Fundamental Algorithms

Chapter 2: Advanced Heaps

Sevag Gharibian

(based on slides of Christian Scheideler)

WS 2019

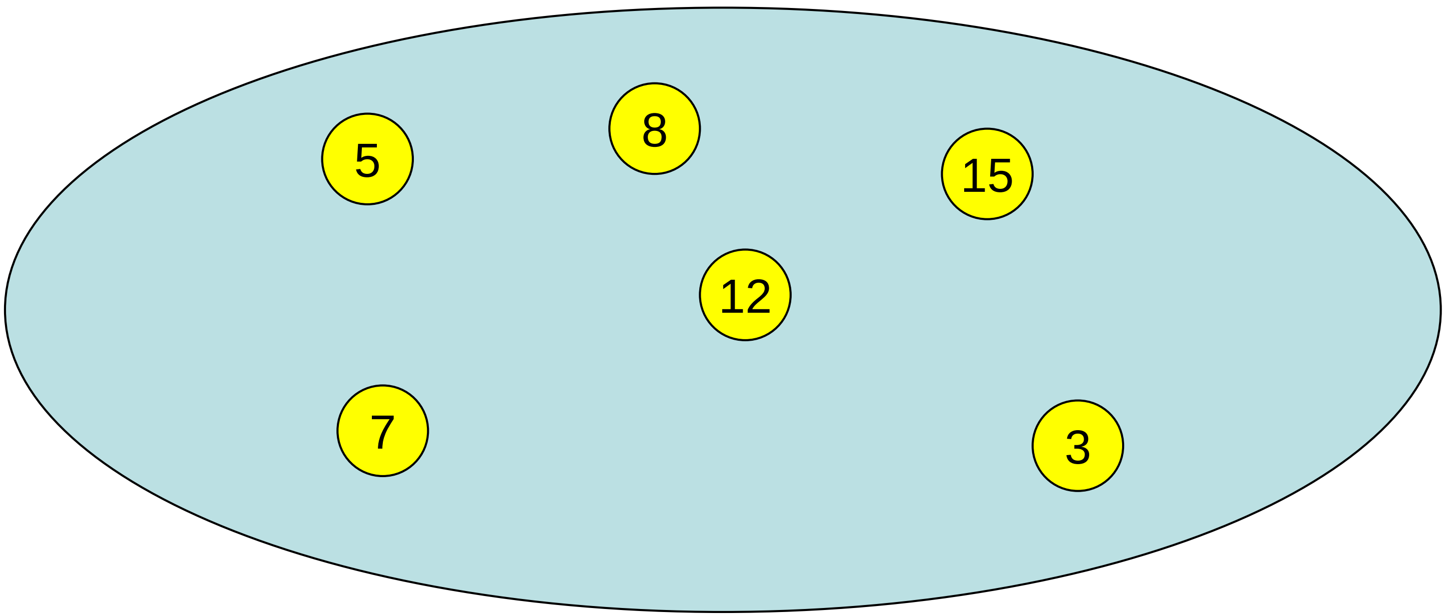
Contents

A heap implements a priority queue.

We will consider the following heaps:

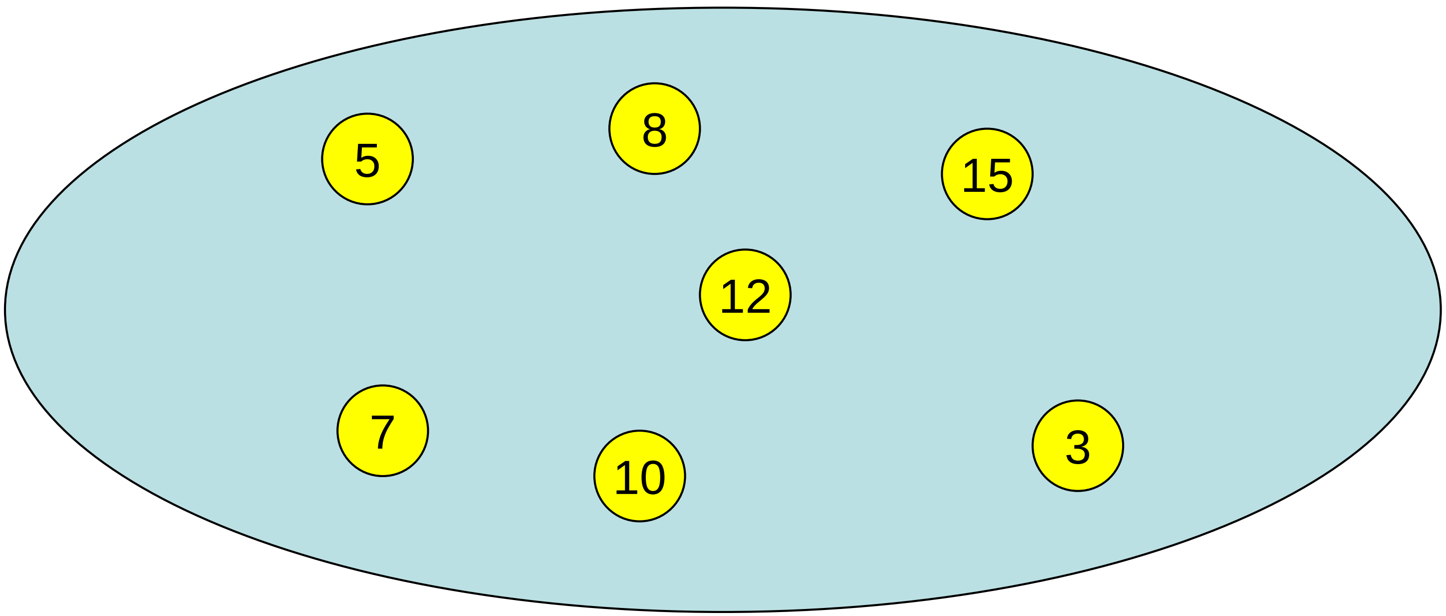
- Binomial heap
- Fibonacci heap
- Radix heap

Priority Queue



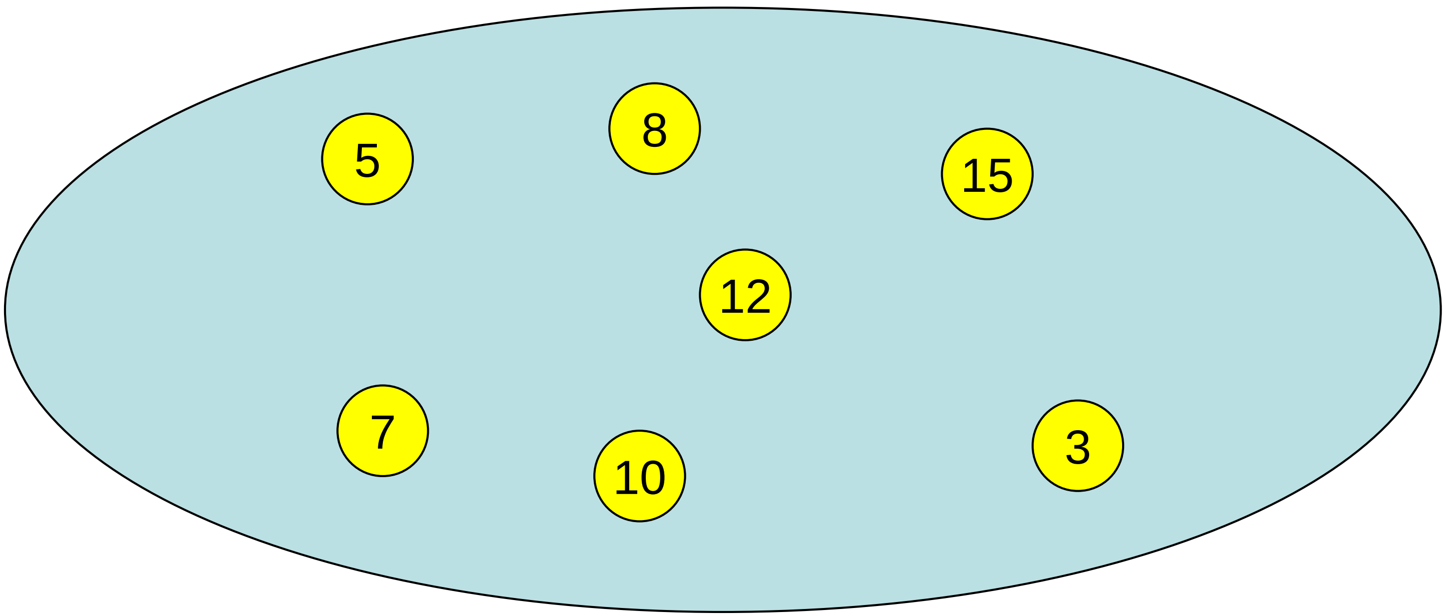
Priority Queue

insert(10)



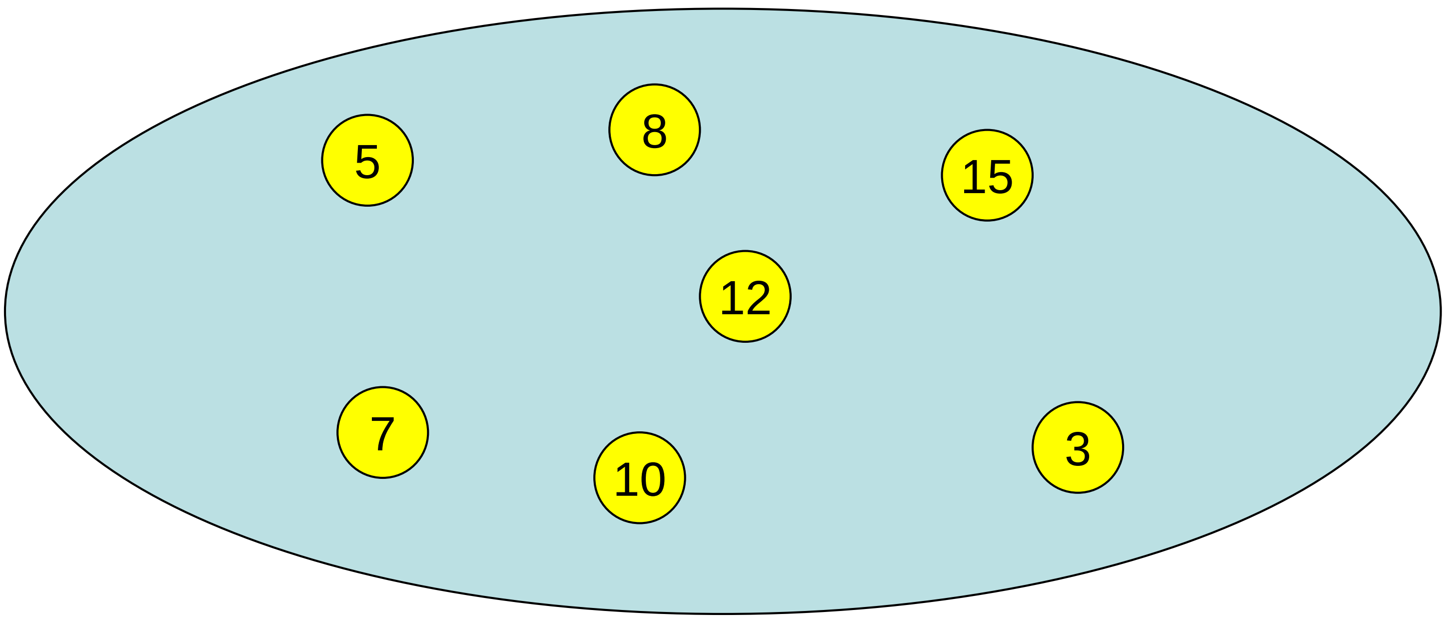
Priority Queue

min() outputs 3 (minimal element)



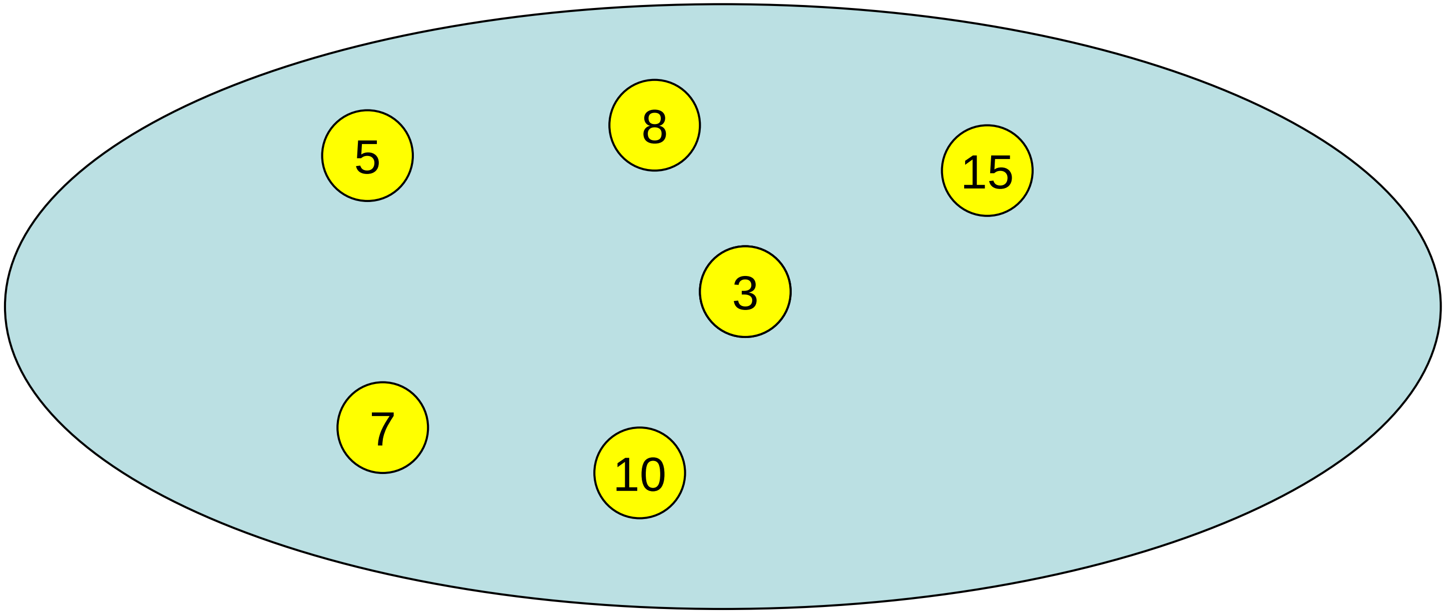
Priority Queue

deleteMin()



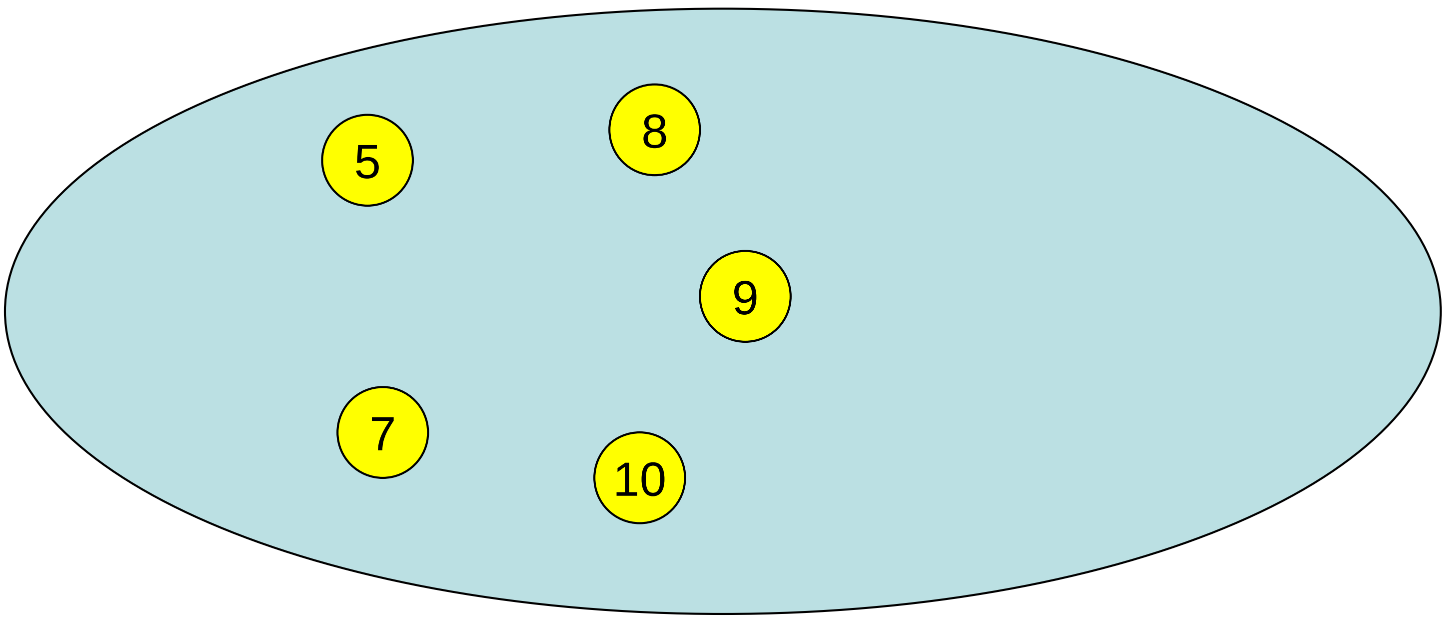
Priority Queue

decreaseKey(12,9) (note: 9 is the offset)



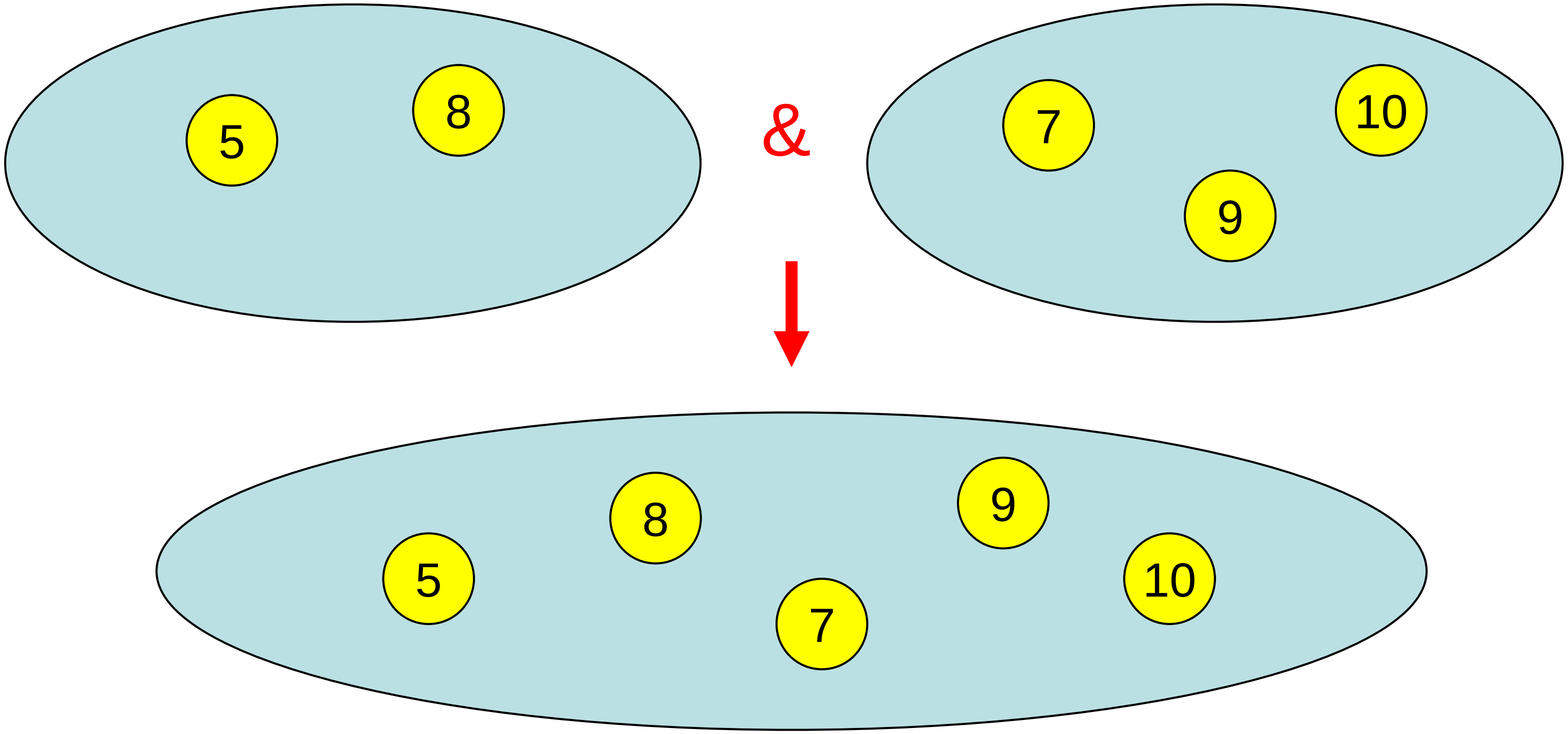
Priority Queue

delete(8)



Priority Queue

merge(Q,Q')



Priority Queue

M: set of elements in priority queue

Every element e identified by key(e).

Operations:

- M.build($\{e_1,\ldots,e_n\}$): $M:=\{e_1,\ldots,e_n\}$
- M.insert(e: Element): $M:=M\cup\{e\}$
- M.min: outputs $e\in M$ with minimal key(e)
- M.deleteMin: like M.min, but additionally $M:=M\setminus\{e\}$, for that e with minimal key(e)

Extended Priority Queue

Additional operations:

- M.delete(e: Element): M:=M\{e}
- M.decreaseKey(e:Element, Δ): key(e):=key(e)-Δ
- M.merge(M´): M:=M∪M´

Note: in delete and decreaseKey we have direct access to the corresponding element and therefore do not have to search for it.

Why Priority Queues?

- Sorting: Heapsort
- Shortest paths: Dijkstra´s algorithm
- Minimum spanning trees: Prim´s algorithm
- Job scheduling: EDF (earliest deadline first)

Why Priority Queues?

Problem from the ACM International Collegiate Programming Contest:

- A number whose only prime factors are 2,3,5 or 7 is called a humble number. The sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 24, 25, 27, ... shows the first 20 humble numbers.
- Write a program to find and print the n-th element in this sequence

Solution: use priority queue to systematically generate all humble numbers, starting with queue just containing 1. Repeatedly do:

- x:=M.deleteMin
- M.insert(2x); M.insert(3x); M.insert(5x), M.insert(7x) (assumption: only inserts element if not already in queue)

Priority Queue

- Priority Queue based on unsorted list:
  - build($\{e_1,\ldots,e_n\}$): time O(n)
  - insert(e): O(1)
  - min, deleteMin: O(n)
- Priority Queue based on sorted array:
  - build($\{e_1,\ldots,e_n\}$): time O(n log n) (needed for sorting)
  - insert(e): O(n) (rearrange elements in array)
  - min, deleteMin: O(1)

Better structure needed than list or array!

