Algorithm Engineering 7
Standarddatenstrukturen

Karsten Weicker

F IMN, HTWK Leipzig

1. Dictionary
   - Splay-Trees
2. Prioritätswarteschlange
   - Binomial-Heaps
3. Mengen
4. Mengenpartitionen
   - Union-Find
5. Indexerstellung

Überblick

Dictionary

Eingabe

Menge von n Datenelementen, identifizierbar durch ein (oder mehrere) Schlüssel

Problembeschreibung

Aufbau und Unterhalt der Datenstruktur für effizientes Finden, Einfügen und Löschen von Elementen

Dictionary

Lösung: unsortierte verkettete Liste oder Feld

- bis höchstens 50–100 Elemente
- verkettete Strukturen können schlechte Cache-Performance haben
- interessant: selbstorganisierte Liste – bei Zugriff oder Einfügen Element an den Anfang der Liste

Lösung: sortierte verkettete Liste oder Feld

- sortierte verkettete Liste ist nutzlos (bis auf Vermeidung von Duplikaten)
- sortiertes Feld nur wenn wenige Einfüge- und Löschoperationen

Dictionary

Lösung: Hash-Tabelle

- für 100 – ca. 10 Millionen Elemente
- Hashing mit offener Adressierung: bessere Cache-Performance als Bucketing, evtl. Probleme bei hohem Load-Faktor
- Größe der Tabelle bei Bucketing: etwa Anzahl der Elemente, 30% mehr bei offenem Hashing
- Hash-Funktion gut gewählt? Statistik über Bucket-Verteilung betrachten

Dictionary

Lösung: binärer Suchbaum

- unbalanciert: nur gut bei zufälligen Einfügeoperationen
- balancierte Bäume: Rot-Schwarz-Bäume sind State-of-the-Art
- Splay-Bäume bei häufig identischen sequentiellen Zugriffen

Dictionary

Lösung: B-Baum

- falls Elemente nicht komplett in den Hauptspeicher passen (ab ca. 1 Million) – mit modernen Caches eher abgeschwächt
- viel Festplattenaktivität ist ein Indikator für B-Bäume

Splay-Trees

SPLAY(Element *wurzel*, Schlüssel *gesucht*)

Rückgabewert: neue Wurzel; Seiteneffekte

$\left.\begin{array}{l} \mathit{wurzel} \rightarrow \\ \mathit{rot} \leftarrow \end{array}\right\}$ SPLAY-R(*wurzel*, *gesucht*)

if *wurzel* $\neq$ **null** und *rot*

then **if** *gesucht* $<$ *wurzel.wert*

  then *wurzel* $\rightarrow$ ROTIERE-RECHTS(*wurzel*)

  else *wurzel* $\rightarrow$ ROTIERE-LINKS(*wurzel*)

return *wurzel*

Splay-R(Element *wurzel*, Schlüssel *gesucht*)

Rückgabewert: neue Wurzel, Info bzgl. der Rotation; Seiteneffekte
switch
case *wurzel* $=$ **null** :
  return null , **true**
case *wurzel* $\neq$ **null** und *wurzel.wert* $=$ *gesucht* :
  return *wurzel*, **false**
case *wurzel* $\neq$ **null** und *wurzel.wert* $<$ *gesucht* :
  $\left.\begin{array}{l} wurzel.links \rightarrow \\ rot \leftarrow \end{array}\right\}$ Splay-R(*wurzel.links*, *gesucht*)
  if *rot*
  then [*wurzel* $\rightarrow$ Splay-Rotation-Links(*wurzel*, *gesucht*)
  return *wurzel*, $\neg$*rot*
case *wurzel* $\neq$ **null** und *wurzel.wert* $>$ *gesucht* :
  $\left.\begin{array}{l} wurzel.rechts \rightarrow \\ rot \leftarrow \end{array}\right\}$ Splay-R(*wurzel.rechts*, *gesucht*)
  if *rot*
  then [*wurzel* $\rightarrow$ Splay-Rotation-Rechts(*wurzel*, *gesucht*)
  return *wurzel*, $\neg$*rot*

$\textsc{Splay-Rotation-Links}$(Knoten $e\ell$, Schlüssel *gesucht*)

