Inhaltsverzeichnis

Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Institut für Theoretische Informatik
Peter Widmayer
Michael Gatto, Beat Gfeller

Karatsuba-Rechenbeispiel

Wir berechnen hier $2328 \cdot 1742$ mit der Methode von Karatsuba/Ofman.
Zuerst eine kurze Repetition des Schemas:

$$
\begin{array}{rr|r|r|}
AB \cdot CD = & & & \\
\hline
A \cdot C & & A \cdot C & \\
 & & B \cdot D & \\
 & & & B \cdot D \\
 & & (A-B)\cdot & \\
 & & (D-C) & \\
\hline
\end{array}
$$

Um also $2328 \cdot 1742$ zu berechnen, müssen wir das Schema wie folgt anwenden: $A = 23$, $B = 28$, $C = 17$, $D = 42$.
Wir müssen also zuerst $A \cdot C = 23 \cdot 17$ ausrechnen:

$$
\begin{array}{rr|r|r|}
23 \cdot 17 = & 2 & & \\
 & & 2 & \\
 & 2 & 1 & \\
 & & 2 & 1 \\
 & & -6 & \\
\hline
 & 3 & 9 & 1 \\
\hline
\end{array}
$$

Danach $B \cdot D = 28 \cdot 42$:

$$
\begin{array}{rr|r|r|}
28 \cdot 42 = & 8 & & \\
 & & 8 & \\
 & 1 & 6 & \\
 & & 1 & 6 \\
 & 1 & 2 & \\
\hline
 & 11 & 7 & 6 \\
\hline
\end{array}
$$

Schliesslich müssen wir noch $(A-B) \cdot (D-C) = (23-28) \cdot (42-17) = -5 \cdot 25$ rechnen (das könnte man auch im Kopf, aber der Vollständigkeit halber hier mit Karatsuba). Wir rechnen hier $5 \cdot 25$ und setzen dann erst ein $-$ vor das Resultat:

$$
\begin{array}{rr|r|r|}
05 \cdot 25 = & 0 & & \\
 & & 0 & \\
 & 2 & 5 & \\
 & & 2 & 5 \\
 & -1 & -5 & \\
\hline
 & 1 & 2 & 5 \\
\hline
\end{array}
$$

Also ist $-5 \cdot 25 = -125$.
Jetzt haben wir alle Zwischenresultate zusammen, um $2328 \cdot 1742$ zu berechnen:

$$
\begin{array}{rr|r|r|}
2328 \cdot 1742 = & 391 & & \\
 & & 3 & 91 \\
 & 11 & 76 & \\
 & & 11 & 76 \\
 & -(1 & 25 &) \\
\hline
 & 405 & 53 & 76 \\
\hline
\end{array}
$$

Wir erhalten $2328 \cdot 1742 = 4'055'376$.

1 TODO

- REKURSIONSGLEICHUNGEN
- Fluss Algorithmen (9.7) (dinic etc)
- 2-dimensionaler Range-Tree (8.3/8.5)
- Branch and Bound

2 Algorithmen

2.1 $\mathcal{O}$-Notation

Hier immer $f : \mathbb{N} \to \mathbb{R}^+$!

Obere Schranke $\mathcal{O}(g) := \{f | \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \forall n \geq n_0 : f(n) \leq cg(n)\}$

Untere Schranke $\Omega(g) := \{f | \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \ \forall n \geq n_0 : f(n) \geq cg(n)\}$

Asymptotisch gleich $\Theta(g) := \{f | \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \ \forall n \geq n_0 : c^{-1}g(n) \leq f(n) \leq cg(n)\}$

Wachstumsordnung

$$1 << \log\log n << \frac{\log n}{\log\log n} << \log n << \log^2 n << \sqrt{n}$$

$$n << n\log n << n^{1+\varepsilon} << n^2 << n^3 << c^n << n! << n^n$$

2.2 Amortisierte Analyse

Die tatsächlichen Kosten der i-ten Operation heissen t_i.

2.2.1 Potentialfunktion

Die Potentialfunktion Φ_i beschreibt das Potential nach der i-ten Operation. Demnach sind die amortisierten Kosten der i-ten Operation definiert als $a_i = t_i + \Phi_i - \Phi_{i-1}$. Damit folgt für m Operationen:

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} (t_i + \Phi_i - \Phi_{i-1}) = \left(\sum_{i=1}^{m} t_i\right) + \Phi_m - \Phi_0$$

Und dann

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i - \Phi_m + \Phi_0$$

Wenn es also gelingt, die amortisierten Kosten und den Term $\Phi_0 - \Phi_m$ abzuschätzen, hat man auch eine Abschätzung für die tatsächlichen Kosten. Wenn man dann noch $\Phi_0 \leq \Phi_m$ hat, sind die amortisierten Kosten eine obere Schranke für die tatsächlichen Kosten.

2.3 Dynamische Programmierung

1. *Definition der DP-Tabelle:* Welche Dimension hat die Tabelle? Was ist die Bedeutung jedes Eintrags?
2. *Berechnung eines Eintrags:* Wie berechnet sich ein Eintrag aus den Werten von anderen Einträgen? Welche Einträge hängen nicht von anderen Einträgen ab?
3. *Berechnungsreihenfolge:* In welcher Reihenfolge kann man die Einträge berechnen, so dass die jeweils benötigten anderen Einträge bereits vorher berechnet wurden?
4. *Auslesen der Lösung:* Wie lässt sich die Lösung am Ende aus der Tabelle auslesen?

