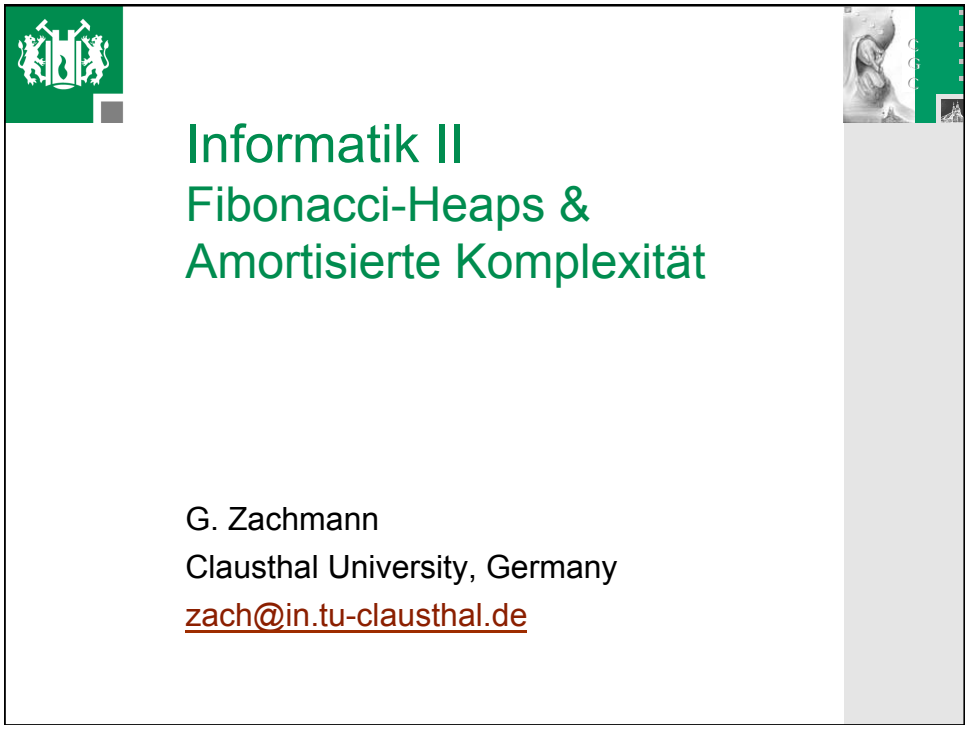


Informatik II
Fibonacci-Heaps & Amortisierte Komplexität

G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de

Motivation für amortisierte Laufzeit

- Betrachte Stack mit Operationen push, pop, multipop
  - multipop entfernt *k* Elemente auf einmal

```
class stack( object ):
    def multipop( self, k ):
        while not self.isempty() and k > 0:
            self.pop(); k -= 1
```

- Laufzeit der Operationen
  - T(push) ∈ $O(1)$
  - T(pop) ∈ $O(1)$
  - T(multipop) ∈ $O(k)$
- Betrachte nun eine Sequenz von Operationen $o_1, \ldots, o_n$
  - wobei der Stack am Anfang und am Ende leer sein sollen und $o_i \in$ {push, pop, multipop}
- Frage: was ist die max. Gesamtlaufzeit irgendeiner solchen Sequenz?

- Naive Worst-Case-Analyse liefert Gesamtlaufzeit für diese Sequenz von $O(n^2)$, denn:
  - Stack hat max. n Elemente
  - Worst-Case T(multipop) $\in O(n)$
  - Worst-Case-Laufzeit für gesamte Sequenz $\in O(n^2)$
    nämlich: $n \cdot \max\{ T(push), T(pop), T(multipop) \}$
- Problem: diese Worst-Case-Laufzeit ist zwar nicht falsch, aber extrem "großzügig" (*not tight*)
  - Einfaches Aufsummieren aller Worst-Case-Laufzeiten ist zu pessimistisch
- Lösung: amortisierte Analyse

- Wozu gibt es eigtl. Datenstrukturen?
  1. DS selbst löst ein Problem (z.B. Datenbank), nämlich bestimmte Daten speichern und wiederfinden
  2. DS ist Komponente eines Algorithmus (z.B. Heap in Heapsort)
- Fall 2 stellt eigentlich eine Dekomposition eines Gesamtalgorithmus dar; ein Teil davon ist ein ADT, gegeben als:
  1. Spezifikation eines ADT
  2. Constraints, wie lange welche Operation des ADT benötigen darf (damit der Gesamtalgorithmus eine bestimmte angepeilte Effizienz erreicht)
- Szenario:
  Algorithmus *A* verwendet DS *D* und wendet im Verlauf des Algorithmus eine Sequenz von Operationen auf *D* an: $o_1, \ldots, o_n$
  - Diese Operationen sind alle Teil der Schnittstelle / des ADT