Rückgabewert: neue Wurzel
switch
case $e\ell =$ **null** : **return null**
case $e\ell.links =$ **null** : **return** $e\ell$
case $e\ell.links \neq$ **null** und $gesucht < e\ell.links.wert$:
  $e\ell \rightarrow \textsc{Rotiere-Rechts}(e\ell)$
  $e\ell \rightarrow \textsc{Rotiere-Rechts}(e\ell)$
case $e\ell.links \neq$ **null** und $gesucht > e\ell.links.wert$:
  $e\ell.links \rightarrow \textsc{Rotiere-Links}(e\ell.links)$
  $e\ell \rightarrow \textsc{Rotiere-Rechts}(e\ell)$
return ℓ

$\textsc{Splay-Rotation-Rechts}(\text{Knoten } e\ell, \text{Schlüssel } \mathit{gesucht})$

Rückgabewert: neue Wurzel
switch
case $e\ell = \textbf{null}$: **return null**
case $e\ell.\mathit{rechts} = \textbf{null}$: **return** $e\ell$
case $e\ell.\mathit{rechts} \neq \textbf{null}$ und $\mathit{gesucht} > e\ell.\mathit{rechts}.\mathit{wert}$:
    $e\ell \rightarrow \textsc{Rotiere-Links}(e\ell)$
    $e\ell \rightarrow \textsc{Rotiere-Links}(e\ell)$
case $e\ell.\mathit{rechts} \neq \textbf{null}$ und $\mathit{gesucht} > e\ell.\mathit{rechts}.\mathit{wert}$:
    $e\ell.\mathit{rechts} \rightarrow \textsc{Rotiere-Rechts}(e\ell.\mathit{rechts})$
    $e\ell \rightarrow \textsc{Rotiere-Links}(e\ell)$
return ℓ

SUCHEN-SPLAY(Schlüssel *gesucht*)

```
Rückgabewert: gesuchte Daten bzw. Fehler falls nicht enthalte
anker → SPLAY(anker, gesucht)
if anker = null  oder anker.wert ≠ gesucht
then [ error "Element nicht gefunden"
else  [ return anker.daten
```

EINFÜGEN-SPLAY(Schlüssel *neuerWert*, Daten *nDat*)

```
Rückgabewert: nichts falls erfolgreich bzw. Fehler sonst
if anker = null
then [ anker → allokiere Knoten(neuerWert, nDat, null , null )
else ⌈ anker → SPLAY(anker, gesucht)
       switch
       case neuerWert = anker.wert : error "Element schon enthalten"
       case neuerWert < anker.wert :
            anker ← allokiere Knoten(neuerWert, nDat, anker.links, anke
       case neuerWert > anker.wert :
     ⌊      anker ← allokiere Knoten(neuerWert, nDat, anker, anker.rech
```

LÖSCHEN-SPLAY(Schlüssel *löschWert*)

Rückgabewert: nichts falls erfolgreich bzw. Fehler sonst
anker $\rightarrow$ SPLAY(*anker*, *löschWert*)
if *anker* = **null** oder *löschWert* $\neq$ *anker.wert*
then **error** "Element nicht gefunden"
else **if** *anker.links* = **null**
    then *anker* $\rightarrow$ *anker.rechts*
    else *neuerAnker* $\rightarrow$ SPLAY(*anker.links*, ∞)
        neuerAnker.rechts $\rightarrow$ *anker.rechts*
        anker $\rightarrow$ *neuerAnker*

Überblick

Prioritätswarteschlange

Eingabe

Menge an Datenelementen mit numerischen (oder total-geordneten) Schlüsseln

Problembeschreibung

Aufbau und Unterhalt der Datenstruktur für schnellen Zugriff auf das kleinste oder größte Element

Prioritätswarteschlange

Lösung: sortiertes Feld oder Liste

- gut bei Zugriff und Löschen auf das Minimum/Maximum
- schlecht bei nachträglichen Einfügeoperationen

Lösung: Heap

- alle ändernden Operationen in logarithmischer Zeit
- gut, wenn eine obere Grenze für die Elementzahl bekannt ist

Prioritätswarteschlange

Lösung: Bounded Height Priority Queue

- Schlüssel sind ganze Zahlen zwischen 1 und n
- Organisation: n verkettete Listen, Zeiger auf die nicht-leere Liste mit kleinstem/größtem Schlüssel
- gut in Graphalgorithmen für Verwaltung der Knoten sortiert nach ihrem Kantengrad
- beste Wahl für kleine, diskrete Schlüsselmengen

