# Fundamental Algorithms

# Chapter 2: Advanced Heaps

Christian Scheideler

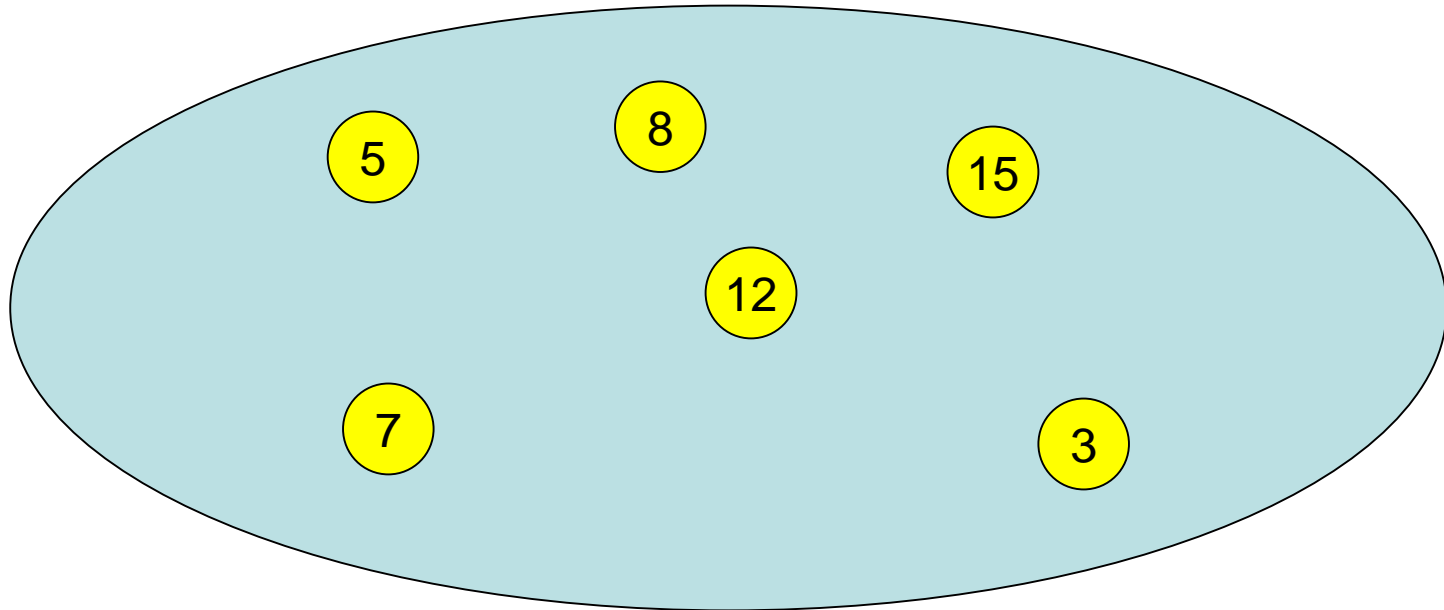WS 2017

# Contents

A heap implements a priority queue.

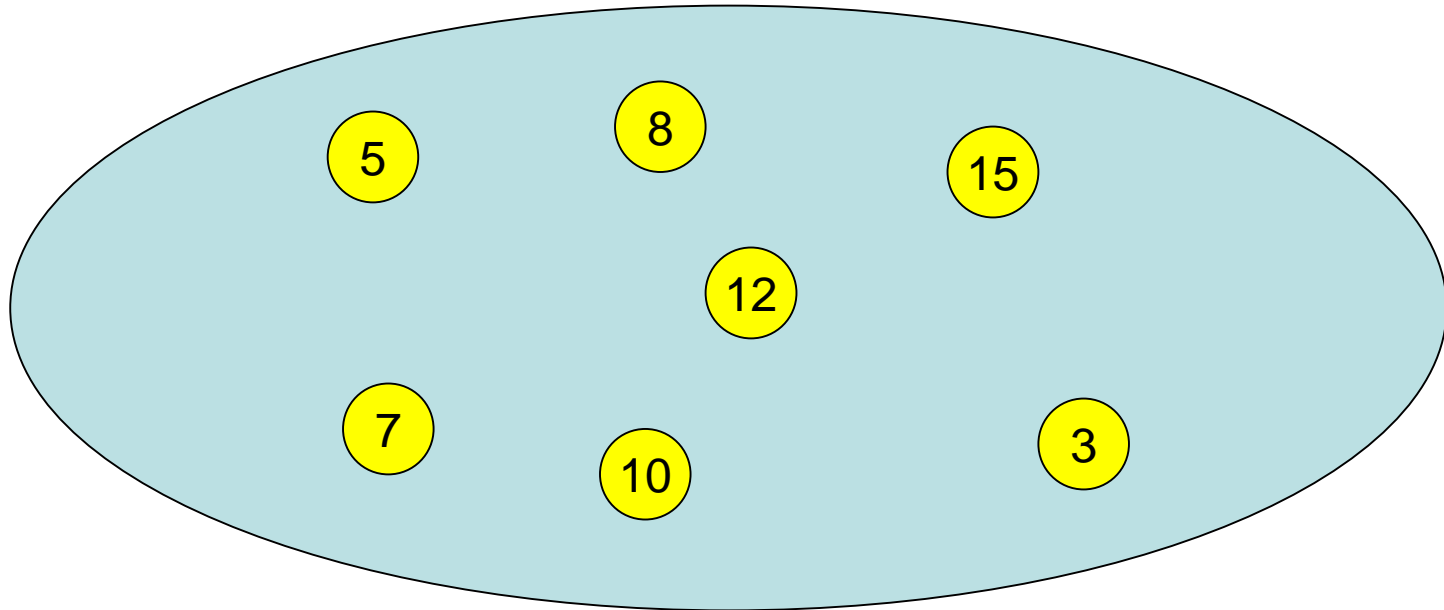We will consider the following heaps:

- Binomial heap

- Fibonacci heap
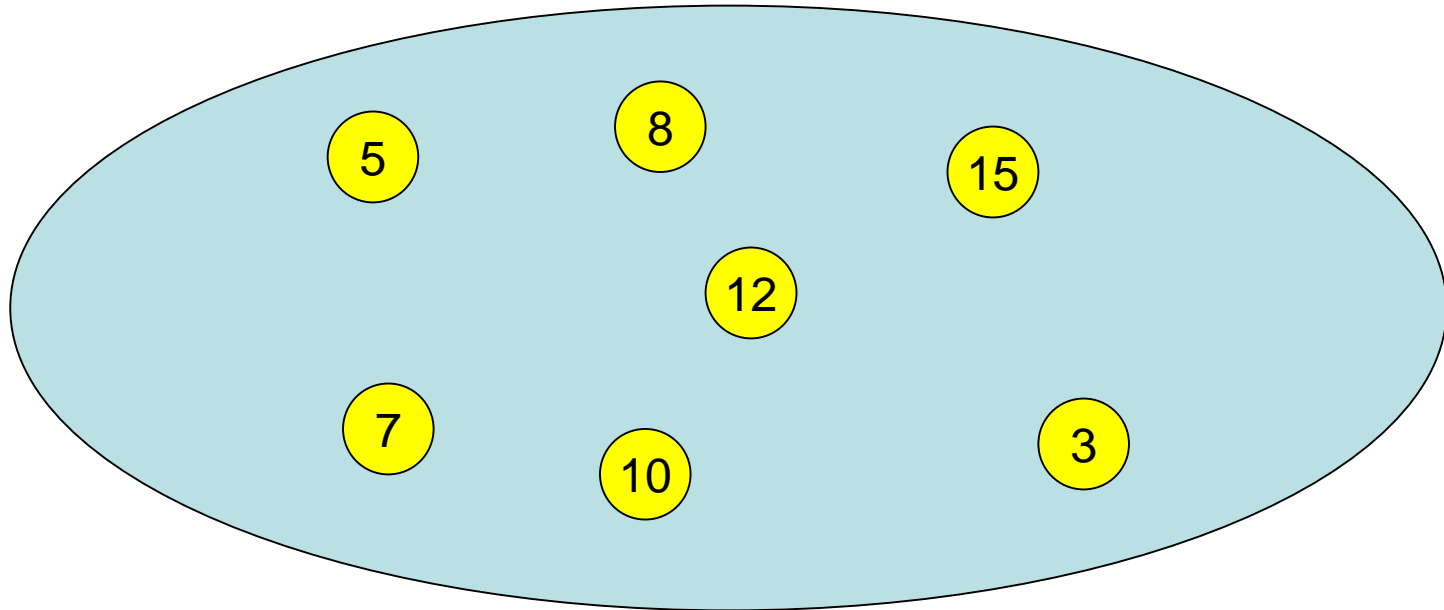
- Radix heap

# Priority Queue
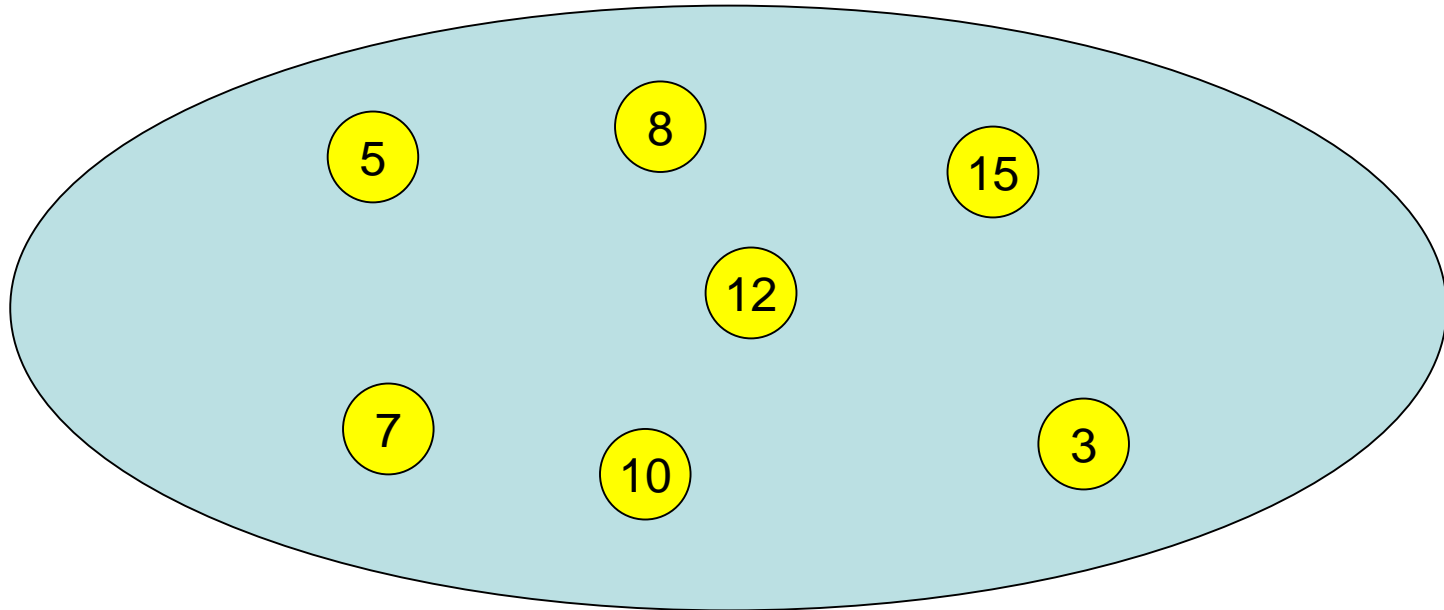
# Priority Queue

insert(10)

# Priority Queue

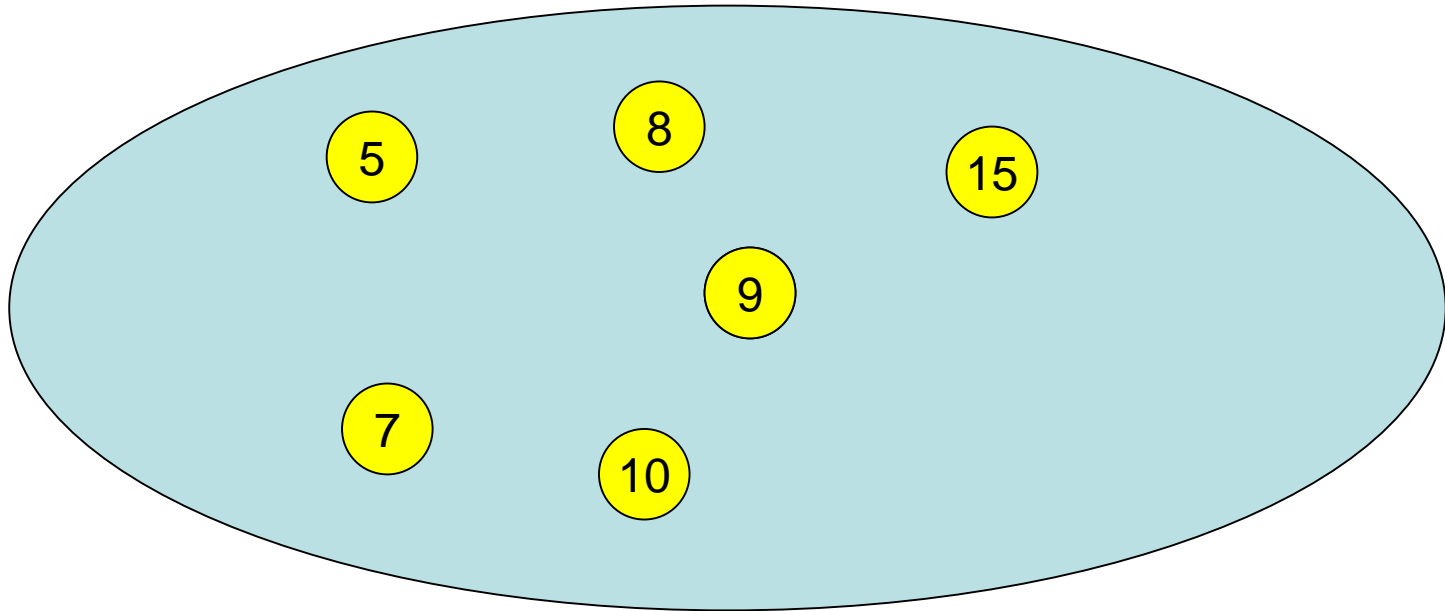min() outputs 3 (minimal element)

# Priority Queue

deleteMin()

# Priority Queue

decreaseKey(12,3)

# Priority Queue
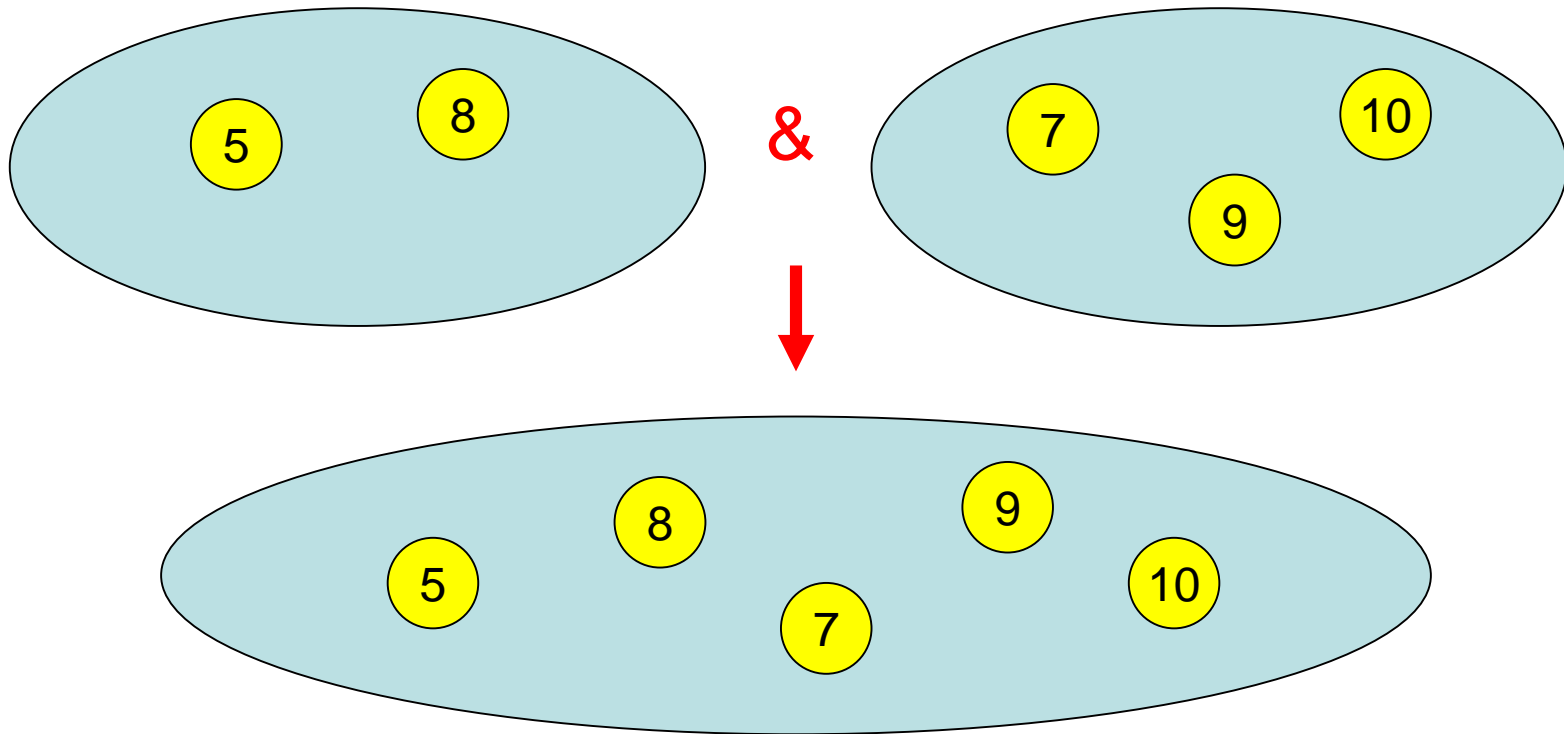
delete(8)
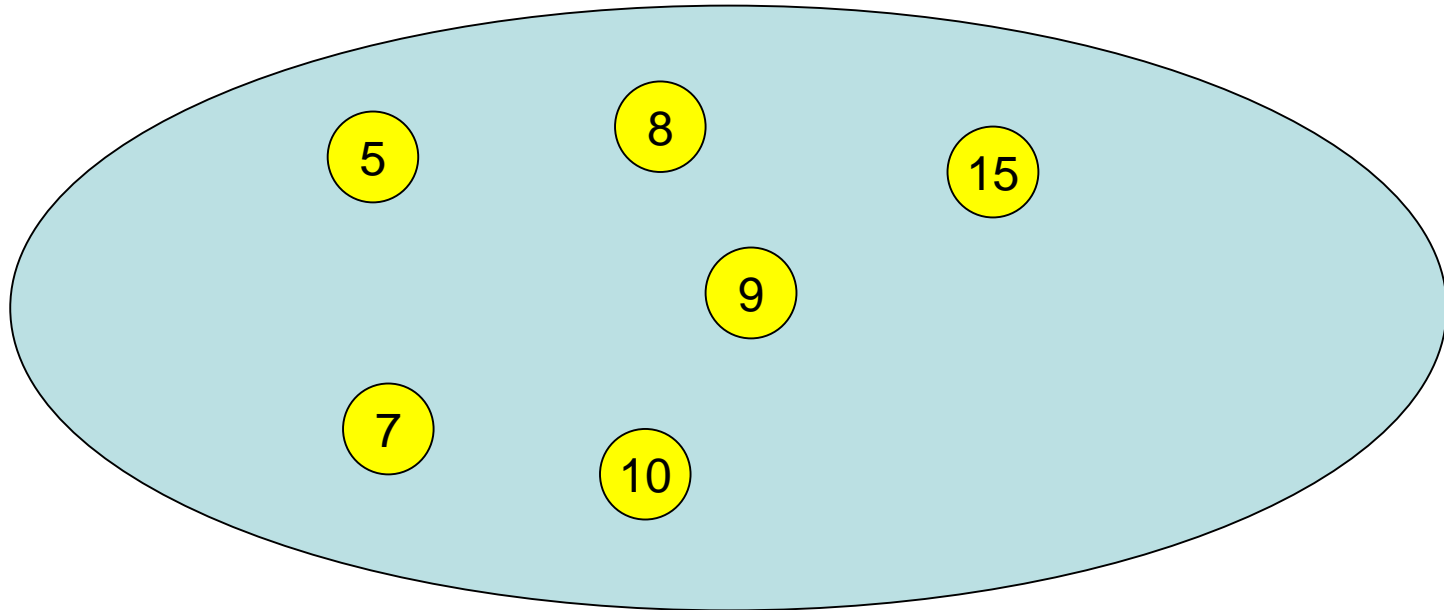
# Priority Queue

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, $\Delta$): key(e):=key(e)-$\Delta$
- M.merge(M´): M:=M$\cup$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 heap to systematically generate all humble numbers, starting with heap 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 heap)

# 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)
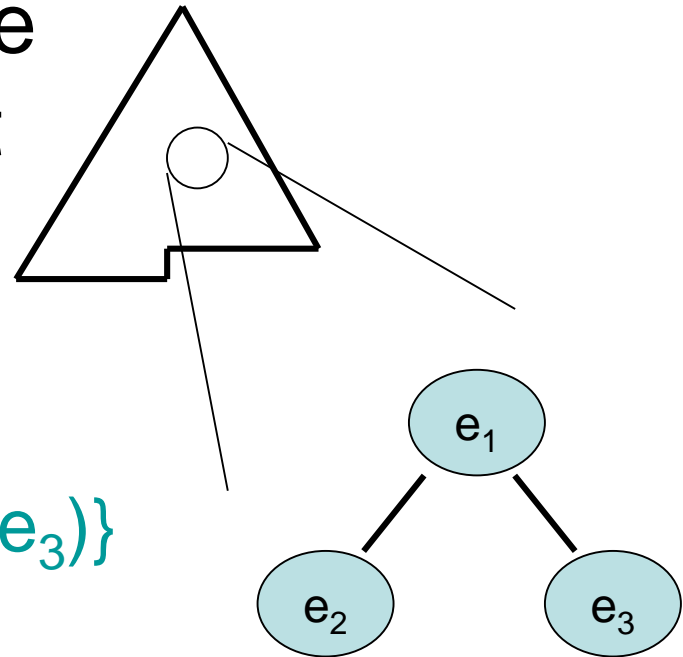  - min, deleteMin: $O(1)$

Better structure needed than list or array!