Die Amortisierte Laufzeit

- Damit *A* im Worst-Case effizient ist, ist es *nicht* notwendig, daß
  $\forall o_i \in$ ADT: T(o_i) effizient ist
- Es genügt, daß

$$\forall \text{ Sequenzen } o_1, \ldots, o_n : \sum_{i=1}^{m} T(o_i) \text{ effizient}$$

- Definition:
  Sei ein ADT *D* mit einer Menge Operationen o_i gegeben, und seien $A(o_i) : \mathbb{R} \to \mathbb{R}$ "Laufzeitfunktionen".
  Wenn nun gilt

$$\forall m \, \forall \text{ Sequenzen } o_1, \ldots, o_m : \sum_{i=1}^{m} T(o_i) \leq \sum_{i=1}^{m} A(o_i),$$

  dann heißen die A(o_i) amortisierte Worst-Case-Laufzeiten.

- Bemerkungen:
  - Amortisierte Laufzeit ist *keine* Avergarge-Case-Analyse!
    - Kein Mittelwert über alle möglichen Eingaben (z.B. Quicksort)
    - Kein Mittelwert über alle möglichen, zufälligen Entscheidungen eines Algorithmus (Skipliste)
  - Amortisierte Laufzeit erlaubt uns, der einen Operation etwas mehr Laufzeit zuzuschieben als sie wirklich benötigt und einer anderen dafür etwas abzuziehen
  - Diese Technik kam Ende der 80er Jahre und führte zu vielen neuen effizienten Algorithmen
- Es gibt 3 Methoden, um amortisierte Analyse durchzuführen:
  - Aggregatmethode
  - Bankkonto-Paradigma
  - Potentialmethode

Bankkonto-Paradigma

- Engl.: accounting method
- Manche Operationen bekommen höhere, mache niedrigere Kosten zugewiesen, als sie eigentlich verursachen
  - Verwende ein "Bankkonto" zur Buchhaltung
    - Typischerweise mehrere Konten, die bestimmten Elementen der DS zugewiesen werden
  - Amortisierte Kosten einer Operation = tatsächliche Kosten (Laufzeit) ± "Guthaben" vom Bankkonto
    - Tatsächliche Kosten > amortisierte Kosten ⇔ Guthaben wird vom Konto verbraucht
    - Tatsächliche Kosten < amortisierte Kosten ⇔ Guthaben wird auf Konto eingezahlt (für später)

Beispiel: der Stack mit multipop-Operation

- Tatsächliche Kosten der Operationen:
  T(push) = 1, T(pop) = 1, T(multipop) = k
- Amortisierte Analyse:
  - Verwende pro Element auf dem Stack ein Konto
  - Weise diesen (kraft unserer Intuition) folgende amortisierte Kosten zu:

    A(push): 1 + 1
    A(pop): 0
    A(multipop): 0
  - Dann gilt:

$$\sum T(o_i) \leq \sum A(o_i)$$

- Beweis:
  - Der Stack ist am Anfang und am Ende leer
  - Für jede beliebige Sequenz $o_1, \ldots, o_m$ kann man $\sum T(o_i)$ (hier) leicht ausrechnen, da jedes Element genau einmal gepusht und wieder gepopt werden kann$\rightarrow \sum T(o_i) = 2k$, k = #Push-Operationen in der Sequenz
  - $\sum A(o_i) = 2k \rightarrow$ Beh.
- Also: die amortisierte Kosten sind somit

Operation	Worst-Case	Worst-Case amortisiert
push	$O(1)$	$O(1)$
pop	$O(1)$	$O(1)$
multipop	$O(n)$	$O(1)$

mit n = Anzahl Elemente auf dem Stack

Fibonacci-Heaps und deren amortisierte Kosten

Erinnerung: P-Queues

- Bekannte Datenstrukturen zum chronologischen Einfügen und Entnehmen von Elementen: Stack (LIFO = "last in - first out") und Queue (FIFO = "first in - first out")
- Erinnerung: *Priority Queue* (P-Queue)
  - Elemente erhalten beim Einfügen einen Wert für ihre Priorität
  - Entnommen wird immer das Element mit höchster Priorität
  - Beispiel: "To-do"-Liste
  - Ein Element (z.B. Knoten im Heap) enthält den eigentlichen Inhalt sowie die ihm zugeordnete Priorität
- Priorität ist meist ein Integer oder Float (wobei eine kleinere Zahl für eine höhere Priorität steht)

