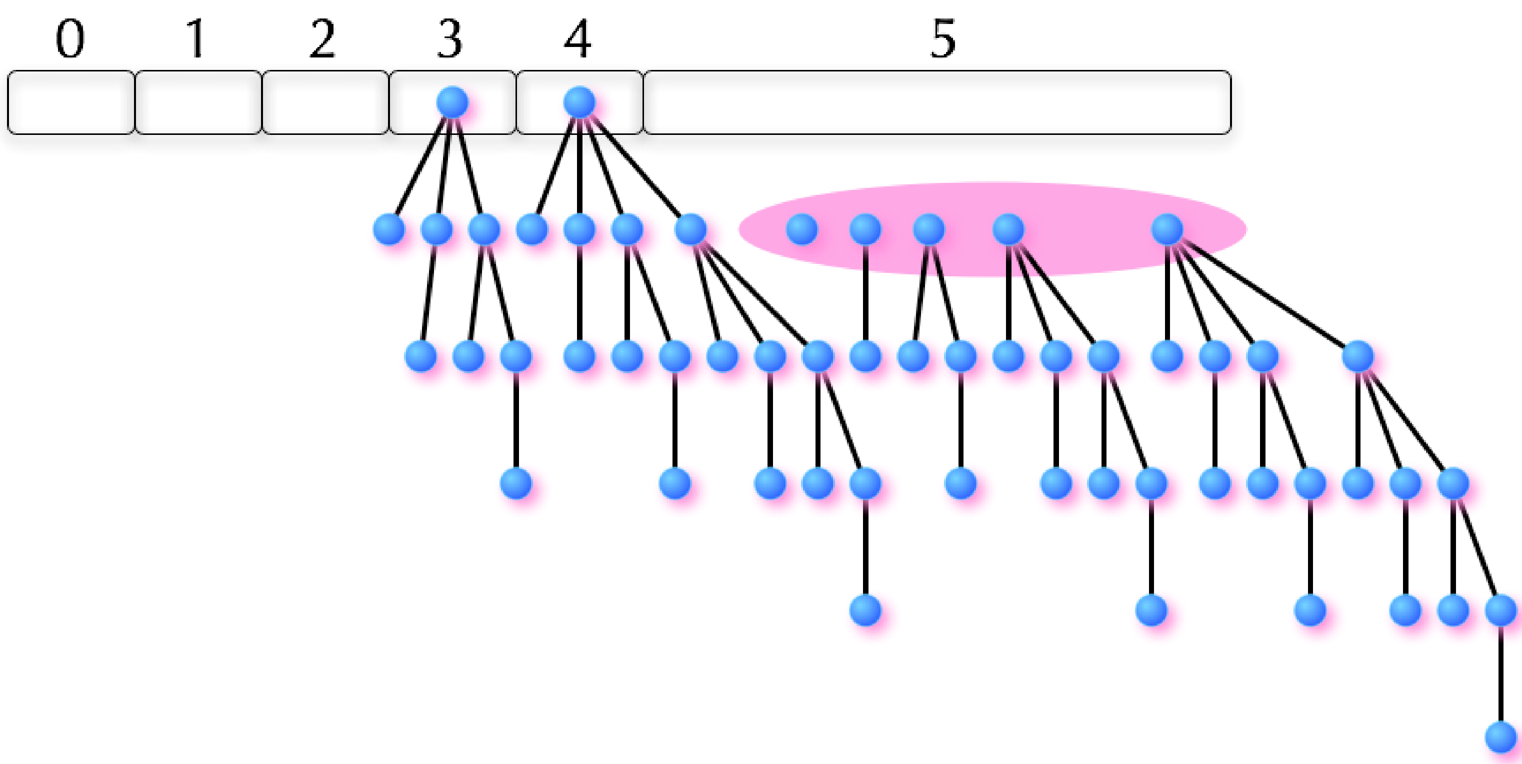
Deletemin() : Search all tree roots for minimum key



Deletion in a Binomial Heap

Deletemin() :

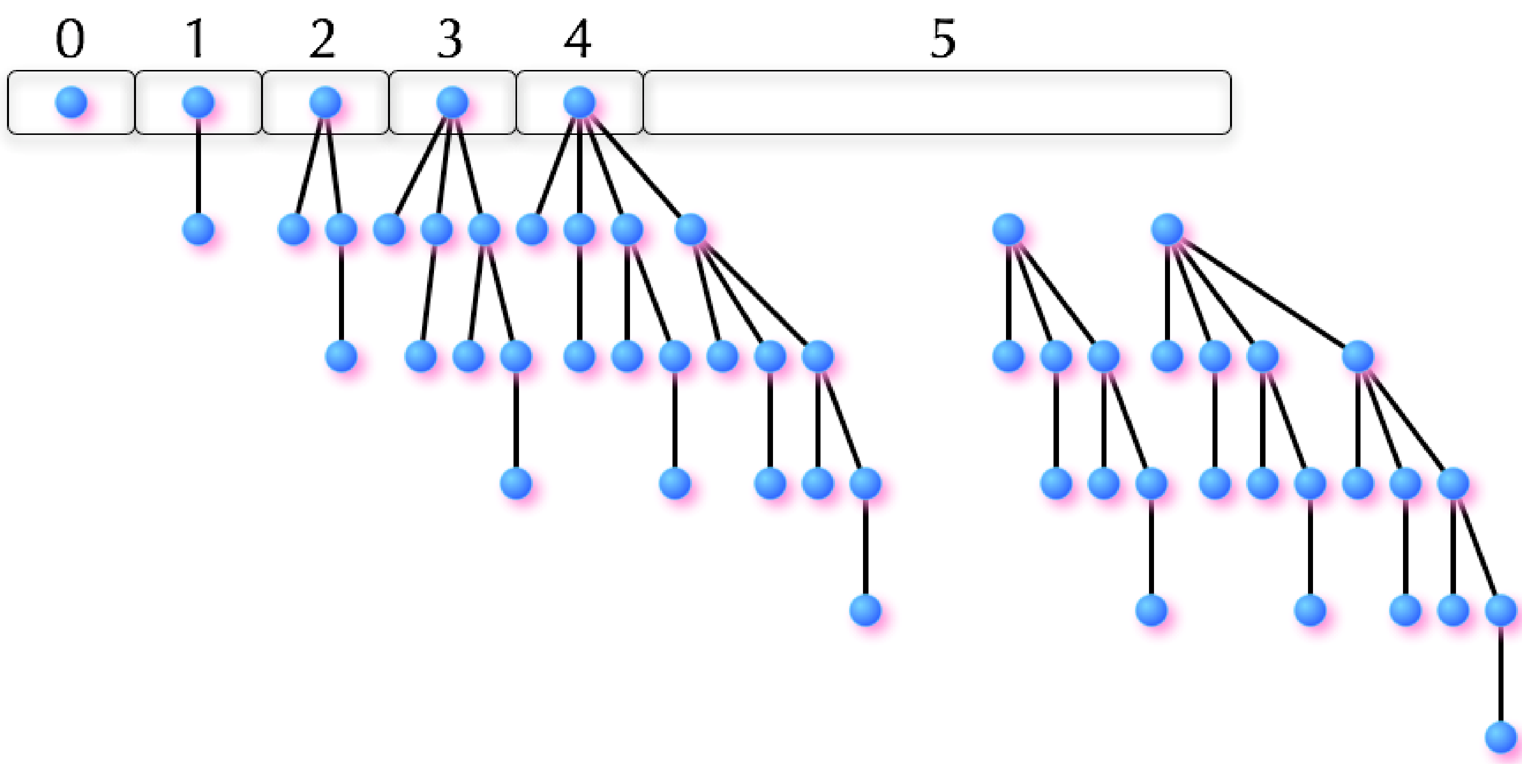
Delete this root and insert its children one at a time



Deletion in a Binomial Heap

Deletemin() :

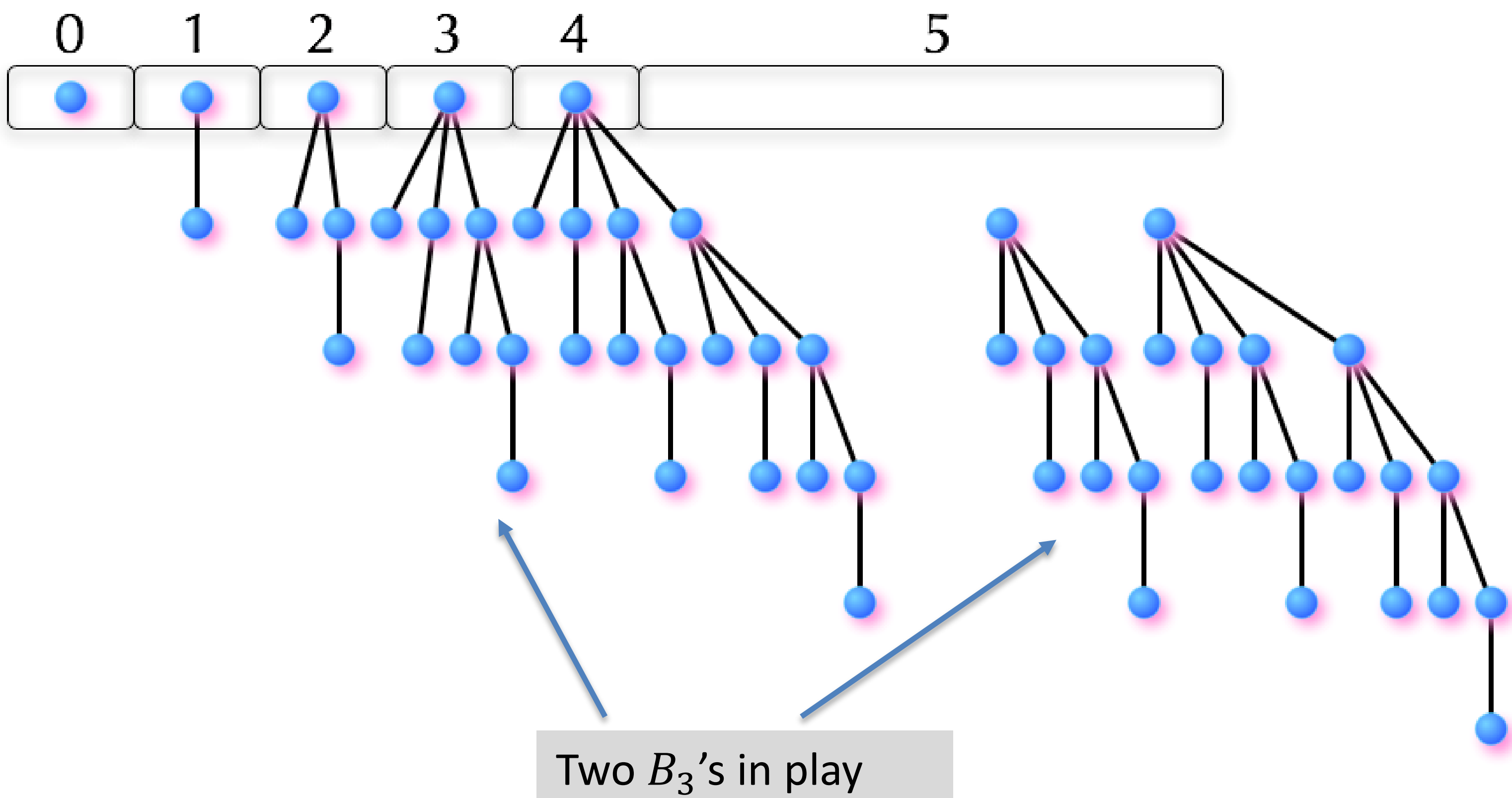
…after inserting children B_0, B_1, B_2



Deletion in a Binomial Heap

Deletemin() :