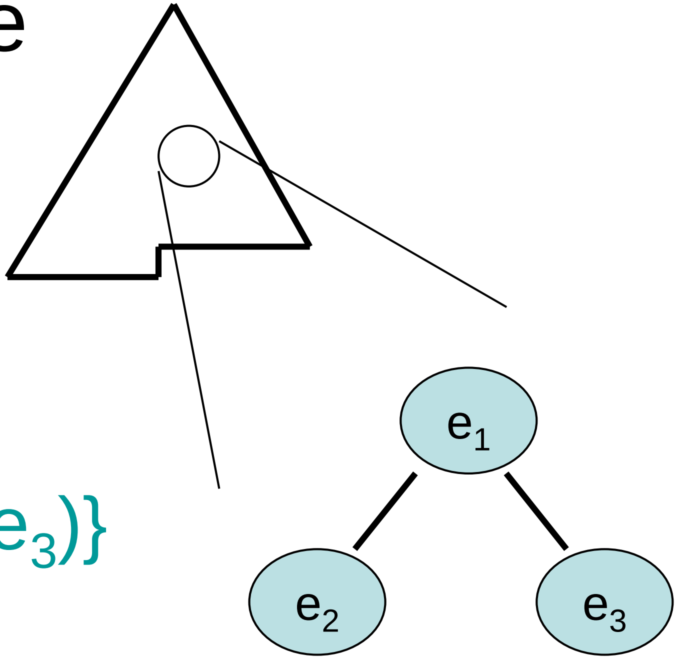
Binary Heap

Idee: use binary tree instead of list

Preserve two invariants:

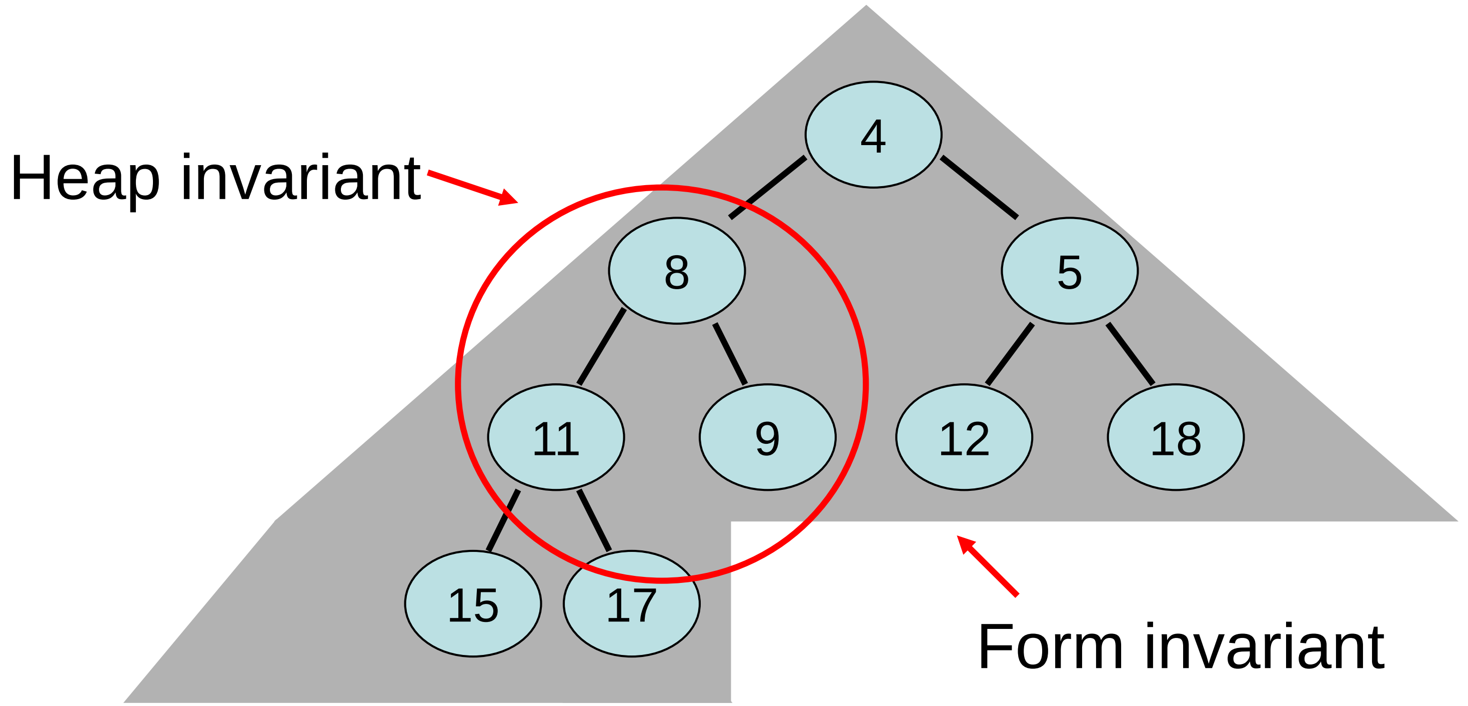
- Form invariant: complete binary tree up to lowest level
- Heap invariant:

  $\text{key}(e_1) \leq \min\{\text{key}(e_2), \text{key}(e_3)\}$



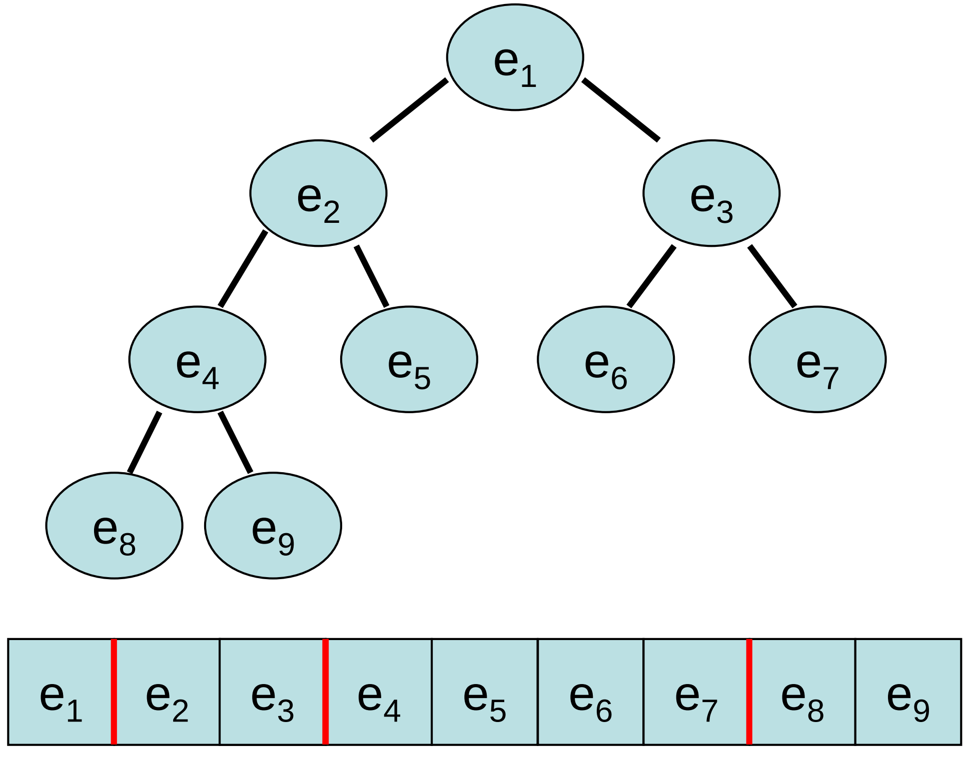
Binary Heap

Example:



Binary Heap

Representation of binary tree via array:



Binary Heap

Representation of binary tree via array:

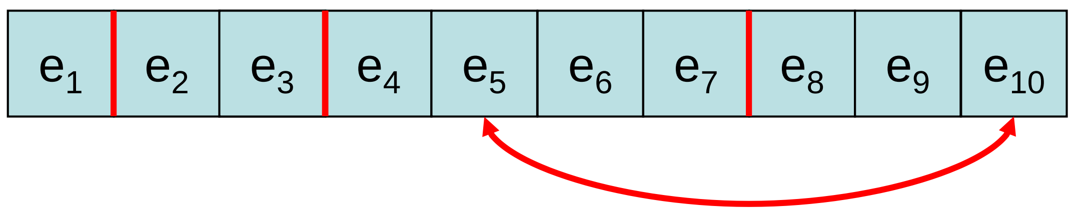
e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9

- H: Array [1..N] of Element (N ≥ #elements n)
- Children of e in H[i]: in H[2i], H[2i+1]
- Form invariant: H[1],…,H[n] occupied
- Heap invariant: for all $i \in \{2,\dots,n\}$,

$$key(H[i]) \geq key(H[\lfloor i/2 \rfloor])$$

Binary Heap

Representation of binary tree via array:



insert(e):

- Form invariant: $n:=n+1$; $H[n]:=e$
- Heap invariant: as long as e is in $H[k]$ with $k>1$ and $key(e)<key(H[\lfloor k/2\rfloor])$, switch e with parent

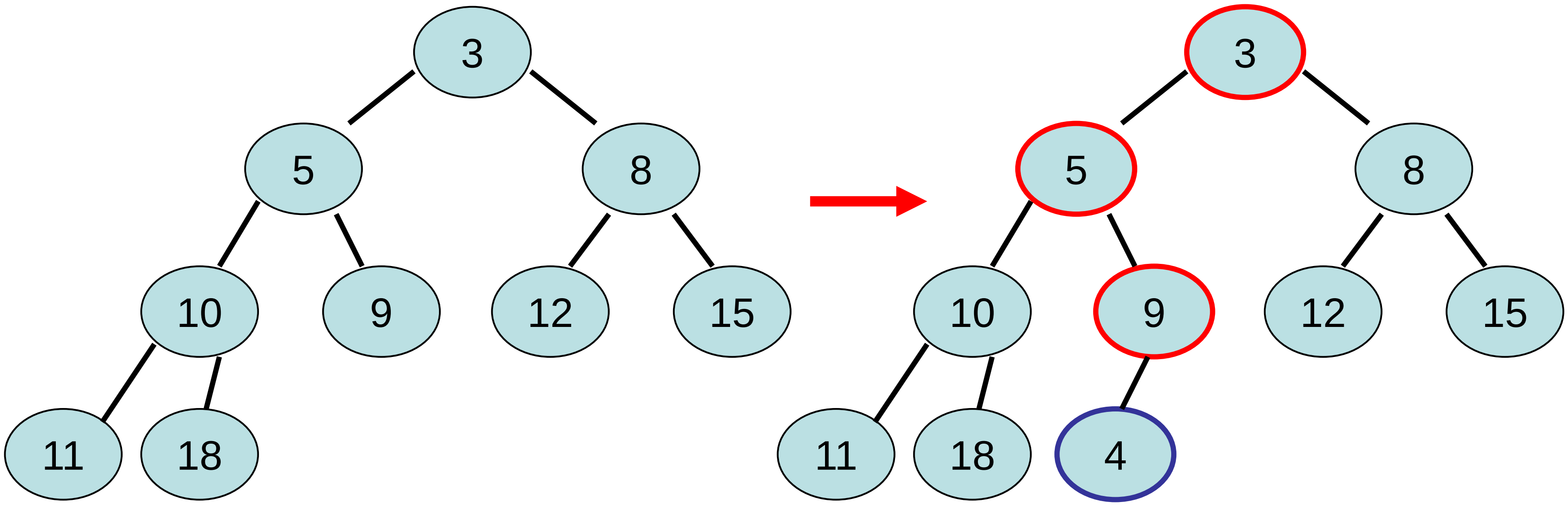
Insert Operation

insert(e: Element):
   n:=n+1; H[n]:=e
   heapifyUp(n)

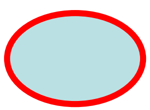
heapifyUp(i: Integer):
   while i>1 and key(H[i])<key(H[⌊i/2⌋]) do
      H[i] ↔ H[⌊i/2⌋]
      i:=⌊i/2⌋

Runtime: O(log n)

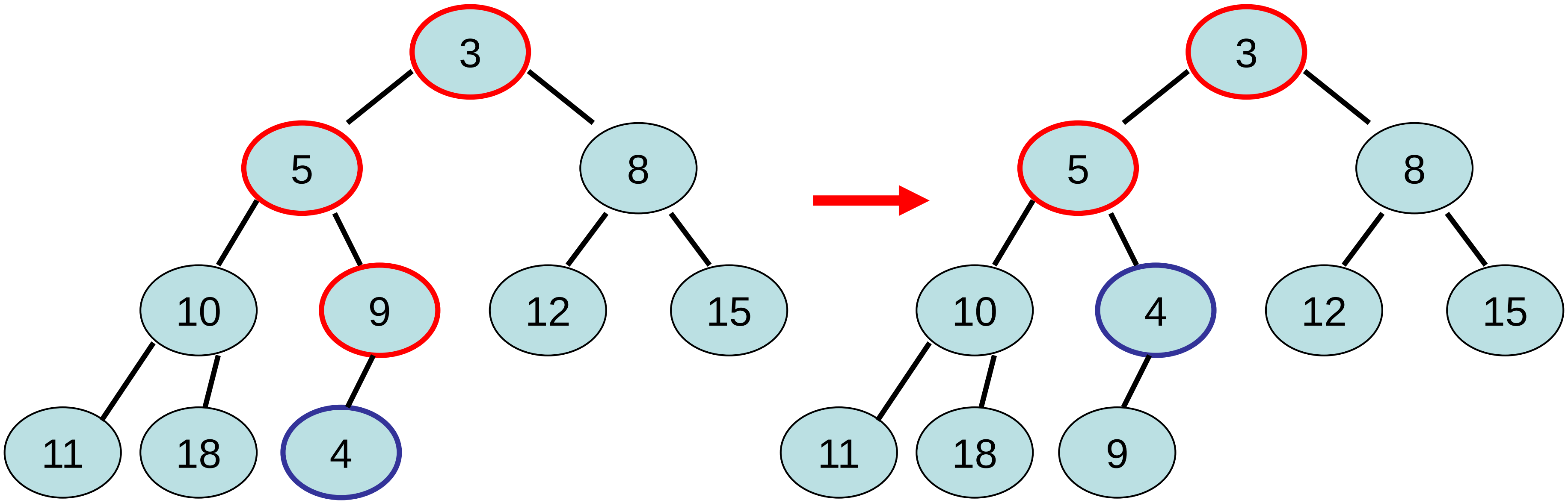
Insert Operation - Correctness



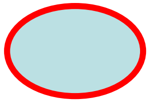
Invariant: H[k] is minimal w.r.t. subtree of H[k]

 : nodes that may violate invariant

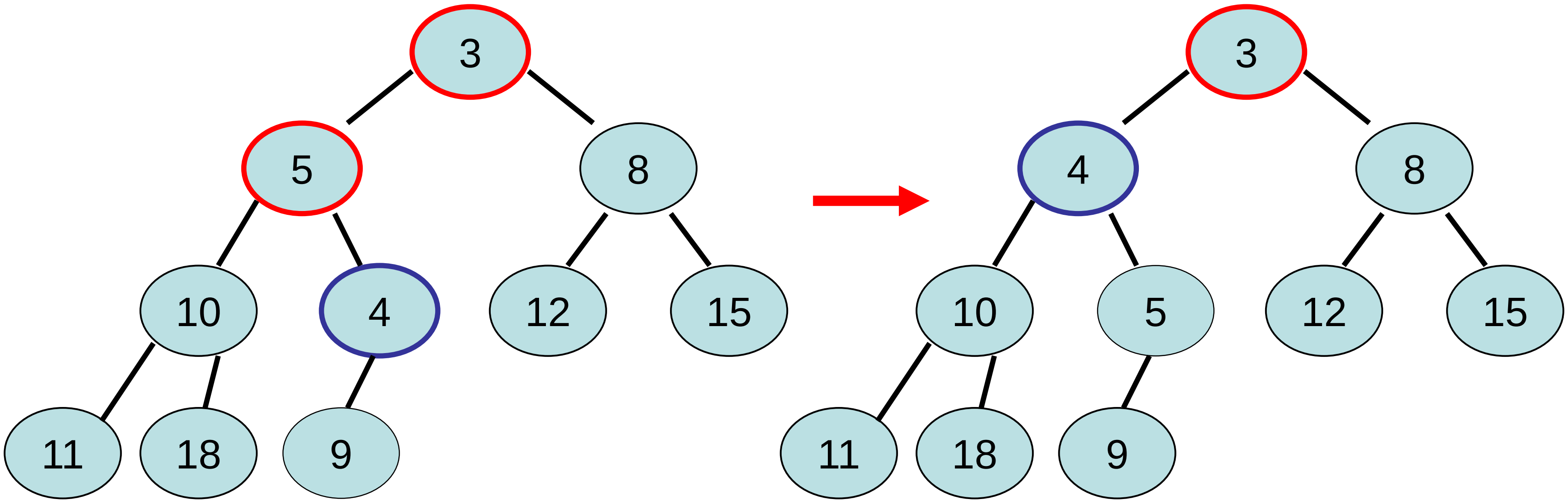
Insert Operation - Correctness



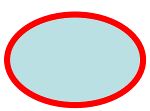
Invariant: H[k] is minimal w.r.t. subtree of H[k]

 : nodes that may violate invariant

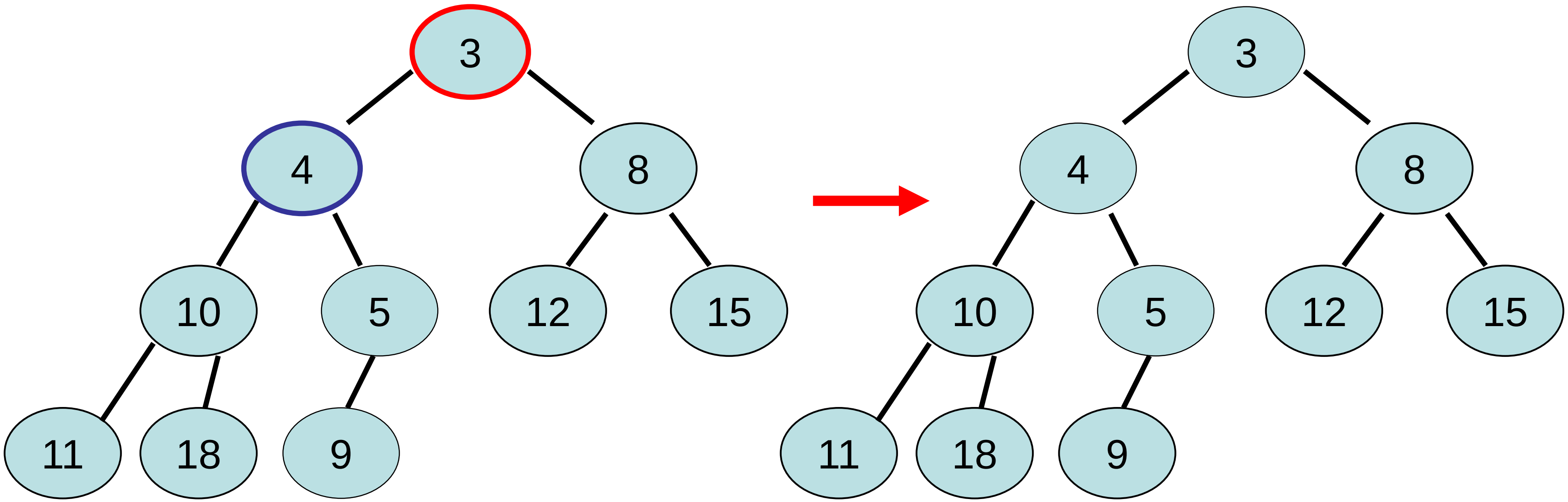
Insert Operation - Correctness



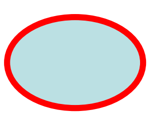
Invariant: H[k] is minimal w.r.t. subtree of H[k]

 : nodes that may violate invariant

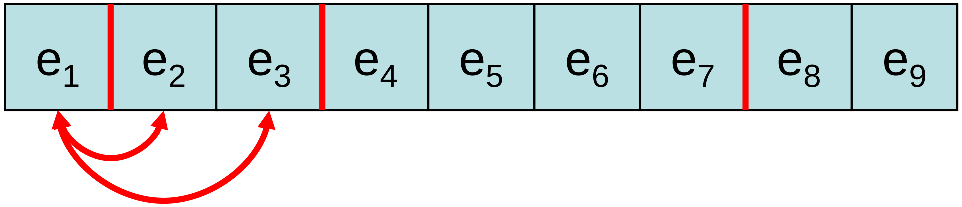
Insert Operation - Correctness



Invariant: H[k] is minimal w.r.t. subtree of H[k]

 : nodes that may violate invariant

Binary Heap



deleteMin:

- Form invariant: H[1]:=H[n]; n:=n-1
- Heap invariant: start with e in H[1]. Switch e with the child with minimum key until $H[k] \leq \min\{H[2k],H[2k+1]\}$ for the current position k of e or e is in a leaf

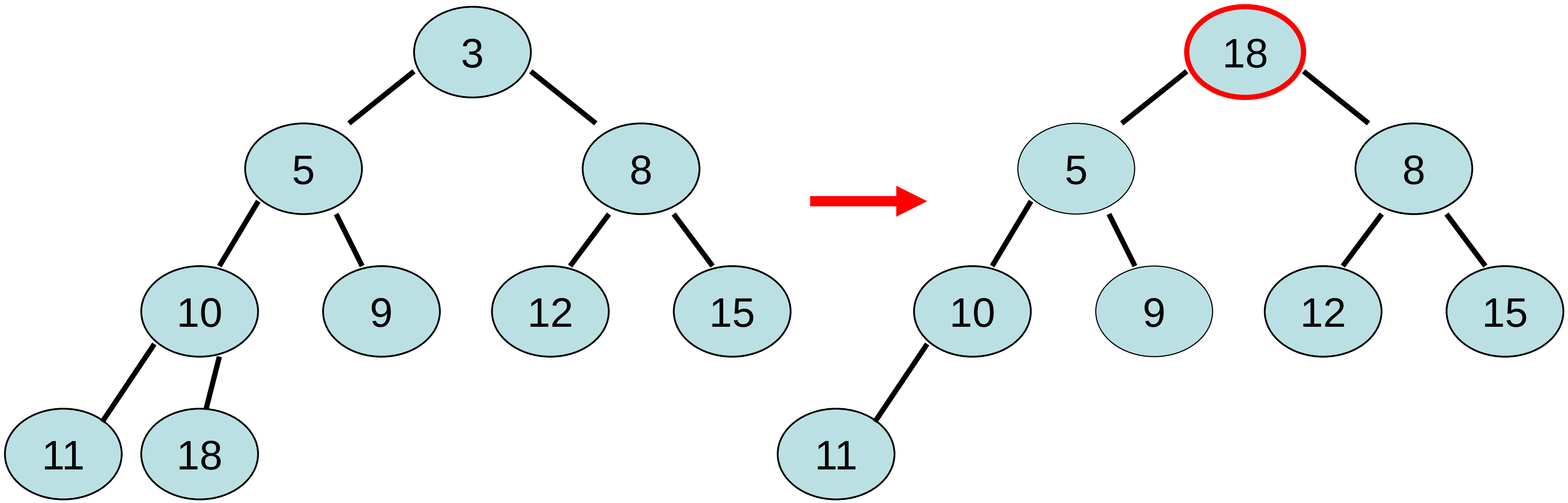
Binary Heap

deleteMin(): Runtime: $O(\log n)$

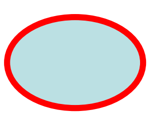
```
deleteMin():
    e:=H[1]; H[1]:=H[n]; n:=n-1
    heapifyDown(1)
    return e

heapifyDown(i: Integer):
    while 2i≤n do                // i is not a leaf position
        if 2i+1>n then m:=2i  // m: pos. of the minimum child
        else
            if key(H[2i])<key(H[2i+1]) then m:=2i
                                          else m:=2i+1
        if key(H[i])≤key(H[m]) then return  // heap inv. holds
        H[i] ↔ H[m]; i:=m
```

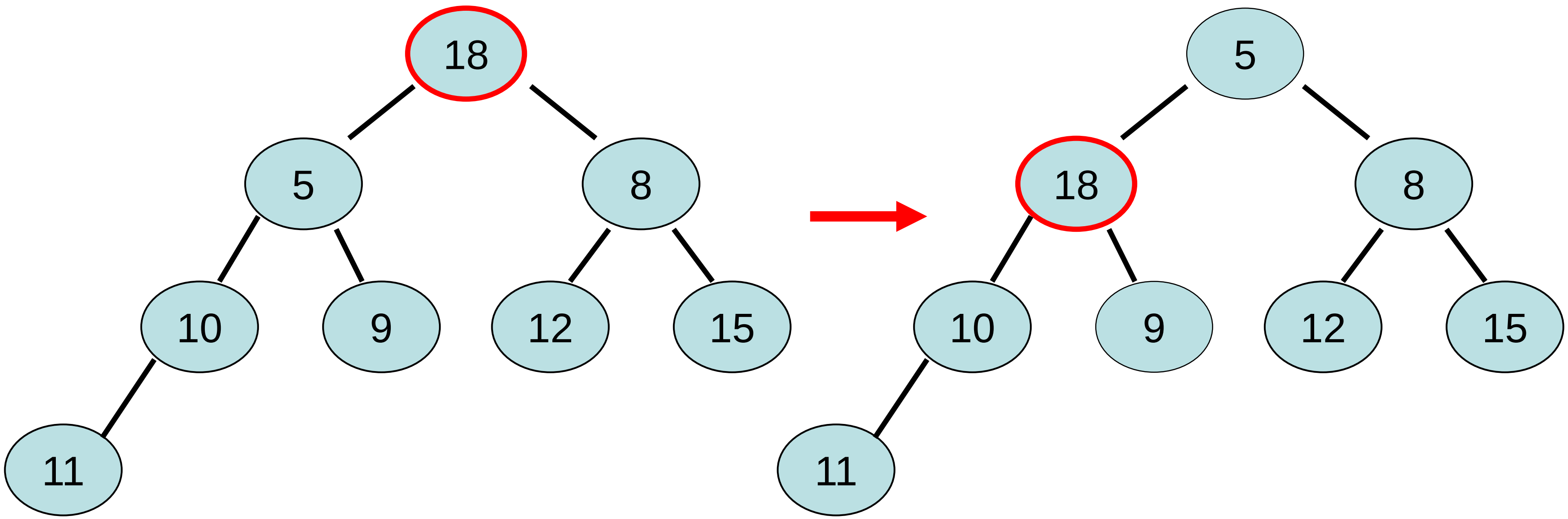
deleteMin Operation - Correctness



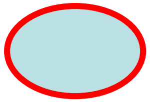
Invariant: H[k] is minimal w.r.t. subtree of H[k]

 : nodes that may violate invariant

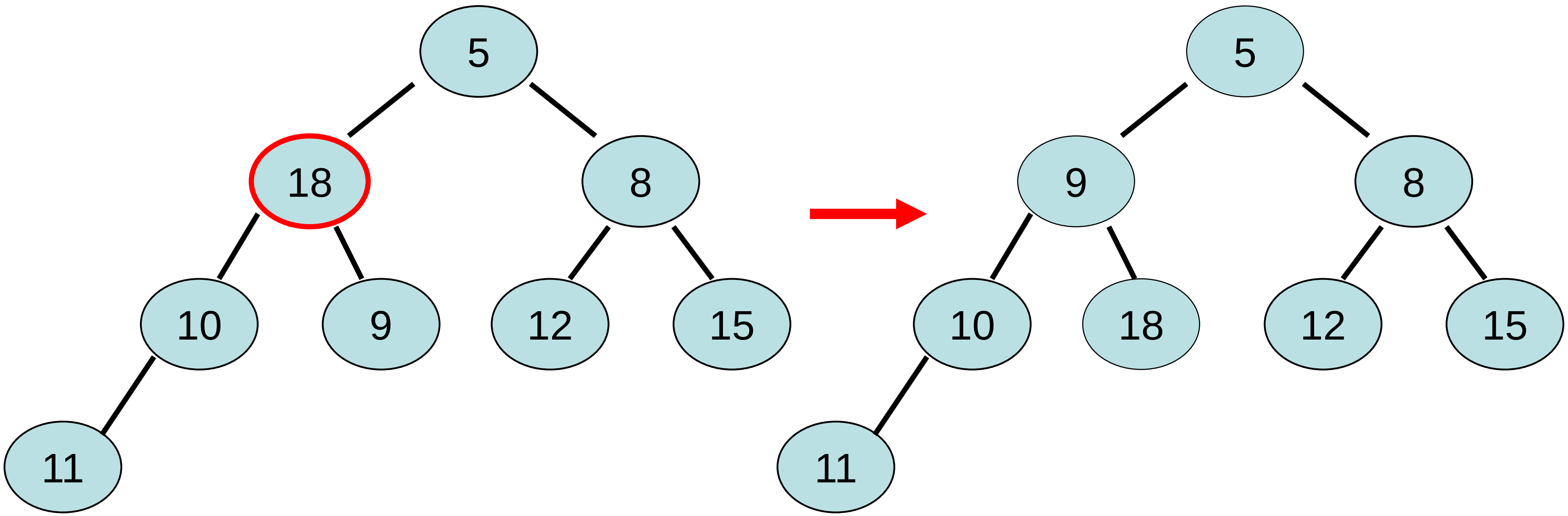
deleteMin Operation - Correctness



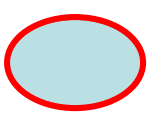
Invariant: H[k] is minimal w.r.t. subtree of H[k]

 : nodes that may violate invariant

deleteMin Operation - Correctness



Invariant: H[k] is minimal w.r.t. subtree of H[k]

 : nodes that may violate invariant

Binary Heap

Naive implementation:

build($\{e_1,\ldots,e_n\}$):

- Call insert(e) n times.
- Runtime $O(n \log n)$.

