Warteschlange (Priority Queue)

- Dynamische Menge von Elementen (Schlüssel, Info)
- Elemente sollen nach Prioritäten sortiert aus der Warteschlange entnommen werden
- Priorität basiert auf Schlüssel
- Zwei Versionen: Min-Priorität (Element mit kleinstem Schlüssel hat höchste Priorität) oder Max-Priorität (Element mit größtem Schlüssel hat höchste Priorität, symmetrisch)
- Hier: Min-Priorität

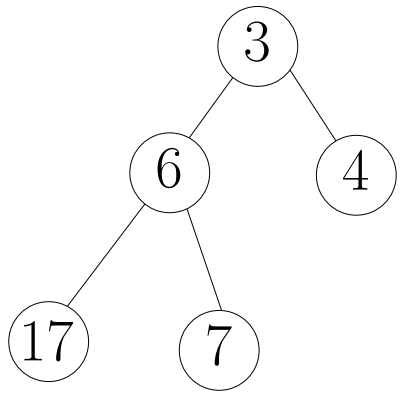
Annahme: Man kann auf Elemente in der Priority Queue direkt in konstanter Zeit zugreifen (über Zeiger)

Operationen

- MAKEQUEUE:
  erzeugt eine leere Warteschlange
- UNION(H_1, H_2):
  liefert neue Warteschlange mit allen Elementen in $H_1 \dot{\cup} H_2$
- MIN(H):
  gibt Element x mit kleinstem Schlüssel in H zurück
- EXTRACT-MIN(H):
  gibt Element x mit kleinstem Schlüssel in H zurück und löscht es
- INSERT(x, H):
  fügt x in H ein
- DELETE(x, H):
  löscht x aus H
- DECREASE-KEY(x, H, k):
  weist x in H den kleineren Schlüssel k zu

Binary Heap

- Mögliche Implementierung durch *perfektem* binären Baum, der die *Heapeigenschaft* erfüllt
- Binärer Baum ist *perfekt* wenn
  - sich alle Blätter auf benachbarten Ebenen befinden
  - in jeder Ebene alle Blätter möglichst weit links sind
  - es ≤ 1 Knoten mit einem Kind gibt
- Baum erfüllt (Min-)*Heapeigenschaft* falls für jeden Knoten ausser der Wurzel der Schlüsselwert größer ist als der Schlüsselwert des Vaterknotens



Binary Heap

- Wir kennen diese Datenstruktur von Heapsort
- Operationen können mit Heapify als Grundoperation implementiert werden
  - MAKEQUEUE - $O(1)$
  - UNION - $O(n)$
  - MIN - $O(1)$
  - EXTRACT-MIN - $O(\log n)$
  - INSERT - $O(\log n)$
  - DELETE - $O(\log n)$
  - DECREASE-KEY - $O(\log n)$

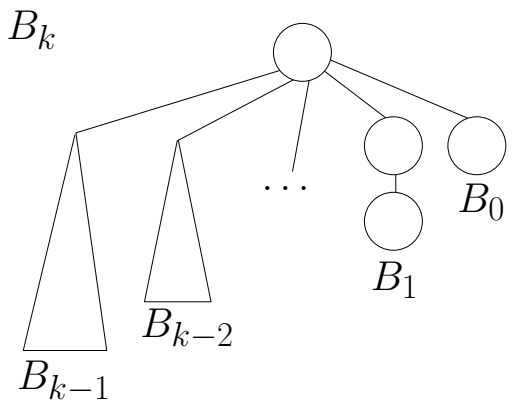
Alternative Datenstruktur

- Erlaubt schnellere Vereinigung (UNION) von Warteschlangen
- Nützlich für Mehrprozessor-Multitasking-Systeme
- Erlaubt Weiterentwicklung in noch bessere Datenstruktur