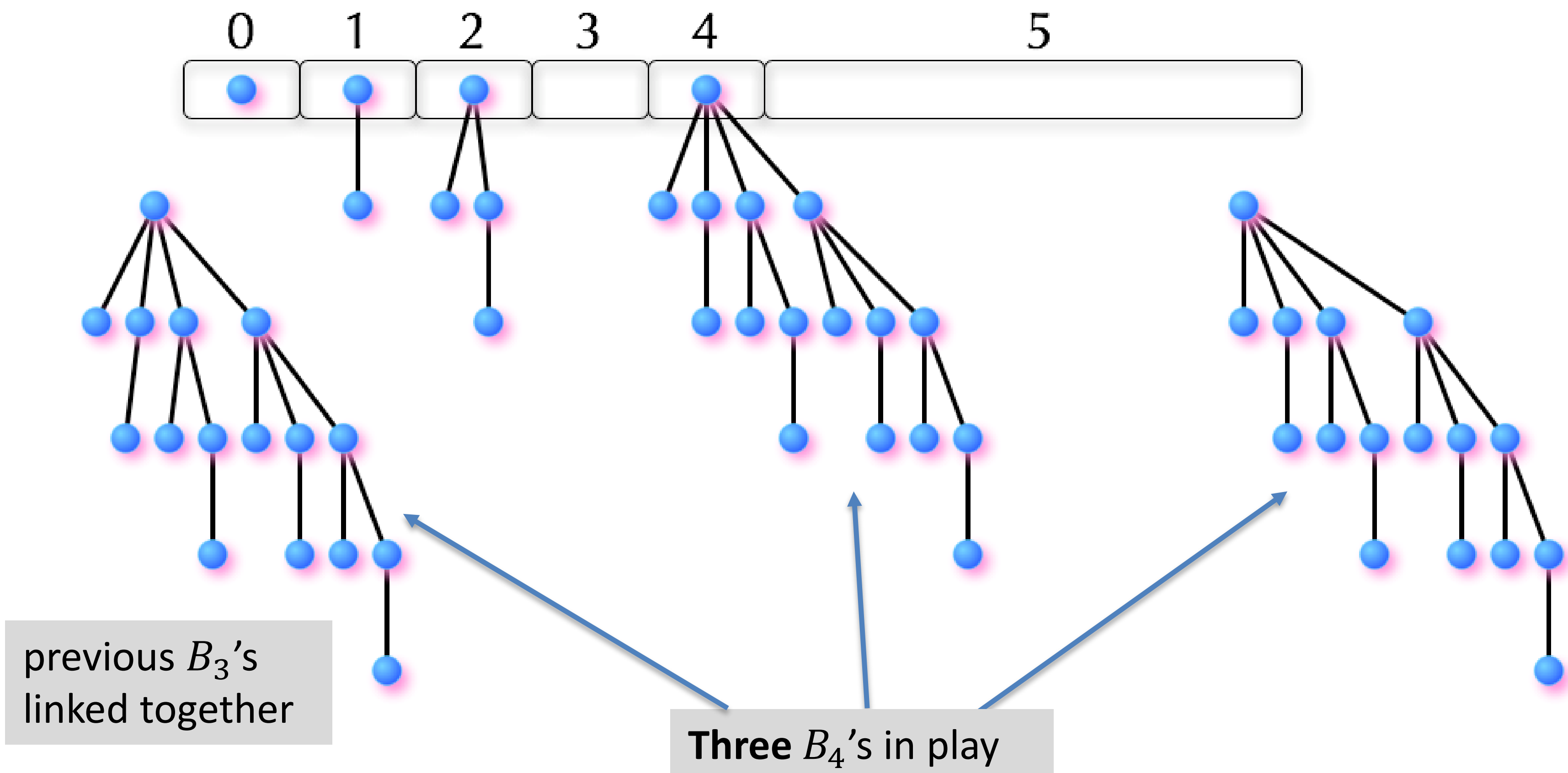
...after inserting children B_0, B_1, B_2



Deletion in a Binomial Heap

Deletemin() :

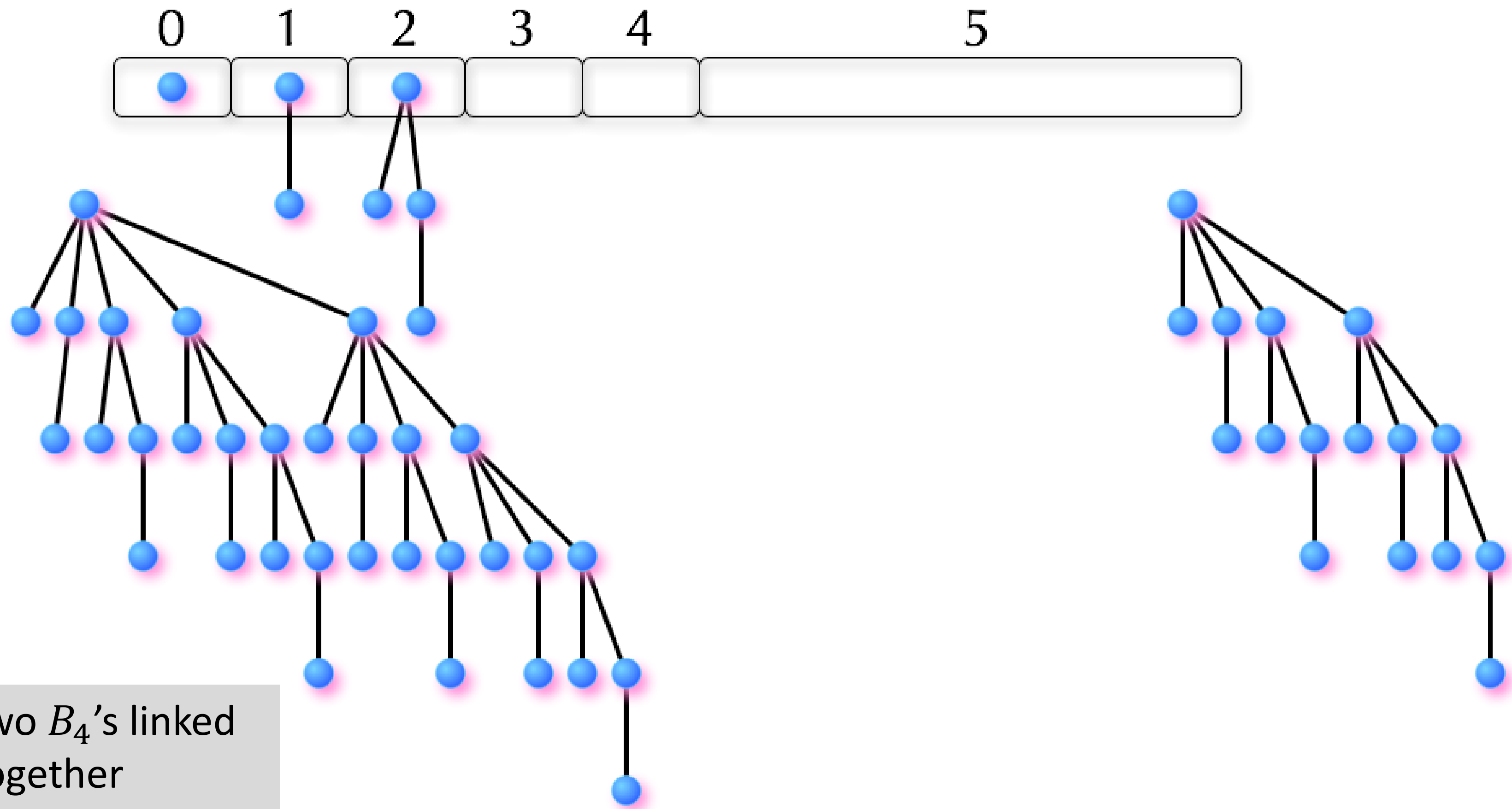
…after linking two B_3 trees



Deletion in a Binomial Heap

Deletemin() :

…after linking two B_3 trees…and two B_4 trees

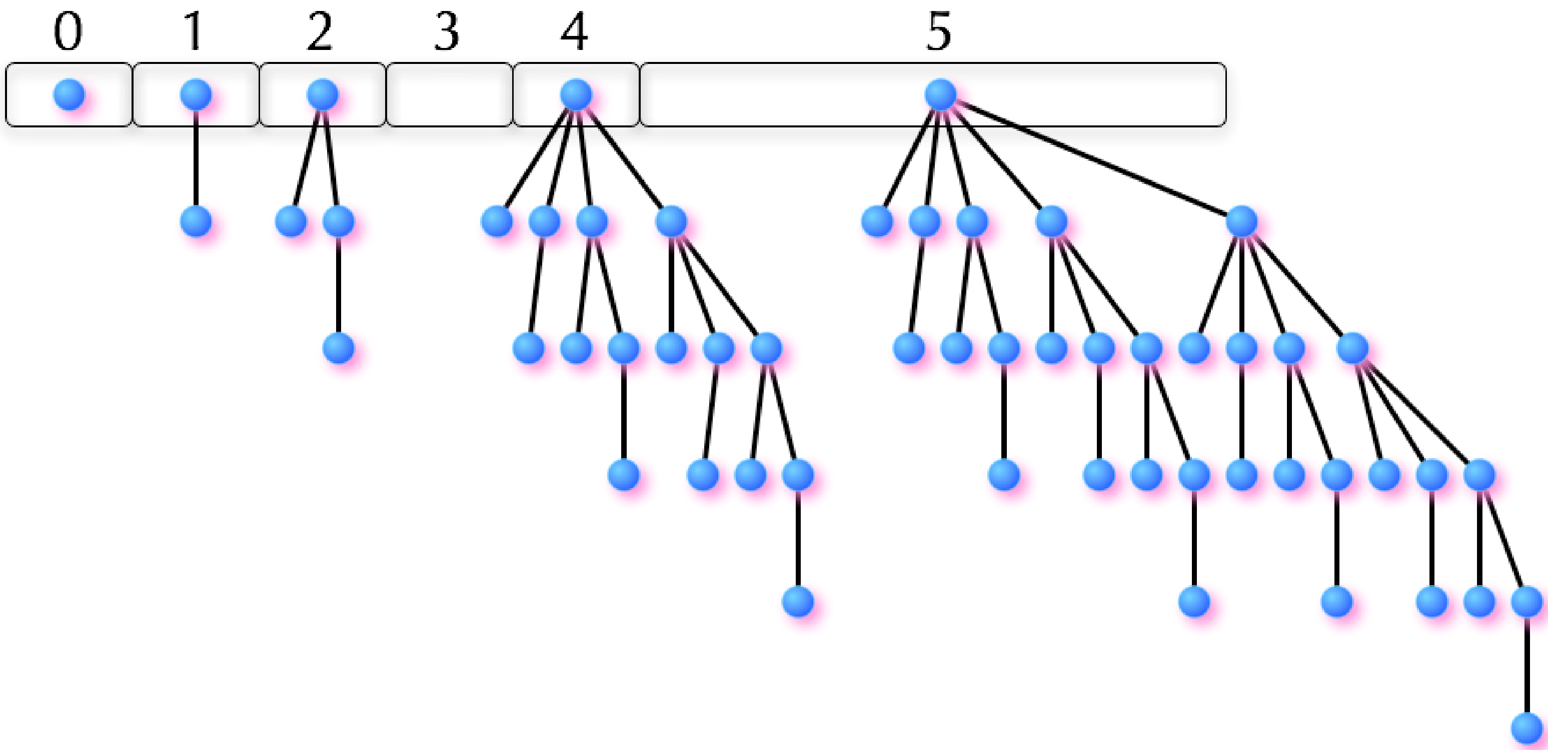


two B_4's linked together

Deletion in a Binomial Heap

Deletemin() :

No more than one tree per rank, so we're done.



Binomial *Heaps*

- Structure of a binomial heap:
  (1) A list of heap-ordered binomial trees
  (2) At most one tree of each rank
- Insert(X,k): $O(\log n)$ operations (# of trees)
- Deletemin():
  - Scan **roots** of trees in collection to find the one with min key
  - Remove and return min root **r**, insert children of **r**

Claim:
Deletemin takes $O(\log n)$ operations in the worst case.

- Up to $O(\log n)$ insertions, and one insertion can take $O(\log n)$ time! What?

Binomial *Heaps*

- Structure of a binomial heap:
  (1) A list of heap-ordered binomial trees
  (2) At most one tree of each rank
- Insert(X,k): $O(\log n)$ operations (# of trees)
- Deletemin():
  - Scan **roots** of trees in collection to find the one with min key
  - Remove and return min root **r**, insert children of **r**

Claim:
Deletemin takes $O(\log n)$ operations in the worst case.