2.4 Sortieren

Selectionsort funktioniert via Auswählen der Position des minimalen Schlüssels j_1 und nachfolgigem Vertauschen von $a[1]$ mit $a[j_1]$. Dann wird j_2 bestimmt, und $a[j_2]$ mit $a[2]$ vertauscht, etc. Nicht stabil. In-place.
$\Theta(n^2)$ Vergleiche, $\Theta(n)$ Vertauschungen.

Insertionsort Die zu sortierenden Elemente werden nacheinander betrachtet und in die jeweils bereits sortierte, anfangs leere Teilfolge an der richtigen Stelle eingefügt. Wenn also $a[1]$ bis $a[i-1]$ schon sortiert sind, wird $a[i]$ nach links verschoben, bis es am richtigen Platz ist. Stabil. In-place.
$\mathcal{O}(n^2)$ Vergleiche und Vertauschungen

Shellsort Wie Insertionsort, aber versucht das die Verschiebungsschritte zu vergrössern, indem eine Folge abnehmender Inkremente (z.B. $5, 3, 1$) verwendet wird. Eine Teilfolge kann dann 5-sortiert sein. Nicht stabil. In-place.
Bounds natürlich abhängig von Folge von Inkrementen.

Bubblesort Verwendet als Bewegung *nur* das Vertauschen benachbarter Datensätze. Läuft durch, und nimmt immer das grössere Element von zwei benachbarten mit. Stabil. In-place.
$\mathcal{O}(n^2)$ Vertauschungen und Vergleiche.

Quicksort Sortieren durch rekursives Teilen. Ein Pivotelement k wird gewählt, dass die Elemente in zwei Teile teilt, alle Elemente kleiner bzw. grösser k. Die Elemente werden korrekt um k angeordnet (durch Vertauschen), so dass das Problem jetzt nur noch aus dem Sortieren der zwei Teilfolgen besteht. Nicht stabil. In-place (abgesehen vom Rekursionsstack).
$\mathcal{O}(n \log n)$ Zeit.

Heapsort Heap herstellen. Das kleinste (erste) Element ist das nächste in der Ordnung. Entferne es, und stelle die Heapbedingung für die restlichen Schlüssel wieder her. Nicht stabil. In-place.
$\mathcal{O}(n \log n)$ Zeit.

Mergesort Die Gesamtfolge wird in zwei Teilfolgen geteilt, die rekursiv mit Mergesort sortiert werden. Danach werden die beiden Teilfolgen wieder zu einer Folge verschmolzen, indem man zwei Zeiger auf die ersten Elemente setzt, das kleinere der beiden nimmt und den Zeiger vorrückt. Natürlicher Mergesort versucht schon existierende bitonische Läufe auszunutzen. Stabil. Nicht in-place.
$\mathcal{O}(n \log n)$ Zeit.

2.4.1 Radixsort

Radixsorts nutzen Eigenschaften neben Vergleichbarkeit der Schlüssel aus. Schlüssel sind Wörter über einem aus m Elementen bestehenden Alphabet, also bei z.B. $m = 10$ Dezimalzahlen, $m = 2$ Binärzahlen.

Radix-exchange-sort startet beim MSB, teilt die Folge in zwei und ruft sich rekursiv auf den beiden Teilfolgen auf. Dort wird dann aufgrund des nächsten Bits geteilt, etc. Geteilt wird wie bei Quicksort, zwei Zeiger laufen ÃŒber die Schlüssel und vertauschen wenn nötig. Stabil. In-place.
$\mathcal{O}(N * b)$. Gut, wenn Anzahl Bits $b = \log N$. Nicht gut bei wenigen langen Schlüsseln.

Bucketsort Verteilungsphase: Schlüssel werden basierend auf der ls Ziffer auf Buckets verteilt (also $14, 24, 64, 44$ in einen Bucket). Sammelphase: Die Schlüssel werden eingesammelt, nach Bucket sortiert, innerhalb der Buckets das "unterste" zuerst. Es folgt eine Verteilphase basierend auf der nächsten Ziffer, und ebenso eine Sammelphase. Während der zweiten Verteilphase bleibt die Ordnung der ersten erhalten, die kleinsten ls Ziffern sind "unten". Stabil. Nicht in-place.
$\mathcal{O}(n+k)$ Laufzeit, mit k Anzahl Buckets. Mit k konstant $\mathcal{O}(n)$.

2.5 Suchen

Median der Mediane ist ein Vorgehen, mit dem ein günstiges Pivot-Element gefunden werden kann. Man teilt die N Elemente in $\lfloor \frac{N}{5} \rfloor$ Gruppen auf, sortiert in diesen Gruppen in konstanter Zeit und wendet dieses Verfahren rekursiv auf die Mediane der 5er Gruppen an. Man erhält den Median der Mediane, welcher man als Pivot verwenden kann.