More careful implementation:

build($\{e_1,\ldots,e_n\}$):

for $i:=\lfloor n/2 \rfloor$ downto 1 do
    heapifyDown(i)

- Fact: $H(i)$ for $\lfloor n/2 \rfloor+1 <= i <=n$ are *leaves* of heap

- Runtime: Why should this be faster than $O(n \log n)$?

Careful analysis

More careful implementation:

build($\{e_1,\ldots,e_n\}$):
  for $i:=\lfloor n/2 \rfloor$ downto 1 do
    heapifyDown(i)

Observation: Cost of heapifyDown(i) is O(h), for h the height of the subtree rooted at H(i).

Height(i): #edges on longest simple path from i to leaf

Careful analysis

build({e_1,…,e_n}):
  for i:=$\lfloor n/2 \rfloor$ downto 1 do
    heapifyDown(i)

Facts for n-element heap:

1. Height(root)= $\lfloor \log(n) \rfloor$
2. #nodes of height h $\leq \lceil n/2^{h+1} \rceil$

Runtime (use fact $\sum_{k=0}^{\infty} kx^k = x/(1-x)^2$ for $|x| \leq 1$):

$$\sum_{h=0}^{\lfloor \log(n) \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n \sum_{h=0}^{\lfloor \log(n) \rfloor} \frac{h}{2^h}) = O(n).$$

Binary Heap

Runtime:

- build($\{e_1,\dots,e_n\}$): $O(n)$
- insert(e): $O(\log n)$
- min: $O(1)$
- deleteMin: $O(\log n)$

Extended Priority Queue

Additional Operations:

- M.delete(e: Element): $M:=M\setminus\{e\}$
- M.decreaseKey(e:Element, Δ): $key(e):=key(e)-\Delta$
- M.merge(M´): $M:=M\cup M´$

- delete and decreaseKey can be implemented with runtime $O(\log n)$ in binary heap (if position of e is known)
- merge is expensive ($\Theta(n)$ time)!

Ouch!

- M.merge(M´): $M := M \cup M'$

- merge is expensive ($\Theta(n)$ time)!
- merging binary heaps M and M‘ requires „starting from scratch“, i.e. building a new binary heap containing all elements of M and M‘
- Bad news if our application needs many merges. Can we do better?
- Yes! Via Binomial Heaps.

Binomial Heap

Goal: Maintain costs of Binary Heaps, but bring cost of merge from $\Theta(n)$ to $O(\log n)$.

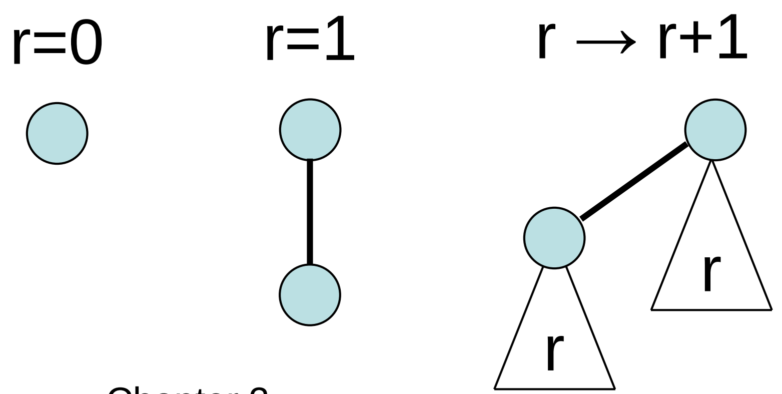
Binomial heap is collection of binomial trees

So let us first define binomial trees!

Binomial Heap

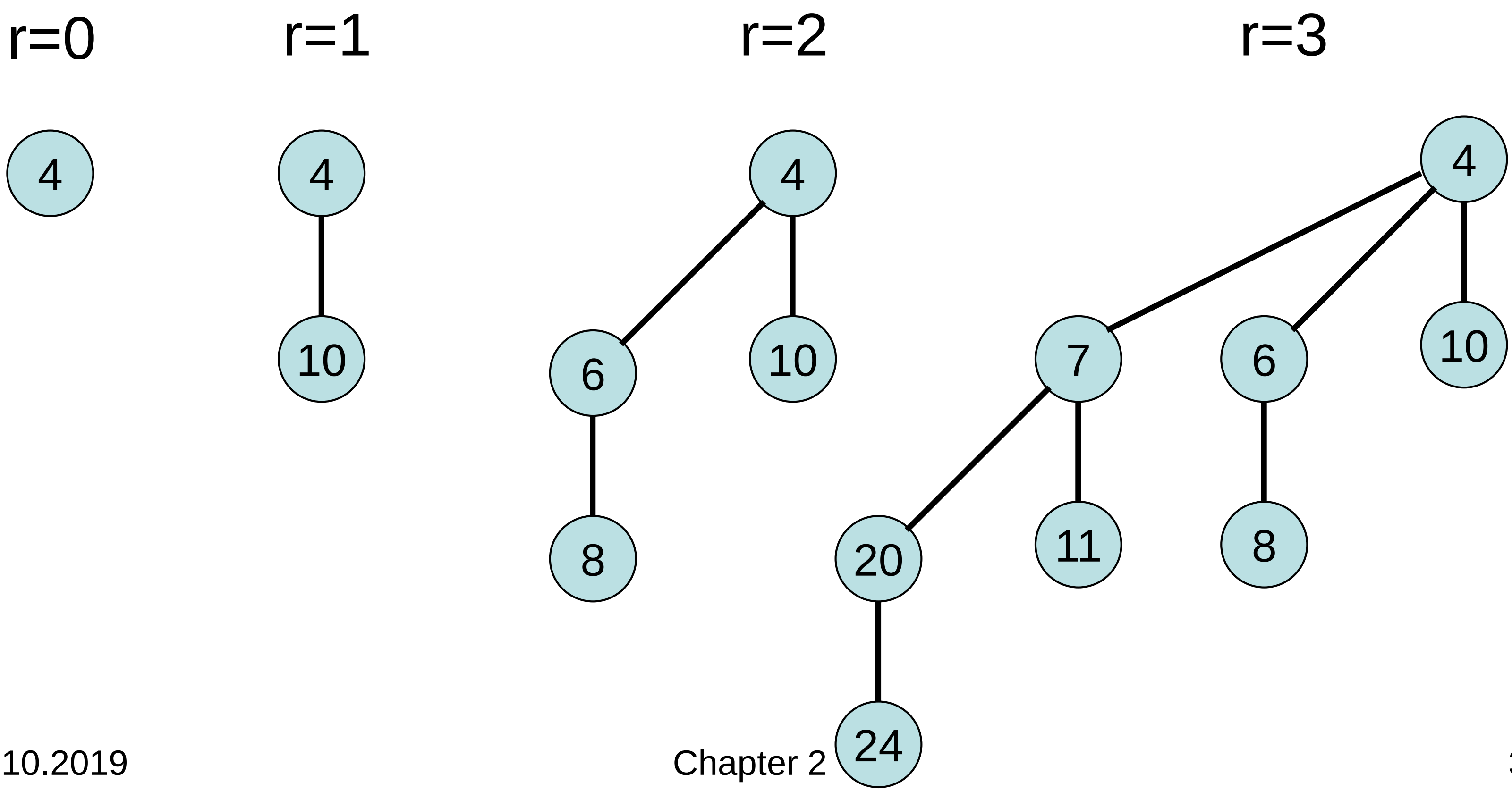
Binomial trees:

- defined recursively for *rank* r
- Tree B_r is two trees B_{r-1} linked together.
- Form invariant:



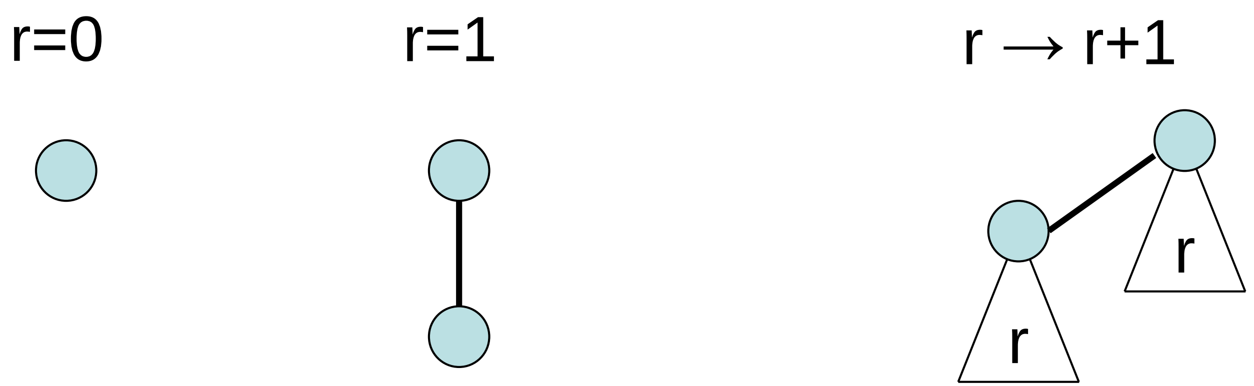
Binomial Trees

Examples of Binomial trees:



Binomial Trees

Properties of Binomial trees:



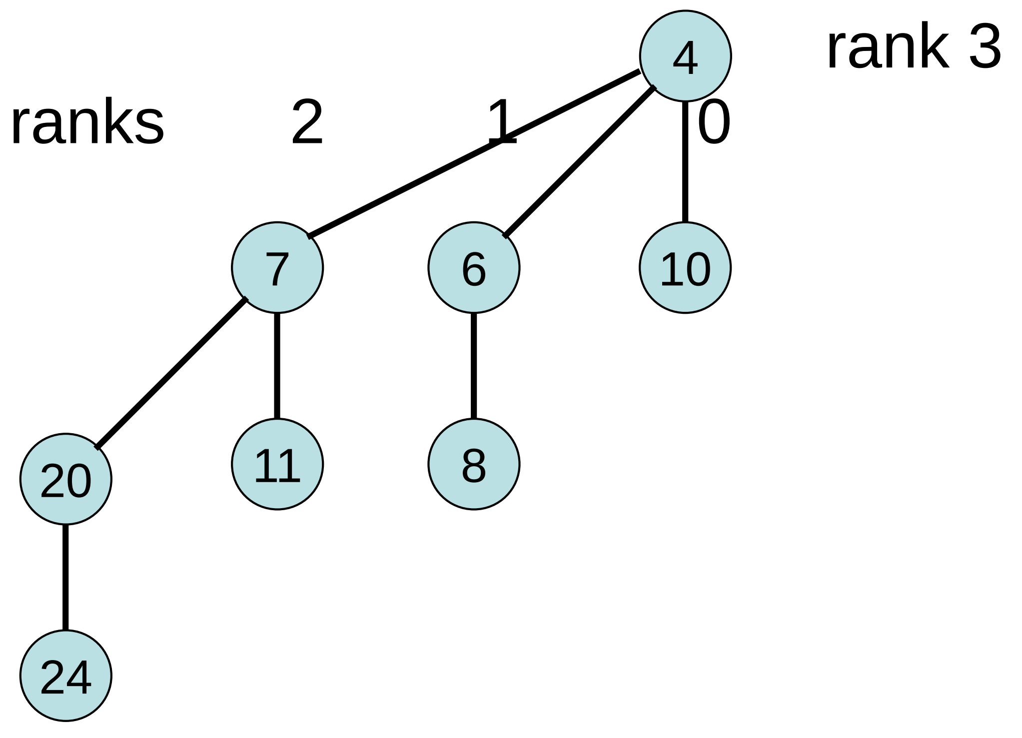
- 2^r nodes

number of neighbors

- maximum degree r (at root)
- root deleted: Tree splits into Binomial trees of rank 0 to r-1 (exactly one of each rank!)

Binomial Trees

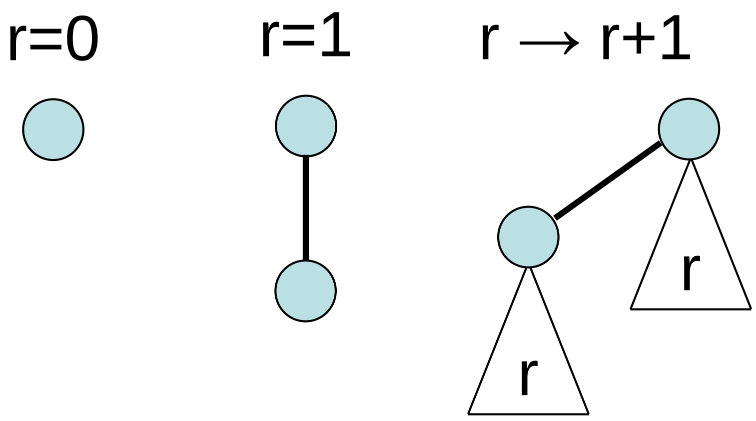
Example for decomposition into Binomial trees of rank 0 to r-1 (exactly one per rank)



Binomial Heap

Binomial trees:

- defined recursively for *rank* r
- Tree B_r is two trees B_{r-1} linked together.
- Form invariant:

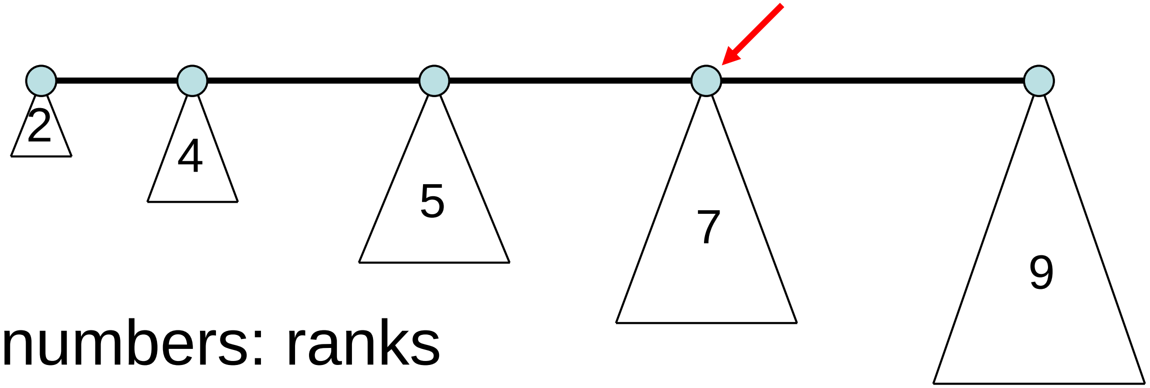


- Heap invariant:
  (key(Parent)≤key(Children))

Binomial Heap

Binomial Heap:

- linked list of Binomial trees, ordered by ranks
- for each rank at most 1 Binomial tree
- pointer to root with minimal key (optional)



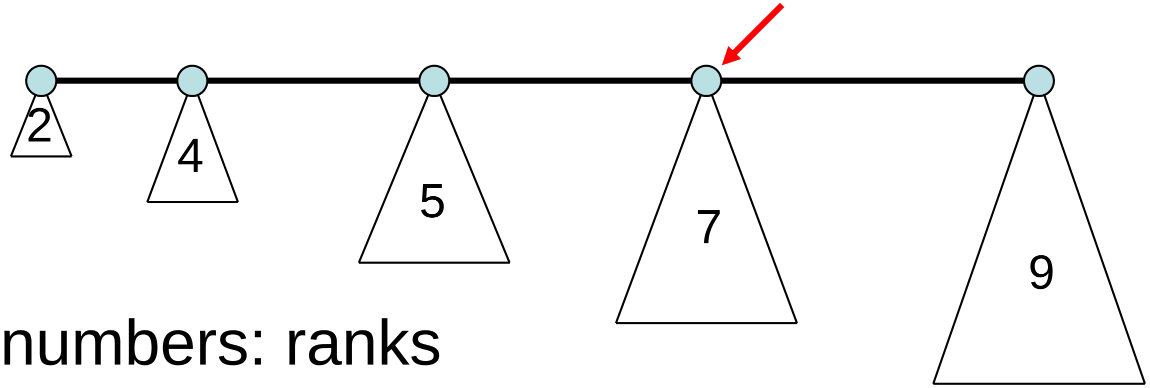
numbers: ranks

Binomial Heap

Data type:

binTree:

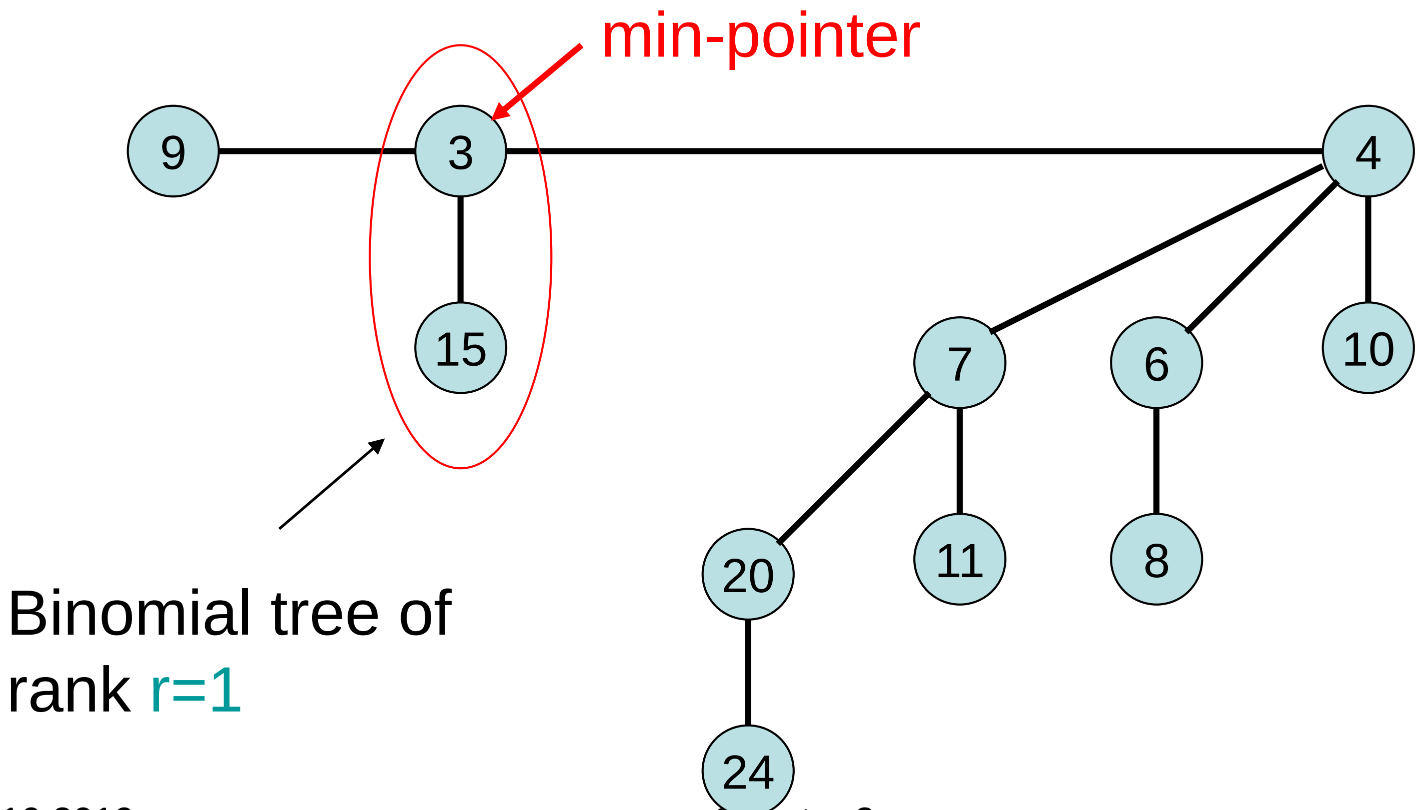
parent: binTree
prev: binTree
next: binTree
key: Integer
rank: Integer
Children: binTree



numbers: ranks

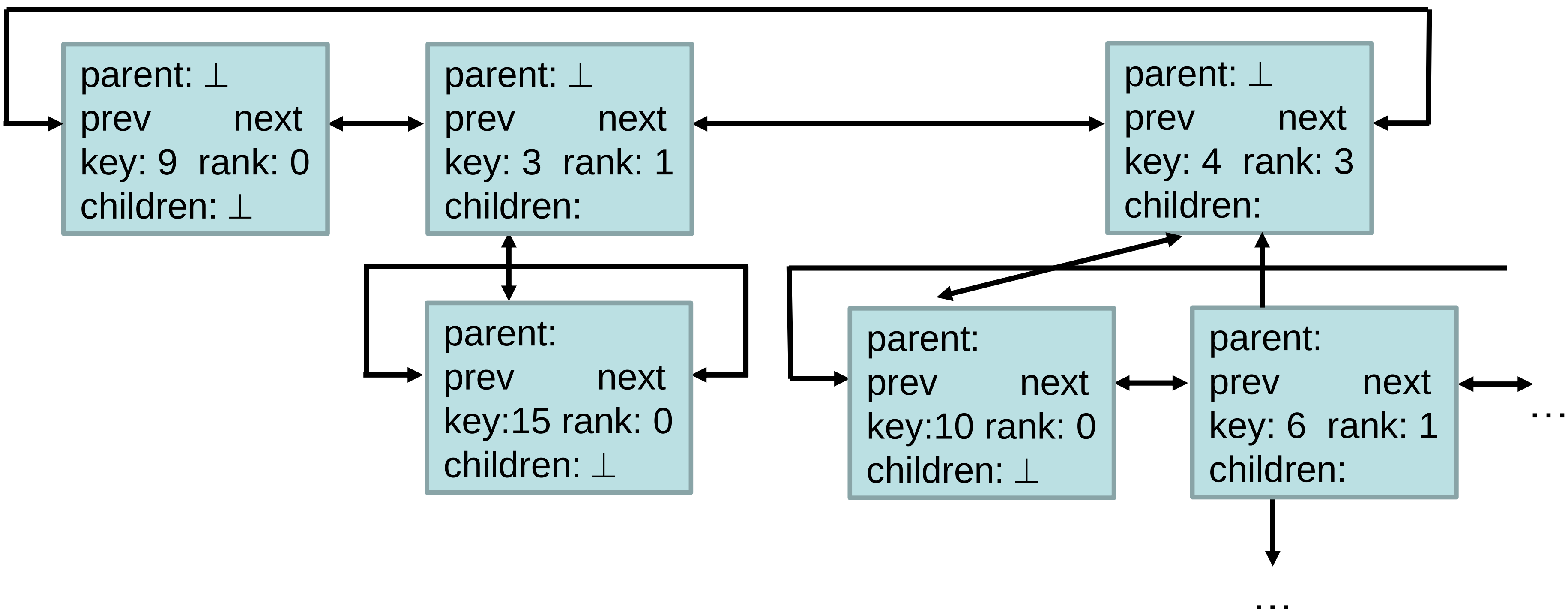
Binomial Heap

Example of a correct Binomial heap:



Binomial Heap

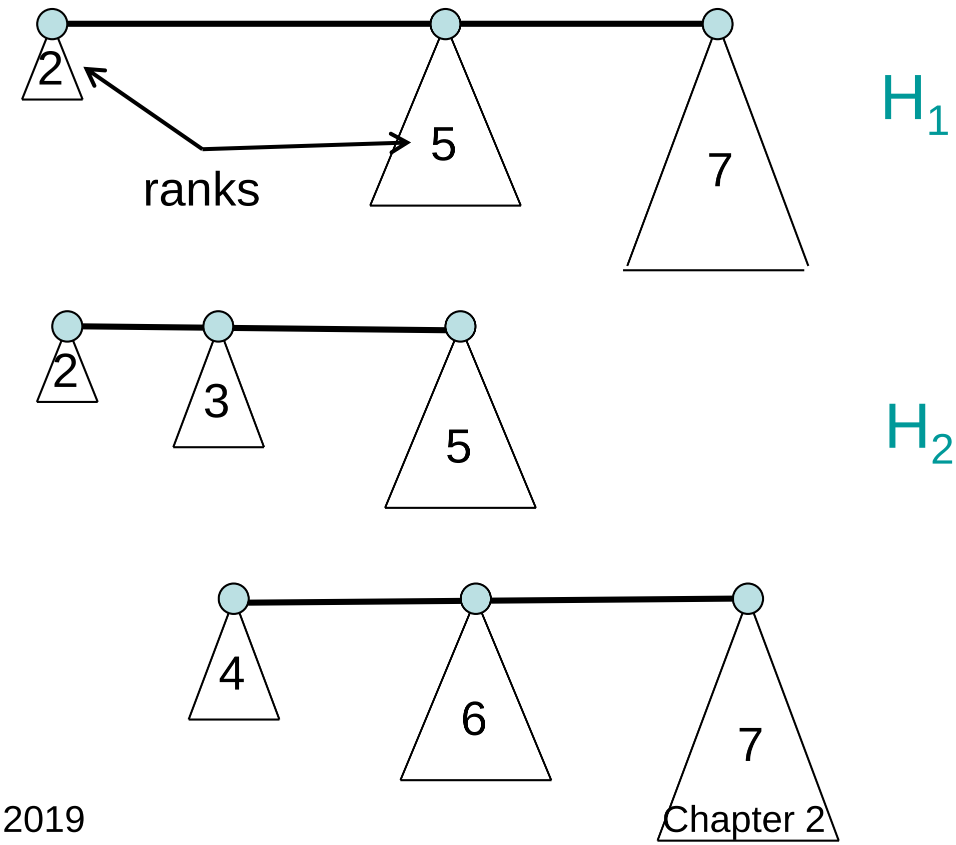
Example of a correct Binomial heap:



Binomial Heap

Question: How many times can a distinct rank appear between both trees? 2.

