INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Skript

zur Vorlesung

Effiziente Algorithmen und Datenstrukturen

von

Prof. A. Steger

im

Wintersemester 1996/97

Vorläufige Version
11. August 1997

Inhaltsverzeichnis

Kapitel 1

Grundlagen

Ausarbeitung: *Angelika Steger*

1.1 Ein einführendes Beispiel

In diesem Abschnitt wollen wir an einem einfachen Beispiel einige prinzipielle Ziele und Vorgehensweisen dieser Vorlesung erläutern. Dazu betrachten wir das folgende

Szenario
GEGEBEN: $n = 10$ Millionen natürliche Zahlen $a_0, \ldots, a_{n-1} \in \mathbb{N}$.
AUFGABE: Sortiere sie!

Einen ersten (naiven) Lösungsansatz erhalten wir leicht aus den beiden folgenden Überlegungen:

(1) Ein Element ($n = 1$) können wir sortieren (denn da ist ja nichts zu tun...)!

(2) Wenn wir $n-1$ Elemente sortieren können, dann auch n Elemente: Wir fügen einfach a_{n-1} an der "richtigen" Stelle ein.

Drücken wir diese Vorgehensweise in C-Syntax aus, erhalten wir das folgende kleine Programm:

```
InsertionSort(int* A, int n)
{ int i, j, tmp;
    for (i = 1; i < n; i++) {
        tmp = A[i];
        j = i - 1;
        while ((A[j] > tmp) && (j >= 0)) {
            A[j + 1] = A[j]; j--; }
        A[j + 1] = tmp; }
}
```

Abbildung 1.1 veranschaulicht den Algorithmus INSERTIONSORT an einem Beispiel mit $n = 7$ Elementen. Die zu sortierenden Elemente haben wir hier untereinander gemalt.

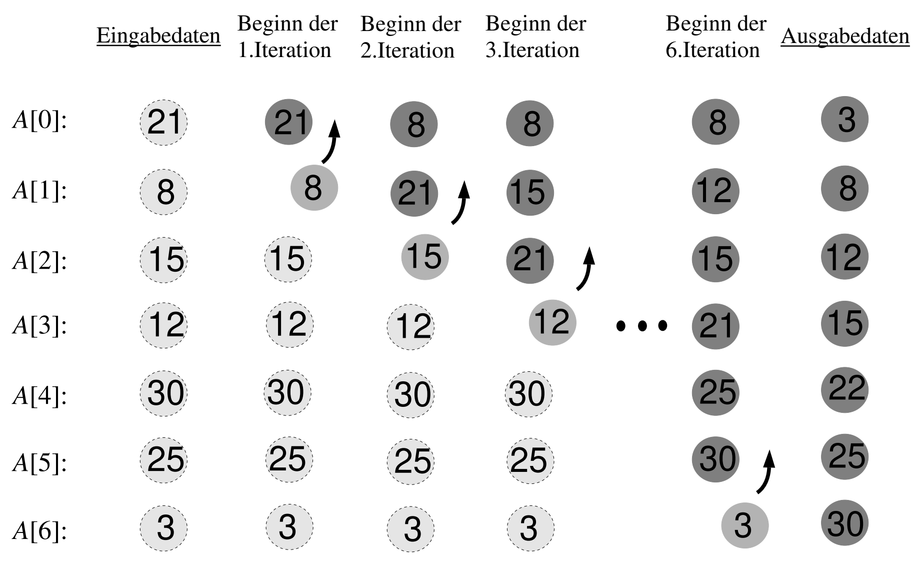


Abbildung 1.1: InsertionSort angewandt auf $n = 7$ Elemente

Vor der i-ten Iteration wurden die Elemente $A[i], \ldots, A[n]$ noch nicht betrachtet und sind daher hellgrau dargestellt.

Einen anderen, zweiten Algorithmus erhalten wir, in dem wir unsere Vorüberlegung (2) leicht modifizieren. Dazu überlegen wir uns:

(2') Wenn wir $\lceil \frac{n}{2} \rceil$ Elemente sortieren können, dann auch n Elemente: Wir vereinigen (engl. *merge*) einfach zwei sortierte Teillisten, nämlich diejenige bestehend aus den (sortierten) ersten $\lfloor \frac{n}{2} \rfloor$ Elementen mit derjenigen bestehend aus den (sortierten) letzten $\lceil \frac{n}{2} \rceil$ Elementen.

Formulieren wir dies wieder in C, erhalten wir das folgende Programm[1]:

```
MergeSort(int* A, int n)
{
    if (n > 1) {
        /* Sortiere rekursiv die beiden Teile */
        MergeSort(&A[0], n/2);
        MergeSort(&A[n/2], (n+1)/2);
        /* Vereinige die beiden sortierten Teile */
        Merge(&A[0], n/2, n); }
}
```

Die Prozedur Merge soll die beiden vorsortierten Hälften des Arrays A vereinen. Sie könnte beispielsweise so aussehen:

```
Merge(int*A, int l, int m)
```

[1] Um die Darstellung übersichtlich zu halten, wurde auf zusätzliche Optimierungen verzichtet.

```
{ int i=0, j=l, k=0; int* B=(int*)malloc(m*sizeof(int));
    while ( (i < l) && (j < m) ) {
        if (A[i] < A[j]) {
            B[k] = A[i]; i++; k++; }
        else {
            B[k] = A[j]; j++; k++; }   }
    for (;i < l;i++)
        { A[m-l+i] = A[i]; }
    for (i=0; i < k; i++)
        { A[i] = B[i]; }
    free(B);
}
```

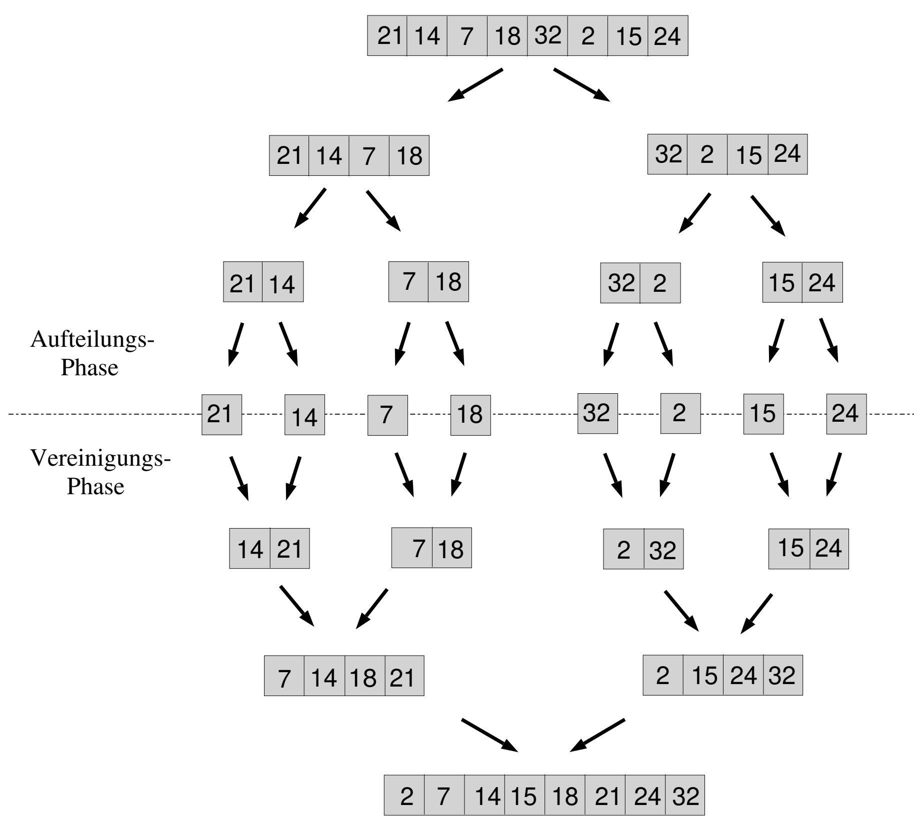


Abbildung 1.2: MergeSort angewandt auf $n = 8$ Elemente

Abbildung 1.1 veranschaulicht den Algorithmus MergeSort an einem Beispiel mit $n = 8$ Elementen, wobei wir diesmal die zu sortierenden Elemente nebeneinander dargestellt haben.

Bemerkung 1.1 *Dieses Prinzip des Aufteilens und anschließenden Wiedervereinens ist ein Verfahren, das an vielen Stellen auftritt. Bekannt ist es auch unter dem Namen* „Teile und Herrsche" *oder engl.* „Divide and Conquer"*. Wir werden es in Kapitel 3 nochmals ausführlich erläutern.*

Zurück zu unserem Szenario: wir haben nunmehr zwei Algorithmen entworfen, die das Verlangte leisten. Als Frage stellt sich nun: Welchen der beiden Algorithmen sollen wir verwenden?

Notwendig für eine fundierte Beantwortung dieser Frage ist eine **Analyse der Algorithmen**. Darunter werden wir im Verlaufe dieser Vorlesung ganz allgemein immer die folgenden beiden Punkte verstehen:

I) den Nachweis der *Korrektheit* des Algorithmus. Daß er also das Verlangte - in unserem Fall das Sortieren - auch wirklich für alle (zulässigen) Eingaben tut.

II) die Bestimmung der benötigten Ressourcen, bspw. Laufzeit und/oder Speicher.

Korrektheitsanalyse der Sortieralgorithmen
InsertionSort: Wir zeigen durch Induktion über die Anzahl der Iterationen der **for**-Schleife: *Nach i Iterationen gilt* $A[0] \leq A[1] \leq \cdots \leq A[i]$. Als Induktionsverankung wählen wir $i = 0$: hierfür ist die Behauptung trivialerweise richtig. Für den Induktionsschritt $i \Rightarrow i+1$ überlegen wir uns, daß sich an der Reihenfolge der ersten i Elemente untereinander nichts ändert und die **while**-Schleife andererseits genau dafür sorgt, daß das $i+1$-te Element an der "richtigen" Stelle eingefügt wird.