Auswahl Mit dem Median der Mediane als gutes Pivot-Element kann das i-te Element einer Folge in $\mathcal{O}(N)$ gefunden werden.

Binäre Suche erfolgt nach dem Divide-and-conquer Prinzip. Sucht auf einer geordneten Folge von Schlüsseln indem es den gesuchten Schlüssel mit dem mittigen Element der Folge vergleicht und somit die Hälfte der Schlüssel ausschliessen kann.

Fibonacci Suche kommt ohne Divisionen auf, man teilt den Suchbereich (bequemerweise $F_n - 1$) in zwei Fs auf: $F_{n-2} - 1$ und $F_{n-1} - 1$ (zwei Bereiche und ein Einzelelement). $\mathcal{O}\log n$ also wie Binär.

Exponentielle Suche ist nützlich, wenn N sehr gross ist. Es wird zuerst mit exponentiellen Schritten ein Bereich festgelegt, in dem sich das gesuchte Element befindet, und dieser Bereich wird dann z.B. mit BS durchsucht.

Interpolationssuche macht intuitiv Sinn, versucht ungefähre Position des Schlüssels zu schätzen. Ist aber potentiell langsamer als BST.

2.5.1 Hashen

Chaining nennt sich die Strategie, Überläufer in verketteten Listen zu speichern.

Offenes Hashen heisst, dass man Überläufer nicht verkettet, sondern versucht eine andere Position in der Hashtabelle zu finden. Bedeutet natürlich, dass das Entfernen langsamer wird. Braucht eine Sondierungsfunktion.

Sondierung $(h(k) - s(j,k)) \mod m$ mit $h(k)$ als Hashfunktion, $s(j,k)$ als Sondierungsfunktion, wobei $j = 1, 2, 3 \ldots$

Lineares Sondieren mit Sondierungsfunktion $s(j,k) = j$

Quadratisches Sondieren heisst $s(j,k) = (\lceil \frac{j}{2} \rceil)^2 (-1)^j$

Double Hashing mit $s(j,k) = j * h'(k)$, wobei $h'(k)$ eine zweite Hashfunktion ist.

Dynamisches Hashen addressiert das Problem von kleinen und grossen Belegungen, welche bei offenem Hashing ineffizient sind (Speicher- bzw Suchineffizient). Bei dynamischem Hashing kann sich die Hashtabelle an die Belegung anpassen, also schrumpfen oder wachsen.

Lineares Hashing mit schrittweiser Hinzunahme von Behältern wenn der Belegungsfaktor grösser wird. Funktioniert mit maximal zwei Hashfunktionen, $h_L = h_1$ und $h_{L+1} = h_2$.
Es gilt $h_{L+1}(k) = \begin{cases} h_L(k) \text{ oder} \\ h_L(k) + m_0 * 2^L \end{cases}$
Beispiel: $h_L(k) = k \mod (m_0 * 2^L)$ Beim Linearen Hashing wird immer nur um einen Behälter vergrössert.

Virtuelles Hashing verdoppelt hingegen in jedem Expansionsschritt seine Kapazität, vermeidet also Überlaufblöcke. Dafür sind mehr Hashfunktionen nötig, die gespeichert werden müssen.

Erweiterbares Hashing Not a clue wie das funktioniert.

2.5.2 Selbstanordnende Listen

Sind nützlich bei grossen Unterschieden in der Zugriffshäufigkeit auf Elemente, dann lohnt es sich nämlich, die häufiger genutzten möglichst weit vorne zu platzieren.

Move-to-front bedeutet, dass ein gebrauchtes Element an den Anfang verschoben wird. Die restliche Ordnung verändert sich nicht. Asymptotisch optimal.

Transpose heisst schlicht dass ein abgerufenes Element einen Platz nach vorne rutscht.

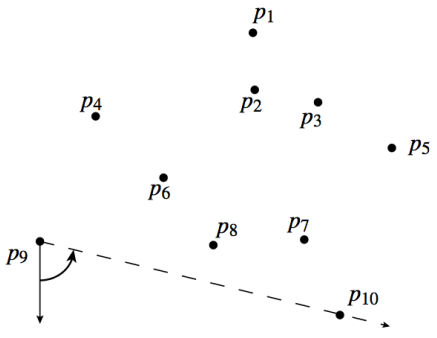
Frequency Count ordnet jedem Element einen Zähler zu, und ordne nach jedem Zugriff die Liste nach dem Zähler.

2.6 Geometrische Algorithmen

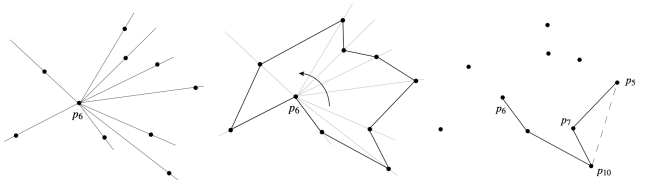
2.6.1 Konvexe Hülle

Die konvexe Hülle einer Menge von Punkten in der Ebene ist jenes Polynom, welches man erhält, wenn man eine Schnur straff um die Punkte spannt. D.h. wenn eine Gerade zwischen zwei Punkten alle anderen Punkte auf einer Seite hat, ist sie Teil der konvexen Hülle.

