Algorithmen und Datenstrukturen

Prioritätswarteschlangen mit binären Heaps

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Magnus Bender (Übungen)

sowie viele Tutoren

Kurze Wiederholung

- Abstrakte Datentypen
  - Arrays vs. Tupel
  - Listen vs. Arrays
  - Keller und Schlangen
- APIs
  - Mehr hierzu auch in der Vorlesung Software-Engineering
  - Realisierung von Arrays und Listen
- Bucket Sort

Danksagung

Die nachfolgenden Präsentationen wurden mit einigen Änderungen übernommen aus der Vorlesung „Effiziente Algorithmen und Datenstrukturen“ (Kapitel 2: Priority Queues) gehalten von Christian Scheideler an der TUM

http://www14.in.tum.de/lehre/2008WS/ea/index.html.de

Prioritätswarteschlangen

Geg.: $[e_1,\ldots,e_n]$ eine Menge von Elementen

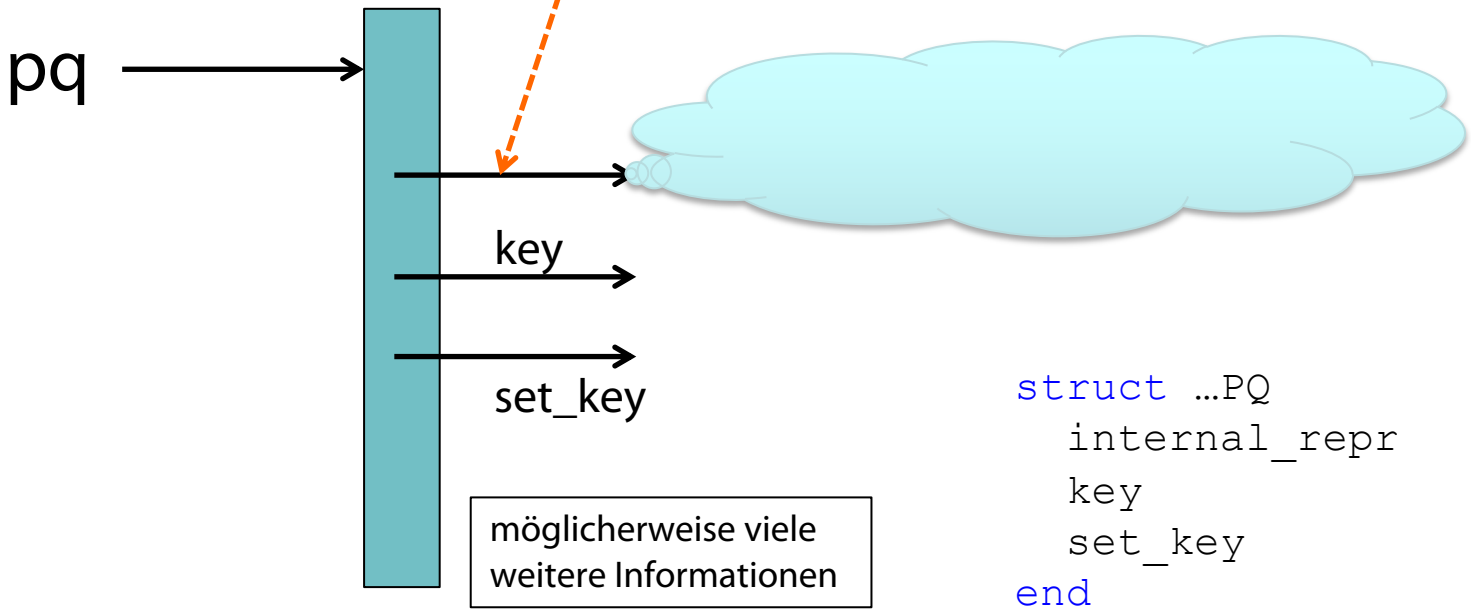
Ann.: Priorität eines jeden Elements e wird identifiziert über die Funktion key

Operationen:

- function build(elements, key, set_key) liefert neue Warteschlange, in der die Elemente e_i nach key(e_i) priorisiert verwaltet werden
  - Funktionen key, set_key werden von der erzeugten Warteschlange verwaltet
- procedure insert(e, pq) fügt Element e mit Priorität key(e) in pq ein, verändert pq sofern e noch nicht in pq
- function min_element(pq) gibt Element mit minimalem key(e) zurück
- procedure delete_min(pq): löscht das minimale Element in pq, und pq wird verändert, wenn etwas gelöscht wird
- function empty_pq(pq) prüft, ob Warteschlange pq leer

Prioritätswarteschlangen als ADTs

Im ADT-Sinne nur „intern“ verwendet, dann über pq.internal_repr referenziert




```
struct …PQ
   internal_repr
   key
   set_key
end
```

Erweiterte Prioritätswarteschlangen

Zusätzliche Operationen:

- procedure delete(e, pq) löscht e aus pq, falls vorhanden, verändert ggf. pq
- procedure decrease_key(e, pq, Δ):
  pq.set_key(e, pq.key(e)-Δ), verändert evtl. pq
- procedure merge(pq, pq‘) fügt pq und pq‘ zusammen, verändert ggf. pq und auch pq‘

Prioritätswarteschlangen

- Einfache Realisierung mittels unsortierter Liste:
  - build: Zeit O(n)
  - insert: O(1)
  - min_element, delete_min: O(n)
- Realisierung mittels sortiertem Feld:
  - build: Zeit O(n log n) (Sortieren)
  - insert: O(n) (verschiebe Elemente im Feld)
  - min: O(1)
  - delete_min: O(n) (verschiebe Elemente im Feld)

Bessere Struktur als Liste oder Feld möglich?

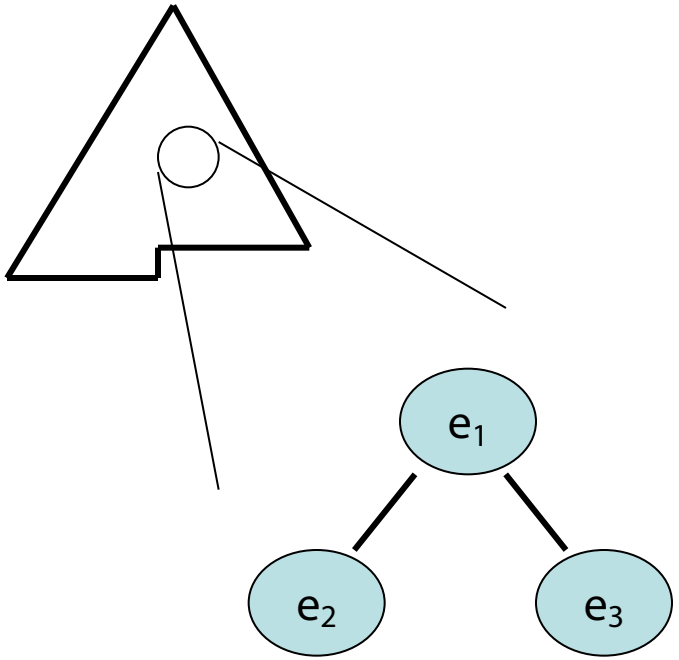
Binärer Heap (Wiederholung)

Idee: Verwende binären Baum statt Liste

Bewahre zwei Invarianten:

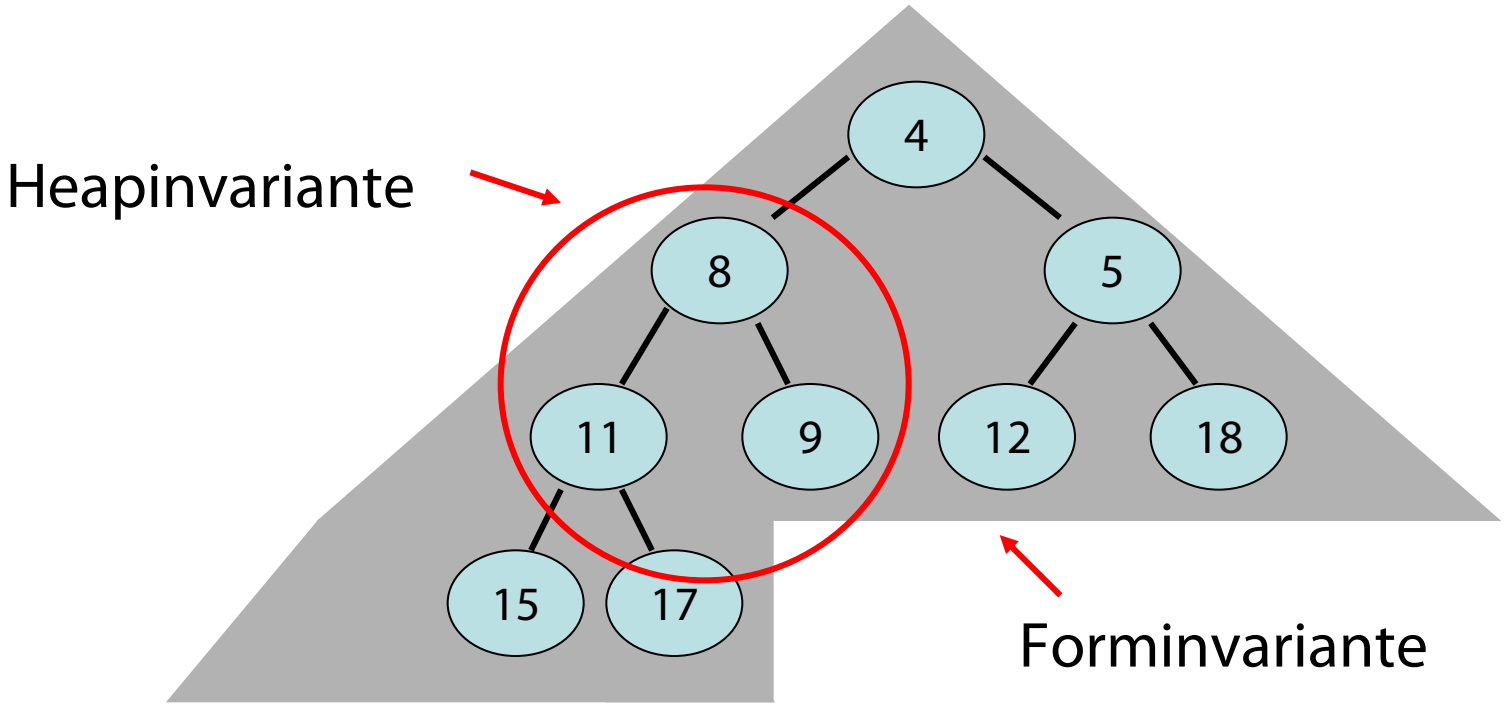
- Form-Invariante:
  vollst. Binärbaum bis auf unterste Ebene
- (Min)Heap-Invariante:

$key(e_1) \leq \min(\{ key(e_2), key(e_3) \})$
für die Kinder e_2 und e_3 von e_1



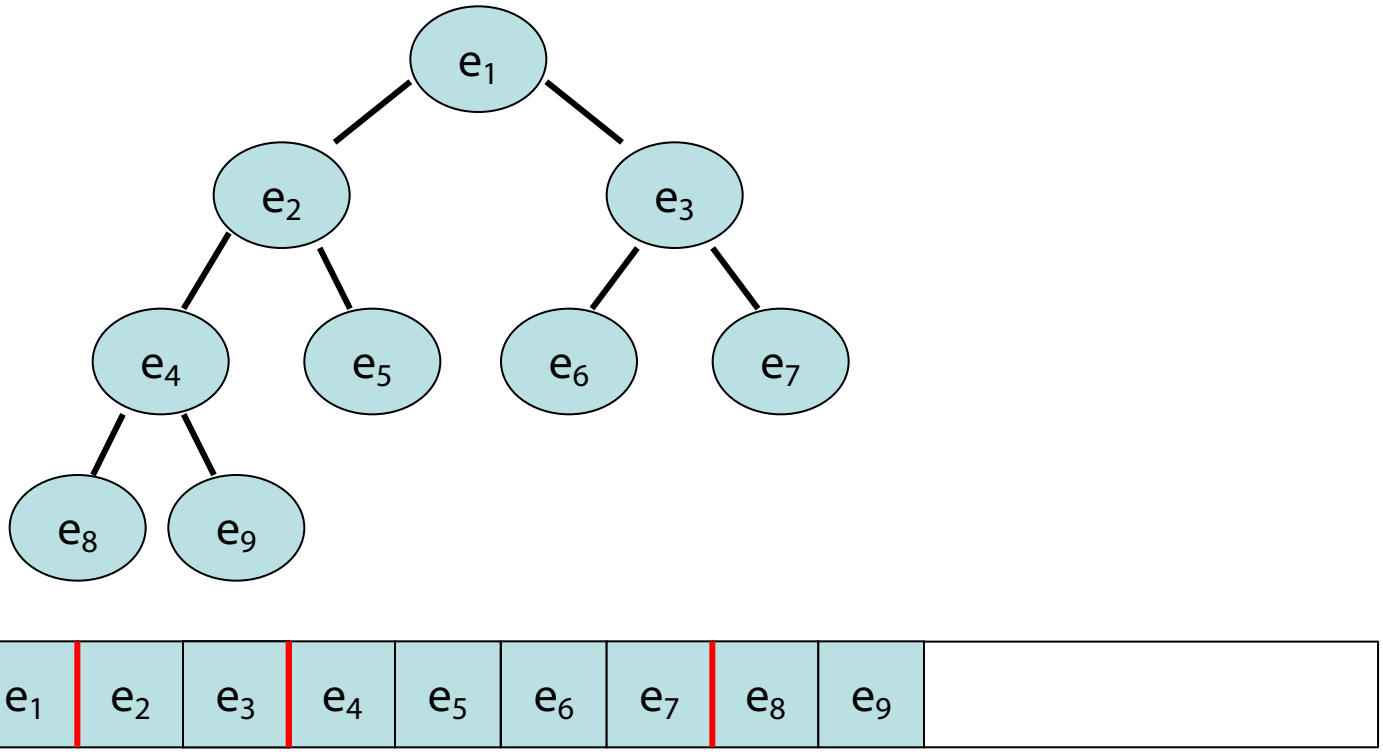
Binärer Heap (Wiederholung)

Beispiel:



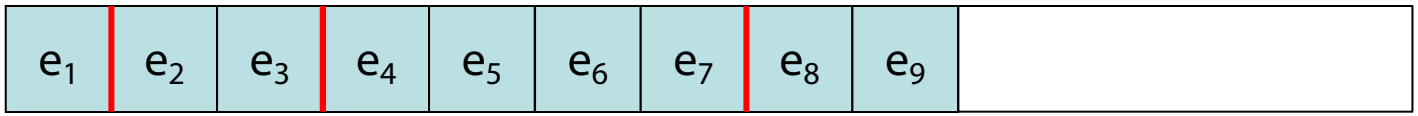
Binärer Heap (Wiederholung)

Realisierung eines Binärbaums als Feld:



Binärer Heap (Wiederholung)

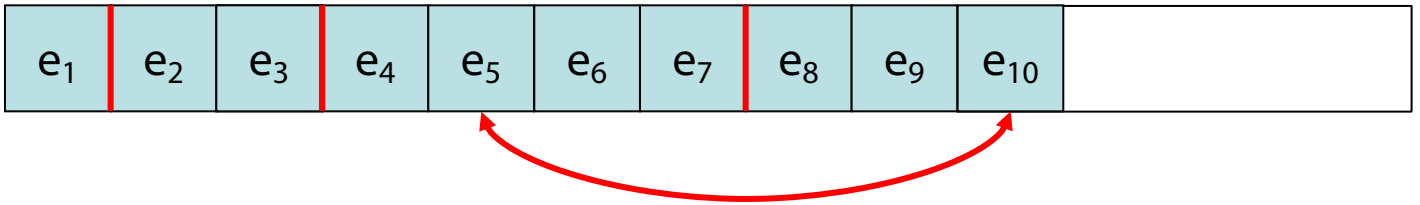
Realisierung eines Binärbaums als Feld:



- H: Array [1..n]
- Kinder von e in H[i]: in H[2i], H[2i+1]
- Form-Invariante: H[1],…,H[k] besetzt für k≤n
- Heap-Invariante:
  key(H[i])≤min({ key(H[2i]), key(H[2i+1]) })

Binärer Heap (Wiederholung)

Realisierung eines Binärbaums als Feld:



insert(e, pq): Sei H das Trägerfeld von pq
(H = pq.internal_repr)

- Form-Invariante: n = n+1; H[n] = e
- Heap-Invariante: vertausche e mit Vater bis key(H[⌊k/2⌋]) ≤ key(e) für e in H[k] oder e in H[1]

Insert Operation

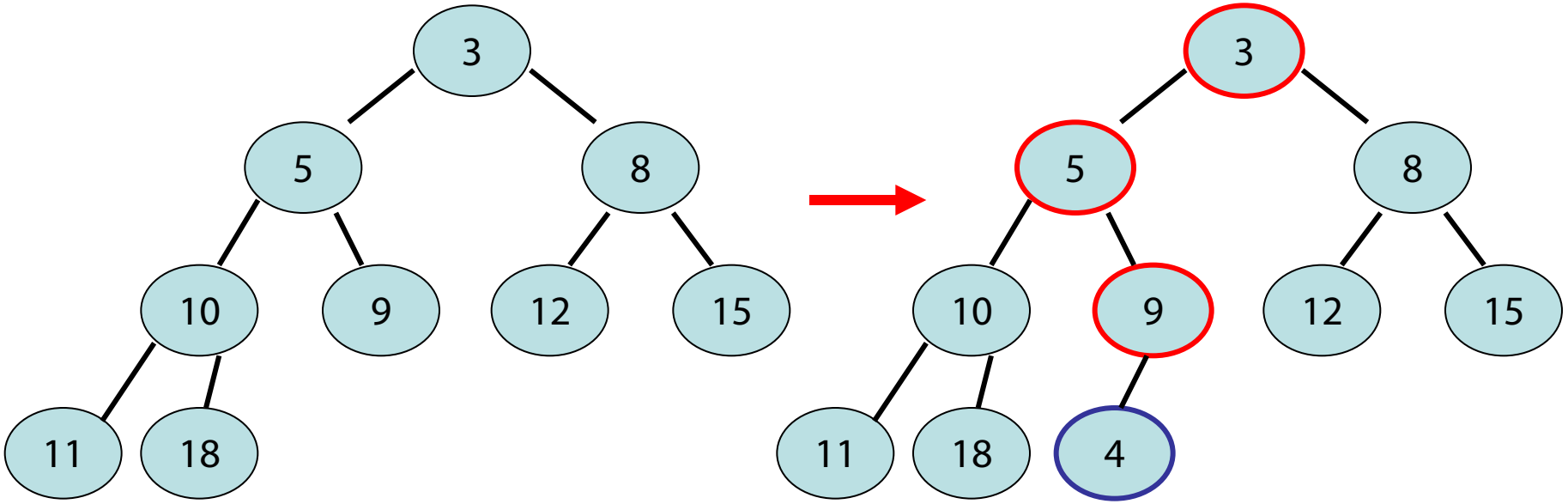
```
function insert(e, pq)
    H = pq.internal_repr
    n = length(H) + 1

    insert!(H, n, e)
    sift_up(n, pq)
end
function sift_up(i, pq)
    H = pq.internal_repr
    while i > 1 && pq.key(H[parent(i)] ) > pq.key(H[i])
        temp = H[i]
        H[i] = H[parent(i)]
        H[parent(i)] = temp

        i = parent(i)
    end
end
```

