Grundlagen Algorithmen und Datenstrukturen
TUM Sommersemester 2011 (2)
Dozent: Hanjo Täubig

Janosch Maier

3. August 2011

Inhaltsverzeichnis

1 Sortieren

1.1 Externes Sortieren

- Hauptspeicher M, externer Speicher Blockgröße B ($B|M$), $3B \leq M \rightarrow$ Blocktransvers minimieren durch Mergesort.
- Runs der Größe M in Speicher Laden, in-place sortieren, zurück schreiben.
- Sortierte Runs mergen. 2 Eingabeblöcke (nachladen, wenn leer), ein Ausgabebeblock (schreiben, wenn voll) benötigt.
- **Block-Transfers:** $(\frac{2n}{B})(1 + \lceil \log_{\frac{M}{B}}(\frac{n}{M}) \rceil)$
- Nur ein Merge nötig, wenn $n \leq \frac{M^2}{B}$

1.2 Heap Sort

Bilde Binären Heap (2.1) $\Rightarrow$ kleinstes Element steht vorne im Array. Tausch mit letztem Element. Bilde erneuten Heap ohne letztes Element, …

Analyse Laufzeit $\in \mathcal{O}(n \log n)$, in-place, nicht stabil

2 Prioritätswarteschlangen

- Element $e \in M$, Menge der Elemente mit Priorität $prio(e)$
- Operationen build, insert, min, deleteMin

Adressierbare Prioritätswarteschlangen

- insert (gibt Handle zurück), remove(Handle), decreaseKey(Handle, newPriority), merge

	Unsortierte Liste:	Sortierte Liste
build	$\in \mathcal{O}(n)$	$\in \mathcal{O}(n \log n)$
insert	$\in \mathcal{O}(1)$	$\in \mathcal{O}(n)$
min / deleteMin	$\in \mathcal{O}(n)$	$\in \mathcal{O}(1)$

2.1 Binärer Heap

Fast vollständiger Binärbaum, Prio(Elternknoten) $\leq$ Prio(Kindknoten) (Heap-Bedingung). Als Array darstellbar. Hierachisch von oben nach unten, und links nach rechts sortiert.

Bei Sift-Up wird ein Element so lange nach Oben verschonben, bis die Heap-Bedingung erfüllt ist. Bei Sift-Down, wird ein Element so lange mit dem Kleineren Kindknoten vertauscht, bis die Heap-Bedingung wieder erfüllt ist.

2.1.1

Laufzeiten min $\in \mathcal{O}(1)$; insert $\in \mathcal{O}(\log n)$; deleteMin $\in \mathcal{O}(\log n)$; build $\in \mathcal{O}(n)$, merge $\in \Theta(n)$

Ungeeignet als Adressierbare PQ, da Array Indizes nicht als Handles verwendet werden können (Ändern sich bei Operationen)

2.2 Binomial Heaps

Verkettete Liste aus Binomialbäumen; Rang der Bäume sind paarweise verschieden; Zeiger auf Wurzel mit minimaler Priorität

2.2.1 Binomialbaum

- Ordnung $0 \Rightarrow 1$ Knoten
- Ordnung $k \Rightarrow$ Wurzel mit Grad k, Kinder der Ordnung $k-1, k-2, ..., 0$

Heap Bedingung: Prio(Elternknoten) $\leq$ Prio(Kindknoten)

Baum von Rang r

- r ist Höhe (max. Pfadlänge von der Wurzel aus), maximaler Grad (der Wurzel)
- $\begin{pmatrix} r \\ l \end{pmatrix}$ Knoten auf Level l
- Insgesamt 2^r Knoten
- Wurzel entfernen $\Rightarrow$ r-Bäume von Rang $0, ..., r-1$

2.2.2 Operationen

Merge: Wie Addition von Binärzahlen

2.2.3 Laufzeiten

min $\in \mathcal{O}(1)$; merge, insert, deleteMin (Wurzel löschen + merge), decreaseKey (sift-up), remove (prio $= -\infty$, sift-up, deleteMin) $\in \mathcal{O}(\log n)$.