# Binary Heap

**Idee:** use binary tree instead of list

**Preserve two invariants:**
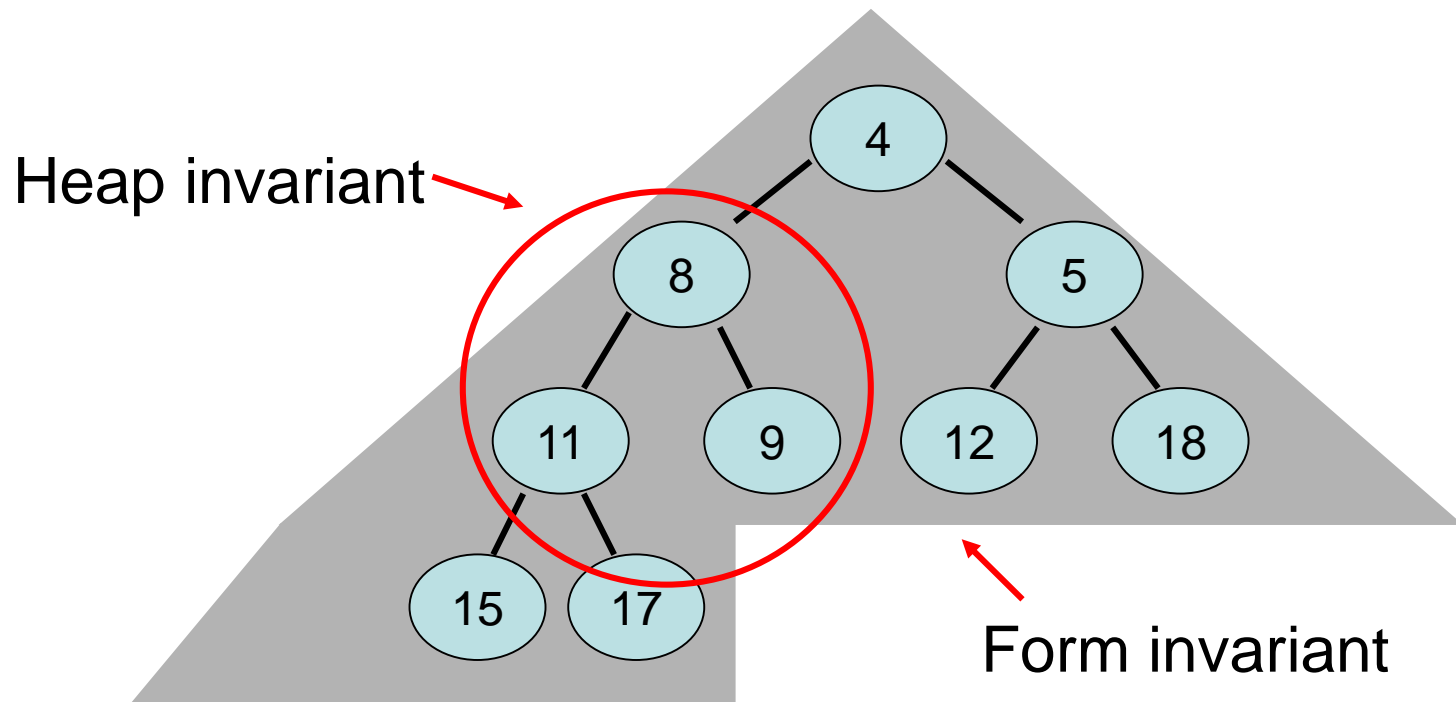
- **Form invariant:** complete binary tree up to lowest level

- **Heap invariant:**

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

# Binary Heap

Example:

Heap invariant

Form invariant

# Binary Heap

Representation of binary tree via array:

# Binary Heap

Representation of binary tree via array:

| $e_1$ | $e_2$ | $e_3$ | $e_4$ | $e_5$ | $e_6$ | $e_7$ | $e_8$ | $e_9$ |
|---|---|---|---|---|---|---|---|---|
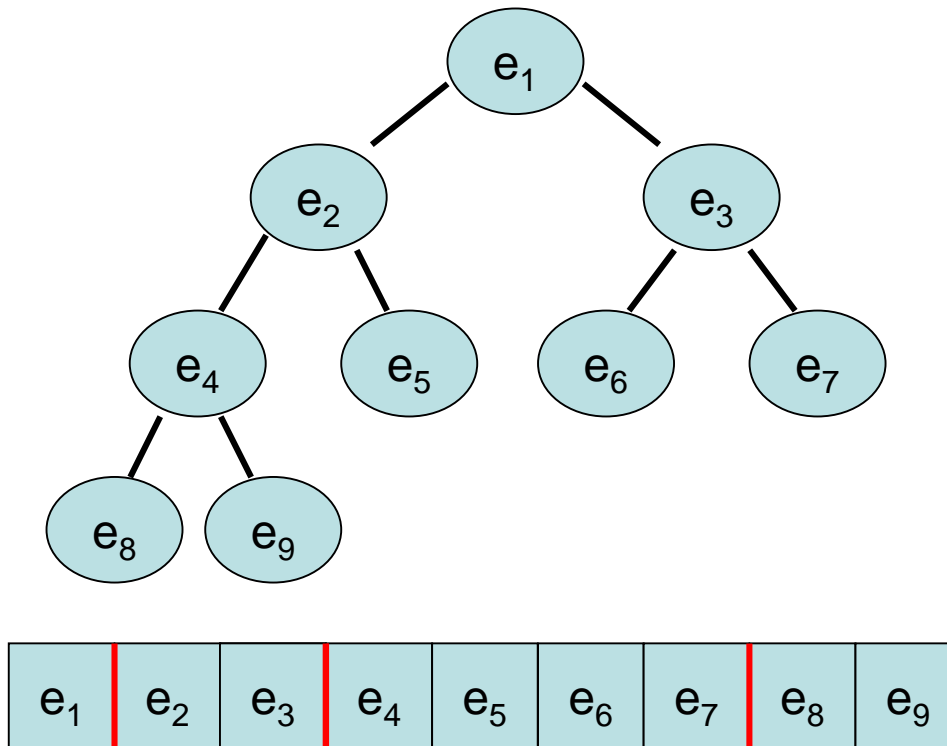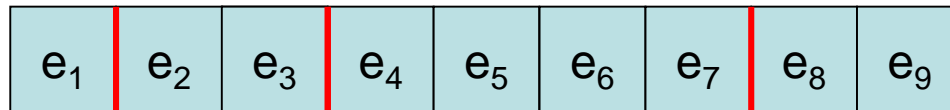
- H: Array [1..N] of Element   (N ≥ #elements n)
- Children of e in H[i]: in H[2i], H[2i+1]
- Form invariant: H[1],…,H[n] occupied
- Heap invariant: for all $i \in \{2,…,n\}$,
  $$key(H[i]) \geq key(H[\lfloor i/2 \rfloor])$$

# Binary Heap

Representation of binary tree via array:
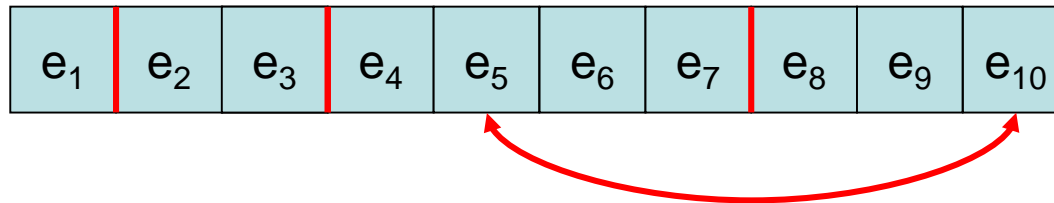


insert(e):

- 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

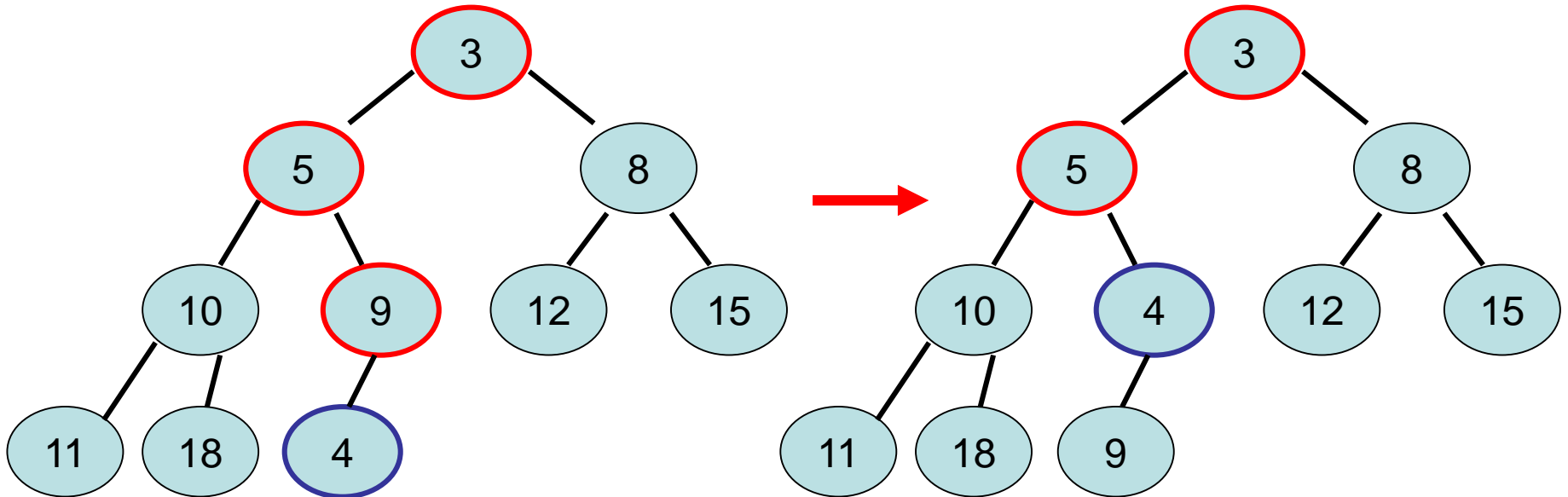Chapter 2

# Binary Heap

| $e_1$ | $e_2$ | $e_3$ | $e_4$ | $e_5$ | $e_6$ | $e_7$ | $e_8$ | $e_9$ |
|---|---|---|---|---|---|---|---|---|

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

heapifyDown(i: Integer):
   while $2i \leq n$ 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])$\leq$key(H[m]) then return   // heap inv. holds
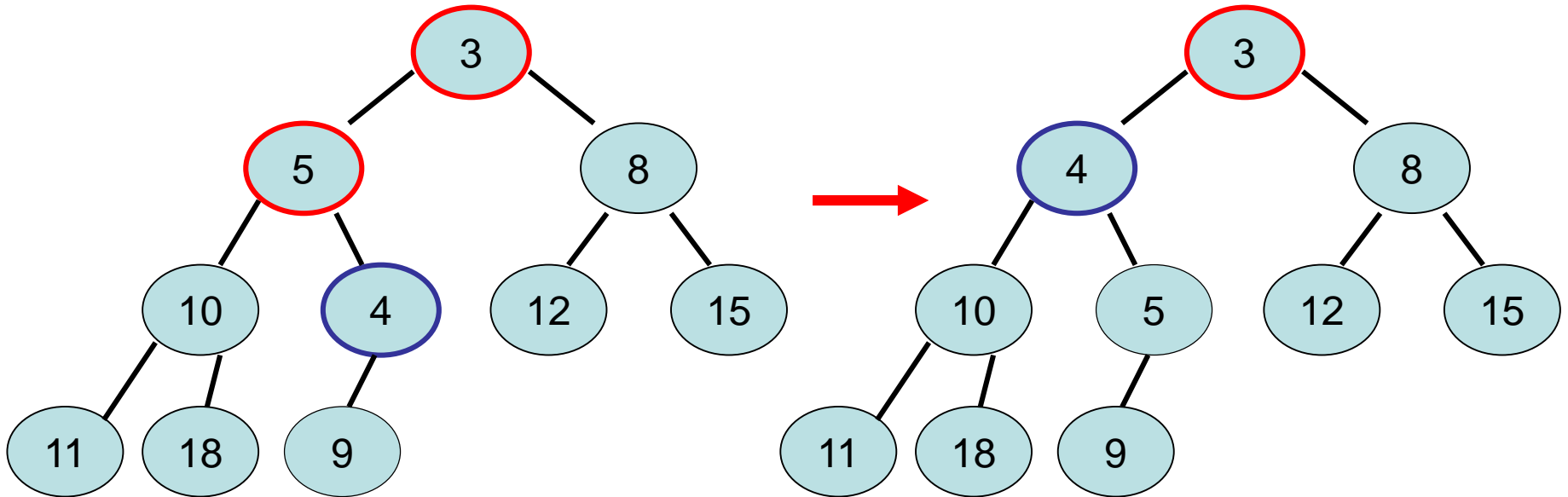     H[i] $\leftrightarrow$ H[m]; i:=m

# deleteMin Operation - Correctness
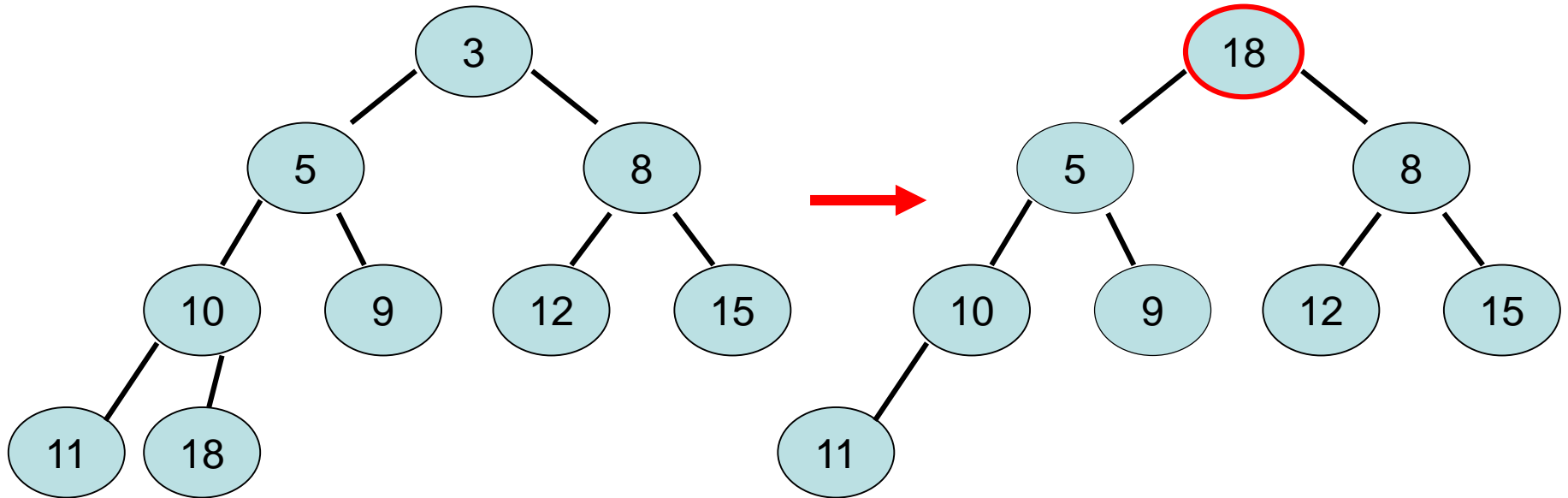


Invariant: H[k] is minimal w.r.t. subtree of H[k]

⬭ : nodes that may violate invariant

# deleteMin Operation - Correctness
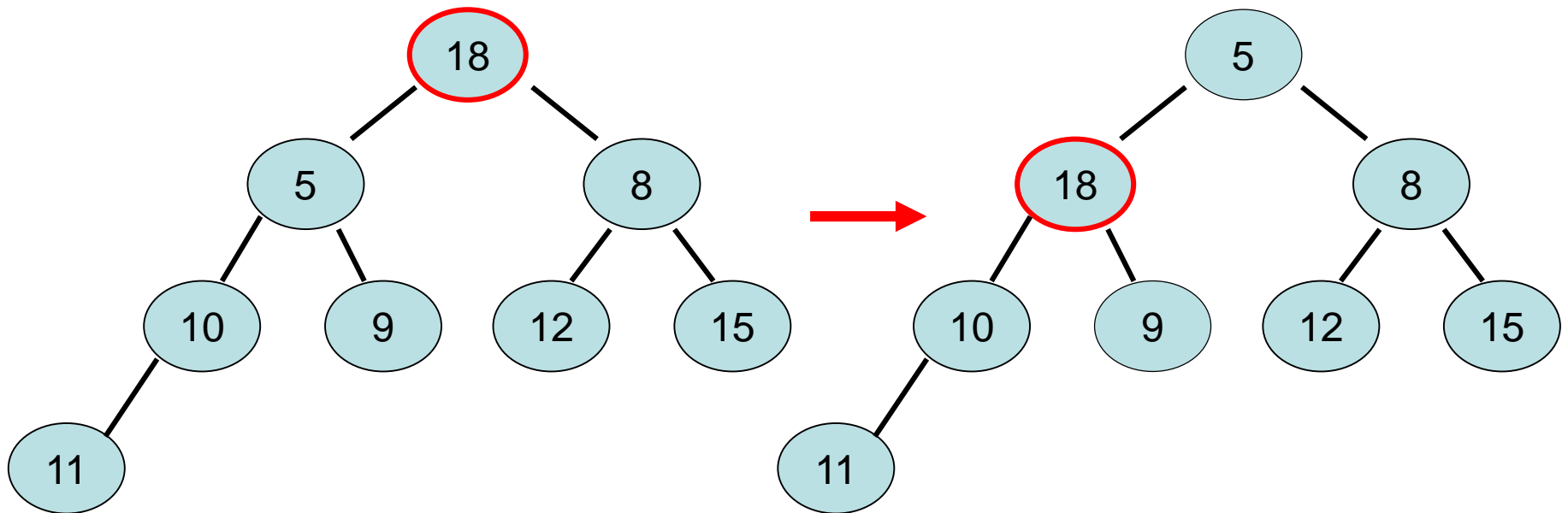


Invariant: H[k] is minimal w.r.t. subtree of H[k]
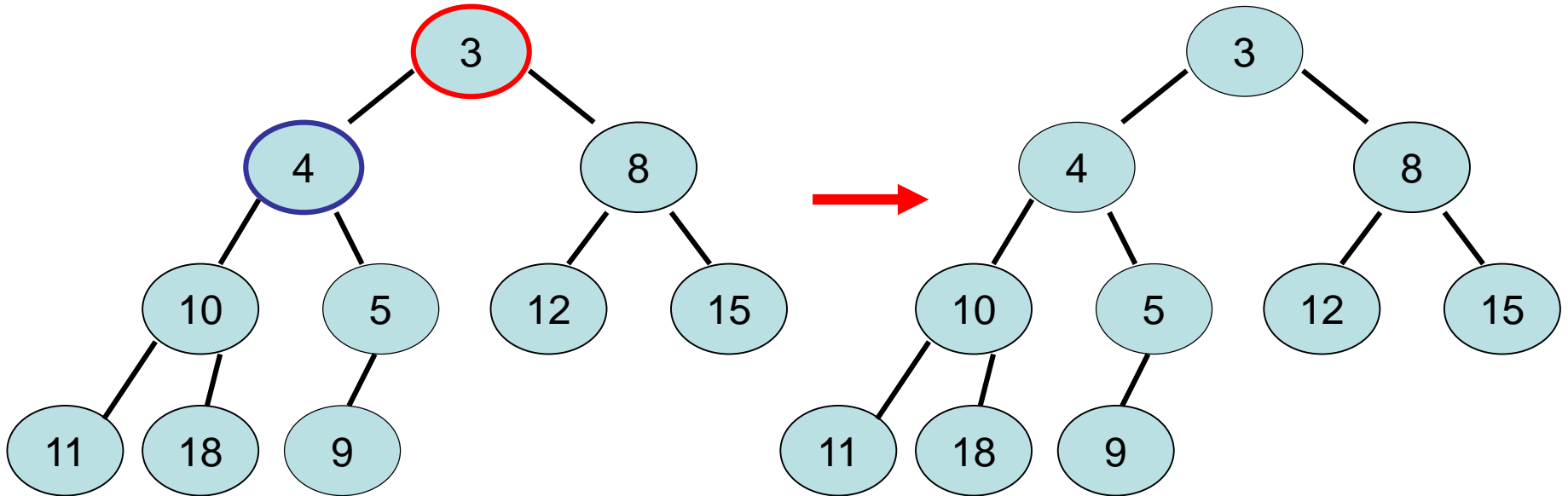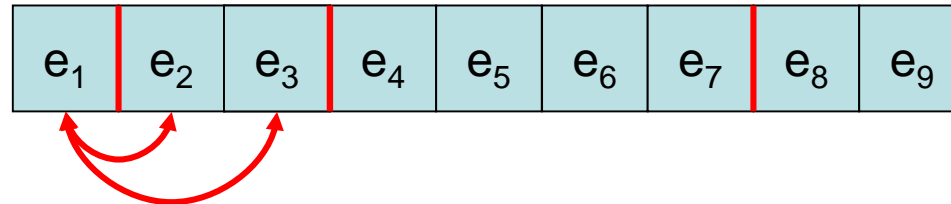
⬭ : nodes that may violate invariant