Laufzeit: O(log n)

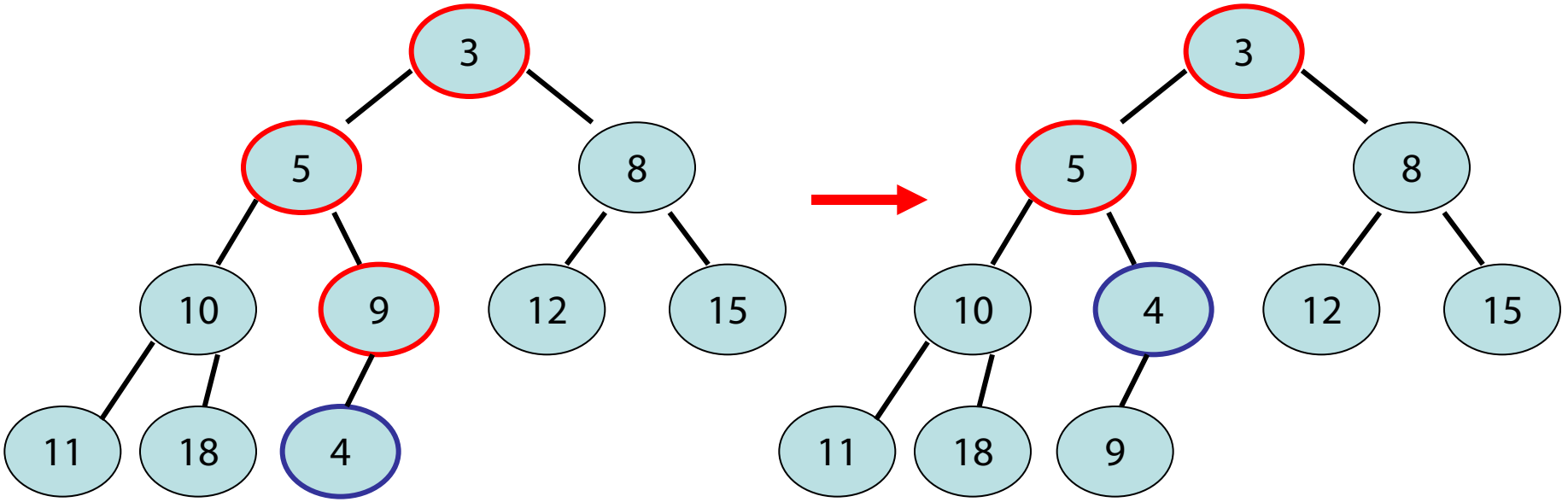
Insert - Binärer Heap



Invariante: H[k] ist minimal für Teilbaum von H[k]

: Knoten, die Invariante eventuell verletzen

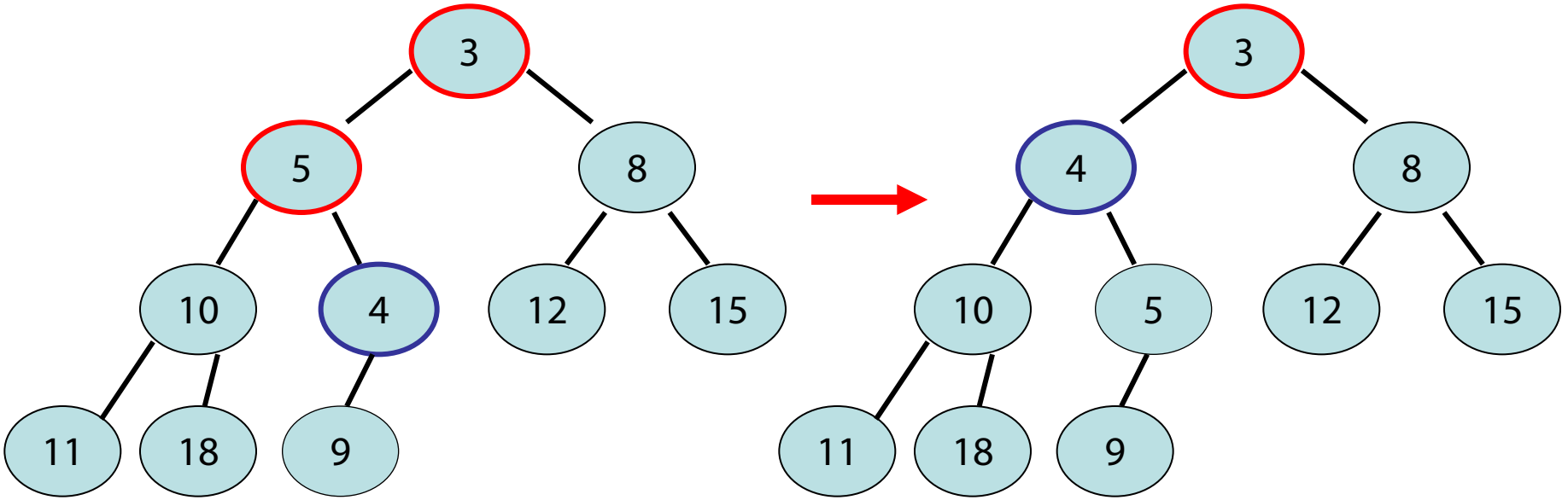
Insert Operation - Korrektheit



Invariante: H[k] ist minimal für Teilbaum von H[k]

: Knoten, die Invariante eventuell verletzen

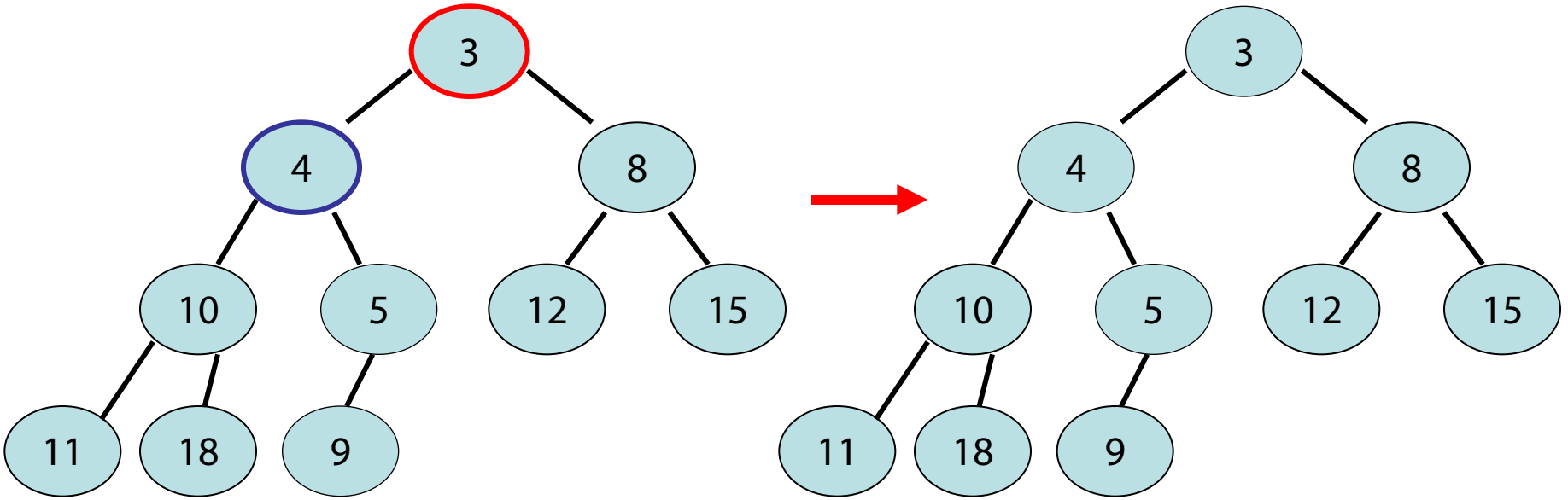
Insert Operation - Korrektheit



Invariante: H[k] ist minimal für Teilbaum von H[k]

: Knoten, die Invariante eventuell verletzen

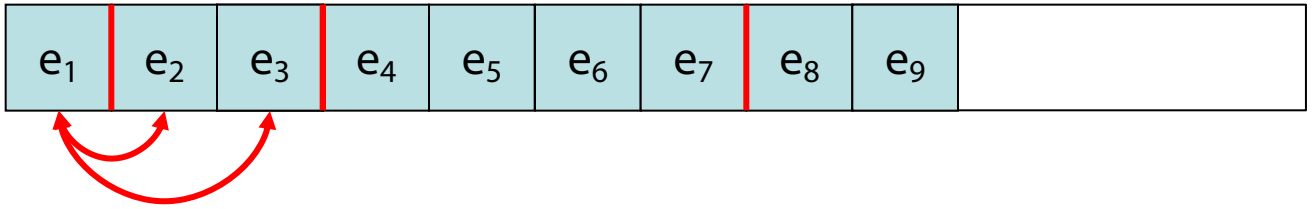
Insert Operation - Korrektheit



Invariante: H[k] ist minimal für Teilbaum von H[k]

: Knoten, die Invariante eventuell verletzen

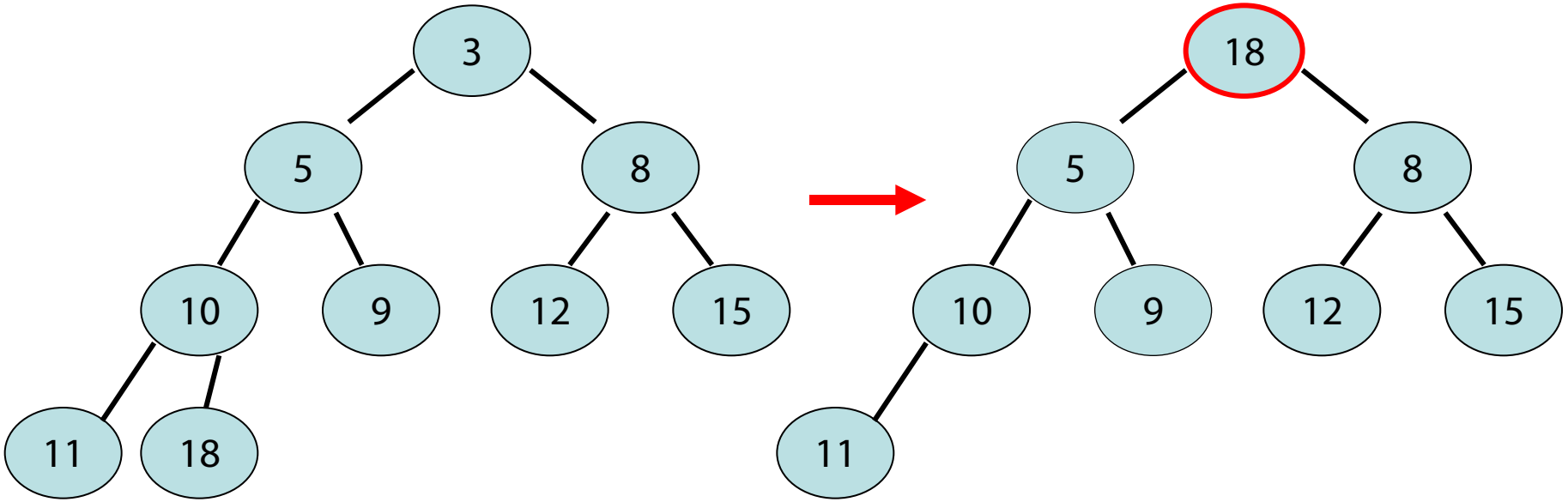
Delete Min: Binärer Heap



delete_min(pq):

- Form-Invariante: H[1] = H[n]; n = n-1
- Heap-Invariante: starte mit Element e in H[1].
  Vertausche e mit Kind mit min Schlüssel bis
  H[k] ≤ min({ H[2k], H[2k+1] }) für Position k von e
  oder e in Blatt

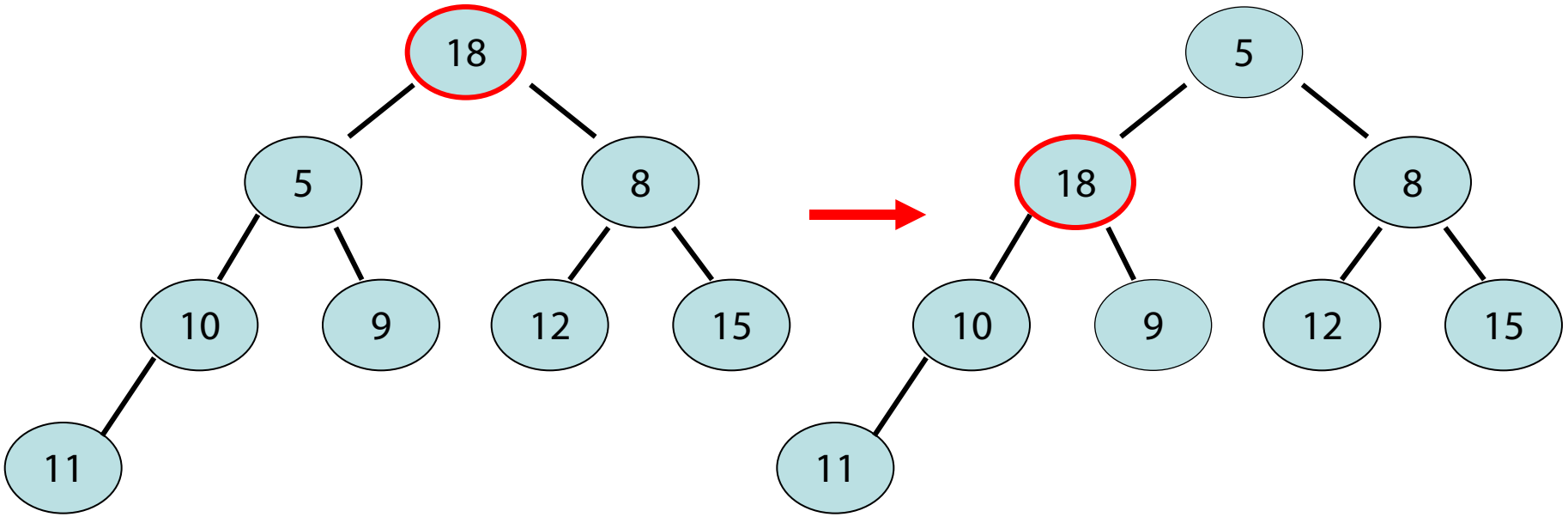
Delete Min Operation - Korrektheit



Invariante: H[k] ist minimal für Teilbaum von H[k]

: Knoten, die Invariante eventuell verletzen

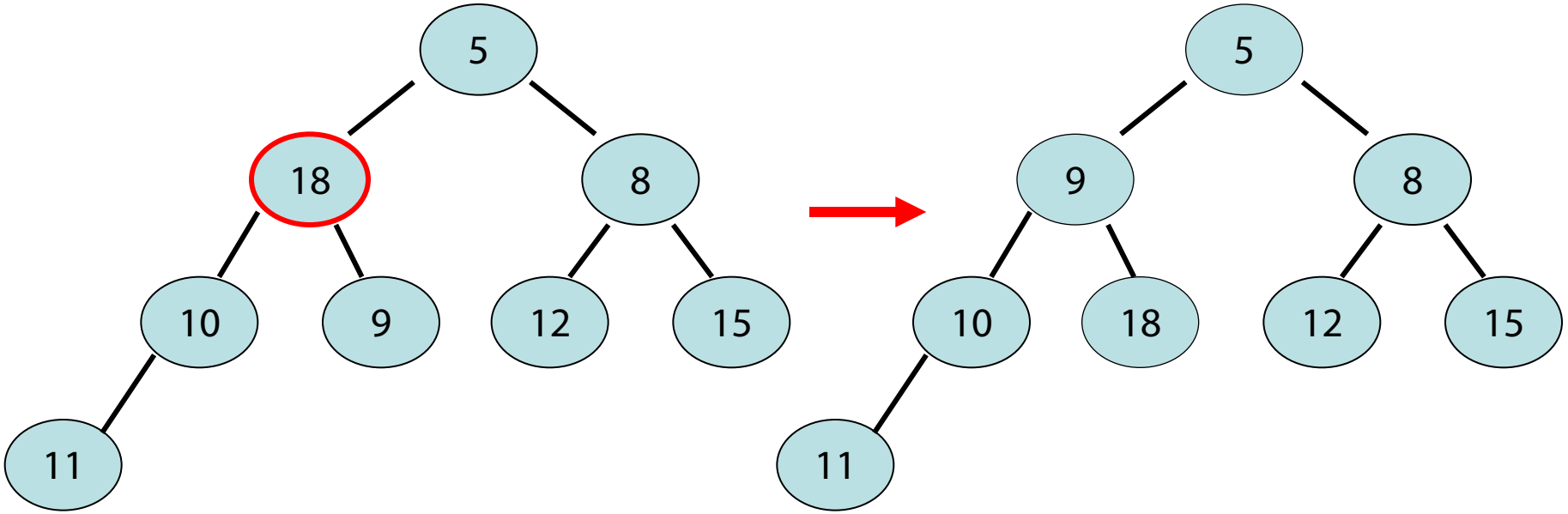
Delete Min Operation - Korrektheit



Invariante: H[k] ist minimal für Teilbaum von H[k]