MergeSort: Hier kann man die Korrektheit beispielsweise durch Induktion über die Rekursionstiefe zeigen. Die Details wollen wir an dieser Stelle aber dem Leser überlassen.

Laufzeitanalyse von InsertionSort
Zunächst eine Vorbemerkung: Für eine genaue Analyse der Laufzeit müßten wir eigentlich noch einige zusätzliche Annahmen über Maschinentyp, etc. machen. Wir kommen im nächsten Abschnitt auf diese Problematik zurück. Für den Moment wollen wir einfach annehmen, daß jede der verwendeten Operationen (Addition, logischer Vergleich, Erhöhung-/Erniedrigung um Eins, Speicherzuweisung), eine Zeiteinheit benötigt.

Den Algorithmus InsertionSort können wir dann wie folgt analysieren:

	Kosten	Anzahl Ausführungen
InsertionSort(int* A, int n)		
{ int i, j, tmp;		
for $(i = 1; i < n; i++)$ {	2	n
$tmp = A[i]$;	1	$n-1$
$j = i - 1$;	2	$n-1$
while $((A[j] > tmp)$ && $(j >= 0))$ {	3	$\leq (2 + \cdots + n)$
$A[j+1] = A[j]; j--;$ }	3	$\leq (1 + \cdots + n - 1)$
$A[j+1] = tmp$; }	2	$n-1$
}		

Bezeichnen wir also mit $T_{InsertionSort}(n)$ die Laufzeit von InsertionSort bei n Elementen, so erhalten wir:

$$\begin{aligned} T_{InsertionSort}(n) &\leq 2n + (1+2+2)\cdot(n-1) + 3\cdot \underbrace{\sum_{i=2}^{n} i}_{=\frac{1}{2}n(n+1)-1} + 3\cdot \underbrace{\sum_{i=1}^{n-1} i}_{=\frac{1}{2}(n-1)n} \\ &= 3n^2 + 7n - 8. \end{aligned}$$

Beachte: Der Wert $3n^2 + 7n - 8$ ist nur eine *obere Schranke* für die Laufzeit. Für einzelne Daten kann der Algorithmus durchaus schneller sein. (Man überlege sich beispielsweise die Laufzeit, falls die Daten bereits zu Beginn richtig sortiert sind!) Andererseits überlegt man sich auch leicht ein Beispiel, für das die Laufzeit von InsertionSort auch wirklich $3n^2 + 7n - 8$ ist. (Wie sieht dies aus?!)

Laufzeitanalyse von MergeSort

Zur Bestimmung der Laufzeit von MergeSort müssen wir zunächst die Laufzeit der Hilfsprozedur Merge abschätzen. Dies geht ganz ähnlich wie eben, und wir wollen die Details an dieser Stelle dem Leser überlassen und halten nur das Ergebnis fest:

$$T_{Merge}(m) \le 10(m+1).$$

Die Laufzeit $T_{MergeSort}$ von MergeSort bei n Elementen kann man damit wie folgt abschätzen, wobei wir der Einfachheit halber annehmen wollen, daß $n = 2^{\log_2 n}$ eine 2er-Potenz ist:

$$\begin{aligned}
& T_{MergeSort}(n) && (1.1)\\
\le\ & 2T_{MergeSort}(\frac{n}{2}) + 10(n+1) \\
=\ & 2(2T_{MergeSort}(\frac{n}{4}) + 10(\frac{n}{2}+1)) + 10(n+1) \\
& \vdots \\
=\ & 2\Big(\ldots\Big(\underbrace{2\,T_{MergeSort}(\frac{n}{\underbrace{2^{\log_2 n}}_{=1}})}_{=1} + 10(\frac{n}{2^{\log_2 n-1}}+1)\Big) + \ldots + 10(\frac{n}{2}+1)\Big) + 10(n+1) \\
=\ & 2^{\log_2 n} + \sum_{i=1}^{\log_2 n} 2^{i-1}\cdot 10(\frac{n}{2^{i-1}}+1) \\
=\ & n + 10\sum_{i=1}^{\log_2 n}(n + 2^{i-1}) \ = \ n + 10n\log_2 n + 10\underbrace{\sum_{i=0}^{\log_2 n-1} 2^i}_{=n-1} \\
=\ & 10n\log_2 n + 11n - 10. && (1.2)
\end{aligned}$$

(Für Zahlen n, die keine 2er-Potenz sind, kann man mit etwas mehr technischem Aufwand eine ähnliche Schranke zeigen. Wir wollen uns an dieser Stelle diese Mühe aber ersparen und nehmen einfach mal an, die Schranke aus (1.1) gelte für alle n.)

Wie schon bei InsertionSort, gilt auch hier: für einzelne Datensätze kann die Laufzeit erheblich schneller sein, als in (1.1) angegeben. Wiederum gilt aber auch: es gibt Datensätze (welche?), für die die Laufzeit auch wirklich so groß ist, wie in (1.1) angegeben.

Kehren wir abschließend noch einmal zu unserem Szenario zurück. Wir wollten 10 Millionen Elemente sortieren:

InsertionSort benötigt hierfür unter der (unrealistischen!) Annahme, daß jede der Operationen für die wir eine Zeiteinheit veranschlagt haben, nur einen Maschinenbefehl benötigt, auf einer 100 MIPS[2] Maschine

$$\begin{aligned}(\,3 \cdot (10^7)^2 + 7 \cdot 10^7 - 8\,)\ /\ 10^8 &\approx 3 \cdot 10^6 \text{ Sekunden}\\ &\approx 34,8 \text{ Tage.}\end{aligned}$$

MergeSort andererseits bewältigt die gleiche Aufgabe in etwa

$$(\,10^8 \cdot \log_2 10^7 + 11 \cdot 10^7 - 10\,)\ /\ 10^8 \approx 24 \text{ Sekunden!}$$

Bemerkung 1.2 *Es gibt noch andere/bessere Sortierverfahren. Einige davon werden wir in Kapitel 4 kennenlernen.*

1.2 Analyse von Algorithmen

1.3 Elementare Datenstrukturen

1.4 Amortisierte Laufzeit

[2]MIPS = Million Instructions Per Second

Kapitel 2

Höhere Datenstrukturen

Ausarbeitung: *Jens Ernst*

Szenario: Gegeben seien eine Menge von *Datensätzen* $x_1, \ldots, x_n$, wobei jeder Datensatz x durch einen Schlüssel $\texttt{key}[x]$ gekennzeichnet ist. Gesucht ist eine Datenstruktur S, die die folgenden Operationen (bzw. einen Teil davon) möglichst effizient realisiert:

Find(k, S)	liefert $x \in S$ mit $\texttt{key}[x] = k$ bzw. *nil*
Insert(x,S)	fügt den Datensatz x ein
Delete(x,S)	löscht den Datensatz x
DecreaseKey(x,Δ,S)	ersetzt $\texttt{key}[x]$ durch $\texttt{key}[x] - \Delta$
FindMin(S)	liefert $x \in S$ mit $\texttt{key}[x] = \min\{\texttt{key}[y] \mid y \in S\}$
FindMax(S)	analog
ExtractMin(S)	wie FINDMIN(S), der ausgegebene Datensatz wird gelöscht
Union(S_1,S_2)	vereinigt die beiden Datenstrukturen S_1 und S_2 zu einer neuen Datenstruktur S

Für diese Anforderungungen gibt es verschiedene Realisierungsmöglichkeiten. Diese hängen u.A. davon ab, welche der Operationen unterstützt werden sollen und was für Annahmen an das *Universum* U, aus dem die Schlüssel kommen, man macht. Letzteres kann beispielsweise vollständig geordnet sein, nur teilweise geordnet, oder auch gänzlich ungeordnet. Weiter können die Schlüssel eindeutig sein (d.h., zu jedem Zeitpunkt gibt es immer nur höchstens ein Element, das einen bestimmten Schlüssel hat) oder eben nicht, usw.

Grob unterscheidet man zwischen den folgenden Datenstrukturen:

1. *"Suchbäume"*
   - Operationen: *Find, Insert, Delete, FindMin, FindMax, FindPred, FindSucc*
   - siehe Abschnitt 2.1
2. *"Vorrangwarteschlange" ("priority queue")*
   - Operationen: *Insert, Delete, ExtractMin, DecreaseKey*
   - siehe Abschnitt **??**

3. *"Mengen"*
   - Operationen: *Find, Union*
   - siehe Abschnitt **??**
4. *"Wörterbuch" ("dictionary")*
   - Operationen: *Find, Insert, Delete*
   - Implementierung als: hash–Tabellen, balancierte Bäume

2.1 Suchbäume

Liegen die betrachteten Schlüssel nicht in einem geordneten Universum U, so müssen beim Suchen alle Elemente betrachtet werden. Ist das Universum aber total[1] geordnet, so läßt sich die Suche nach einem bestimmten Element effizienter gestalten.

In diesem Abschnitt wollen wir einige Datenstrukturen vorstellen, die eine solche effiziente Suche erleichtern: Suchbäume. Dazu treffen wir folgende

Annahmen: *1.* Die Schlüssel sind aus einem total geordneten Universum; dies trifft beispielsweise auf Zahlen zu, aber auch auf Strings mit der lexikographischen Ordnung.
2. Jeder Schlüssel kommt höchstens einmal vor. (Dies ist keine unbedingt notwendige Annahme, vereinfacht aber an einigen Stellen die Darstellung.)