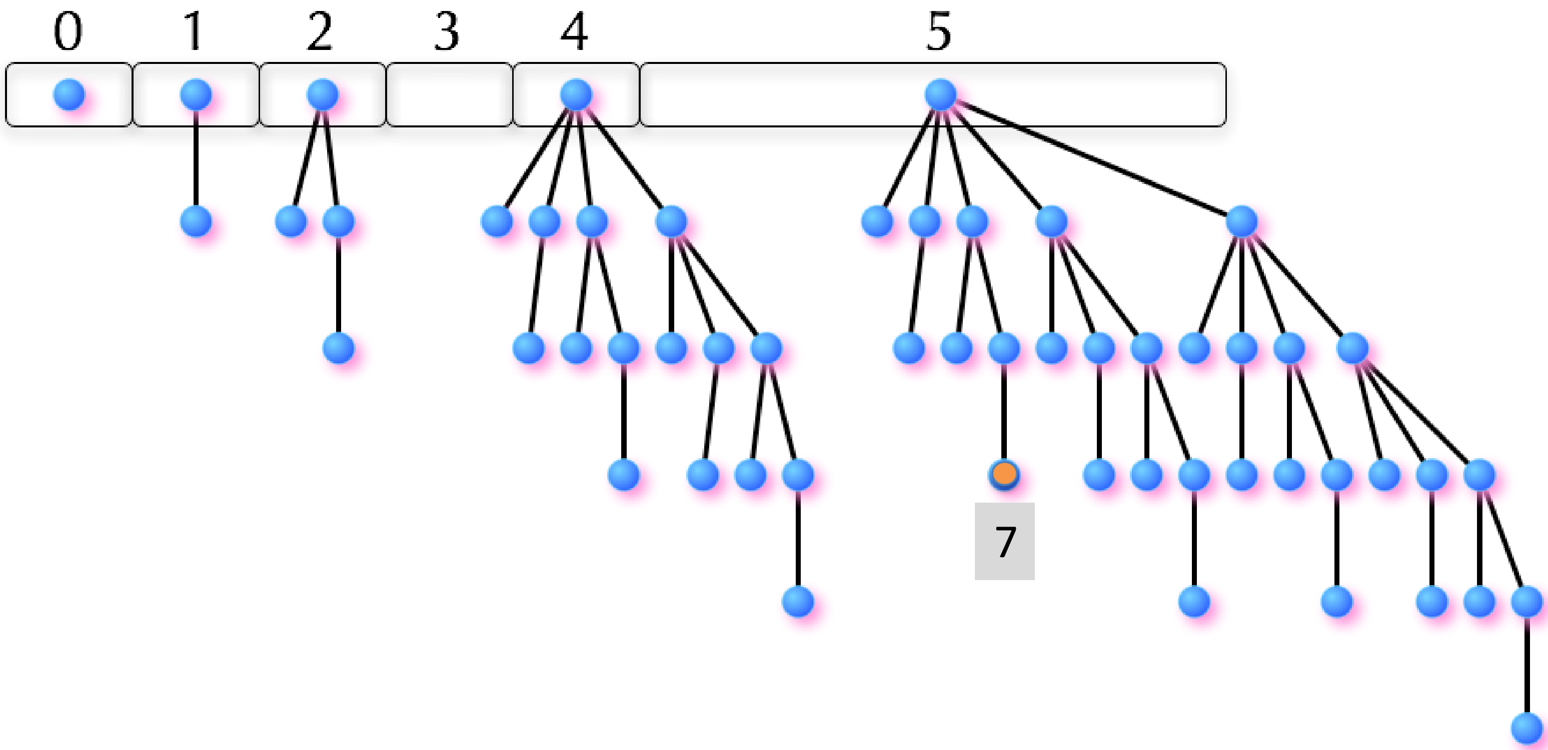
- Initially: n elements and $O(\log n) + 1$ trees in the collection
- One operation links two trees → one fewer in the collection
- Can only perform $O(\log n)$ links **across all the insertions**.

Binomial *Heaps*

- Structure of a binomial heap:
  (1) A list of heap-ordered binomial trees
  (2) At most one tree of each rank
- Insert(X,k): $O(\log n)$ operations (# of trees)
- Deletemin(): $O(\log n)$ operations (max # children)
- DecreaseKey(X,k):
  - (Can jump straight to the right key)
  - Decrease the value
  - Swap up the tree to fix the ordering

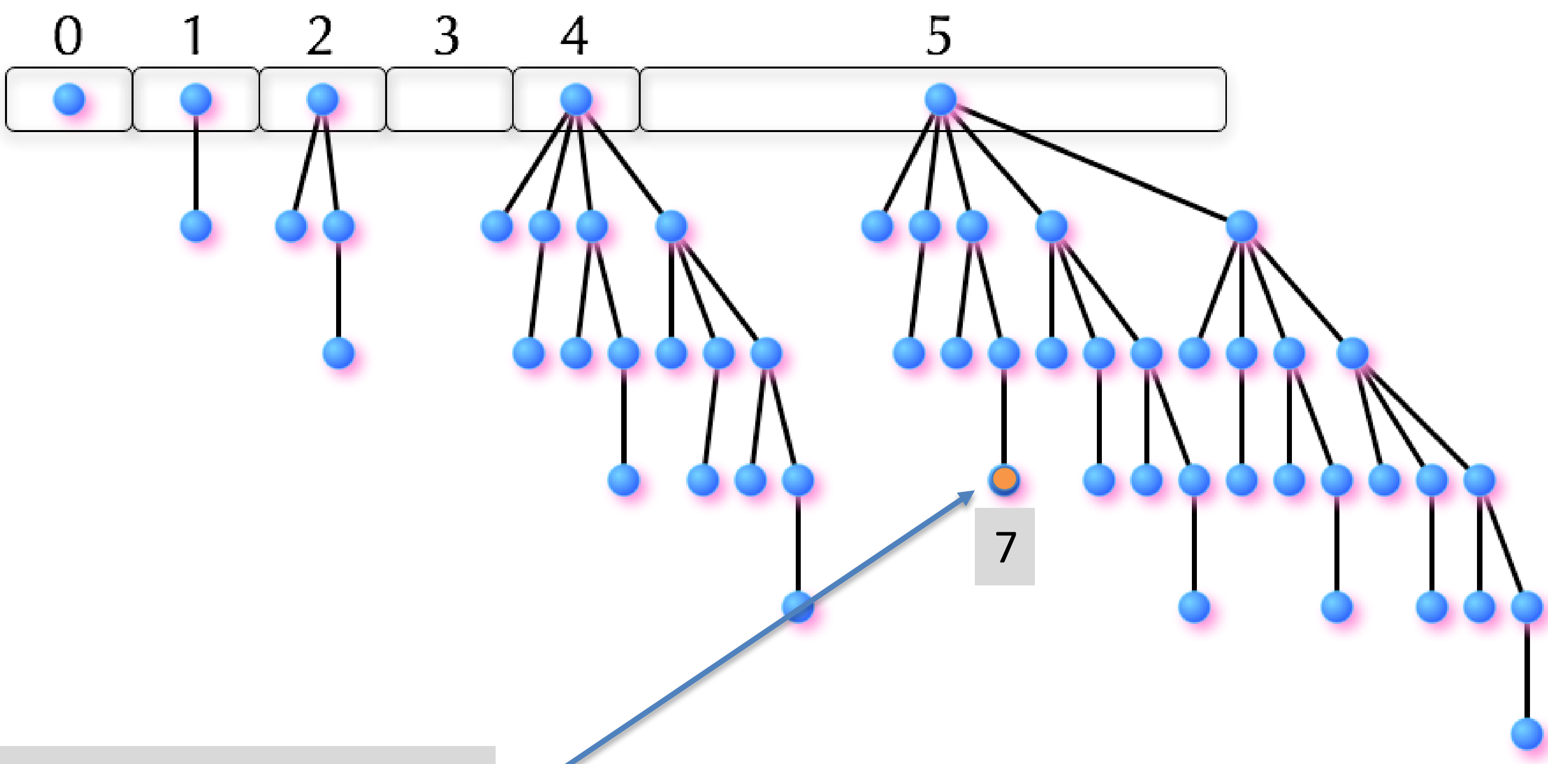
Decreasekey in a Binomial Heap

Decreasekey(**x**,4) :



Decreasekey in a Binomial Heap

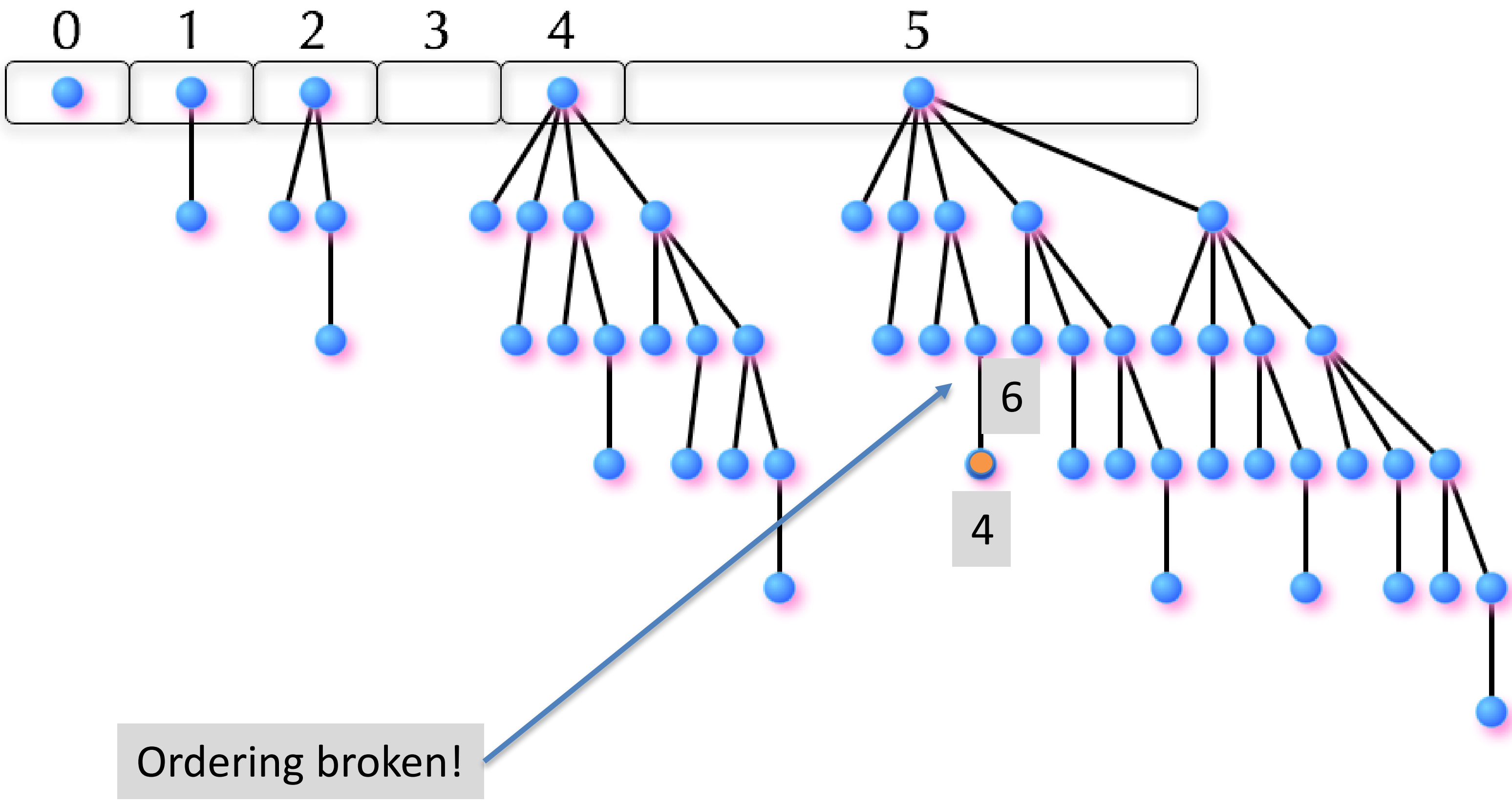
Decreasekey(**x**,4) :



(Find the node in O(1) operations by maintaining array of pointers)