: Knoten, die Invariante eventuell verletzen

Delete Min Operation - Korrektheit



Invariante: H[k] ist minimal für Teilbaum von H[k]

: Knoten, die Invariante eventuell verletzen

Binärer Heap

```
function delete_min(pq)
    H = pq.internal_repr; n = length(H); e = H[1]; H[1] = H[n]
    deleteat!(H, n)
    sift_down(1, pq)
    return e
end

function sift_down(i, pq)
    H = pq.internal_repr
    while !is_leaf(i, H)
        if !exists(right_child(i), H)
            m = left_child(i)
        else
            if pq.key(H[left_child(i)]) < pq.key(H[right_child(i)])
                m = left_child(i)
            else
                m = right_child(i)
            end
        end
        if pq.key(H[i]) <= pq.key(H[m]) return # Invariante erfuellt
        else temp = H[i]; H[i] = H[m]; H[m] = temp
        end
        i = m
    end
end
```

Laufzeit: O(log n)

Prioritätswarteschlange mit binärem Heap

Operator	Laufzeit
insert	O(log n)
min_element	O(1)
delete_min	O(log n)

```
struct BinaererHeapPQ
  internal_repr
  key
  set_key
end
```

Binärer Heap

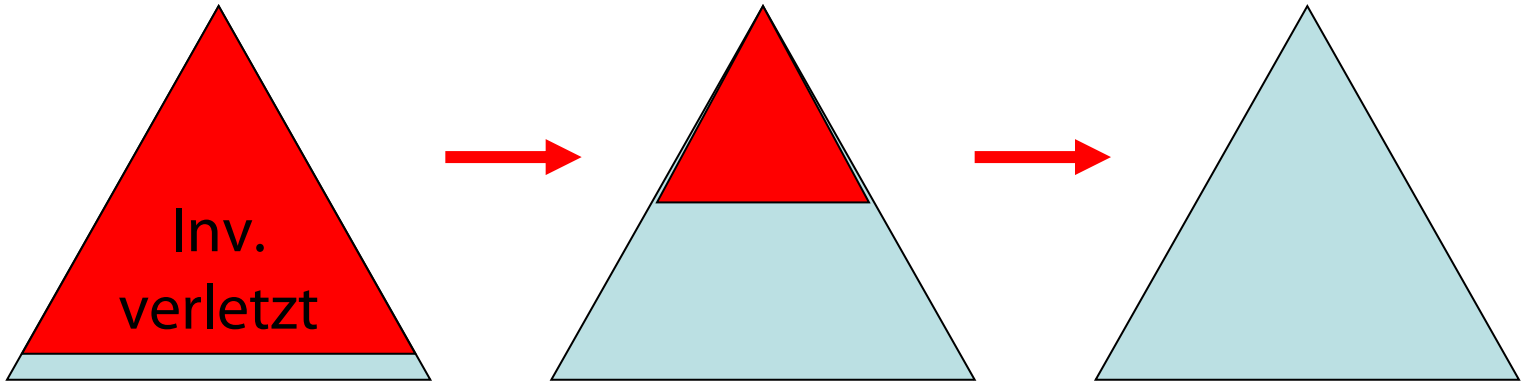
build(elements, key, set_key):

- Naive Implementierung:
  über n insert(e, pq)-Operationen. Laufzeit O(n log n)
- Bessere Implementierung:

```
function build(elements, key, set_key)
    pq = BinaryHeapPQ(elements, key, set_key)
    H = pq.internal_repr; n = length(H);
    for i = parent(n):-1:1
        sift_down(i, pq)
    end
    return pq
end
```

Binärer Heap: Operation build

Setze H[i] = e_i für alle i. Rufe sift_down(i, pq) für i = parent(n) runter bis 1 auf.



Invariante: Für alle j>i: H[j] minimal für Teilbaum von H[j]

Aufwand? Sicher O(n log n), siehe vorige Überlegungen

Unnötig pessimistisch (besser gesagt: „asymptotisch nicht eng“)

Aufwand für build

$$\log_a x = \frac{\log_b x}{\log_b a} \qquad \log_4 16 = \frac{\log_2 16}{\log_2 4}$$

- Die Höhe des Baumes, in den eingesiebt wird, nimmt zwar von unten nach oben zu, ...
- ... aber für die meisten Knoten ist die Höhe „klein“ (die meisten Knoten sind unten)
- Ein n-elementiger Heap hat Höhe $\lfloor \log n \rfloor$...
- ... und maximal $\lceil n/2^{h+1} \rceil$ viele Knoten (Teilbäume) mit Höhe $h \in \{0,\ldots,\lfloor \log n \rfloor\}$
- siftDown, aufgerufen auf Ebene h, braucht h Schritte: O(h)
- Der Aufwand für build ist also

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

Robert W. Floyd: Algorithm 245: Treesort 3.
In: Communications of the ACM. 7, Nr. 12, S. 701, **1964**

Herleitung

- $O\left(n\sum_{h=0}^{\infty}\frac{h}{2^h}\right) = O\left(n\sum_{h=0}^{\infty}h\left(\frac{1}{2}\right)^h\right)$

Geometrische Reihe $\sum_{h=0}^{\infty}(x)^h = \frac{1}{1-x}$ für $x < 1$ // nach x abl.

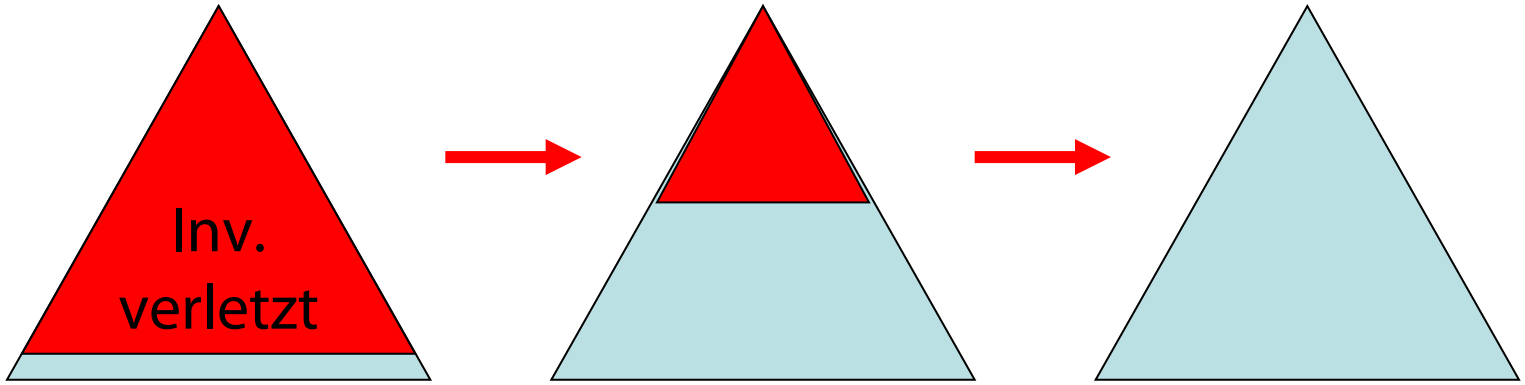
- $\sum_{h=0}^{\infty}hx^{h-1} = \frac{1}{(1-x)^2}$ // mal x nehmen
- $\sum_{h=0}^{\infty}hx^{h} = \frac{x}{(1-x)^2}$ // $x = \frac{1}{2}$

$$O\left(n\underbrace{\sum_{h=0}^{\infty}\frac{h}{2^h}}\right) = O(n)$$

konstant

Binärer Heap: Operation build

Setze H[i] = e_i für alle i. Rufe sift_down(i, pq) für i = parent(n) runter bis 1 auf.



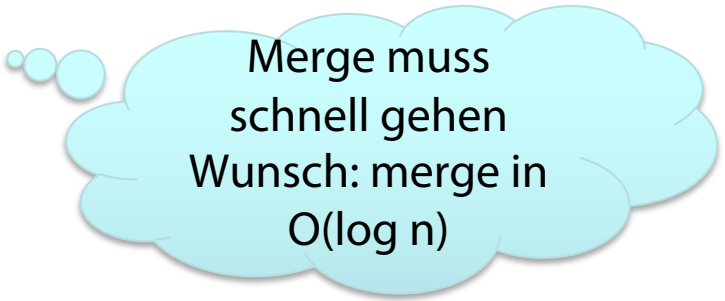
Invariante: Für alle j>i: H[j] minimal für Teilbaum von H[j]

Aufwand ist gekennzeichnet durch eine Funktion in O(n)

Anwendungen für merge

1. Lastumverteilung
   - Delegierung der Aufträge für einen Prozessor an einen anderen (evtl. neu hinzugeschalteten) Prozessor
2. Reduce-Operation

   (siehe MapReduce Programmiermodell)
   - Mischung von parallel ermittelten Ergebnissen, jeweils mit Bewertung bzw. Sortierung



Prioritätswarteschlangen als ADTs

- Unsortierte Liste?

Operator	Laufzeit
insert	O(1)
min_element	O(n)
delete_min	O(n)
build	O(n)

- Sortiertes Array?

Operator	Laufzeit
insert	O(n)
min_element	O(1)
delete_min	O(n)
build	O(n log n)

- Binärer Heap:

Operator	Laufzeit
insert	O(log n)
min_element	O(1)
delete_min	O(log n)
build	O(n)

Können wir Prioritätswartesschlangen auch schnell verschmelzen (merge)?

Quiz-Aufgabe – Vorbereitung

1. Bitte jetzt Handy oder Laptop rausholen bzw. Vordruck bereitlegen
2. https://moodle.uni-luebeck.de/mod/quiz/view.php?id=397132
3. Test mittels „Test jetzt durchführen“ bzw. „Test wiederholen“ öffnen
4. Fragen beantworten (nächste Folie)
5. **Anschließend „Versuch abschließen“ und 2x „Abgeben“ klicken**

Quiz-Aufgabe – Fragen

- **Frage 1**

  Welche Aussagen zu binären Heaps treffen zu? Binäre Heaps ...

  - ❑ **A** ... speichern Elemente intern in einem Feld.
  - ❑ **B** ... verwenden die Idee, einen Binärbaum als Feld zu realisieren.
  - ❑ **C** ... benutzen Zeiger (Pointer) um Kinder zu referenzieren.
  - ❑ **D** ... setzen die Operation `min_element` in O(log n) um.

- **Frage 2**

  Welche Aussagen zu Prioritätswarteschlangen (PQs) allgemein treffen zu?

  - ❑ **A** Die Operation `delete_min` soll schnell sein.
  - ❑ **B** PQs können effizient mit unsortierten Listen realisiert werden.
  - ❑ **C** PQs können zur Verwaltung von Prozessen in PCs genutzt werden.
  - ❑ **D** PQs müssen nie verschmolzen werden.

Binomial-Heap zum schnellen Verschmelzen

Binomial-Heap basiert auf sog. Binomial-Bäumen

Binomial-Baum muss erfüllen:

- Form-Invariante (r: Rang):

r=0 r=1 r → r+1

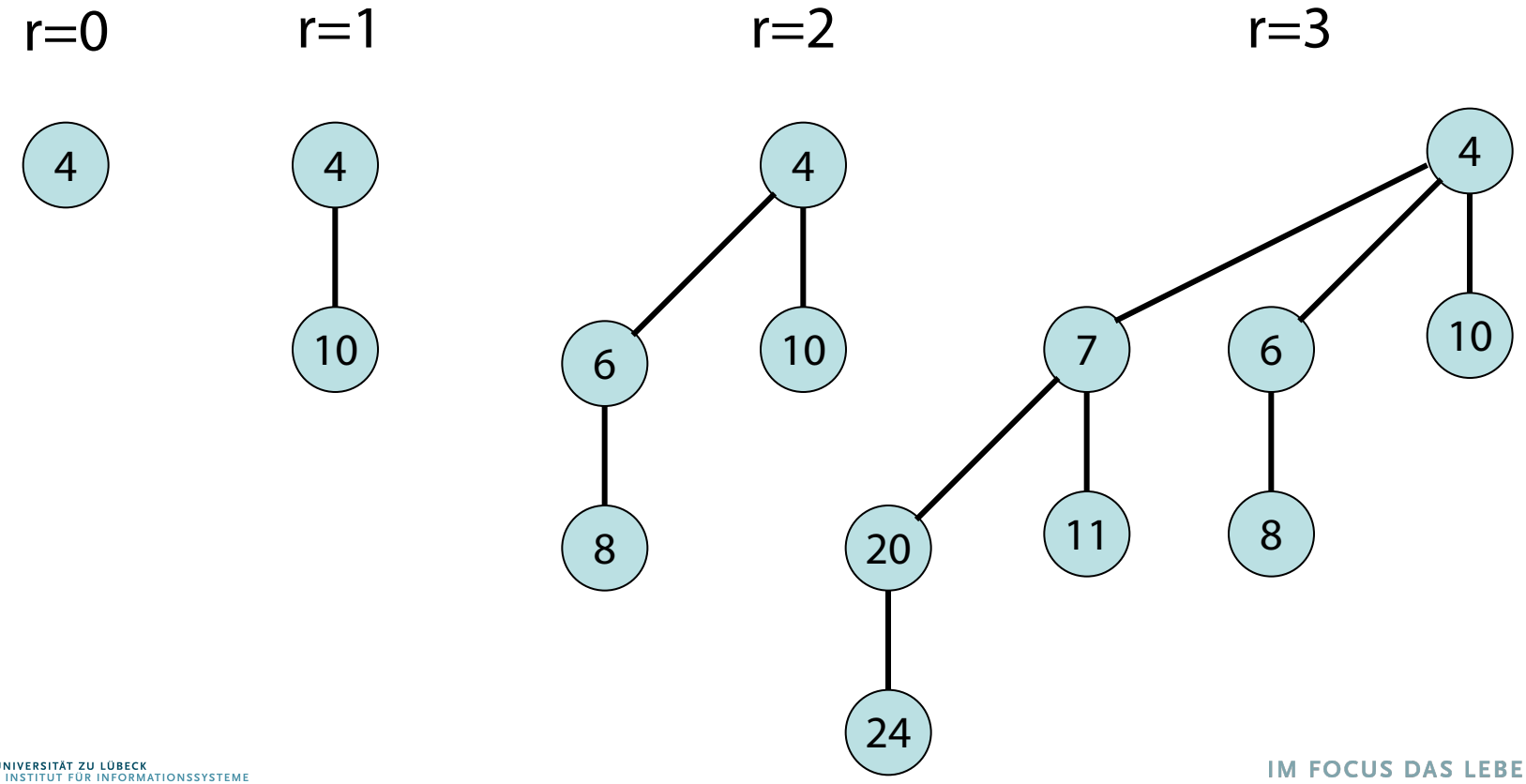
- MinHeap-Invariante: key(Vater) ≤ key(Kinder)

Jean Vuillemin: A data structure for manipulating priority queues. Communications of the ACM 21, S. 309–314, **1978**

Binomial-Heap

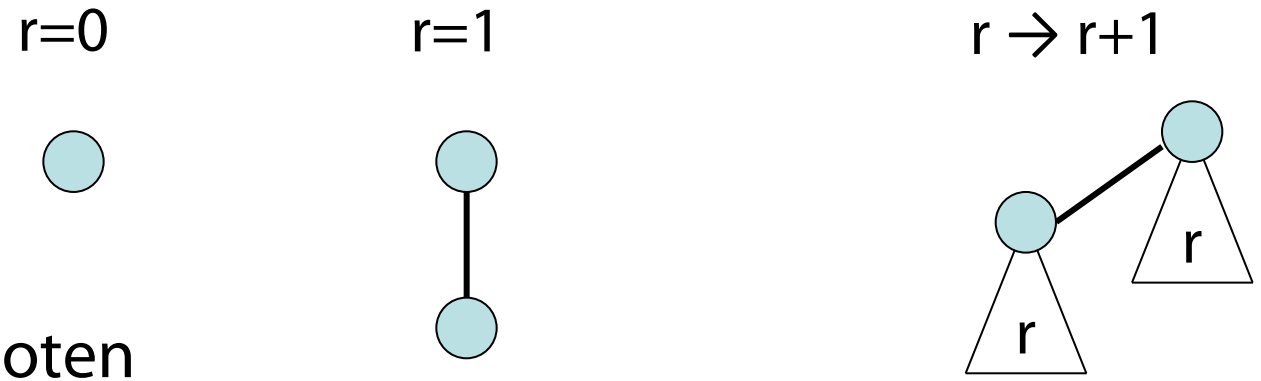
Beispiel für korrekte Binomial-Bäume:

Hier mit MinHeap-Eigenschaft



Binomial-Heap

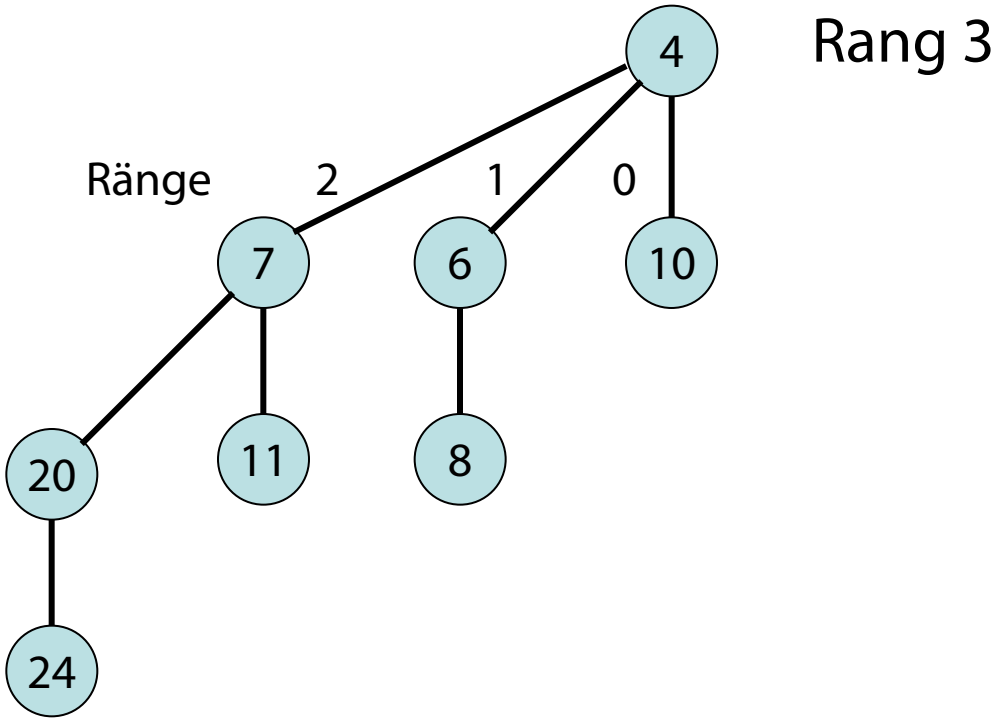
Eigenschaften von Binomial-Bäumen:



- 2^r Knoten
- maximaler Grad r (bei Wurzel)
- Wurzel weg: Zerfall in Binomial-Bäume mit Rang 0 bis r-1

Binomial-Heap

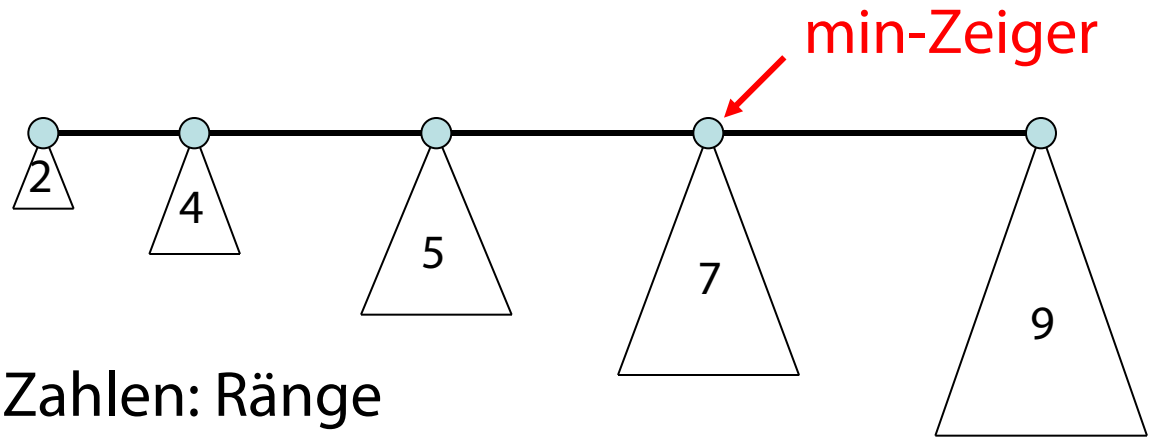
Beispiel für Zerfall in Binomial-Bäume mit Rang 0 bis r-1



Binomial-Heap

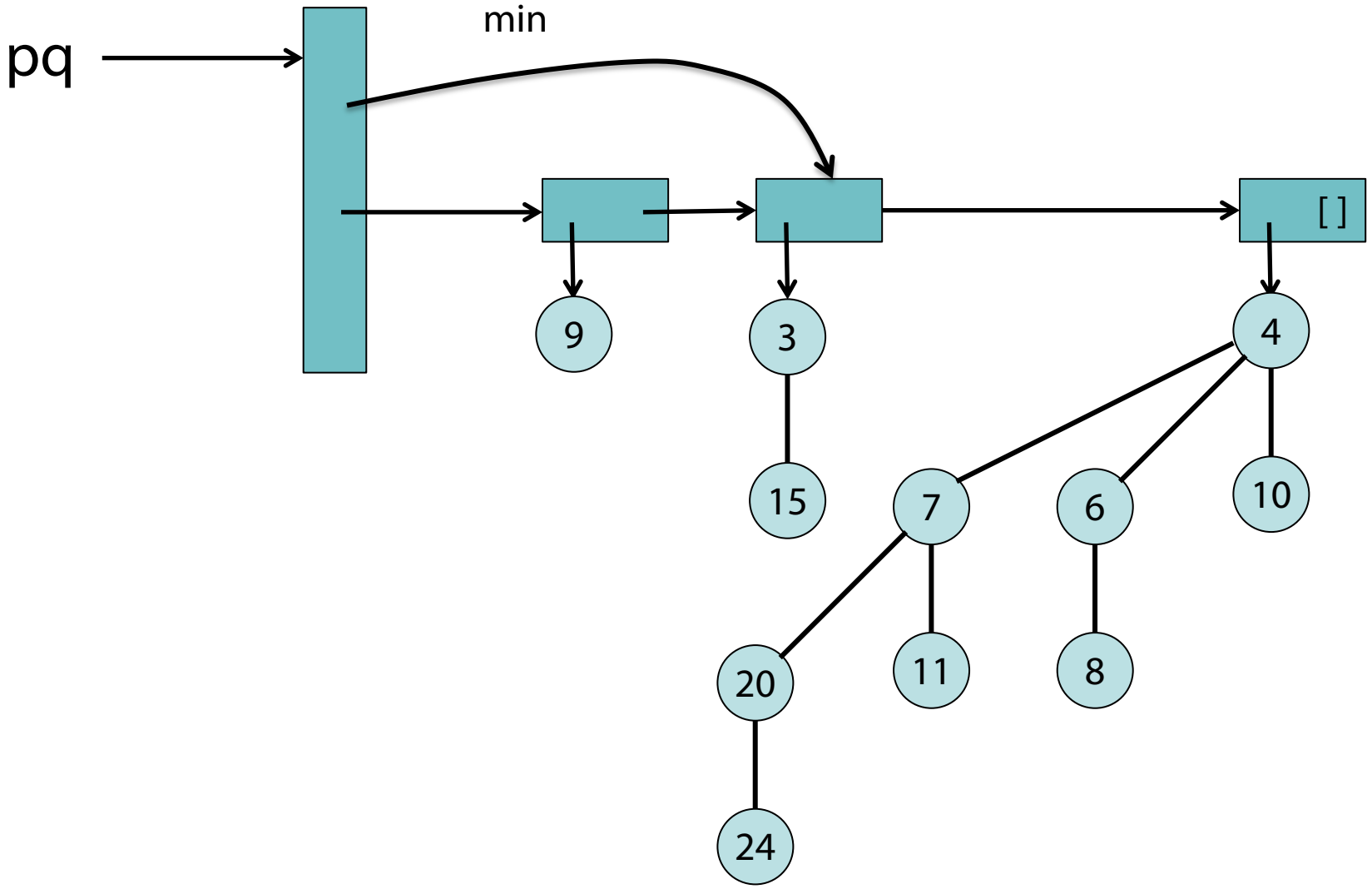
Binomial-Heap:

- verkettete Liste von Binomial-Bäumen
- Pro Rang maximal 1 Binomial-Baum
- Zeiger auf Wurzel mit minimalem key



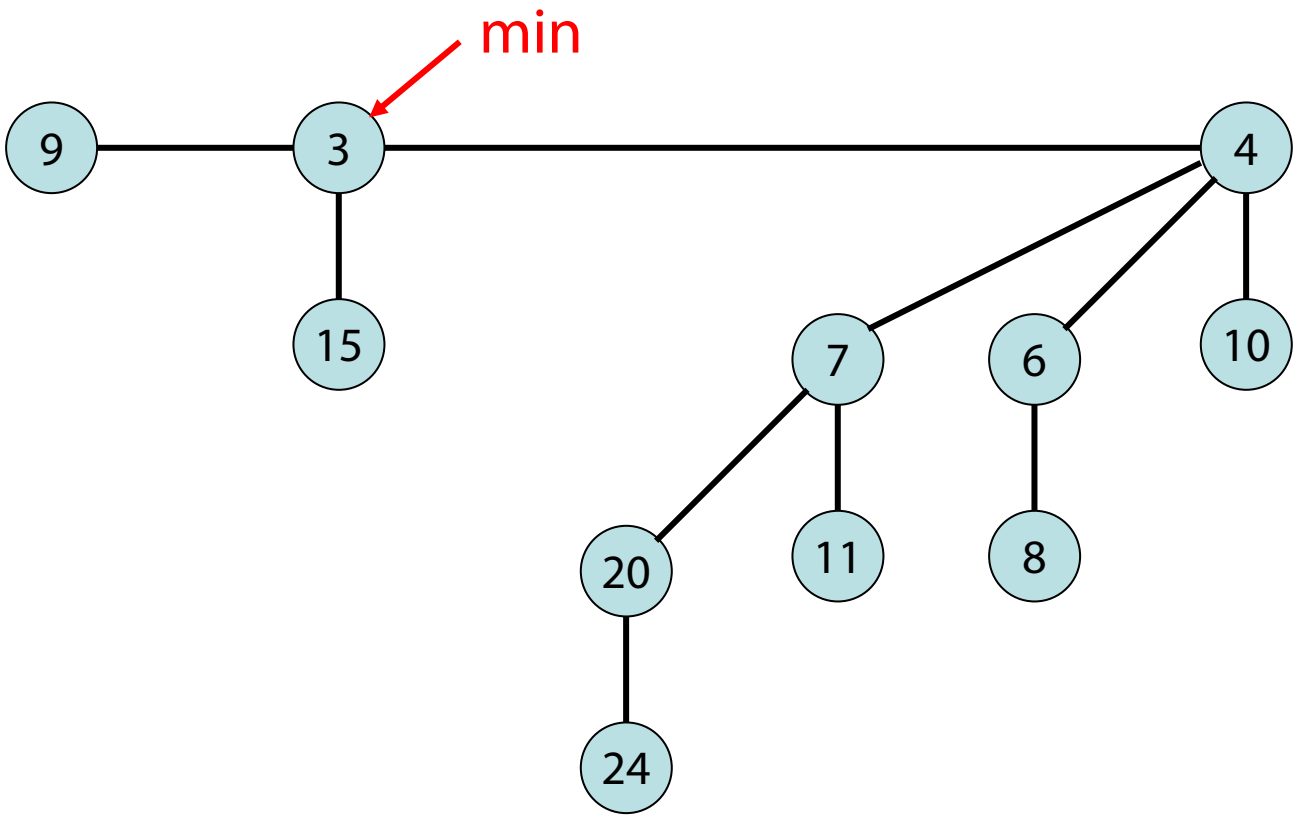
Zahlen: Ränge

Prioritätswarteschlangen als Binomial-Heaps



Binomial-Heap

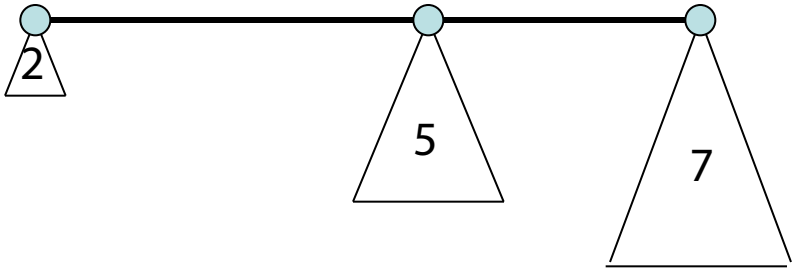
Abstrakte Darstellung:



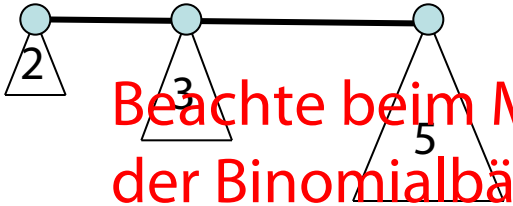
Anzahl der Bäume auf der Kette

- Binomial-Heap-Invariante:
  Pro Rang maximal 1 Binomial-Baum
- Was heißt das?
- Für n Knoten können höchstens log n viele Binomialbäume in der Kette vorkommen
  (dann müssen alle Knoten untergebracht sein)

Beispiel einer Merge-Operation



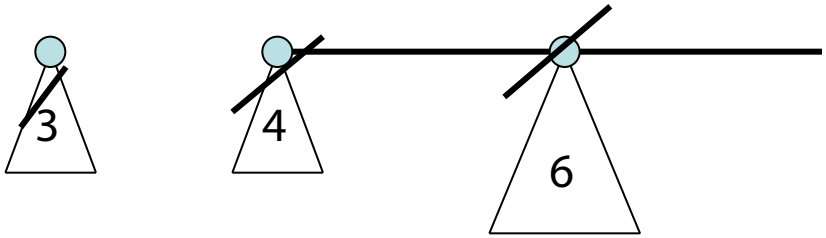
H_1



H_2

Beachte beim Mergen der Binomialbäume die Heap-Eigenschaft!

Zahlen geben die Ränge an



Ergebnis-Heap

Operationen auf Binomial-Heaps

Sei B_i: Binomial-Baum mit Rang i

- merge(pq, pq'): Aufwand für Merge-Operation: O(log n)
- insert(e, pq): Merge mit B_0, Zeit O(log n)
- min_element(pq): spezieller Zeiger, Zeit O(1)
- delete_min(pq): sei Minimum in Wurzel von B_i, Löschen von Minimum: $B_i \rightarrow B_0,\ldots,B_{i-1}$. Diese zurückmergen in Binomial-Heap. Zeit dafür O(log n).

Beispielimplementierung in Julia vorhanden.

Binomial-Heap

- decrease_key(e, pq, Δ): sift_up-Operation in Binomial-Baum von e und aktualisierte min-Zeiger. Zeit O(log n)

- delete(e, pq): (min-Zeiger zeigt nicht auf e) setze pq.set_key(e, -∞) und wende sift_up-Operation auf e an, bis e in der Wurzel; dann weiter wie bei delete_min. Zeit O(log n)

Beispielimplementierung in Julia vorhanden.

Zusammenfassung

Laufzeit	Binärer-Heap	Binomial-Heap
insert	O(log n)	O(log n)
min_element	O(1)	O(1)
delete_min	O(log n)	O(log n)
delete	O(log n)	O(log n)
decrease_key	O(log n)	O(log n)
merge	O(n)	O(log n)

Jean Vuillemin: A data structure for manipulating priority queues. Communications of the ACM 21, S. 309–314, **1978**

Zusammenfassung

Laufzeit	Binärer-Heap	Binomial-Heap
insert	O(log n)	O(log n)
min_element	O(1)	O(1)
delete_min	O(log n)	O(log n)
delete	O(log n)	O(log n)
decrease_key	O(log n)	O(log n)
merge	O(n)	O(log n)

Jean Vuillemin: A data structure for manipulating priority queues. Communications of the ACM 21, S. 309–314, **1978**

Datenstruktur Fibonacci-Heap

- Baut auf Binomial-Bäumen auf,
  aber erlaubt lazy merge und lazy delete.

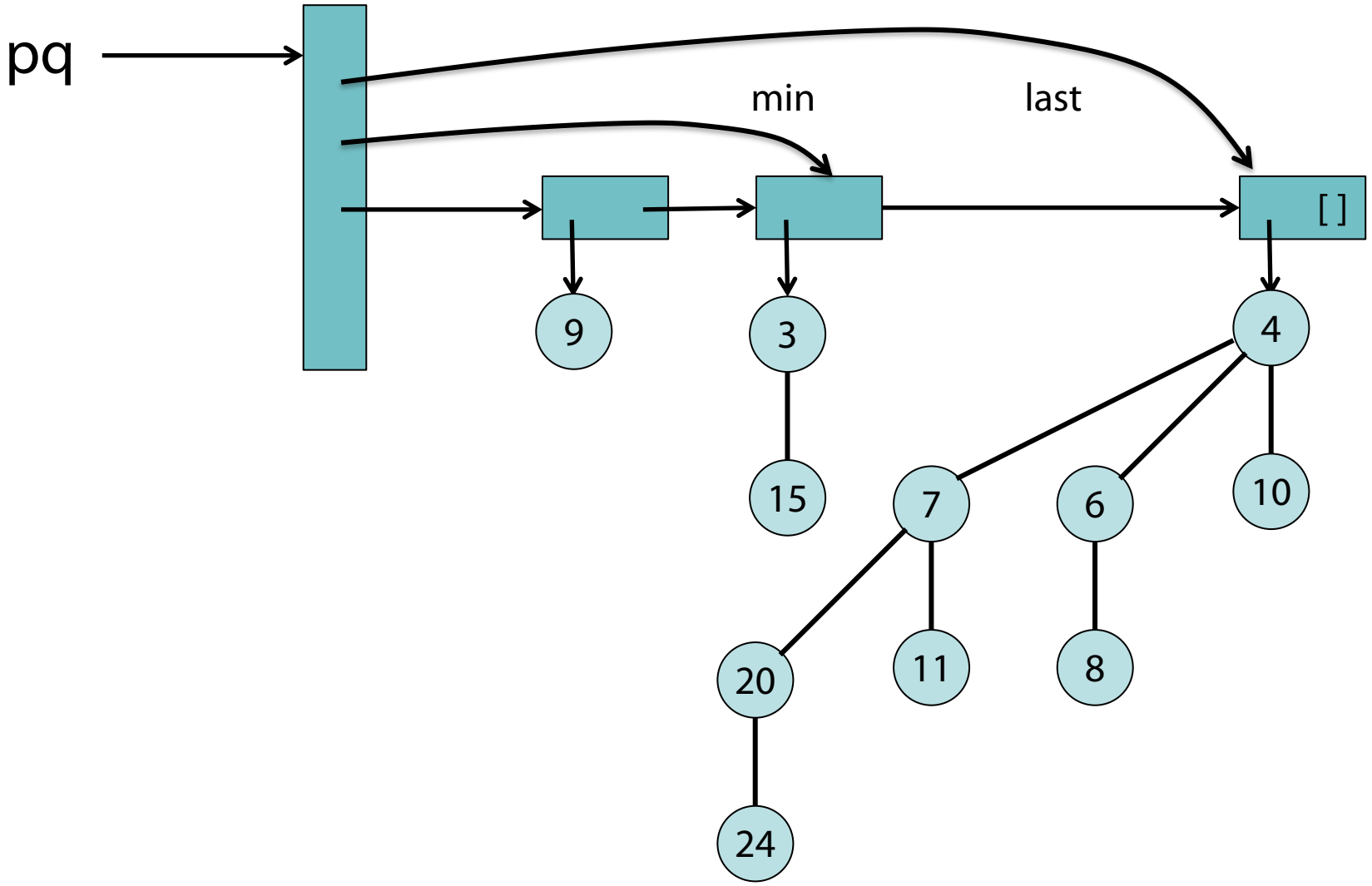
- Lazy merge: keine Verschmelzung von Binomial-Bäumen gleichen Ranges bei merge, nur Verkettung der Wurzellisten