Prioritätswarteschlange

Lösung: binärer Suchbaum

- kleinstes Element – ganz links; größtes – ganz rechts
- sinnvoll bei weiteren benötigten Operationen
- sinnvoll bei unbegrenztem Schlüsselbereich und unbekannter Anzahl der Elemente
- beste Lösung: wenn Minimum und Maximum gefragt

Lösung: Fibonacci oder Pairing Heaps

- entworfen für viele Decrease-Key-Operationen
- sinnvoll für sehr große Berechnungen

Prioritätswarteschlangen – Vergleich

	Init(n)	In-sert	Get-Min	Del-Min	Decr.-Key
Feld	$\mathcal{O}(n)$	$\Theta(1)^*$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\Theta(1)$
sort. Feld	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	$\Theta(1)$	$\Theta(1)$	$\mathcal{O}(n)$
Heap	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\Theta(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Binom.H.					
Fib.H.					

	Delete	Merge	Dijkstra-Alg.
Feld	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\lvert V\rvert^2)$
sort. Feld	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\lvert E\rvert \cdot \lvert V\rvert)$
Heap	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}((\lvert V\rvert + \lvert E\rvert) \cdot \log \lvert V\rvert)$
Binom.H.			
Fib.H.			

* = amortisiert

Binomial-Heaps

Link(BH-Knoten Y, Z)

$Y.elter \leftarrow Z$

$Z.kind.lbruder \leftarrow Z$

$Y.rbruder \leftarrow Z.kind$

$Z.kind \leftarrow Y$

$Y.lbruder \leftarrow \textbf{null}$

$Z.grad \leftarrow Z.grad + 1$

BH-Merge(Binom Heaps H_1, H_2)

```
H ← vereinige Top-Listen von H1 und H2 mit steigendem Grad
if H = null
then [ return H
x ← H.kopf
nachf ← x.rbruder
while nachf ≠ null
do ⌈ if x.grad ≠ nachf.grad oder
               (nachf.rbruder ≠ null  und nachf.rbruder.grad = x.grad)
     then [ x ← nachf
     else ⌈ if x.key ≤ nachf.key
            then ⌈ x.rbruder ← nachf.rbruder
                   if nachf.rbruder ≠ null
                   then [ nachf.rbruder.lbruder ← x
                 ⌊ Link(nachf, x)
            else ⌈ if x.lbruder = null
                   then [ H.kopf ← nachf
                   else ⌈ nachf.lbruder ← x.lbruder
                        ⌊ x.prev.rbruder ← nachf
                   Link(x, nachf)
          ⌊      ⌊ x ← nachf
   ⌊ nachf ← x.rbruder
return H
```

Prioritätswarteschlangen – Vergleich

	Init(n)	In-sert	Get-Min	Del-Min	Decr.-Key
Feld	$\mathcal{O}(n)$	$\Theta(1)^*$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\Theta(1)$
Heap	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\Theta(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Binom.H.	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Fib.H.	$\mathcal{O}(n)$	$\Theta(1)$	$\Theta(1)$	$\mathcal{O}(\log n)^*$	$\Theta(1)^*$

	Delete	Merge	Dijkstra-Alg.
Feld	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\|V\|^2)$
Heap	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}((\|V\| + \|E\|) \cdot \log \|V\|)$
Binom.H.	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}((\|V\| + \|E\|) \cdot \log \|V\|)$
Fib.H.	$\mathcal{O}(\log n)^*$	$\Theta(1)$	$\mathcal{O}(\|V\| \log \|V\| + \|E\|)$

$*$ = amortisiert

Überblick

1. Dictionary
   - Splay-Trees
2. Prioritätswarteschlange
   - Binomial-Heaps
3. Mengen
4. Mengenpartitionen
   - Union-Find
5. Indexerstellung

Mengen

Eingabe

Auf einem Universum $U = \{u_1, \ldots, u_n\}$ sind unterschiedliche Teilmengen $S_1, \ldots, S_m$ möglich

Problembeschreibung

eine Teilmenge S_i speichern, sodass effizient

- $u_j \in S_i$ getestet werden kann,
- Vereinigung bzw. Schnitt mit S_k möglich ist und
- Elemente eingefügt oder gelöscht werden können.