Binomial Baum

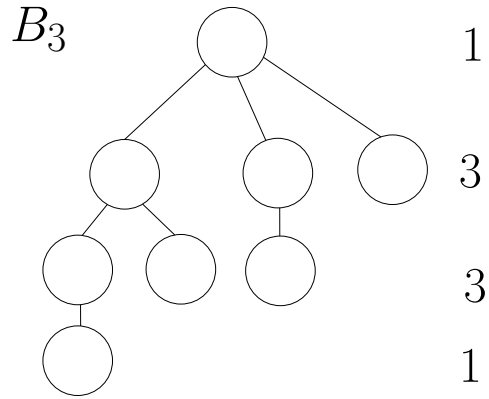
Definition

- B_0: Baum mit einem Knoten
- $B_i, i > 0$: Verkettung zweier Bäume B_{i-1}, wobei ein B_{i-1} an Wurzel des anderen B_{i-1} als linkstes Kind angehängt wird
- wir nennen i den *Grad* (oder auch *Rang*) von B_i



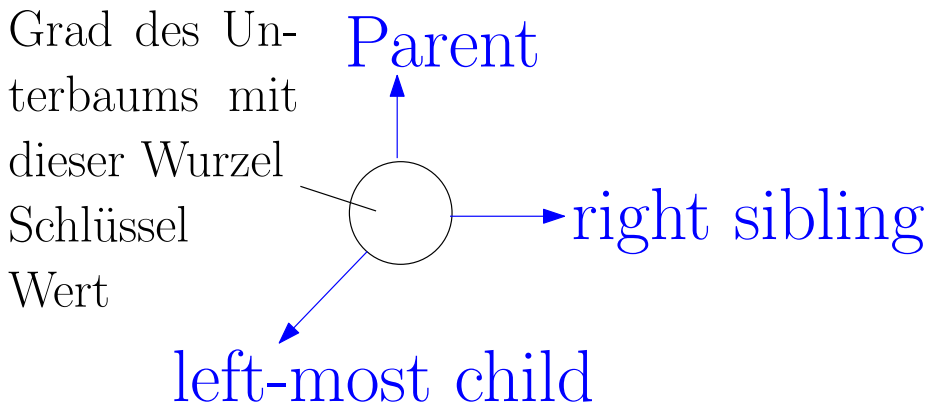
Name *Binomial* Baum

Anzahl der Knoten auf einer Tiefenstufe = Binomialkoeffizienten



Binomial Baum

- Speichern der Elemente als Knoten
- Knoten haben viele Kinder



Parent-Pointer ist nur für die DECREASE-KEY() Operation notwendig.

Binomial Baum (Vuillemin 1978)

- Eigenschaften:
  - B_i hat 2^i Knoten
  - Wurzel von B_i hat i Kinder
  - B_i hat Höhe i

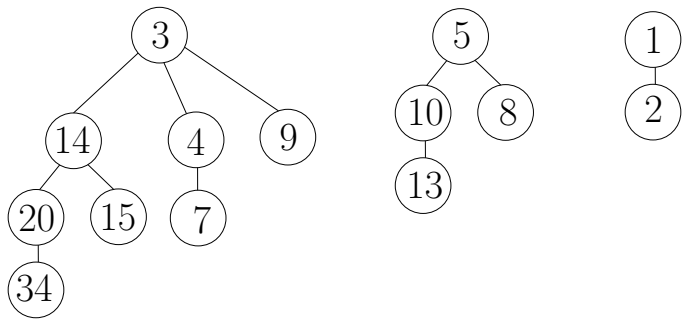
    (Höhe ist # Kanten auf längstem Blatt–Wurzel Pfad)
- Binomial Baum mit Heapeigenschaft erlaubt schnelles Finden des Elements mit der kleinstem Schlüssel (Wurzel)
- Vereinigung zweier Binomialbäume gleichen Grades i mit Heapeigenschaft:
  - LINK(p, q): Baum mit größerem Schlüsselwert an der Wurzel wird als linkstes Kind an die Wurzel des anderen Baum angehängt

    (das erzeugt Binomialbaum vom Grad $i + 1$)

Binomial Heap

- Problem: Binomial Baum kann nur Mengen S von Elementen darstellen, die 2^i Knoten haben
- Binomial Heap oder Binomial Queue
  - Menge von Binomial Bäumen mit 1 Knoten pro Element
  - Jeder Baum hat die Heapeigenschaft
  - Keine zwei Bäume haben denselben Grad
- Für jede Menge S mit $|S| = n$ kann eine solche Menge von Binomial Bäumen gefunden werden.

  Ein Binomialbaum für jede '1' in der Binärdarstelleung von n



Der maximale Grad ist $\log_2 n$.

