Vorlesung Datenstrukturen

Prioritätswarteschlangen

Maike Buchin

18. und 23.5.2017

Prioritätswarteschlange

Häufiges Szenario: dynamische Menge von Objekten mit Prioritäten, z.B. Aufgaben, Prozesse, in der immer das Objekt mit höchster Priorität gefunden werden soll.

Gesucht: Datenstruktur zum effizienten Abfragen und Modifikation

Formal: Prioritätswarteschlagen (engl. priority queue) verwalten Mengen Q von Einträgen mit Schlüsseln unter den Operationen

- Q.build$(e_1, \ldots, e_n)$
- Q.insert(e)
- Q.min
- Q.deleteMin

optional:

- Q.decreaseKey(h, k)
- Q.remove(h)
- Q.merge(Q')

Wie gut lässt sich dies mit Listen oder Arrays implementieren?

6.1 Heaps (Binärheaps)

Ein *heap* (dt. Halde oder Haufen) ist ein linksvollständiger Binärbaum mit der *(Heap-)Eigenschaft*: Schlüssel sind entlang jedes Weges von einem Blatt zur Wurzel schwach monoton fallend.
Äquivalent: Der Schlüssel eines jede Knoten (ausser der Wurzel) ist nicht kleier als der seines Elternknoten.

Darstellung: Wir speichern einen Heap in einem Array $h[1 \ldots n]$, wobei die Wurzel in $h[1]$ steht, left(i) in $h[2i]$, right(i) in $h[2i+1]$, und parent(i) in $h[\lfloor i/2 \rfloor]$ für $i \geq 2$,
Heap-Eigenschaft: für $2 \leq j \leq n$ gilt $h[\lfloor j/2 \rfloor] \leq h[j]$

Folgerung: $h[1]$ enthält minimalen Schlüssel

Anmerkung: sortiertes Array ist ebenfalls ein heap, aber mit stärkerer Bedingung $h[j-1] \geq h[j]$.
Die größere Flexibilität von Heaps erlaubt effizientes Einfügen und Löschen des Minimum.

6.1 Heaps (Binärheaps)

RUB

Einfügen und Löschen des Minimum

Idee: Einfügen am Ende und gelöschtes Minimum durch letzten Eintrag ersetzen. Dann die Heap-Eigenschaft entlang eines Weges zur Wurzel bzw. Blatt wieder herstellen.

Procedure *insert(e : Element)*
  assert $n < w$
  $n{+}{+};\ h[n] := e$
  $\mathit{siftUp}(n)$

Procedure $\mathit{siftUp}(j : \mathbb{N})$
  assert nur $h[j]$ ist eventuell zu klein
  if $j = 1 \vee h[\lfloor j/2 \rfloor] \leq h[j]$ **then return**
  $\mathit{swap}(h[j], h[\lfloor j/2 \rfloor])$
  assert nur $h[\lfloor j/2 \rfloor]$ ist eventuell zu klein
  $\mathit{siftUp}(\lfloor j/2 \rfloor)$
  assert h ist Heap

6.1 Heaps (Binärheaps)

Einfügen und Löschen des Minimum

Function *deleteMin* : *Element*

  assert $n > 0$
  $result = h[1] : Element$
  $h[1] := h[n]; \quad n{-}{-}$
  $siftDown(1)$
  return *result*

Procedure $siftDown(j : \mathbb{N})$

  assert nur $h[j]$ ist eventuell zu groß
  if $2j \leq n$ **then** // j ist kein Blatt
    if $2j+1 > n \vee h[2j] \leq h[2j+1]$ **then** $m := 2j$ **else** $m := 2j+1$
    assert der Schlüssel im Geschwisterknoten von m,
      falls ein solcher existiert, ist nicht größer als der in m
    if $h[j] > h[m]$ **then** // $h[j]$ ist zu groß
      $swap(h[j], h[m])$
      assert nur $h[m]$ ist eventuell zu groß
      $siftDown(m)$
  assert h ist Heap

6.1 Heaps (Binärheaps)

Einfügen und Löschen des Minimum

Korrektheit: von Insert und delMinimum folgen aus Korrektheit von SiftUp, SiftDown

Laufzeit:

- von SiftUp ist $O(\text{depth } j)$
- von SiftDown ist $O(\text{height } j)$
- von Insert und delMinimum ist $O(\log n)$

Build for Heaps

RUB

Ein Heap auf n Elementen lässt sich effizient bottom-up aufbauen.