Definition: Ein *Suchbaum* ist ein Baum, bei dem an den Knoten Datenelemente und/oder Verwaltungsinformationen gespeichert werden, die ein effizientes Suchen erleichtern. Bei einem *externen Suchbaum* werden die Datenelemente nur an den Blättern gespeichert (siehe z.B. (a,b)–Bäume). Bei *internen Suchbäumen* liegen die Datenelemente an den internen Knoten. Zeiger auf Blätter sind NIL–Pointer. Daher kann man auch sagen: An *allen* Knoten sind Daten gespeichert.

2.1.1 Höhenbalancierte Bäume — (a, b)-Bäume

Definition: Ein *(a,b)–Baum* ist ein externer Suchbaum, bei dem alle Blätter dieselbe Tiefe haben, und für den gilt:

- $a \leq \#$ Kinder $\leq b$ an allen Knoten außer der Wurzel
- $2 \leq \#$ Kinder $\leq b$ an der Wurzel
- $b \geq (2a - 1)$
- Für alle inneren Knoten v bezeichne $\rho(v)$ die Anzahl Kinder von v.

[1] Bei totaler Ordnung ist jedes Element mit jedem auf $<, =, >$ vergleichbar.

- An jedem inneren Knoten v werden folgende Verwaltungsinformationen eingetragen: $\rho(v)-1$ Schlüssel $K_1,\ldots,K_{\rho(v)-1}$, so daß gilt
  $K_{i-1} <$ alle Schlüssel im i–ten Unterbaum von $v \leq K_i$,
  hierbei sei $K_0 = -\infty$ und $K_{\rho(v)} = +\infty$

Die von (a,b)–Bäumen unterstützten Operationen sind *Find*, *Insert* und *Delete*.

Bemerkung 2.1 *Für den Spezialfall $a = 2, b = 3$ ergeben sich die sog. (2,3)–Bäume, und im Falle $b = 2a-1$ spricht man von B–Bäumen.*

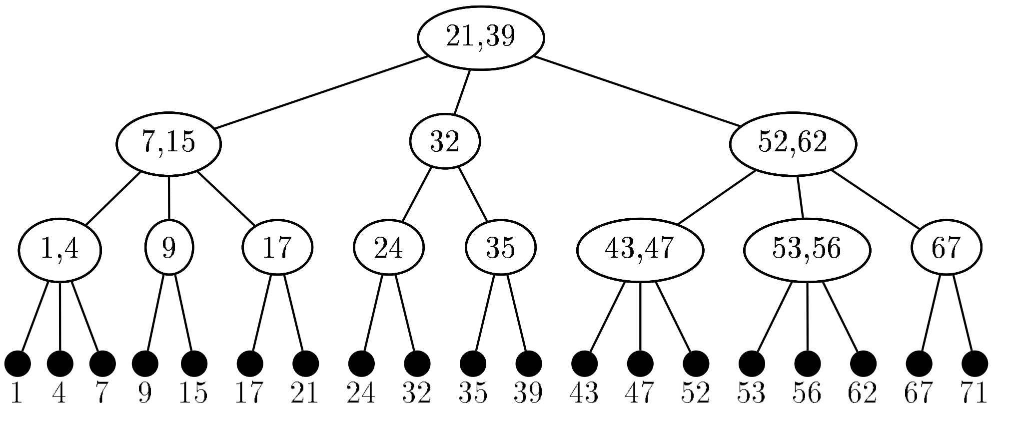


Abbildung 2.1: Beispiel eines (2,3)-Baumes

Theorem 2.2 *Sei T ein (a,b)–Baum mit n Blättern und Höhe h. Dann gilt:*

(i) $2 \cdot a^{h-1} \leq n \leq b^h$

(ii) $\log_b(n) \leq h \leq 1 + \log_a(\frac{n}{2})$

Beweis: An jedem inneren Knoten, der nicht die Wurzel von T ist, gilt für jeden Unterbaum: $a^h \leq n \leq b^h$, wobei h die Höhe des Unterbaums ist und n die Anzahl seiner Blätter. Denn: für $h = 0$ ist $n = 1 = a^0 = b^0$. Für den Induktionsschritt habe nun v mindestens a und höchstens b Kinder, und für den Unterbaum eines jeden gelte, daß er zwischen a^{h-1} und b^{h-1} Blätter habe. Dann hat der Unterbaum an v zwischen $a \cdot a^{h-1}$ und $b \cdot b^{h-1}$ Blätter. Also gilt obige Behauptung.
An der Wurzel liegen mindestens 2 und höchstens b Kinder vor, und deren Unterbäume haben die Höhe $h-1$, also je zwischen a^{h-1} und b^{h-1} Blätter. Somit liegt die Zahl der Blätter zwischen $2 \cdot a^{h-1}$ und $b \cdot b^{h-1} = b^h$, und das ist die Behauptung *(i)*. Für *(ii)* logarithmiere man jeweils die beiden Teilungleichungen, *q.e.d.* ∎

Operationen auf (a,b)–Bäumen

Find(k, T):

Es werden die an den internen Knoten abgelegten Verwaltungsinformationen verwendet, für ein Verfahren analog der binären Suche.

```
v := Wurzel von T;
while (v nicht Blatt) do
    i := min{j | 1 ≤ j ≤ ρ(v) und k ≤ K_j};
    v := i-tes Kind von v
od;
if key[v] = k
    then return v
    else return nil
fi;
```

Bemerkung: Da, wie oben gezeigt, ein (a,b)–Baum logarithmische Höhe hat und die Zahl der Schritte zur Minimumsbildung durch $b = O(1)$ beschränkt ist, ist *Find* in Zeit $O(\log(n))$ ausführbar, wobei natürlich a und b konstant sind.

Insert(x, T):

Das Vorgehen für das Einfügen eines neuen Blattes ist folgendes:

1. Führe *Find(x)* aus. Diese Suche führt zu einem Blatt w. Falls `key`$[w]$ = `key`$[x]$: STOP, diesen Schlüssel gibt es schon. Ansonsten ist w das Datenelement mit dem kleinsten Schlüssel, der größer ist als der von x.

2. In diesem Falle füge x links vom gefundenen Blatt w ein. Hierdurch erhöht sich der Verzweigungsgrad $\rho(v)$ des gemeinsamen Vaters v von x und w.

3. Wird hierdurch $\rho(v) > b$, so führe folgende Rebalancierung aus:

```
v := gemeinsamer Vater von x und w;
while ρ(v) > b do
    if (v ≠ Wurzel ) then u := Vater von v
    else u:= neue Wurzel mit einzigem Kind v;
    Zerlege die Menge G aller Kinder von v in Mengen G_1 (mit den ⌈(b+1)/2⌉
    kleinsten Kindern) und G_2 (übrigen Kindern);
    'Zerteile' Vater v in zwei Väter v_1,v_2 und mache v_1 zu Vater von G_1 und
    v_2 zu Vater von G_2; (*)
    v:=u; (**)
od
```

Bemerkung: `(*)` Aus dem Sohn v von u werden zwei Söhne v_1 und v_2, also erhöht sich der Grad $\rho(u)$ um eins. `(**)` Wird er größer als b, also gleich $(b + 1)$, so ist ein analoges Vorgehen an u nötig, u.s.w. Dies wird durch die Anweisung `v:=u` und die `while`–Schleife bewirkt. Die Verwaltungsinformationen an den behandelten Knoten sind jeweils neu zu berechnen.

Korrektheit der Teilung: Eine Teilung der Kindermenge eines Knotens und seine Aufspaltung in zwei Knoten ist notwendig, wenn die Kinderzahl den Wert $(b+1)$ erreicht. Durch die Teilung wird $|G_1| = \lceil \frac{b+1}{2} \rceil$ und $|G_2| = \lfloor \frac{b+1}{2} \rfloor$. Liegen diese nun im zulässigen Bereich zwischen a und b? Es gilt:

$$\lceil \tfrac{b+1}{2} \rceil \leq b \Leftrightarrow \tfrac{b+1}{2} \leq b \Leftrightarrow b \geq 1$$

$$\lfloor \tfrac{b+1}{2} \rfloor \geq a \Leftrightarrow \tfrac{b+1}{2} \geq a \Leftrightarrow b \geq 2a+1$$

Letzteres gilt nach Voraussetzung.

Bemerkung: Der Zeitaufwand für *Insert* ist $O(\log(n))$, wobei n die Anzahl der Blätter des Baums T ist[2].

Beweis: Die Operation *Find* (Laufen zu einem Blatt) kostet Zeit $O(\log(n))$. Eine Teilungsoperation muß auf jeder Ebene des Baums $\leq 1\times$ ausgeführt werden und kostet jedesmal konstante Zeit. Da die Höhe des Baums $O(\log(n))$ ist, folgt die Behauptung, *q.e.d.*

Delete(x, T):
Analog zu *Insert* erfordert *Delete* auch ein Vorgehen in drei Schritten:

1. Führe *Find* aus und laufe dadurch zu einem Blatt.
2. Falls das Blatt das Element x enthält, so lösche es. Dadurch verkleinert sich der Grad $\rho(v)$ des Vaters v.
3. Wird $\rho(v) < a$, so wird Rebalancierung nötig. Für diese gibt es, wenn man einen bestimmten Nachbarn von v betrachtet, zwei Fälle:
   (i) Falls der Nachbar von v den Grad $\leq a$ hat, so verschmelze v mit diesem Nachbarn.
   (ii) Falls der Nachbar von v den Grad $> a$ hat, so soll v ein Kind von seinem Nachbarn adoptieren.