Operationen

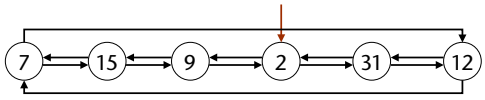
- Kernoperationen:
  - Insert: ein neues Element hinzufügen (zusammen mit seiner Priorität)
  - Extract-Max: das Element mit der höchsten Priorität (niedrigster Zahl) entfernen
- Weitere Operationen, die man bei P-Queues oft braucht:
  - Rückgabe des Elements mit höchster Priorität (ohne Entfernen)
  - Ein gegebenes Element "wichtiger" machen, d.h. seine Priorität erhöhen
  - Ein gegebenes Element wird überflüssig, d.h. es wird entfernt
  - Zwei P-Queues zusammenfügen (merge)

Das API

- Operationen einer P-Queue) *Q:*
  - `Q.insert(object o, int p):` füge ein neues Element mit Priorität *p* ein
  - `Q.access_min():` gib das Element mit der höchsten Priorität zurück
  - `Q.extract_min():` entferne das Element mit der höchsten Priorität und liefere dieses zurück
  - `Q.increase_prio(object o, int p):` setze die Priorität von Element *o* auf den Wert *p* herauf
  - `Q.delete(object o):` entferne Element *o*
  - `Q.merge(PQueue P):` vereinige *Q* mit P-Queue *P*
  - `Q.isEmpty():` gibt an, ob *Q* leer ist
- Bemerkung: Die effiziente Suche nach einem bestimmten Element oder Key wird in P-Queues nicht unterstützt! Für `increase_prio()` und `delete()` muß man das entsprechende Element also bereits kennen, d.h., einen Pointer auf dieses Element im Heap haben. Die Queue muß dann intern sich allerdings (möglichst effizient) re-organisieren.

Implementationsmöglichkeit 1: Liste

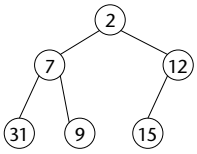
- Idee: Doppelt verkettete zirkuläre Liste mit zusätzlichem Zeiger auf das Minimum (= Knoten mit maximaler Priorität)
- Implementierung der Operationen:



  - insert: füge das neues Element irgendwo ein und (falls nötig) aktualisiere den Minimum-Zeiger
  - accessmin: gib den Minimalknoten zurück
  - deletemin: entferne den Minimalknoten, aktualisiere den Minimum-Zeiger (laufe durch Liste)
  - decreasekey: setze den Wert herab (höhere Prio) und aktualisiere den Minimum Zeiger
  - delete: falls der zu entfernende Knoten der Minimalknoten ist, führe deletemin aus, ansonsten entferne den Knoten
  - merge: hänge die beiden Listen aneinander

Implementationsmöglichkeit 2: Heap

- Idee: Speichere die Elemente in einem Heap
  - Hier soll das Minimum ganz oben im Heap stehen
  - Kann man als Array speichern (vgl. Heap-Sort)



2	7	12	31	9	15

- Operationen:
  - insert: füge das neue Element an der letzten Stelle ein und stelle durch Vertauschungen die Heapordnung wieder her
  - accessmin: der Minimalknoten steht immer an der ersten Position
  - deletemin: ersetze das erste Element durch das letzte, dann versickere
  - decreasekey: setze den Schlüssel herab und stelle durch Vertauschungen die Heapordnung wieder her
  - delete: ersetze das entfernte Element durch das letzte, dann versickere
  - merge: füge alle Elemente des kleineren Heaps nacheinander in den größeren ein

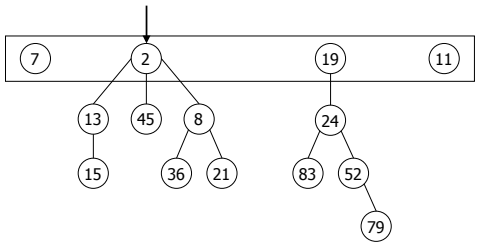
Vergleich der beiden Implementationen

	lineare Liste	Heap	???
insert	$O(1)$	$O(\log n)$	$O(1)$
accessmin	$O(1)$	$O(1)$	$O(1)$
deletemin	$O(n)$	$O(\log n)$	$O(\log n)$
decreasekey	$O(1)$	$O(\log n)$	$O(1)$
delete	$O(n)$	$O(\log n)$	$O(\log n)$
merge	$O(1)$	$O(m \log(n+m))$	$O(1)$

- Lassen sich die Vorteile von Listen und Heaps verbinden?
- Antwort: ja, allerdings "nur" mit amortisierter Laufzeit

Fibonacci-Heaps: Die Idee

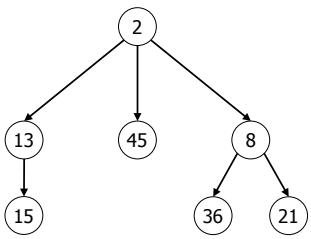
- Liste von Bäumen (beliebigen Verzweigungsgrades), die alle heap-geordnet sind