# deleteMin Operation - Correctness
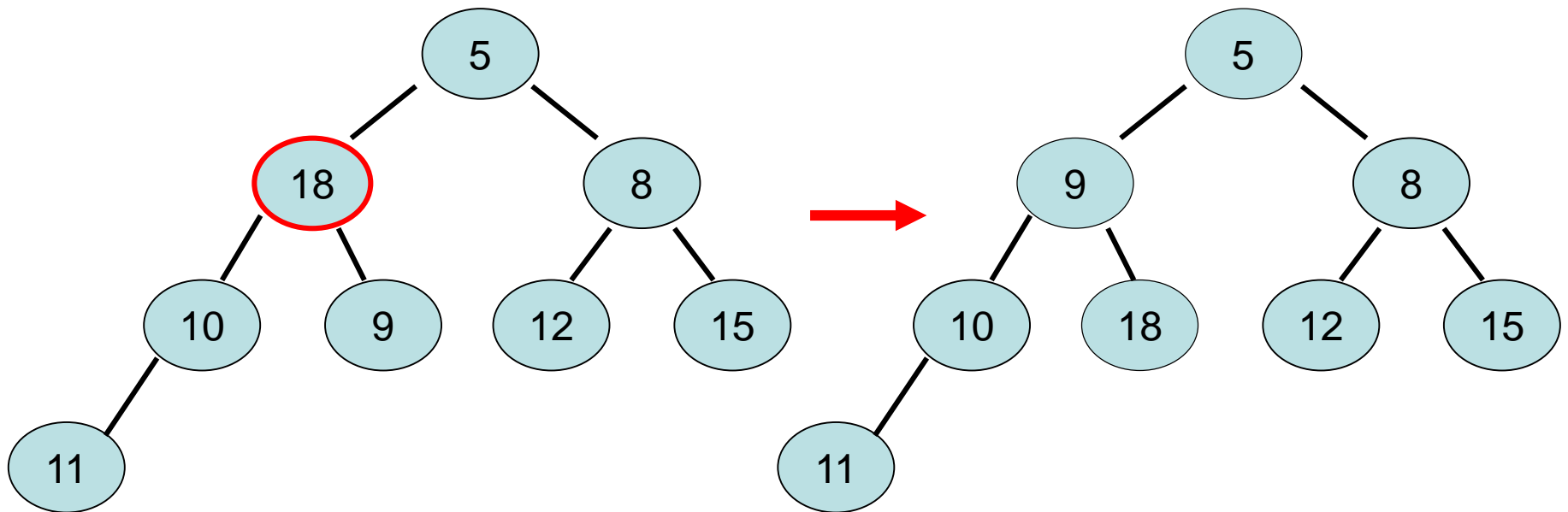


Invariant: H[k] is minimal w.r.t. subtree of H[k]

⬭ : nodes that may violate invariant

# Binary Heap

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

- Naive implementation: via n insert(e) operations. Runtime O(n log n)
- Better implementation:

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

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

- Runtime (with k=$\lceil \log n \rceil$):
  $$O\left(\sum_{1 \leq l < k} 2^l(k-l)\right) = O\left(2^k \sum_{j \geq 1} j/2^j\right) = O(n)$$

# Binary Heap

Call HeapifyDown(i) for i=⌊n/2⌋ down to 1:
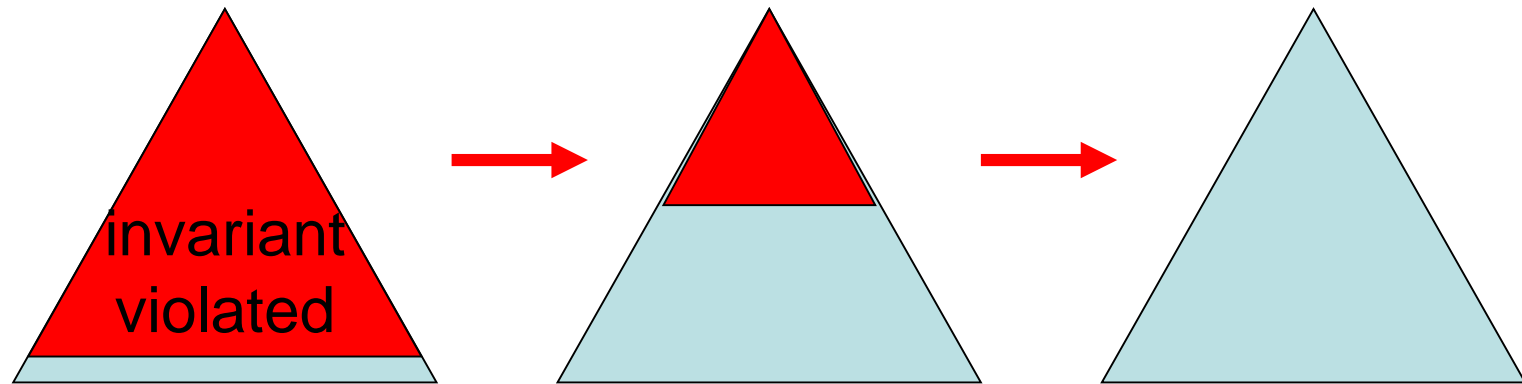


Invariant: ∀j>i: H[j] min w.r.t. subtree of H[j]

# Binary Heap

Runtime:

- build($\{e_1,\ldots,e_n\}$): 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\{e}
- M.decreaseKey(e:Element, $\Delta$): 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)!

# Binomial Heap

Binomial heap is based on binomial trees

Binomial tree has to satisfy:

- Form invariant (r: rank):

r=0        r=1        r → r+1
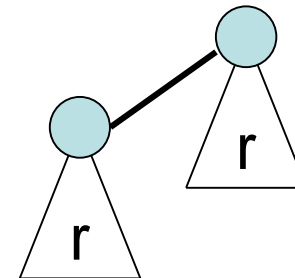


- Heap invariant (key(Parent)≤key(Children))

# Binomial Heap

Examples of correct Binomial trees:
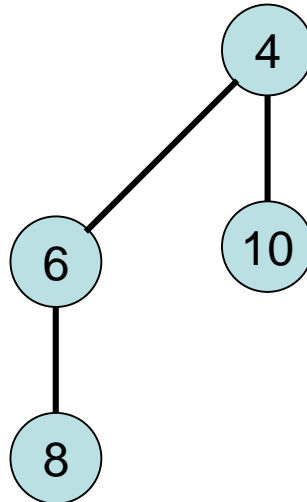
r=0          r=1               r=2                    r=3
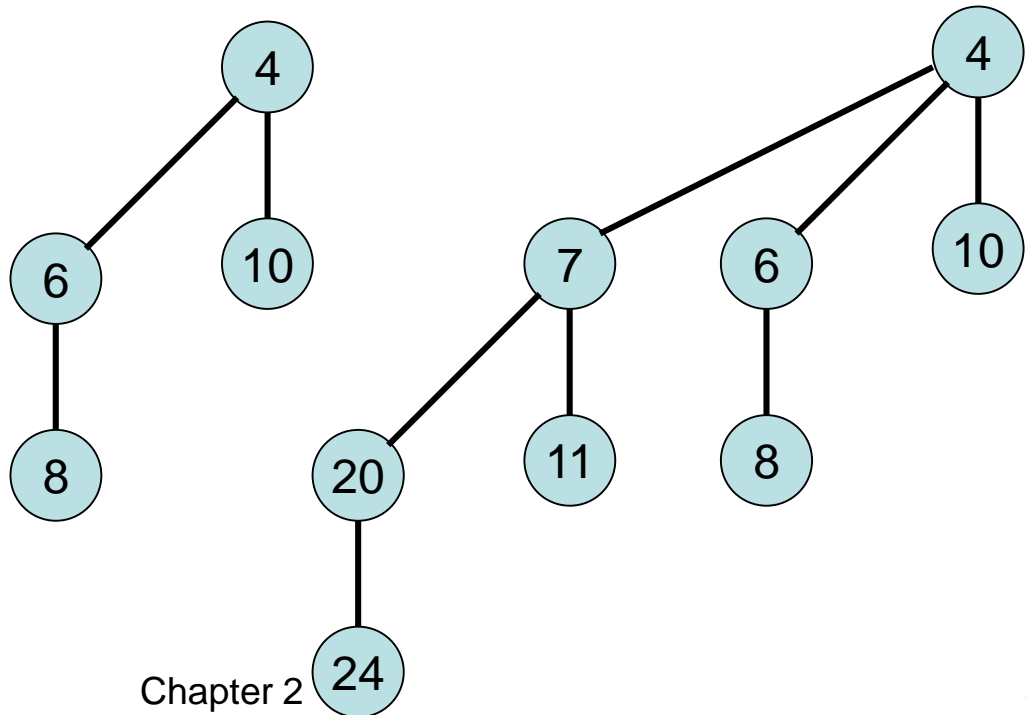
# Binomial Heap

Properties of Binomial trees:

r=0                    r=1                    r → r+1



- $2^r$ nodes
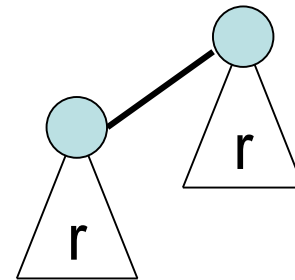
number of neighbors

- maximum degree r (at root)
- root deleted: Binomial tree decomposes into Binomial trees of rank 0 to r-1

# Binomial Heap

Example for decomposition into Binomial trees of rank 0 to r-1



ranks     2     1    0    rank 3
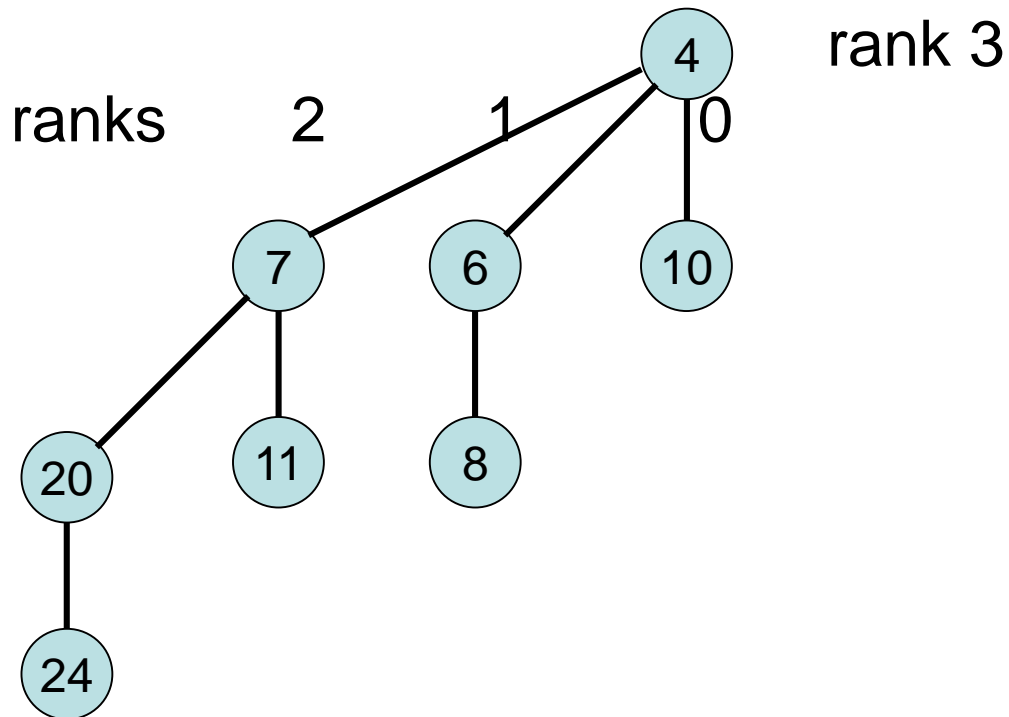
# Binomial Heap

Binomial Heap:

- linked list of Binomial trees, ordered by ranks

- for each rank at most 1 Binomial tree

- pointer to root with minimal key

numbers: ranks
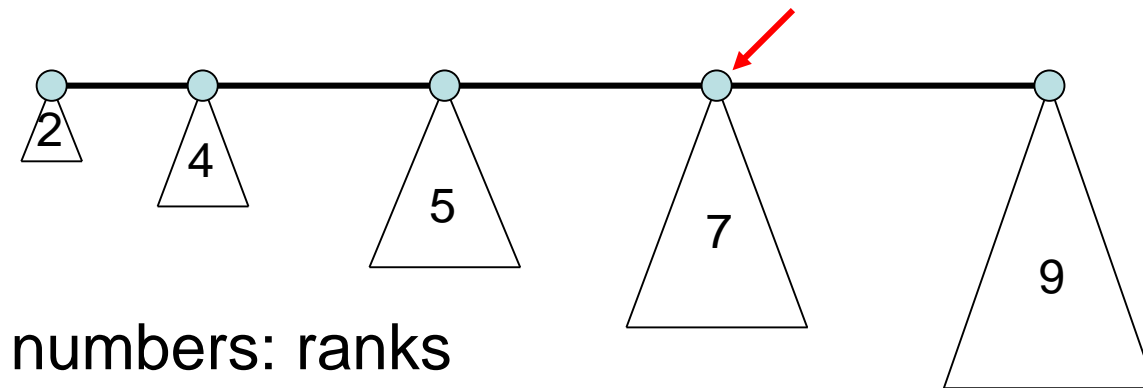
# Binomial Heap

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:



min-pointer

9 — 3 — 4

15
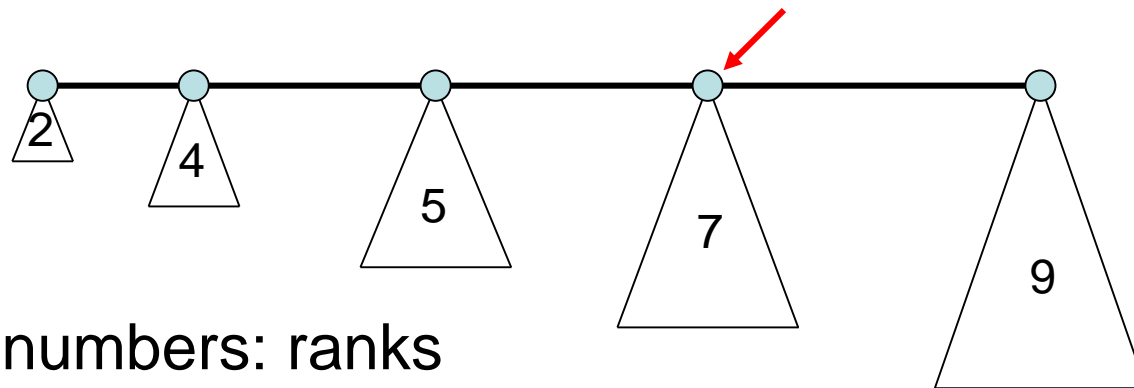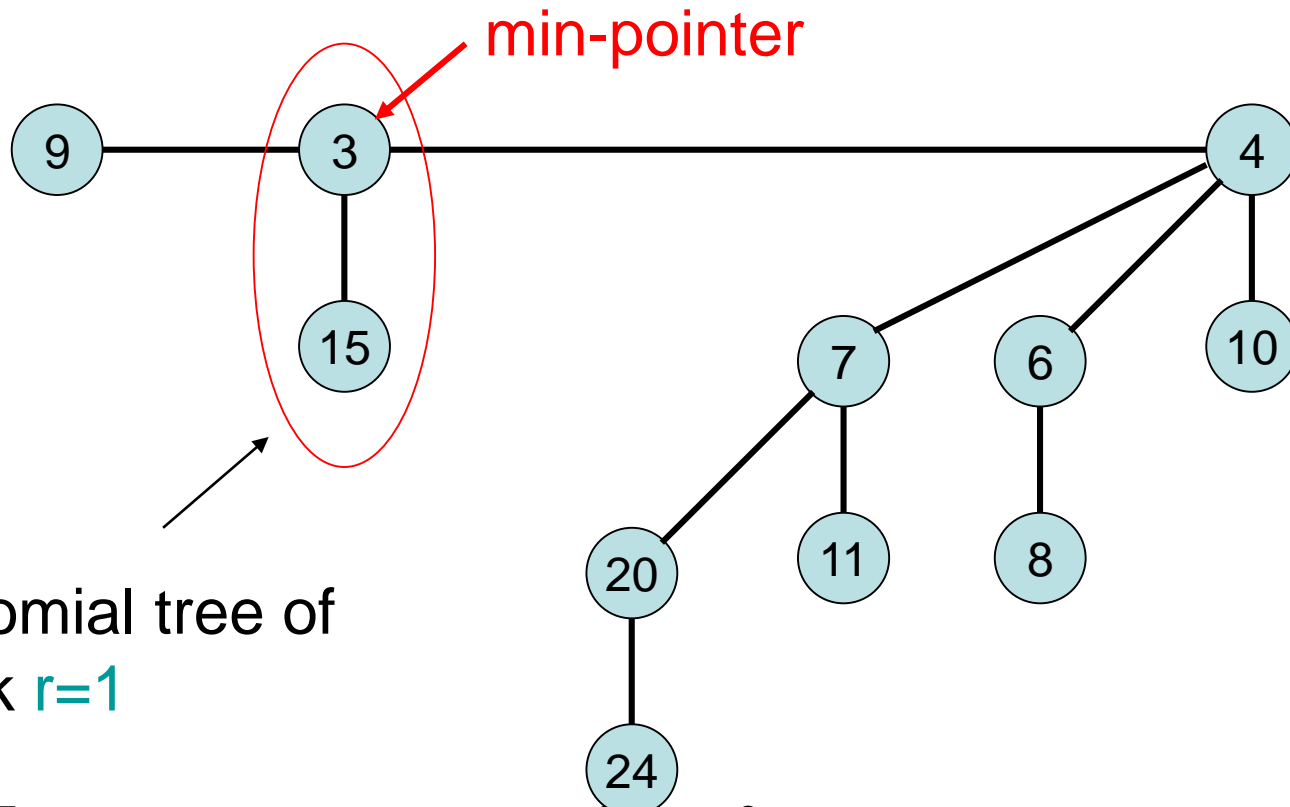
Binomial tree of rank r=1

7   6   10

20   11   8

24

Chapter 2

# Binomial Heap

Example of a correct Binomial heap:
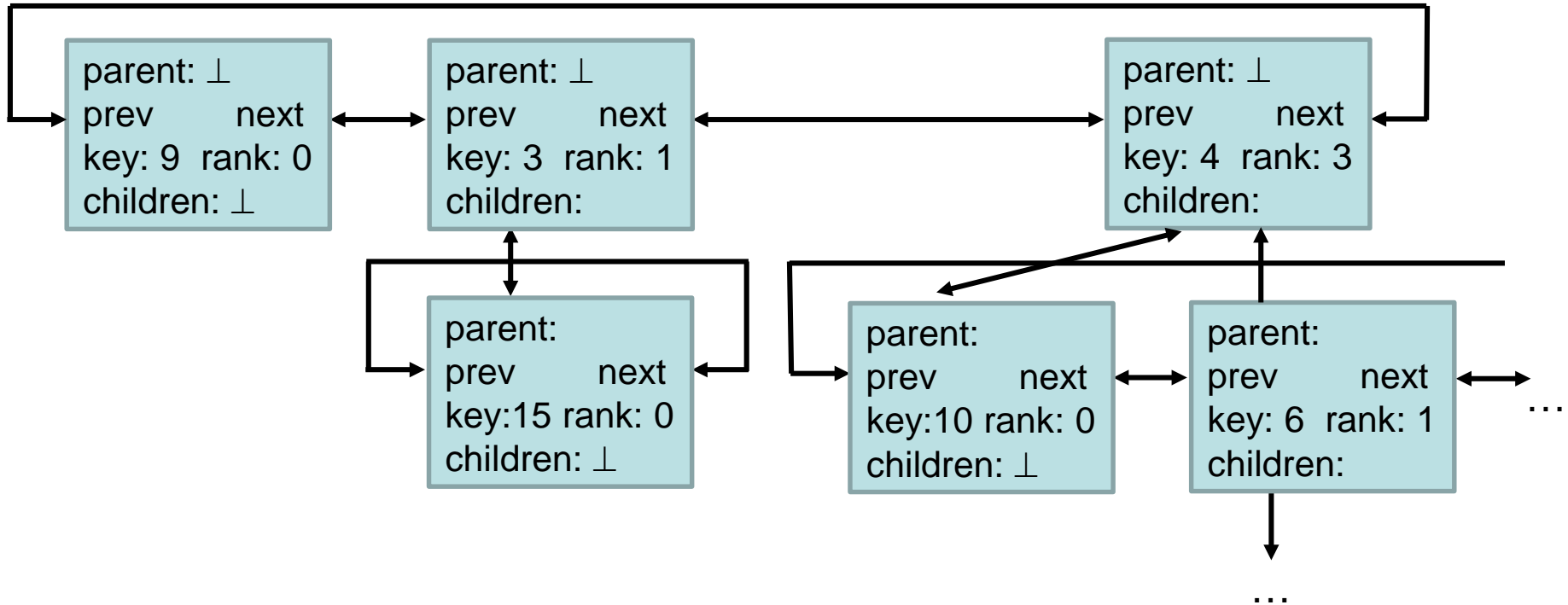
# Binomial Heap