Procedure *buildHeapBackwards*
  for $j := \lfloor n/2 \rfloor$ **downto** 1 **do** *siftDown*(j)

Korrektheit: folgt aus Korrektheit des SiftDown

Laufzeit: $\mathrm{O}\left(\sum_{0 \leq \ell < k} 2^{\ell}(k-\ell)\right) = \mathrm{O}\left(2^k \sum_{0 \leq \ell < k} \frac{k-\ell}{2^{k-\ell}}\right) = \mathrm{O}\left(2^k \sum_{j \geq 1} \frac{j}{2^j}\right) = \mathrm{O}(n)$

HeapSort

RUB

Idee:

- baue Heap
- lösche sukzessive das Minimum

Laufzeit: $\sum_{i=2}^{n} O(\log i) = \Theta(n \log n)$

Korrektheit: folgt aus Schleifeninvariante:
$P(k)$: $h[k+1 \ldots n]$ enthält die $n-k$ kleinsten Elemente sortiert
$h[1 \ldots k]$ ist ein heap auf den k größten Elemente

Anmerkung: heapSort ist inplace

Addressierbare Binärheaps [Aufgabe 6.7]

RUB

Binärheaps implementieren Prioritätswarteschlangen mit:

- build(), min() in $O(1)$
- deleteMin(), insert(e) in $O(\log n)$
- build($e_1, \ldots, e_n$) in $O(n)$

Binärheaps erlauben $O(1)$ Zugriff auf $h[i]$ aber unterstützen keine Griffe auf die Elemente in h.

Ein Ausweg: Verwende zwei Zeiger zwischen Elementen und ihrer Position im Heap.

Die Laufzeiten der Operationen verändern sich dadurch asymptotisch nicht, aber decreaseKey und remove lassen sich in $O(\log n)$ Zeit realisieren.

Die Operation **merge** lässt sich mit Binärheaps allerdings nur in $O(n)$ Zeit realisieren.

6.2 Addressierbare Prioritätswarteschlangen

RUB

Ziel: effiziente(re) Implementierung aller Operationen durch eine flexiblere Struktur als Binärheaps.

- links-vollständiger Binärbaum wird ersetzt durch einen Wald
- Einträge werden gespeichert in Heapknoten mit fester Position im Speicher
- Baumstruktur wird über Zeiger zwischen Knoten dargestellt

Pairing Heaps – allgemein

Ein Pairing Heap ist ein heap-geordneter Wald (d.h. eine Menge von Heaps). Es wird die Menge aller Wurzel sowie ein Zeiger *minPtr* auf die minimale Wurzel abgespeichert.

Drei grundlegende Operationen:

- *Hinzufügen* eines Baumes
- *Zusammenfügen* von zwei Bäumen
- *Herausschneiden* eines Unterbaums

Pairing Heaps – allgemein

RUB

Ein Pairing Heap ist ein heap-geordneter Wald (d.h. eine Menge von Heaps). Es wird die Menge aller Wurzel sowie ein Zeiger *minPtr* auf die minimale Wurzel abgespeichert.

Damit lassen sich die Operationen der PWS umsetzen:

- **insert:** erzeug neuen heap mit nur einem Knoten und füge diesen dem Wald hinzu
- **build**$(e_1, \ldots, e_n)$: n inserts
- **delMin:** entferne Wurzel, auf die *minPtr* zeigt. Füge deren Kinder als neue Wurzel hinzu und finde neues Minimum. Rebalanciere für Effizienz.
- **decreaseKey(h,k):** schneide Unterbaum an h heraus und füge als neue Wurzel hinzu; strukturiere ggfs. um.
- **remove**(h): decreaseKey$(h,$ min-1$)$; deleteMin;
- **merge:** vereinige Wurzelmengen und vergleiche MinPtr

Pairing Heaps – speziell

RUB

Wir betrachten die Variante von Pairing Heaps, die nur aus einem Baum bestehen. Rebalancieren verwendet: combine(Wurzel1, Wurzel2): hänge kleinere an größere Wurzel

Rebalancieren nach delMin:

- $h_1 =$ combine $(r_1, r_2), \ldots, h_m =$ combine (r_{m-1}, r_m), mit $m = k/2$ falls k gerade $O(k)$
- combine $(h_1,$ combine $(h_2,$ combine$(\ldots$ combine$(h_{m-1}, h_m))))$

Rebalancieren nach insert, decreaseKey, merge:

- es entstehen zwei Bäume, die mit einem combine zusammengefügt werden $O(1)$

Analyse: mit noch geschickterer Rebalancierung (auch nach decreaseKey) kann man zeigen, dass deleteMin amortisiert nur $O(\log n)$ kostet; decreaseKey kostet dann aber mehr als $O(1)$