- Definition: Ein Baum heißt heap-geordnet, wenn der Schlüssel jedes Knotens größer oder gleich dem Schlüssel seines Mutterknotens ist (sofern er eine Mutter hat)
- Die Wurzeln der Bäume sind in einer doppelt verketteten, zirkulären Liste miteinander verbunden (Wurzelliste)
- Der Einstiegspunkt ist ein Zeiger auf den Knoten mit minimalem Schlüssel

Exkurs: Bäume

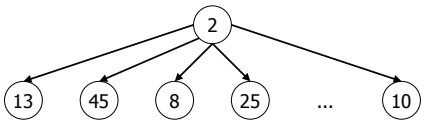
- Bäume lassen sich als Verallgemeinerung von Listen auffassen:
  - es gibt genau ein Anfangselement („Wurzel“)
  - jedes Element (außer der Wurzel) ist Nachfolger von genau einem Knoten
  - jedes Element kann beliebig viele Nachfolger („Söhne“) haben



Repräsentation von Bäumen

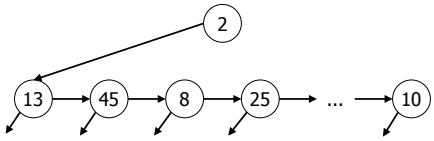
- Bei Bäumen mit hohem Verzweigungsgrad ist es aus Speicherplatzgründen ungünstig, in jedem Knoten Zeiger auf alle Söhne zu speichern.

```
class TreeNode {
  int key;
  TreeNode children[];
}
```

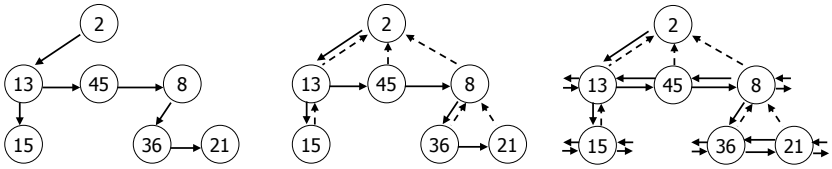


- Eine platzsparende Alternative ist die Child-Sibling-Darstellung:
  - alle Söhne sind in einer Liste untereinander verkettet, dadurch genügt es, im Vaterknoten einen Zeiger auf den ersten Sohn zu speichern

```
class TreeNode {
  int key;
  TreeNode child;
  TreeNode sibling;
}
```

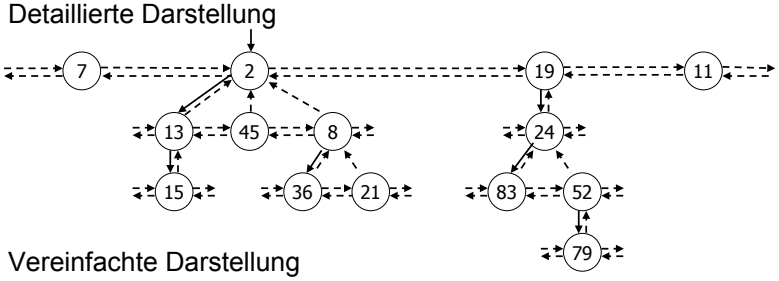


Child-Sibling-Repräsentation

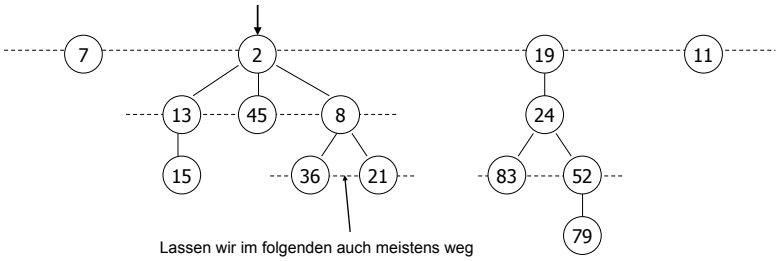


- Um sich im Baum auch aufwärts bewegen zu können, fügt man einen Zeiger auf den Vaterknoten hinzu
- Um das Entfernen von Söhnen (und das Aneinanderhängen von Sohn-Listen) in $O(1)$ zu realisieren, verwendet man doppelt verkettete zirkuläre Listen
- Also hat jeder Knoten 4 Zeiger: child, parent, left, right