Jarvis' Marsch ist die intuitive Technik, man startet an dem Punkt mit der kleinsten x-Koordinate, und 'lässt die Schnur nach unten baumeln', und rotiert sie dann gegen den Uhrzeigersinn, bis man auf einen nächsten Punkt trifft. $\mathcal{O}(Nh)$, mit h Anzahl Punkte auf der Hülle. Effizient, wenn die Hülle aus wenigen Punkten besteht, aber potentiell $\mathcal{O}(N^2)$, z.B. bei einem Kreis.



Graham's Scan Wähle einen Punkt und errechne die $n-1$ Linien aus allen anderen Punkten. Sortiere zyklisch nach der Steigung (Winkel) und entferne schliesslich alle "Links-Knicke". Amortisierte Laufzeit $\mathcal{O}(n \log n)$.



2.6.2 Scan-line-Prinzip

Eine Beschreibung eines Scanline-Algorithmus muss immer folgende Punkte beinhalten:

1. *Haltepunkte.* In welcher Richtung verläuft die Scanline? Was sind die Haltepunkte?
2. *Scanline-Datenstruktur.* Welche Objekte muss die Datenstruktur verwalten? Welche Operationen müssen unterstützt werden? Was ist eine angemessene Datenstuktur?
3. *Aktualisierung.* Was passiert, wenn die Scanline auf einen Haltepunkt trifft?
4. *Auslesen der Lösung.* Wie lässt sich die Lösung auslesen?

3 Datenstrukturen

3.1 Listen

3.1.1 Skip-Listen

Randomisierte Skip-Listen Eine randomisierte Datenstruktur, bestehend aus verschiedenen Ebenen. Die unterste Ebene ist eine normale verkettete Liste. Die höheren Ebenen kann man sich als Überholspur vorstellen. Ein Element wird in der untersten Ebene eingefügt, und dann mit gewissen Wahrscheinlichkeiten p in die Ebenen darüber eingefügt.

Perfekte Skip-Liste Die perfekte Skip-Liste hat ebenfalls eine verkettete Liste als Niveau 0. Dazu kommt aber, dass jedes 2^i-te Element eine Zeiger das Element 2^i Positionen weiter hat, für $i = 0, \ldots, \lfloor \log N \rfloor$. Es hat also im Niveau 1 die Elemente $2, 4, 6, \ldots$, im Niveau 2 $4, 8, 12$ etc.

3.2 Heaps - "Halden"

Max-Heapbedingung besagt $k_i \leq k_{\lfloor \frac{i}{2} \rfloor}$ für $2 \leq i \leq N$.

Min-Heapbedingung besagt $k_i \geq k_{\lfloor \frac{i}{2} \rfloor}$ für $2 \leq i \leq N$.

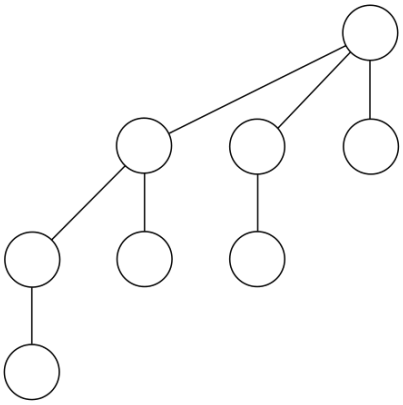
Jeder Schlüssel ist kleiner (grösser) als der Vaterschlüssel.
Bei einem Min-(Max-)Heap kann das Minimum (Maximum) in einem Schritt bestimmt werden.

3.2.1 Binomial-Heap

Der Binomial-Heap besteht aus Listen von Binomial-Bäumen verschiedener Ordnung.

Der Binomial-Baum ist rekursiv definiert:

- Ein Binomial-Baum der Ordnung 0 besteht aus einem einzenen Knoten.
- Ein Binomial-Baum der Ordnung k besitzt eine Wurzel mit Grad k deren Kinder genau die Ordnung $k-1, k-2, \ldots, 0$ besitzen.



Ein Binomial-Baum mit Ordnung 3

In einem Binomial-Heap gelten folgende Bedingungen.

- Die Heap-Bedingung: Jeder Schlüssel ist kleiner als sein Vorgänger.
- Für jede natürliche Zahl k gibt es nur einen Binomial-Baum der Ordnung k

Ein Binomial-Heap unterstützt die folgenden Operationen.

insert – Einfügen

remove – Löschen

extractMin – Rückgabe und Entfernung vom Element mit dem minimalen Schlüssel, also der höchsten Priorität

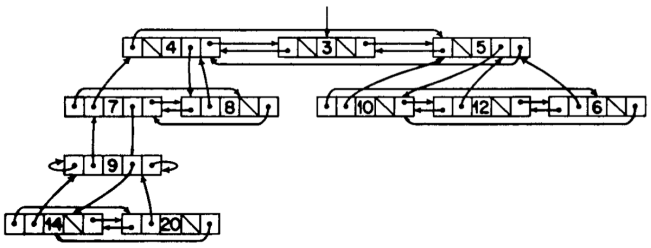
changeKey – zum Ändern des Schlüssels eines Elements.

merge – Vereinigung von zwei Binomial Heaps.