   Bemerkung: Im Fall *(i)* wird der Grad des Vaters der beiden verschmolzenen Knoten kleiner, und eventuell $< a$, so daß sich die Verletzung der Balancierungsbedingung nach oben fortpflanzen kann. Im Fall *(ii)* gilt dies sicher nicht. Da nun v i.a. zwei Nachbarn hat, hat man in manchen Fällen die Wahl zwischen beiden Methoden. In diesem Fall wählt man natürlich Methode *(ii)*.

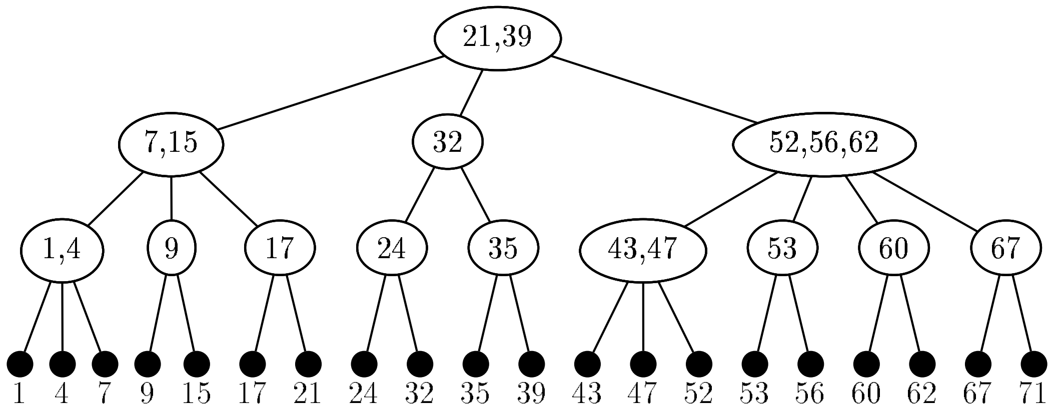
Theorem 2.3 *Für (a,b)–Bäume benötigen alle Operationen, Find, Insert, Delete die Zeit* $O(\log(n))$.

Bemerkung (Wahl von a):

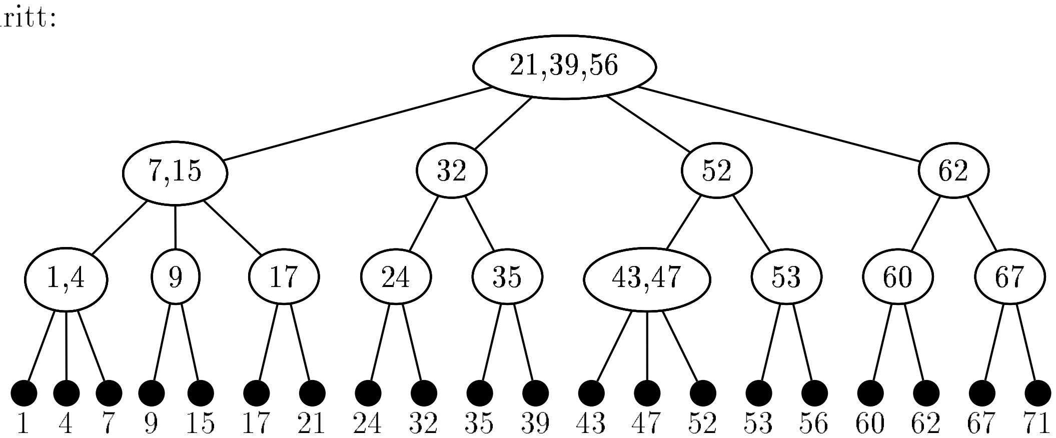
- Wird der (a,b)–Baum im Hauptspeicher gehalten, so wähle a *klein*, dann sind die Kosten für das Suchen minimal.

[2] Zur Erinnerung: Die Blätter haben alle dieselbe Tiefe.

1.Schritt:



2.Schritt:



3.Schritt:

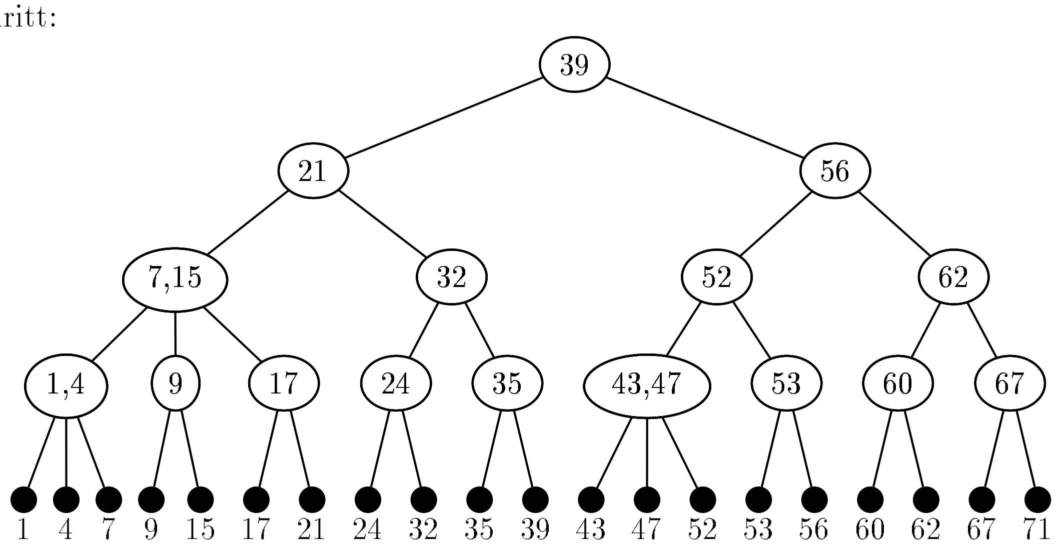


Abbildung 2.2: Beispiel für die Ausführung einer *Insert*-Operation: Wir fügen in den (2,3)-Baum aus Abbildung 2.1 den Schlüssel 60 ein.

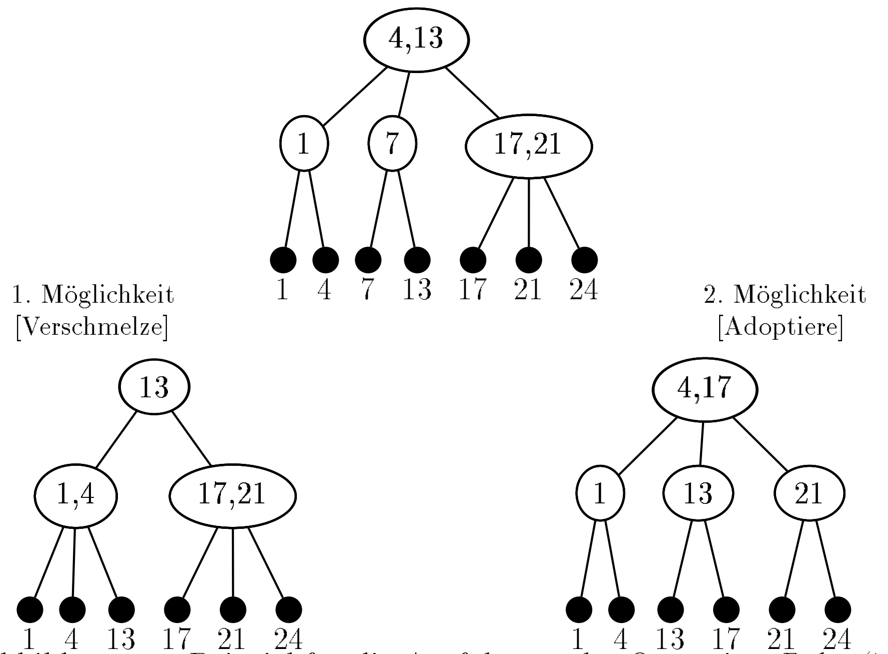


Abbildung 2.3: Beispiel für die Ausführung der Operation $Delete(7)$

- Ist er dagagen im Sekundärspeicher abgelegt, wo die Zugriffszeit für das erste byte eines Zugriffs sehr groß und für die restlichen bytes klein ist, so wähle *a groß*, so daß möglichst wenig erste Zugriffe stattfinden.

2.1.2 Selbstorganisierte binäre Suchbäume — Splay–Trees

Ausarbeitung: *Alexander Hall*

Motivation, Grundidee

Da in der Realität vorallem die Laufzeit von vielen aufeinanderfolgenden Operationen und nicht die Laufzeit einer einzelnen Operation zählt, wäre ein interner Suchbaum mit möglichst geringen amortisierten Kosten wünschenswert.

Dabei wäre eine Möglichkeit, nach der sogenannten „Move–To–Root“–Heuristik vorzugehen: Ein Element, auf das eine Zugriffsoperation angewandt wird, wird zur Wurzel befördert. Auf dieses kann dann später schneller zugegriffen werden. Das ist vorteilhaft, wenn man davon ausgeht, daß in der Praxis häufig mehrfach hintereinander auf dasselbe Element zugegriffen wird.

Die Heuristik läßt sich mit Rechts-/Linksrotationen (wie bei AVL–Bäumen, siehe Abbildung 2.4) durchführen ([**?**], [**?**]). Allerdings reicht dies noch nicht aus, um die worst–case Kosten zu reduzieren. Dazu wird „Move–To–Root“ mit paarweisen Rotationen durchgeführt, die gleichzeitig den Baum balancieren $\rightsquigarrow$ splaying ([**?**]).