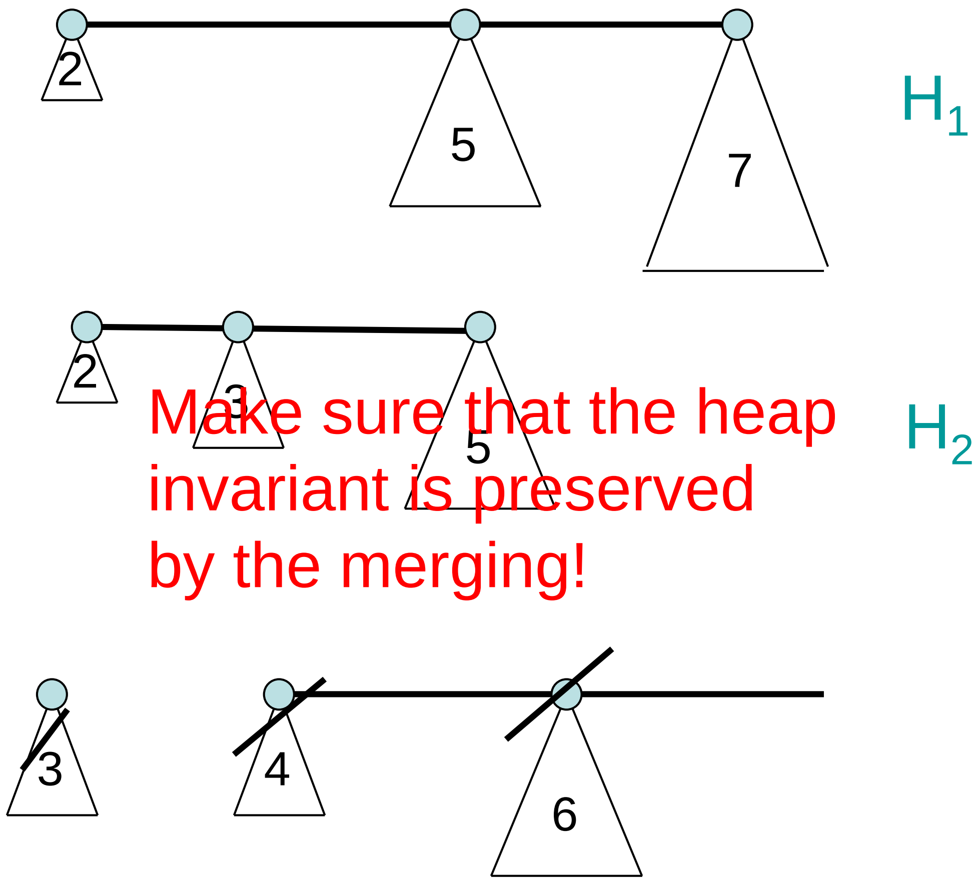
Merge of Binomial heaps H_1 and H_2:



Idea: binary addition

$$
\begin{array}{r}
10100100 \\
+\ \ 101100 \\
\hline
11010000
\end{array}
$$

Example of Merge Operation



numbers denote the ranks

Make sure that the heap invariant is preserved by the merging!

outcome

Binomial Heap

Runtime of merge operation: $O(\log n)$ because

- the largest rank in a Binomial heap with n elements at most $\log n$ (see analogy with binary numbers), and
- at most one Binomial tree is allowed for each rank value

B_i: Binomial tree of rank i

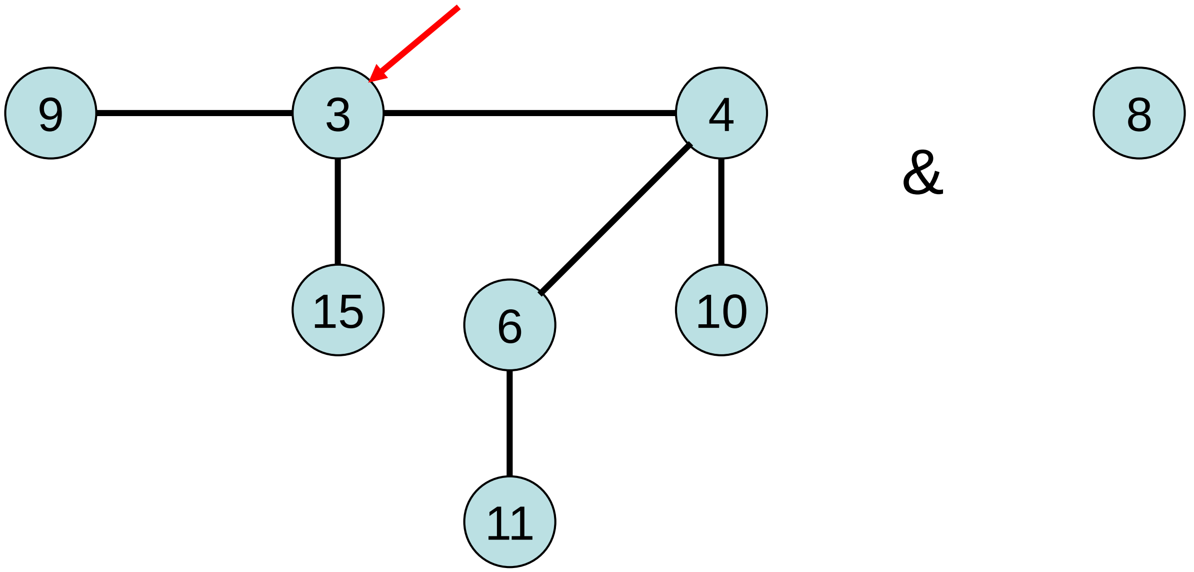
- insert(e): merge existing heap with B_0 containing only element e
- min: use min-pointer, time $O(1)$. (Without min-pointer, $O(\log n)$.)
- deleteMin: let the min-pointer point to the root of B_i.
  In H, deleting the root of B_i results in Binomial trees $B_0,\dots,B_{i-1}$.
  - Obs: Since $B_0,\dots,B_{i-1}$ have distinct ranks, can link them immediately to make a temporary Binomial heap H'. Then merge H and H'.

Remarks:

- insert and deleteMin reduce to merge, yielding runtime of $O(\log n)$.
- If using min-pointer, update min-pointer after insert and deleteMin. Additive cost: $O(\log n)$.

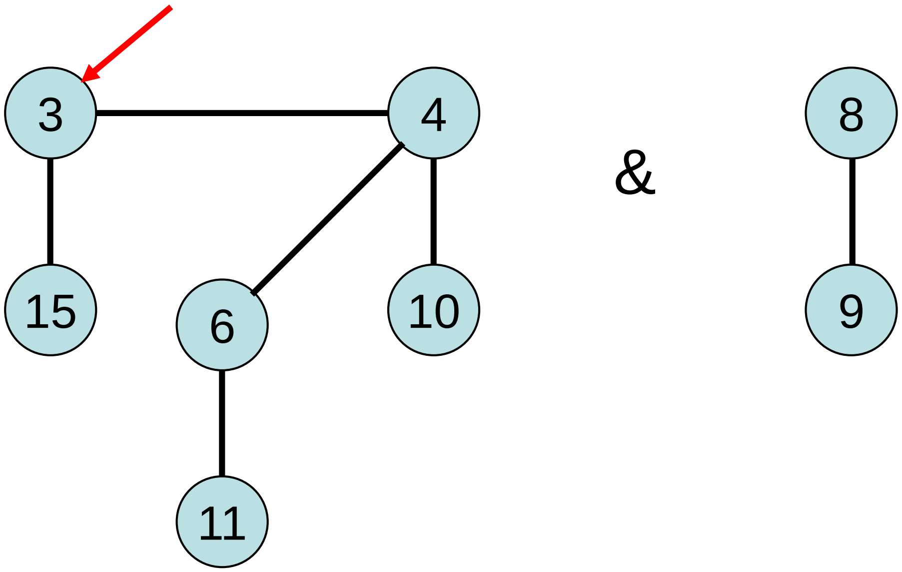
Example of Insert Operation

Insert(8):



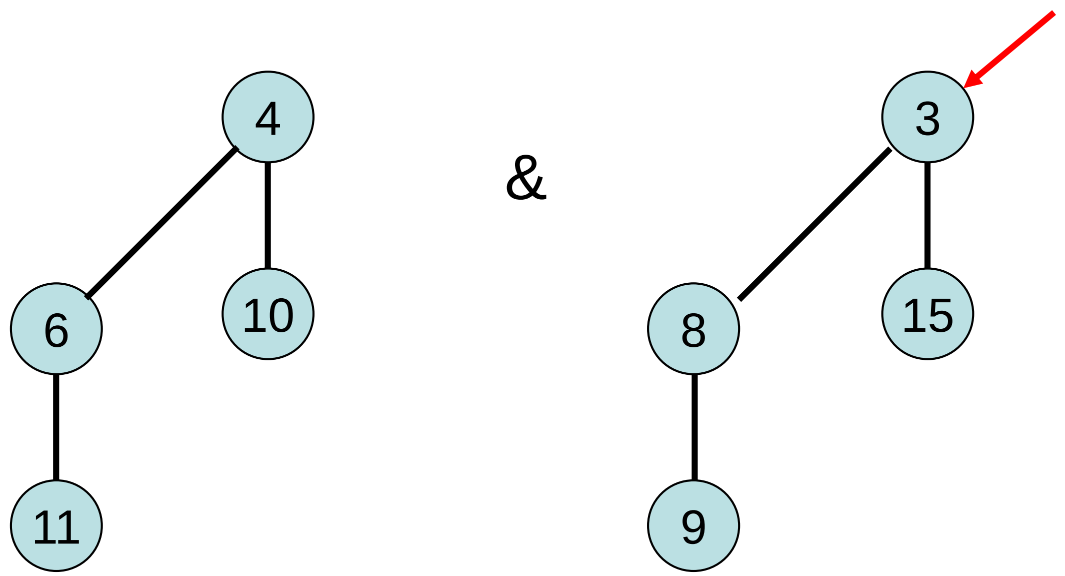
Example of Insert Operation

Insert(8):



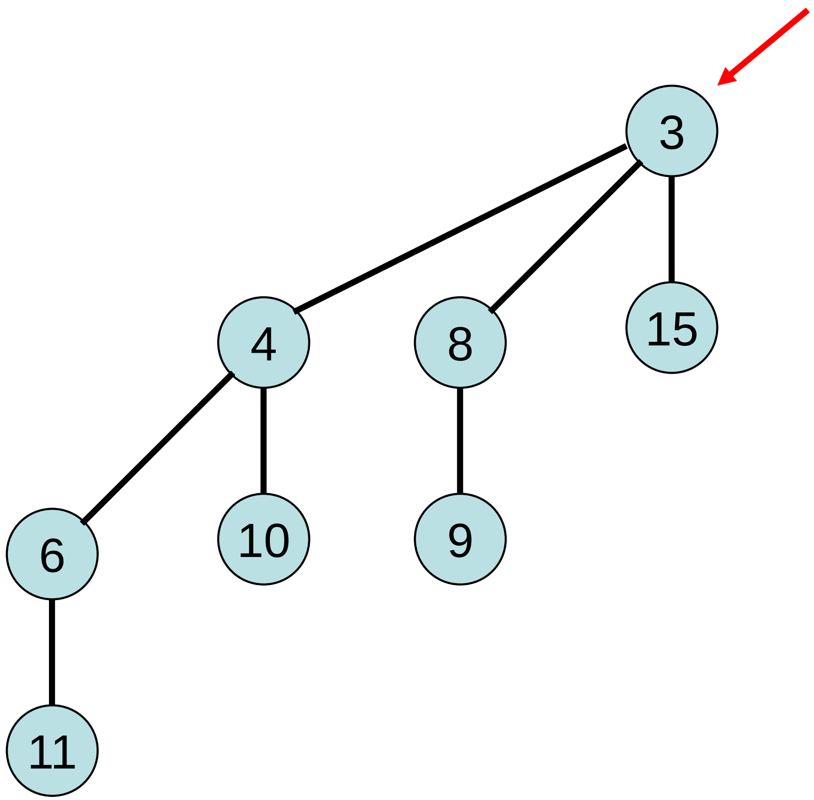
Example of Insert Operation

Insert(8):



Example of Insert Operation

Outcome of Insert(8):



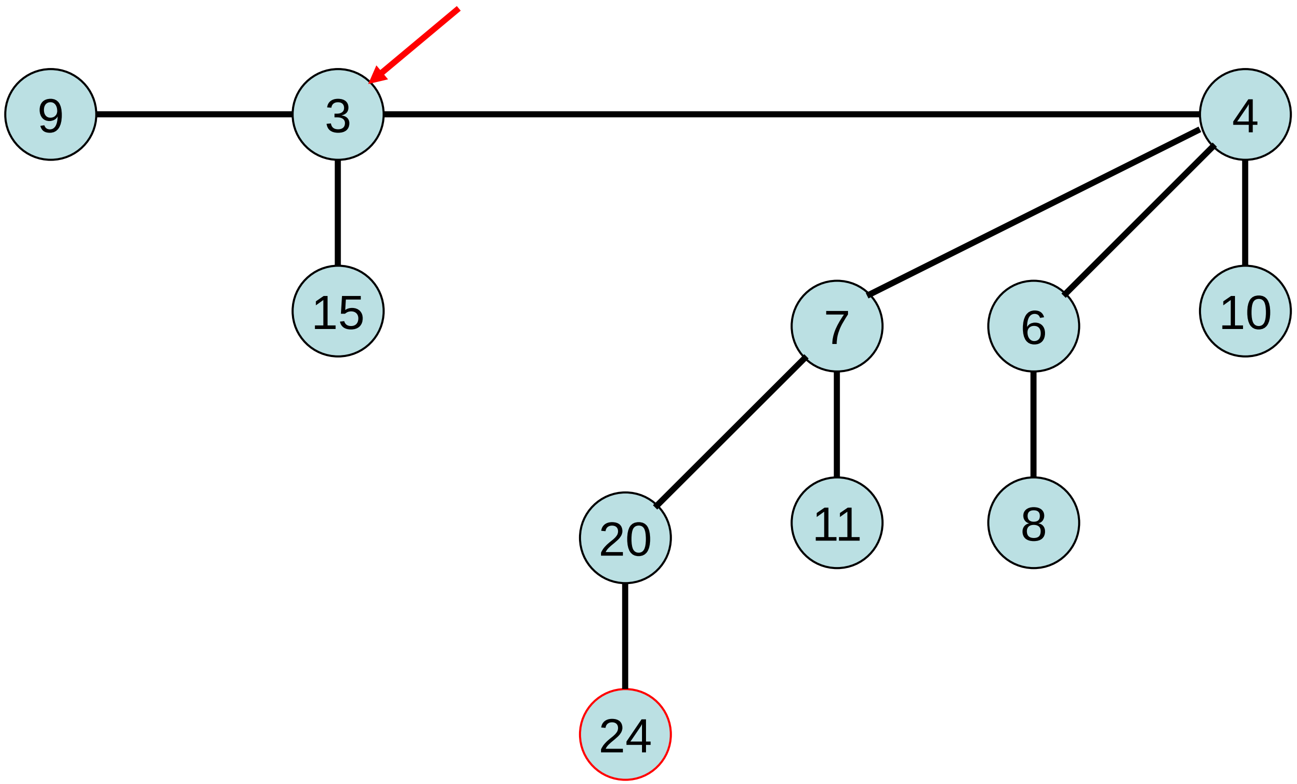
Binomial Heap

- decreaseKey(e,Δ): perform heapifyUp operation in Binomial tree starting with e, update min-pointer. Time: $O(\log n)$
  - Note: Does *not* change ranks, only *keys*, so suffices to locally relabel nodes of tree containing e.

- delete(e): reduce to deleteMin!
  - call decreaseKey(e,$-\infty$), then deleteMin Time: $O(\log n)$

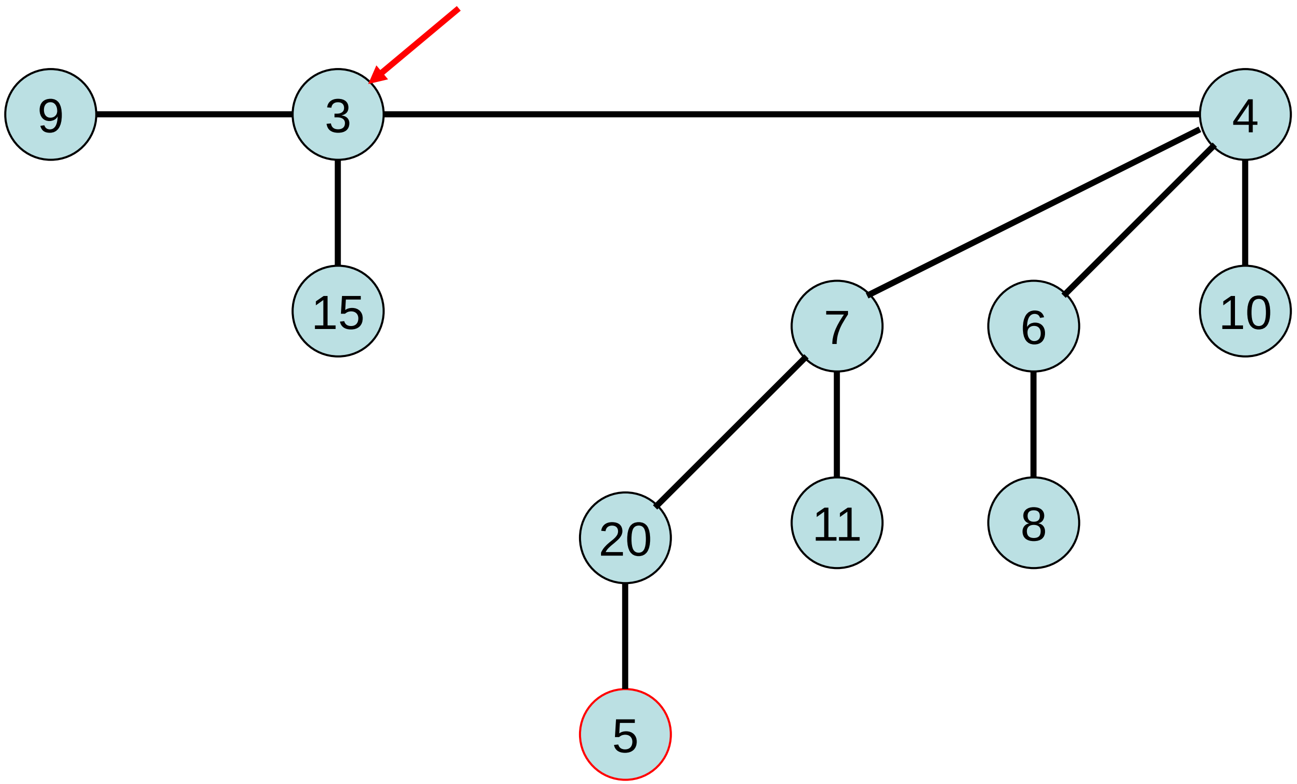
Example of decreaseKey

decreaseKey(24,19):



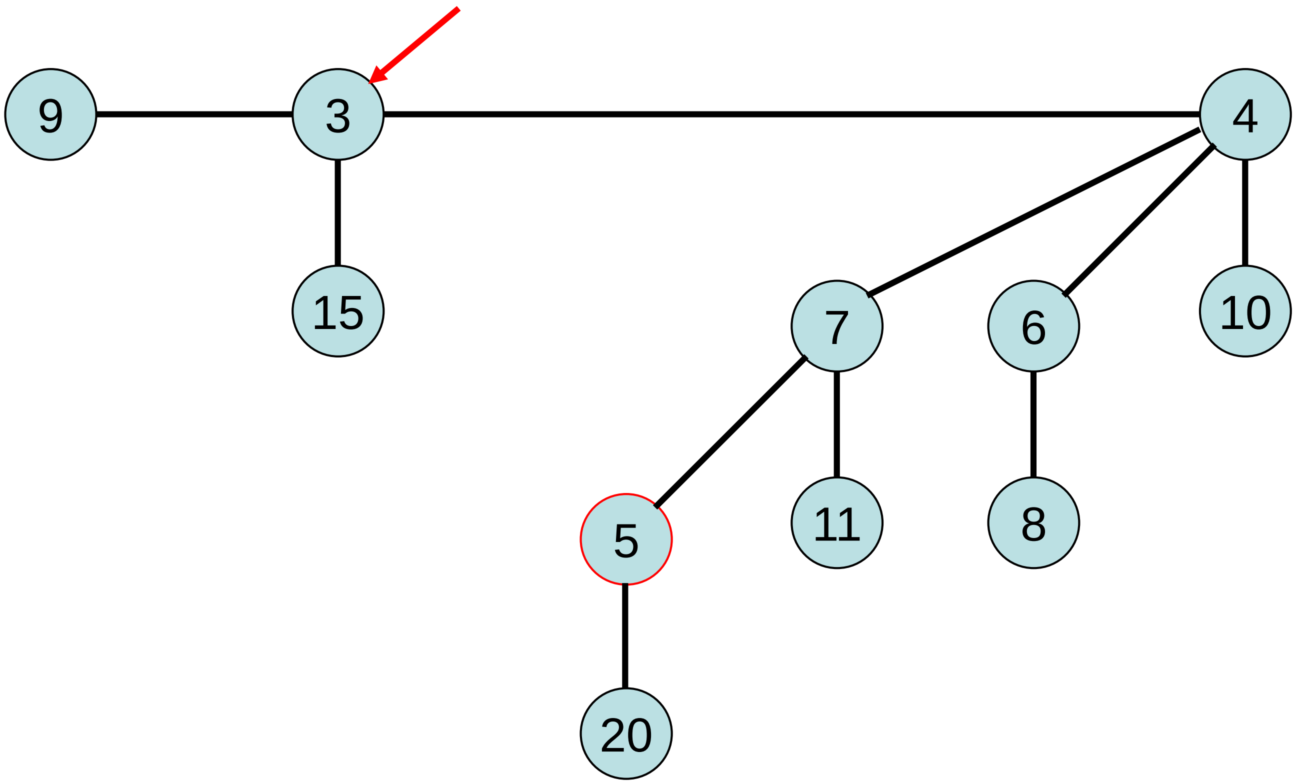
Example of decreaseKey

decreaseKey(24,19):



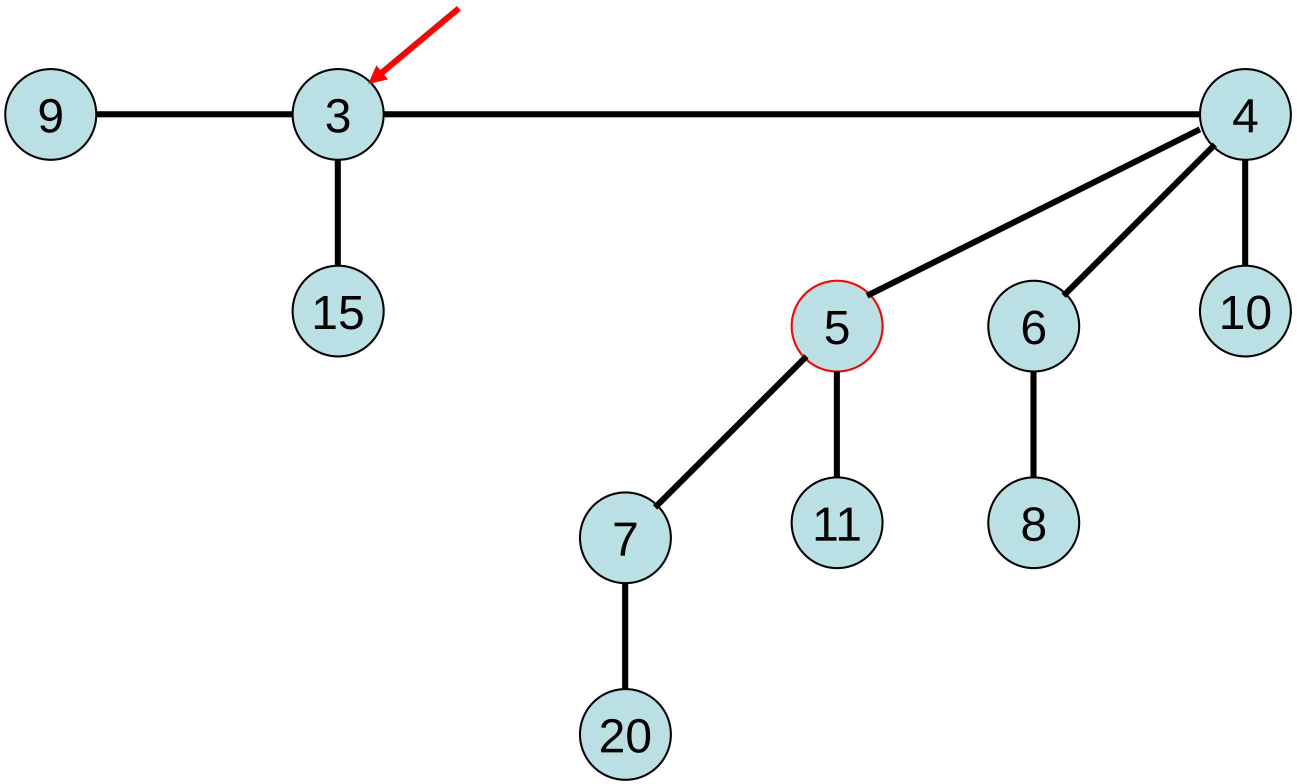
Example of decreaseKey

decreaseKey(24,19):



Example of decreaseKey

Outcome of decreaseKey(24,19):



Recall: Binomial Heap

Goal: Maintain costs of Binary Heaps, but bring cost of merge from $\Theta(n)$ to $O(\log n)$.