## Merge of Binomial heaps $H_1$ and $H_2$:



ranks

$H_1$

$H_2$
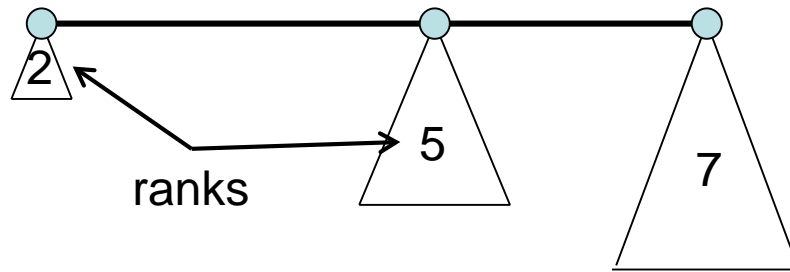
like binary addition

10100100
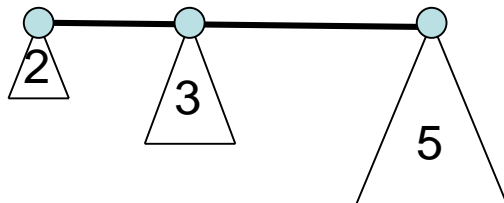
+  101100

_____
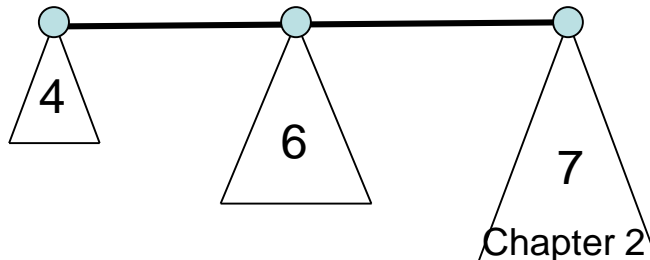
11010000

# Example of Merge Operation
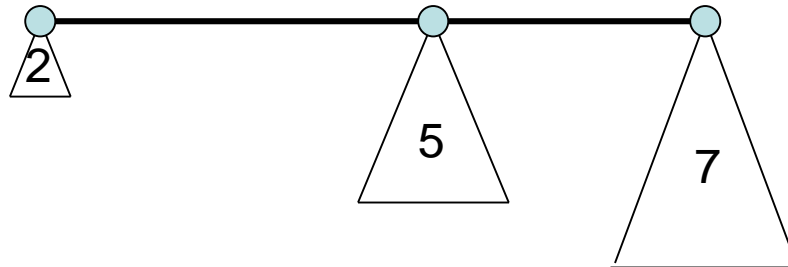
$H_1$

numbers denote
the ranks

$H_2$
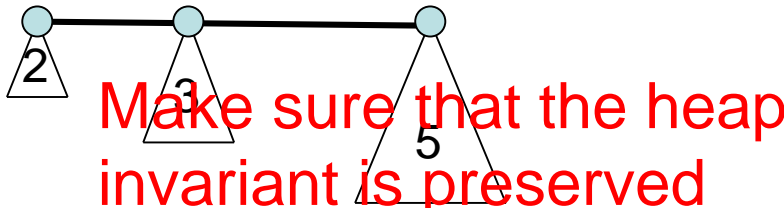
Make sure that the heap
invariant is preserved
by the merging!
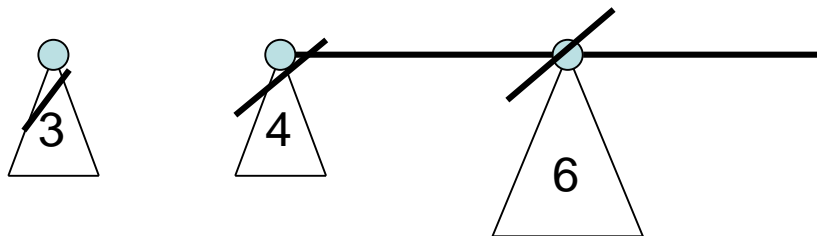
outcome

# 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)
- deleteMin: let the min-pointer point to the root of $B_i$. Deleting the root in $B_i$ results in Binomial trees $B_0,\ldots,B_{i-1}$. These have to be merged back into Binomial heap.

Thus, the insert and deleteMin operations can be reduced to the merge operation, which implies a runtime of O(log n).
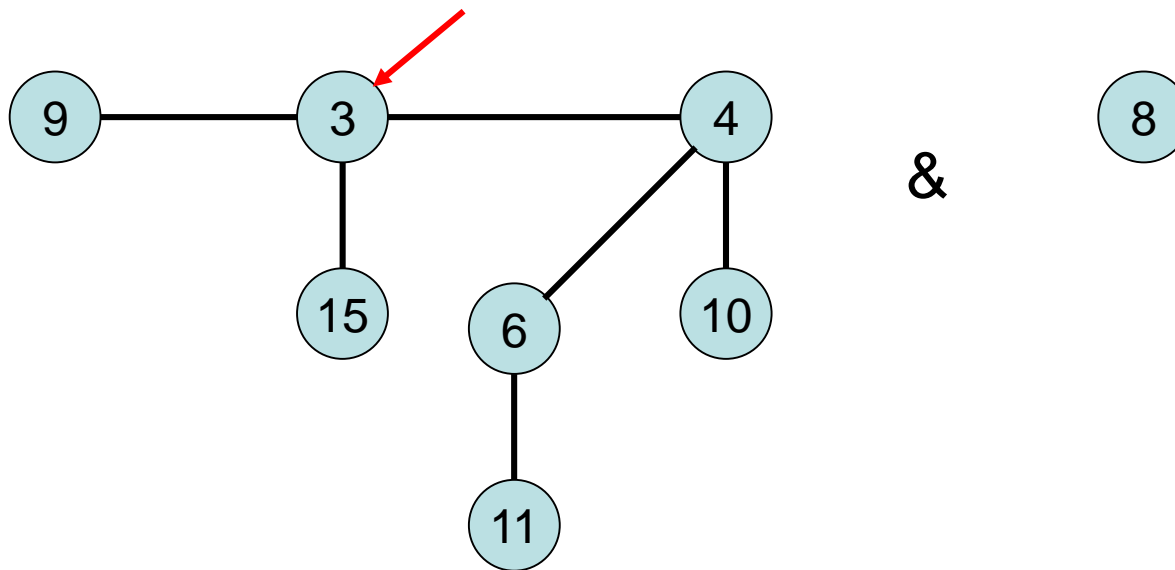
# Example of Insert Operation

Insert(8):

# Example of Insert Operation

Insert(8):



Chapter 2

# Example of Insert Operation

Insert(8):

4    3

&

6    10

8    15

11

9
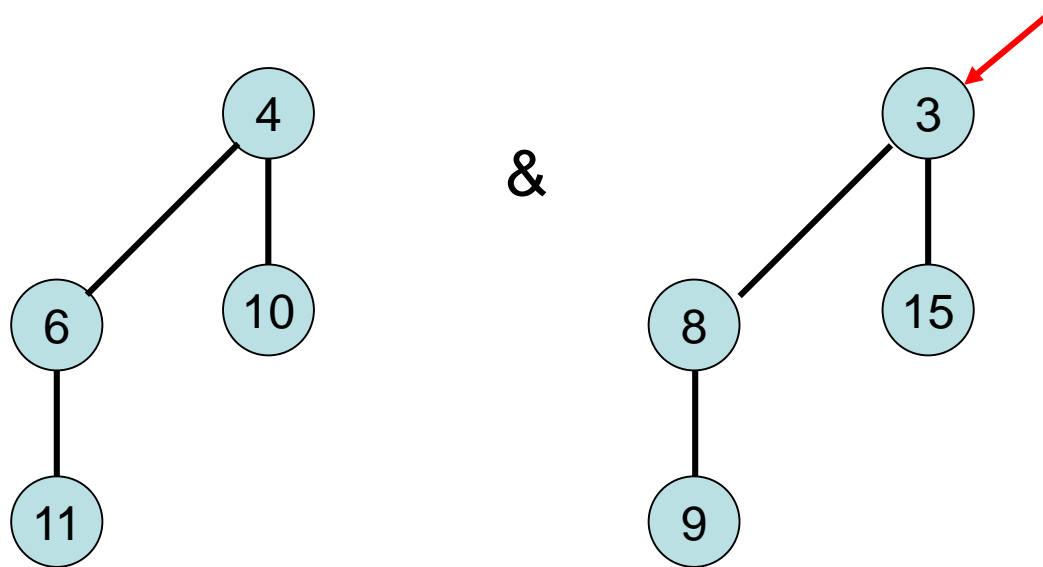
# Example of Insert Operation

Outcome of Insert(8):
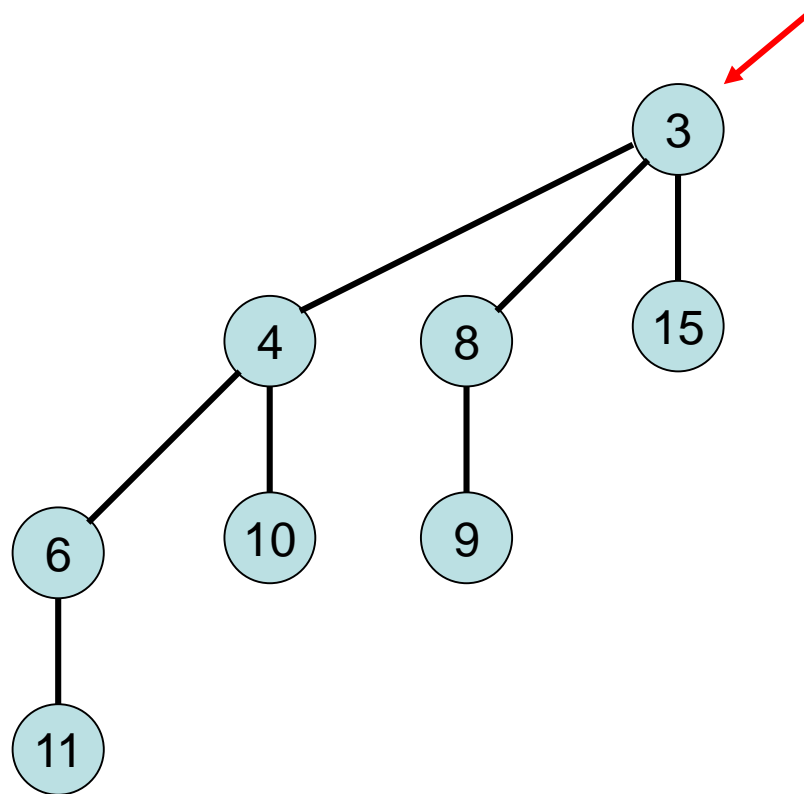
# Binomial Heap

- decreaseKey(e,$\Delta$): perform heapifyUp operation in Binomial tree starting with e, update min-pointer. Time O(log n)

- delete(e): (min-pointer does not point to e) set key(e):= -$\infty$ and perform heapifyUp operation starting with e until e is in a root; then continue like in deleteMin when removing e (but without updating the min-pointer!).
Time O(log n)