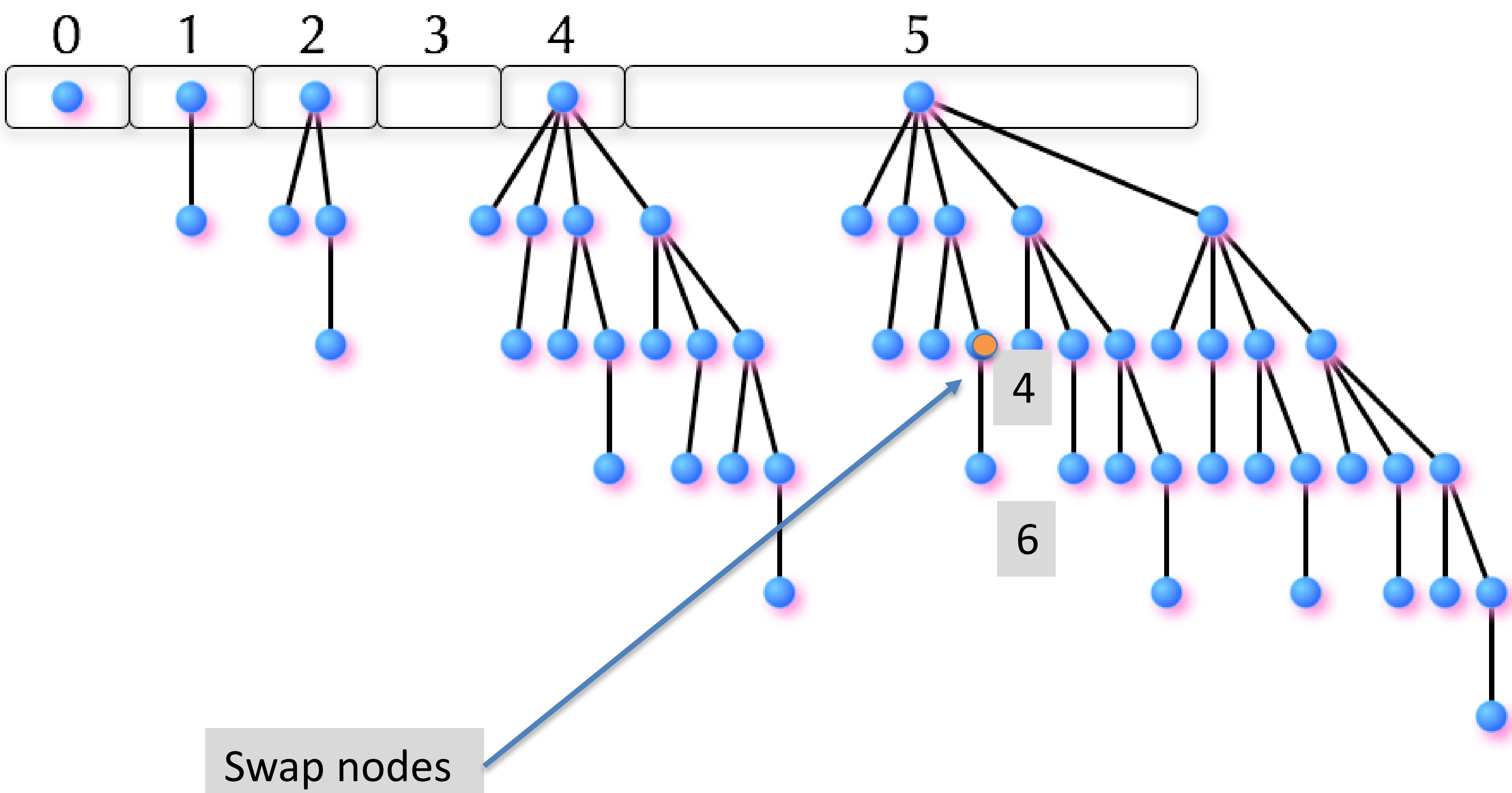
Decreasekey in a Binomial Heap

Decreasekey(**x**,4) :



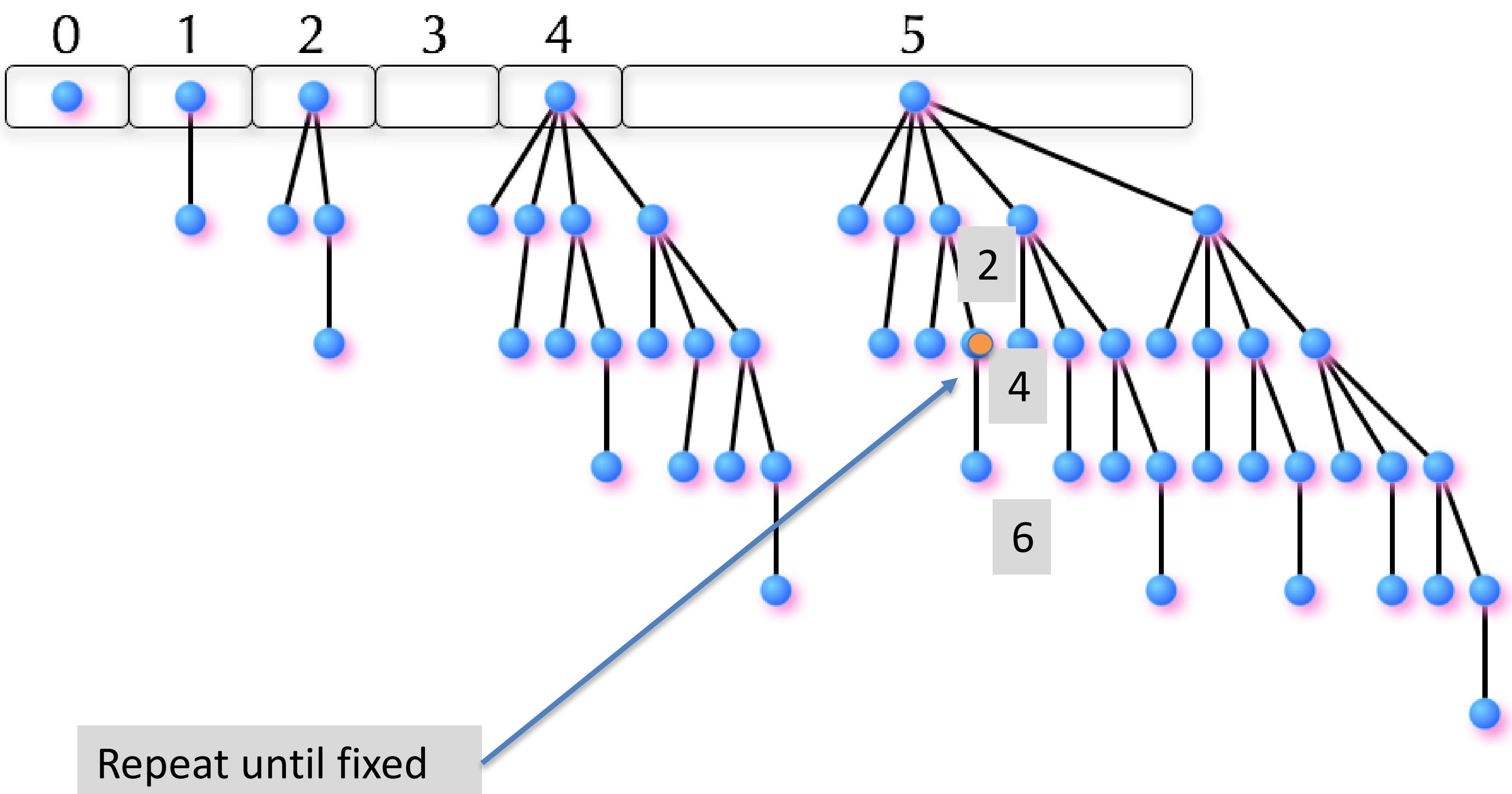
Decreasekey in a Binomial Heap

Decreasekey(**x**,4) :



Decreasekey in a Binomial Heap

Decreasekey(**x**,4) :

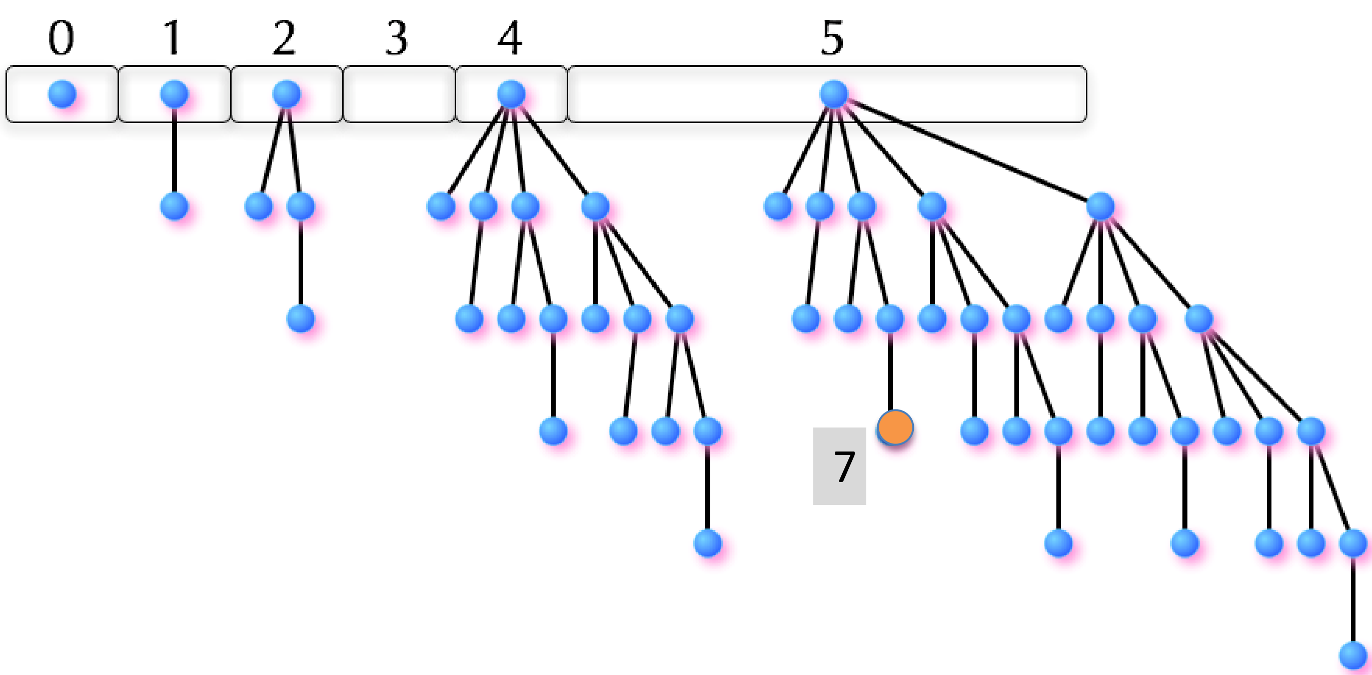


Binomial *Heaps*

- Structure of a binomial heap:
  (1) A list of heap-ordered binomial trees
  (2) At most one tree of each rank
- Insert(X,k): $O(\log n)$ operations (# of trees)
- Deletemin(): $O(\log n)$ operations (max # children)
- DecreaseKey(X,k): $O(\log n)$ operations (max tree height)

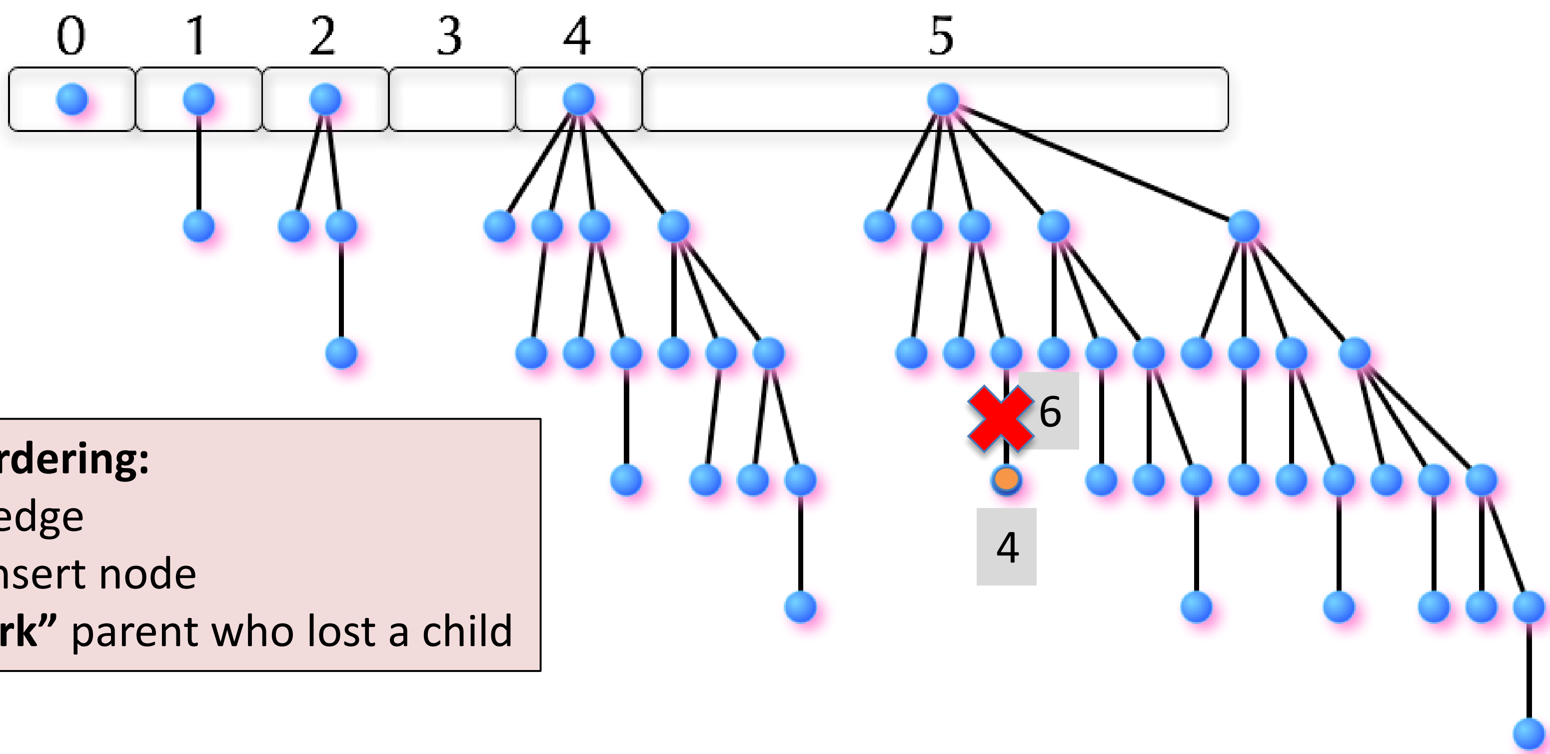
The Fibonacci Heap

- Like binomial heap, but not so picky about the tree structure
- Collection of heap ordered trees (not necessarily binomial)
- At most one tree per "rank" (definition will change)
- Insert & Deletemin same as in binomial heap
- **New DecreaseKey**