Alle diese Operationen können in einer *Worst-Case-Zeit* von $\mathcal{O}(\log n)$ implementiert werden.

3.2.2 Fibonacci-Heap

Der Fibonacci-Heap ist vom Binomial-Heap abgeleitet, also auch eine Priority-Queue. Ein F-Heap ist eine Kollektion heapgeordneter Bäueme mit disjunkten Schlüsselmengen. Alle Operationen ausser *deleteMin* ($\mathcal{O}(\log n)$) lassen sich in amortisiert $\mathcal{O}(1)$ implementieren.



Ein Fibonacci Heap mit Pointern, aber ohne Markierung

Jeder Knoten hat einen Pointer auf sein linkes Geschwister, sein Elternteil, ein Kind und das rechte Geschwister. Das mittlere Feld ist der Schlüssel. Es ist ein Markierungsbit in jedem Knoten vorhanden. Die Kinder eines Knotens sind in einer verketteten zyklischen Liste gespeichert. Der Rang eines Knotens ist die Anzahl seiner Kinder.

Operationen auf dem Fibonacci Heap

makeHeap() Gibt einen Pointer auf den Null-Knoten zurück.

findMin(h) Es gibt einen Pointer auf das minimale Element von h, also ist $findMin$ trivial.

deleteMin(h) Die schwierigste Operation. Wir beginnen indem wir das minimale Element x von h entfernen, die Liste der Kinder von x und die der Wurzeln ausser x verketten und wiederholen dann den folgenden Schritt bis er nicht mehr anwendbar ist:

- *Linking Step.* Finde zwei Bäume deren Wurzel denselben Rang hat, und verschmelze sie. (Der neue Baum hat jetzt einen höheren Rang als die Ränge der verschmolzenen).

Sobald keine Bäume mit dem selben Rang übrig sind bilden wir eine Liste aus den übrigen Wurzeln, wobei wir auch das neue Minimalelement finden.

insert(i, h) Wir kreieren einen neuen Heap mit i als einzigem Element und rufen $meld(h, h_{\text{neu}})$ auf.

meld(h_1, h_2) Wir vereinigen die Wurzellisten von h_1 und h_2 und bestimmen das Minimum aus den beiden vorherigen Minimas.

decreaseKey(Δ, i, h) Wir subtrahieren Δ von i, finden den Knoten x der i enthält und entfernen die Kante zwischen x und seinem Elternteil $p(x)$. Wir entfernen also den Parent-Pointer von x auf $p(x)$ und löschen x aus der Liste der Kinder von $p(x)$. Damit haben wir den Teilbaum mit x als Wurzel zu einem neuen Baum in h gemacht, und müssen noch überprüfen, ob x das neue Minimalelement ist.

delete(i, h) Finde den Knoten x der i enthält, entferne die Kante zu $p(x)$. Erstelle eine neue Wurzelliste durch verketten der Kinder von x mit der alten Wurzelliste.

Ein wichtiges Detail ist noch zu beachten. Nachdem Knoten x zum Kind eines anderen Knoten gemacht wurde, sobald x zwei seiner Kinder verliert, entfernen wir die Kante zwischen x und $p(x)$. Wir nennen diesen Schnitt *Cascading Cut*, Kaskadierender Schnitt. Dies ist auch der Sinn der Markierungen. Wenn x Kind eines anderen Knoten wird, unmarkieren wir x. Wenn wir eine Kante zwischen x und $p(x)$ entfernen, und $p(x)$ keine Wurzel ist, markieren wir $p(x)$ falls er noch nicht markiert ist, und entfernen die Kante zu *seinem* Elternteil. Bleh.

3.3 Bäume

Bäume sind verallgemeinerte Listen, in denen jedes Element (Knoten) eine begrenzte Anzahl Söhne haben kann. Keine Duplikate!

3.3.1 Traversierung von Binärbäumen

Preorder Wurzel, Linker Teilbaum, Rechter Teilbaum

Postorder Linker Teilbaum, Rechter Teilbaum, Wurzel

Inorder Linker Teilbaum, Wurzel, Rechter Teilbaum

3.3.2 Binärsuchbäume

Vanilla Binärsuchbaum mit einer Wurzel, und grösseren rechten Nachfolgern, kleineren linken Nachfolgern.

Suchen Rekursiv in Teilbäumen T. Abbruch wenn T keine Wurzel ist. Wenn $x <$ Wurzel von T suche in linkem Teilbaum, sonst im rechten.

Einfügen Suchen nach Knoten, einfügen als Blatt des Knoten in dem die Suche endete.

Löschen Wenn Blatt, löschen. Wenn einen Sohn, löschen und Sohn hochziehen. Wenn zwei Söhne, symmetrischer Nachfolger suchen und zu löschender Knoten ersetzen.

AVL-Bäume sind balancierte binäre Suchbäume. In jedem Teilbaum ist der Höhenunterschied höchstens 1.

Suchen Genau wie bei einem normalen binären Suchbaum.

Einfügen Wie beim Suchbaum, aber dann Balanceinformationen erneuern. Falls Unbalanciert: Einfach- oder Doppelrotation bis wieder balanciert, dann weiter BI erneuern.

