Algorithmen und Datenstrukturen II

4. Hilfsstrukturen

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik
Hochschule München

Letzte Änderung: 06.05.2020 16:40

Inhaltsverzeichnis

Vorrangswarteschlangen (*Priority Queues*)

- Datenstruktur zur Speicherung einer Menge von Elementen, für die eine Ordnung definiert ist
- *engl.* **Priority Queue**
- mit folgenden Operationen:
  - Initialisieren der leeren Struktur
  - Einfügen eines Elements ($Insert$)
  - Minimum suchen ($AccessMin$)

  - Minimum entfernen ($DeleteMin$)

- auch bekannt als **Heap**
- kann z.B. mit der Datenstruktur `Heap` oder mit balancierten Bäumen implementiert werden

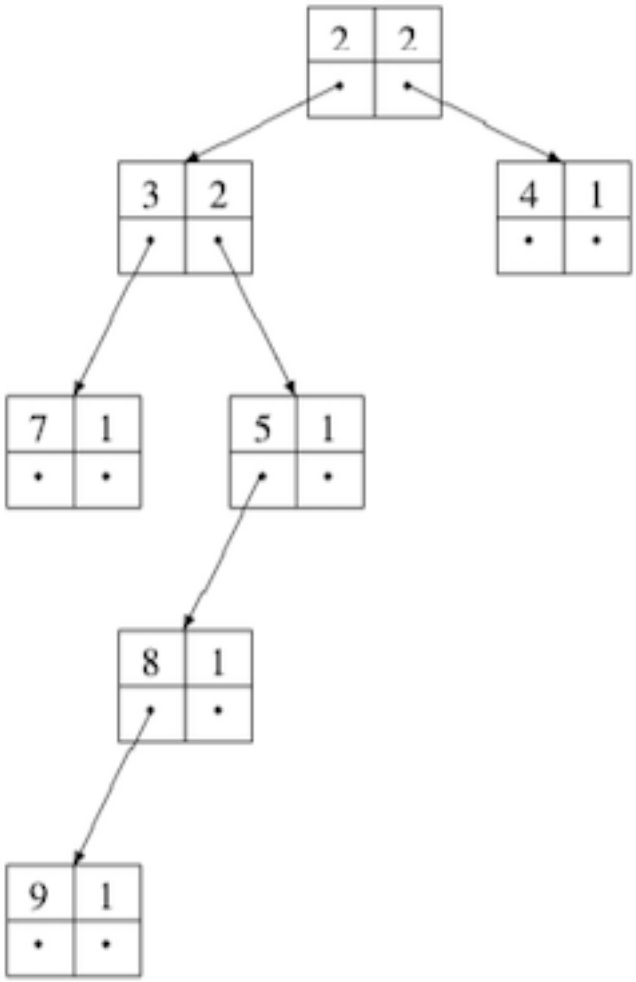
Implementierung durch Linksbäume (*Leftish Trees*)

- bei balancierten Bäumen hat **jeder** Pfad von der Wurzel zu einem Blatt eine Länge der Größenordnung $\mathcal{O}(logN)$
- zur Implementierung von Priority Queues reichen geringere Anforderungen:
  - Schlüsselwerte der Nachfolger müssen stets größer als der Schlüssel des Vorgängers sein
  - es muss wenigstens einen Pfad von der Wurzel zu einem Blatt mit Länge $\mathcal{O}(logN)$ geben
- Einfügen und Verschmelzen zweier Bäume entlang dieses Pfades dann genauso gut wie bei balancierten Bäumen
- ein binärer Baum heißt **Linksbaum**, wenn gilt: jeder Knoten enthält neben Schlüssel und Zeigern auf die Nachfolger, ein Distanzfeld, in dem die Entfernung zum nächstgelegenen Blatt festgehalten wird
- für jeden inneren Knoten `p` muss gelten:

  `p.key < p.left.key` und `p.key < p.right.key`
- außerdem muss für jeden inneren Knoten `p` gelten:

  `p.dist = 1 + min{p.left.dist, p.right.dist}` und `p.left.dist >= p.right.dist`
- d.h. es gilt: `p.dist = 1 + p.right.dist`
- also ist der kürzestes Pfad von der Wurzel zu einem Blatt immer der Pfad zum rechtesten Blatt

Beispiel und Code



- in Go z.B.

```
type leftishTree *node
type node struct {
  key         int
  dist        int
  left, right leftishTree
  info        interface{}
}
```

Alle Operationen auf Verschmelzen zurück führen

- Zugriff auf Minimum (Wurzel) in konstanter Zeit
- da jeder Teilbaum eines Linksbaumes wieder ein Linksbaum ist, können alle anderen Operationen auf das Verschmelzen zweier Linksbäume zurück geführt werden

- *Einfügen*: Verschmelzen des Linksbaumes mit einem der nur den neuen Schlüssel enthält
- *Entfernen des Minimums*: Entfernen der Wurzel und Verschmelzen der beiden Teilbäume
- *Entfernen eines beliebigen inneren Knotens* `p`:
  - Linksbaum zerfällt in drei Teilbäume: Die Nachfolger von `p` und der Baum `A` oberhalb von `p`
  - in `A` wird `p` durch Blatt ersetzt und eventuell mit dem Geschwisterknoten vertauscht (um Linksbaumeigenschaft wieder herzustellen)
  - auf dem Pfad von dem neuen Blatt zur Wurzel müssen die Distanzen angepasst werden
  - dann werden die drei (Links-)Bäume zu einem neuen verschmolzen
- *Herabsetzen eines Schlüssels*: Entfernen und Einfügen mit neuem Schlüsselwert
- Einfügen und Entfernen in $\mathcal{O}(logN)$ Schritten, wenn Verschmelzen entsprechend effizient
- **Achtung**: Das Adjustieren der Distanzen und gegebenenfalls Vertauschen von Teilbäumen kann im schlechtesten Fall $\Omega(N)$ Schritte erfordern

