Notizen zu Algorithmen II

Jens Ochsenmeier

16. Februar 2018

Inhaltsverzeichnis

1

Fortgeschrittene Datenstrukturen

Wir werden uns in diesem Kapitel mit Prioritätslisten beschäftigen. Es gibt noch viele weitere fortgeschrittene Datenstrukturen, z.B.

- monotone ganzzahlige Prioritätslisten (später im Kapitel "kürzeste Wege")
- perfektes Hashing
- Suchbäume mit fortgeschrittenen Operationen
- externe Prioritätslisten (später im Kapitel "Externe Algorithmen")
- Geometrische Datenstrukturen (siehe Kapitel "Geometrische Algorithmen")

1.1 Adressierbare Prioritätslisten

Eine **adressierbare Prioritätsliste** muss folgene Funktionen implementieren:

BUILD($\{e_1, \ldots, e_n\}$)	$M \coloneqq \{e_1, \ldots, e_n\}$
SIZE	**return** $\lvert M \rvert$
INSERT(e)	$M \coloneqq M \cup \{e\}$
MIN	**return** $\min M$
DELETEMIN	$e \coloneqq \min$; $M \coloneqq M \setminus \{e\}$; **return** e
REMOVE(h : Handle)	$e \coloneqq h$; $M \coloneqq M \setminus \{e\}$; **return** e
DECREASEKEY(h : Handle, k : Key)	key(h) $\coloneqq k$
MERGE(M')	$M \coloneqq M \cup M'$

Adressierbare Prioritätslisten haben viele Anwendungen, beispielsweise im *Dijkstra-Algorithmus* für kürzeste Wege oder in der Graphpartitionierung. Allgemein lassen sich adressierbare Prioritätslisten gut bei Greedy-Algorithmen verwenden, bei denen sich die Prioritäten (begrenzt) ändern.

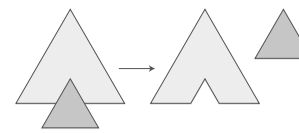
Datenstruktur

Als grundlegende Datenstruktur wird ein Wald heap-geordneter Bäume verwendet. Hier wird also der Binary Heap verallgemeinert

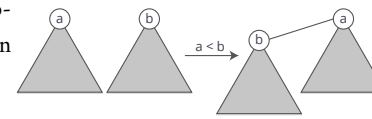
- Baum → Wald
- zwei Kindknoten → beliebig viele Kindknoten

Wir verwenden folgende grundlegende Operationen zur Bearbeitung solcher Wälder:

- **cut**: Teilbaum ausschneiden und als neuen Baum speichern



- **link**: Baum 2 mit größerem Wurzelknoten als Baum 1 an Baum 1 als Kindknoten des Wurzelknotens anhängen



- **union**: union(a, b) = link$(\min(a, b), \max(a, b))$

Dijkstras Algorithmus

Dijkstras Algorithmus kann die Distanz zwischen einem Startknoten s und jedem anderen Knoten des Graphen berechnen.

```
DIJKSTRA(s : Node, T : Tree)
// Initialisieren: Distanz zu jedem Knoten ist ∞, zu Startknoten 0.
d = ⟨∞, ..., ∞⟩
d[s] = 0
// Startknoten zu PQ hinzufügen.
Q.insert(s)
```

- $d = \langle \infty, \ldots, \infty \rangle$

  Zu Beginn ist die Distanz zu jedem Knoten ∞.

- parent$[s] = s, d(s) = 0$

  Der Startknoten wird initialisiert.

- Q.insert(s)

  Prioritätsliste wird mit Startknoten initialisiert.

1.2 Pairing Heaps

Pairing Heaps müssen folgende Funktionen implementieren:

```
INSERTITEM(h : Handle)
  newTree(h)
NEWTREE(h : Handle)
  forest := forest ∪ {h}
  if *h < min then minPtr := h
DECREASEKEY(h : Handle, k : Key)
  key(h) := k
  if h not a root then cut(h) else updateMinPtr(h)
DELETEMIN() : Handle
  m := minPtr
  forest := forest \ {m}
  foreach child h of m do newTree(h)
  pairwiseRootUnion()
  updateMinPtr()
  return m
PAIRWISEROOTUNION()
  // see picture
MERGE(o : AdressablePQ)
  if *minPtr > *(o.minPtr) then minPtr := o.minPtr
  forest := forest ∪ o.forest
  o.forest := ∅
```

Einige Funktionalitäten lassen sich gut veranschaulichen:

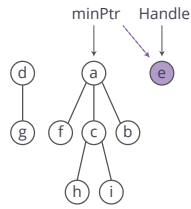


Abbildung 1.1. `newTree`: Element e wird hinzugefügt (ggf. auch ein ganzer Baum) und — falls nötig — der `minPtr` angepasst.

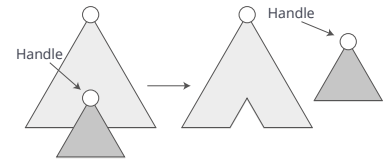


Abbildung 1.2. `decreaseKey`: Durch Herabsetzen des Keys wird eventuell die Heap-Eigenschaft verletzt, deswegen wird der Teilbaum ausgeschnitten und als neuer Baum hinzugefügt.