The Fibonacci Heap

- Like binomial heap, but not so picky about the tree structure
- Collection of heap ordered trees (not necessarily binomial)
- At most one tree per "rank" (definition will change)
- Insert & Deletemin same as in binomial heap
- New DecreaseKey

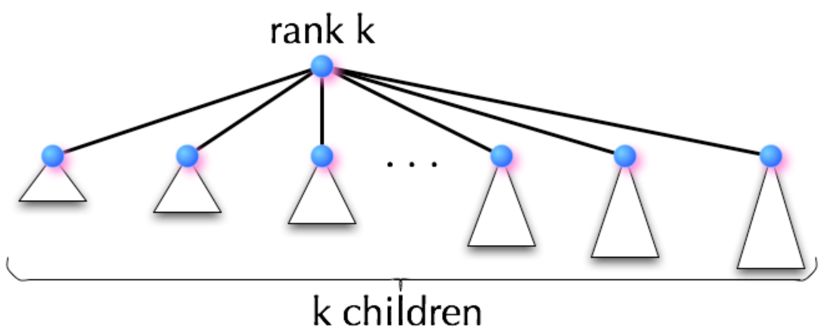


To fix ordering:

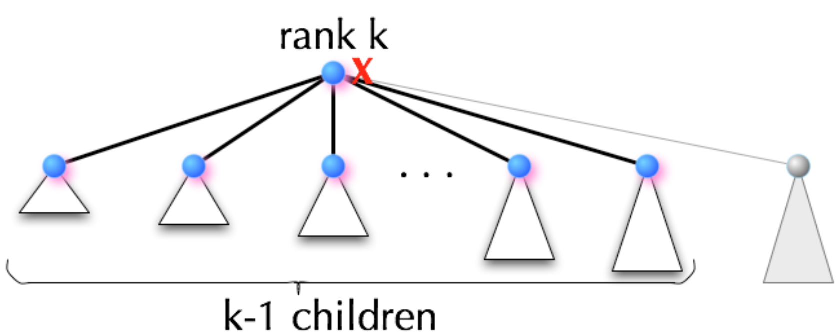
- Cut edge
- Re-insert node
- **"Mark"** parent who lost a child

Slightly different definition of "rank"

- In binomial tree "rank" ≡ "number of children"
- In Fibonacci heap a rank k node EITHER has k children



- OR: It has $k-1$ children and is marked (lost one)



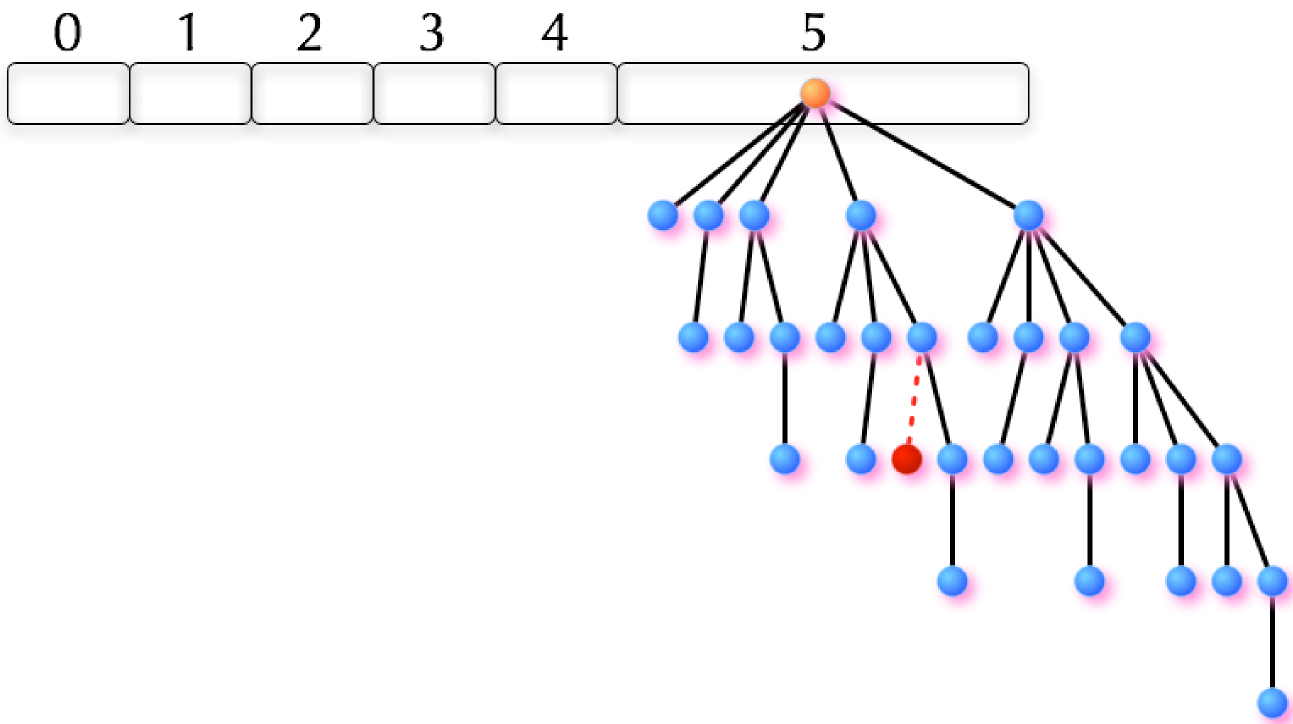
The Fibonacci Heap

- Like binomial heap, but not so picky about the tree structure
- Collection of heap ordered trees (not necessarily binomial)
- At most one tree per “rank” (definition will change)
- Insert & Deletemin same as in binomial heap

- Decreasekey(x,k) : ***cuts*** link from x to parent(x)
  - A node that has lost one child is ***marked***
  - A node that has lost two children is ***unmarked*** and **cuts *the link to its parent and reinserts***