- Lazy delete: erzeugt unvollständige Binomial-Bäume

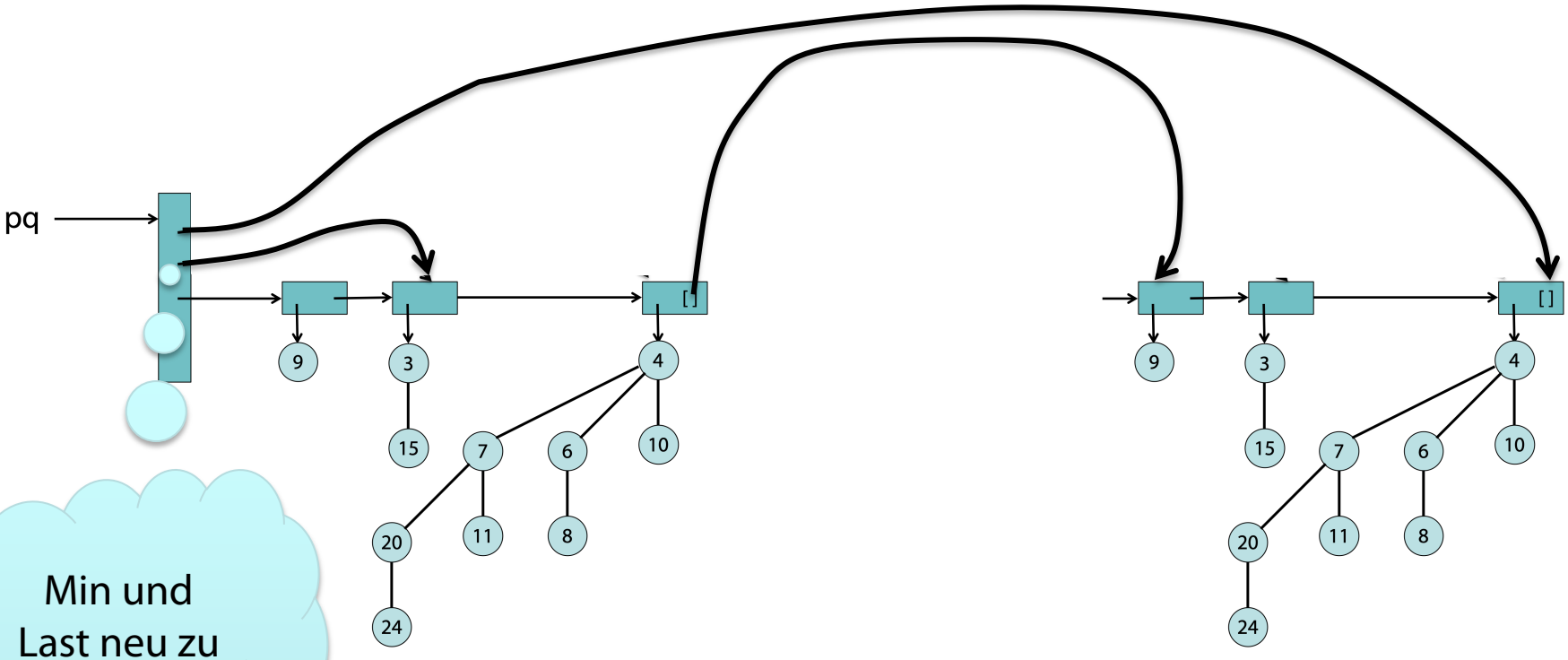
Der Name rührt von der Analyse der Datenstruktur her, bei der Fibonacci-Zahlen eine große Rolle spielen (wird nachher deutlich)

Michael L. Fredman, Robert E. Tarjan: Fibonacci heaps and their uses in improved network optimization algorithms. In: Journal of the ACM. 34, Nr. 3, S. 596–615, **1987**

Prioritätswarteschlangen als Binomial-Heaps



Verkettung der Liste zweier PQs

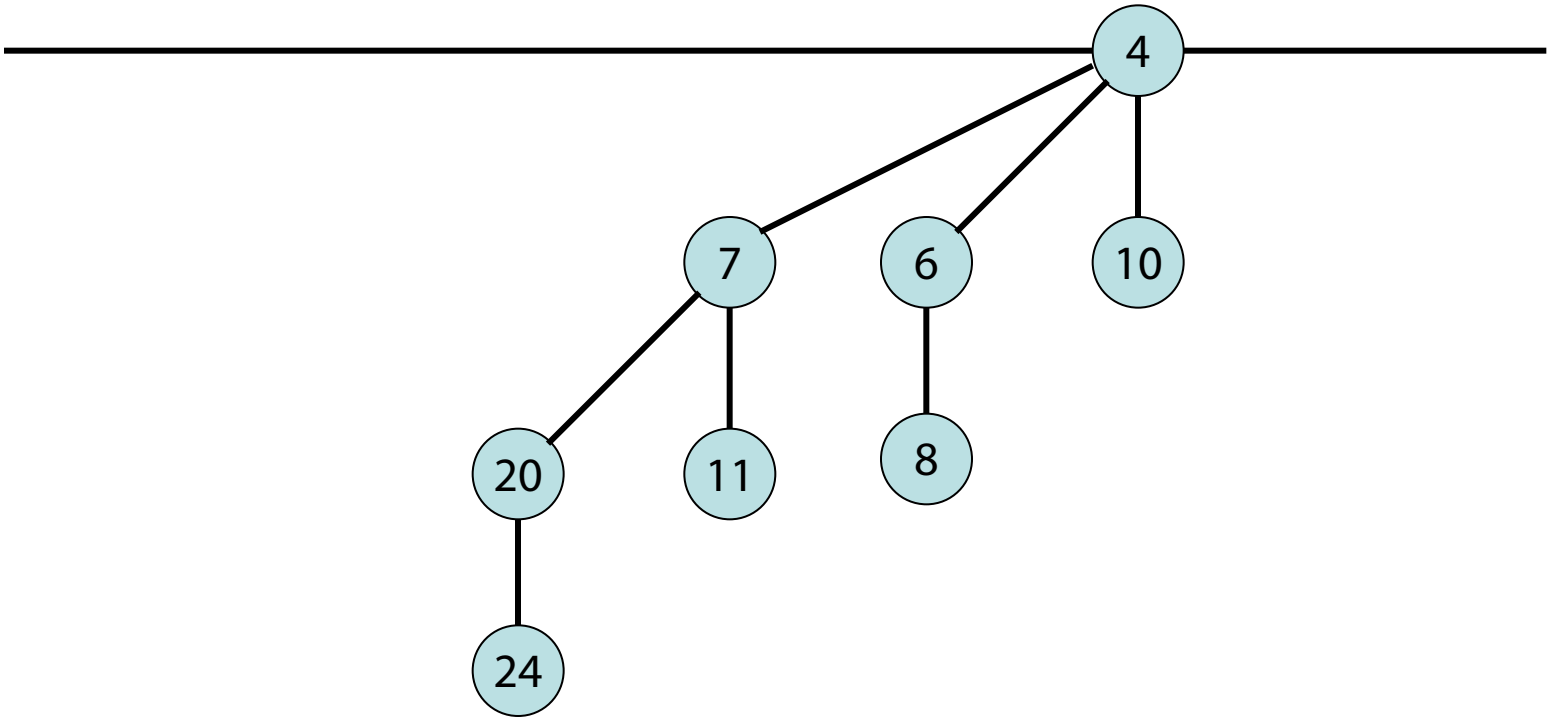


Min und Last neu zu bestimmen

Fibonacci-Heap

Baum in Binomial-Heap: Zentrale Invariante

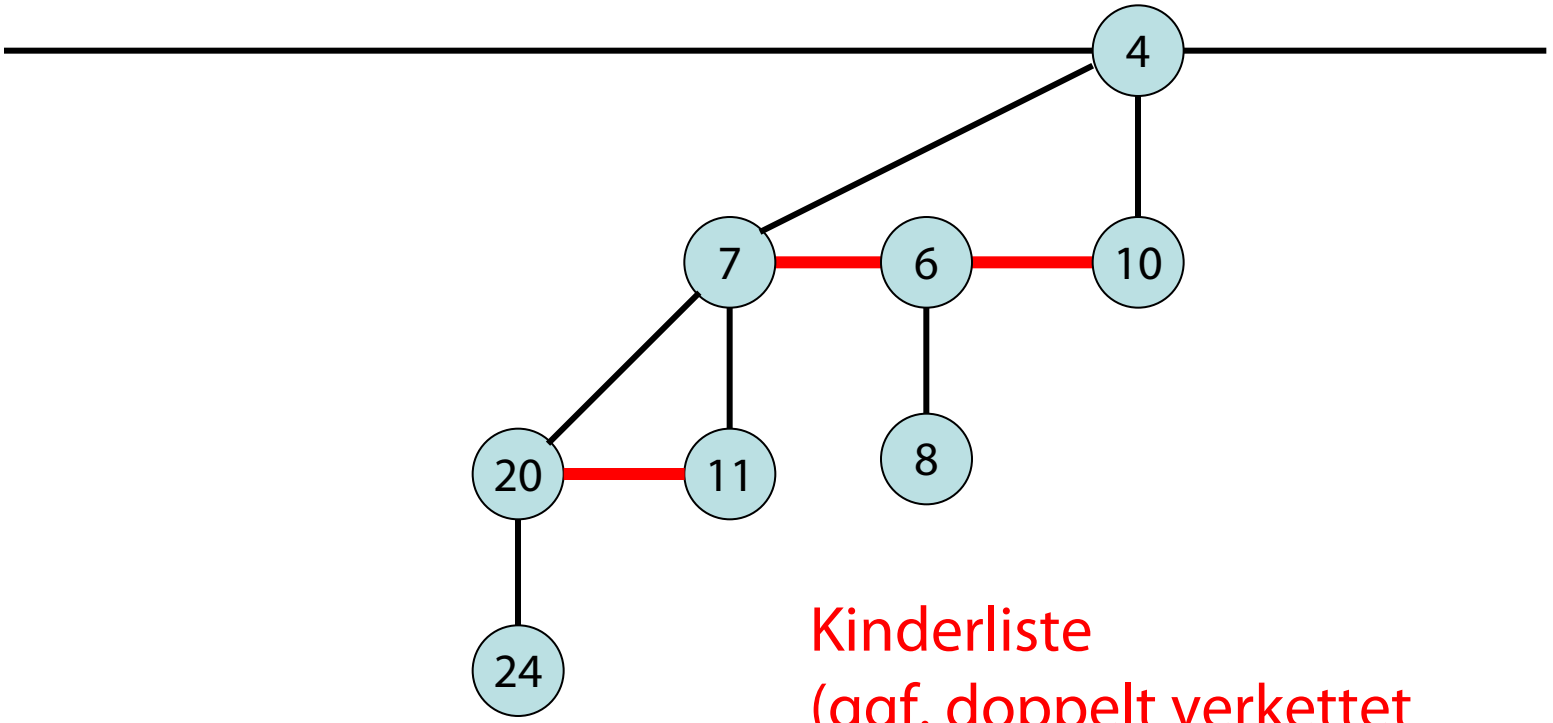
Rang = Anzahl der Kinder des Wurzelknotens



Fibonacci-Heap: Anpassung der Struktur

Baum in Fibonacci-Heap: Zentrale Invariante

Rang = Anzahl der Kinder des Wurzelknotens



Kinderliste
(ggf. doppelt verkettet
dito für Wurzelliste)

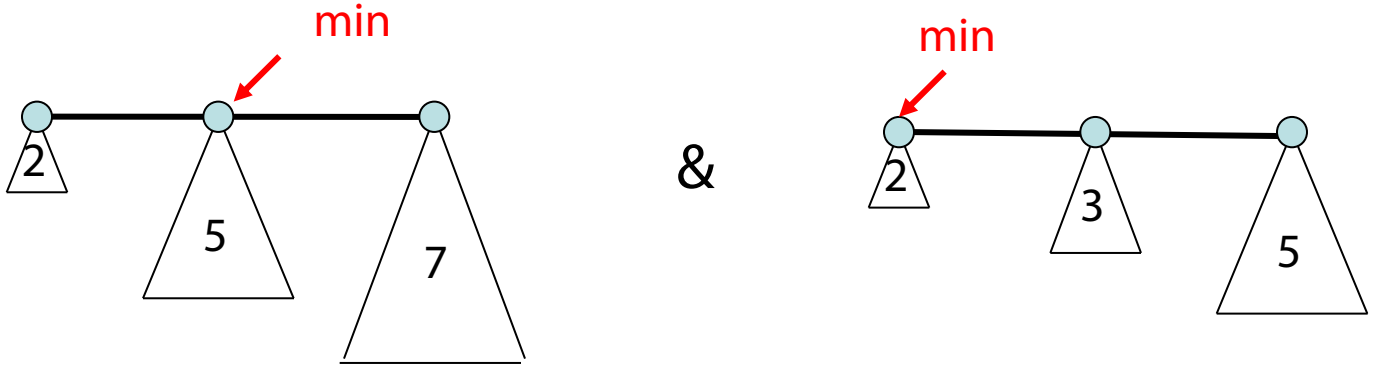
Fibonacci-Heap

Für jeden Knoten wird gespeichert:

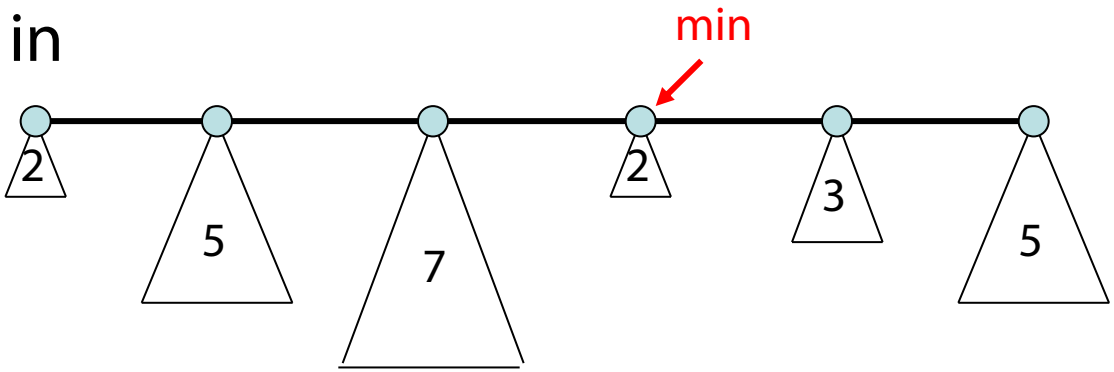
- im Knoten gespeichertes Element
- Vater
- Liste der Kinder (mit Start- und Endpunkt)
- Rang

Fibonacci-Heap: Lazy-Merge

Lazy merge von



resultiert in



Wurzellisten hintereinander-hängen: O(1)

Problem:

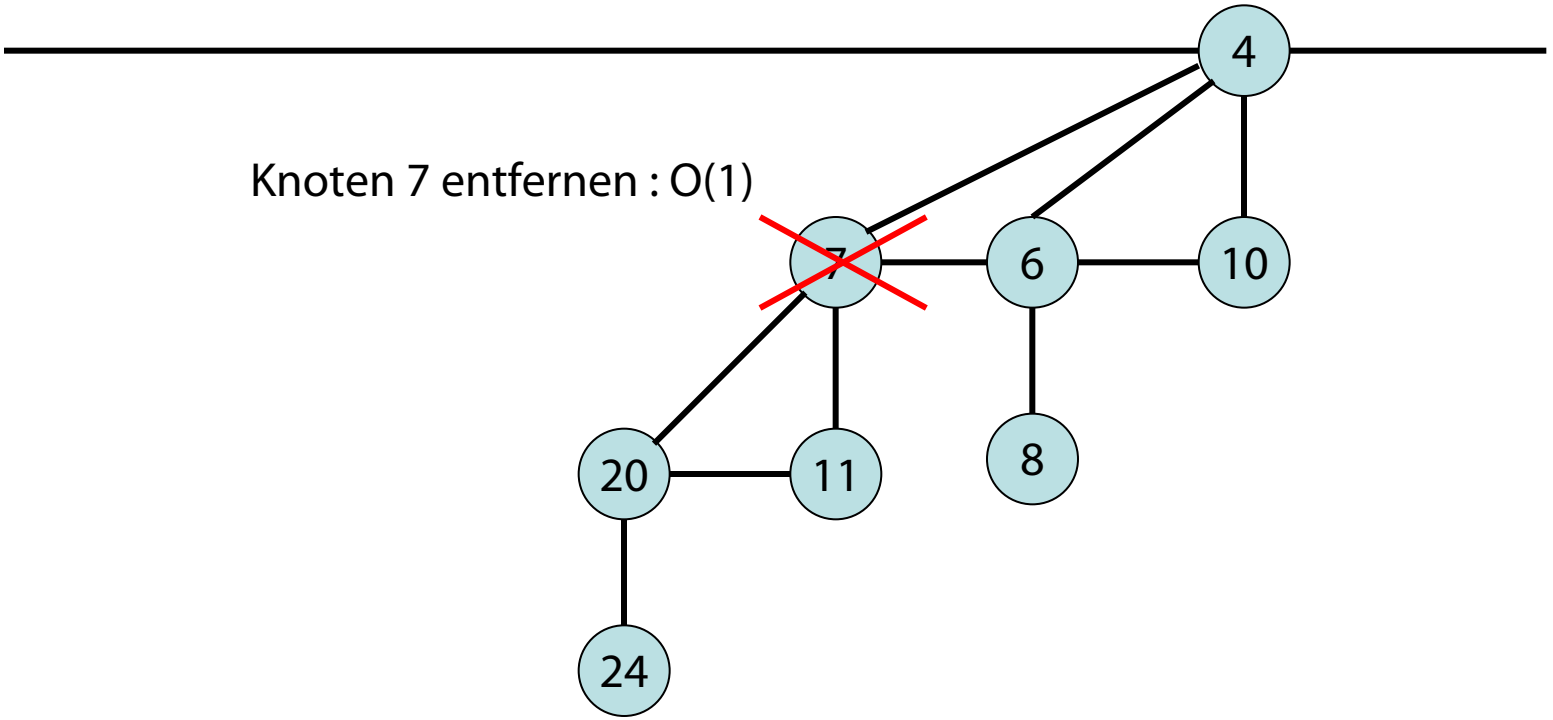
- Bäume gleichen Ranges treten in der Wurzelliste auf
- Binomial-Heap-Eigenschaft kann verletzt sein

Fibonacci-Heap: Lazy-Delete

Lazy delete:

Kinderliste in Wurzelliste integrieren: O(1)

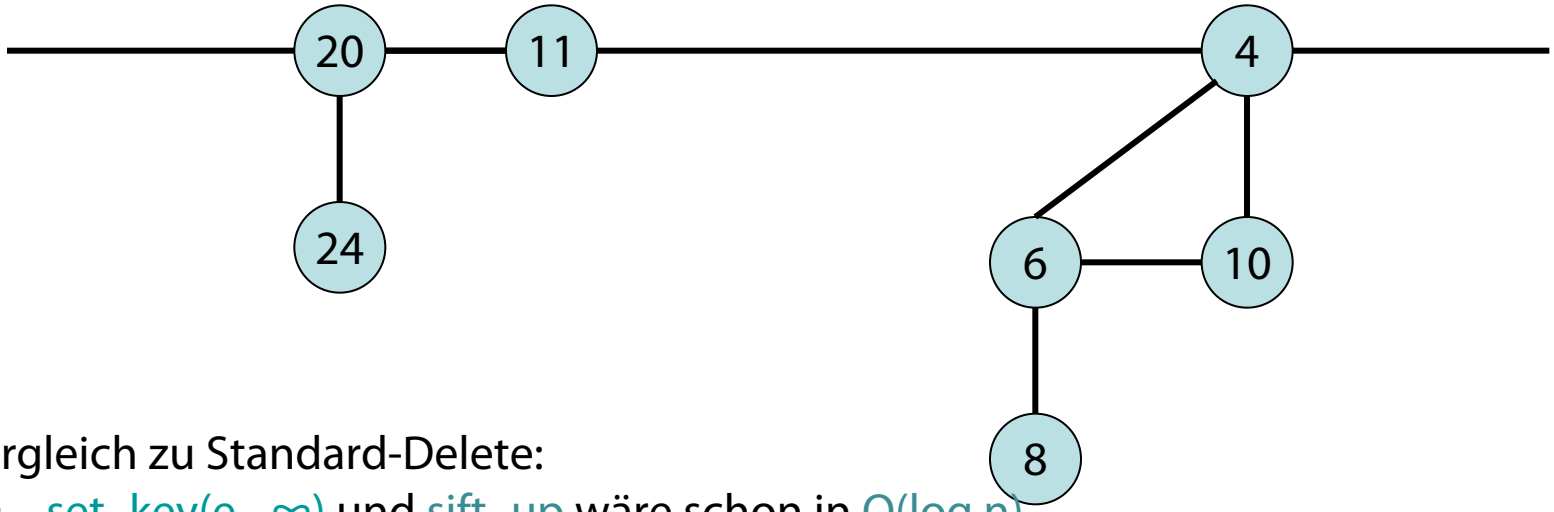
Knoten 7 entfernen : O(1)



Fibonacci-Heap: Lazy-Delete

Lazy delete:

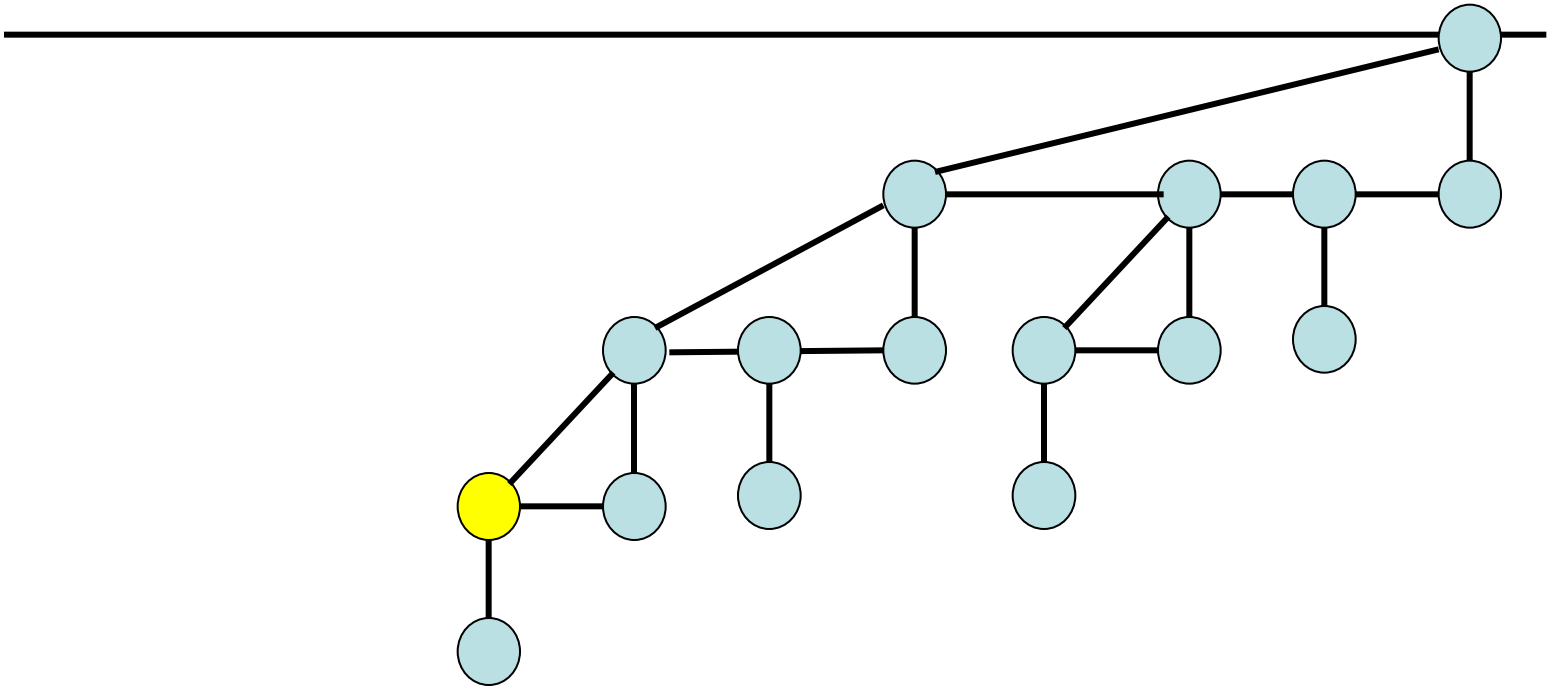
Annahme: Knoten 7 ist im direkten Zugriff
Aufwand: O(1), da gegebener Knoten 7 in O(1) Zeit entfernbar und Kinderliste von 7 in O(1) Zeit in Wurzelliste integrierbar



- Vergleich zu Standard-Delete:
  - set_key(e, -∞) und sift_up wäre schon in O(log n)
- Also kein sift_up
- Problem:
  - Bäume gleichen Ranges treten in der Wurzelliste auf
  - Binomial-Heap-Eigenschaft kann verletzt sein

Fibonacci-Heap

Beispiel für delete(e, pq)



Fibonacci-Heap: Übersicht

Operationen:

- merge(pq, pq'): Verbinde Wurzellisten, aktualisiere min-Pointer: Zeit O(1)
- insert(e, pq): Füge B_0 (mit e) in Wurzelliste ein, aktualisiere min-Pointer. Zeit O(1)
- min_element(pq): Gib Element, auf das der min-Pointer zeigt, zurück. Zeit O(1)
- delete_min(pq), delete(e, pq), decrease_key(e, pq, Δ): noch zu bestimmen…

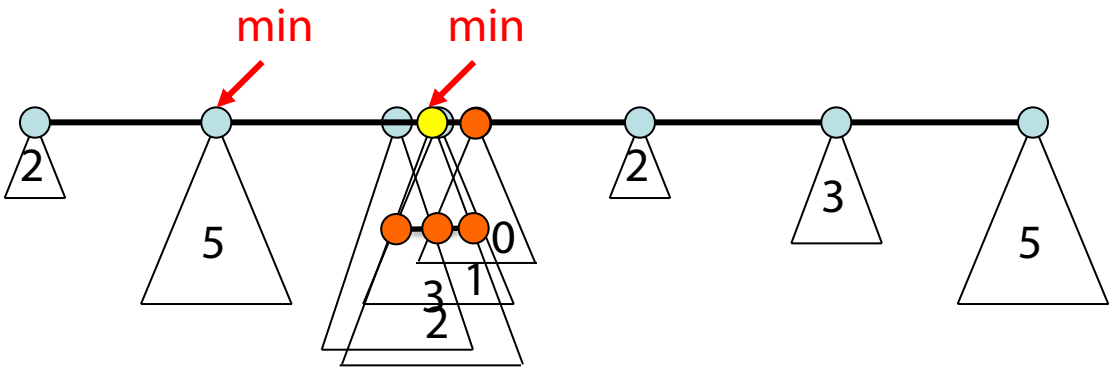
Fibonacci-Heap

delete_min(pq): Diese Operation hat Aufräumfunktion.

```
function delete_min(pq)
    node = min_node(pq)
    # entferne node aus Wurzelliste und konkateniere Kinderliste
      von node mit Wurzelliste
    delete(node, pq.roots); add_children_to_roots(node, pq)
    # merge zwei Bäume mit gleichem Rang i zu Baum mit Rang i+1,
    # solange es entsprechende Bäume gibt (wie bei zwei Binomial-
      Bäumen)
    full_merge(pq)
    # aktualisiere den min-Pointer
    update_min(pq)
end
```

- Durch die Integration der Kinderliste in die Wurzelliste können dort Bäume gleichen Ranges auftreten, die Struktur wird jedoch danach konsolidiert
- Die schon durch delete auftretenden Heaps gleichen Ranges werden gleich mit behandelt!

Beispiel: delete_min

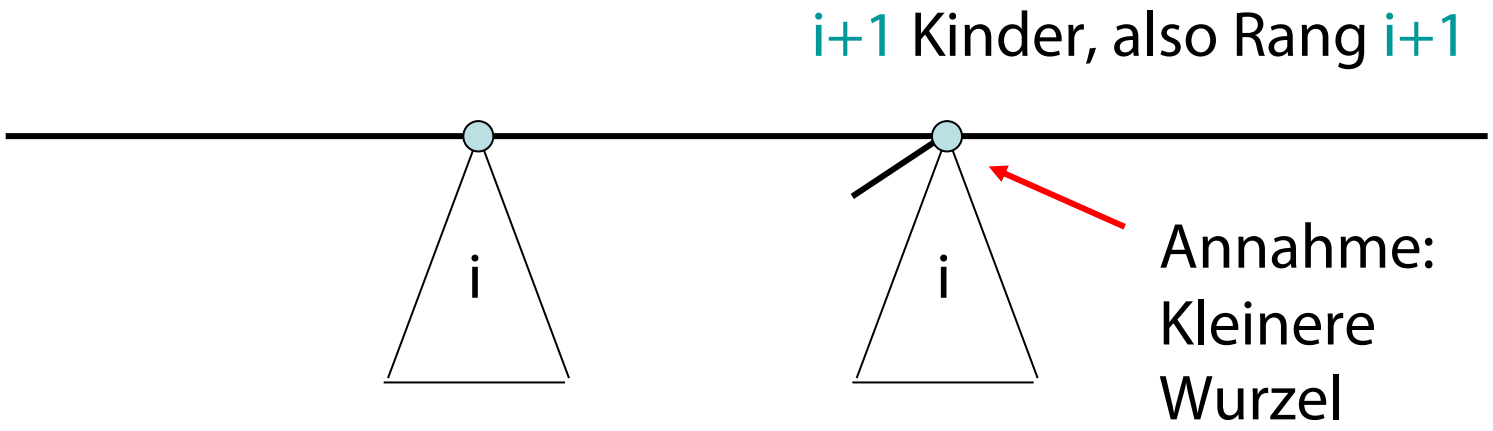


vorher

Binomial-Heap-Eigenschaft gilt auch

Verschmelzung zweier Bäume mit Rang i

(d.h. Wurzelknoten haben i Kinder):

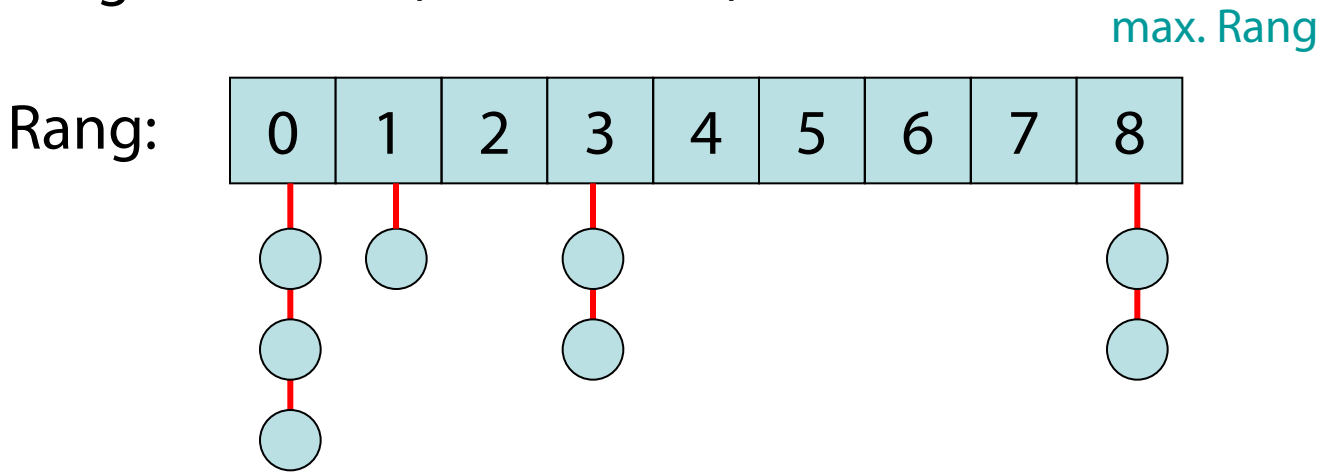


- Problem: Kinder mit gleichem Rang stehen nicht mehr nebeneinander

Fibonacci-Heap

Effiziente Findung von Wurzeln mit gleichem Rang:

- Scanne vor while-Schleife alle Wurzeln und speichere diese nach Rängen in Feld (Eimerkette!):



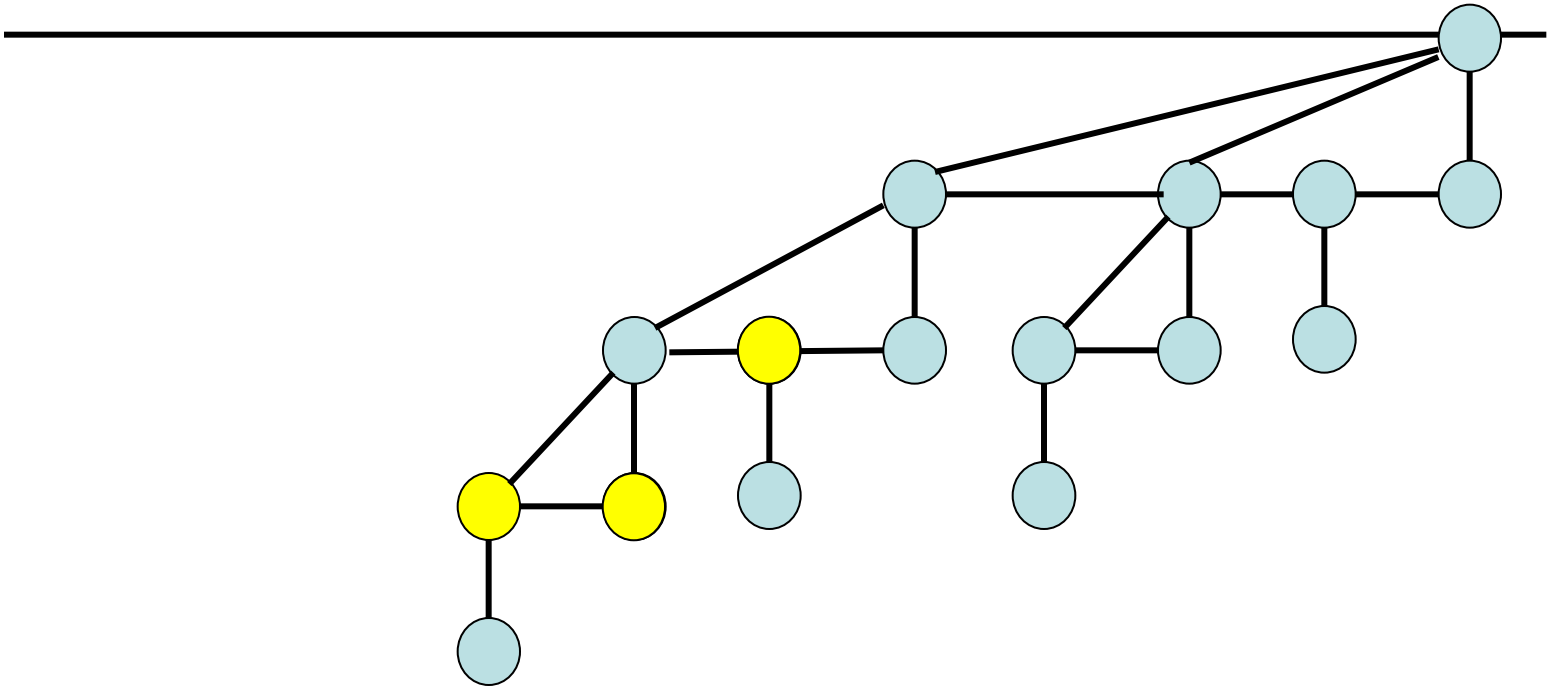
- Merge dann wie bei Binomialbäumen von Rang 0 an bis maximaler Rang erreicht

Operation delete

- Konsolidierung im Inneren der Heaps
- Ziel O(log n)