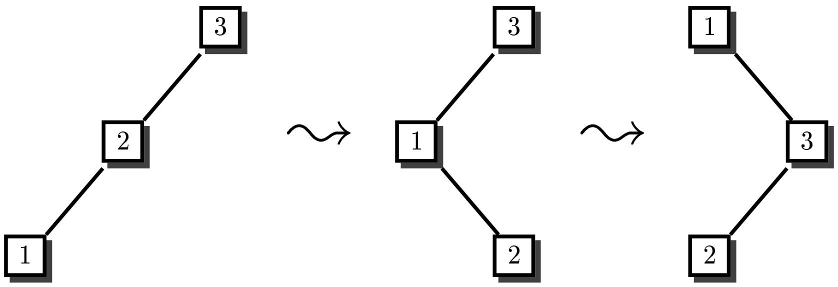


Abbildung 2.4: Beispiel für einfache Rechts-/Linksrotationen

Implementierung

Splay–Trees werden als binäre Suchbäume umgesetzt, d.h. jeder Knoten hat höchstens zwei Söhne, alle Knoten im linken Unterbaum haben einen kleineren Schlüssel, alle im rechten Unterbaum haben einen größeren Schlüssel. Es gibt keine „expliziten“ Forderungen an die Höhenbalanciertheit oder dergleichen. Die noch zu definierenden Operationen sollen den Baum jedoch möglichst gut „aufräumen“.

Splay(x,T) Macht x durch eine Reihe von *Splay–Operationen* zur Wurzel des Baums T. Für jede *Splay Operation* gibt es die in Abbildung 2.5 zu sehenden drei Möglichkeiten.

Abbildung 2.6 und Abbildung 2.7 zeigen die Aktionen, die bei den jeweiligen Fällen angestoßen werden.

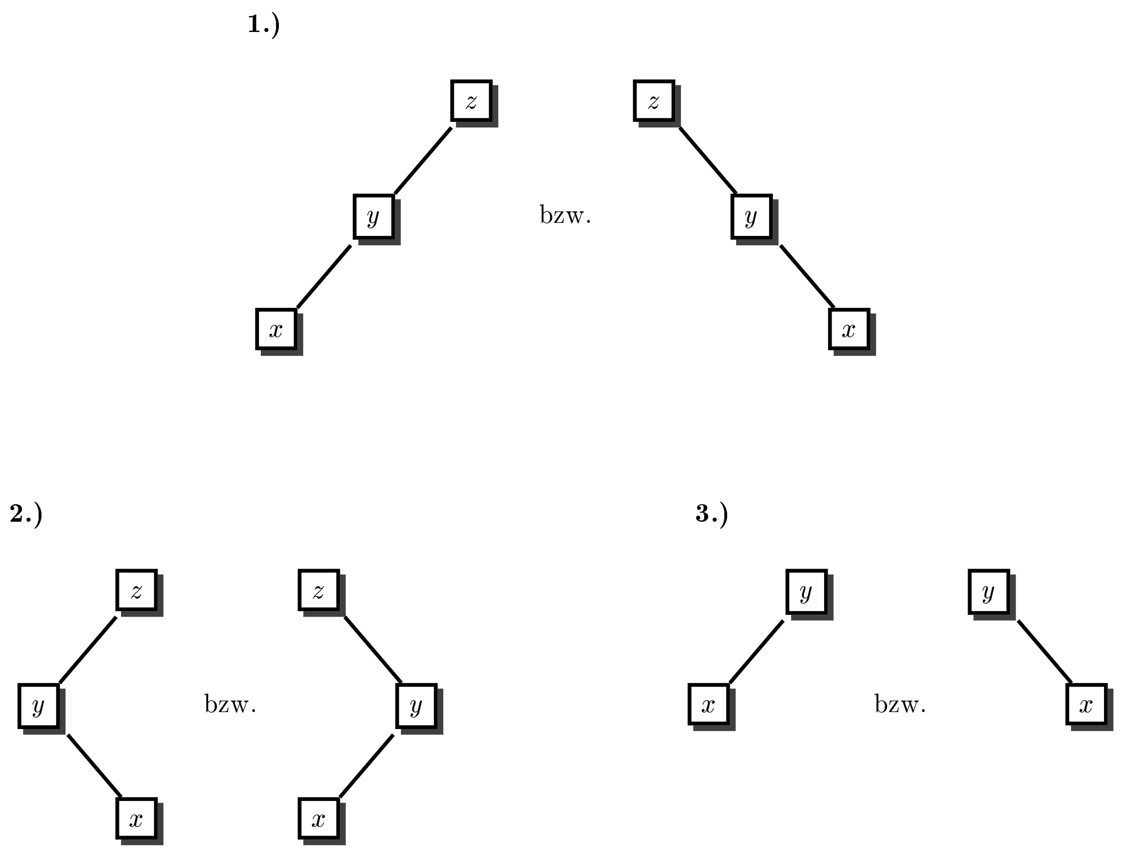
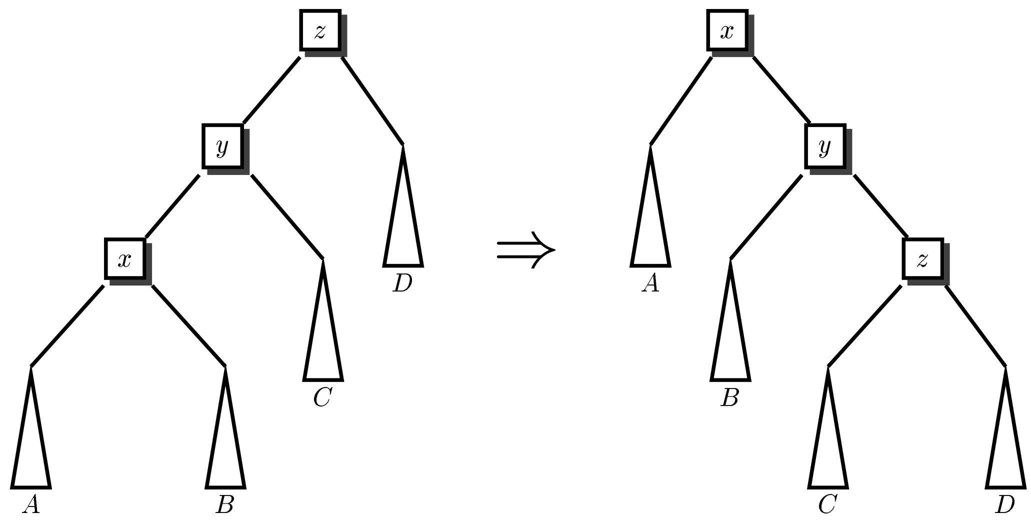


Abbildung 2.5: Mögliche Ausschnitte aus einem Baum

zu 1:



zu 2:

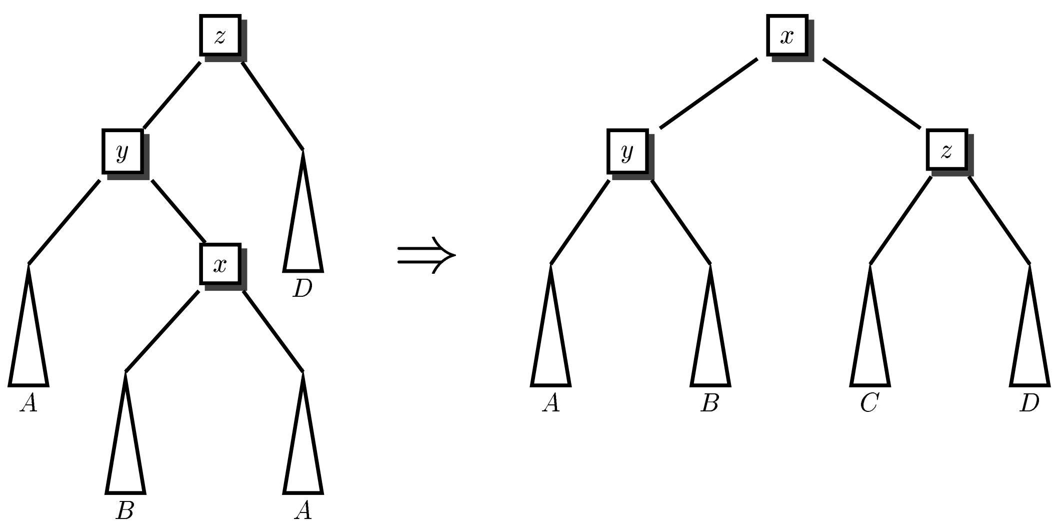


Abbildung 2.6: Aktionen, um x nach oben zu transportieren

zu 3:

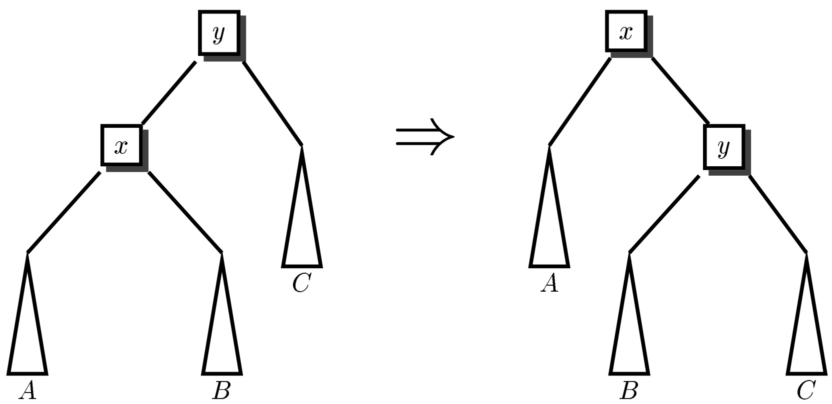


Abbildung 2.7: Aktionen, um x nach oben zu transportieren

Laufzeitanalyse von Splay(x,T)

Notation $\forall v \in T$

- size(v) = Anzahl der Knoten im Unterbaum mit Wurzel v
- $\varphi(v) = \log_2(\text{size}(v))$, Potential von v
- $\phi(T) = \sum\limits_{v \in T} \varphi(v)$, Potential des Baumes

Lemma 2.4 *Die amortisierte Laufzeit einer* Splay–Operation *ist* $3(\varphi_{nach}(x) - \varphi_{vor}(x))$ *für Fall 1 und 2,* $3(\varphi_{nach}(x) - \varphi_{vor}(x)) + 1$ *für Fall 3.*

Dabei bezeichnen $\varphi_{vor}(x)$ bzw. $\varphi_{nach}(x)$ das Potential vor bzw. nach einer *Splay–Operation*. Das Potential ändert sich nur für x, y und z.