# Example of decreaseKey
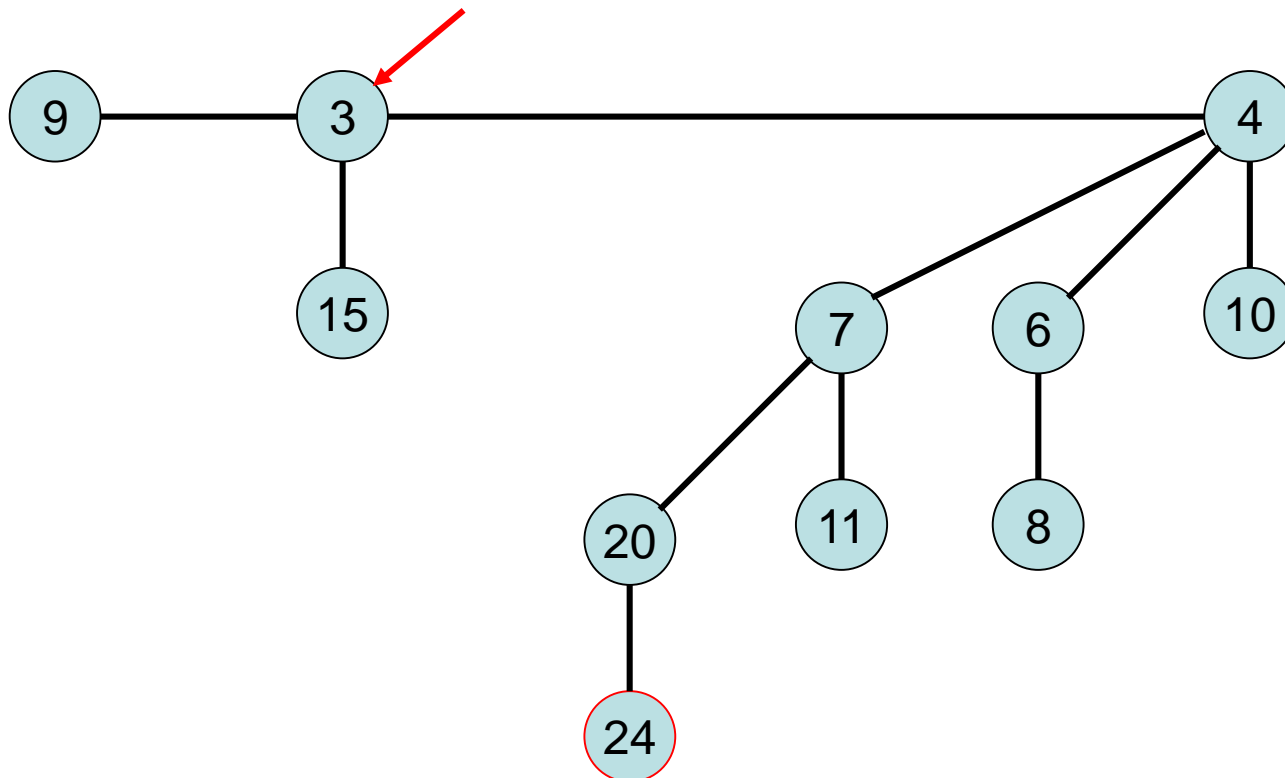
decreaseKey(24,19):

# Example of decreaseKey
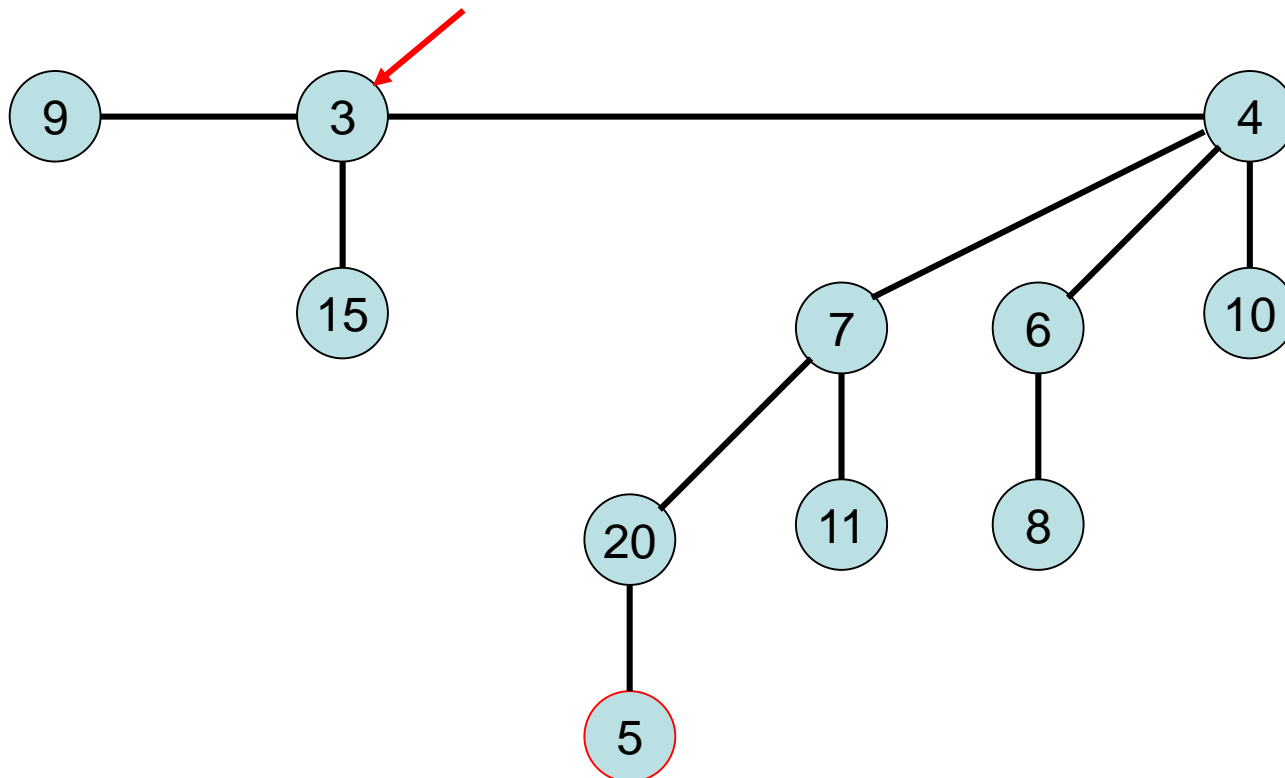
decreaseKey(24,19):

# Example of decreaseKey
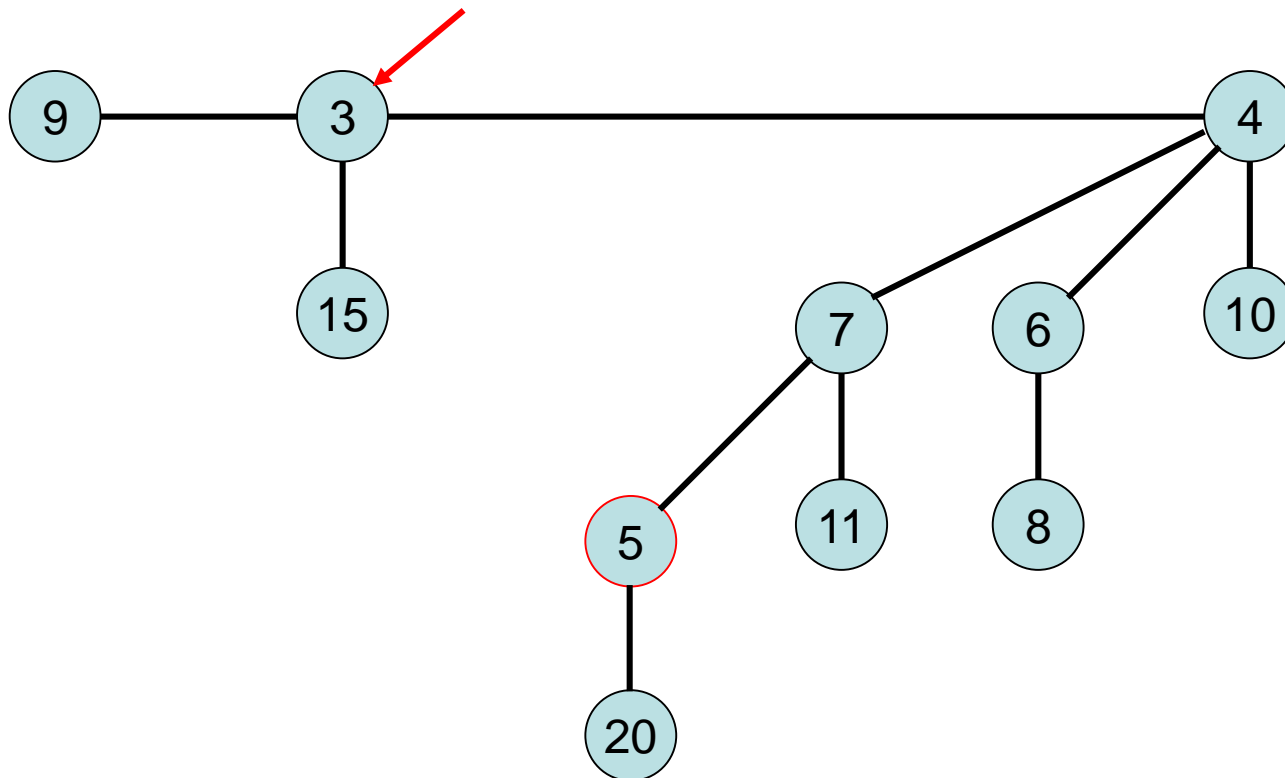
decreaseKey(24,19):

# Example of decreaseKey

Outcome of decreaseKey(24,19):
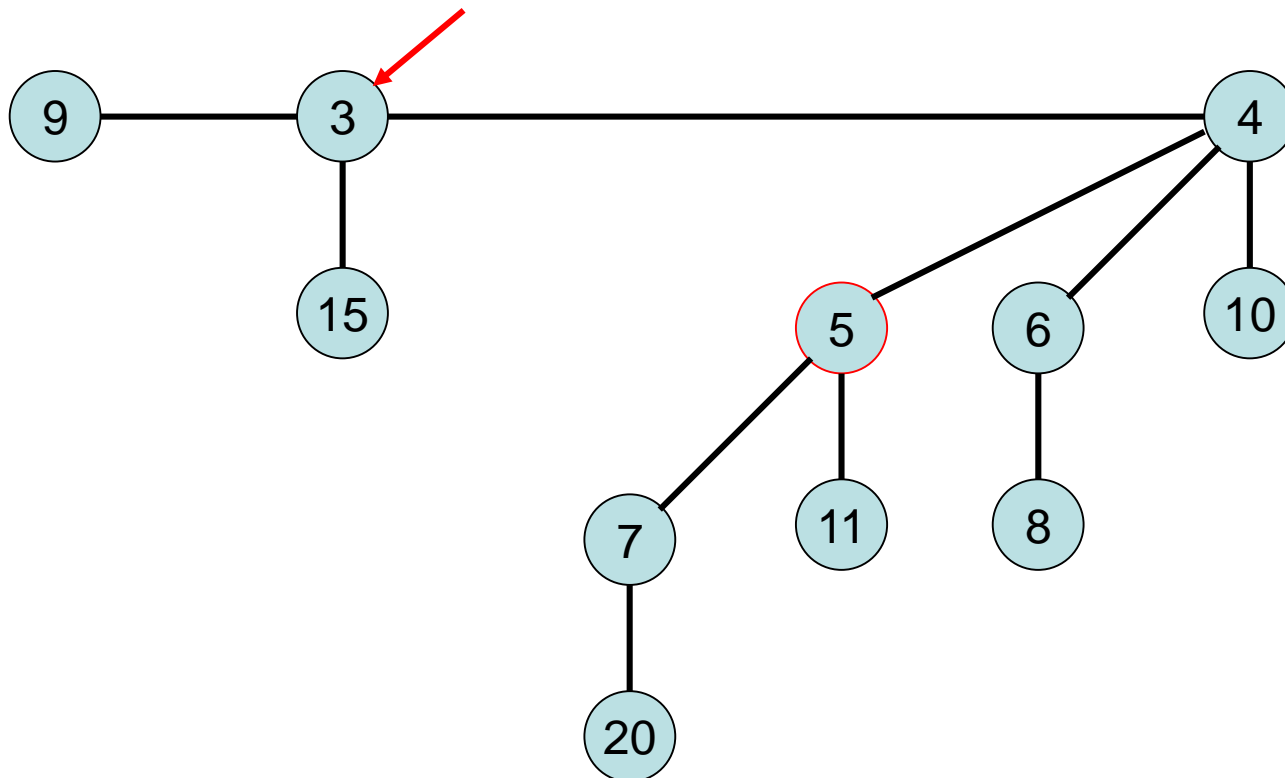
# Fibonacci Heap

- 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:



Every parent only knows first and last child of list

Every child knows its parent

List of siblings
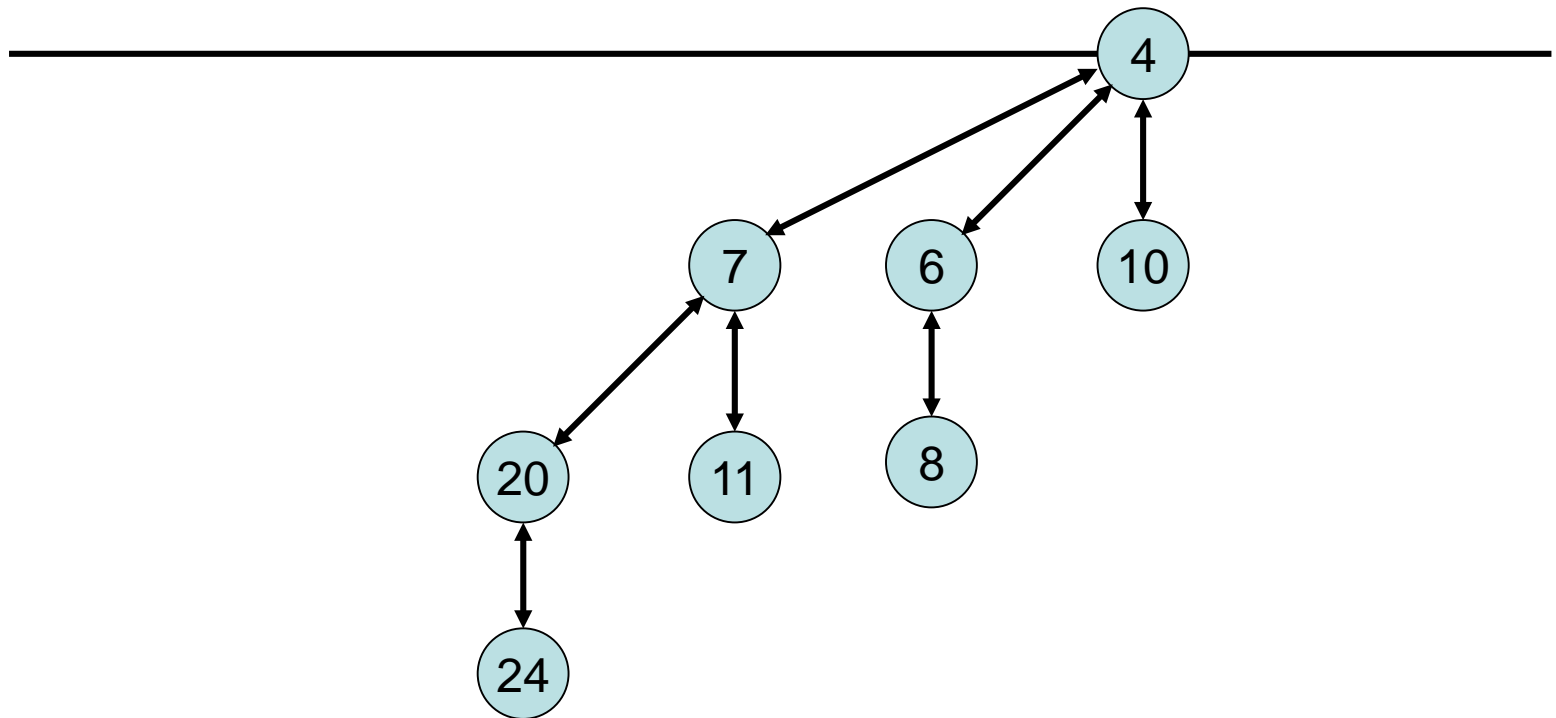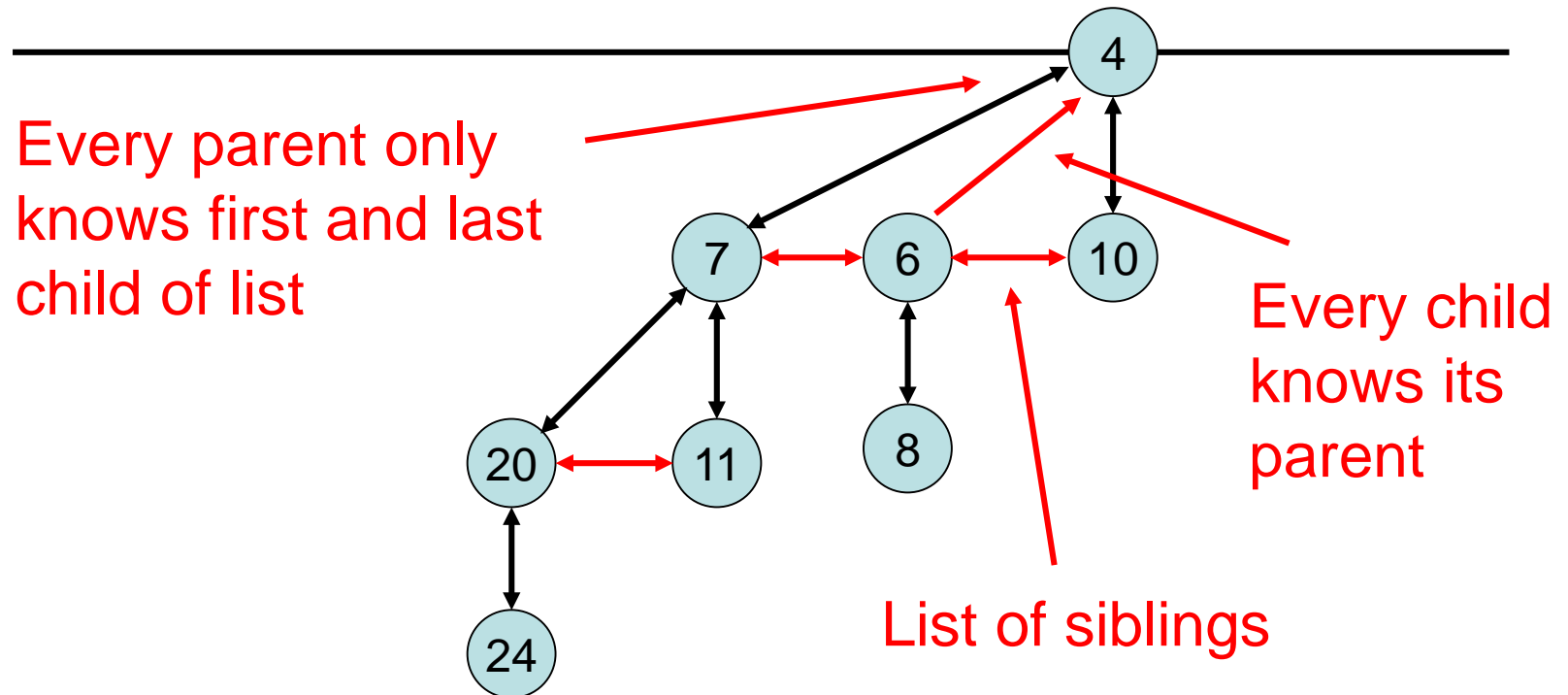
# Fibonacci Heap

Tree in a Fibonacci heap:

Data type fibTree:

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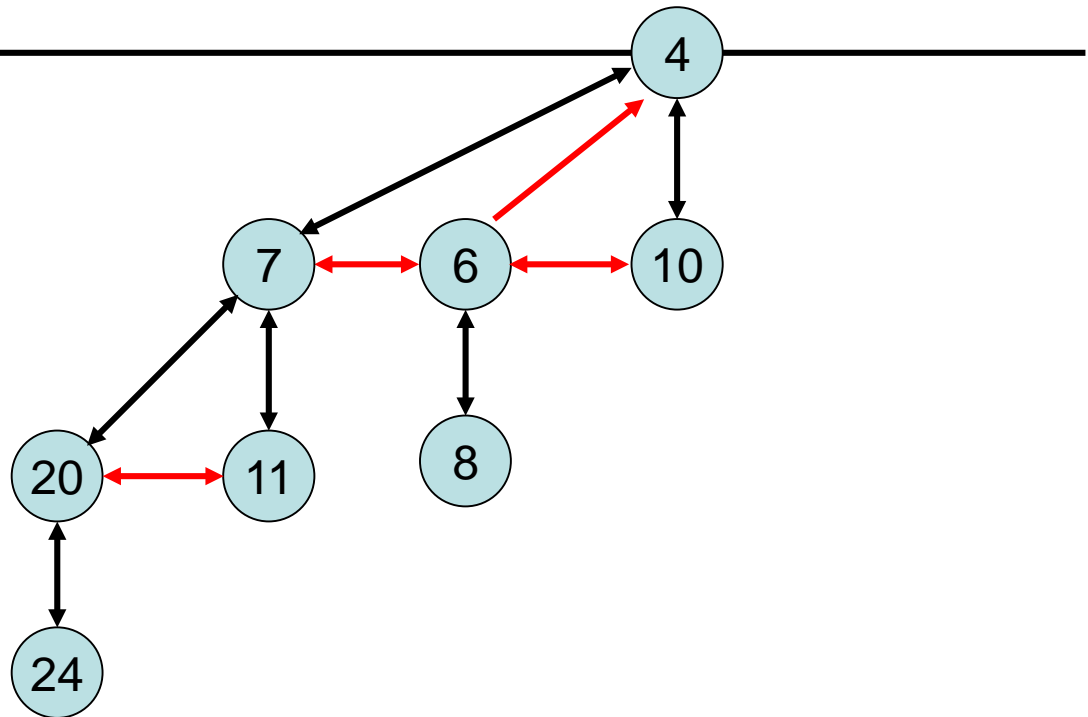
# Fibonacci Heap

Lazy merge of
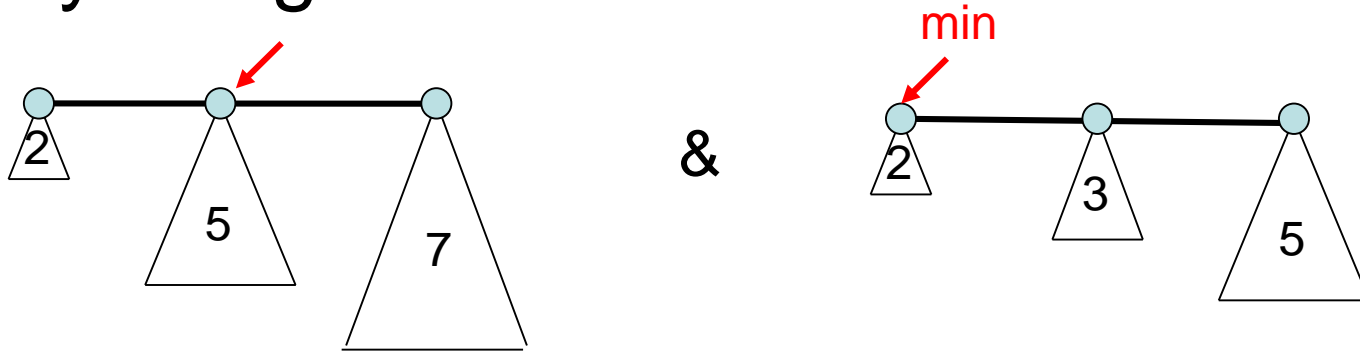


results in
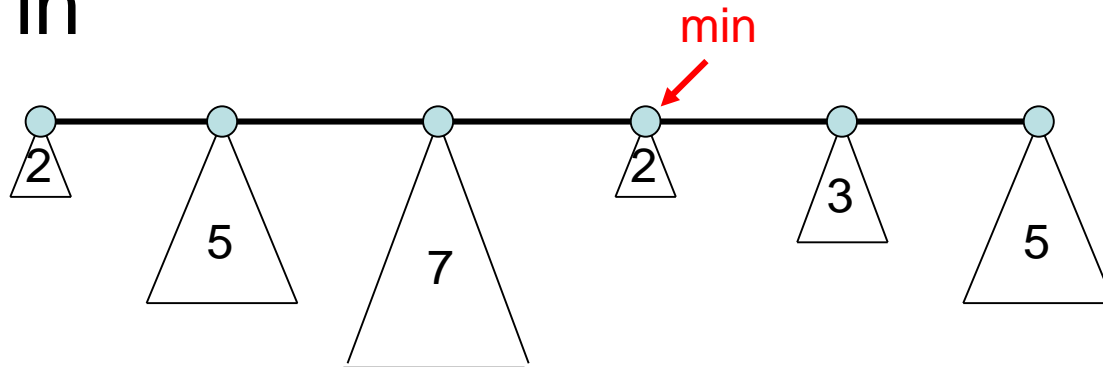
# Fibonacci Heap

Lazy delete:
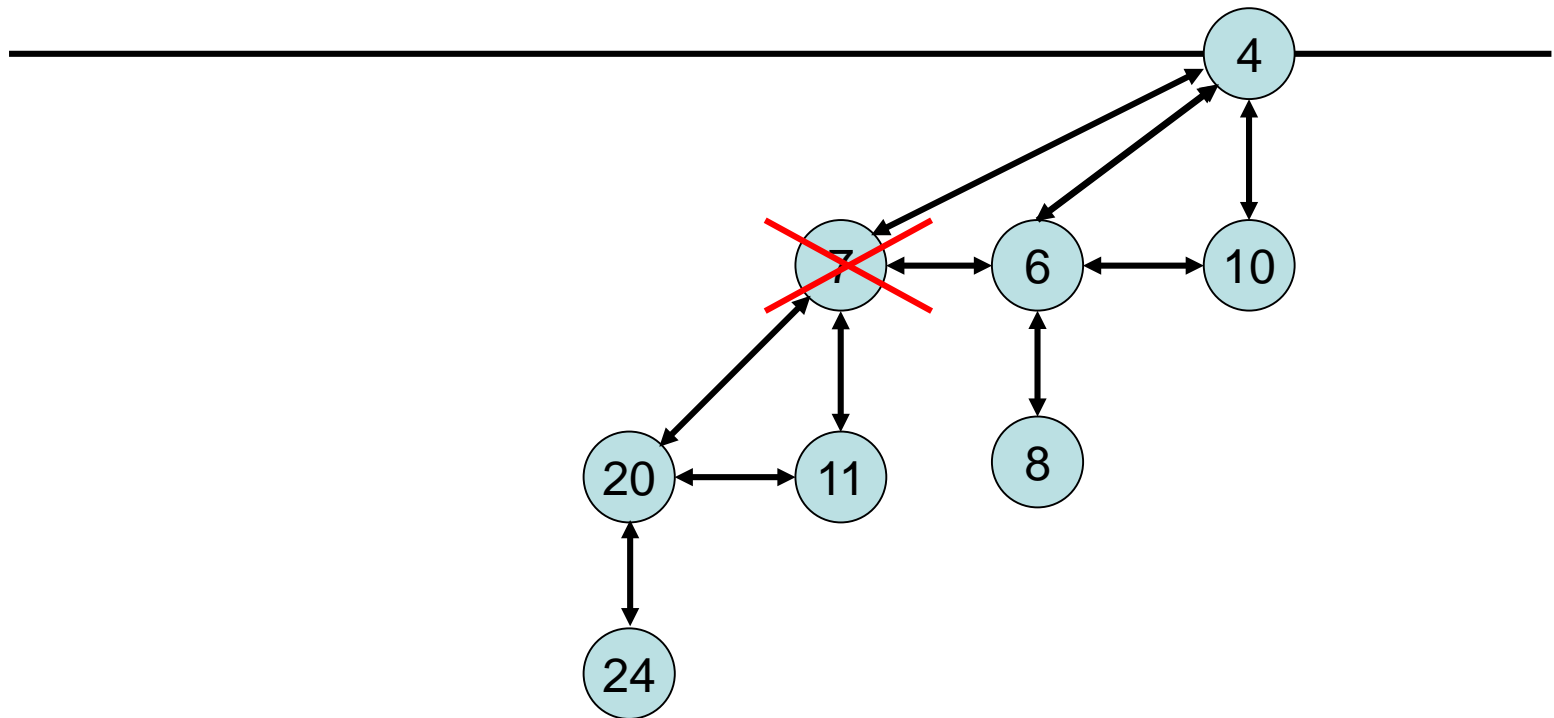
# Fibonacci Heap

Lazy delete:
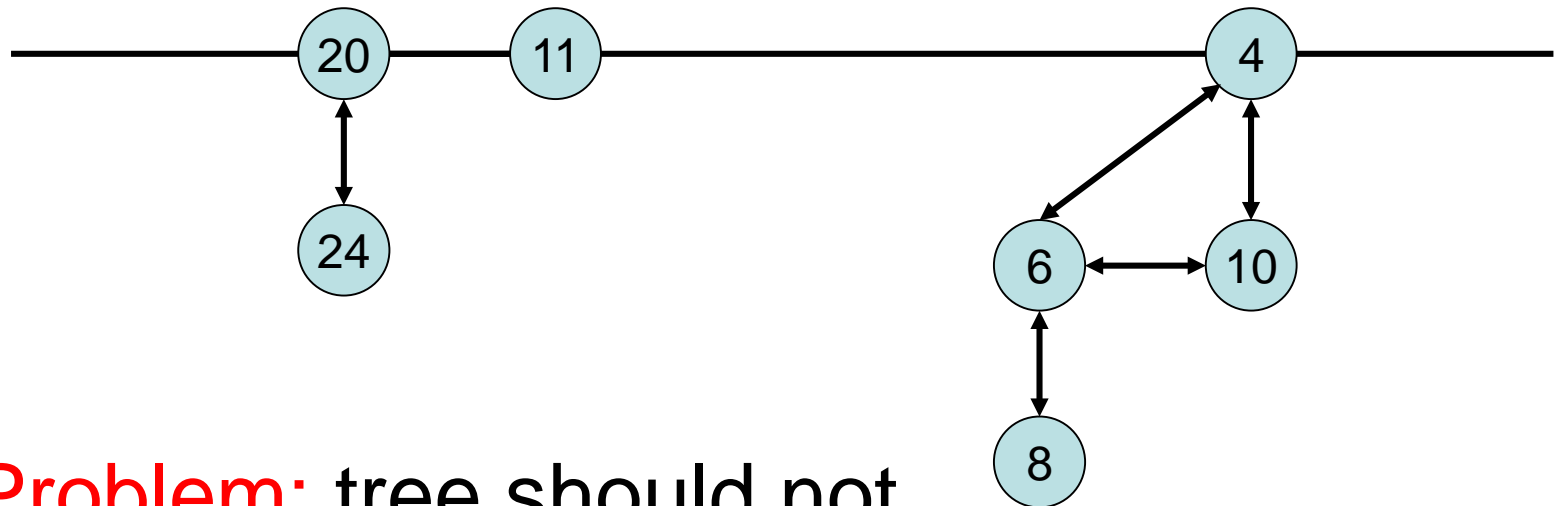


Problem: tree should not
lose too many nodes that way
→ is checked with variable mark