- Detaillierte Darstellung



- Vereinfachte Darstellung



Lassen wir im folgenden auch meistens weg

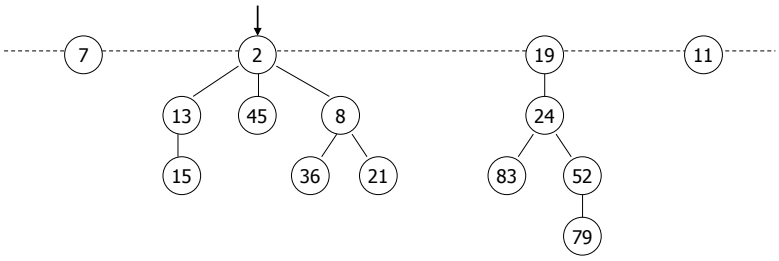
Das Knotenformat in Fibonacci-Heaps

```
class FibNode {
    Object content;          // der eigentliche Inhalt
    int key;                 // Schlüssel (Priorität)
    FibNode parent, child;   // Zeiger auf Vater und einen Sohn
    FibNode left, right;     // Zeiger auf linken und rechten
                             // Nachbarn
    int rank;                // Anzahl der Söhne dieses Knotens
    boolean mark;            // Markierung
}
```

- Die Zahl rank gibt an, wie viele Söhne der Knoten hat (= der Rang des Knotens)
- Die Bedeutung der Markierung mark wird später deutlich. Diese Markierung gibt an, ob der Knoten bereits einmal einen seiner Söhne verloren hat, seitdem er selbst zuletzt Sohn eines anderen Knotens geworden ist

Die "einfachen" Operationen

- **`Q.accessmin():`** gib den Knoten **`Q.min`** zurück (bzw. **`NULL`**, wenn *Q* leer ist)
- **`Q.insert(int k):`** erzeuge einen neuen Knoten *N* mit Schlüssel *k* und füge ihn in die Wurzelliste von *Q* ein. Falls *k* < **`Q.min.key`**, aktualisiere den Minimum-Zeiger (setze **`Q.min`** = N), gib den neu erzeugten Knoten zurück

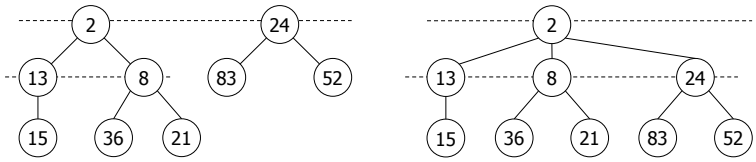


Manipulation von Bäumen in Fibonacci-Heaps

- Zur Implementierung der übrigen Operationen auf Fibonacci-Heaps benötigen wir drei Basis-Methoden zur Manipulation von Bäumen in Fibonacci-Heaps:
  - link = "Wachstum" von Bäumen: zwei Bäume werden zu einem neuen verbunden
  - cut = "Beschneiden" von Bäumen im Inneren: ein Teilbaum wird aus einem Baum herausgetrennt und als neuer Baum in die Wurzelliste eingefügt
  - remove = "Spalten" von Bäumen an der Wurzel: entfernt die Wurzel eines Baums und fügt die Söhne der Wurzel als neue Bäume in die Wurzelliste ein

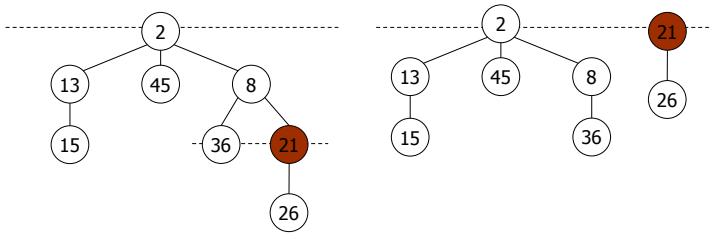
Baummanipulation link

- Input: 2 Knoten mit demselben Rang *k* in der Wurzelliste
- Methode: vereinige zwei Bäume mit gleichem Rang, indem die Wurzel mit größerem Schlüssel zu einem neuen Sohn der Wurzel mit kleinerem Schlüssel gemacht wird.
- Nachbedingung: die Gesamtzahl der Bäume verringert sich um 1, die Knotenzahl ändert sich nicht
- Output: 1 Knoten mit Rang *k*+1
- Laufzeit: *O*(1)



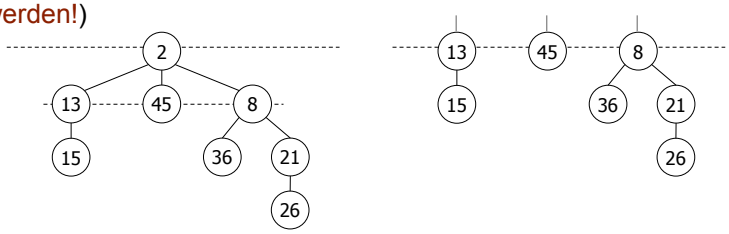
Baummanipulation cut

- Input: 1 Knoten, der nicht in der Wurzelliste ist
- Methode: trenne den Knoten (samt dem Teilbaum, dessen Wurzel er ist) von seinem Vater ab und füge ihn als neuen Baum in die Wurzelliste ein.
- Nachbedingung: die Gesamtzahl der Bäume erhöht sich um 1, die Knotenzahl ändert sich nicht
- Laufzeit: *O*(1)