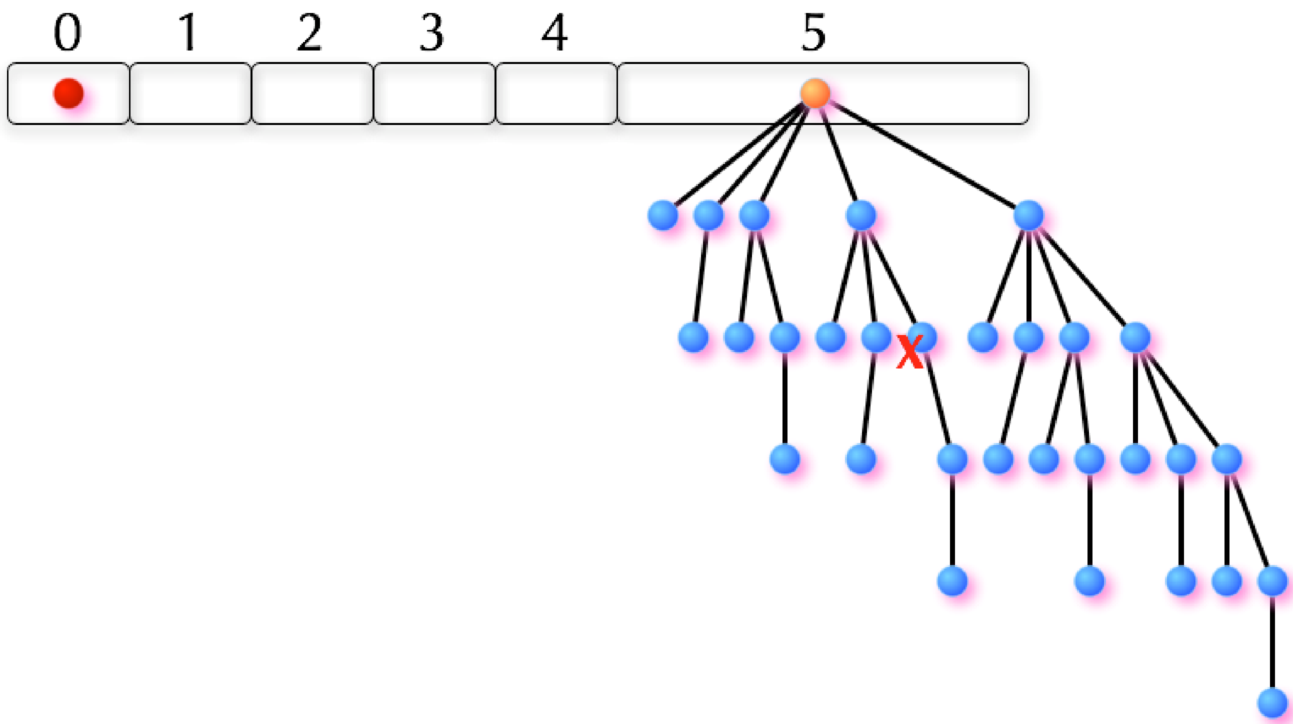
Decreasekey in action

- Cut link from node to its parent, mark the parent



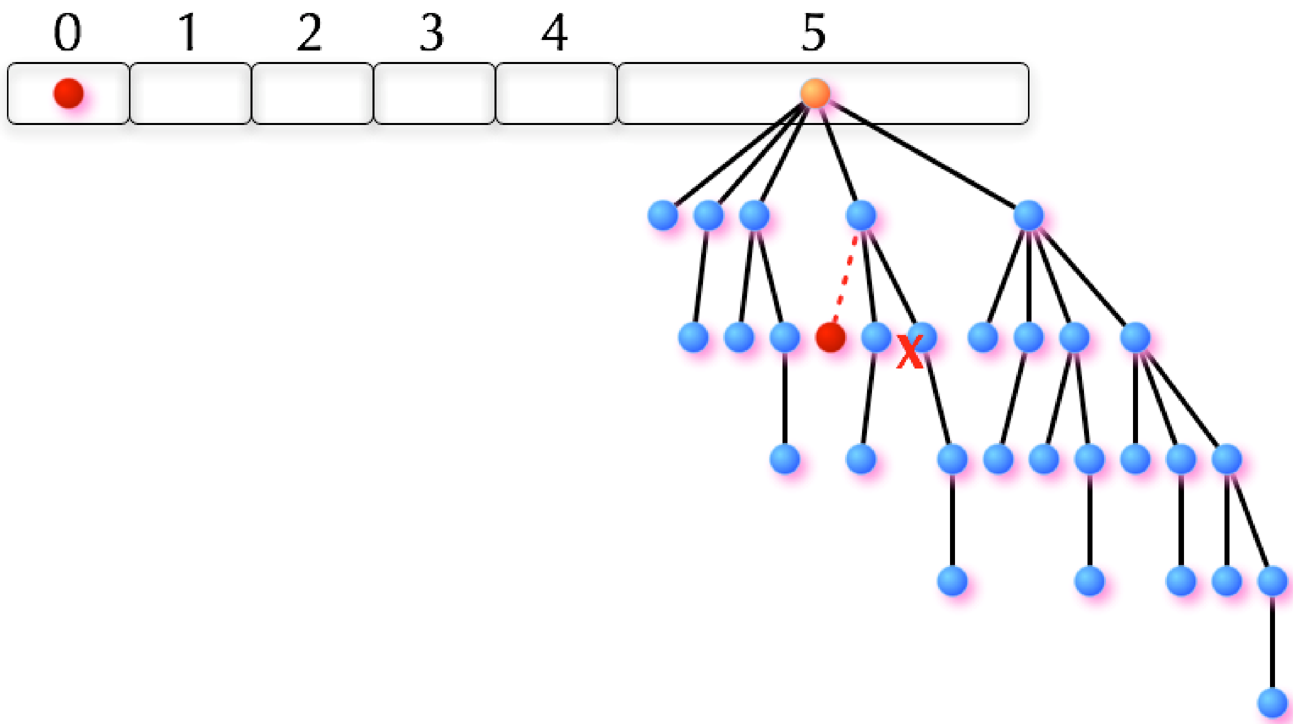
Decreasekey in action

- Cut link from node to its parent, mark the parent



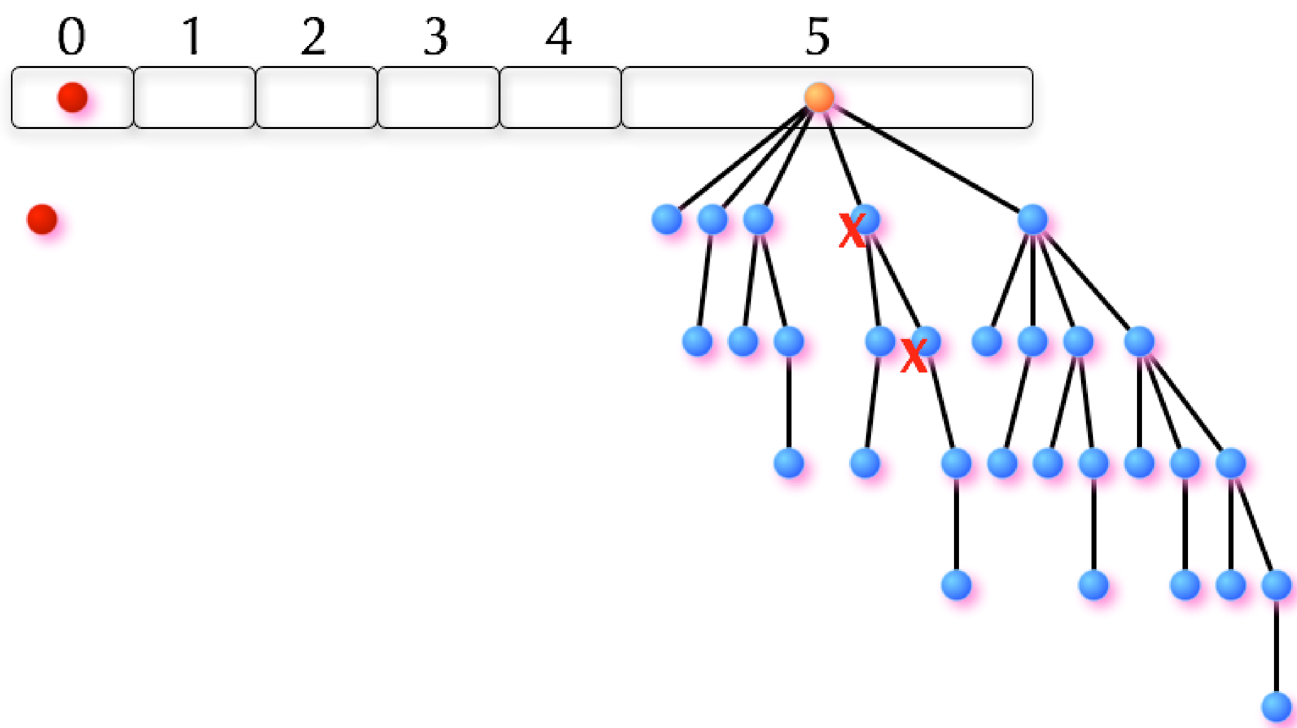
Decreasekey in action

- Cut link from node to its parent, mark the parent



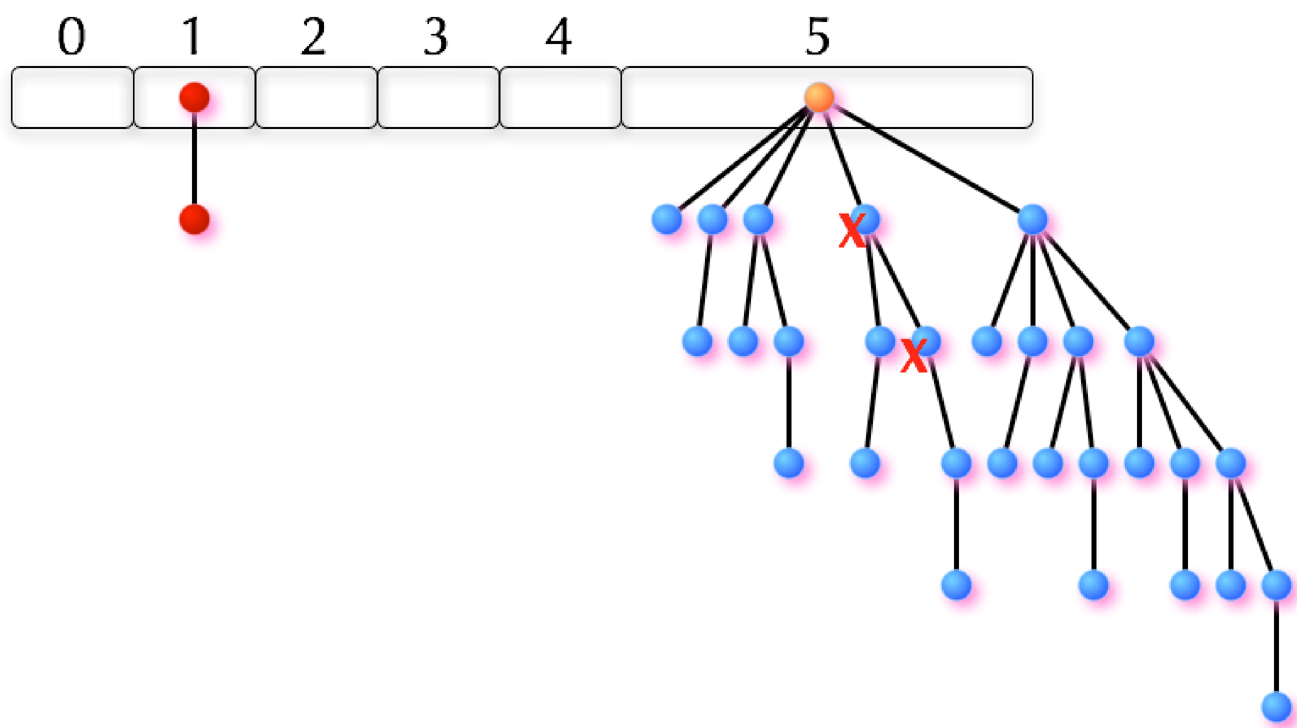
Decreasekey in action

- Too many rank 0 nodes, so link them



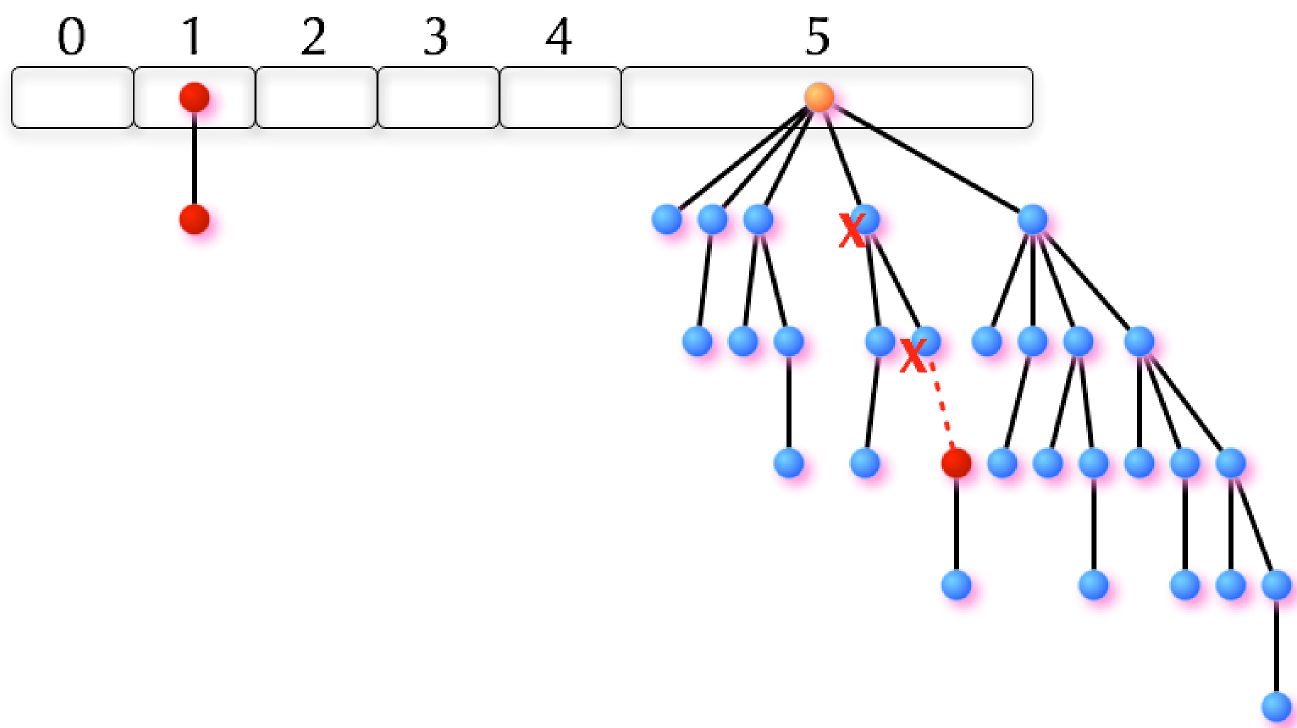
Decreasekey in action

- Too many rank 0 nodes, so link them



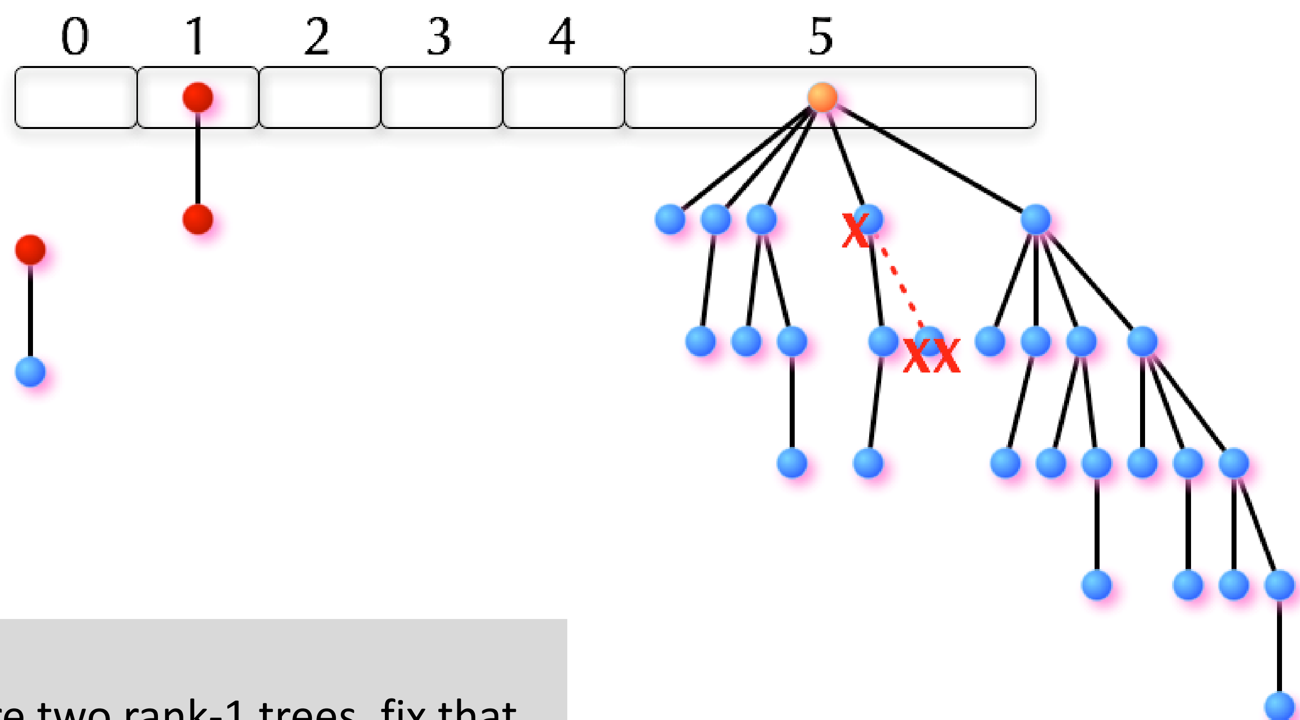
Decreasekey in action

- New decreasekey: cut link from node to its parent, mark the parent



Decreasekey in action

- Parent lost two children
- Cut it and set it to be *unmarked*

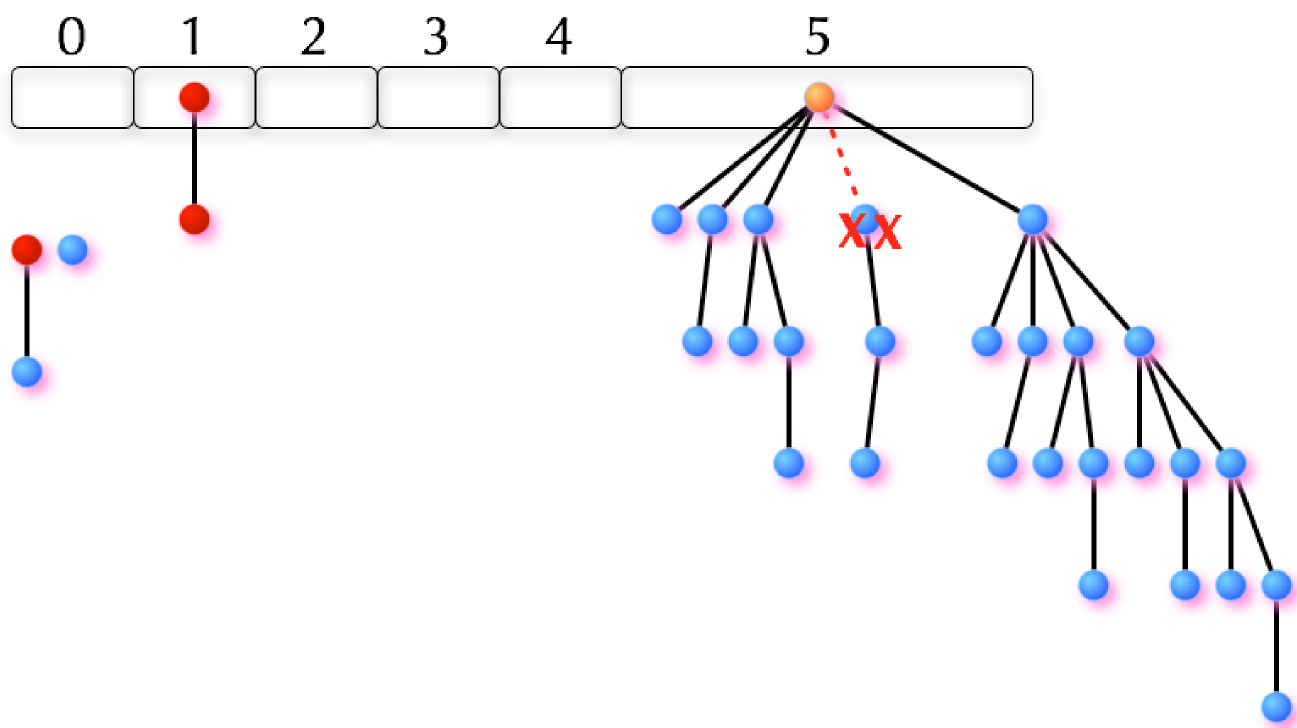


To-do list:

- There are two rank-1 trees, fix that
- A node is double-marked, reinsert it