Beweis:

$$\underbrace{\text{amortisierte Laufzeit}}_{=:a} = \underbrace{\text{tatsächliche Laufzeit}}_{\text{o.E.: =2 für Fall 1 und 2, =1 für Fall 3}} + \text{Potentialzunahme}$$

1. Fall:

$$\begin{aligned}
a &= 2 \;+\; \varphi_{nach}(x) - \varphi_{vor}(x) \;+\; \varphi_{nach}(y) - \varphi_{vor}(y) \;+\; \varphi_{nach}(z) - \varphi_{vor}(z) \\
&\quad [\; \varphi_{nach}(x) = \varphi_{vor}(z) \qquad \varphi_{nach}(y) \leq \varphi_{nach}(x) \;] \\
&\leq 2 \;-\; \varphi_{vor}(x) \;+\; \varphi_{nach}(x) - \varphi_{vor}(y) \;+\; \varphi_{nach}(z) \\
&\quad [\; \varphi_{vor}(y) \geq \varphi_{vor}(x) \;] \\
&\leq 2 + \varphi_{nach}(x) + \varphi_{nach}(z) - 2\varphi_{vor}(x) \\
&= 2 + \varphi_{nach}(x) - 3\varphi_{vor}(x) + \varphi_{nach}(z) + \varphi_{vor}(x) \qquad (2.1)
\end{aligned}$$

$$\varphi_{nach}(z) + \varphi_{vor}(x) = \log_2(\text{size}_{nach}(z) \cdot \text{size}_{vor}(x)) \tag{2.2}$$

Wegen $a \cdot b \leq \left(\frac{a+b}{2}\right)^2$:

$$\begin{aligned} \text{size}_{nach}(z) \cdot \text{size}_{vor}(x) &\leq \left(\frac{\text{size}_{nach}(z) + \text{size}_{vor}(x)}{2}\right)^2 \\ &\leq \left(\frac{\text{size}_{nach}(x)}{2}\right)^2 \qquad (2.3) \end{aligned}$$

Gleichung (2.3) in (2.2):

$$\begin{aligned} \varphi_{nach}(z) + \varphi_{vor}(x) &\leq 2\log_2(\text{size}_{nach}(x)) - 2\log_2 2 \\ &= 2\varphi_{nach}(x) - 2 \qquad (2.4) \end{aligned}$$

Gleichung (2.4) in (2.1):

$$\begin{aligned} a &\leq 2 + \varphi_{nach}(x) - 3\varphi_{vor}(x) + 2\varphi_{nach}(x) - 2 \\ &= \underline{3(\varphi_{nach}(x) - \varphi_{vor}(x))} \end{aligned}$$

2. Fall:

$$\begin{aligned} a &= 2 + \varphi_{nach}(x) - \varphi_{vor}(x) + \varphi_{nach}(y) - \varphi_{vor}(y) + \varphi_{nach}(z) - \varphi_{vor}(z) \\ &\quad [\ \varphi_{nach}(x) = \varphi_{vor}(z) \qquad \varphi_{vor}(y) \geq \varphi_{vor}(x)\ ] \\ &\leq 2 + \underbrace{\varphi_{nach}(y) + \varphi_{nach}(z)}_{\leq 2\varphi_{nach}(x)-2 \quad \text{analog zu Fall 1}} - 2\underbrace{\varphi_{vor}(x)}_{\geq 0} \\ &\leq \underline{3(\varphi_{nach}(x) - \varphi_{vor}(x))} \end{aligned}$$

3. Fall:

$$\begin{aligned} a &= 1 + \varphi_{nach}(x) - \varphi_{vor}(x) + \varphi_{nach}(y) - \varphi_{vor}(y) \\ &\quad [\ \varphi_{nach}(x) = \varphi_{vor}(y) \qquad \varphi_{nach}(y) \leq \varphi_{nach}(x)] \\ &\leq 1 + \underbrace{\varphi_{nach}(x) - \varphi_{vor}(x)}_{\geq 0} \\ &\leq \underline{1 + 3(\varphi_{nach}(x) - \varphi_{vor}(x))} \end{aligned}$$

$\Rightarrow$ für alle drei Fälle stimmt die Behauptung. ■

Korollar 2.5 *Die amortisierten Kosten von* Splay(x,T) *sind höchstens* $1 + 3\log_2 n$*, wobei* n *die Anzahl der Knoten in* T *sei.*

Beweis: Sei $\varphi_i(x)$ das Potential von x *vor* Beginn der i-ten von insgesamt, sagen wir, k Splay–Operationen.
Nach Lemma 2.4:

$$\text{amortisierte Laufzeit} = \sum_{i=1}^{k} (3\varphi_{i+1}(x) - 3\varphi_i(x)) + 1$$

Die 1 kommt dabei durch den dritten Fall zustande. Dieser tritt höchstens einmal auf, falls sich x genau unter der Wurzel von T befindet.

$$\text{amortisierte Laufzeit} = 3\underbrace{\varphi_{k+1}(x)}_{\leq \log_2 n} - 3\underbrace{\varphi_1(x)}_{\geq 0} + 1$$

D.h. Splay(x,T) hat die amortisierten Kosten von $1 + 3\log_2 n$. ■

Operationen für Splay–Trees

Find(k,T) Den Knoten mit dem Schlüssel k finden.

Implementierung:

Suchpfad von der Wurzel zu Knoten v bestimmen, wobei entweder key(v) = k
oder, falls so ein Knoten nicht existiert, key(v) der größte Schlüssel kleiner k
oder der kleinste Schlüssel größer k in T ist;
Splay(v,T);
if (key(v) = k)
    return v;
else
    return $NULL$;

Split(k,T) T in zwei Bäume T_1 und T_2 aufspalten, wobei T_1 alle $x \in T$ mit key(x) $\leq k$ enthält und T_2 alle übrigen Knoten.

Implementierung:

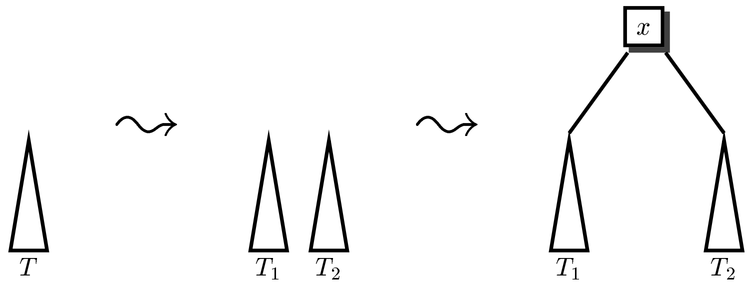
Find(k,T);
if (key (root (T)) $\leq k$)
    breche rechte Kante der Wurzel auf;
else
    breche linke Kante der Wurzel auf;

Insert(x,T) x in den Baum T einfügen.

Implementierung:

/*x zur Wurzel und T zerlegen:*/
(T_1,T_2) = Split(key(x),T);
if (key (root (T_1)) = key (x))
    STOP; /*Schlüssel existiert bereits*/
else
    mache x zur Wurzel von T mit den beiden UB T_1 und T_2;

Skizze:

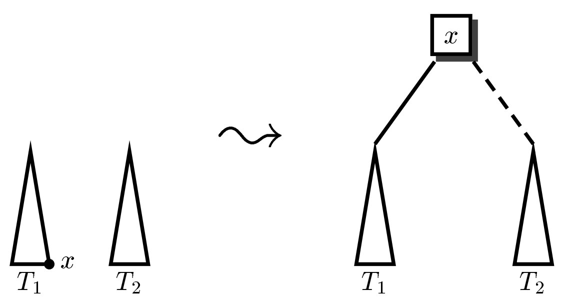


Join(T_1,T_2) Verschmelzen von T_1 und T_2 zu einem Baum T, wobei key$(x) \leq$ key(y) $\forall x \in T_1, y \in T_2$.

Implementierung:

bestimme größten Knoten x in T_1: laufe von Wurzel jeweils rechte Kante hinunter;
Splay(x,T_1);
mache Wurzel von T_2 zum Nachfolger von x;

Skizze:



Delete(k,T) Knoten mit Schlüssel k löschen, falls vorhanden.

Implementierung:

Find(k,T);
if (key (root (T)) $\neq k$)
    STOP; /*Schlüssel nicht vorhanden*/
else {
    T_1 und T_2 Unterbäume der Wurzel von T;
    T = Join(T_1,T_2);
}

Die *amortisierte* Laufzeit jeder dieser Operation ist offenbar jeweils $\mathcal{O}(\log n)$.