Operationen Binomial Heap

- MAKEQUEUE: $O(1)$
- MIN(H): $O(\log n)$

  Finde kleinste aller Wurzeln
- UNION(H_1, H_2): $O(\log n)$

  Vereinigung zweier Binomial Heaps verwendet als Grundoperation Vereinigung zweier Binomial Bäume gleichen Grades und ist analog zur Addition von Binärzahlen
- EXTRACT-MIN(H): $O(\log n)$
  - Entferne Binomial Baum B_i mit kleinstem Schlüsselwert in Wurzel aus H
  - Vereinige Kinder von Wurzel von B_i mit dem geänderten H mit UNION

Operationen Binomial Heap

- INSERT(x, H): $O(\log n)$
  - Produziere B_0 mit Wert x
  - Vereinige B_0 und H durch UNION
- DECREASE-KEY(x, H, k): $O(\log n)$
  - Verkleinere Schlüssel von x
  - Tausche x so lange mit Vaterknoten bis Heapeigenschaft wieder hergestellt ist
- DELETE(x): $O(\log n)$
  - DECREASE-KEY($x, H, -\infty$)
  - EXTRACT-MIN(H)

Lazy Binomial Heap (Fredman, Tarjan 1987)

- Menge von Binomial Bäumen mit 1 Knoten pro Element (Menge verwaltet als “Wurzelliste”)
- Jeder Baum hat die Heapeigenschaft
- ~~Keine zwei Bäume haben denselben Grad~~
- MINZEIGER auf Wurzel mit kleinstem Schlüssel

Ziel: Jede Operation braucht nur $O(1)$ Zeit, nur EXTRACTMIN() und DELETE() brauchen $O(\log n)$ Zeit.

Aber es sind **amortisierte** Zeitschranken.

Lazy Binomial Heap

Hilfsoperation CONSOLIDATE(H):

Erzeuge aus Wurzelliste durch wiederholtes Anwenden von LINK() auf Bäume gleichen Grades eine Wurzelliste mit höchstens einem Baum pro Grad.

Setze dabei auch den MINZEIGER auf die Wurzel mit kleinstem Schlüssel.

Accounting-Invariante: Jede Wurzel in der Menge besitzt 1 €.

Mit dieser Invariante kostet CONSOLIDATE() nur $O(\log n)$ amortisierte Zeit: jede LINK() Operation (zusammen mit mglichen Update des MINZEIGERS) bezahlt der Euro der "Verlierer"-Wurzel, die damit aus der Wurzelliste verschwindet. Es bleiben höchstens $O(\log n)$ Wurzeln in der Wurzelliste (höchstens eine für jeden Grad), die dann in $O(\log n)$ Zeit behandelt werden können.

Operationen Lazy Binomial Heap

- MAKEQUEUE: $O(1)$
- MIN(H): $O(1)$ (über MINZEIGER)
- UNION(H_1, H_2): $O(1)$ (konkateniere Wurzellisten, update MINZEIGER)
- EXTRACT-MIN(H): $O(\log n)$ (ersetze MINZEIGER-Wurzel durch ihre Kinderliste; CONSOLIDATE())

  (Für Erhalt der Accounting-Invariante muss jeder der $\leq \log_2 n$ Knoten der Kinderliste einen € bekommen; deswegen amortisierte Kosten $O(\log n)$)
- INSERT(x, H): $O(1)$ (erzeuge B_0 für x, lege € drauf hänge an Wurzelliste von H)
- DECREASE-KEY(x, H, k): $O(\log n)$ (wie bei Binomial Heap durch "sift-up", MINZEIGER aufrechterhalten)
- DELETE(x): $O(\log n)$ (wie bei Binomial Heap durch DECREASE-KEY(x, H, $-\infty$) gefolgt von EXTRACT-MIN(H))

Wunschziel

- DECREASE-KEY() auch in konstanter Zeit
- Verzicht auf Parent-Pointers
- nur einfach verkettete Sibbling-Listen

Wunschziel: Versuch 1, Teilbäume wegschneiden

Bei DECREASE-KEY(x, H, k) den Knoten p für x mit seinem Teilbaum aus dem enthaltenden Baum herausschneiden und in die Wurzelliste einfügen; Schlüssel auf k verringern, wenn nötig update von MINZEIGER.