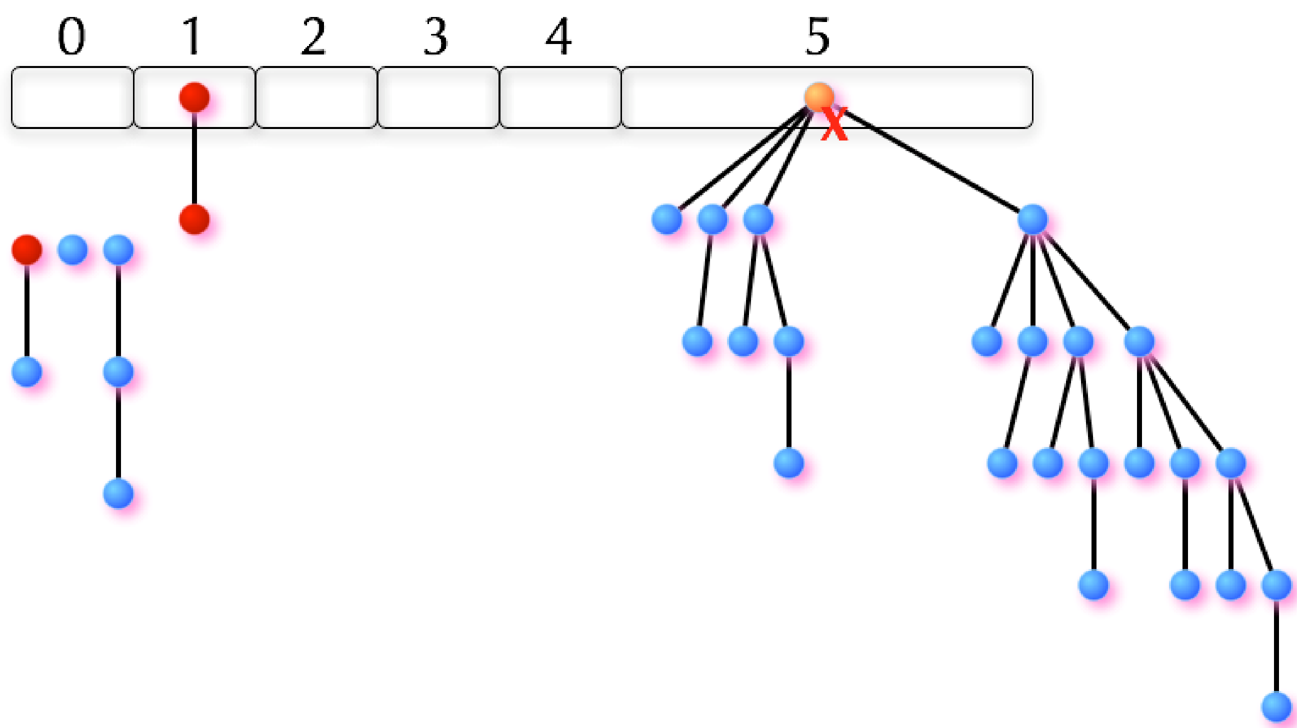
Decreasekey in action

- *Its parent* lost two children
- Cut it as well and set it to be unmarked



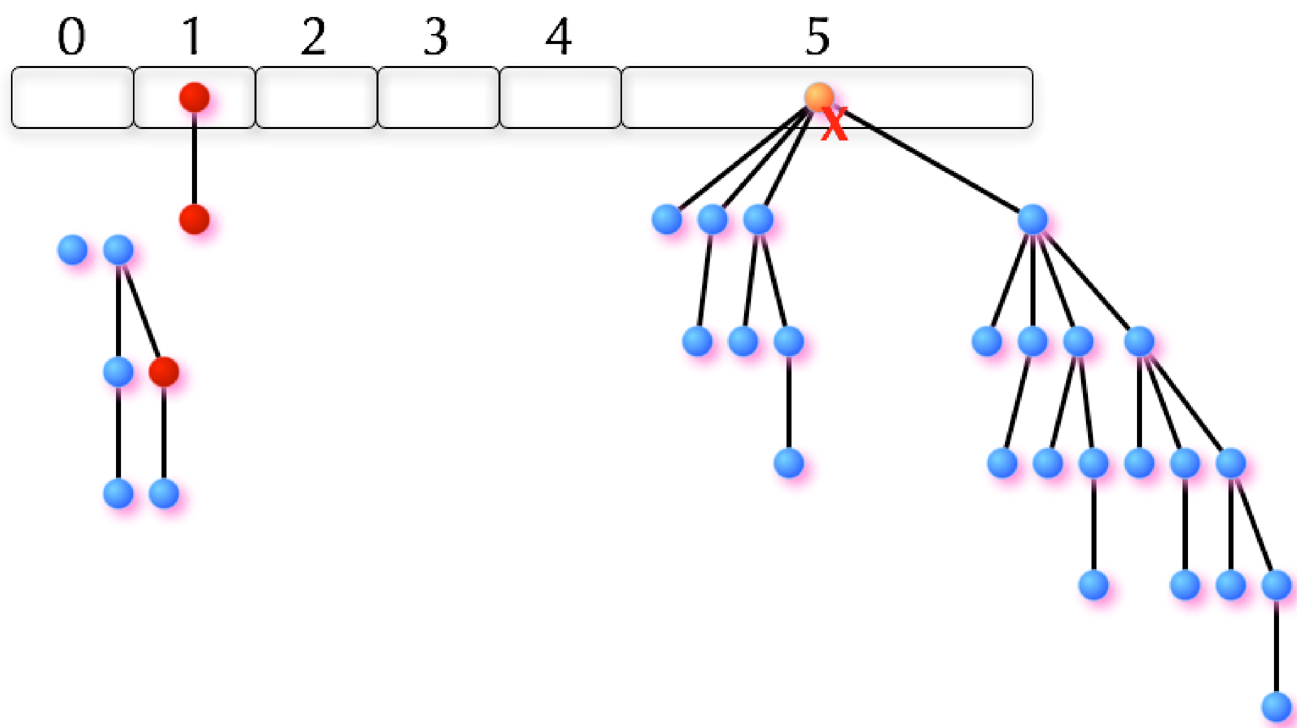
Decreasekey in action

- Too many rank 1 nodes, so link them



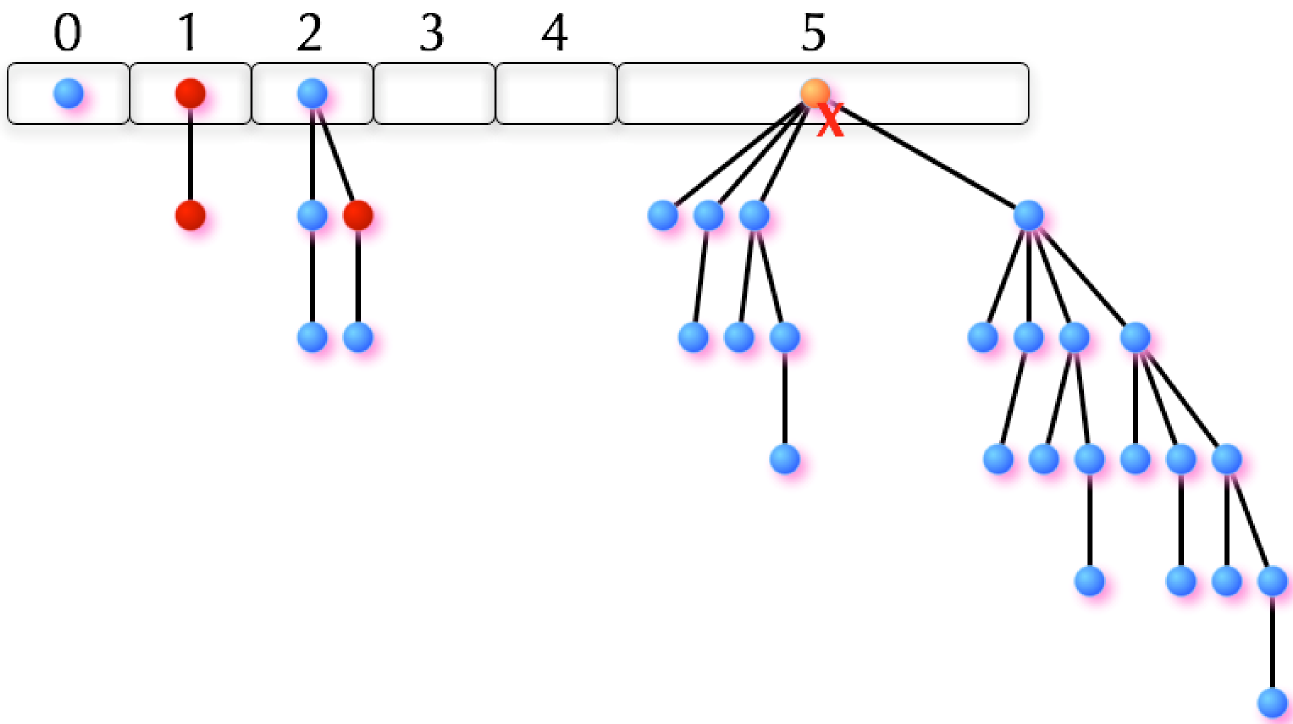
Decreasekey in action

- Too many rank 1 nodes, so link them



Decreasekey in action

- At most one tree per rank, so we're done.



The Challenge

- **Binomial Heaps:**
  - Insert, DeleteMin, DecreaseKey all cost $O(\log n)$ time in the worst case.

- **Fibonacci Heaps:**
  - No changes in Insert, DeleteMin
  - Aiming for $O(1)$ time DecreaseKey
  - DecreaseKey calls Insert
  - **How are we going to get O(1) Insert and DecreaseKey? . Are we doomed?**

The Challenge

- **Binomial Heaps:**
  - Insert, DeleteMin, DecreaseKey all cost $O(\log n)$ time **in the worst case.**

Need "average-case" analysis
- Repeated insertions/cuts can happen
- But very often?