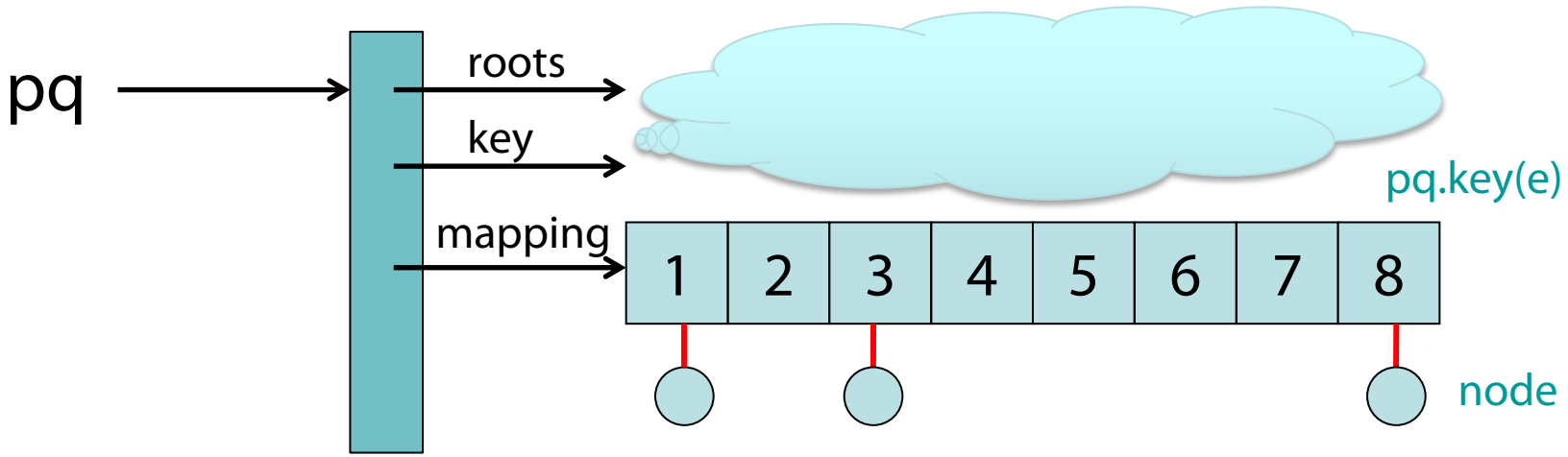
Fibonacci-Heap

Beispiel für delete(e, pq): (: is_marked = true)



Zuordnung Elemente zu Knoten – Idee

- Funktion delete(e, pq) bekommt Element e aus der Anwendung
- Intern müssen wir e auf Baumknoten node abbilden
- Zuordung zwischen Element und Knoten?
- Zusätzliches Feld mapping: pq.mapping[pq.key(e)] = node
  in FibonacciHeapPQ



Zuordnung Elemente zu Knoten – Julia

```
function delete(e, pq)
    node = lookup(pq.key(e), pq.mapping)    # pq.mapping[pq.key(e)]
    delete_mapping(pq.key(e), pq.mapping)   # Entferne e aus mapping

    # Knoten aus PQ loeschen
    delete_node(node, pq)
end
```

- Element e wird in Feld pq.mapping nachgeschlagen und an Aufruf delete_node(node, pq) weitergegeben
- Elemente müssen bei insert, build, … in pq.mapping eingetragen werden
- Problem: Keine doppelten Prioritäten & große Felder
- *Bessere Lösung in Einheit 9*

Fibonacci-Heap

Sei parent(node) Vater von Knoten node. Die Eigenschaft is_marked speichert, ob ein Kind entfernt wurde.

```
function delete_node(node, pq) # lazy delete
    if min_node(pq) == node   delete_min(pq) # node ist min-Wurzel
    else # haenge node aus, füge Kinder von node in Wurzelliste ein
           (entmarkiere entsprechend)
        unhook(node, pq); add_children_to_roots(node, pq)
    end
    while !isnothing(parent(node)) && !isnothing(parent(parent(node)))
        # solange node nicht ungueltig oder Wurzel
        node = parent(node)
        if !is_marked(node)
            node.is_marked = true; return
        else # markiert, also schon Kind weg
            # haenge node samt Unterbaum in Wurzelliste
            unhook(node, pq)
            node.is_marked = false # Wurzeln benoetigen kein Mark
            node.parent = nothing
            insert(node, pq.roots)
        end
    end
end
```

Fibonacci-Heap: decrease_key

```
function decrease_key(e, pq, delta)
    node = lookup(pq.key(e), pq.mapping)    # pq.mapping[pq.key(e)]
    delete_mapping(pq.key(e), pq.mapping)   # Entferne e aus mapping

    # Knoten aus PQ in der Priorität ändern
    decrease_key_node(node, pq, delta)
end
```

Fibonacci-Heap: decrease_key_node

```
function decrease_key_node(node, pq, delta)
    node.is_marked = false; unhook(node, pq); insert(node, pq.roots)
    pq.set_key(node.value, pq.key(node.value) - delta)
    if pq.key(min_element(pq)) > pq.key(node.value)
        pq.min = node
    end
    node_orig = node
    while !isnothing(parent(node)) && !isnothing(parent(parent(node)))
        # solange node nicht ungueltig oder Wurzel
        node = parent(node)
        if !is_marked(node)
            node.is_marked = true; break
        else # markiert, also schon Kind weg
            # haenge node samt Unterbaum in Wurzelliste
            unhook(node, pq)
            node.is_marked = false # Wurzeln benoetigen kein Mark
            node.parent = nothing
            insert(node, pq.roots)
        end
    end
    node_orig.parent = nothing # Ursprünglicher node hat keinen Vater mehr
end
```

Fibonacci-Heap mit markierten Fehlern

Zeitaufwand:

- delete_min():
  O(1 + max. Rang + #Baumverschmelzungen)
  - Aushängen der Wurzel: O(1)
  - Eimerkette: max. Rang
  - Baumverschmelzungen selbst produzieren auch Aufwand (Anzahl von n abhängig?)

- delete(x), decrease_key(x, Δ):
  O(1 + #kaskadierende Schritte)
  d.h. #umgehängter markierter Knoten
  - (Anzahl von n abhängig?)

Strukturfehler: Verschiebung der Arbeit

- Statt bei jedem merge einen Aufwand von O(log n) zu leisten, ...
- ... wird die Arbeit bei einem delete_min mit übernommen, ...
- ... mit der Idee, dass man die entsprechenden Strukturen dort sowieso anfassen muss
- Das Umverteilen kann sich über eine längere Sequenz von Operationen durchaus amortisieren
- Nicht immer ist alles einfach intuitiv fassbar
  - Vgl. auch die diskutierten Effekte in build für binäre Heaps

Amortisierte Analyse

- S: Zustandsraum einer Datenstruktur
- F: beliebige Folge von Operationen $<Op_1, Op_2, Op_3, \ldots, Op_n>$
- s_0: Anfangszustand der Datenstruktur

$$s_0 \xrightarrow{Op_1} s_1 \xrightarrow{Op_2} s_2 \xrightarrow{Op_3} \ldots \xrightarrow{Op_n} s_n$$

- Zeitaufwand $T(F) = \sum_{i=1}^{n} T_{Op_i}(s_{i-1})$

Amortisierte Analyse

Zeitaufwand $T(F) = \Sigma_{i=1}^{n} T_{Op_i}(s_{i-1})$

Für den Zeitbedarf definieren wir eine Menge von Abschätzungsfunktionen $A_{Op}(s)$, eine pro Operation Op

Die Menge $\{ A_X(s) \mid X \in \{Op_1, Op_2, Op_3, \ldots, Op_n\}$ heißt Familie amortisierter Zeitschranken falls für jede Sequenz F von Operationen gilt

$$T(F) = \Sigma_{i=1}^{n} T_{Op_i}(s_{i-1}) \leq c + \Sigma_{i=1}^{n} A_{Op_i}(s_{i-1})$$

für eine Konstante c unabhängig von F

Amortisierte Analyse: Potentialmethode

Behauptung: Sei S der Zustandsraum einer Datenstruktur, sei s_0 der Anfangszustand und sei ϕ:S $\rightarrow R_{\geq 0}$ eine nichtnegative Funktion.

$\phi : S \rightarrow R_{\geq 0}$ wird auch Potential genannt.

Für eine Operation X und einen Zustand s mit $s \xrightarrow{X} s'$ definiere $A_X(s)$ über die Potentialdifferenz:

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

Dann sind die Funktionen $A_X(s)$ eine Familie amortisierter Zeitschranken.

Amortisierte Analyse: Potentialmethode

Zu zeigen: $T(F) \leq c + \Sigma_{i=1}^{n} A_{Op_i}(s_{i-1})$

Beweis:

$$\Sigma_{i=1}^{n} A_{Op_i}(s_{i-1}) = \Sigma_{i=1}^{n} [\phi(s_i) - \phi(s_{i-1}) + T_{Op_i}(s_{i-1})]$$

$$= T(F) + \Sigma_{i=1}^{n} [\phi(s_i) - \phi(s_{i-1})]$$

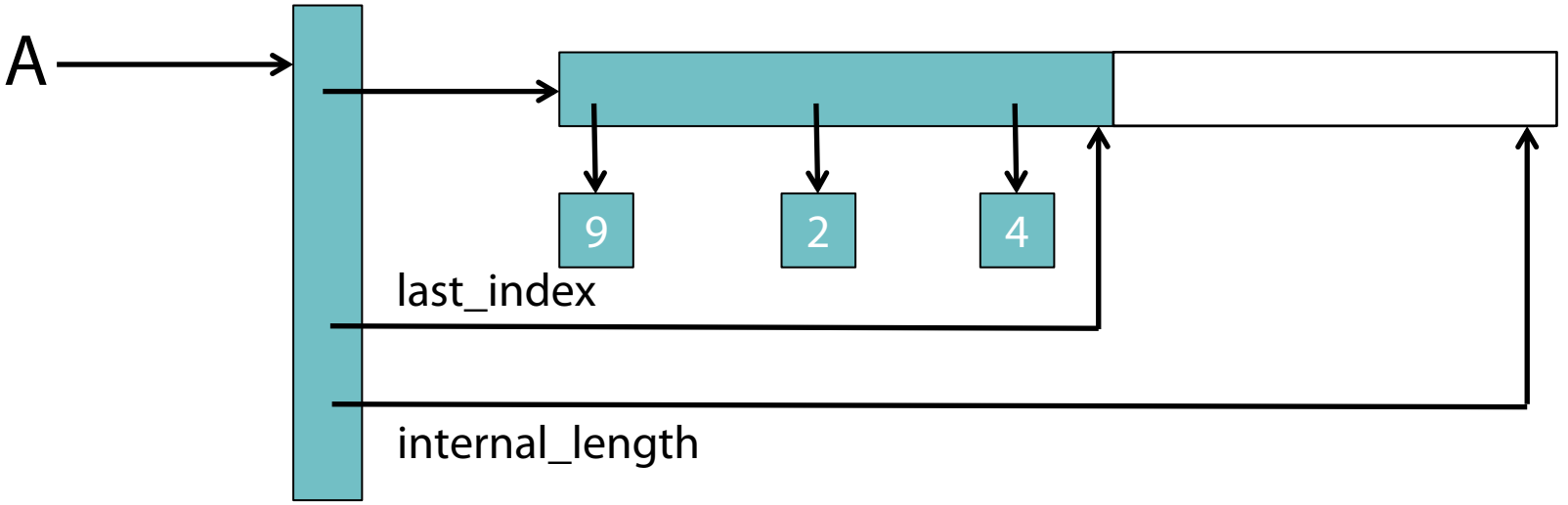
$$= T(F) + \phi(s_n) - \phi(s_0)$$

$$T(F) = \phi(s_0) - \phi(s_n) + \Sigma_{i=1}^{n} A_{Op_i}(s_{i-1})$$

$$\leq \phi(s_0) + \Sigma_{i=1}^{n} A_{Op_i}(s_{i-1})$$

konstant

Amortisierte Analyse: Arrays



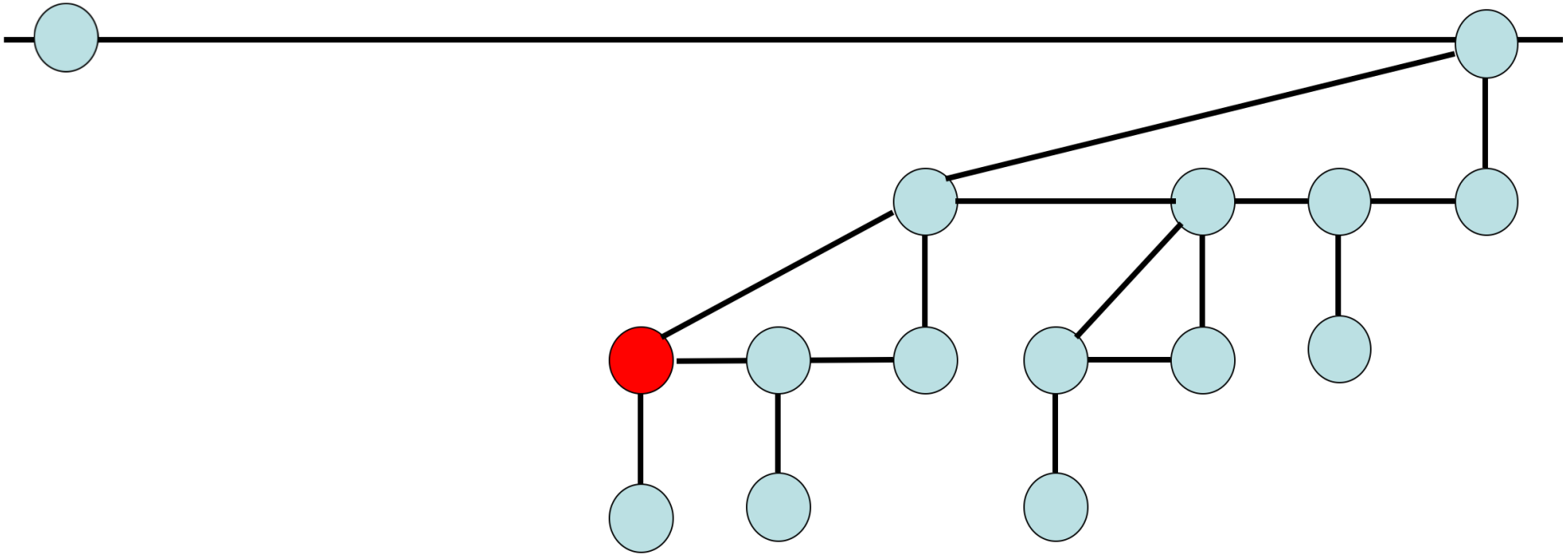
- Potential: max(0, A.last_index - A.internal_length / 2)
- Insert: Potentialerhöhung um 1 oder Erniedrigung auf 0 beim Verdoppeln der internen Länge

Muss die interne Repräsentation beim Vergrößern immer kopiert werden?

Amortisierte Analyse: Potentialmethode

Für Fibonacci-Heaps verwenden wir für das Potential den Begriff Balance (bal)

bal(s) = #Bäume + 2·#markierte Knoten im Zustand s



Fibonacci-Heap: insert, merge, min_element

- t_i: Zeit für Operation i ausgeführt im Zustand s
- a_i: amortisierter Aufwand für Operation i
  $a_i = t_i + \Delta bal_i$ mit $\Delta bal_i = bal(s')-bal(s)$, falls $i:s \rightarrow s'$ im aktuellen Zustand s ausgeführt wird

Amortisierte Kosten der Operationen:
$bal(s) = \#Bäume(s) + 2 \cdot \#markierte\ Knoten(s)$

- insert: $t=O(1)$ und $\Delta bal_{insert} = +1$, also $a_{insert}=O(1)$
- merge: $t=O(1)$ und $\Delta bal_{merge} = 0$, also $a_{merge}=O(1)$
- min: $t=O(1)$ und $\Delta bal_{min\_element} = 0$, also $a_{min\_element}=O(1)$
- delete_min: ???

Fibonacci-Heap

- Inv (noch zu zeigen):
- Ein Fibonacci-Heap aus n Elementen hat
  Bäume vom Rang maximal O(log n)
  (wie beim Binomial-Heap) auch, wenn durch delete einige Knoten entfernt wurden

Fibonacci-Heap: Eigenschaften

Invariante 1: Sei x ein Knoten im Fibonacci-Heap mit rang(x)=k. Seien die Kinder von x sortiert in der Reihenfolge ihres Anfügens an x. Dann ist der Rang des i-ten Kindes ≥i-2.

Beweis der Gültigkeit:

- Beim Einfügen des i-ten Kindes ist rang(x)=i-1.
- Das i-te Kind hat zu dieser Zeit auch Rang i-1.
- Danach verliert das i-te Kind höchstens eines seiner Kinder[1], d.h. sein Rang ist ≥i-2.

[1] Bei einem schon markierten Vater eines gelöschten Knotens wird konsolidiert

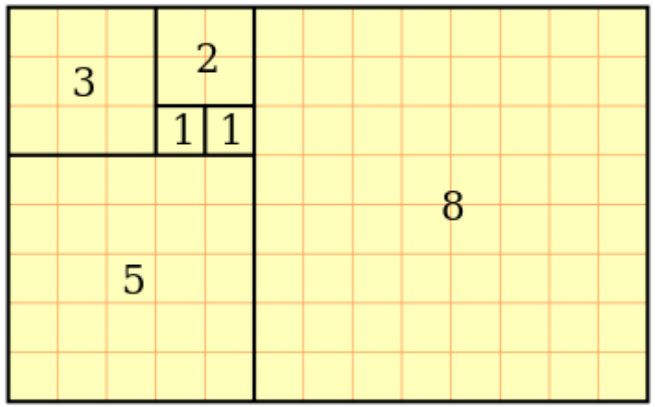
Fibonacci-Heap: Eigenschaften

Invariante 2: Sei x ein Knoten im Fibonacci-Heap mit rang(x)=k. Dann enthält der Baum mit Wurzel x mindestens F_{k+2} Elemente, wobei F_{k+2} die (k+2)-te Fibonacci-Zahl ist.

Einschub: Definition der Fibonacci-Zahlen:

- $F_0 = 0$ und $F_1 = 1$
- $F_i = F_{i-2} + F_{i-1}$ für alle $i > 1$

Daraus folgt, dass $F_{i+2} = 1 + \Sigma_{j=0}^{i} F_j$



[Wikipedia]

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, usw.