Verschmelzen (Merge)

```
func Merge(a, b LeftishTree) LeftishTree {
    if a == nil {
        return b
    }

    if b == nil {
        return a
    }

    if a.Key > b.Key {
        a, b = b, a
    } // now a.key <= b.key

    a.Right = Merge(a.Right, b)

    if a.Left == nil && a.Right != nil ||
        a.Left != nil && a.Right != nil &&
            a.Right.Dist > a.Left.Dist {
        a.Right, a.Left = a.Left, a.Right
    }

    if a.Right == nil {
        a.Dist = 1
```

```
    } else {
        a.Dist = a.Right.Dist + 1
    }

    return a
}
```

Union-Find-Strukturen

- in vielen Algorithmen ist eine Teilaufgabe eine Menge von Objekten in Äquivalenzklassen einzuteilen
- bei den Union-Find-Strukturen beginnen wir mit einer sehr feinen Einteilung
- diese wird durch sukzessive Vereinigung der Mengen vergröbert

Operationen

- $MakeSet(e, i)$
  - erzeugt eine neue Menge i mit einzigem Element e
  - e darf in keiner anderen Menge der Kollektion vorkommen
- $Find(x)$
  - liefert den Namen der Menge, die x enthält
- $Union(i, j, k)$
  - vereinigt die Mengen i und j zu einer neuen Menge mit dem Namen k
  - i und j werden entfernt
  - i und j müssen verschieden sein
- statt einem Namen, kann man ein Element aus der Menge als Repräsentant auswählen
  - heißt **kanonisches Element**
  - dann reicht $MakeSet(e)$, $Find(x)$ und $Union(e, f)$

Beispiel: Kruskal MST-Algorithmus

- wir beginnen mit der Kollektion K von einelementigen Knotenmengen
- wir vereinigen zwei Mengen, wenn Sie durch eine Kante minimalen Gewichts miteinander verbunden werden können
- das Verfahren endet, wenn K nur noch eine Menge enthält

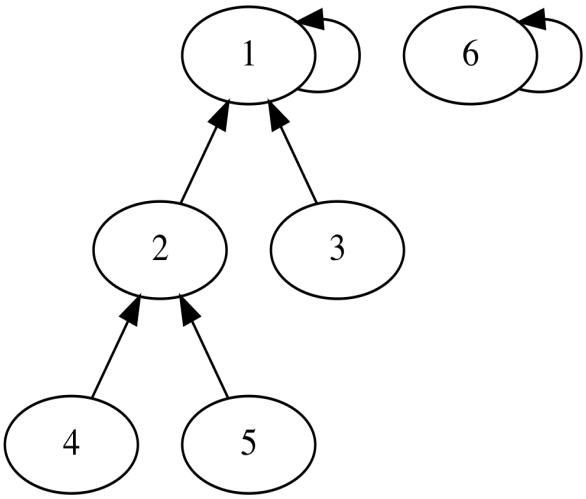
Mögliche Lösung des Union-Find-Problems

- jede Menge der Kollektion wird durch (nichtsortierten) Baum beliebiger Ordnung repräsentiert
  - die Knoten des Baumes sind die Elemente der Mengen
  - die Wurzel ist das kanonische Element
- jeder Knoten im Baum enthält einen Zeiger auf seinen Vorgänger
- Speicherung z.B. als Map
  - Schlüssel ist Knotenbezeichner
  - Wert ist Bezeichner des Vorgängers
  - vereinfacht als Array, wenn nur Zahlen 1,…,N

Operationen

```
type Collection map[int]int
func (k Collection) MakeSet(x int) {
    k[x] = x
}
func (k Collection) Union(e, f int) {
    k[f] = e
}
func (k Collection) Find(x int) int {
    y := x
    for ; k[y] != y; y = k[y] {
    }
    return y
}
```

Beispiel




```
k := NewCollection()
for i := 1; i <= 6; i++ {
  k.MakeSet(i)
}
k.Union(1, 2)
k.Union(1, 3)
k.Union(2, 4)
k.Union(2, 5)
```

- *MakeSet* und *Union* sind in konstanter Zeit ausführbar
- *Find* benötigt im schlechtesten Fall $\Omega(N)$ Schritte (entarteter Baum)

Verbessertes *Union*

- damit der Baum nicht entartet, den "kleineren" Baum unter den "größeren" hängen
- einfache Idee: Fügen Größe = Anzahl der Elemente im Baum zur Wurzel (bzw. zu allen Einträgen) hinzu
- Beispiel

```
type ElementInfo struct {
  parent int
  size   uint
}
type Collection map[int]*ElementInfo
```

```
func (k Collection) MakeSet(x int) {
    k[x] = &ElementInfo{parent: x, size: 1}
```

```
}
func (k Collection) Union(e, f int) {
    if k[e].size < k[f].size {
        e, f = f, e
    }
    // e is bigger
    k[f].parent = e
    k[e].size += k[f].size
}

func (k Collection) Find(x int) int {
    y := x
    for ; k[y].parent != y; y = k[y].parent {
    }
    return y
}
```

- *Find* jetzt $\mathcal{O}(logN)$
- *Union* immer noch konstant