Baummanipulation remove

- Input: 1 Knoten mit Rank *k* aus der Wurzelliste
- Methode: entferne die geg. Wurzel des Baums und füge statt dessen deren *k* Söhne in die Wurzelliste ein
- Nachbedingung: die Zahl der Bäume erhöht sich um *k*-1, die Gesamtzahl der Knoten verringert sich um 1
- Laufzeit: *O*(1) (sofern die Vaterzeiger der Söhne nicht gelöscht werden!)

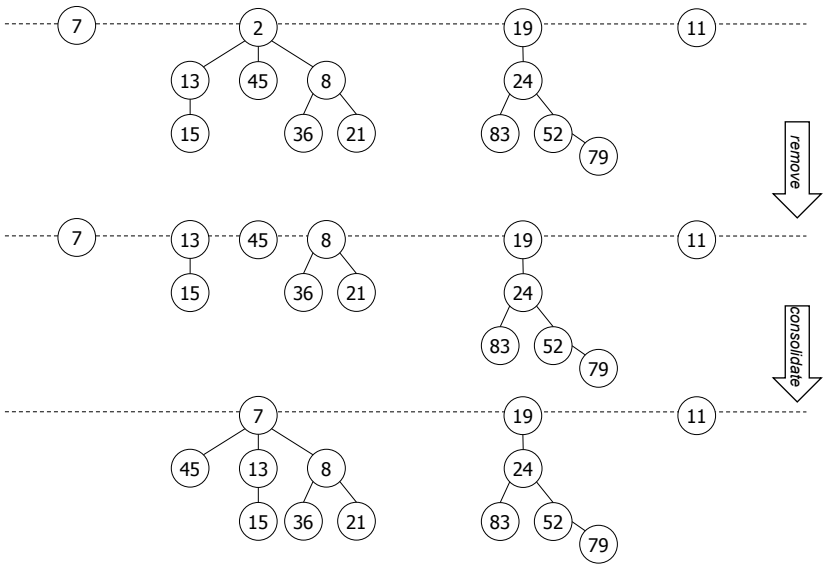


- Weitere Operationen: mit Hilfe der drei Manipulationsmethoden **`link`**, **`cut`**, **`remove`** lassen sich die noch fehlenden Operationen **`deletemin`**, **`decreasekey`**, **`delete`** beschreiben

Entfernen des minimalen Knotens (**`deletemin`**)

1. Entferne den Minimalknoten (mit **`remove`**).
2. "Konsolidiere" die Wurzelliste:
   - verbinde (mit **`link`**) je zwei Wurzelknoten mit demselben Rang, und zwar solange, bis nur noch Knoten mit unterschiedlichem Rang in der Wurzelliste vorkommen
   - füge den größeren unter dem kleineren ein, damit Heap-Eigenschaft erhalten bleibt
   - entferne dabei evtl. vorhandene Vaterzeiger der (ehem.) Wurzelknoten
3. Finde unter den verbliebenen Wurzelknoten das neuen Minimum
4. Gib den entfernten Knoten zurück

Beispiel



Weitere höhere Operationen

- **`merge:`**
  1. hänge Wurzelliste von Q an Wurzelliste von P
  2. aktualisiere Minimum-Zeiger von P: falls **`P.min.key < Q.min.key`**, setze **`Q.min = P.min`**
- **`decreasekey`**, **`delete`**, und **`merge`**:
  "we refer the interested reader to …"

Laufzeiten

	lineare Liste	Heap	Fibonacci-Heap
insert	$O(1)$	$O(\log n)$	$O(1)$
accessmin	$O(1)$	$O(1)$	$O(1)$
deletemin	$O(n)$	$O(\log n)$	?
decreasekey	$O(1)$	$O(\log n)$	?
delete	$O(n)$	$O(\log n)$	?
merge	$O(1)$	$O(m \log(n+m))$	$O(1)$