Löschen Wie beim BSB, aber danach wieder balancieren.

Huffman-Baum wird erstellt durch die Huffman-Kodierung, ein Kompressionsverfahren. Die Mächtigkeit des Alphabets über dem kodiert wird heisst m. Man kreiert einen Wald, der aus den verwendeten Zeichen und ihren Häufigkeiten besteht. Dann nimmt man die geringsten m Häufigkeiten und addiert sie, und verschmilzt sie zu einem Baum (der Ordnung m) mit der Summe als Wurzel. Dann nimmt man wieder die zwei kleinsten Häufigkeiten, was (beim Binäralphabet) die Summe und ein neues Element oder zwei neue Elemente sein kann.

Optimale Suchbäume versuchen die (erwarteten) Suchkosten zu minimieren.

Die Menge $S = k_1, \ldots, k_n$ von n verschiedenen Schlüssel,

Die Häufigkeit a_i mit der nach k_i gesucht wird,

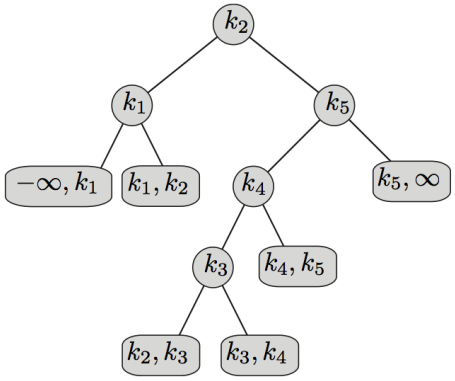
Das offene Interval (k_0, k_{n+1}) von Schlüsseln, nach denen gesucht werden kann (Oft gilt $k_0 = -\infty$ und $k_{n+1} = \infty$),

Die Häufigkeit b_j mit der nach einem Schlüssel $x \in (k_j, k_{j+1})$ gesucht wird.

Das Gewicht W des Suchbaums die summierte Häufigkeit seiner Knoten: $W = \sum_i a_i + \sum_j b_j$

Die gewichtete Pfadlänge P misst wie viele Schlüsslvergleiche total nötig sind:

$$P = \sum_{i=1}^{n} (depth[k_i] + 1) * a_i + \sum_{j=0}^{n} depth(leaf(k_j, k_{j+1})) * b_j$$



Ein Suchbaum heisst optimal, wenn P minimal ist. Es gilt, dass jeder Teilbaum eines optimalen Suchbaums auch ein optimaler Suchbaum ist. Somit eignet sich die dynamische Programmierung für einen Algorithmus.

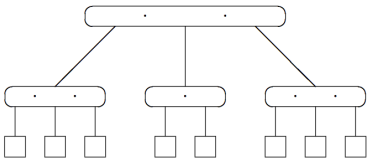
Splay-Bäume Splay-Bäume sind reine binäre Suchbäume, die sich durch eine Variante der Move-To-Root Strategie selbst anordnen. Alle Standard-Operationen haben eine amortisierte Laufzeit von $\mathcal{O}(\log n)$. Die *splay*-Operation ist wiefolgt definiert:

- (zig) Wenn $p(x)$, der Vater von x, die Wurzel ist, rotiere die Kante welche x mit $p(x)$ verbindet. (terminal case)
- (zig-zig) Wenn $p(x)$ *nicht* die Wurzel ist und x und $p(x)$ beides rechte oder linke Kinder sind, rotiere die Kante welche $p(x)$ mit dem Grossvater $g(x)$ verbindet, und rotiere dann die Kante welche x mit $p(x)$ verbindet.
- (zig-zag) Wenn $p(x)$ *nicht* die Wurzel ist und x ein linkes, $p(x)$ ein rechtes Kind sind, rotiere die Kante welche x mit $p(x)$ verbindet, und dann die Kante, welche x mit dem neuen $p(x)$ verbindet.

3.3.3 B-Baum

Nützlich, wenn nicht der ganze Baum im Hauptspeicher Platz hat. Ein B-Baum der Ordnung m ist ein Baum mit den folgenden Eigenschaften:

1. Alle Blätter haben die gleiche Tiefe.
2. Jeder Knoten mit Ausnahme der Wurzel und der Blätter hat wenigstens $\lceil \frac{m}{2} \rceil$ Söhne
3. Die Wurzel hat wenigstens 2 Söhne.
4. Jeder Knoten hat höchstens m Söhne.
5. Jeder Knoten mit i Söhnen hat $i-1$ Schlüssel



Ein B-Baum der Ordnung 3

3.4 Scanline Datenstrukturen

Eine Überlappungsanfrage ist äquivalent zu einer Aufspiessanfrage und einer Bereichsanfrage. Um eine Überlappung mit $[a,b]$ zu prüfen, genügt es also

1. alle Intervalle $[a',b']$ zu finden, die der linke Randpunkt a aufspiesst und
2. alle Intervalle $[a',b']$ zu finden, deren linker Randpunkt a' im Bereich $[a,b]$ liegt.