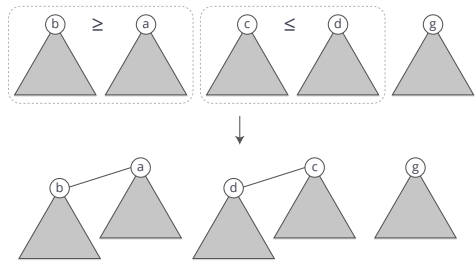


Abbildung 1.3. `pairwiseRootUnion` fügt jeweils zwei Bäume des Waldes zu einem größeren Baum zusammen, indem die beiden Wurzeln miteinander verglichen werden und eine der beiden Wurzeln Kindknoten der anderen wird.

Repräsentation

Meistens speichert man Pairing Heaps als doppelt verkettete Liste der Wurzeln. Die Baum-Items können beispielsweise folgendermaßen gespeichert werden:

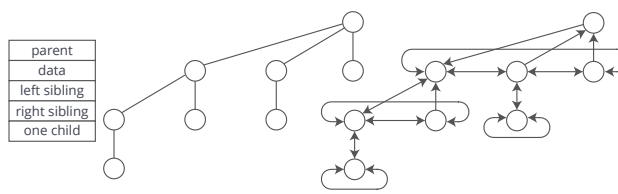


Abbildung 1.4. Speichern der Baum-Items. Man kann hier noch Speicherplatz einsparen, indem man *left sibling* und *parent* zusammenfasst. Allerdings muss man dann alle Geschwisterknoten traversieren, damit man zum Elternknoten kommen kann, da nur der linkeste Kindknoten den Elternknoten speichert.

Analyse

- `insert` und `deleteMin` gehen in $O(1)$.
- `deleteMin` und `remove` gehen jeweils in $O(\log n)$ amortisiert.
- `decreaseKey` ist schwieriger zu analysieren, geht aber amortisiert in $O(\log \log n) \leq T \leq O(\log n)$ und ist in der Praxis sehr schnell.

Wir werden als nächstes *Fibonacci-Heaps* verwenden, um noch mehr Leistung rauszukitzeln.

1.3 Fibonacci-Heaps

Mithilfe von Fibonacci-Heaps erhalten wir eine amortisierte Komplexität von $O(\log n)$ für `deleteMin` und $O(1)$ für alle anderen Operationen.
Fibonacci-Heaps speichern ein paar Zusatzinformationen pro Knoten ab, wodurch neue Hilfsfunktionen kreiert werden könnnen, die diese Beschleunigung ermöglichen:

- **Rank** eines Knotens: Anzahl direkter Kinder
- **Mark**: Knoten, die ein Kind verloren haben, werden markiert
- **Vereinigung nach Rank**: Union nur für gleichrangige Wurzeln
- **Kaskadierende Schnitte**: Knoten, die beide markiert sind (also ein Kind verloren haben), werden geschnitten

Repräsentation

Die Repräsentation ist analog zu der von Pairing Heaps, die Wurzeln werden wieder als doppelt verkettete Liste gespeichert und die Baum-Items als Parameterliste:

parent
data, *rank*, *mark*
left sibling
right sibling
one child

Funktionalität

`insert` und `merge` werden wie gehabt implementiert. `decreaseKey` verwendet die neue `cascadingCut`-Methode und `deleteMin` Union-by-Rank. Wir beschleunigen Union-by-Rank, indem wir ein Feld pro Rank bereitstellen und Knoten in diese

Felder eintragen. Sollte das passende Feld bereits belegt sein, so wird der neue Knoten mit dem Knoten, der sich im Feld befindet gelinkt und entsprechend verschoben.

```
DELETEMIN() : Handle
  m := minPtr
  forest := forest \ {m}
  foreach child h of m do newTree(h)
  while ∃ a, b ∈ forest : rank(a) ≡ rank(b) do
    union(a, b)
  updateMinPtr()
  return m
DECREASEKEY(h : Handle, k : Key)
  key(h) := k
  cascadingCut(h)
CASCADINGCUTh : Handle
  assert h is not a root
  p := parent(h)
  unmark(h)
  cut(h)
  if p is marked then
    cascadingCut(p)
  else mark(p)
```

Eine amortisierte Analyse von `deleteMin` ergibt $\Omega(\log n)$ für vergleichsbasiertes `deleteMin`.

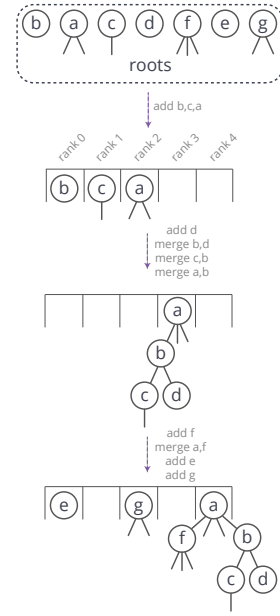


Abbildung 1.5. Fast Union-by-Rank.