Mengen

Lösung: Bit-Vektor

- Bit-Vektor der Länge n
- erstaunlich platzsparend auch für großes n
- Einfügen/Löschen durch Bit-Flip
- Vereinigung/Schnitt durch logisches Und/Oder
- Iterieren durch alle Elemente ist ineffizient für großes n

Mengen

Lösung: Dictionary-Datenstrukturen

- geht auch bei großem/unklaren U
- platz- und zeitsparender für kleine Teilmengen
- sortiertes Dictionary: effiziente Vereinigung/Schnitt durch synchrones Inorder-Traversieren

Mengen

Lösung: Bloom-Filter

- mit k Hash-Funktionen werden für jedes Element k Einträge in einem Bit-Array gesetzt
- Test: $u_j \in S_i \Leftrightarrow$ alle k Felder gesetzt (Vorsicht: false positives!)
- sehr platzsparend
- gut geeignet, wenn approximative Aussagen genügen

Überblick

Mengenpartitionen

Eingabe

eine Menge $S = \{s_1, \ldots, s_n\}$

Problembeschreibung

- Verwaltung der Einteilung von S in verschiedene Teilmengen
- effizienter Test auf Enthaltensein zweier Elemente in der gleichen Menge; sowie Vereinigung von Teilmengen

Mengenpartitionen

Lösung: mehrere Dictionaries

- jede Teilmenge in eigenem Dictionary
- leichtes Vereinigen
- Test auf Enthaltensein ist teuer

Lösung: Vektor

- in einem Vektor/Feld wird jedem Element die Nummer der Teilmenge zugeordnet
- Test auf Enthaltensein ist einfach
- Vereinigung zweier Teilmengen ist sehr teuer

Mengenpartitionen

Lösung: Dictionary mit Teilmengenattribut

- analog zum Vektor – nur in Dictionary

Lösung: Union-Find-Datenstruktur

- Wald mit Verzeigerung zu einem Vertreter jeder Menge
- insbesondere durch Pfadverkürzung beste Lösung
- Aufspalten von Teilmengen wird nicht unterstützt

Union-Find: Einfacher Ansatz

```
Erzeuge-Einelementige-Menge( Element x)
    knoten ← new Knoten(x)
    knoten.elter ← knoten
    return knoten

Finde-Vertreter( Knoten knoten)
    while knoten.elter ≠ knoten
    do [ knoten ← knoten.elter
    return knoten

Vereinige( Knoten k1, Knoten k2)
    wurzel1 ← Finde-Vertreter(k1)
    wurzel2 ← Finde-Vertreter(k2)
    wurzel1.elter ← wurzel2
```

Union-Find: Pfadverkürzung

ERZEUGE-EINELEMENTIGE-MENGE-S(Element x)

$knoten \leftarrow$ **new** $Knoten(x)$
$knoten.elter \leftarrow x$
$knoten.rang \leftarrow 0$
return $knoten$

VEREINIGE-S(Knoten k_1, Knoten k_2)

$wurzel_1 \leftarrow$ FINDE-VERTRETER-S(k_1)
$wurzel_2 \leftarrow$ FINDE-VERTRETER-S(k_2)
if $wurzel_1.rang > wurzel_2.rang$
then $wurzel_2.elter \leftarrow wurzel_1$
else $wurzel_1.elter \leftarrow wurzel_2$
    if $wurzel_1.rang = wurzel_2.rang$
    then $wurzel_2.rang \leftarrow wurzel_2.rang + 1$

Union-Find: Pfadverkürzung

FINDE-VERTRETER-S(Knoten *knoten*)

if *knoten.elter* $\neq$ *knoten*
then [*knoten.elter* $\leftarrow$ FINDE-VERTRETER-S(*knoten.elter*)
return *knoten.elter*

Überblick

Indexerstellung

Eingabe

ein statischer Text $s = s_1 \ldots s_n$

Problembeschreibung

- in dem Text sollen effizient Suchanfragen t – möglichst mit Laufzeit $\mathcal{O}(|t|)$ durchgeführt werden

Suffix-Tree

Lösung: Suffix-Bäume

- Suffix-Bäume speichern alle Suffixe der Wörter im Text
- Struktur: Baum als TRIE
- Konstruktion geht in $\mathcal{O}(|s|)$ (Algorithmus von McCreight)

$\textsc{Indexaufbau-Intuitiv}($ Zeichenkette $s = s_1 \dots s_n)$

```
T → leerer Baum
for i ← 1, ..., n
do  füge Suffix s_i ... s_n in T ein
```