- Goal is achieved.
- But... *can we do better*?
- Yes, if we work with *amortized* costs.

Fibonacci Heap

- Goal: To bring amortized cost of operations *not* involving deletion of an element down to O(1).
- Price we pay: Fibonacci Heaps more complicated to implement in practice, large constants hidden in Big-Oh notation

Summary

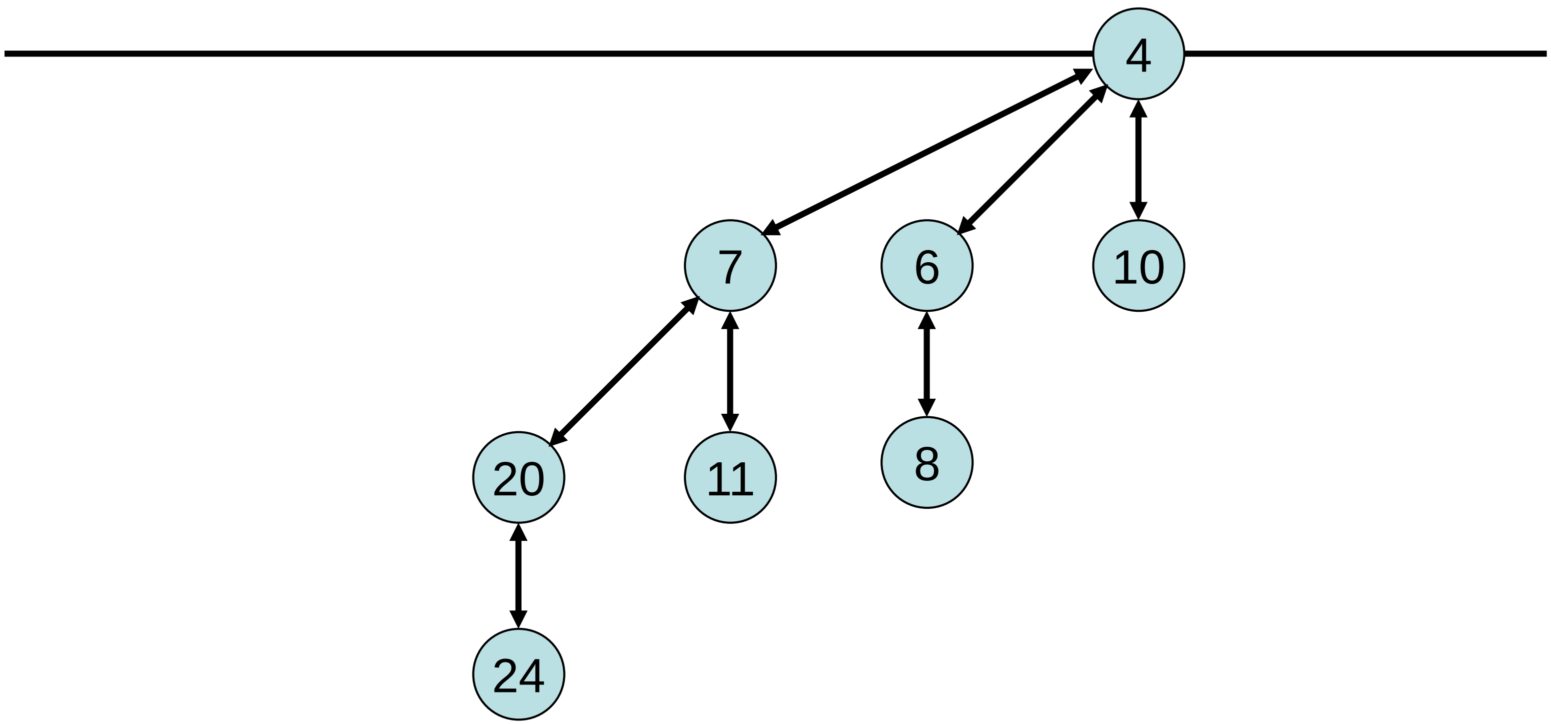
Runtime	Binomial Heap	Fibonacci Heap
insert	O(log n)	O(1)
min	O(1)	O(1)
deleteMin	O(log n)	O(log n) amor.
delete	O(log n)	O(log n) amor.
merge	O(log n)	O(1)
decreaseKey	O(log n)	O(1) amor.

Fibonacci Heap

- Based on Binomial trees, but it allows lazy merge and lazy delete.
- Lazy merge: no merging of Binomial trees of the same rank during merge, only concatenation of the two lists
- Lazy delete: creates incomplete Binomial trees

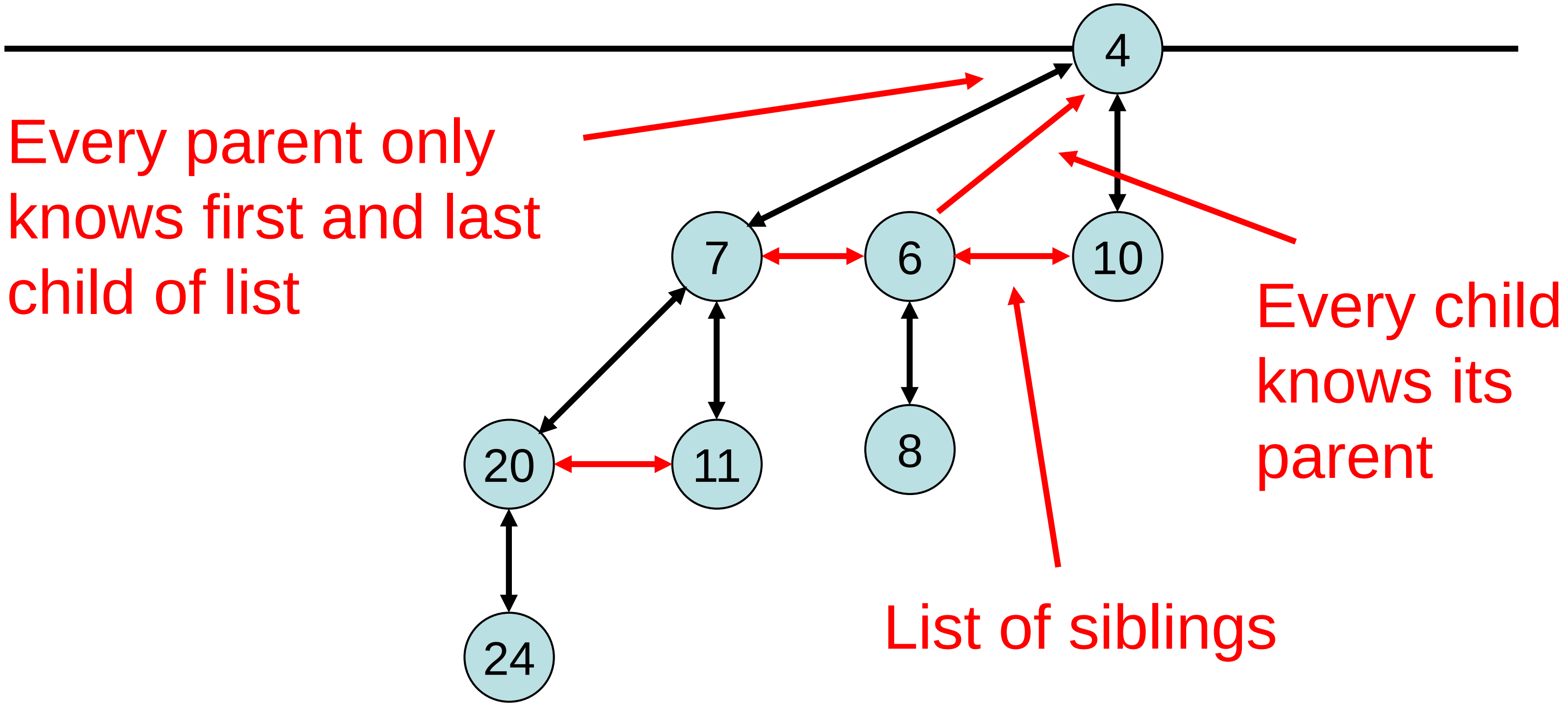
Fibonacci Heap

Tree in a Binomial heap:



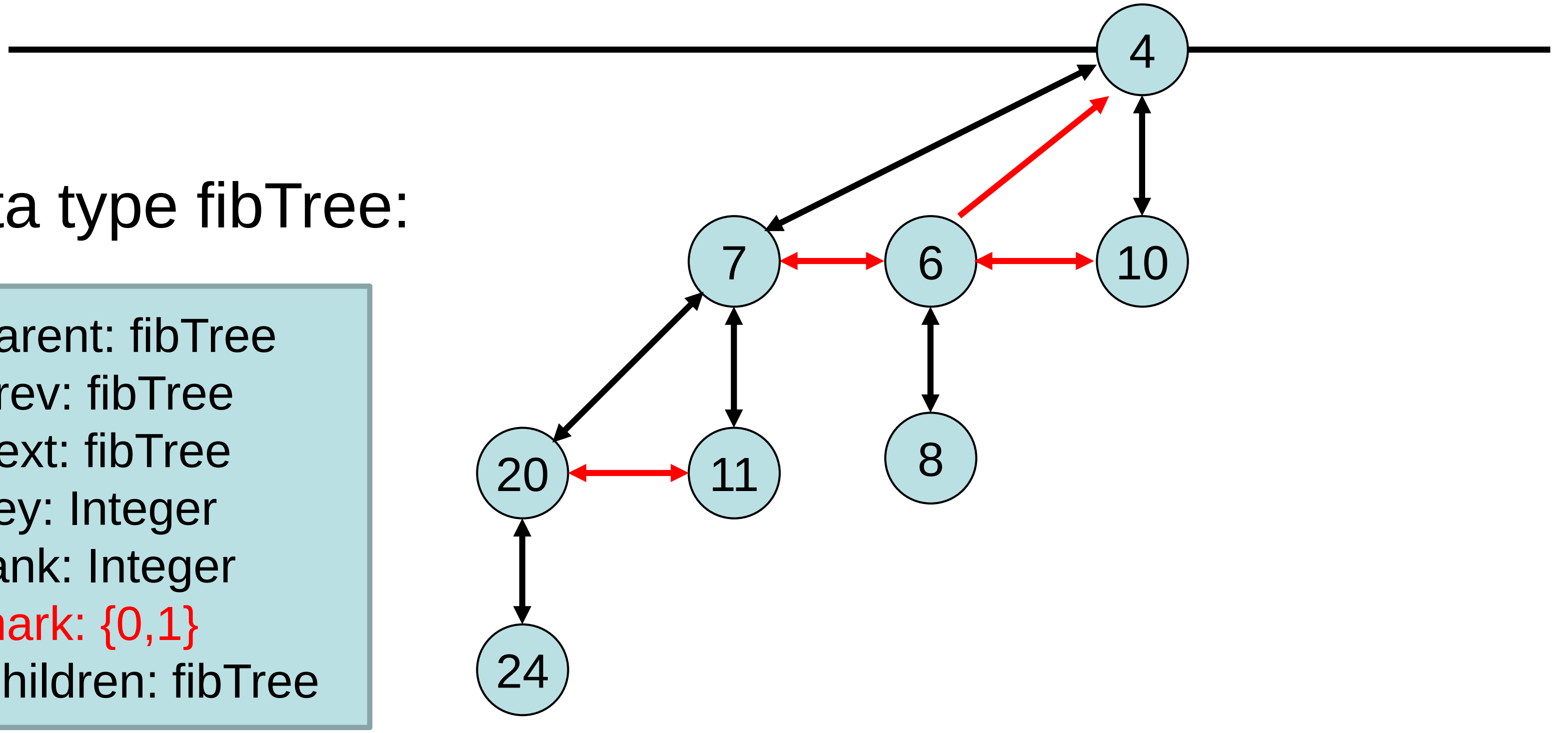
Fibonacci Heap

Tree in a Fibonacci heap:



Fibonacci Heap

Tree in a Fibonacci heap:

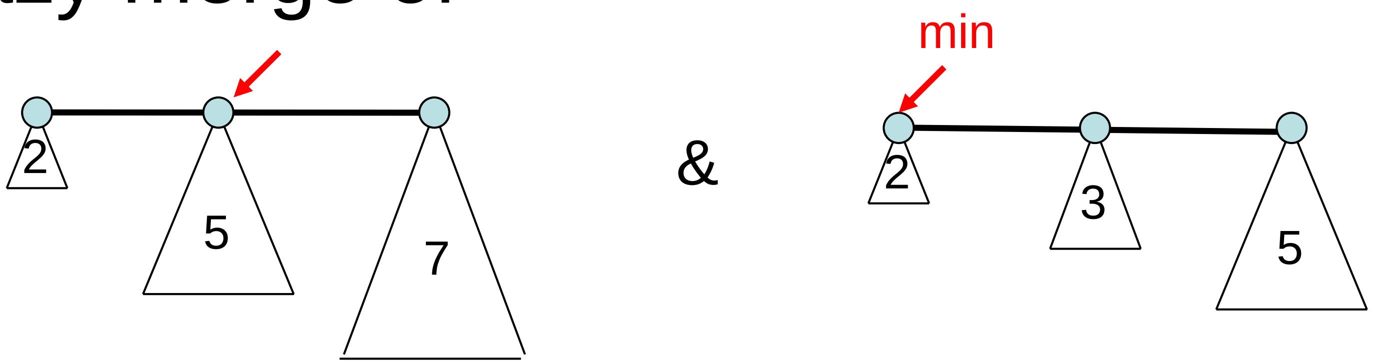


Data type fibTree:

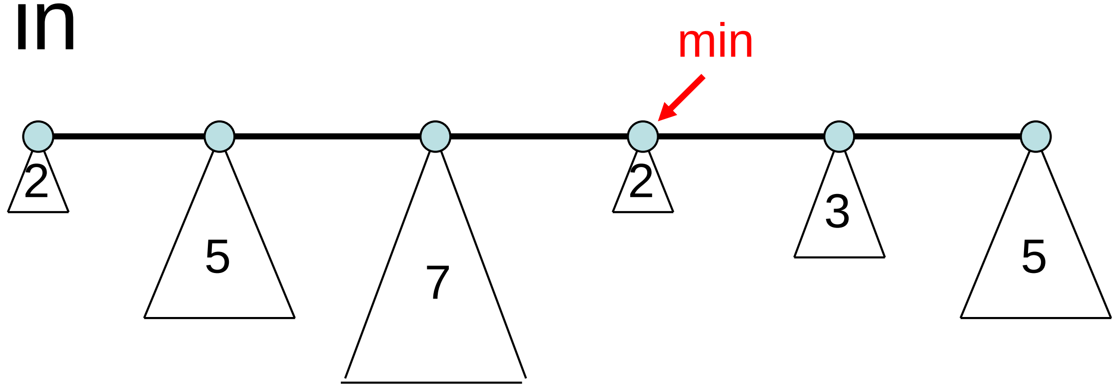
parent: fibTree
prev: fibTree
next: fibTree
key: Integer
rank: Integer
mark: {0,1}
Children: fibTree

Fibonacci Heap

Lazy merge of

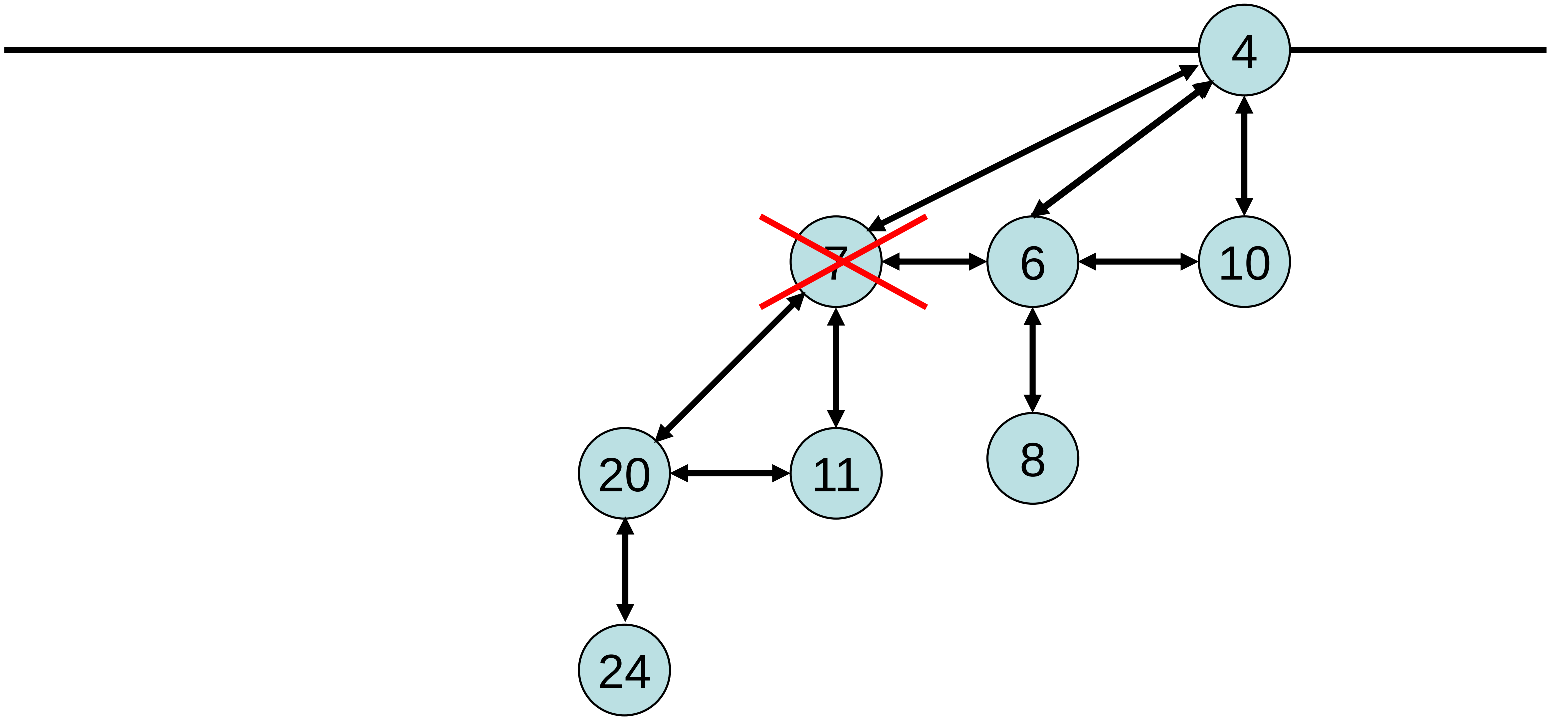


results in



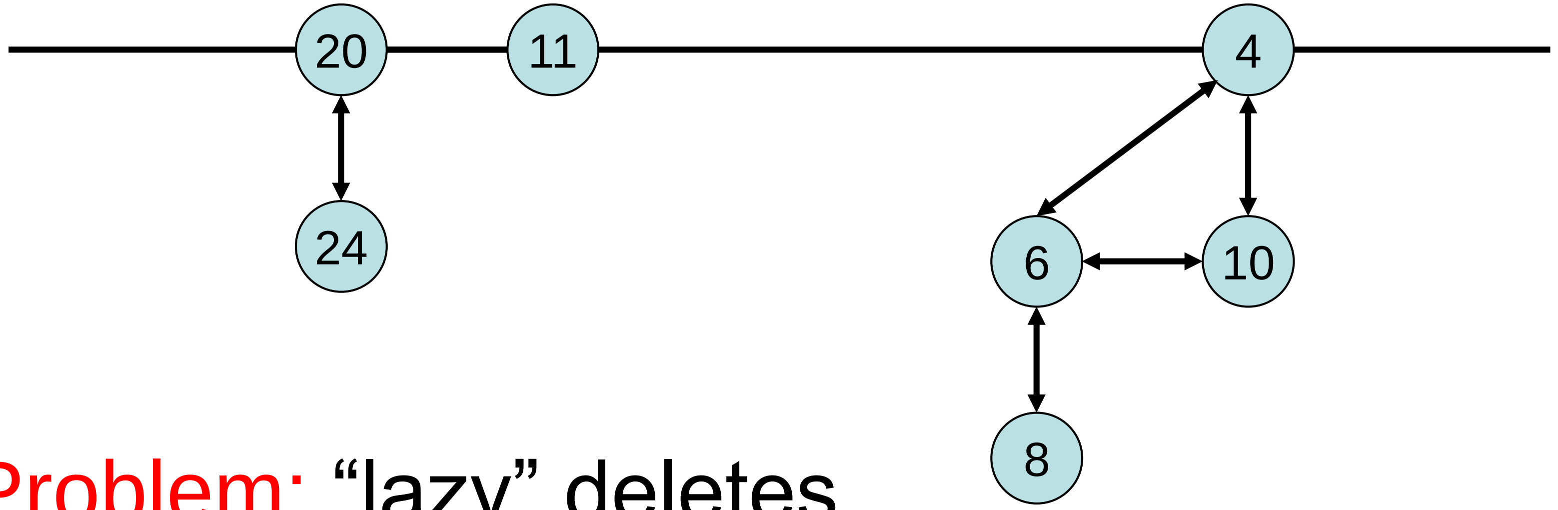
Fibonacci Heap

Lazy delete:



Fibonacci Heap

Lazy delete:



Problem: “lazy” deletes
(not deleteMin!) should not happen “too often”
without a cleanup step
→ use new variable mark to keep track

Fibonacci Heap

For any node v in the Fibonacci heap:

- parent(v) points to the parent of v (if v is a root, then parent(v)=⊥)
- prev(v) and next(v) connect v to its preceding and succeeding siblings
- key(v) stores the key of v
- rank(v) is equal to the number of children of v
- mark(v) stores whether v has lost a child from a lazy delete (unless v is a root node, in which case where mark(x)=0)
- Children(v) points to the first child in the childlist of v (this is sufficient for the data structure, but for the formal presentation of the Fibonacci heap we assume that v knows the first and last child in its childlist)

parent: fibTree
prev: fibTree
next: fibTree
key: Integer
rank: Integer
mark: {0,1}
Children: fibTree

Fibonacci Heap

Fibonacci heap is a list of Fibonacci trees

Fibonacci tree has to satisfy:

- Form invariant:
  Every node of rank r has exactly r children.
- Heap invariant:
  For every node v, key(v)≤key(children of v).
  The min-pointer points to the minimal key among all keys in the Fibonacci heap.

Fibonacci Heap

Operations:

- merge: concatenate root lists, update min-pointer.
  Time $O(1)$
- insert(x): add x as B_0 (with mark(x)=0) to root list, update min-pointer. Time $O(1)$
- min(): output element that the min-pointer is pointing to. Time $O(1)$
- deleteMin(), delete(x), decreaseKey(x,Δ): to be determined…

Fibonacci Heap

deleteMin(): This operation will clean up the Fibonacci heap. Let the min-pointer point to x.