2.3 Sonstige Prioritätswarteschlangen

2.3.1 Fibonacci-Heap

min, insert, merge $\in \mathcal{O}(1)$ (worst case); decreaseKey $\in \mathcal{O}(1)$ (amortisiert); deleteMin, remove $\in \mathcal{O}(\log n)$ (amortisiert)

2.3.2 Ganzzahlige Werte

decreaseKey, insert $\in \mathcal{O}(1)$; deleteMin $\in \mathcal{O}(\log \log n)$ möglich.

3 Suchstrukturen

Wörterbuch find(k) gibt Element mit Key == k (oder Null) zurück. Hashing!

Suchstruktur Locate(k) gibt erstes Element mit (Key $\geq k$) zurück. Liste mit Navigationsstruktur, dass $\mathcal{O}(\text{locate})(n) < \Theta(n)$

3.1 Binärer Suchbaum

$k \in T$ ist Knoten, T_1 ist linker Kindbaum, T_2 ist rechter Kindbaum von k.

- $k_1 \leq k < k_2; \forall k_1 \in T_1, k_2 \in T_2$
- $grad(k) \leq 2$
- Jedes Element taucht genau einmal im Baum auf

3.1.1 Operationen

Insert locate; Element in Liste einfügen; neues Suchbaumblatt erstellen

Remove locate; Element aus Liste löschen; Suchknoten löschen; evtl. Vatersuchknoten anpassen

3.1.2 Laufzeiten

Entartung möglich $\Rightarrow$ im worst case: locate $\Theta(n)$

3.2 AVL-Baum

Beschränkung der Höhenunterschiede für Teilbäume auf $[-1, 0, +1]$ zur Balancierung des Baumes. Vermerkt für jeden Elternknoten.

Worst Case: Fibonacci-Baum (Anzahl innerer Knoten sind Fibonacci Zahlen)

$$F_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right] \tag{1}$$

Fibonacci-Baum der Stufe $h+1$ = Wurzel + Kinder der Stufe h und $h-1$. $\Rightarrow$ Anzahl der Elemente quadratisch von Höhe abhängig. $\Rightarrow$ Höhe logarithmisch von Anzahl der Elemente abhängig.

3.2.1 Operationen

Insert und remove können Balance-Bedingung verletzen $\Rightarrow$ Umsortierung durch Rotieren. Doppelrotation möglich und teilweise nötig.

3.2.2 Laufzeiten

locate, insert, remove $\in \mathcal{O}(\log n)$

3.3 (a,b)-Baum

$d(v)$ ist Anzahl der Kinder eines Knotens, $t(v)$ ist Tiefe eines Knotens, s Anzahl der Elemente des Baumes

- Alle Blätter in der selben Tiefe (Form-Invariante)
- Für alle Internen Knoten gilt $a \leq d(v) \leq b$, Wurzel (außer, wenn Baum nur ein Blatt) $2 \leq d(v) \leq b$

Es gilt: $b \geq 2a - 1, a \geq 2$ wichtig für verschmelzen (nicht größer als b), aufspalten (nicht kleiner, als a)

Tiefe des Baumes $\leq 1 + \lfloor \log_a \frac{n+1}{2} \rfloor$

3.3.1 Operationen

Einfügen locate() $\rightarrow$ Einfügen in Liste, sofern Elternknoten Platz für Suchelement hat, dieses einfügen; Sonst: Knoten Teilen, mittleres Element in Elternknoten ziehen.

Löschen find() $\rightarrow$ Element Löschen. Wenn Invariante nicht mehr gültig, Kinder von Benachbarten Knoten Klauen, wenn nicht mehr möglich $\rightarrow$ mit Nachbarknoten verschmelzen.

Min / Max Gebe erstes / letzte Element der Liste aus $\in \mathcal{O}(1)$.

Bereichsanfragen Anfrage: $[x, y]$. Locate(x); Über liste Laufen, bis Element $>$ y. $\in \mathcal{O}(n)$.

Konkatanation O.b.d.a. Schlüssel in T_1 kleiner als in T_2. Lösche ∞ von T_1, verfahre wie bei remove. Wurzel des kleineren Baumes mit Knoten des anderen Baumes auf selbem Level verschmelzen, verfahre wie bei insert.