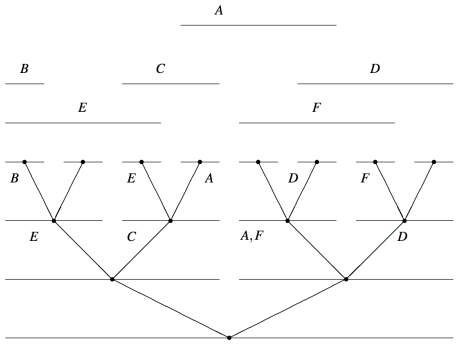
3.4.1 Segment-Bäume

Konstruiert von einem vollständigen Binärbaum, der alle elementaren Segmente als Blätter hat. See Pic :]

delete braucht Wörterbuch mit den Intervallen. Dann $\mathcal{O}(\log N)$.

insert ist in $\mathcal{O}(\log N)$ Schritten ausführbar.

Aufspiessanfragen brauchen $\mathcal{O}(\log N + k)$ mit k als Grösse der Antwort.

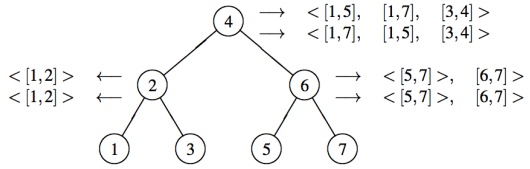


Ein Segment Baum

Aufspiessanfrage nach x wird realisiert indem der Segment-Baum als Suchbaum für x benutzt wird, d.h. wir suchen das Elementarsegment welches x beinhaltet. Dann geben wir als Antwort alle Listen von Intervallen an, die wir in den Knoten auf dem Weg angetroffen haben.

3.4.2 Intervall-Bäume

In einem Intervall-Baum hat jeder Knoten zwei verkettete Listen von Intervallen, die u- und die o-Liste. Die u-Liste ist eine nach aufsteigenden unteren Endpunkten sortiert, die o-Liste nach absteigenden oberen Endpunkten. Ein Intervall $[l,r]$ kommt in beiden Listen desjenigen Knotens im Skelett des Intervall-Baumes mit minimaler Tiefe vor, dessen Schlüssel im Intervall $[l,r]$ liegt.
Für die Menge der Intervalle $\{[1,2],[1,5],[3,4],[5,7],[6,7],[1,7]\}$ haben wir demnach den folgenden Intervall-Baum:



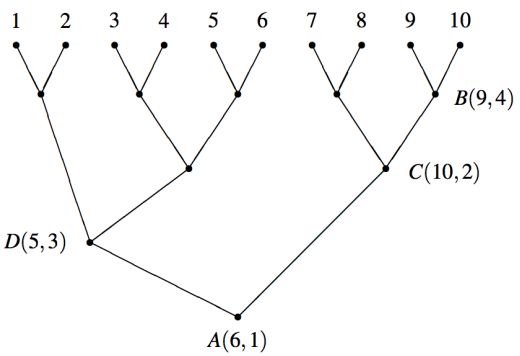
delete ist in $\mathcal{O}(\log N)$ Schritten ausführbar.

insert ist in $\mathcal{O}(\log N)$ Schritten ausführbar.

Aufspiessanfragen brauchen $\mathcal{O}(\log N + k)$ mit k als Grösse der Antwort.

3.4.3 Prioritäts-Suchbäume

sind 1.5-dimensional, und speichern 2-dimensionale Daten. Jeder Punkt (x,y) (also jedes Intervall $[x,y]$) wird auf einem Suchpfad von der Wurzel zum Blatt x an einem inneren Knoten entsprechend seinem y-Wert abgelegt. Wir haben also gleichzeitig einen Min-Heap über y und einen BSB über x. Mit A,B,C,D wie angegeben:



insert wird am Anfang immer in die Wurzel gehen, dann dem Suchbaum folgen.

delete sucht zuerst nach dem zu löschenden Punkt, und dann der Sohn mit dem kleineren y-Wert hochgezogen (um die Heap-Bedingung nicht zu verletzen).

Bereichsanfrage in $\mathcal{O}(\log N + k)$ Schritten.

Es gibt auch eine Möglichkeit, den Prioritäts-Suchbaum voll dynamisch zu realisieren, siehe S.524 in Widmayer.

3.5 Union-Find-Struktur

Der einfachste Weg das Union-Find-Problem zu lösen, ist jede Kollektion K als ein unsortierter Baum beliebiger Ordnung darzustellen. Dabei ist die Wurzel des Baumes das kanonische Element. Union-Find-Strukturen unterstützen die folgenden drei Operationen:

Make-set(e, i) kreiert aus dem einzigen Element e eine Menge i. Es wird vorausgesetzt, dass e in keiner anderen Menge vorkommt. $\mathcal{O}(1)$

Find(x) findet den Namen i der Menge, die x enthält.

Union(i, j, k) vereinigt die Mengen i, j zu k. i und j werden aus der Kollektion der Mengen entfernt, k hinzugefügt. $\mathcal{O}(1)$

Vereinigung nach Höhe nennt sich das Verfahren, den weniger hohen Baum an den höheren Baum zu hängen.