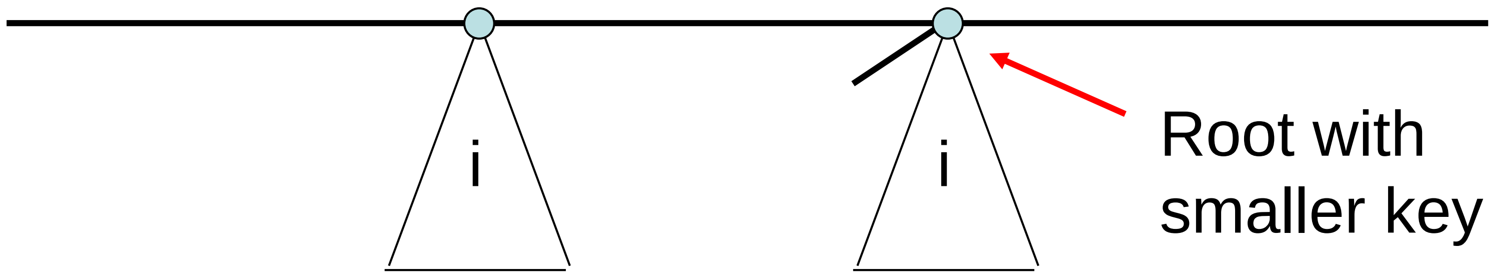
Algorithm deleteMin():

- remove x from root list
- for every child c in child list of x, set parent(c):=⊥ and mark(c):=0 // mark not needed for root nodes
- integrate child list of x into root list
- while ≥2 trees of the same rank i do
  merge trees to a tree of rank i+1
  (like with two Binomial trees)
- update min-pointer

Fibonacci Heap

Merging of two trees of rank i
(i.e., root has i children):

i+1 children, thus rank i+1

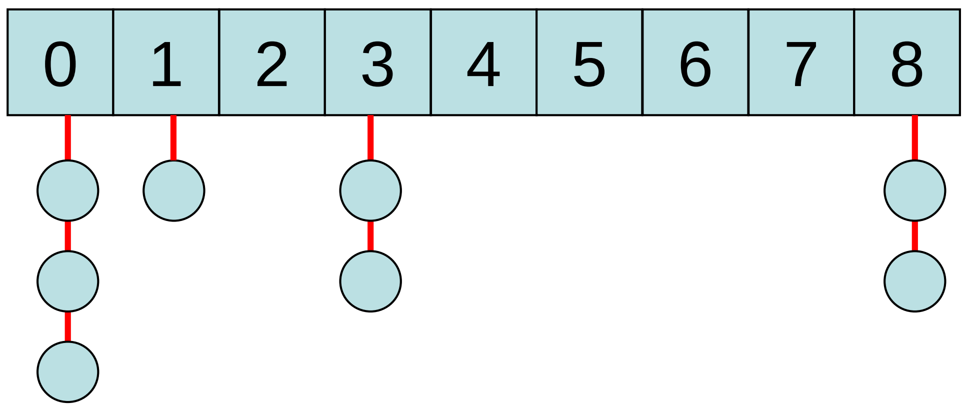


Fibonacci Heap

Efficient searching for roots of the same rank:

- Before executing the while-loop, scan all roots and store them according to their rank in an array:

Rank:



- Merge like for Binomial trees starting with rank 0 until the maximum rank has been reached (like binary addition)

Fibonacci Heap

Ideas behind delete(x) operation:

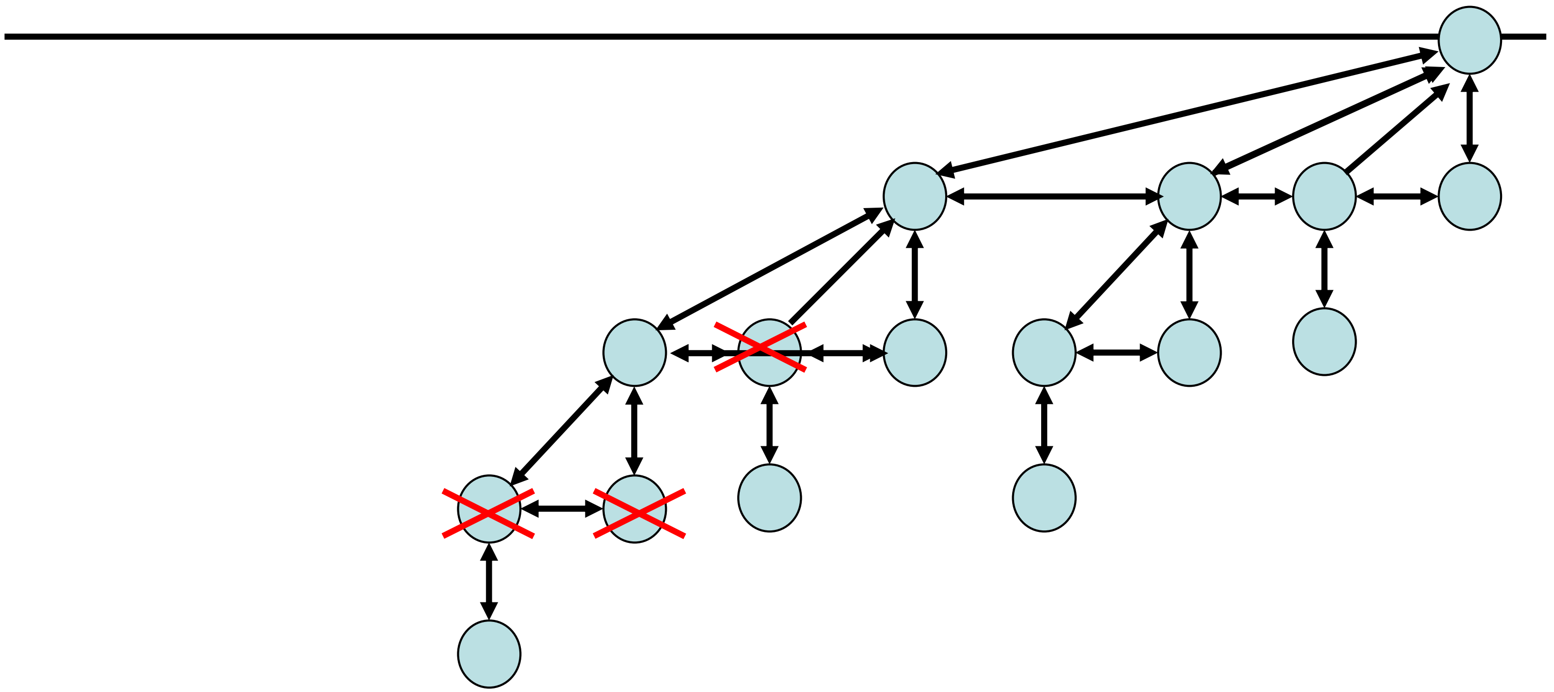
- Like deleteMin(), except:
  - Since node being deleted is potentially not a root, need to use the mark variables now
  - Each time a node v loses a *second* child, v is promoted to a separate tree in the root list of the heap
  - No “cleanup” or “consolidation step” based on ranks as for deleteMin() is performed.

Fibonacci Heap

```
Algorithm delete(x):
   if x is min-root then deleteMin()
   else
      y:=parent(x)
      delete x
      for every child c in child list of x, set parent(c):=⊥ and
      mark(c):=0
      add child list to root list
      while y≠NULL do      // parent node of x exists
         rank(y):=rank(y)-1  // one more child gone
         if parent(y)=⊥ then return // y is root node: done
         if mark(y)=0 then { mark(y):=1; return }
         else      // mark(y)=1, so one child already gone
            x:=y; y:=parent(x)
            move x with its subtree into the root list
            parent(x):=⊥; mark(x):=0  // roots do not need mark
```

Fibonacci Heap

Example for delete operations: (● : mark=1)



Fibonacci Heap

```
Algorithm decreaseKey(x,Δ):
  y:=parent(x)
  move x with its subtree into root list
  parent(x):=NULL; mark(x):=0
  key(x):=key(x)-Δ
  update min-pointer
  while y≠NULL do        // parent node of x exists
      rank(y):=rank(y)-1  // one more child gone
      if parent(y)=NULL then return // y is root node: done
        if mark(y)=0 then { mark(y):=1; return }
        else    // mark(y)=1, so one child already gone
          x:=y; y:=parent(x)
          move x with its subtree into the root list
          parent(x):=NULL
          mark(x):=0  // roots do not need mark
```

Fibonacci Heap

Runtime:

- deleteMin(): O(max. rank + #tree mergings)
- delete(x): O(max. rank + #cascading cuts)
  i.e., #relocated marked nodes
- decreaseKey(x,Δ): O(1 + #cascading cuts)

We will see: runtime of deleteMin can reach $\Theta(n)$, but on average over a sequence of operations much better (even in the worst case).

Amortized Analysis

Consider a sequence of n operations on an initially empty Fibonacci heap.

- Sum of individual worst case costs too high!
- Average-case analysis does not mean much
- Better: amortized analysis, i.e., average cost of operations in the worst case (i.e., a sequence of operations with overall maximum runtime)

Amortized Analysis

Recall:

Theorem 1.5: Let S be the state space of a data structure, s_0 be its initial state, and let $\phi:S\rightarrow\mathbb{R}_{\geq 0}$ be a non-negative function. Given an operation X and a state s with $s \xrightarrow{X} s'$, we define

$$A_X(s) := T_X(s) + (\phi(s') - \phi(s)).$$

Then the functions $A_X(s)$ are a family of amortized time bounds.

Amortized Analysis

For Fibonacci heaps we will use the potential function

bal(s):= #trees + 2·#marked nodes in in state s

node v marked: mark(v)=1

But: Before we do amortized analysis, useful to understand ranks and sizes of subtrees in heap.

Fibonacci Heap

Lemma 2.1: Let x be a node in the Fibonacci heap with $rank(x)=k$. Let the children of x be sorted in the order in which they were added below x. Then the rank of the i-th child is $\geq i-2$.

Proof:

- When the i-th child is added, $rank(x)=i-1$.
- Only step which can add i-th child is "consolidation step" of deleteMin. Thus, the i-th child must have also had rank $i-1$ at this time.
- Afterwards, the i-th child loses at most one of its children, i.e., its rank is $\geq i-2$. (Why?)

Fibonacci Heap

Theorem 2.2: Let x be a node in the Fibonacci heap with $rank(x)=k$. Then the subtree with root x contains at least F_{k+2} elements, where F_k is the k-th Fibonacci number.

Definition of Fibonacci numbers:

- $F_0 = 0$ and $F_1 = 1$
- $F_k = F_{k-1}+F_{k-2}$ for all $k>1$

One can prove: $F_{k+2} = 1 + \sum_{i=1}^{k} F_i$.

Fibonacci Heap

Proof of Theorem 2.2:

- Let f_k be the minimal number of elements in a tree of rank k.
- From Lemma 2.1 we get:

1. child

$$f_k \geq (f_{k-2}+f_{k-3}+\ldots+f_0 + 1) + 1$$

root

- Moreover, $f_0=1$ and $f_1=2$
- It follows from the Fibonacci numbers:

$$f_k \geq F_{k+2}$$

Fibonacci Heap

- It is known that $F_{k+2} > \Phi^{k+2}$ with

$$\Phi=(1+\sqrt{5})/2 \approx 1{,}618034$$

- Hence, a tree of rank k in the Fibonacci heap contains at least $1{,}61^{k+2}$ nodes.
- Therefore, a Fibonacci heap with n elements contains trees of rank at most $O(\log n)$ (like in a Binomial heap)

Fibonacci Heap

- t_i: time for operation i
- bal_i: value of $bal(s)$ after operation i
  ($bal(s)$ = #trees + 2·#marked nodes)
- a_i: amortized runtime of operation i
  $a_i = t_i + \Delta bal_i$ with $\Delta bal_i = bal_i - bal_{i-1}$

Amortized runtime of operations:

- insert: $t=O(1)$ and $\Delta bal=+1$, so $a=O(1)$
- merge: $t=O(1)$ and $\Delta bal=0$, so $a=O(1)$
  - #trees before merge = total #trees in both heaps
- min: $t=O(1)$ and $\Delta bal=0$, so $a=O(1)$

Fibonacci Heap

Let H_i denote the heap after operation i.

Theorem 2.3: The amortized runtime of deleteMin() is O(log n).
Proof:

- Actual cost: $t_i = O(rank(x) + \#trees(H_{i-1}))$. Why?
  - Move children of x to separate trees in heap: O(rank(x))
  - Consolidate $O(rank(x) + \#trees(H_{i-1}))$ trees: $O(rank(x) + \#trees(H_{i-1}))$
  - Update min-pointer: $O(rank(x) + \#trees(H_{i-1}))$
  - Theorem 2.2 says $rank(x) = O(logn)$

- Potential function before deleteMin():
  - $bal_{i-1} = \#trees(H_{i-1}) + 2\#markednodes(H_{i-1})$ (by def.)
  - $bal_i \leq O(logn) + 2\#markednodes(H_{i-1})$. Why?
    - deleteMin() can only *un*mark nodes
    - Consolidation step creates heap with *unique* root ranks. Theorem 2.2 implies $\#trees(H_{i-1}) \leq O(logn)$.
- Amortized cost: $a_i = t_i + bal_i - bal_{i-1} = O(logn)$.

Fibonacci Heap

Theorem 2.4: Amortized runtime of delete(x) is $O(\log n)$.

Proof: (x is not the min-element – otherwise like Th. 2.3)

- Insertion of child list of x into root list:
  $\Delta bal \leq rank(x)$
- Every cascading cut (i.e., relocation of a marked node) increases the number of trees by 1:
  $\Delta bal =$ #cascading cuts
- Every cascading cut removes one marked node:
  $\Delta bal = -2\cdot$#cascading cuts
- The last cut possibly introduces a new marked node:
  $\Delta bal \in \{0,2\}$

Fibonacci Heap

Theorem 2.4: The amortized runtime of delete(x) is $O(\log n)$.

Proof:

- Altogether:
  $\Delta bal_i \leq$ rank(x) $-$ #cascading cuts $+ O(1)$
  $= O(\log n) -$ #cascading cuts
  because of Theorem 2.2
- Real runtime (in appropriate time units):
  $t_i = O(\log n) +$ #cascading cuts
- Amortized runtime:
  $a_i = t_i + \Delta bal_i = O(\log n)$

Fibonacci Heap

Thm 2.5: Amortized runtime of decreaseKey(x,Δ) is O(1).

Proof:

- Every cascading cut increases the number of trees by 1:
  $\Delta bal =$ #cascading cuts
- Every cascading cut removes a marked node:
  $\Delta bal \leq -2\cdot$#cascading cuts
- The last cut possibly creates a new marked node:
  $\Delta bal \in \{0,2\}$
- Altogether: $\Delta bal_i = -$ #cascading cuts $+ O(1)$
- Real runtime: $t_i =$ #cascading cuts $+ O(1)$
- Amortized runtime: $a_i = t_i + \Delta bal_i = O(1)$

Summary

Runtime	Binomial Heap	Fibonacci Heap
insert	O(log n)	O(1)
min	O(1)	O(1)
deleteMin	O(log n)	O(log n) amor.
delete	O(log n)	O(log n) amor.
merge	O(log n)	O(1)
decreaseKey	O(log n)	O(1) amor.

Summary

Great, but… *can we do better?*

Yes… *if we're willing to make assumptions about the input*

Radix Heap

Assumptions:

1. At all times, maximum key – minimum key <= constant C. (Think of fixed architecture, like 32-bit ints.)
2. Insert(e) only inserts elements e with $key(e) \geq k_{min}$ (k_{min}: minimum key).

The priority queue we implement is called a "monotone" priority queue, i.e. top-priority element's key monotonically increases.

Radix Heap

Idea:

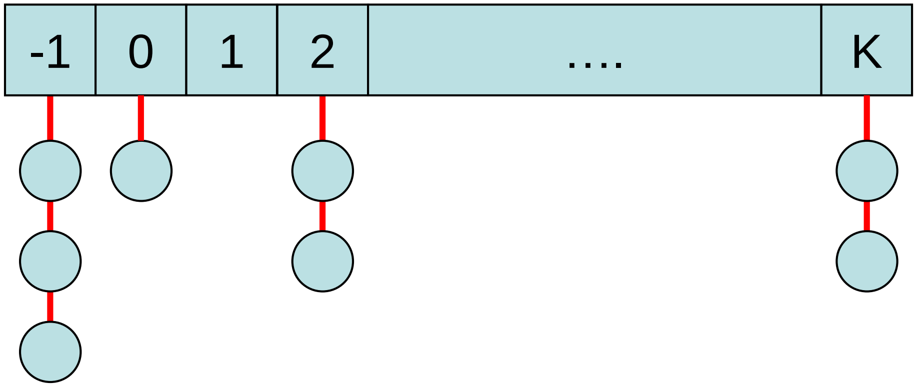
Define $K = \lceil \log C \rceil$.

Two integers x and y s.t. $|x - y| \leq C \leq 2^K$ must agree on all bits after (i.e. more significant than) K.

Thus: suffices to keep track of first K bit positions.

Radix Heap

Let B[-1..K] be array of lists B[-1] to B[K], where $K = \lceil \log C \rceil$.



Invariant: Each e stored in B[msd(k_{min},key(e))]

- msd(k_{min},key(e)):
  - most significant bit for which binary representations of k_{min} and key(e) differ (-1: no difference)
  - If $k_{min} = -\infty$ (heap empty), msd returns -1.

Radix Heap

Example for $msd(k_{min},k)$:

- let $k_{min}=17$, or in binary form, 10001
- $k=17$: $msd(k_{min},k)=-1$
- $k=18$: in binary 10010, so $msd(k_{min},k)=1$
- $k=21$: in binary 10101, so $msd(k_{min},k)=2$
- $k=52$: in binary 110100, so $msd(k_{min},k)=5$

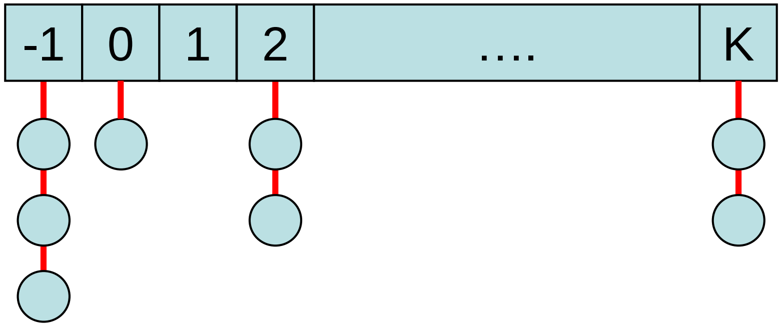
Computation of msd for $a \neq b$:

$$msd(a,b)=\lfloor \log(a \oplus b) \rfloor$$

where $\oplus$ denotes bit-wise xor.

Time: $O(1)$ (with appropriate machine instruction set)

Radix Heap



min():

- output k_{min} in B[-1]

Runtime: O(1)

Radix Heap

insert(e): ($key(e) \geq k_{min}$)

- $i := msd(k_{min}, key(e))$
- store e in B[i]

Runtime: O(1)

delete(e): ($key(e) > k_{min}$, otherwise call deleteMin())

- Remove e from its list B[j]

Runtime: O(1) (assuming have pointer to e)

decreaseKey(e,Δ): ($key(e) - \Delta \geq k_{min}$, $\Delta > 0$)

- call delete(e) and insert(e) with $key(e) := key(e) - \Delta$

Runtime: O(1)

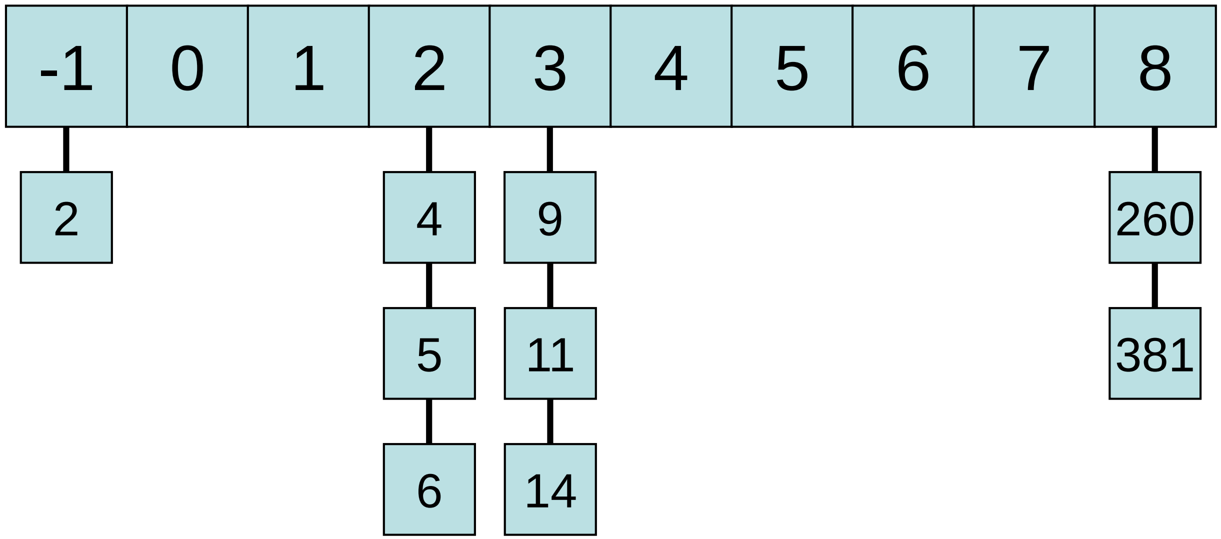
Radix Heap

deleteMin():

- if B[-1] is unoccupied, heap is empty, we are done
- else, remove some e from B[-1]
- find minimal i so that $B[i] \neq \emptyset$ (if there is no such i or i=-1 then we are done)
- determine k_{min} in B[i]
- distribute nodes in B[i] among B[-1],…,B[i-1] w.r.t. the new k_{min}

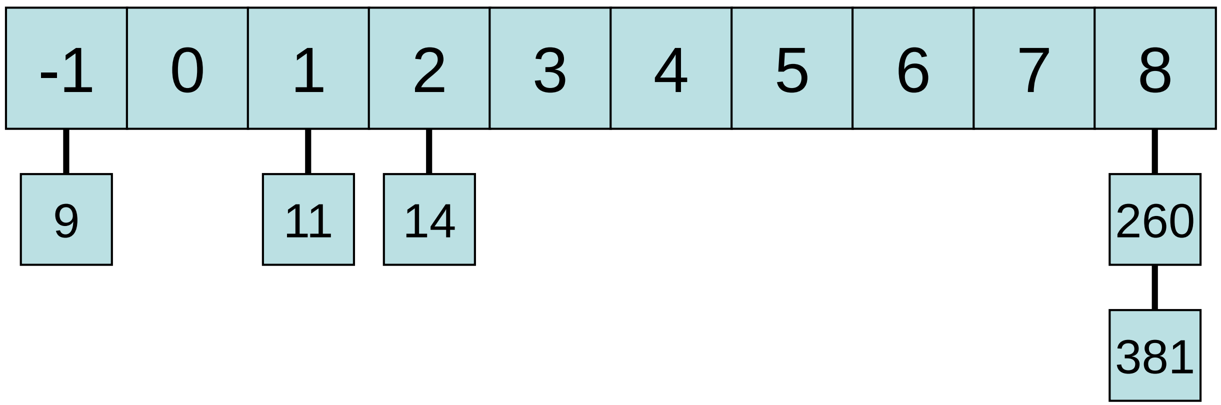
Question: What about the bins B[j] for j>i? Do their elements need to be moved as well?

Radix Heap



We consider a sequence of deleteMin operations

Radix Heap



We consider a sequence of deleteMin operations

Radix Heap

Claim: In deleteMin(), after we distribute nodes in B[i] among B[-1],…,B[i-1] w.r.t. the new k_{min}, all nodes e in B[j], j>i do not have to be moved, i.e. msd(k_{min},key(e))=j.

Proof:

- Assume the new min element is to be drawn from B[i].
- By def, B[i] agrees with (the old) k_{min} on all bits > i, but disagrees on bit i.
- Similarly, B[j] for j>i agrees with k_{min} on all bits > j, but disagrees on bit j.
- By transitivity, B[i], B[j] hence agree on all bits > j, and they disagree on bit j.
- Thus, msd(B[i],B[j]) =j.