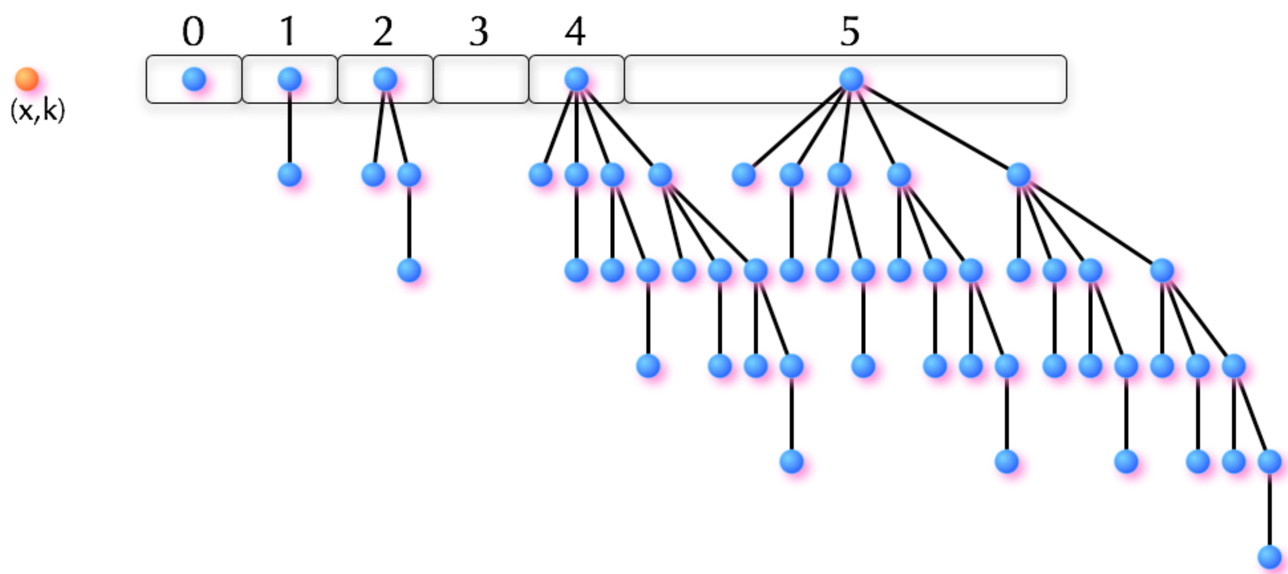
- **Fibonacci Heaps:**
  - No changes in Insert, DeleteMin
  - Aiming for $O(1)$ time DecreaseKey
  - DecreaseKey calls Insert
  - **How are we going to get O(1) Insert and DecreaseKey? .** **Are we doomed?**

Amortized Analysis

- **Statement:** Total running time is at most
  - O((# insert) + (# decreasekey) + log n*(# deletemin)).
  - *On average*, insert, decreasekey run in time O(1) and deletemin runs in time O(log n).
- **Strategy:** Maintain a "potential" ϕ in the analysis.
  - Some function of the current state of the heap.
  - Let c_i = actual running time of i-th operation.
  - Let D_i = Fibonnaci heap after the i-th operation.
  - Let $\widehat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1})$.
  - Total running time $\sum_{i=1}^{n} c_i = (\sum_{i=1}^{n} \widehat{c_i}\ ) - \phi(D_n) + \phi(D_0)$
  - So, if $\phi(D_0) = 0$ and $\phi(D_n) \geq 0$, $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \widehat{c_i}$
    - If $\widehat{c_i} \leq O(1)$ whenever i is insert/decreasekey and $\widehat{c_i} \leq O(\log n)$ when i is deletemin, we prove what we want!

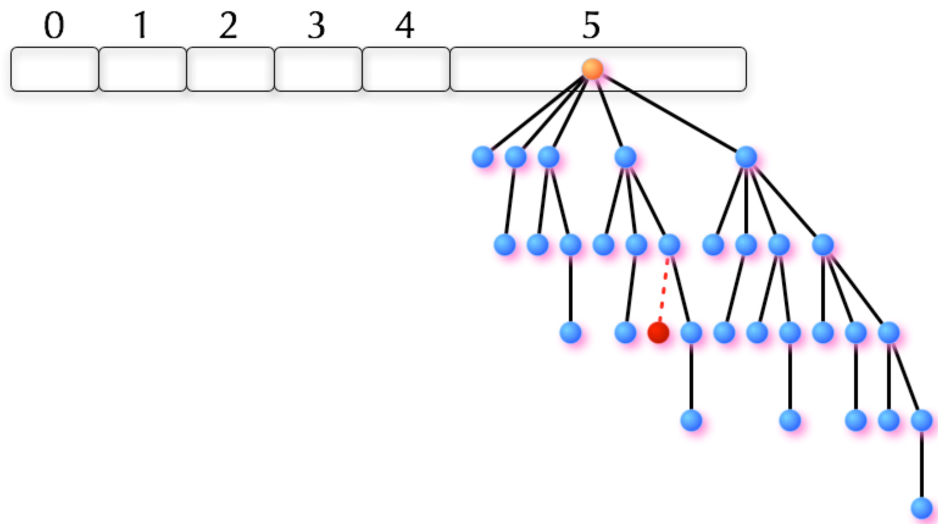
Our Potential Function

- **Insert:**
  - O(1) work to link two trees, reducing # of trees by 1
  - Might do this repeatedly…
  - But that's okay if **potential depends on the # of trees**



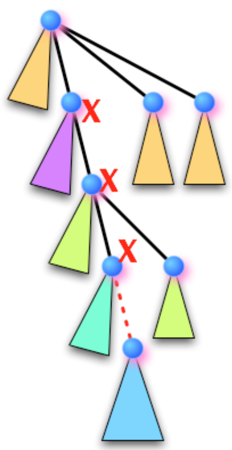
Our Potential Function

- **DecreaseKey:**
  - O(1) work to cut
  - Might have to do this repeatedly…
  - Last cut in the chain:
    - **Increases** the number of trees by 1
    - **Increases** the number of marked nodes by 1



Our Potential Function

- **DecreaseKey:**
  - O(1) work to cut
  - Might have to do this repeatedly…
  - All other cuts in the chain:
    - **Increase** the number of trees by 1
    - **Decrease** the number of marked nodes by 1

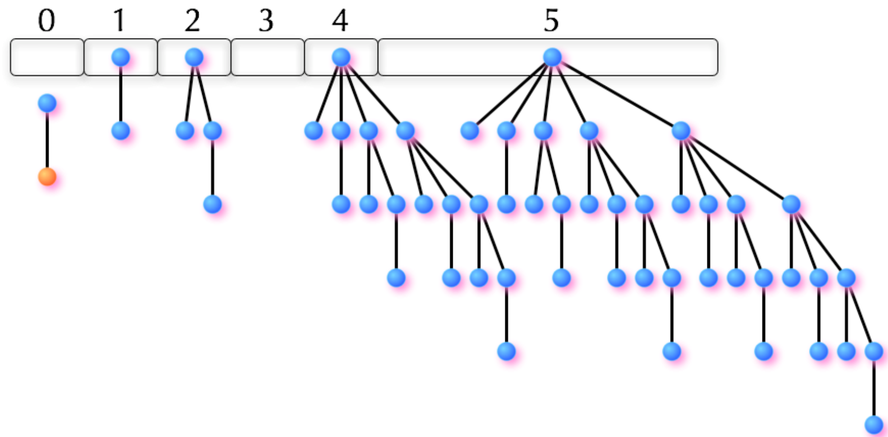


Our Potential Function

ϕ **=2(# marked nodes) + (# trees)**

Amortized Cost

- Insert
  - $c_i = 1 + \#links$
  - $\Delta\phi = 1 - \#links$
  - $\widehat{c_i} = 2$

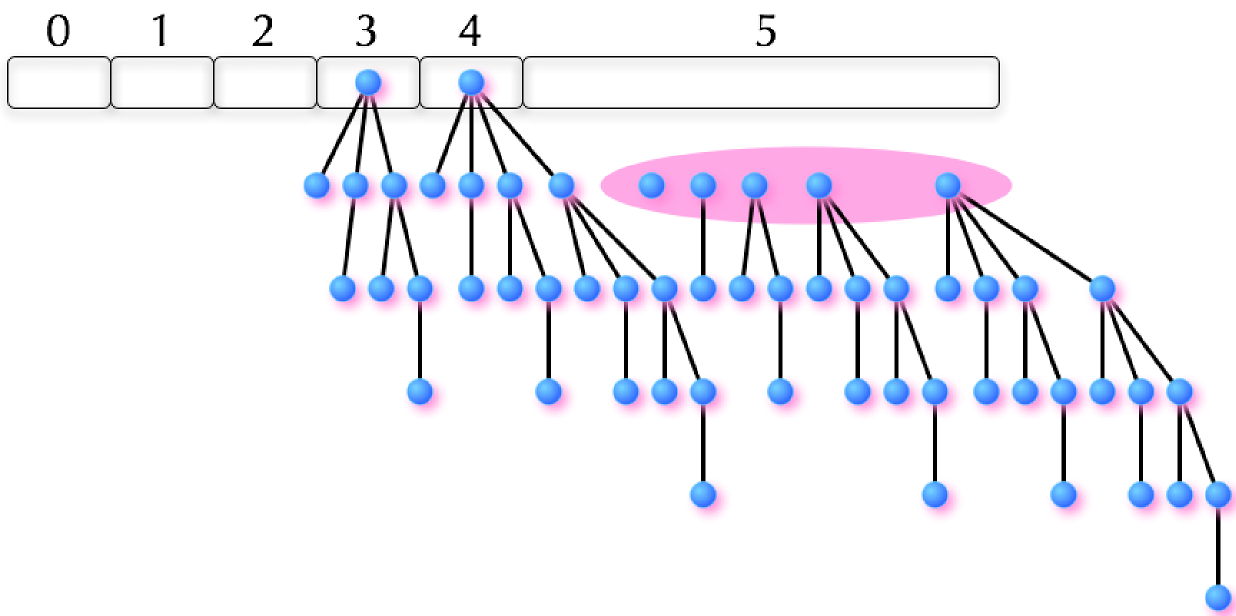


- If you want to more precise bound $c_i \leq a(1 + \#links)$ for some constant a, define
  - $\phi = a \cdot$ (2(# marked nodes) + (# trees))
  - Everything will be multiplied by a, but fine.

Amortized Cost

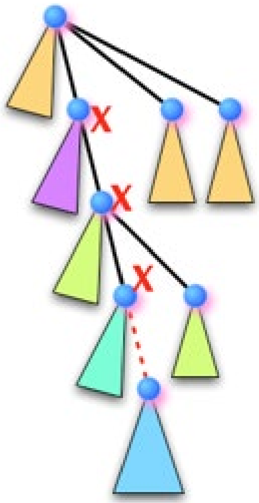
- Deletemin
  - $c_i = \#trees + \#children + \#links$
  - $\Delta\phi = \#children - 1 - \#links$
  - $\hat{c_i} \leq \#trees + 2 \cdot \#children \leq O(\log n)$

Lemma] Tree of rank i has at least i-th Fibonnaci number = $\Omega(1.682^i)$.



Amortized Cost

- Decreasekey
  - $c_i = 1 + \#cuts + \#links$
  - Last cut in the chain
    - Increase #trees by 1
    - Increase #markednodes by 1
  - Every other cut in chain
    - Increase #trees by 1
    - Decrease #markednodes by 1
  - $\Delta\phi = 3 - (\#cuts - 1) - \#links$
  - $\widehat{c_i} = c_i + \Delta\phi = 5.$



Amortized Analysis

- **Statement:** Total running time of Fibonacci Heaps is at most
  - O((# insert) + (# decreasekey) + log n*(# deletemin)).
  - *On average*, insert, decreasekey run in time O(1) and deletemin runs in time O(log n).

Dijkstra with Fibonacci heaps

- Input: directed graph G with positive lengths, source s
- $\mathrm{dist}[s] = 0$, $\mathrm{dist}[v] = \infty$ for all $v \neq s$, $\mathrm{prev}[s] = \emptyset$. $A = \emptyset$.
- For every $v \in V$
  - **Insert($\boldsymbol{Q, v, dist}[\boldsymbol{v}]$)**

Insert $O(n)$ times
$O(n)$ operations **total**

- While $A \neq V$
  - $\boldsymbol{v = Deletemin(Q)}$
  - $A \leftarrow A \cup \{v\}$.
  - For every $(v, w) \in E$
    - If $\mathrm{dist}[w] > \mathrm{dist}[v] + \ell(v, w)$
      - $\mathrm{dist}[w] = \mathrm{dist}[w] + \ell(v, w), \mathrm{prev}[w] = v$.
      - **Decreasekey(Q, $\boldsymbol{w, dist}[\boldsymbol{w}]$).**

Deletemin $O(n)$ times
$O(n \log n)$ operations

Decreasekey $O(m)$ times
$O(m)$ operations **total**

Total: $O(m + n \log n)$ operations

Wrapup

- Upshot: Dijkstra takes $O(m + n \log n)$ time.
- Choosing a good potential function can be tricky!