Nachteile:

- benötigt doppelt verkettete Sibbling-Liste
- Keine gute untere Schranke für Größe eines Baums mit Wurzel von Grad i.

  D.h. Wurzelgrade können sehr groß werden, und damit wird EXTRACT-MIN() zu teuer.

Wunschziel: Versuch 2, leere Knoten

Bei DECREASE-KEY(x, H, k) den Knoten p für x leer machen und in der Struktur lassen, aber einen neuen neuen Knoten q für das Stück x mit neuem Schlüssel k in die Wurzelliste einfügen, wenn nötig update von MINZEIGER.

Nachteile:

- Übung!!

Wunschziel: Versuch 3, hohle Knoten

“Hollow Heaps” (Hansen, Kaplan, Tarjan, Zwick (2015))

Bei DECREASE-KEY(x, H, k) den Knoten p für x leer, also “hohl”, machen und in der Struktur lassen, aber einen neuen Knoten q für das Stück mit neuem Schlüssel k in die Wurzelliste einfügen; wenn nötig update von MINZEIGER.

die ersten 2 Kinder (vom höchsten Grad) bleiben bei hohlem p, der Rest der Sibbling-Liste wird zur Kinderliste von q gemacht der offizielle Grad von p bleibt unverändert; der Grad von q wird auf 1 mehr als dem Grad seines ersten Kindes gesetzt (und auf 0, falls kein Kind existiert)

Erweiterte Accounting-Invariante:
Jede Wurzel in der Wurzelliste besitzt 1 €.
Jeder hohle Knoten und jedes seiner ≤ 2 Kinder besitzt 1 €.

Für den Erhalt dieser Invariante legt man je 1 € auf p und seine höchstens 2 verbleibenden Kinder, sowie 1 € auf q

Operationen Hollow Heap

- MAKEQUEUE: $O(1)$
- MIN(H): $O(1)$ (über MINZEIGER)
- UNION(H_1, H_2): $O(1)$ (konkateniere Wurzellisten, update MINZEIGER)
- EXTRACT-MIN(H): ??$O(\log n)$?? (ersetze MINZEIGER-Wurzel p durch ihre Kinderliste; CONSOLIDATE(), wobei hohle Knoten durch ihre Kinder ersetzt werden)

  (Für Erhalt der Accounting-Invariante muss jeder der *grad(p)* Knoten der Kinderliste einen € erhalten; deswegen amortisierte Kosten $O(1 + grad(p))$)
- INSERT(x, H): $O(1)$ (erzeuge B_0 für x, lege 1 € drauf und hänge an Wurzelliste von H)
- DECREASE-KEY(x, H, k): $O(1)$ (wie auf vorheriger Folie beschrieben; braucht $O(1)$ tatsächliche Zeit plus höchstens 4 €)
- DELETE(x): ??$O(\log n)$?? (durch DECREASE-KEY(x, H, $-\infty$) gefolgt von EXTRACT-MIN(H))

Hollow Heaps, Maximaler Grad

Wie groß kann der Grad eines Knotens im Hollow Heap werden? Höchstens $O(\log n)$ weil:

Ein Knoten mit Grad $i > 1$ hat mindestens ein Kind vom Grad $i - 1$ und eines vom Grad $i - 2$.

K_i sei Mindestgröße eines Teilbaums mit Wurzelgrad i.

Es gilt $K_0 = 1, \quad K_1 = 2, \quad K_i > 1 + K_{i-1} + K_{i-2}$ für $i > 1$.

Lemma: $K_i \geq (3/2)^i$.

(Beweis durch Induktion; bessere Schranke mit Fibonacci-Zahlen)

$N = n + m$ sei Anzahl der Knoten in Hollow Heap, mit n Anzahl der Stücke, m Anzahl der DECREASE-KEY Operationen seit letztem Rebuild. Das entfernt hohle Knoten. Mache so ein Rebuild, wenn $m = n/2$. Mit zwei zusätzlichen Euros pro DECREASE-KEY kostet das Rebuild amortisiert nichts.

$3n/2 \geq n + m = N \geq (3/2)^i$ also $\boxed{i \leq 1 + \log_{3/2} n = O(\log n).}$