# 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 how many children v has lost due to a lazy delete (unless v is a root node, where mark(x) will always be set to 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 $v$, $key(v) \leq key(\text{children of } v)$.
  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,$\Delta$): 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

Root with
smaller key
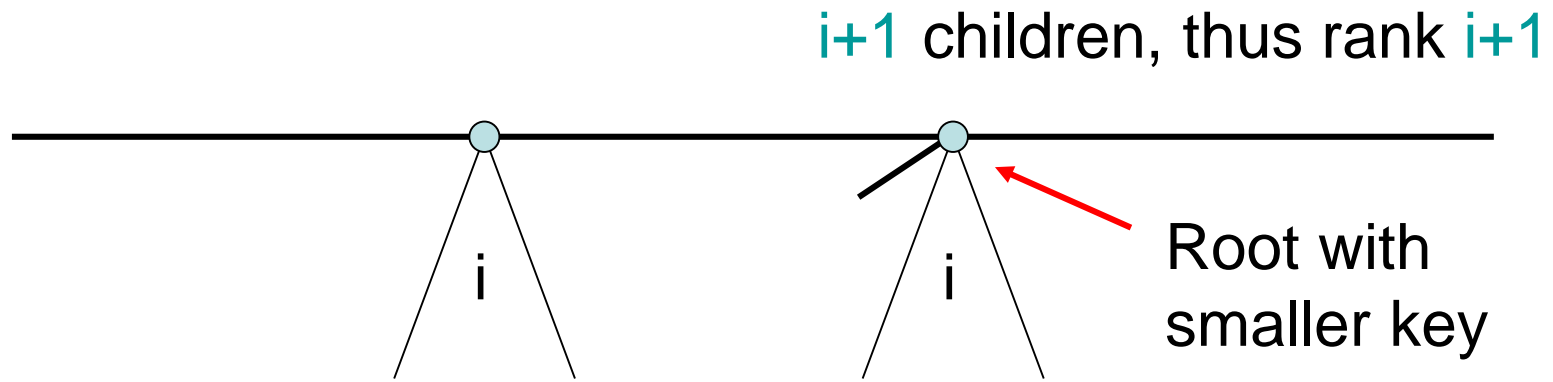
# Fibonacci Heap

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:
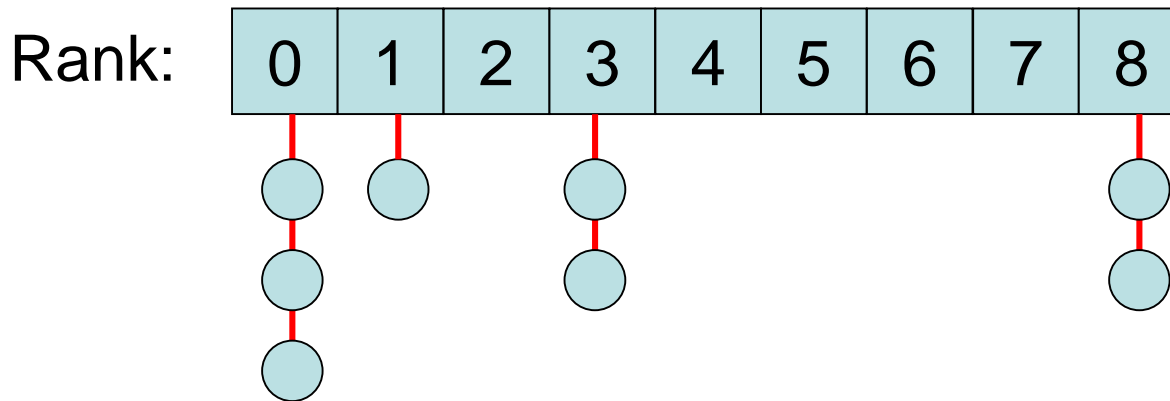


Rank:

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

# Fibonacci Heap

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)
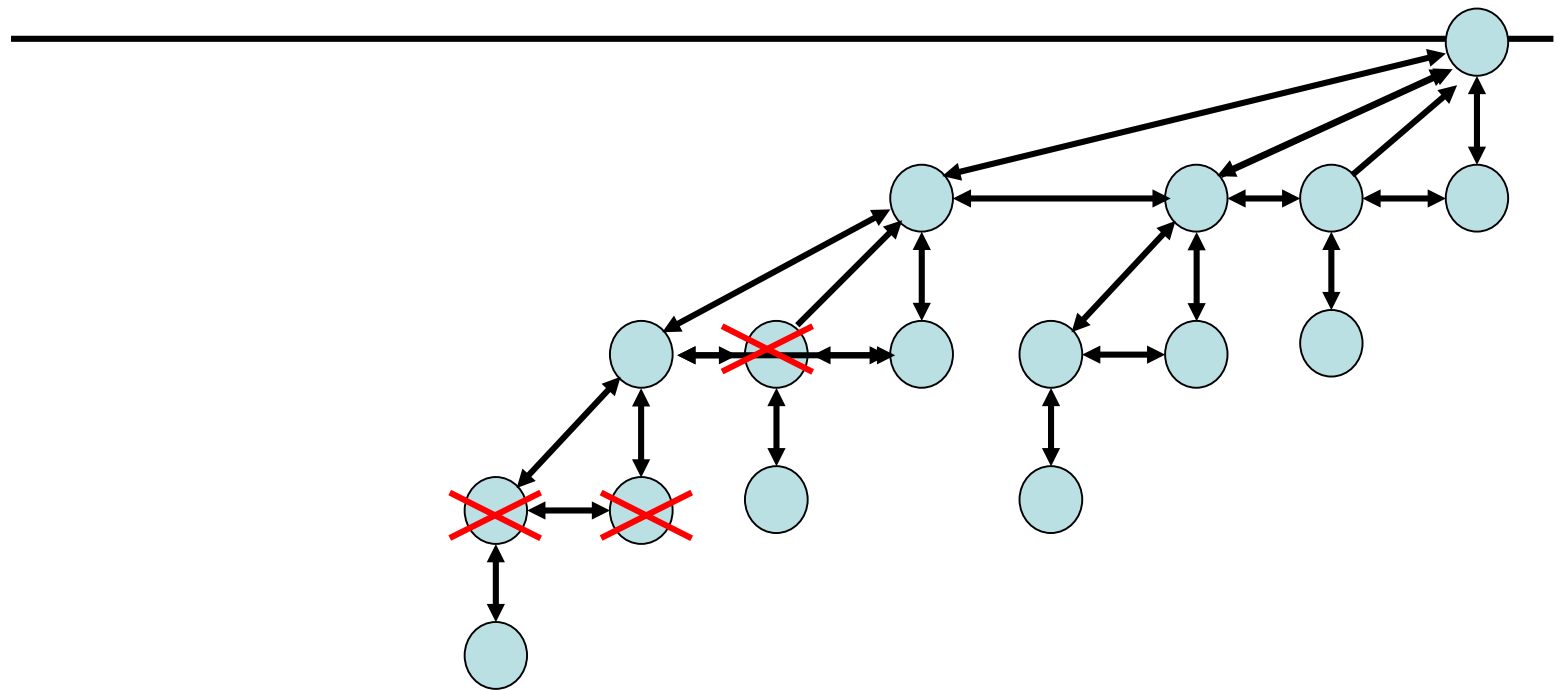
# Fibonacci Heap

Algorithm decreaseKey(x,$\Delta$):
  y:=P(x)
  move x with its subtree into root list
  parent(x):=NULL; mark(x):=0
  key(x):=key(x)-$\Delta$
  update min-pointer
  while y$\neq$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:=P(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(), delete(x):
  O(max. rank + #tree mergings)

- decreaseKey(x,$\Delta$):
  O(1 + #cascading cuts)
  i.e., #relocated marked nodes

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 $\phi : S \to \mathbb{R}_{\geq 0}$ 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

# 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.

- Due to the tree merging rule, 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.

# 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 $k>1$

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

# Fibonacci Heap

Proof of Theorem 2.2:

- Let $f_k$ be the minimal number of elements in a tree of rank $k$.

- From Lemma 2.2 we get:

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

1. child

root

- Moreover, $f_0 = 1$ and $f_1 = 2$

- It follows from the Fibonacci numbers:

$$f_k \geq F_{k+2}$$
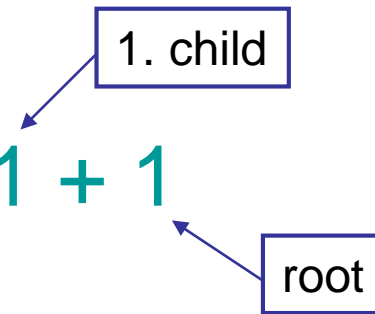
# Fibonacci Heap

- 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 $a = O(1)$
- min: $t = O(1)$ and $\Delta bal = 0$, so $a = O(1)$

# Fibonacci Heap

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

Proof:

- Insertion of child list of $x$ into root list:
  $\Delta bal \leq rank(x) - 1$
  since #trees increases by rank(x)-1 and some marks might be set from 1 to 0
- Every merging reduces #trees by 1:
  $\Delta bal = -$ #mergings
- Because of Theorem 2.2 (rank at most O(log n)) we get:
  #mergings = #trees – O(log n)
- Altogether: $\Delta bal_i = rank(x) -$ #trees + O(log n)
- Real runtime (in appropriate time units):
  $t_i =$ #trees + O(rank(x)) = #trees + O(log n)
- Amortized runtime:
  $a_i = t_i + \Delta bal_i = O(\log n)$

# Fibonacci Heap

Theorem 2.4: The 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·#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

Theorem 2.5: The amortized runtime of decreaseKey(x,$\Delta$) 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·#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. |

# Radix Heap

Assumptions:

1. All keys are integer values and have a distance of at most C from each other

2. Insert(e) only inserts elements e with $key(e) \geq k_{min}$ ($k_{min}$: minimum key)

Second assumption: we are searching for a monotonic priority queue

# Radix Heap

Let B[-1..K] be array of lists B[-1] to B[K], where
K=1+⌊log C⌋.
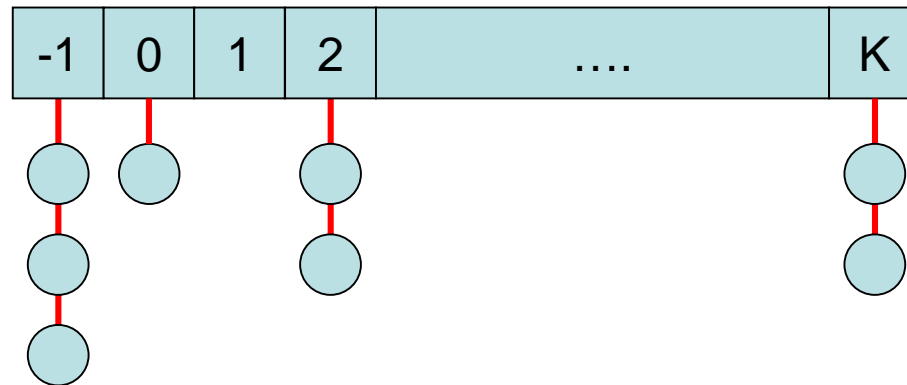


Invariant: Every e stored in $B[\min(\mathrm{msd}(k_{min},\mathrm{key}(e)),K)]$

- $\mathrm{msd}(k_{min},\mathrm{key}(e))$: maximum bit position at which binary representations of $k_{min}$ and $\mathrm{key}(e)$ differ (-1: no difference)

# 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():
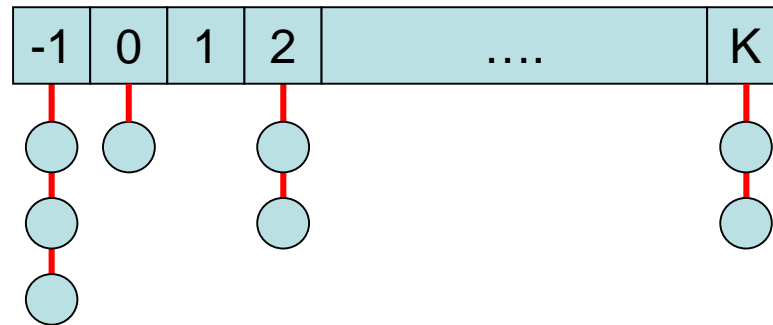
- output $k_{min}$ in B[-1]

Runtime: O(1)

# Radix Heap

insert(e): ( key(e)$\geq k_{min}$ )
- i:=min{msd($k_{min}$,key(e)), K}
- 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)

decreaseKey(x,$\Delta$): ( 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 occupied, remove some e from B[-1] (otherwise the heap is empty and we are done)
- find minimal i so that $B[i] \neq \varnothing$ (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 $k_{min}$

Important: for all e in B[j], j>i, it still holds that $msd(k_{min}, key(e))=j$, i.e., they do not have to be moved.

# Radix Heap
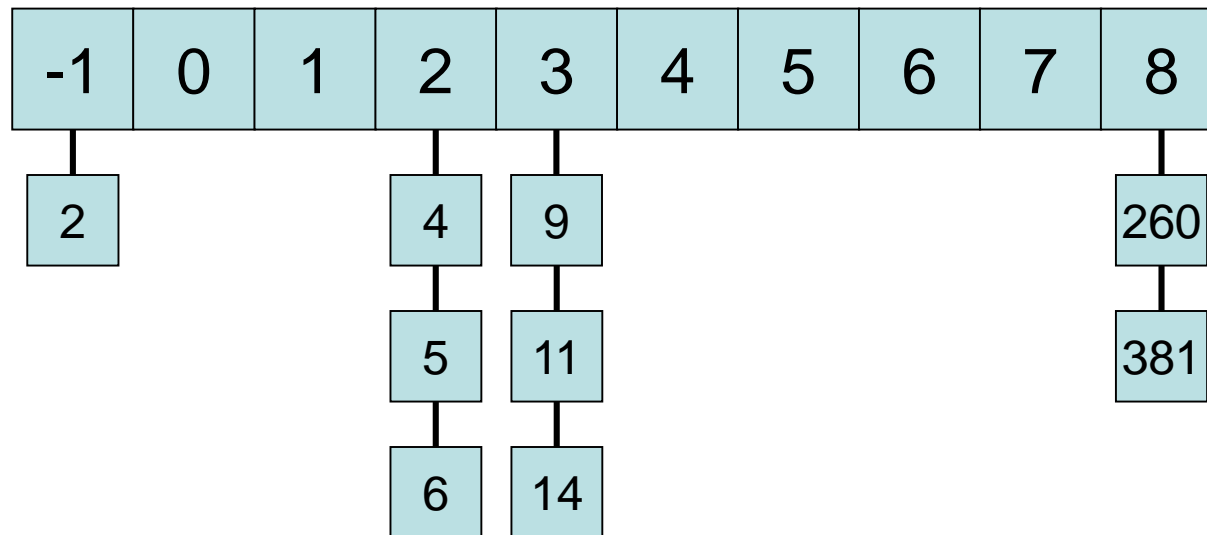
| -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|----|---|---|---|---|---|---|---|---|---|

2

4    9    260

5    11    381

6    14

We consider a sequence of deleteMin operations

# Radix Heap

| -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|----|---|---|---|---|---|---|---|---|---|

9     11    14                  260

381
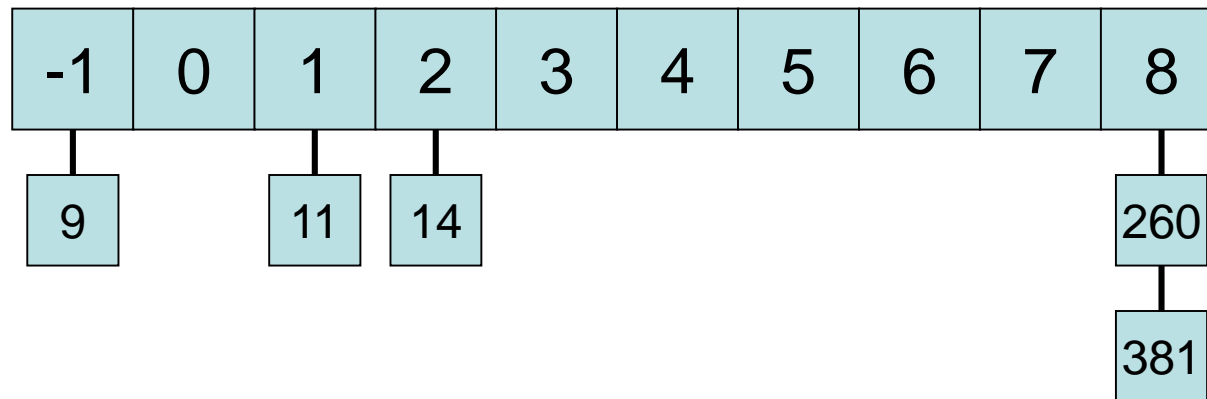
## We consider a sequence of deleteMin operations

# Radix Heap

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 $B[i]$.

all elements in B[i] move to the left

Proof:

- $x = x_{min}$: obviously true ($x$ placed in $B[-1]$)
- $x \neq x_{min}$: we distinguish between two cases: 1) $i < K$, 2) $i = K$

# Radix Heap

## Case i<K:

bit position: i                                    0

old $k_{min}$     | a | 0 |

$x_{min} > k_{min}$

$x_{min}$        | a | 1 |

$x, x_{min}$ in $B[i]$

$x$             | a | 1 |

Thus, $msd(x_{min}, x) < i$.