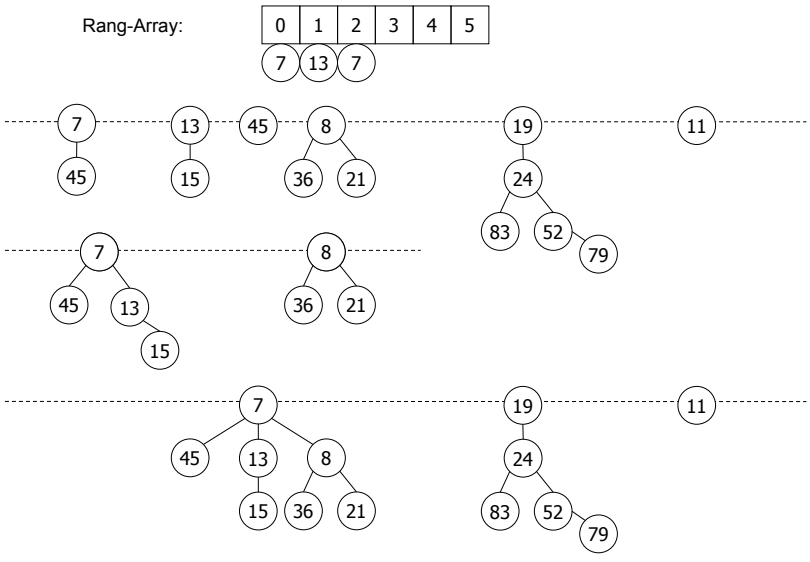
Laufzeitanalyse von `deletemin`

- Laufzeit von **`deletemin`**`()`:
  1. remove: O(1)
  2. consolidate: ?
  3. updatemin: O(#Wurzelknoten nach consolidate)
- Nach dem Konsolidieren gibt es von jedem Rang nur noch höchstens einen Wurzelknoten
- Definiere maxRank(n) als den höchstmöglichen Rang, den ein Wurzelknoten in einem Fibonacci-Heap der Größe n haben kann (Berechnung von maxRank(n) später)
- Nun müssen wir die Komplexität von consolidate bestimmen

Konsolidieren der Wurzelliste

- Wie kann man das Konsolidieren effizient realisieren?
- Beobachtungen:
  - jeder Wurzelknoten muß mindestens einmal betrachtet werden
  - am Ende darf es für jeden möglichen Rang höchstens einen Knoten geben
- Idee:
  - trage die Wurzelknoten der Reihe nach in ein temporäres Array (das sog. "Rang-Array") ein
  - jeder Knoten wird an der Arrayposition eingetragen, die seinem Rang entspricht (ähnlich wie bei Count- und Bucket-Sort)
  - ist eine Position schon besetzt, so weiß man, daß es einen weiteren Knoten mit demselben Rang gibt, kann diese beiden mit **`link`** verschmelzen und den neuen Baum an der nächsthöheren Position im Array eintragen; dort wiederholt sich der link-Vorgang eventuell

Beispiel: consolidate



Analyse von consolidate

```
rankArray = (maxRank(n)+1) * [None]        # erstelle Array
for N in "list of nodes of rootlist":
    while rankArray[N.rank] != None:       # Pos. besetzt
        r_old = N.rank
        N = link( N, rankArray[N.rank] )   # verbinde Bäume
        rankArray[r_old] = None            # lösche alte Pos.
    rankArray[N.rank] = N
```

- Sei k = #Wurzelknoten vor dem Konsolidieren
- Diese k Knoten lassen sich aufteilen in
  W = {Knoten, die am Ende noch in der Wurzelliste sind}
  L = {Knoten, die an einen anderen Knoten angehängt wurden}
- Es gilt: $|W| + |L| = k$ und
  $T(\text{consolidate}) = T(W) + T(L) + c \cdot \text{maxRank}(n)$

Gesamtkosten von `deletemin`

remove	$O(1)$	
Erstellen des Rank-Arrays	$O(\text{maxRank}(n))$	consolidate
link-Operationen	$\|L\| \cdot O(1)$	
restl. Eintragungen	$O(\text{maxRank}(n))$	
Update Minimum-Zeiger	$O(\text{maxRank}(n))$	
Gesamtkosten	$O(\|L\| + \text{maxRank}(n))$	

Amortisierte Analyse

- Beobachtung: bei **`deletemin`** beeinflusst die Zahl der **`link`**-Operationen die tatsächliche Laufzeit
- Idee: spare dafür Guthaben an (Bankkonto-Paradigma!)
- Wir wissen: die Kosten pro **`link`** sind jeweils 1€
  - sorge dafür, daß für jeden Wurzelknoten immer 1€ Guthaben vorhanden ist, mit dem sich die **`link`**-Operation bezahlen lässt, wenn dieser Knoten an einen anderen angehängt wird
- Wann müssen wir etwas "dazu bezahlen"?
  - neue Wurzelknoten können entstehen bei
    - insert: gib dem neu eingefügten Wurzelknoten noch 1€ dazu
    - remove: bezahle für jeden Sohn des entfernten Knotens 1€ dazu, insgesamt also bis zu maxRank(n) €