Wenn Höhen bekannt: $\in \mathcal{O}(1 + |h_1 - h_2|)$, sonst $\in \mathcal{O}(1 + \log(\max\{s_1, s_2\}))$

Aufspalten Auspaltung bei Blatt y: Aufspalten auf Pfad von Wurzel zu y. Mehrere linke / rechte Teilbäume, die zu zwei Teilbäumen konkataniert werden können.

Laufzeit $\in \mathcal{O}(n)$

Knotenfolge

preOder (Hauptreihenfolge) Zuerst Wurzel, dann linker / rechter Teilbaum.

```
preOrder(x) {
        print(x);
        print(preOrder(x.l));
        print(preOrder(x.r));
}
```

preNext():

- Wenn vorhanden: Linkes Kind
- Sonst, Wenn vorhanden: Rechtes Kind
- Sonst, Gehe Richtung Wurzel, bis Element nicht Rechtes Kind ist. Dann: Rechtes Kind.

postOrder (Nebenreihenfolge) Zuerst linker, dann rechter Teilbaum, dann Wurzel

inOrder (Symmetrische Reihenfolge) Zuerst linker Teilbaum, dann Wurzel, dann rechter Teilbaum

4 Graphen

4.1 Netzwerke und Graphen

4.1.1 Graph

Knoten V (Elemente), Kanten E (Verbindungen)

$$G = (V, E) \qquad n = |V| \qquad m = |E| \tag{2}$$

4.1.2 Gerichtete und Ungerichtete Graphen

$$E \subseteq \{\{v, w\}\text{bzw. } (v, w) : v, w \in V\} \tag{3}$$

Anwendung Ungerichtete Graphen stellen symmetrische Beziehungen dar. Gerichtete Graphen können zur Darstellung asymetrischer und kreisfreier Beziehungen genutzt werden.

4.1.3 Adjazenz, Inzidenz, Grad

Adjazenz Zwei benachbarte Knoten

Inzidenz Mit Kanten verbundener Knoten

Grad Anzahl der Nachbarn eines Knotens v: $\deg(v)$, $\deg^-(v), \deg^+(v)$: Eingangs bzw. Ausgangsgrad bei gerichteten Graphen

4.1.4 Annahmen

- endlicher Graph
- einfacher Graph (keine Multikanten, Schleifenfrei)

4.1.5 Gewichtete Graphen

Knoten- bzw. Kantengewicht. $w : E \leftarrow \mathbb{R}$, $w(e)$ ist Gewicht einer Kante

4.1.6 Weg, Pfad und Kreis

Weg Folge von Knoten und Kanten. Länge des Weges int Anzahl der Kanten

Pfad Pfad ist kantendisjunkt. Einfacher Pfad ist Knotenjisjunkt

Kreis Startpunkt eines Weges ist Endpunkt

4.1.7 Operationen

Graph G: Datenstruktur für Graph, **Node**: Datenstruktur für Knoten, **Edge**: Datenstruktur für Kanten

- G.insert(Edge e): Kante hinzufügen
- G.remove(Key i, Key j): Kante Zwischen Knoten mit Schlüssel i, j löschen
- G.insert(Node v): Knoten hinzufügen
- G.remove(Key i): Knoten mit Schlüssel i und alle inzidenten Kanten löschen
- G.find(Key i): Knoten mit Schlüssel i finden
- G.find(Key i, Key j): Kante zwischen Knoten mit Schlüsseln i, j finden

Wenn Anzahl der Knoten konstant: $V = \{0, 1, ..., n-1\}$. Sonst Hashing in Bereich $\{0, ..., \mathcal{O}(n)\}$.

4.2 Grapehnpräsentation

4.2.1 Kantenliste

Liste von Kanten

4.2.2 Adjazenzmatrix

Matrixdarstellung benachbarter Knoten

$$A = (a_{ij})_{1 \leq i,j \leq n} \qquad a_{ij} = 1 \Leftrightarrow (i,j) \in E \tag{4}$$

Speicherplatz $\Theta(n^2)$

4.2.3 Inzidenzmatrix

Matrixdarstellung, benachbarter Knoten mit Kanten

$$A = (a_{ij})_{1 \leq i \leq n, 1 \leq j \leq m} \qquad a_{ij} = 1 \Leftrightarrow (i,x) = j \vee (x,i) = j \quad : \quad j \in E \tag{5}$$

4.2.4 Adjazenzarray

4.2.5 Adjazenzliste

Liste von Knoten mit jeweils einer inzidenten Kantenliste.
Speicherplatz $\mathcal{O}(n+m)$

4.2.6 Implizit

4.3 Breitensuche

Startknoten in Queue speichern und markieren.