# Radix Heap

Fall i=K:



bit position: j≥K      h      0

old $k_{min}$: a | 0

$x_{min} > k_{min}$

$x_{min}$: a | 1 | b | 0

$x > x_{min}$

x: a | 1 | b | 1
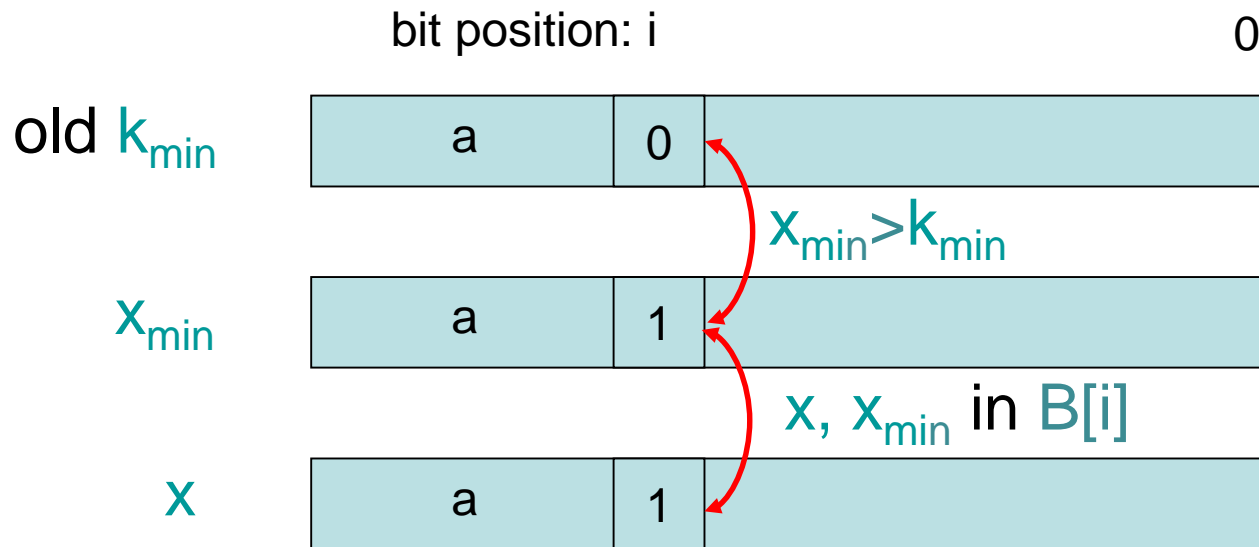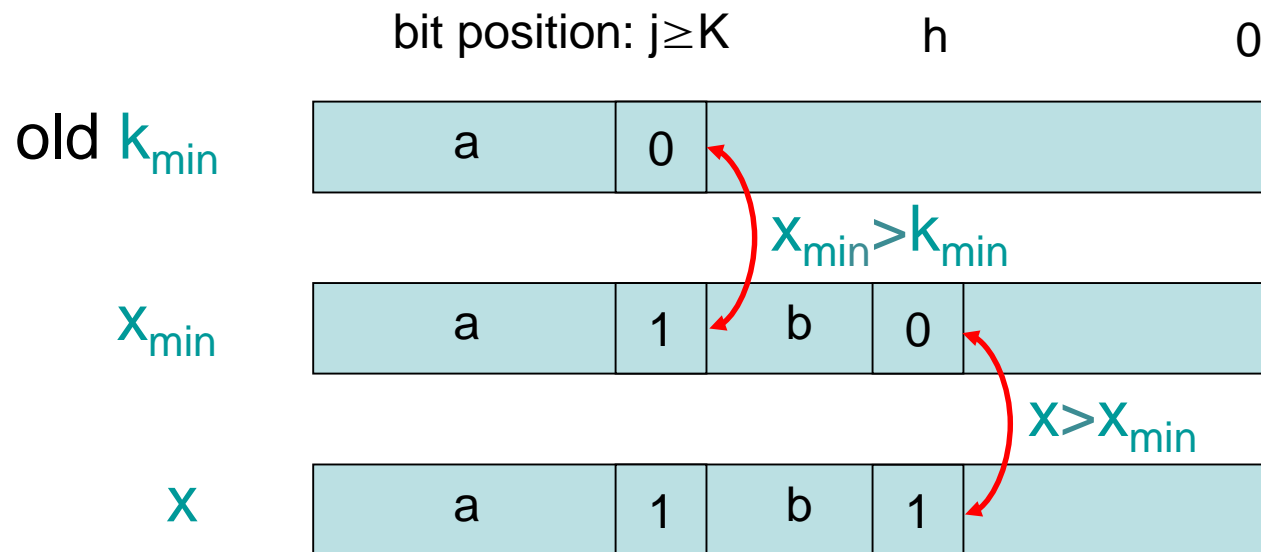
- It holds: $k_{min} < x_{min} < x \leq k_{min}+C$ and $2^K > C$, so x and $x_{min}$ must have identical prefix a1.

# Radix Heap

## Case i=K:

bit position: j≥K          h            0

| old $k_{min}$ | a | 0 | | |
| --- | --- | --- | --- | --- |

$x_{min} > k_{min}$

| $x_{min}$ | a | 1 | b | 0 | |
| --- | --- | --- | --- | --- | --- |

$x > x_{min}$

| x | a | 1 | b | 1 | |
| --- | --- | --- | --- | --- | --- |
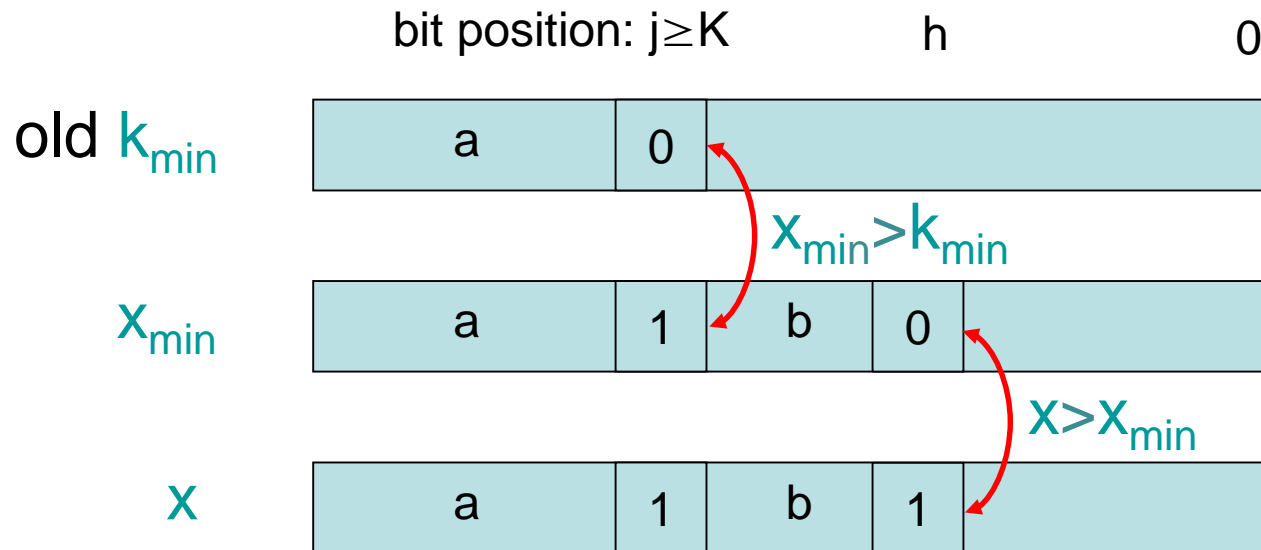
- Let $h = \mathrm{msd}(x_{min}, x)$. Since $2^K > C$, $h$ must be less than $K$.

# Radix Heap

| -1 | 0 | 1 | 2 | …. | K |
|----|---|---|---|----|---|
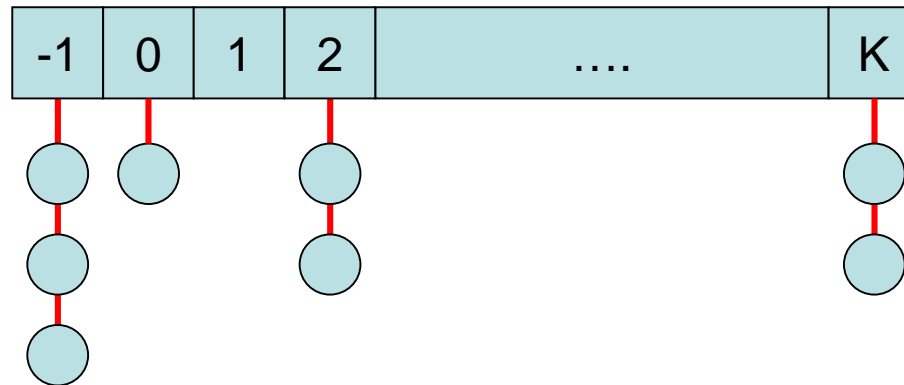
**Consequence:**
- Every element can only be moved at most K times in the Radix heap (due to deleteMin or decreaseKey operations)

- insert(): amortized runtime O(log C).

# 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) | O(1) |

# Extended Radix Heap

Assumptions:

1. All keys are integer values and have a distance of at most $C$ from each other

2. ~~Insert(e) only inserts elements $e$ with key(e)$\geq k_{min}$ ($k_{min}$: minimum key)~~

Second assumption: monotone priority queue

# Extended Radix Heap

| -1 | 0 | 1 | 2 | .... | K |
|----|---|---|---|------|---|

At least one normal element

super elements at the end
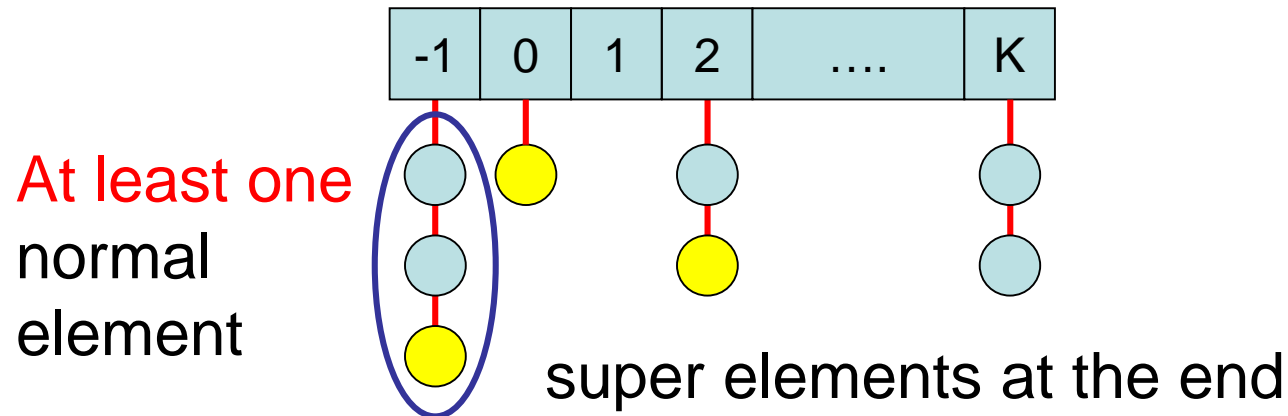
🟡 : "super element" e, which contains a Radix heap with $k_{min}=key(e)$ where $k_{min}$ is the smallest value in the Radix heap of e and $B_e[-1]$ has $\geq 1$ normal element.

Note: super elements may contain super elements

# Extended Radix Heap
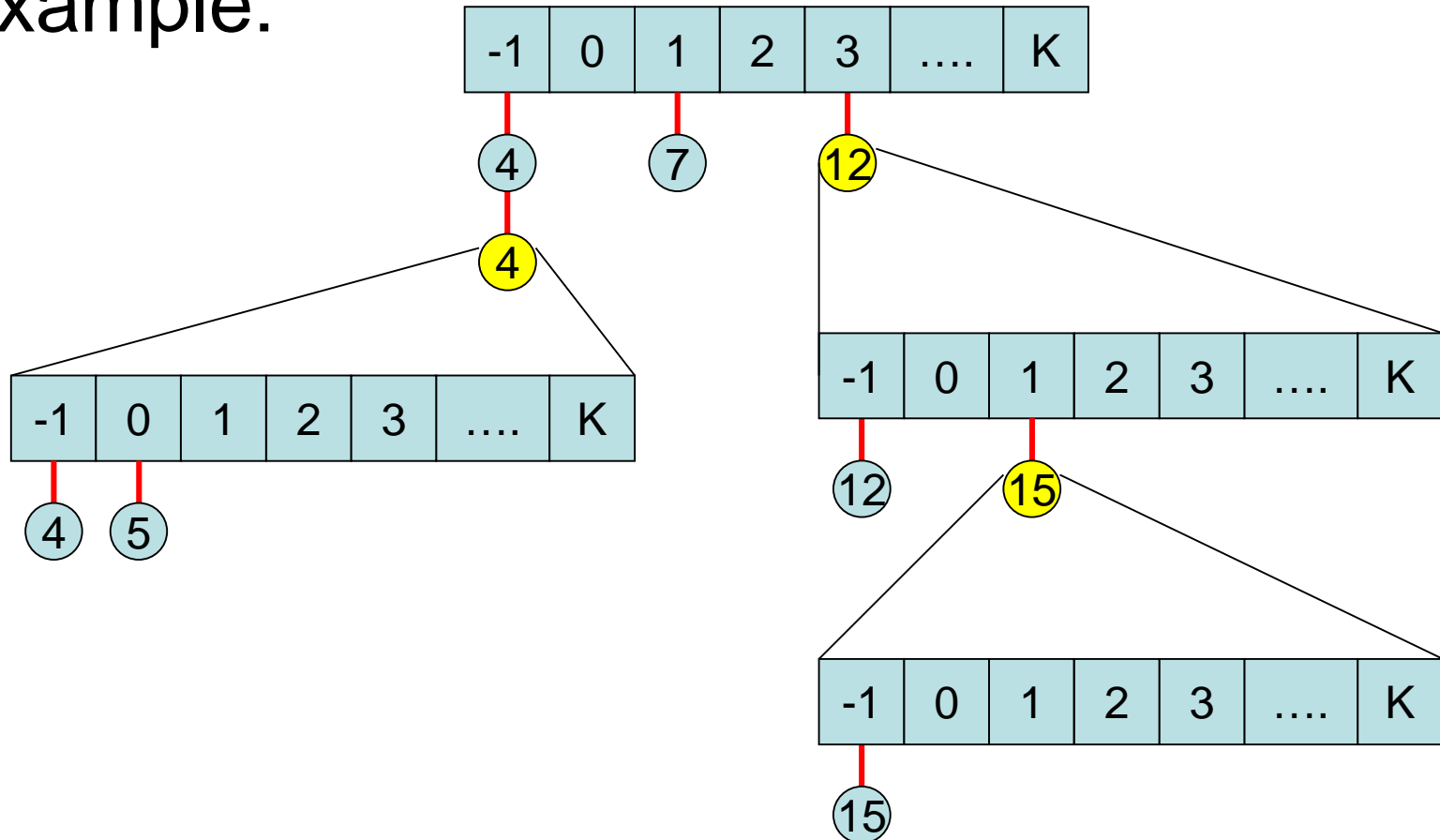
Example:

# 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:

| -1 | 0 | 1 | 2 | 3 | .... | K |
|----|---|---|---|---|------|---|

4

&

→

| -1 | 0 | 1 | 2 | 3 | .... | K |
|----|---|---|---|---|------|---|

4    7

| -1 | 0 | 1 | 2 | 3 | .... | K |
|----|---|---|---|---|------|---|

7

| -1 | 0 | 1 | 2 | 3 | .... | K |
|----|---|---|---|---|------|---|

7
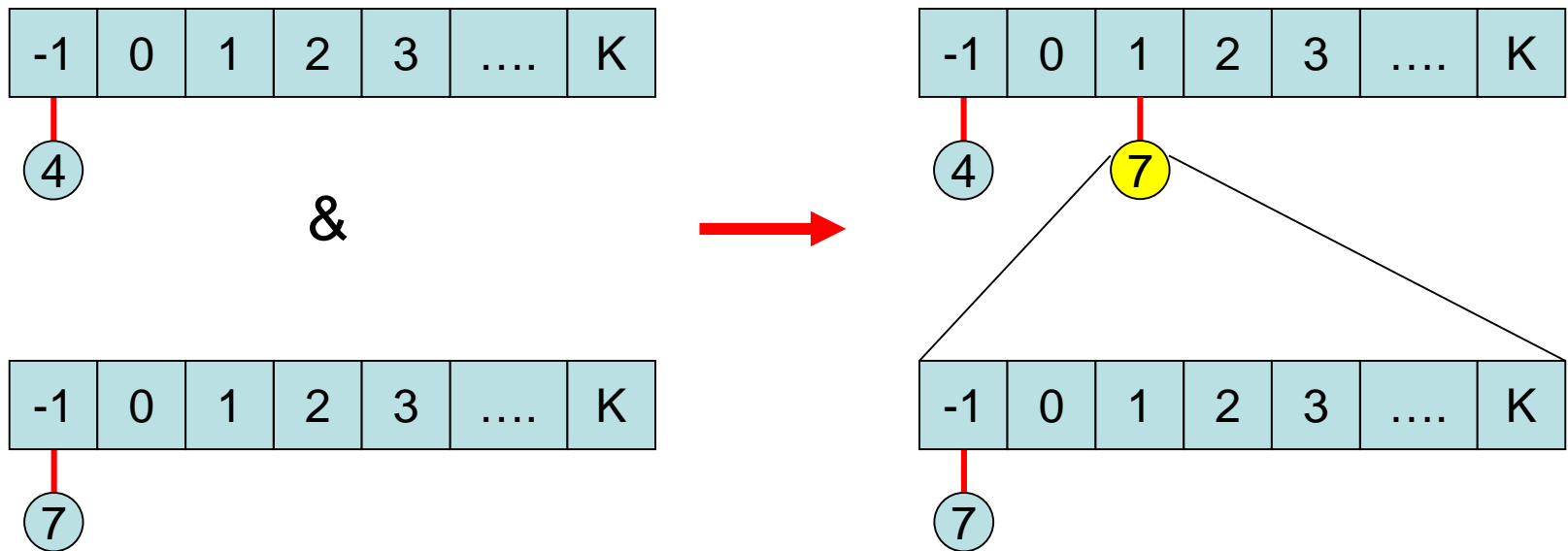
# Extended Radix Heap

insert(e):

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

Runtime: O(1)


min(): like in a standard Radix heap

Runtime: O(1)

# Extended Radix Heap

deleteMin():

- Remove normal element e from B[-1]
  (B: Radix heap at highest level)
- 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():

| -1 | 0 | 1 | 2 | 3 | .... | K |
|----|---|---|---|---|------|---|

(4)  (5)  (7)  (12)

| -1 | 0 | 1 | 2 | 3 | .... | K |
|----|---|---|---|---|------|---|

(12)  (15)

| -1 | 0 | 1 | 2 | 3 | .... | K |
|----|---|---|---|---|------|---|

(15)

# 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:

- like deleteMin() above but on heap of e
- if e was 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!

Runtime: $O(\log C)$ + time for updates

# Extended Radix Heap

delete(10):

| -1 | 0 | 1 | 2 | 3 | .... | K |
|----|---|---|---|---|------|---|

4    7    8

4

| -1 | 0 | 1 | 2 | 3 | .... | K |
|----|---|---|---|---|------|---|

4    5

merge Radix heap of 4
into top Radix heap

| -1 | 0 | 1 | 2 | 3 | .... | K |
|----|---|---|---|---|------|---|

8    10

| -1 | 0 | 1 | 2 | 3 | .... | K |
|----|---|---|---|---|------|---|

~~10~~    14

# Extended Radix Heap

delete(10):

|  -1  |  0  |  1  |  2  |  3  |  ….  |  K  |

4   7   8

4

|  -1  |  0  |  1  |  2  |  3  |  ….  |  K  |

4   5

|  -1  |  0  |  1  |  2  |  3  |  ….  |  K  |

8   14

merge Radix heap of  4
into top Radix heap

|  -1  |  0  |  1  |  2  |  3  |  ….  |  K  |

14

# Extended Radix Heap

delete(10):

| -1 | 0 | 1 | 2 | 3 | .... | K |

(4)  (7)  (8)