Amortisierte Kosten

- **`insert:`**

Erstellen des Knotens	$O(1)$
Einfügen in Wurzelliste	$O(1) + 1$
amortisierte Gesamtkosten	$O(1)$

- **`deletemin:`**

remove	$O(1) + O(\text{maxRank}(n))$
erstellen des Rank-Arrays	$O(\text{maxRank}(n))$
link-Operationen	$\|L\| \cdot O(1)$ wird vom Guthaben bezahlt!
restl. Eintragungen	$O(\text{maxRank}(n))$
Update Minimum-Zeiger	$O(\text{maxRank}(n))$
amortisierte Gesamtkosten	$O(\text{maxRank}(n))$

Zwischenstand

- Amortisierte Kosten:
  - insert O(1)
  - accessmin O(1)
  - deletemin O(maxRank(n))
  - decreasekey O(1) [o. Bew.]
  - delete O(maxRank(n)) [o. Bew.]
  - merge O(1)
- Noch zu zeigen: maxRank(n) $\in O(\log n)$, d.h. der maximle Rang eines Knotens in einem Fibonacci-Heap ist logarithmisch in der Größe n des Fibonacci-Heaps

Berechnung von maxRank(n)

- Erinnerung: Fibonacci-Zahlen

$$F_0 = 0, F_1 = 1$$
$$F_{k+2} = F_{k+1} + F_k \quad \text{für } k \geq 0$$

  - Die Folge der Fibonacci-Zahlen wächst exponentiell mit $F_{k+2} \geq 1.618^k$
- Es gilt außerdem:

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

(Beweis durch vollständige Induktion über k)

- Lemma 1: Sei N ein Knoten in einem Fibonacci-Heap und k = rang(N). Betrachte die Söhne $C_1, \ldots, C_k$ von N in der Reihenfolge, in der sie (mit link) zu N hinzugefügt wurden. Dann gilt:
  1. rang(C_1) ≥ 0
  2. rang(C_i) ≥ i – 2, für i = 2, …, k
- Beweis:
  1. klar
  2. Als C_i zum Sohn von N wurde, waren $C_1, \ldots, C_{i-1}$ schon Söhne von N, d.h. es war rang(N) ≥ i-1.
     Durch **`link`** werden immer Knoten mit gleichem Rang verbunden.
     → Beim Einfügen war auch rang(C_i) ≥ i-1.
     [Durch **`delete`** *kann C_i höchstens einen Sohn verloren haben (wegen cascading cuts), daher muß gelten: C_i.rank ≥ i – 2]*

- Lemma 2: Sei N ein Knoten in einem Fibonacci-Heap und k = rang(N). Sei size(N) die Zahl der Knoten im Teilbaum mit Wurzel N. Dann gilt:
  size(N) ≥ F_{k+2} ≥ 1.618^k
  D.h., ein Knoten mit k Töchtern hat mind. F_{k+2} Nachkommen (inkl. sich selbst).
- Beweis durch Induktion:
  Definiere s_k = min{ size(N) | N mit rang(N) = k },
  d.h., s_k = kleinstmögliche Größe eines Baums mit Wurzelrang k.
  Klar: s_0 = 1 und s_1 = 2.
  Seien wieder $C_1, \ldots, C_k$ die Söhne von N in der Reihenfolge, in der sie zu N hinzugefügt wurden. Dann gilt

$$\text{size}(N) \geq s_k \geq 1 + \sum_{i=1}^{k} \underbrace{\text{size}(C_i)}_{\text{rang}(C_i) \geq i-2} \geq 1 + \sum_{i=1}^{k} F_i = F_{k+2} \geq 1.6818^k$$

- Satz: Der maximale Rang maxRank(n) eines beliebigen Knotens in einem Fibonacci-Heap mit n Knoten ist beschränkt durch $O(\log n)$
- Beweis:
  Sei N ein Knoten eines Fibonacci-Heaps mit n Knoten und sei k = rang(N).
  Es ist n = size(N) ≥ 1.618^k (nach Lemma 2)
  Daher ist $k \leq \log_{1.618}(n) \in O(\log n)$

Zusammenfassung

Operation	lineare Liste	Heap	Fibonacci-Heap
insert	$O(1)$	$O(\log n)$	$O(1)$
accessmin	$O(1)$	$O(1)$	$O(1)$
deletemin	$O(n)$	$O(\log n)$	$O(\log n)$*
decreasekey	$O(1)$	$O(\log n)$	$O(1)$*
delete	$O(n)$	$O(\log n)$	$O(\log n)$*
merge	$O(1)$	$O(m \log(n+m))$	$O(1)$

* amortisierte Kosten