$F_{i+2} = 1 + \Sigma_{j=0}^{i} F_j$ $\quad\quad$ $F_i = F_{i-2} + F_{i-1}$

$$F_{i+2} = F_i + F_{i+1}$$
$$F_{i+2} = F_i + F_{i-1} + F_i$$
$$F_{i+2} = F_i + F_{i-1} + F_{i-2} + F_{i-1}$$
$$F_{i+2} = F_i + F_{i-1} + F_{i-2} + F_{i-3} + F_{i-2}$$

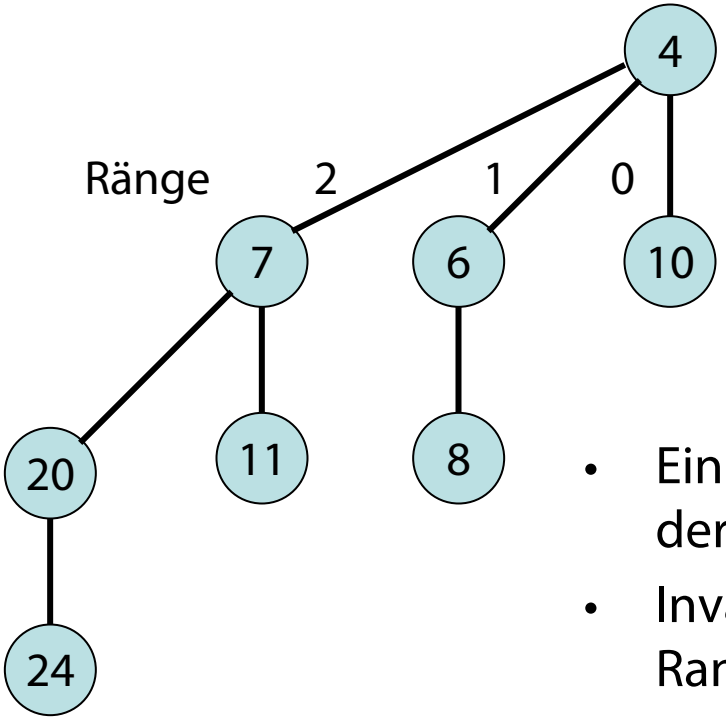
$F_0 = 0$ und $F_1 = 1$

....

$$F_{i+2} = F_i + F_{i-1} + F_{i-2} + F_{i-3} + F_{i-2} + \ldots\ldots\ldots + F_2 + F_1 + F_2$$
$$F_{i+2} = \underbrace{F_i + F_{i-1} + F_{i-2} + F_{i-3} + F_{i-2} + \ldots\ldots\ldots + F_2 + F_1 + F_0}_{\Sigma_{j=0}^{i} F_j} + \underbrace{F_1}_{1}$$

$$F_{i+2} = 1 + \Sigma_{j=0}^{i} F_j$$

Rang eines Knotens im Binomialbaum



Rang 3

- Ein Binomialbaum vom Rang k hat k Kinder, deren Ränge sind k-1, k-2, …. 1, 0
- Invariante 1:
  Rang des i-ten Kindes im Fib. Heap ≥i-2
- Sei f_k die Mindestanzahl von Knoten eines Baumes vom Rang k
- $f_{k-1} + f_{k-2} + f_{k-3} + \ldots + f_2 + f_1 + f_0 + 1 \leq f_k$

Mindestanzahlen Kinder Wurzel

Fibonacci-Heap

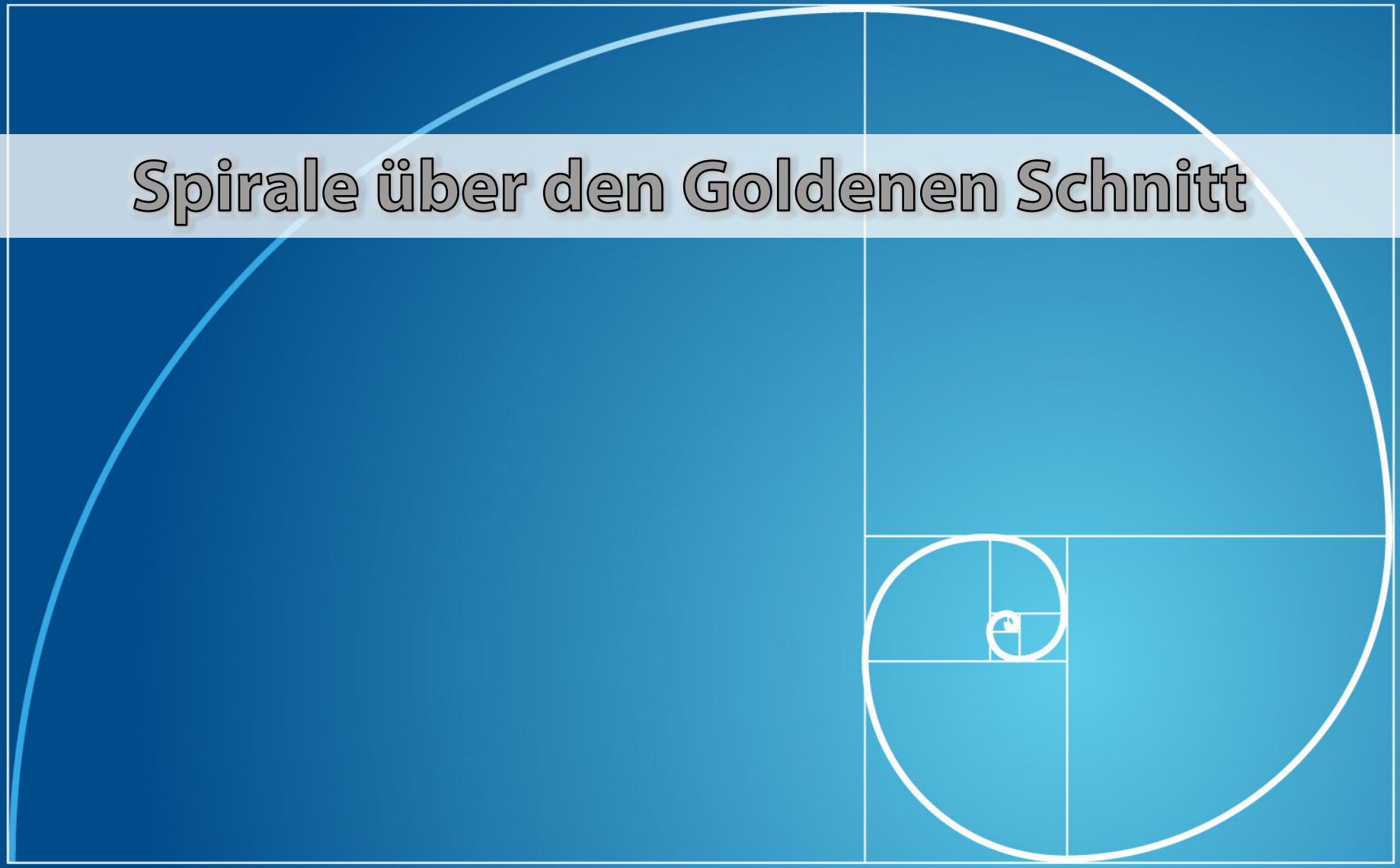
Argument für die Gültigkeit von Invariante 2:

- Sei f_k die Mindestanzahl von Knoten eines Baumes vom Rang k
- $f_k \geq f_{k-1} + f_{k-2} + f_{k-3} + \ldots + f_2 + f_1 + f_0 + 1$
- Weiterhin ist $f_0=1$ und $f_1=2$
- $f_k \geq f_{k-1} + f_{k-2} + f_{k-3} + \ldots + f_2 + 2 + 1 + 1$
- $f_k \geq f_{k-1} + f_{k-2} + f_{k-3} + \ldots + f_2 + 2 + 1 + 0 + 1$
- $\geq F_k + F_{k-1} + F_{k-2} + \ldots + F_3 + F_2 + F_1 + F_0 + 1$
- $= 1 + \Sigma_{j=0}^{k} F_j = F_{k+2}$
- Also folgt nach den Fibonacci-Zahlen:

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

Abschätzung durch Fibonacci-Zahlen

Spirale über den Goldenen Schnitt



$$\frac{a+b}{a} = \frac{a}{b} = \varphi \approx 1{,}61803$$

aus: [Wikipedia]

Verwandtschaft mit dem Goldenen Schnitt [Bearbeiten]

Wie von Johannes Kepler festgestellt wurde, nähert sich der Quotient zweier aufeinander folgender Fibonacci-Zahlen dem Goldenen Schnitt Φ an. Dies folgt unmittelbar aus der Näherungsformel für große *n*:

$$\lim_{n\to\infty} \frac{f_{n+1}}{f_n} = \lim_{n\to\infty} \frac{\Phi^{n+1}}{\Phi^n} = \Phi \approx 1{,}618\ldots$$

Diese Quotienten zweier aufeinander folgender Fibonacci-Zahlen haben eine bemerkenswerte Kettenbruchdarstellung

$$\frac{1}{1} = 1 \qquad \frac{2}{1} = 1+\frac{1}{1} \qquad \frac{3}{2} = 1+\frac{1}{1+\frac{1}{1}} \qquad \frac{5}{3} = 1+\frac{1}{1+\frac{1}{1+\frac{1}{1}}} \qquad \frac{8}{5} = 1+\frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1}}}}$$

Da diese Quotienten im Grenzwert gegen den goldenen Schnitt konvergieren, lässt sich dieser als der unendliche Kettenbruch

$$\Phi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cdots}}}}$$

darstellen.

Die Zahl Φ ist irrational. Das bedeutet, dass sie sich nicht durch ein Verhältnis zweier ganzer Zahlen darstellen lässt, ein Umstand, der wesentlich zu ihrer Bedeutung in Kunst und Natur beiträgt. Am besten lässt sich Φ durch Quotienten zweier aufeinander folgender Fibonacci-Zahlen darstellen. Dies gilt auch für verallgemeinerte Fibonaccifolgen, bei denen f_0 und f_1 beliebige natürliche Zahlen annehmen.

Fibonacci-Heap

- Man hat gezeigt, dass $F_k > \Phi^k$ ist für

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

- $f_k \geq F_{k+2} \geq \Phi^{k+2}$
- D.h. ein Baum mit Rang k im Fibonacci-Heap hat mindestens $1{,}61^{k+2}$ Knoten
- Sei $n \geq 1{,}61^{k+2} = 1{,}61^k * 1{,}61^2$ dann gilt: $n/1{,}61^2 \geq 1{,}61^k$ # n>1, also log n positiv
- Und damit: $k \leq \log_{1{,}61} n/1{,}61^2 = c + \log_{1{,}61} n$

- Sei Rang eine Funktion, die Bäume mit n Knoten in Rangwerte abbildet, dann gilt : Rang ∈ O(log n)

NB: $\log(n)$ bezeichnet den Zweierlogarithmus ($\log(64) = 6$). Bei O-Notation spielt die Basis des Logarithmus keine Rolle, da alle Logarithmen proportional sind. Z.B.: $\log(n) = \ln(n)/\ln(2)$.

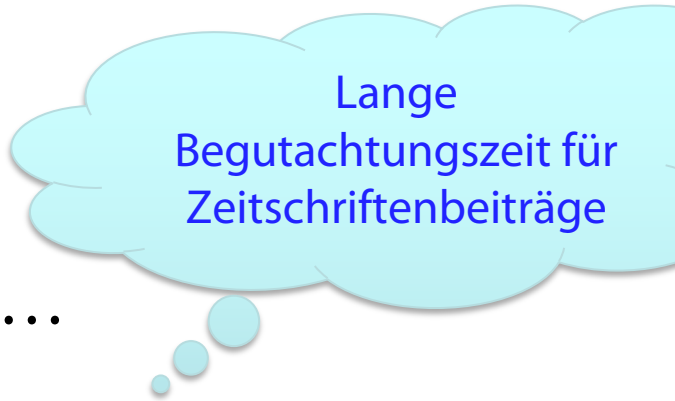
Also gilt Inv

- Ein Fibonacci-Heap aus n Elementen hat also Bäume vom Rang maximal O(log n) (wie beim Binomial-Heap)

- Das ist keine intuitive Einsicht!
- Für diese und andere Arbeiten hat Tarjan den Turing-Award bekommen (1986)

  https://en.wikipedia.org/wiki/Turing_Award

- Weiter mit der amortisierten Analyse…



Michael L. Fredman, Robert E. Tarjan: Fibonacci heaps and their uses in improved network optimization algorithms. In: Journal of the ACM. 34, Nr. 3, S. 596–615, **1987**

Fibonacci-Heap: delete_min

Behauptung:

Die amortisierten Kosten von delete_min() sind in O(log n).

Beweis:

- Einfügen der Kinder von x in Wurzelliste (#Kinder(x) = Rang(x)):
  $\Delta bal_1 =$ Rang(x) – 1 (-1 weil x entfernt wird)
- Jeder Merge-Schritt verkleinert #Bäume um 1:
  $\Delta bal_2 =$ -(#Merge-Schritte)
- Wegen Inv (Rang der Bäume max. O(log n))
  gilt: #Merge-Schritte = #Bäume – O(log n)
- Insgesamt: Δbal_{delete_min} = Rang(x) – 1 - #Bäume + O(log n)
- Laufzeit (in geeigneten Zeiteinheiten):
  t_{delete_min} = #Bäume[1] + O(log n)
- Amortisierte Laufzeit:
  $a_{delete\_min} = t_{delete\_min} + \Delta bal_{delete\_min} \in O(\log n)$

[1] Realisierung der Eimerkette

Fibonacci-Heap: delete

Behauptung: Die amortisierten Kosten von delete(x) sind O(log n).

Beweis: (x ist kein min-Element – sonst wie oben)

- Einfügen der Kinder von x in Wurzelliste:
  $\Delta bal_1 \leq rang(x)$
- Jeder kaskadierende Schritt (Entfernung eines markierten Knotens) erhöht die Anzahl Bäume um 1:
  $\Delta bal_2 =$ #kaskadierende Schritte
- Jeder kaskadierende Schritt entfernt eine Markierung:
  $\Delta bal_3 = -2 \cdot$#kaskadierende Schritte
- Der letzte Schritt von delete erzeugt evtl. eine Markierung:
  $\Delta bal_4 \in \{0,2\}$

Fibonacci-Heap: delete (Forts.)

Behauptung: Die amortisierten Kosten von delete(x) sind O(log n).

Beweis (Fortsetzung):

- Insgesamt:
  Δbal_{delete} = Rang(x) - #kaskadierende Schritte + O(1)
  = O(log n) - #kaskadierende Schritte
- Laufzeit (in geeigneten Zeiteinheiten):
  t_{delete} = O(1) + #kaskadierende Schritte
- Amortisierte Laufzeit:
  $a_{delete} = t_{delete} + \Delta bal_{delete} \in O(\log n)$

Fibonacci-Heap: decrease_key

Behauptung: Die amortisierten Kosten von decrease_key(x,Δ) sind O(1).

Beweis:

- Jeder kask. Schritt erhöht die Anzahl Bäume um 1:
  $\Delta bal_1 =$ #kaskadierende Schritte
- Jeder kask. Schritt entfernt eine Markierung (bis auf x):
  $\Delta bal_2 \leq -2 \cdot$(#kaskadierende Schritte$-1$)
- Der letzte Schritt erzeugt evtl. eine Markierung:
  $\Delta bal_3 \in \{0,2\}$
- Insgesamt: $\Delta bal_{decrease\_key} =$ - #kask. Schritte + O(1)
- Laufzeit: $t_{decrease\_key} =$ #kask. Schritte + O(1)
- Amortisierte Laufzeit:
  $a_{decrease\_key} = t_{decrease\_key} + \Delta bal_{decrease\_key} = O(1)$

Zusammenfassung: Laufzeitvergleich

Laufzeit	Binärer Heap	Binomial-Heap	Fibonacci-Heap
insert	O(log n)	O(log n)	O(1)
min_element	O(1)	O(1)	O(1)
delete_min	O(log n)	O(log n)	O(log n) amor.
delete	O(log n)	O(log n)	O(log n) amor.
decrease_key	O(log n)	O(log n)	O(1) amor.
merge	O(n)	O(log n)	O(1)

Michael L. Fredman, Robert E. Tarjan: Fibonacci heaps and their uses in improved network optimization algorithms. In: Journal of the ACM. 34, Nr. 3, S. 596–615, **1987**

Zusammenfassung: Laufzeitvergleich

Laufzeit	Binärer Heap	Binomial-Heap	Fibonacci-Heap
insert	O(log n)	O(log n)	O(1)
min_element	O(1)	O(1)	O(1)
delete_min	O(log n)	O(log n)	O(log n) amor.
delete	O(log n)	O(log n)	O(log n) amor.
decrease_key	O(log n)	O(log n)	O(1) amor.
merge	O(n)	O(log n)	O(1)

Weitere Entwicklung unter Ausnutzung von Dateneigenschaften: Radix-Heap

Radix-Heap (nur Analyse)

Voraussetzungen [Bearbeiten]

1. alle Schlüssel sind aus den natürlichen Zahlen
2. max. Schlüssel - min. Schlüssel $\leq$ C für ein festes C
3. Monotonie von delete min d.h. die von aufeinander folgenden delete min-Aufrufen zurückgegebenen Werte sind monoton steigend

Laufzeit	Radix-Heap	erw. Radix-Heap
insert	O(log C) amor.	O(log C) amor.
min	O(1)	O(1)
delete_min	O(1) amor.	O(1) amor.
delete	O(1)	O(1) amor.
merge	n/a	O(log C) amor.
decrease_key	O(1)	O(log C) amor.

Ahuja, Ravindra K.; Mehlhorn, Kurt; Orlin, James B.; Tarjan, Robert E., Faster algorithms for the shortest path problem, Journal of the Association for Computing Machinery 37 (2): 213–223, **1990**

Es geht auch ohne amortisierte Analyse …

operation	linked list	binary heap	binomial heap	pairing heap †	Fibonacci heap †	Brodal queue
MAKE-HEAP	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
IS-EMPTY	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$
DELETE-MIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASE-KEY	$O(1)$	$O(\log n)$	$O(\log n)$	$2^{\sqrt{O(\log \log n)}}$	$O(1)$	$O(1)$
DELETE	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
MERGE	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$
MIN	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

† amortized

Fredman, Michael L.; Sedgewick, Robert; Sleator, Daniel D.; Tarjan, Robert E. The pairing heap: a new form of self-adjusting heap. Algorithmica. 1 (1): 111–129, **1986**.

Gerth Stølting Brodal, Worst-case efficient priority queues. Proc. 7th ACM-SIAM Symposium on Discrete Algorithms, pp. 52–58, **1996**

Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan, STRICT FIBONACCI HEAPS, In Proc. 44th Annual ACM Symposium on Theory of Computing, pages 1177-1184, **2012**.

Informatik als Wissenschaft

70+ Jahre Entwicklung und Analyse von Algorithmen und Datenstrukturen