Radix Heap

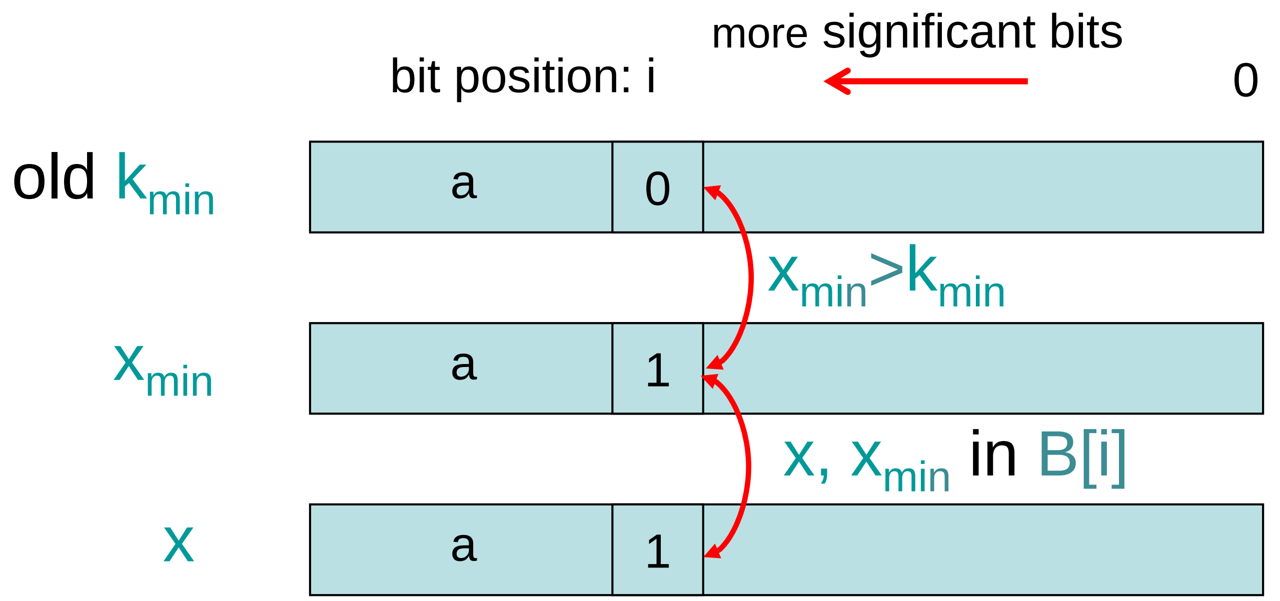
In illustration, *all* elements in new minimal list B[i] were moved (when i>=0) with each deleteMin() call. Let's prove this holds!

Lemma 2.6: Let B[i] be the minimal non-empty list, $i \geq 0$. Let x_{min} be the minimal key in B[i]. Then $msd(x_{min},x)<i$ for all keys x in B[i].

Proof:

- Consider any x in B[i].
- If $x=x_{min}$: x placed in B[-1], so claim holds.
- What if $x \neq x_{min}$?

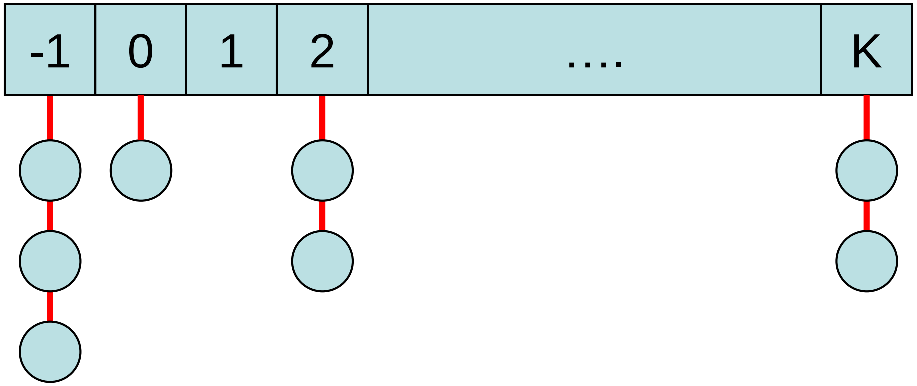
Radix Heap



- By assumption, k_{min} , x_{min} , x agree on all bits after K.
- Since x, x_{min} in B[i], they agree on bits i to K.
- Thus, $msd(x_{min},x)<i$.

Radix Heap

- Lemma 2.6: Let B[i] be the minimal non-empty list, i≥0. Let x_{min} be the minimal key in B[i]. Then $msd(x_{min},x)<i$ for all keys x in B[i].



Consequence:

- Each element can be moved at most K times (due to deleteMin or decreaseKey operations)
- insert(): amortized runtime O(K)=O(log C).
  (i.e. When an item is inserted, it „pays up front“ for later potentially needing to be moved K times)

Summary

Runtime	Fibonacci Heap	Radix Heap
insert	O(1)	O(log C) amor.
min	O(1)	O(1)
deleteMin	O(log n) amor.	O(1) amor.
delete	O(log n) amor.	O(1)
merge	O(1)	???
decreaseKey	O(1) amor.	O(1)

Extended Radix Heap

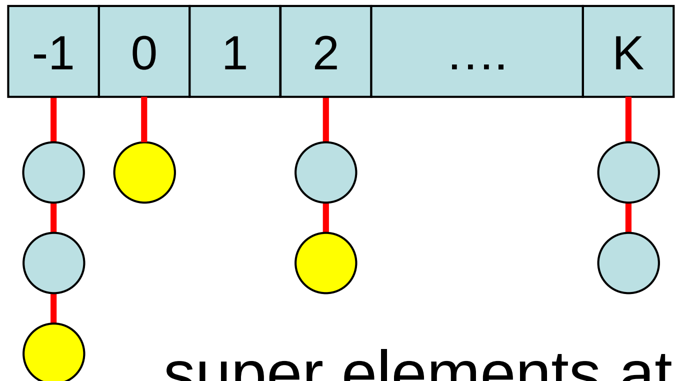
Assumptions:

1. At all times, maximum key – minimum key <= constant C. (Think of fixed architecture, like 32-bit ints.)
2. ~~Insert(e) only inserts elements e with key(e)~~$\cancel{\geq k_{min}}$ ~~(~~$\cancel{k_{min}}$~~: minimum key).~~

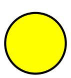
The priority queue we implement is called a “monotone” priority queue, i.e. top-priority element’s key monotonically increases.

Extended Radix Heap

At least one “normal” element in -1



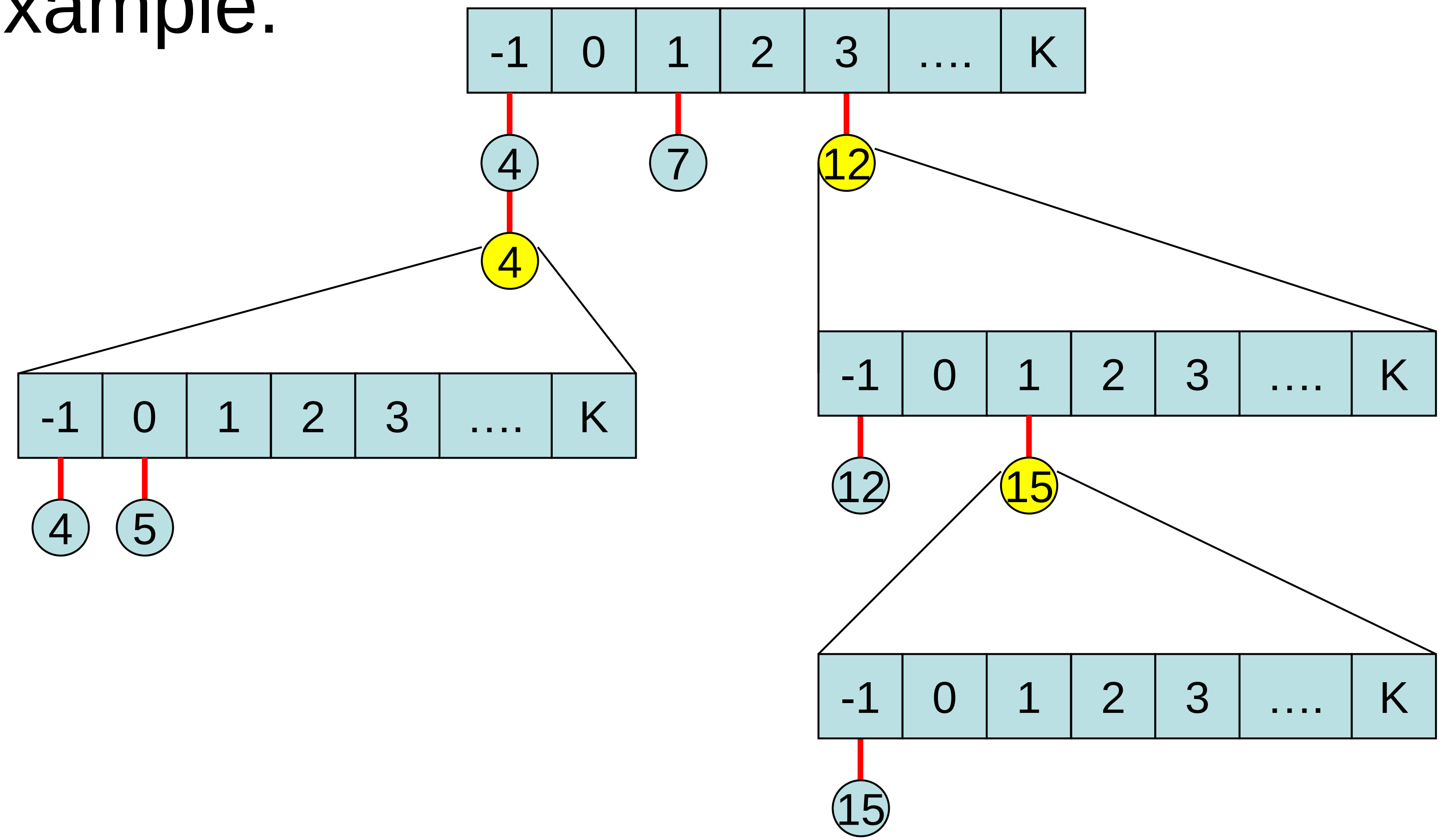
super elements at the end

 : “super element” e contains a Radix heap with k_{min}=key(e) where k_{min} is the smallest value in the Radix heap of e and $B_e[-1]$ has ≥ 1 “normal” element.

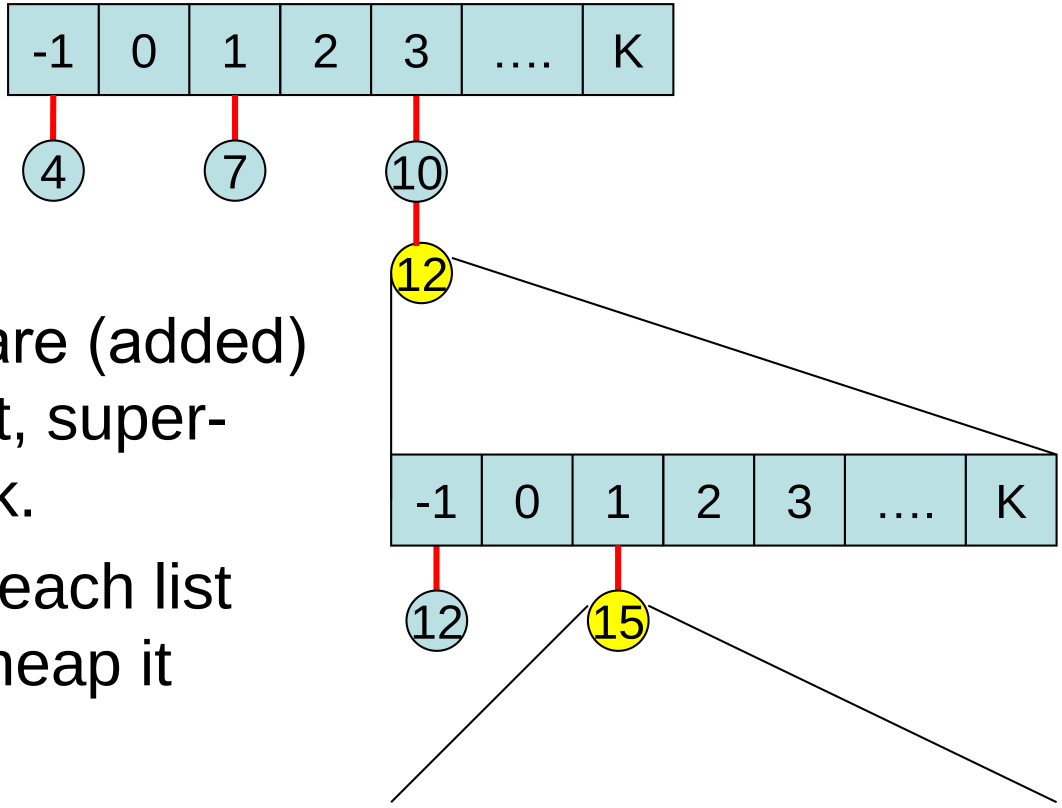
Note: super elements may contain super elements

Extended Radix Heap

Example:



Extended Radix Heap



Further details:

- Every list is doubly-linked.
- “Normal” elements are (added) at the front of the list, super-elements in the back.
- The first element of each list points to the Radix heap it belongs to.

Extended Radix Heap

Merge of two extended Radix heaps B and B' with $k_{min}(B) \leq k_{min}(B')$:

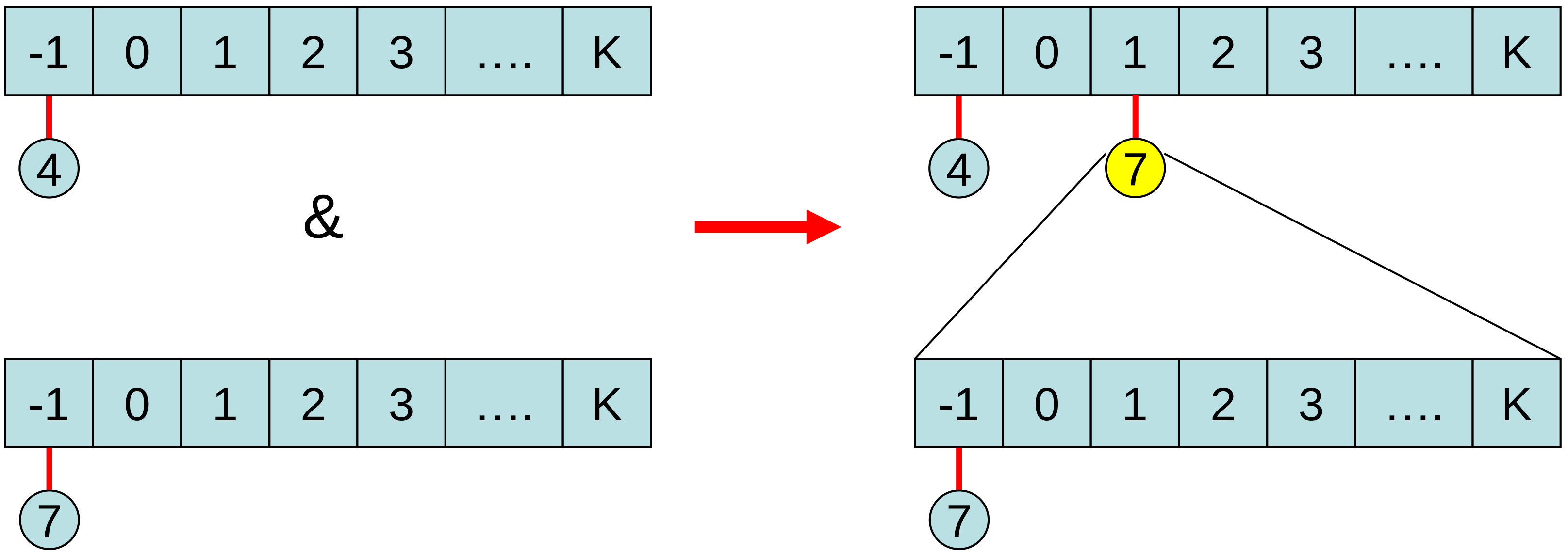
(Case $k_{min}(B) > k_{min}(B')$: flip B and B´)

- transform B' into a super element e with $key(e) = k_{min}(B')$
- call insert(e) on B

Runtime: $O(1)$

Extended Radix Heap

Example of a merge operation:



Extended Radix Heap

insert(e):

- $key(e) \geq k_{min}$: as in standard Radix heap
- else, merge extended Radix heap with a new Radix heap just containing e

Runtime: $O(1)$

min(): like in a standard Radix heap (note -1 bucket has at least one “normal” element)

Runtime: $O(1)$

Extended Radix Heap

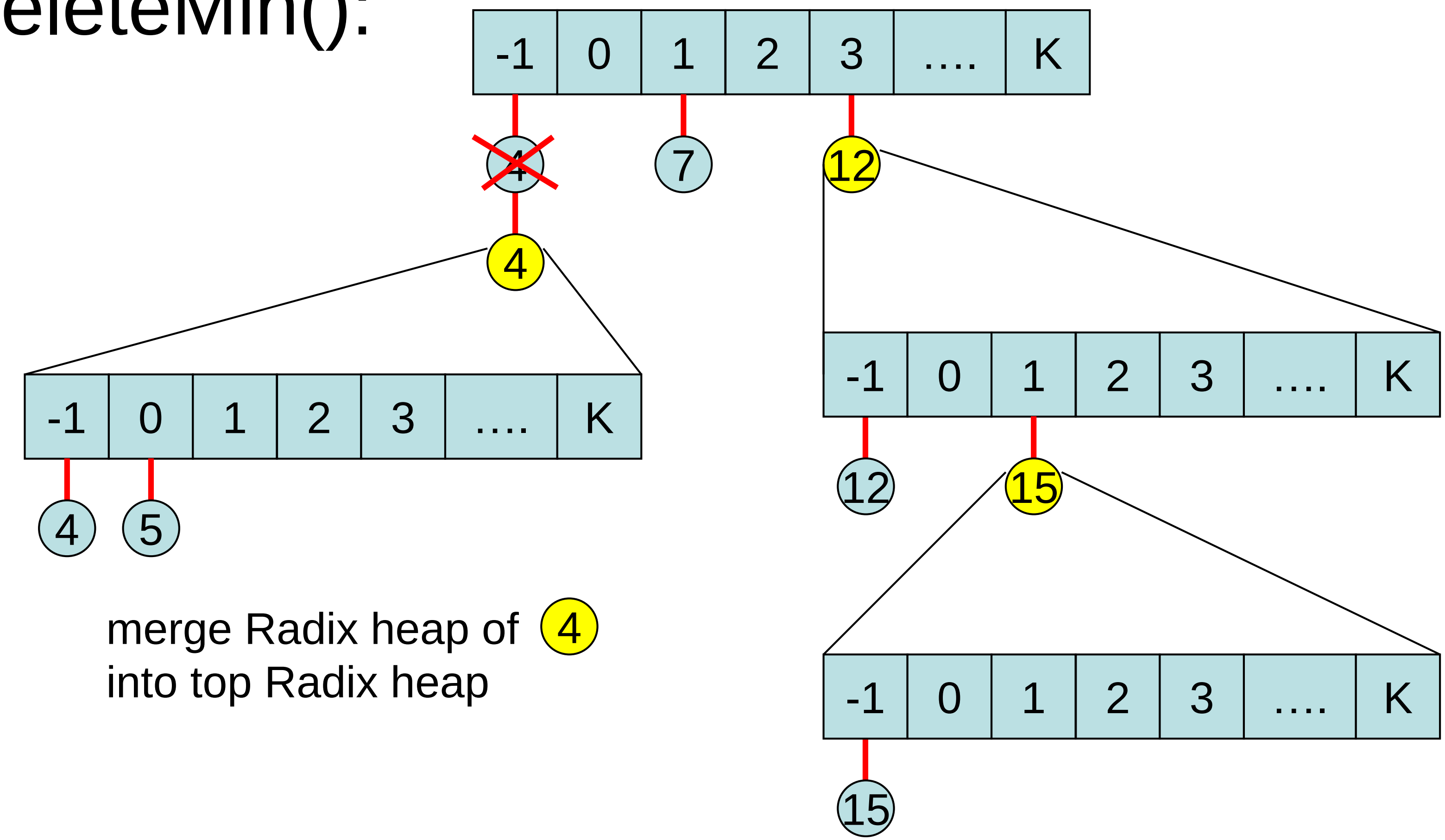
deleteMin():

- Remove normal element e from B[-1]
(B: Radix heap at highest level, i.e. “top” heap)
- If B[-1] does not contain any elements, then update B like in a standard Radix heap (i.e., dissolve smallest non-empty bucket B[i])
- If B[-1] does not contain normal elements any more, then take the first super element e’ from B[-1] and merge the lists of e’ with B
(then there is again a normal element in B[-1]!)

Runtime: O(log C) + time for updates

Extended Radix Heap

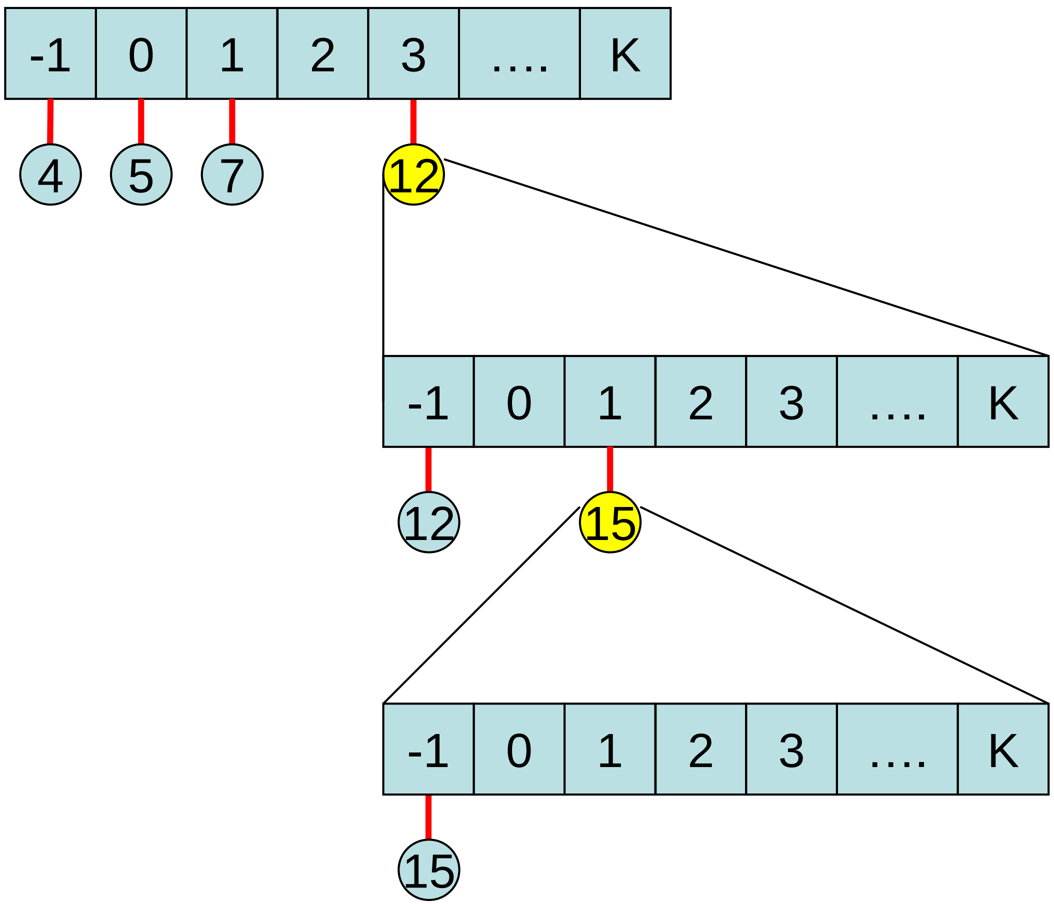
deleteMin():



merge Radix heap of 4
into top Radix heap

Extended Radix Heap

deleteMin():



Extended Radix Heap

delete(e):

Case 1: $key(e) > k_{min}$ for heap of e:

- like delete(e) in a standard Radix heap

Case 2: $key(e) = k_{min}$ for heap of e:

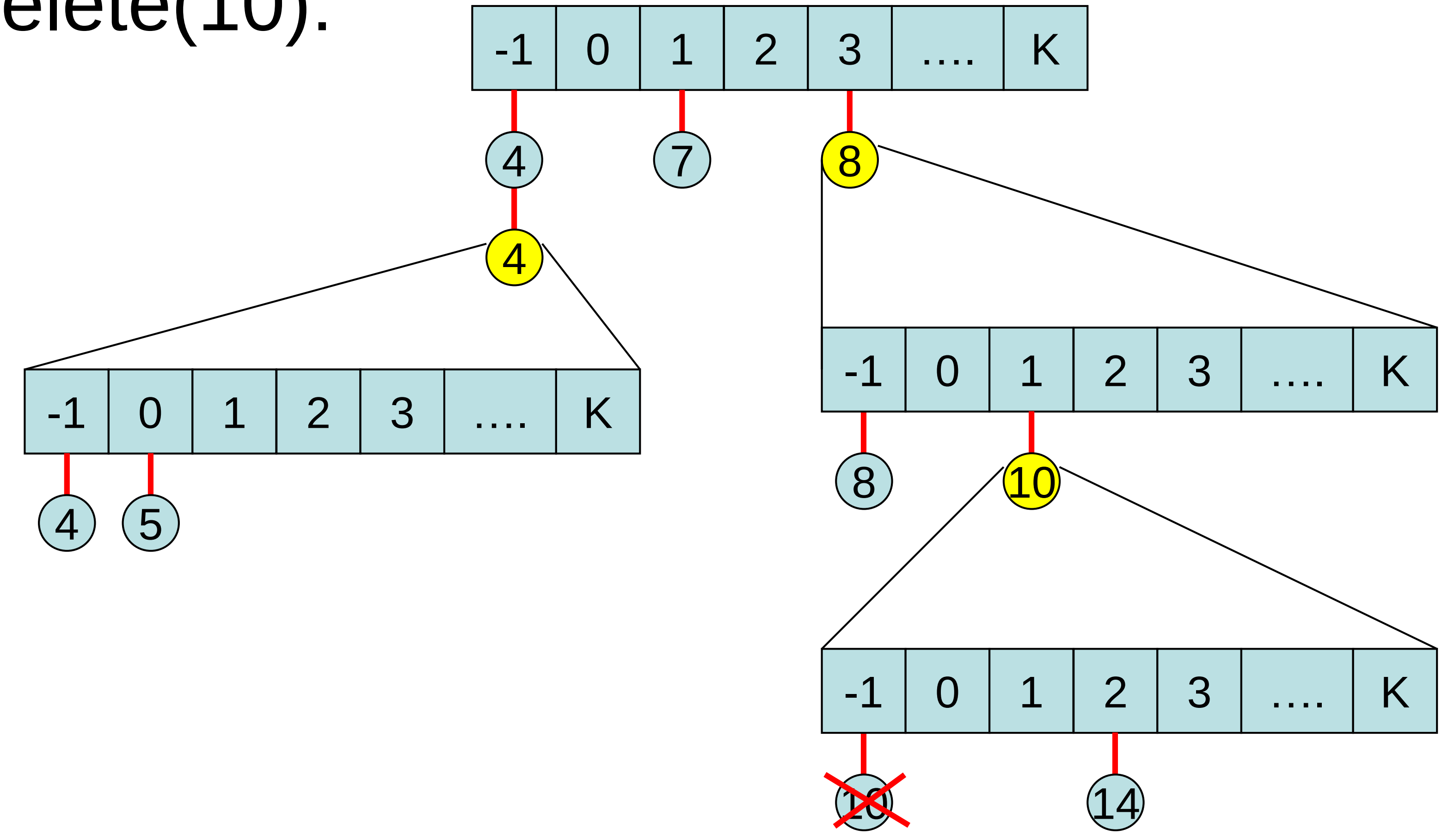
- if e is in “top” Radix heap, proceed like deleteMin()
- Else, e is in Radix heap of super element e’:
  - if e’ is afterwards empty, then remove e’ from heap B’ containing e’
  - if the minimum key in e’ has changed, then move e’ to its correct bin in B’