(4)

| -1 | 0 | 1 | 2 | 3 | .... | K |

(4)  (5)

merge Radix heap of (4)
into top Radix heap

| -1 | 0 | 1 | 2 | 3 | .... | K |

(8)  (14)

| -1 | 0 | 1 | 2 | 3 | .... | K |

(14)
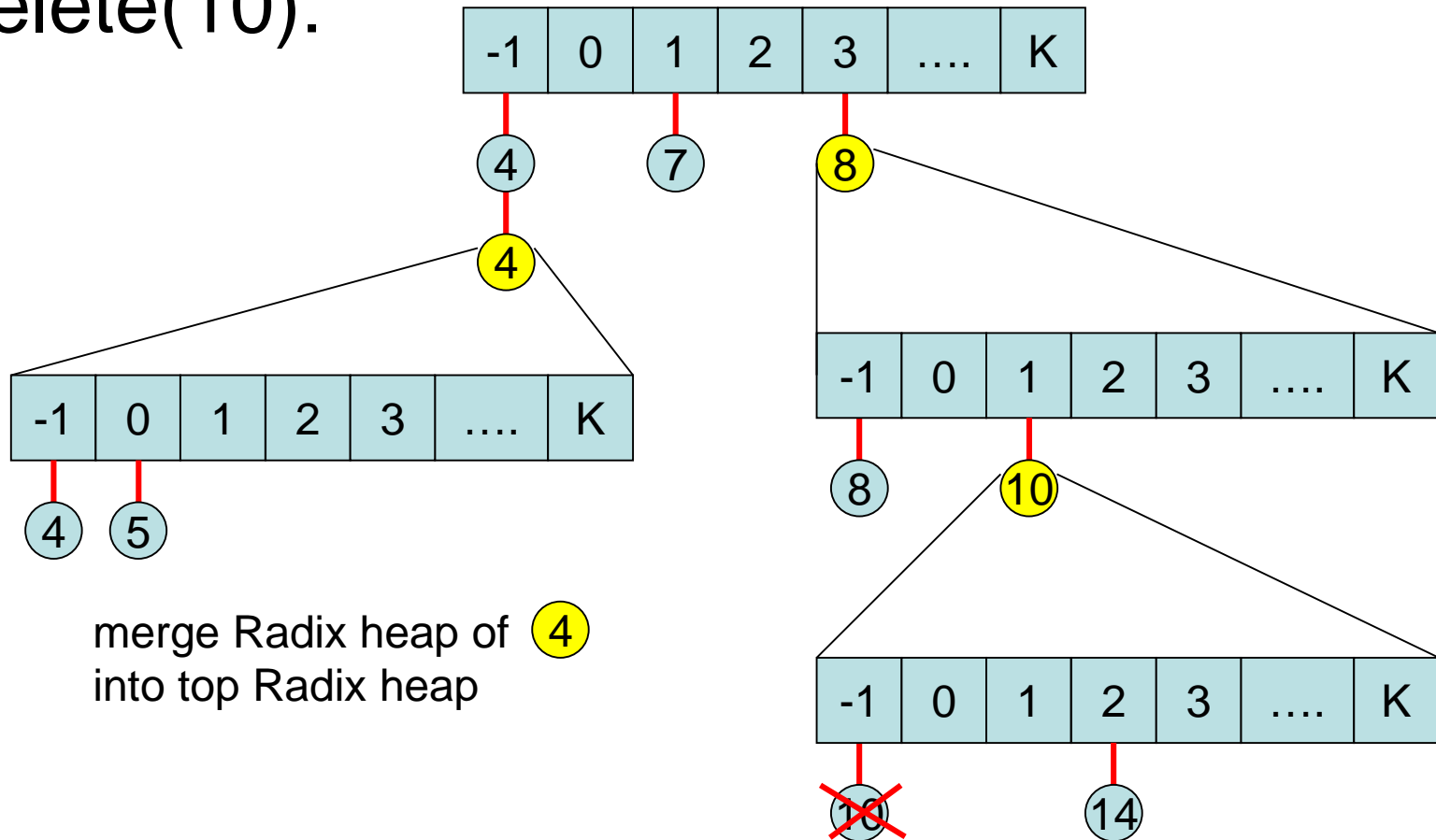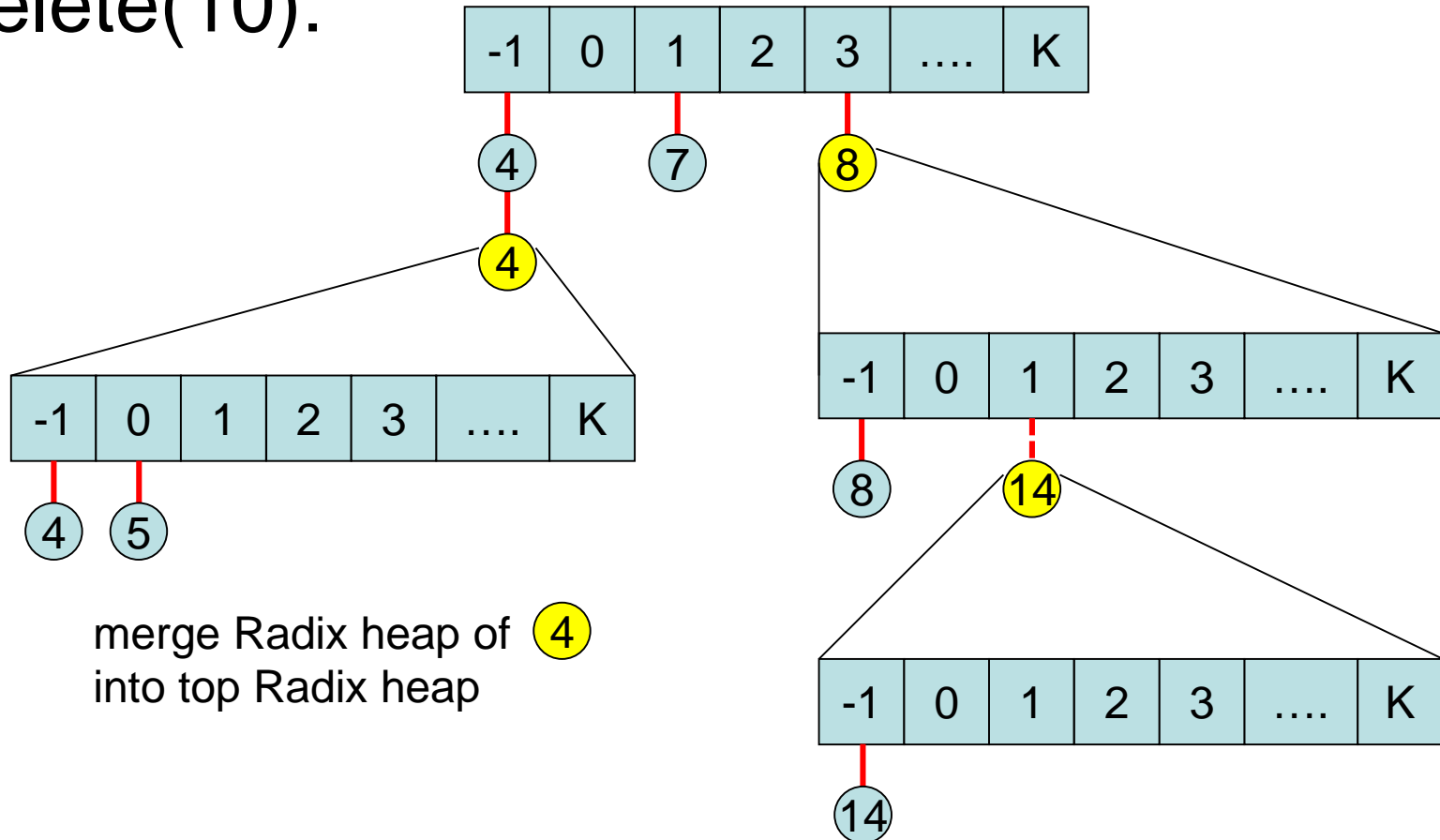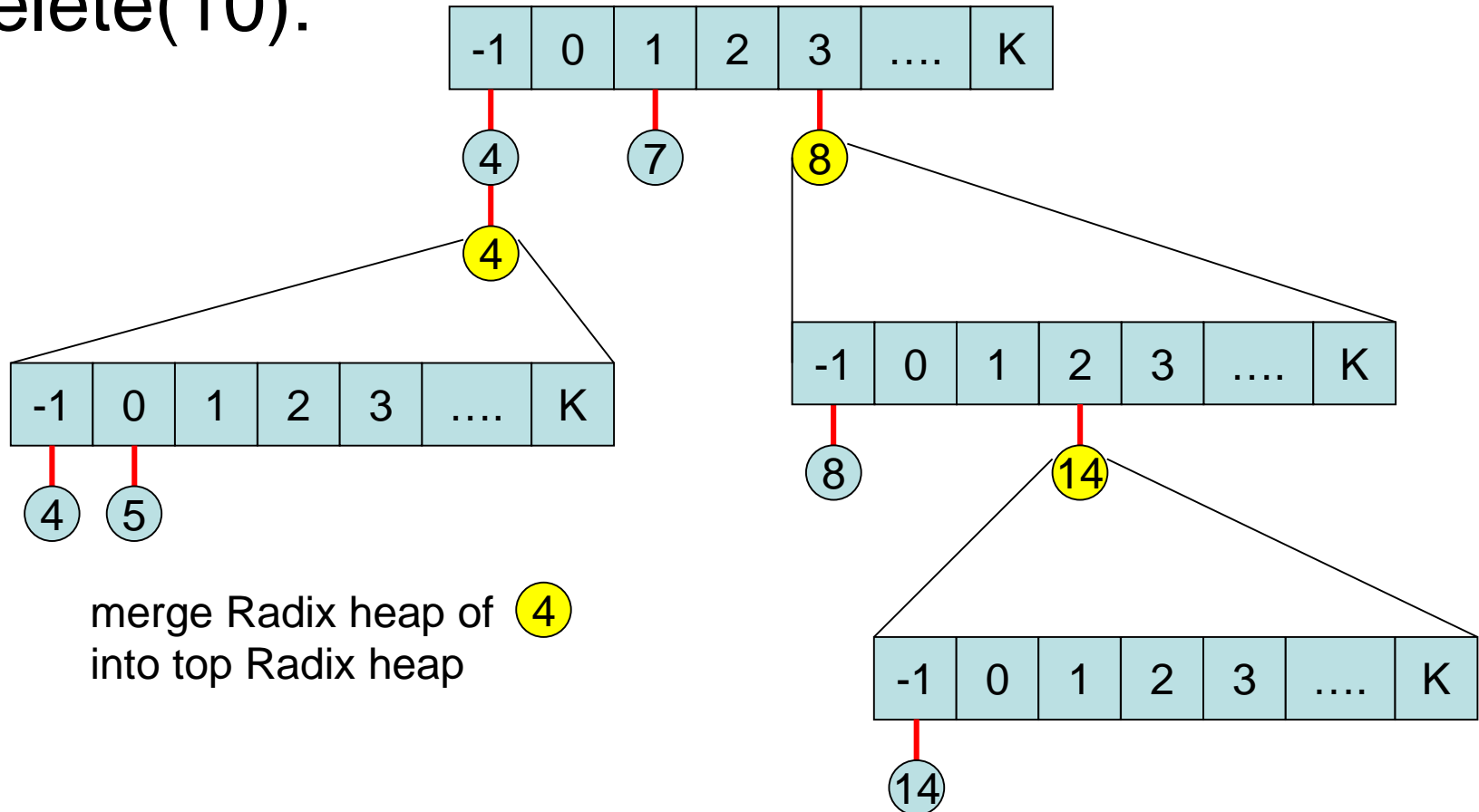
# Extended Radix heap

decreaseKey(e,$\Delta$):
- call delete(e) in heap of e
- set key(e):=key(e)-$\Delta$
- call insert(e) on highest Radix heap

Runtime: O(log C) + time for updates

Amortized analysis: similar to Radix heap
- each time a normal element e is inserted, the potential is increased by K+pos(e) (to compensate for pos(e) left moves of itself and a right move of its superelement e if it is removed as the minimum element in the Radix heap of e)
- each time a superelement e is inserted, the potential is increased by K+pos(e) (to compensate for pos(e) left moves and the merging of up to K lists in its Radix heap if it is removed from B[-1] in deleteMin)

# 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
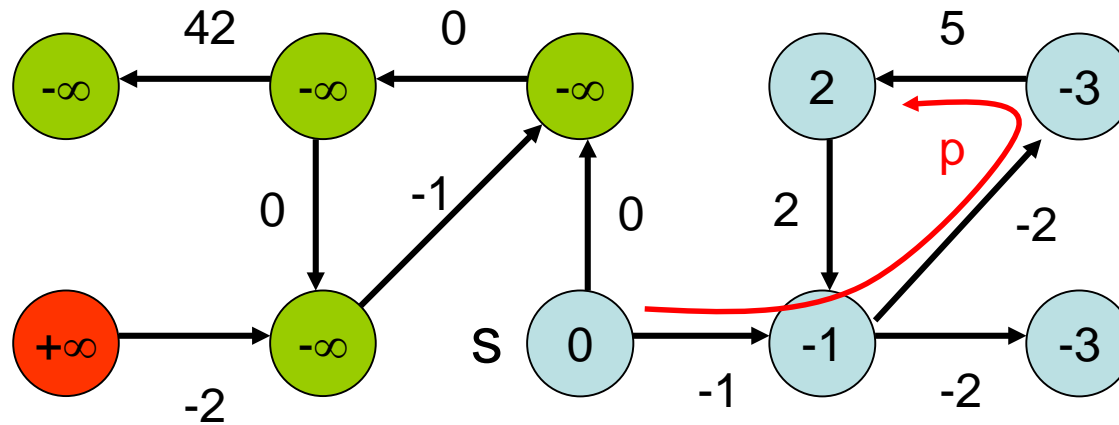- Fibonacci heap
- Radix heap
- Applications

# Shortest Paths
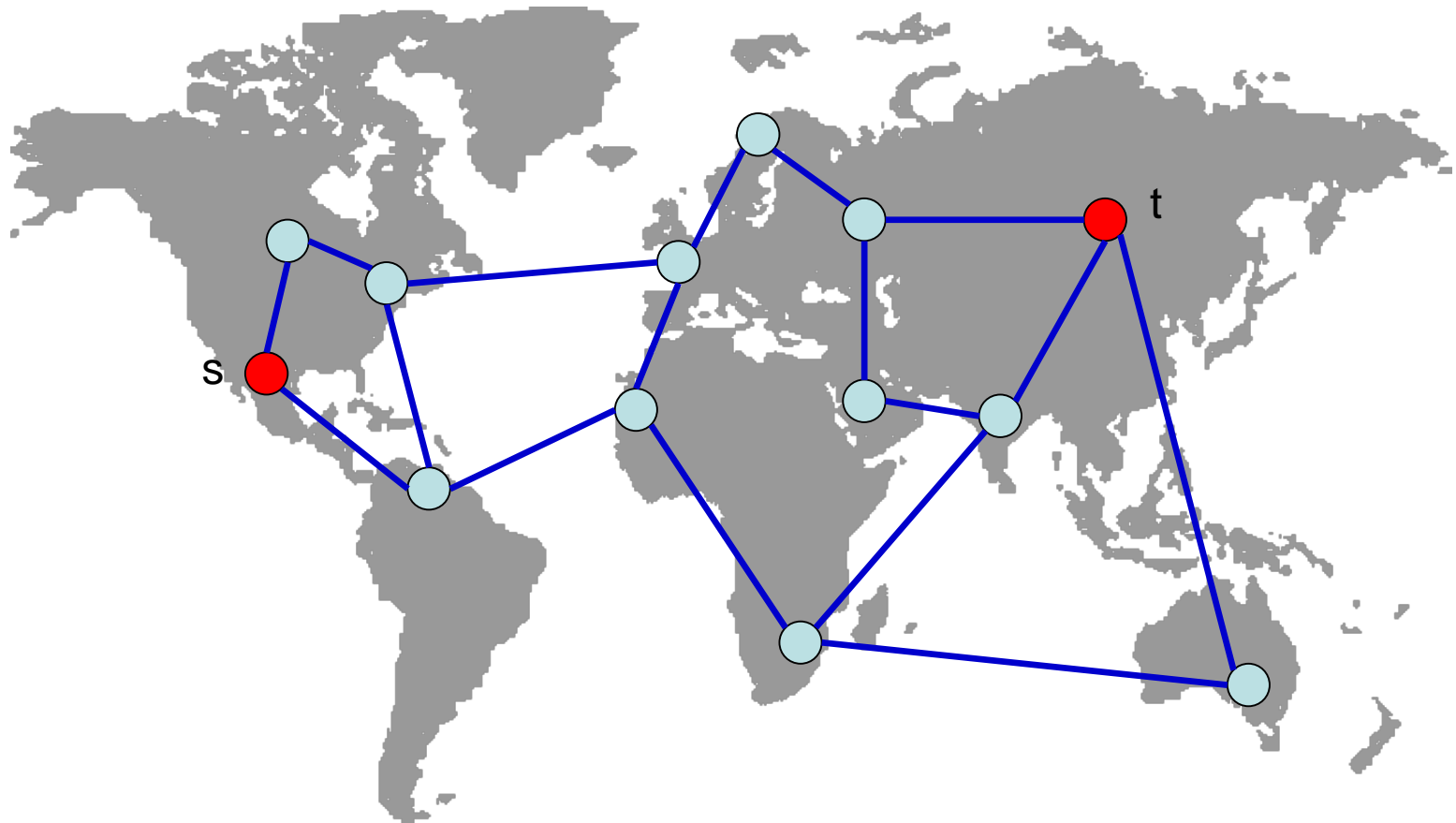


**Central question:** Determine fastest way to get from s to t.

# Shortest Paths



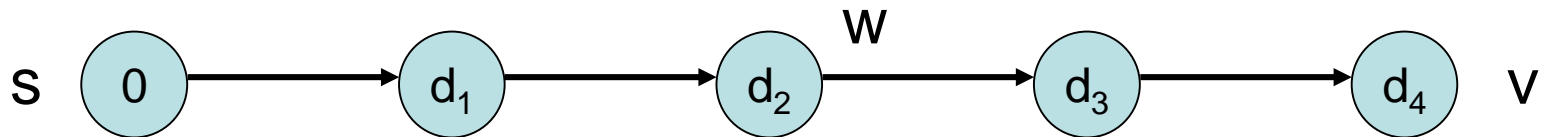Cost of p:
c(p)=-1+(-2)+5=2

μ(s,v): distance from s to v

$$\mu(s,v) = \begin{cases} \infty & \text{no path from s to v} \\ -\infty & \text{path of arbitrarily low cost from s to v} \\ \min\{ c(p) \mid p \text{ is a path from s to v}\} \end{cases}$$

# 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.

s $\quad$ 0 $\;\longrightarrow\;$ $d_1$ $\;\longrightarrow\;$ $d_2$ $\;\overset{w}{\longrightarrow}\;$ $d_3$ $\;\longrightarrow\;$ $d_4$ $\quad$ v

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 has not been in q so far, insert w into q. In order to repair q in case of a change of $d(w)$, perform a decreaseKey operation.

# Dijkstra´s Algorithm

Example: (🔴 : current, 🔵 : done)
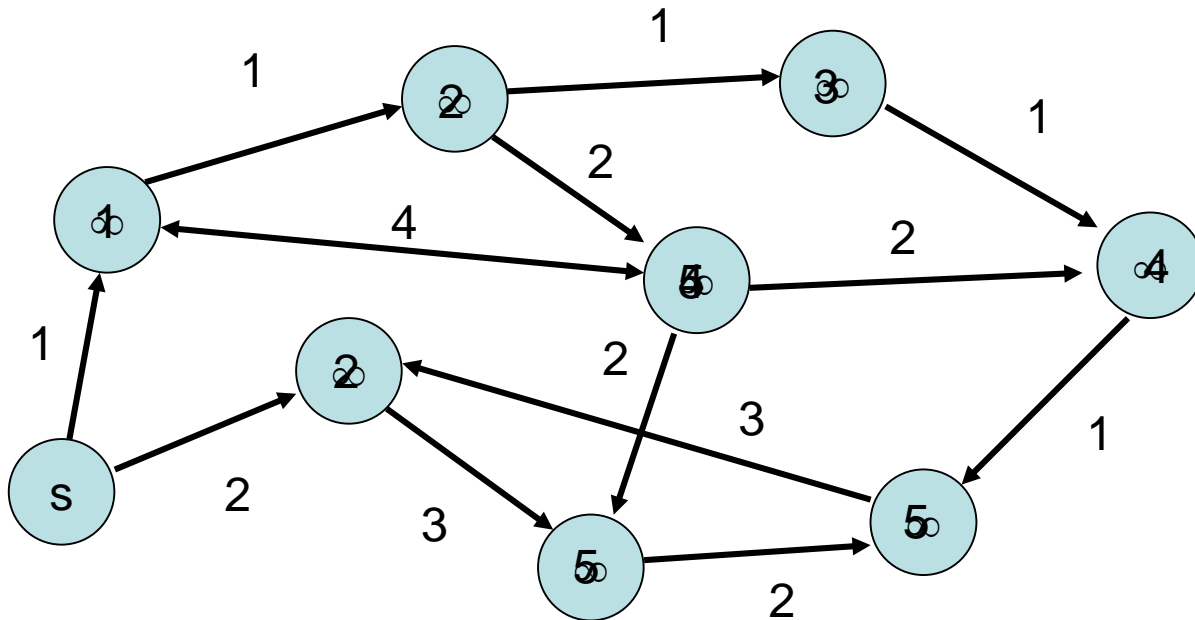
# Dijkstra´s Algorithm

Procedure Dijkstra(s: NodeId)
  d=<∞,…, ∞>: NodeArray of $\mathbb{R}\cup\{-\infty,\infty\}$
  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

- $T_{Op}(n)$: runtime of operation $Op$ on data structure with $n$ elements

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., minima are monotonically increasing)

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?
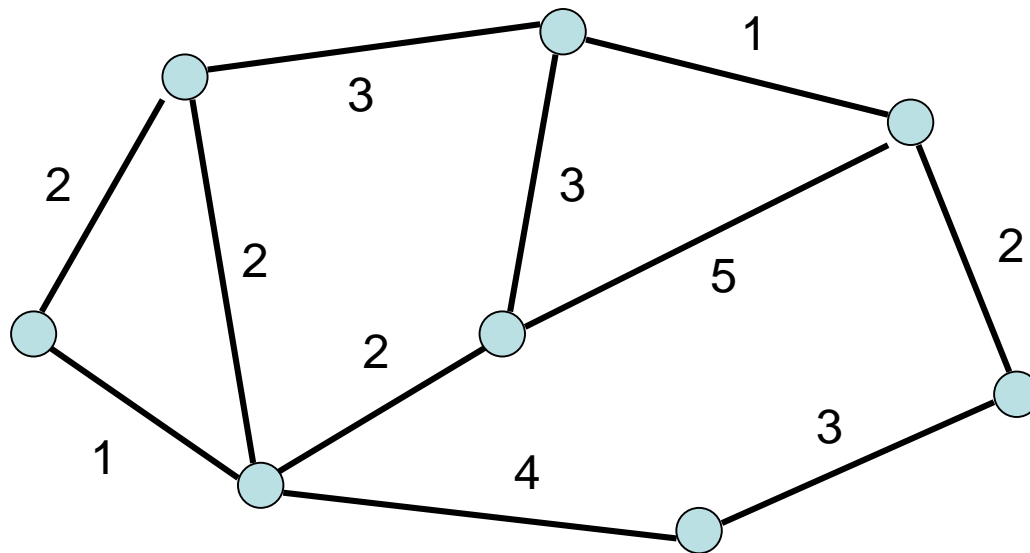
# Minimal Spanning Tree

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

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).
- Tree over all nodes in V with minimum cost: minimal spanning tree (MST)

# Prim´s Algorithm

Procedure Prim(s: NodeId)
  d=<∞,…, ∞>: NodeArray of $\mathbb{R}\cup\{-\infty,\infty\}$
  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))

# Prim´s Algorithm

- $T_{Op}(n)$: 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, $T_{Prim} = O(n \log n + m)$

# Next Chapter

Topic: Search structures