4 Graphen

Ein Graph besteht aus einer Knotenmenge V (vertices) und einer Menge $E \subseteq \binom{V}{2}$ von geordneten (gerichtet) oder ungeordneten (ungerichtet) Paaren von *Knoten* aus V, genannt *Kanten*.

DAGs sind directed acyclic graphs, also gerichtete zyklische Graphen. Sie treten oft in Schedulingproblemen auf. (Siehe Dynamisches Programmieren)

Topologische Sortierungen sind Ordnungen über DAGs, die, wenn z.B. eine Kante von x nach y heisst, dass x vor y eintreten muss, einem zeitlich sinnvollen Ablauf entspricht. Damit eine topologische Sortierung existieren kann, ist $\frac{n(n-1)}{2}$ die maximale Anzahl Kanten.

Matching ist eine Menge von Kanten ohne gleiche Knoten. Ein maximales Matching ist ein Matching, welches mit der Hinzunahme irgendeiner Kante nicht mehr ein Matching ist.

4.1 Repräsentationen eines Graphen

Adjazenzlisten sind Listen für jeden Knoten, in dem die Nachbarn gespeichert werden. Für viele Fälle die bessere Wahl, abgesehen vom Löschen ($\mathcal{O}(d)$) und Testen ob eine Kante im Graph ist ($\mathcal{O}(d)$), $d =$ Anzahl Nachbarn.

Adjazenzmatrizen sind $|V| \times |V|$ Matrizen, welche im Eintrag (i, j) eine Eins speichern, falls es eine Kante von Knoten i zu Knoten j gibt.

4.2 Traversierung

Breadth-First-Search (BFS) – Breitensuche Zuerst werden alle Knoten die k entfernt sind vom Ausgangsknoten besucht, dann $k + 1$ etc.

Depth-First-Search (DFS) – Tiefensuche So weit wie möglich in die Tiefe, dann so wenig wie möglich *backtracken* und wieder runter.

4.3 Spanning Trees - Spannende Bäume

Ein Spannender Baum eines Graphen G ist ein Teilgraph G' mit den gleichen Knoten aber nur einer Teilmenge der Kanten.

Kruskarl Starte mit einem leeren Graph. Nehme die billigste Kante hinzu, die keinen Zyklus kreiert, bis du einen MST hast.

Prim Starte mit einem leeren Graph V' und einem beliebigen Knoten v. Wähle die billigste Kante, die v mit einem Knoten der noch nicht in V' ist verbindet, bis du einen MST hast.

4.4 Kürzeste Wege

Dijkstra Fast identisch zu Prim. Wir haben V' mit schon abgearbeiteten Knoten (zu diesen wissen wir die kürzeste Distanz), wir beginnen mit v. Wir gehen die Nachbarn durch, speichern die kürzere Distanz, entweder die von v oder die, die bis jetzt die kürzeste war. Sobald alle Nachbarn besucht wurden, kommt v in V' und hat seine definitive kürzeste Distanz. *Keine negativen Gewichte erlaubt!* (sonst rennt er im Kreis und heult)
Je nach Priority-Queue-Implementation: $\mathcal{O}(|E| + |V| \log |V|)$ oder gar $\mathcal{O}(|V|^2)$

Bellman-Ford Erster Knoten hat Kosten 0, alle anderen ∞. Wir gehen alle Kanten durch, und prüfen ob wir einem Knoten tiefere Kosten (Kosten Anfangsknoten + Kosten Kante) zuweisen können. Nach einer Iteration haben wir also die kürzesten Wege mit einer Kante, nach 2 mit 2 Kanten etc. Bellman-Ford ist Algorithmus genug, um auch negative Kantengewichte zu überstehen, kann sogar negative Zyklen erkennen.
Laufzeit: $\mathcal{O}(nm)$

4.5 Flüsse

Min Cut = Max Flow

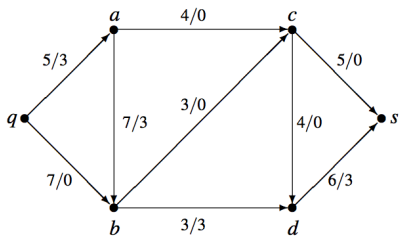
4.5.1 Ford-Fulkerson

Algorithmus zur Optimierung des Flusses.

1. Wähle irgendein Fluss von q nach s.
2. Bilde das Residualnetzwerk. "Subtrahiere" also die Kapazitäten die durch den ersten Fluss benutzt werden.
3. Wähle einen Fluss im Residualnetzwerk, und wiederhole Schritte 2 und 3 bis kein weiterer Fluss existiert.

4.5.2 Dinic

Bleh



Kapazität/Fluss

5 Verschiedenes

5.1 Vollständige Induktion

Kann für ein Prädikat $P(n)$ bewiesen werden, dass $P(n_0)$ und $\forall n \in \mathbb{N} : n > n_0 \wedge P(n) \rightarrow P(n+1)$ gilt, dann folgt daraus $\forall n \in \mathbb{N} : n \geq n_0 \rightarrow P(n)$.

Induktionsannahme (IA) bezeichnet das Prädikat $P(n)$.

Induktionsverankerung (IV) ist der Beweis von $P(n_0)$.

Induktionsschritt (IS) ist der Beweis von $P(n) \rightarrow P(n+1)$.