  Since there is a normal element in B’[-1], both cases have no cascading effects! (don’t have to recurse upwards)

Runtime: O(log C) + time for updates

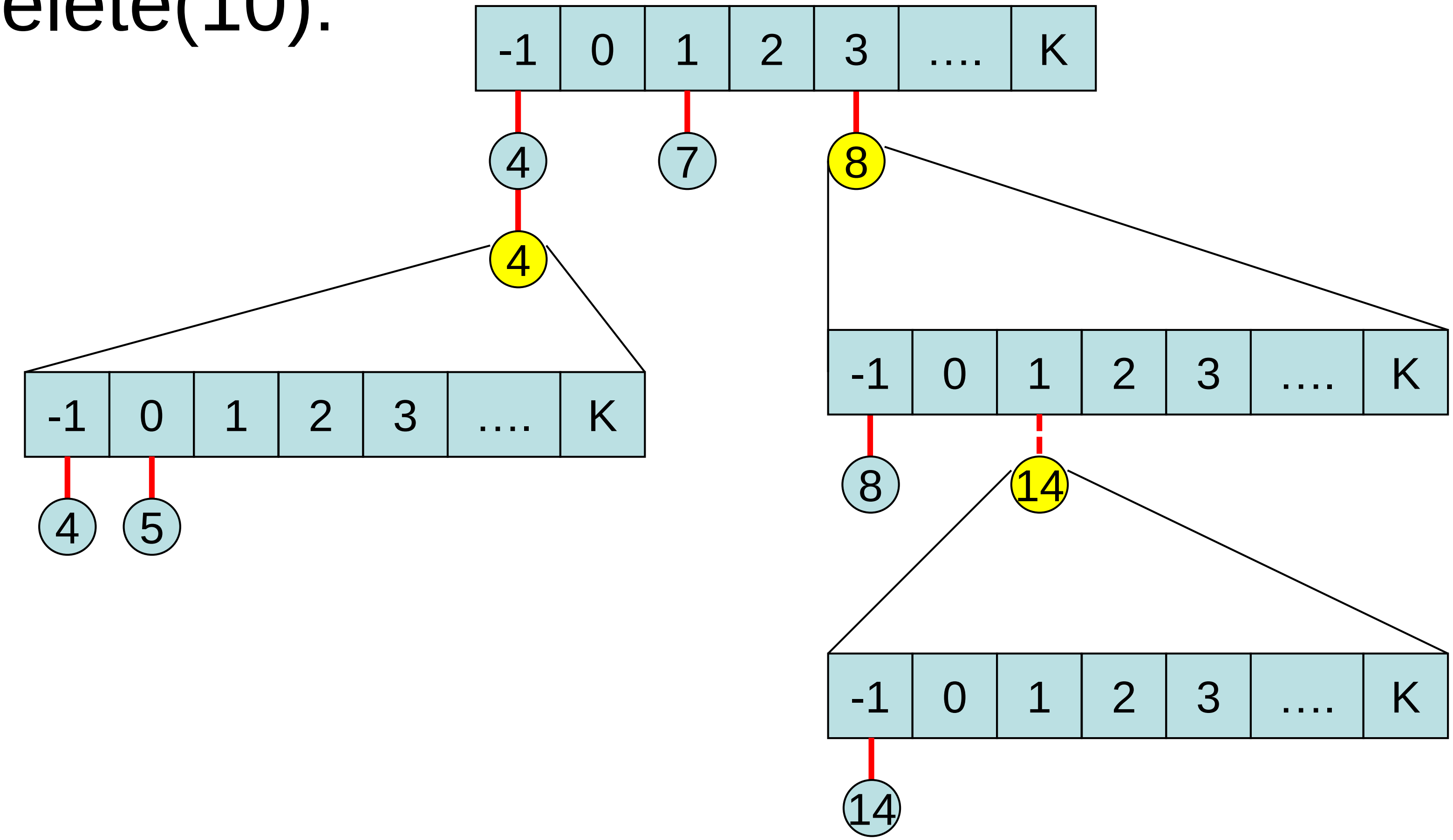
Extended Radix Heap

delete(10):



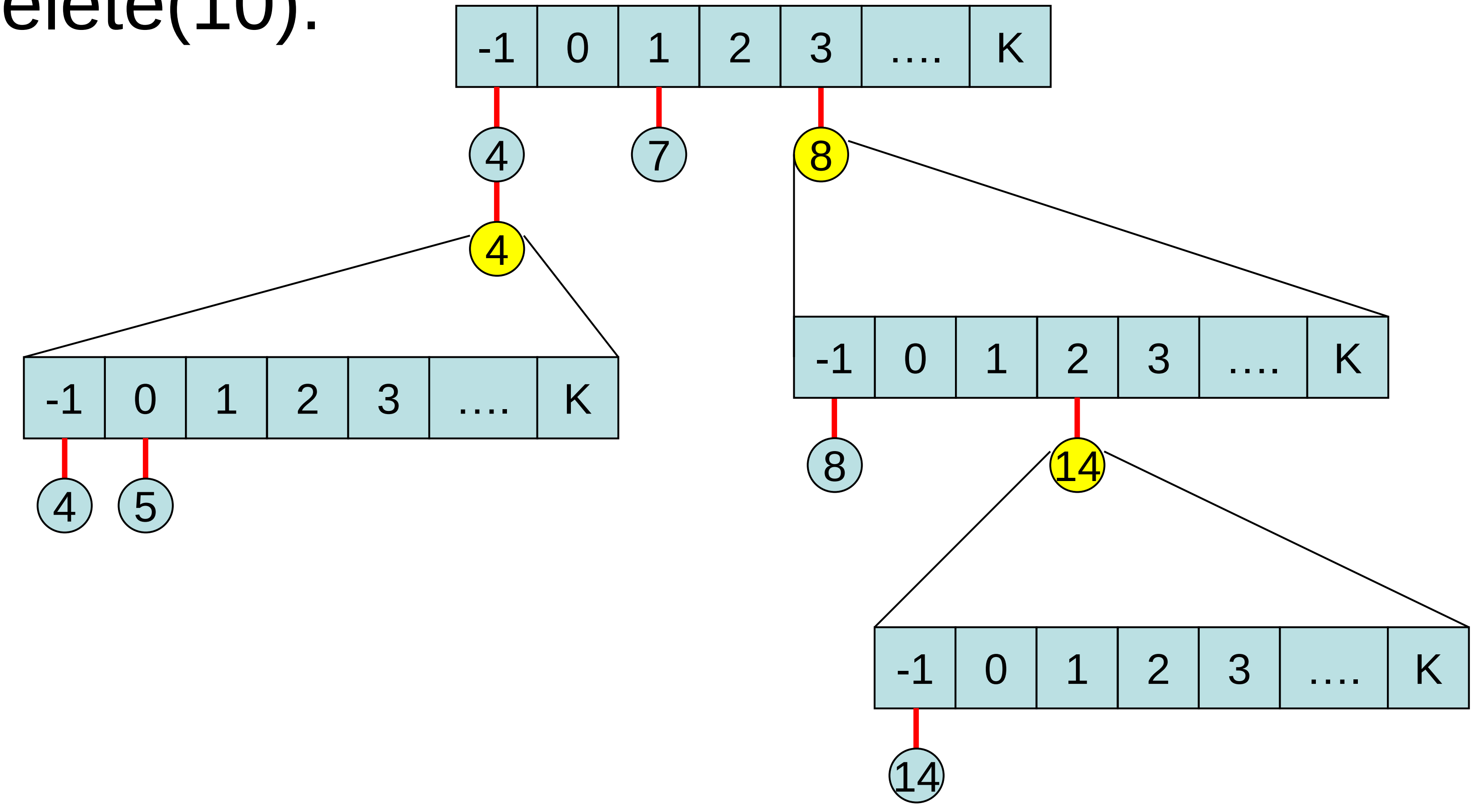
Extended Radix Heap

delete(10):



Extended Radix Heap

delete(10):



Extended Radix heap

decreaseKey(e,Δ): [precondition: key(e) - Δ >= k_{min}]

- call delete(e) in heap of e
- set key(e):=key(e)-Δ
- call insert(e) on "top" Radix heap

Runtime: O(log C) + time for updates

Amortized analysis: similar to Radix heap, omitted here

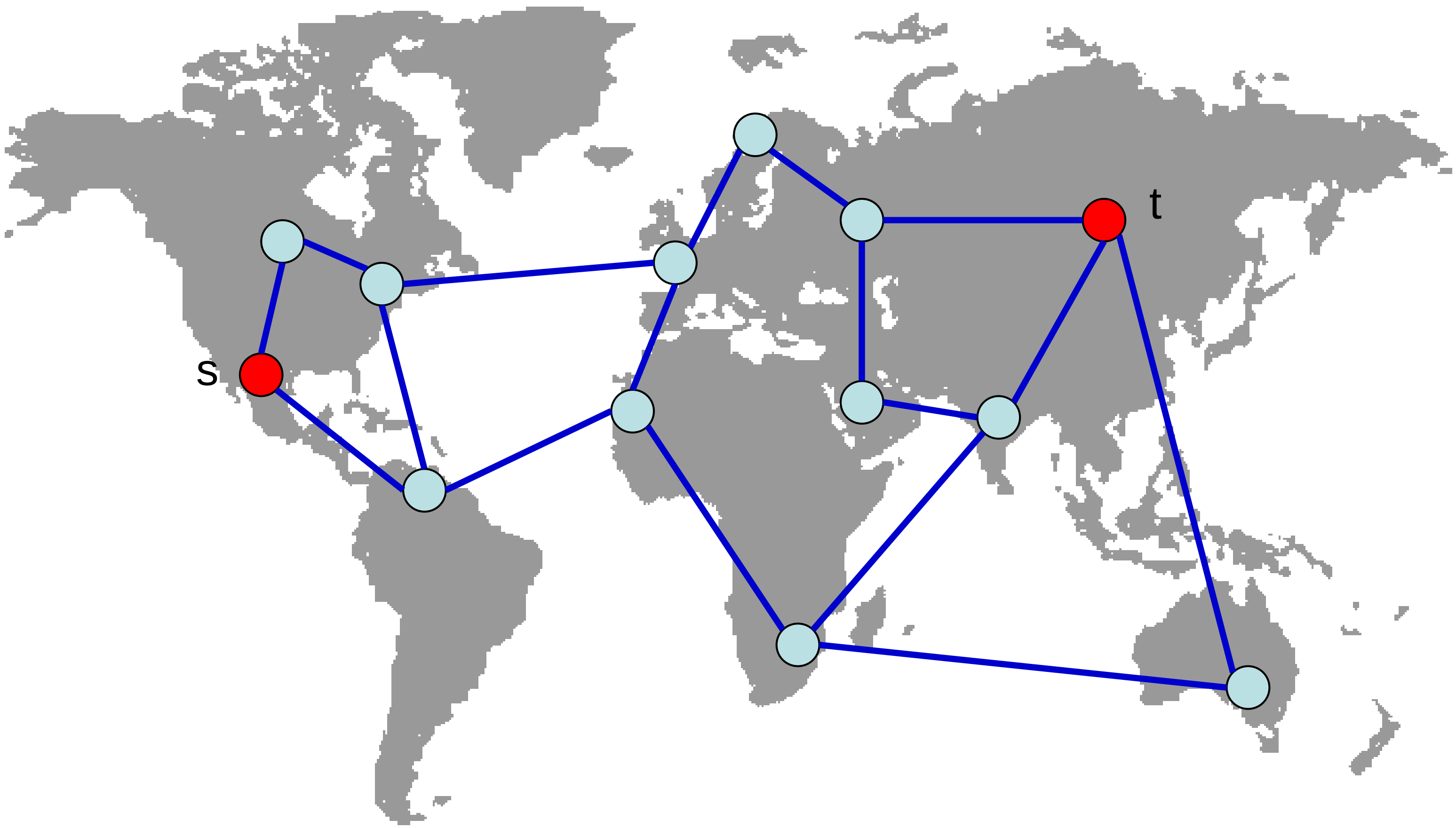
Summary

Runtime	Radix heap	ext. Radix heap
insert	O(log C) amor.	O(log C) amor.
min	O(1)	O(1)
deleteMin	O(1) amor.	O(1) amor.
delete	O(1)	O(1) amor.
merge	???	O(log C) amor.
decreaseKey	O(1)	O(log C) amor.

Contents

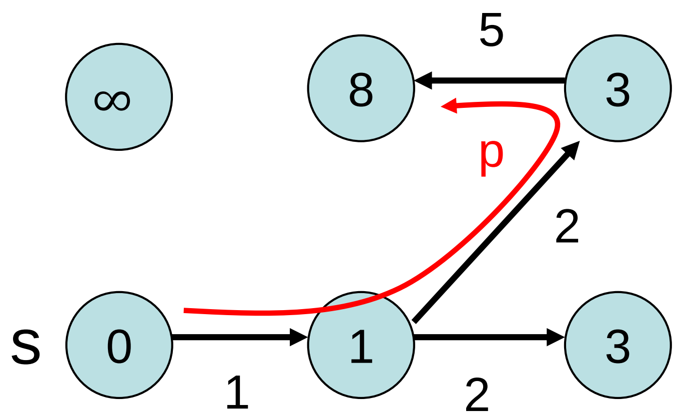
- Binomial heap
- Fibonacci heap
- Radix heap
- Applications

Shortest Paths



Central question: Determine fastest way to get from s to t.

Shortest Paths



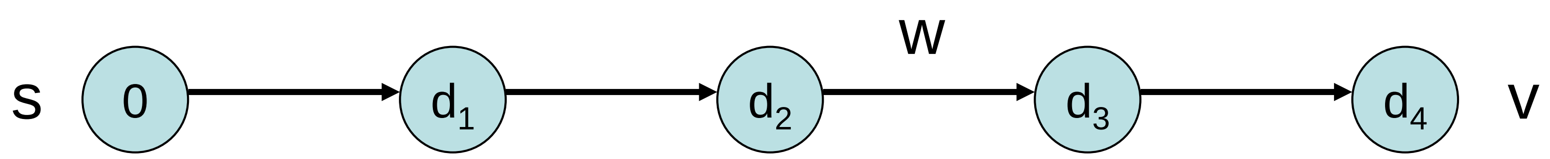
Cost of p:
$c(p)=1+2+5=8$

$d(s,v)$: distance from s to v

$$d(s,v) = \begin{cases} \infty & \text{no path from s to v} \\ \min\{\, c(p) \mid p \text{ is a path from } s \text{ to } v \,\} & \end{cases}$$

Dijkstra's Algorithm

Consider the single source shortest path problem (SSSP), i.e., find the shortest path from a source s to all other nodes, in a graph with arbitrary non-negative edge costs.



Basic idea behind Dijkstra´s Algorithm:

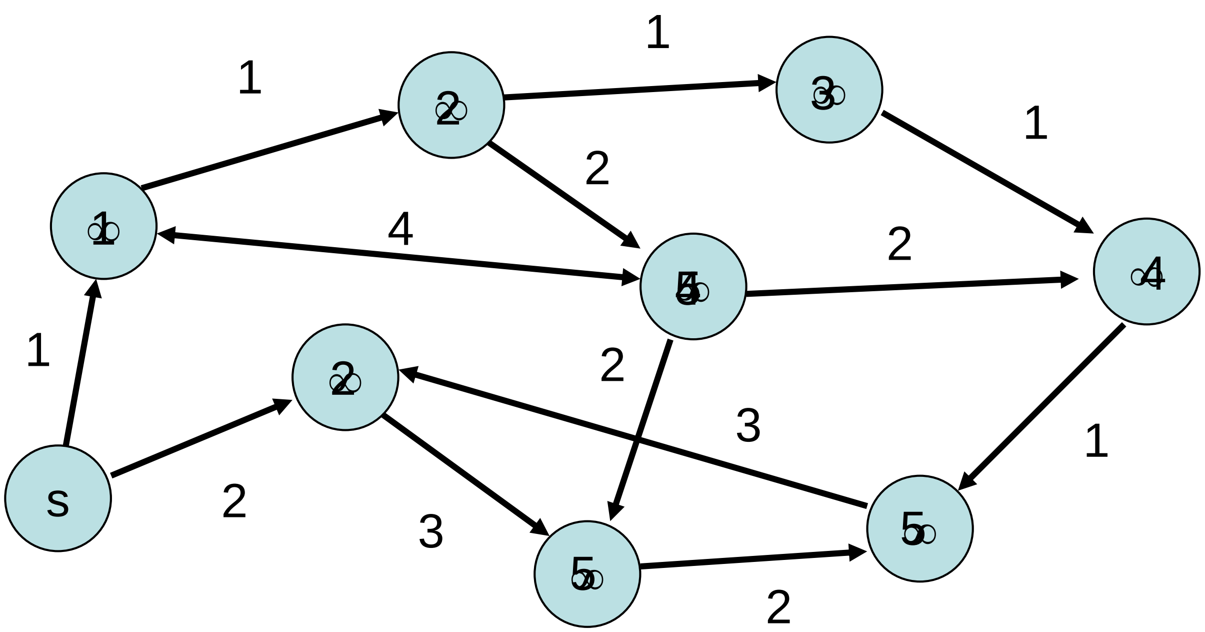
visit nodes in the order of their distance from s

Dijkstra's Algorithm

- Initially, set $d(s):=0$ and $d(v):=\infty$ for all other nodes. Use a priority queue q in which the priorities represent the current distances $d(v)$ from s. Add s to q.
- Repeat until q is empty:
  - Remove node v with lowest $d(v)$ from q (via deleteMin).
  - For all $(v,w)\in E$,
    - set $d(w) := \min\{d(w), d(v)+c(v,w)\}$. If w is already in q, this needs a decreaseKey operation. Else, if w was never in q, insert w into q.

Dijkstra's Algorithm

Example: (: current, : done)



Dijkstra's Algorithm

```
Procedure Dijkstra(s: NodeId)
  d=<∞,…, ∞>: NodeArray of ℝ∪{-∞,∞}
  parent=<⊥,…,⊥>: NodeArray of NodeId
  d[s]:=0; parent[s]:=s
  q=<s>: NodePQ
  while q ≠<> do
    u:=q.deleteMin()  // u: node with min distance
    foreach e=(u,v)∈E do
      if d[v] > d[u]+c(e) then // update d[v]
        if d[v]=∞ then q.insert(v) // v in q?
        parent[v]:=u
        // d[v] set to d[u]+c(e)
        q.decreaseKey(v, d[v]-(d[u]+c(e)))
```

Dijkstra's Algorithm

- Assume input graph has n nodes, m edges
- $T_{Op}(n)$: runtime of operation Op on data structure with n elements

each vertex added/removed precisely once

each time we traverse an edge

Runtime:

$$T_{Dijkstra} = O(n(T_{DeleteMin}(n)+T_{Insert}(n)) + m \cdot T_{decreaseKey}(n))$$

Binary heap: all operations have runtime $O(\log n)$, so

$T_{Dijkstra} = O((m+n)\log n)$

Fibonacci heap: amortized runtimes

- $T_{DeleteMin}(n)=T_{Insert}(n)=O(\log n)$
- $T_{decreaseKey}(n)=O(1)$
- Therefore, $T_{Dijkstra} = O(n \log n + m)$

Dijkstra's Algorithm

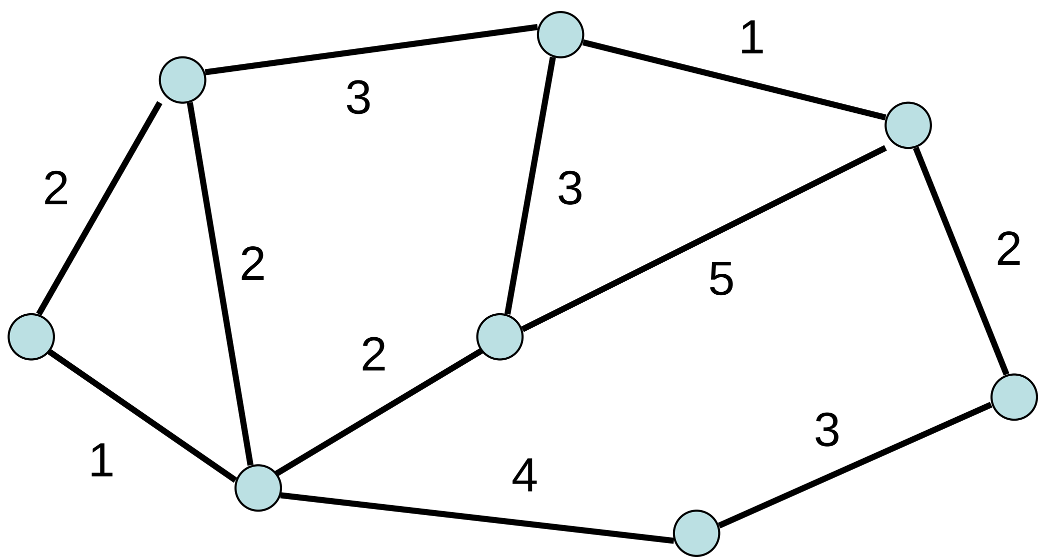
Remark: Dijkstra´s Algorithm does not need a general priority queue but only a monotonic priority queue (i.e., labels are distances, which are monotonically decreasing!)

If all edge costs are integer values in [0,C], use a Radix heap. Its amortized runtimes are

- $T_{DeleteMin}(n)=T_{decreaseKey}(n)=O(1)$
- $T_{Insert}(n)=O(\log C)$
- Thus in this case, $T_{Dijkstra} = O(n \log C + m)$

Minimal Spanning Tree

Problem: Which edges do I need to take in order to connect all nodes at the lowest possible cost?



Minimal Spanning Tree

Input:

- Undirected graph $G=(V,E)$
- Edge costs $c:E\to\mathbb{R}_+$

Output:

- Subset $T\subseteq E$ so that the graph (V,T) is connected and $c(T)=\sum_{e\in T} c(e)$ is minimal

- T always forms a tree (if c is positive). (Why?)
- Tree over all nodes in V with minimum cost: minimal spanning tree (MST)

Prim´s Algorithm

```
Procedure Prim(s: NodeId)
  d=<∞,…, ∞>: NodeArray of ℝ∪{-∞,∞}
  parent=<⊥,…,⊥>: NodeArray of NodeId
  d[s]:=0; parent[s]:=s
  q=<s>: NodePQ
  while q ≠<> do
    u:=q.deleteMin()  // u: node with min distance
    foreach e=(u,v)∈E do
      if d[v] > c(e) then // update d[v]
        if d[v]=∞ then q.insert(v) // v in q?
        parent[v]:=u
        // d[v] set to c(e)
        q.decreaseKey(v, d[v]-c(e))
        store e along with v
```

Prim´s Algorithm

- Assume input graph has n nodes, m edges
- $T_{Op}(n)$: runtime of operation Op on data structure with n elements

Runtime:

$$T_{Prim} = O(n(T_{DeleteMin}(n)+T_{Insert}(n)) + m\cdot T_{decreaseKey}(n))$$

Binary heap: all operations have runtime $O(\log n)$, so $T_{Prim} = O((m+n)\log n)$

Fibonacci heap: amortized runtimes

- $T_{DeleteMin}(n)=T_{Insert}(n)=O(\log n)$
- $T_{decreaseKey}(n)=O(1)$
- Therefore, $T_{Prim} = O(n \log n + m)$

Prim´s Algorithm

Can we use Radix heap? (does a monotone priority queue suffice?)

Next Chapter

Topic: Search structures