Theorem 2.6 *Jede* beliebige *Folge von k* Find, Insert, Delete, Join und Split *Operationen hat* worst case *Laufzeit von* $\mathcal{O}((n+k) \cdot \log n)$*, wobei n die maximale Anzahl der gleichzeitig im Splay–Tree vorhandenen Knoten sei.*

Beweis: Wiederholung: $\phi(T) = \sum_{x \in T} \varphi(x)$ (siehe **??**)

$$\begin{aligned}\text{worst case Laufzeit} &\leq \text{amortisierte Laufzeit} + \text{Potentialabnahme} \\ &= \mathcal{O}(k \cdot \log n) \quad + \quad \phi(T_{Beginn}) - \phi(T_{Ende})\end{aligned}$$

D.h. es ist zu zeigen, daß der maximale Potentialunterschied $\mathcal{O}(n \cdot \log n)$ ist. Dazu überlegen wir uns, daß für $\phi(T)$ folgende untere und obere Schranke gilt:

- unter Schranke: $\phi(T) \geq 0$ (klar, da $\varphi(v) \geq 0 \quad \forall v \in T$)
- obere Schranke: $\phi(T) \leq n \cdot \log_2 n$ (klar, da $\varphi(v) \leq \log_2 n \quad \forall v \in T$)

Die worst case Laufzeit ist kleiner gleich $\mathcal{O}(k \cdot \log n) + \mathcal{O}(n \cdot \log n) = \underline{\mathcal{O}((n+k) \cdot \log n)}$. ■

2.2 Vorrangwarteschlangen – Priority Queues

2.2.1 Fibonacci Heaps

2.2.2 Buckets

Ausarbeitung: *Viktor Schuppan*

Eine relativ einfache Möglichkeit, Vorrangwarteschlangen zu implementieren, stellen Buckets dar. Dazu müssen folgende Annahmen erfüllt sein:

1. Schlüssel sind ganzzahlig.
2. Zu jedem Zeitpunkt gilt für die zu speichernden Elemente:

$$\text{größter Schlüssel} - \text{kleinster Schlüssel} \leq C.$$

Diese Bedingungen sind zum Beispiel beim Algorithmus von Dijkstra erfüllt. C ist hier gleich der maximalen Kantenlänge (siehe auch **??**).

1-level Buckets

1-level Buckets bestehen aus

- einem Array $b[C+1]$ zur Aufnahme der Buckets,
- einer Zahl *minvalue*, die gleich dem kleinsten gespeicherten Schlüssel ist,
- einer Zahl $0 \leq minpos \leq C$, die den Index des Buckets mit dem kleinsten Schlüssel enthält, und

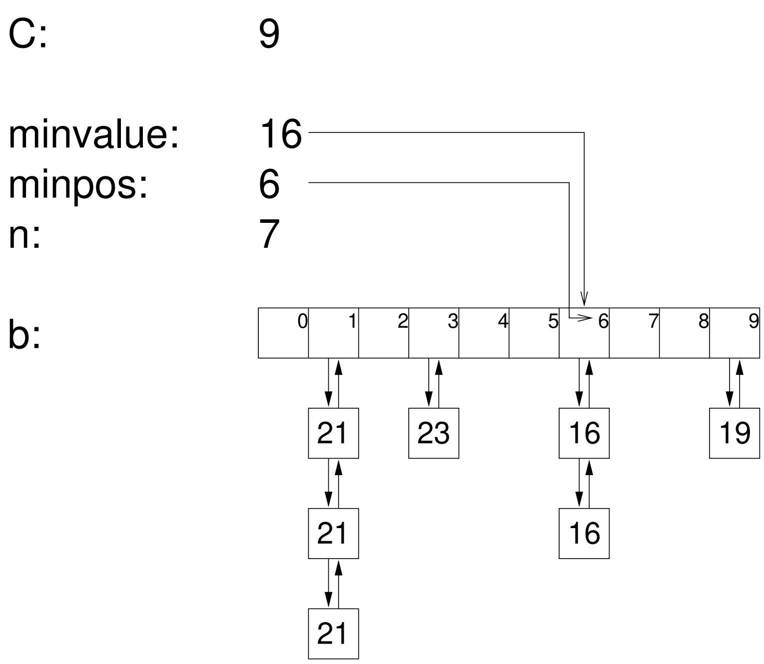


Abbildung 2.8: Ein Beispiel für 1-level Buckets

- einer Zahl n, die die Anzahl der gespeicherten Elemente enthält.

Ein Beispiel zeigt Abbildung 2.8.

Wie für jede Vorrangwarteschlange müssen drei Operationen unterstützt werden:

- *Insert(x)* fügt das Element x in die Vorrangwarteschlange ein,
- *ExtractMin* liefert das erste Element der Vorrangwarteschlange und löscht es und
- *DecreaseKey(x, k)* verringert den Schlüssel des Elementes x auf den Wert k.

Dazu kommt noch eine Initialisierung *Initialize*. Für 1-level Buckets sehen die Operationen[3] wie folgt aus:

Initialize
$n = 0$;
$minvalue = \infty$;

Insert(x)
Füge x in $b[key(x) \bmod (C+1)]$ ein;
$n = n + 1$;
if $(key(x) < minvalue)$
{

[3]Um die Darstellung übersichtlich zu halten, wurde auf jegliche Fehlerabfragen (zum Beispiel, ob die Vorrangwarteschlange leer ist oder Schlüssel im gültigen Bereich liegen) verzichtet.

// x ist jetzt das Element mit dem kleinsten Schlüssel
$minpos = key(x) \bmod (C+1)$;
$minvalue = key(x)$;
}

<u>ExtractMin</u>
Entferne ein beliebiges Element aus $b[minpos]$ und gebe es aus;
$n = n - 1$;
if $(n > 0)$
{
// Suche neues Element mit dem kleinsten Schlüssel
while ($b[minpos]$ ist leer)
$minpos = (minpos + 1) \bmod (C+1)$;
$minvalue =$ Schlüssel eines Elementes aus $b[minpos]$;
}
else
$minvalue = \infty$;

<u>DecreaseKey(x, k)</u>
Entferne x aus dem derzeitigen Bucket;
$n = n - 1$;
$key(x) = k$;
$Insert(x)$;

Bei geeigneter Implementierung der Buckets, zum Beispiel als doppelt verkettete Listen, gilt:

Theorem 2.7 *Die worst case Laufzeiten bei 1-level Buckets sind* $\mathcal{O}(1)$ *bei* Insert *und* DecreaseKey *und* $\mathcal{O}(C)$ *bei* ExtractMin.

Beweis: (Skizze) Wird x am Anfang der Liste eingefügt, so müssen bei *Insert* nur einige Zeiger umgehängt sowie n, *minpos* und *minvalue* angepaßt werden. Bei *ExtractMin* kann ebenfalls das erste Element aus *b[minpos]* entfernt werden, was wieder nur ein paar Zeigeroperationen sind. Die Aktualisierung von n, *minpos* und *minvalue* benötigt auch nur konstante Zeit. Für das Finden des nächstkleinsten Elementes müssen aber möglicherweise alle weiteren Buckets betrachtet werden, im schlimmsten Falle C. Da bei *DecreaseKey* das Element x direkt übergeben wird, sind neben dem Einfügen auch nur wenige Zeigeroperationen und je eine Zuweisung an n und *key(x)* nötig. ∎

2-level Buckets

Bei einem großen Wertebereich der zu speichernden Schlüssel, das heißt bei großem C, und einer geringen Anzahl tatsächlich abgelegter Datenelemente sind 1-level Buckets in zweifacher Hinsicht ungünstig:

- Das Array b belegt statisch Speicherplatz der Größe $\mathcal{O}(C)$, obwohl nur ein kleiner Teil davon wirklich gebraucht wird.

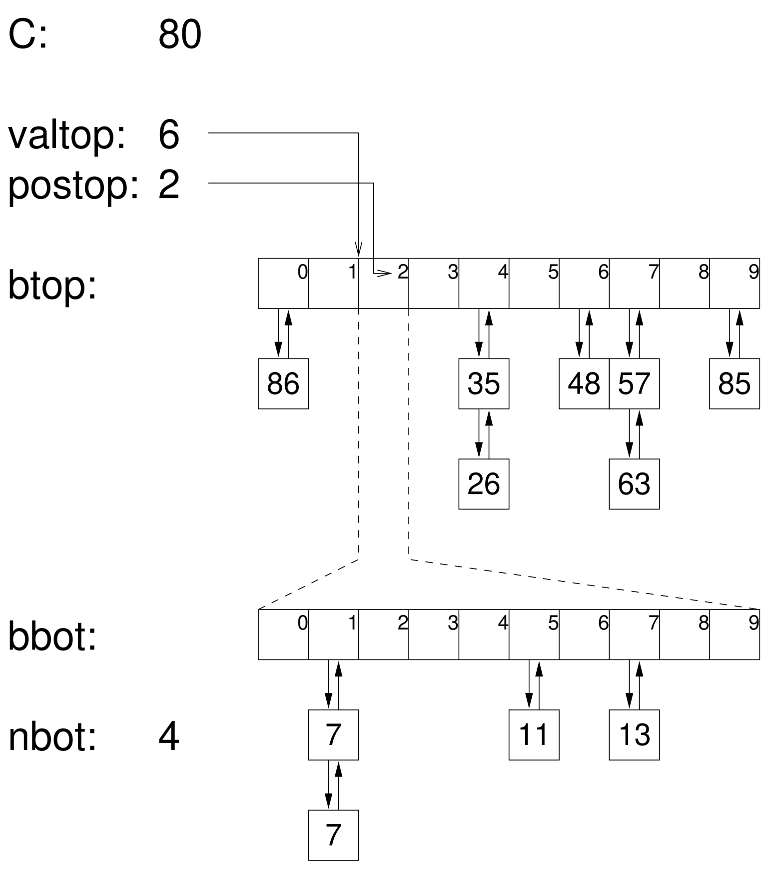


Abbildung 2.9: Ein Beispiel für 2-level Buckets

- Der Zeitbedarf für ein *ExtractMin* nähert sich der worst case Komplexität $\mathcal{O}(C)$, da der nächste nicht-leere Bucket ziemlich weit entfernt sein kann.

2-level Buckets versuchen dem mit folgender Idee abzuhelfen:

> Verwende statt eines Arrays der Länge $C+1$ zwei Arrays der Länge jeweils $B = \lceil\sqrt{C+1}\rceil + 1$.