Erstes Element in Queue nehmen. Alle angrenzenden Knoten (die nicht schon markiert sind) markiere und in Queue speichern. Erstes Element in Queue löschen. So lange bis keine unmarkierten Knoten mehr vorhanden sind.

$$\sum_{v \in V} \deg(v) = 2m \tag{6}$$

Laufzeit bei Adjazenzliste $\in \mathcal{O}(n+m)$
Laufzeit bei einer Adjazenzmatrix: $\in \mathcal{O}(n^2)$
$\Rightarrow$ Adjazenzliste ist im Allgemeinen nicht schlecht.

4.4 Tiefensuche (DFS)

Wie Breitensuche, nur Stack anstatt Queue

- Graph G ist DAG (Directed Acyclic Graph)
- DFS in G enthält keine Rückwärtskante
- $\forall (v,w) \in E$ finishNum[v] > finishNum[w]

4.5 Zusammenhang

Ungerichteter Graph: Pfad von jedem Knoten zu jedem anderen.

Zusammenhangskomponente ist maximal zusammehängender Teilgraph

k-fach zusammenhängend: $|V| > k \wedge G - X$ zusammenhängend ($X \subset V, |X| < k$)

4.6 Artikulationsknoten

v ist Artikulationsknoten, wenn gilt $k(G) < k(G-v)$

Block ist maximal zusammenhängender Teilgraph ohne Artikulationsknoten. Blöcke = Zweifachzusammenhangskomponenten, Brücken, isolierte Knoten

4.7 Starke Zusammenhangskomponenten

$U \subseteq V$ ist stark zusammenhängend, wenn ein gerichteter Pfad von jedem u nach v in G existiert. U ist starke Zusammenhangskomponente, wenn stark zusammenhängend und maximal. Partitionieren Graph. U zu Superknoten $\Rightarrow$ Graph wird DAG.

4.8 Topologische Sortierung

Auf gerichteten Graphen

$(i,j) \in E \Rightarrow topo(i) < topo(j)$. Nur in DAG (Directed Acyclic Graphs)

Knoten ohne Eingangskanten markieren (in Queue einfügen / Platz in Queue ist topologische Nummerierung) und aus Graph löschen. Benachbarte Knoten von entfernem Knoten auf Eingangkanten prüfen, …

4.9 Kürzeste Wege

4.9.1 All Pairs

4.9.2 Single Source

Kürzester Weg von Startknoten zu allen anderen Knoten

Knoten in topologischer Reihenfolge abgehen und Wegkosten aller Nachfolger setzen / aktualisieren ($\sim$ Breitensuche)

4.9.3 Dijkstra Algorithmus

Nichtnegative Kantengewichte

Laufzeit: (z.B. Fibonacci Heap: amortisierte Komplexitat $\mathcal{O}(1)$ für insert und decreaseKey, $\mathcal{O}(\log n)$ deleteMin)
$\mathcal{O}(n)$ insert
$\mathcal{O}(n)$ deleteMin
$\mathcal{O}(m)$ decreaseKey
$\mathcal{O}(m + n \log n)$

$\mathcal{O}(n + m) \log n$ für Binomial Heap

Kreis mit Kantengewichten < 0 führt zu Distanz $-\infty$

4.9.4 Bellman-Ford Algorithmus

Betrachte Pfade in Reihenfolge steigender Kantenanzahl. Durchlaufe alle Kanten n-1 mal.

Distanzverbesserung in n-ter Runde $\Rightarrow$ Kreis mit negativem Kantengewicht. Knoten bekommt Distanz $-\infty$

5 Klausurvorbereitung

(Doppel-)rotation bei avl Bäumen.