Dabei enthält eines der beiden Arrays die meisten Elemente in grob vorsortierter Form, nur die Elemente mit den kleinsten Schlüsseln werden in dem zweiten Array dann endgültig sortiert vorgehalten. Ein Beispiel zeigt Abbildung 2.9.

2-level Buckets können mit folgenden Elementen aufgebaut werden:

- Das Array *btop* nimmt in jedem Bucket Elemente mit Schlüsseln aus einem Intervall der Länge B auf; die Schlüssel stammen aus allen Intervallen außer dem niedrigsten.

- Das Array *bbot* nimmt in jedem Bucket Elemente mit genau einem Schlüssel auf; die Schlüssel stammen nur aus dem niedrigsten Intervall.

- *valtop* ist die linke Intervallgrenze des niedrigsten Intervalls.
- *postop* enthält den Index des Buckets in *btop* für das niedrigste Intervall (das aber in *bbot* gespeichert wird).
- *nbot* ist die Anzahl der Elemente in *bbot*.

Dabei gilt:

- $btop[(postop + i) \bmod B]$ enthält alle Elemente x mit
$$valtop + iB \leq key(x) < valtop + (i+1)B, \text{ wobei } 1 \leq i \leq B-1$$
- $bbot[i]$ enthält alle Elemente x mit
$$key(x) = valtop + i, \text{ wobei } 0 \leq i \leq B-1$$

Dazu ist allerdings folgende zusätzliche Annahme zu machen:

> Hat ein *ExtractMin* ein Element mit Schlüssel k zurückgeliefert, werden bis zum nächsten Aufruf von *ExtractMin* nur Elemente mit Schlüsseln $\geq k$ und $\leq k + C$ eingefügt.

Dies stellt sicher, daß ein Element aus *bbot* nie zurück nach *btop* wandert.

Bemerkung 2.8 *Die Wahl von* B *stellt sicher, daß alle zulässigen Schlüssel Platz in der Vorrangwarteschlange finden.*

Beweis: Am meisten Platz wird benötigt, wenn der kleinste Schlüssel ganz rechts in bbot gespeichert oder dort entfernt worden ist; er hat dann den Wert $valtop + B - 1$. Der größte Schlüssel, der nun vorkommen kann und also Platz finden muß, ist $valtop + B - 1 + C$. Es gilt:

$$\begin{aligned}
& \text{größtmöglicher Schlüsselwert in btop} \\
= \;& valtop + BB - 1 \\
= \;& valtop + B + (B-1)B - 1 \\
= \;& valtop + B + \lceil\sqrt{C+1}\rceil(\lceil\sqrt{C+1}\rceil + 1) - 1 \\
\geq \;& valtop + B + C + 1 + \lceil\sqrt{C+1}\rceil - 1 \\
\geq \;& valtop + B - 1 + C \\
= \;& \text{größter zulässiger Schlüsselwert}
\end{aligned}$$

■

Im folgenden wird eine beispielhafte Implementierung der Operationen[4] für Vorrangwarteschlangen vorgestellt. Die einfache Initialisierung erfordert jedoch eine zusätzliche Bedingung:

[4] Um die Darstellung übersichtlich zu halten, wurde auf jegliche Fehlerabfragen (zum Beispiel, ob die Vorrangwarteschlange leer ist oder Schlüssel im gültigen Bereich liegen) verzichtet. Gleiches gilt für Optimierungen.

Vor dem ersten *ExtractMin* werden nur Elemente mit Schlüsseln ≥ 0 und $\leq C$ eingefügt.

Initialize
$valtop = postop = nbot = 0;$

Insert(x)
$i = (\lfloor \frac{key(x)-valtop}{B} \rfloor + postop) \bmod B$; // Bestimme den Index in *btop*
if $(i == postop)$
{
    Füge x in $bbot[key(x) - valtop]$ ein;
    $nbot = nbot + 1;$
}
else
    Füge x in $btop[i]$ ein;

ExtractMin
if $(nbot == 0)$
{
    // *bbot* ist leer
    while ($btop[postop]$ ist leer) // Suche nächsten nicht-leeren Bucket in *btop*
    {
        $postop = (postop + 1) \bmod B;$
        $valtop = valtop + B;$
    }
    while ($btop[postop]$ ist nicht leer) // Verteile die Elemente auf *bbot*
    {
        Entferne ein beliebiges Element x aus $btop[postop]$;
        Füge x in $bbot[key(x) - valtop]$ ein;
        $nbot = nbot + 1;$
    }
}
// Bestimme das kleinste Element in *bbot*
$j = 0;$
while ($bbot[j]$ ist leer)
    $j = j + 1;$
Entferne ein beliebiges Element aus $bbot[j]$ und gebe es aus;
$nbot = nbot - 1;$

DecreaseKey(x, k)
Entferne x aus dem derzeitigen Bucket;
if $(key(x) < valtop + B)$
    $nbot = nbot - 1;$
$key(x) = k;$
$Insert(x);$

Es gilt bei geeigneter Implementierung der Buckets:

Lemma 2.9 *Die worst case Laufzeiten bei 2-level Buckets sind* $\mathcal{O}(1)$ *bei* Insert *und* DecreaseKey *und* $\mathcal{O}(\sqrt{C}+n)$ *bei* ExtractMin.

Beweis: (Skizze) Die Indexbestimmung und die Aktualisierung von *nbot* bei *Insert* benötigen konstante Zeit. Zum Einfügen müssen ebenfalls nur wenige Zeiger neu belegt werden. Analoges gilt für *DecreaseKey*, da das Element x direkt übergeben wird. In *ExtractMin* muß möglicherweise zunächst der nächste nicht-leere Bucket in *btop* gesucht werden: $\mathcal{O}(B) = \mathcal{O}(\sqrt{C})$. Die darin enthaltenen maximal n Elemente müssen in *btop* einsortiert werden: $n\mathcal{O}(1) = \mathcal{O}(n)$. Dann muß noch das Minimum in *bbot* bestimmt werden: $\mathcal{O}(\sqrt{C})$. Es ergibt sich: $\mathcal{O}(\sqrt{C}) + \mathcal{O}(n) + \mathcal{O}(\sqrt{C}) = \mathcal{O}(\sqrt{C}+n)$. ∎

Damit läßt sich zeigen:

Theorem 2.10 *Die amortisierten Laufzeiten bei 2-level Buckets sind* $\mathcal{O}(1)$ *bei* Insert *und* DecreaseKey *und* $\mathcal{O}(\sqrt{C})$ *bei* ExtractMin.

Beweis:
Das Potential der 2-level Buckets sei

$$\Phi = c_0(\text{Gesamtanzahl Elemente} - \text{Anzahl Elemente in bbot}).$$

Insert: Φ nimmt höchstens um c_0 zu $\Rightarrow \mathcal{O}(1)$ amortisiert.

DecreaseKey: analog.

ExtractMin: Falls *postop* neu gesetzt wird, nimmt Φ um $c_0 t$ ab, wobei

$$t = \text{ Anzahl der neu zu verteilenden Elemente } = btop[postop].$$

Die tatsächliche Laufzeit ist $\mathcal{O}(\sqrt{C})+\mathcal{O}(t)$. Also beträgt die amortisierte Laufzeit $\mathcal{O}(\sqrt{C})$, falls c_0 hinreichend groß ist. ∎

Im Vergleich zu 1-level Buckets reduzieren 2-level Buckets den Speicherplatzbedarf von $\mathcal{O}(C)$ auf $\mathcal{O}(\sqrt{C})$ und die Komplexität der *ExtractMin* Operation von $\mathcal{O}(C)$ auf $\mathcal{O}(\sqrt{C})$ amortisiert.

Die Verallgemeinerung von 2-level Buckets führt zu *k-level Buckets*. Diese bestehen dann aus k Arrays der Länge $\mathcal{O}(\sqrt[k]{C})$. Dadurch lassen sich die Speicher- und Zeitkomplexität weiter verbessern, der Implementierungsaufwand steigt jedoch stark an.

Eine Untersuchung in [?] zeigt am Beispiel des Algorithmus von Dijkstra (siehe **??**), daß 2-level Buckets sehr häufig wesentlich schneller und fast nie erheblich langsamer als 1-level Buckets sind. Es wird auch deutlich, daß 2-level Buckets ein robusteres Verhalten zeigen als 1-level Buckets. Gleiches gilt für 3- und 4-level-Buckets, der Gewinn wird jedoch kleiner.

2.2.3 Radix–Heaps

Ausarbeitung: *Thomas Schickinger*

Grundidee

Radix–Heaps stellen eine Möglichkeit zur effizienten Realisierung von Priority–Queues dar, wobei ähnliche Randbedingungen wie bei den 2–level–Buckets vorausgesetzt werden. Dabei wird die amortisierte Laufzeit der langsamsten Zugriffsfunktion im Vergleich zu diesen verbessert, nämlich von $\mathcal{O}(\sqrt{C})$ auf $\mathcal{O}(\log C)$ (C bezeichne wie bei den 2–level–Buckets die maximale Differenz zwischen zwei Schlüsseln im Heap).

Die Grundidee besteht darin, anstelle einer Hierarchie von Buckets konstanter Größe (wie bei den 2- bzw. k–level–Buckets) Buckets mit exponentiell zunehmender Größe zu verwenden. Somit sind nur noch $\mathcal{O}(\log C)$ Buckets zur Verwaltung der Elemente im Heap nötig. Wie bei den 2–level–Buckets hängt die Laufzeit der „teueren“ Operationen direkt von der Anzahl der Buckets ab.

Implementierung

Randbedingungen Im einzelnen müssen für den Einsatz von Radix–Heaps (im folgenden kurz „Heap“) folgende Randbedingungen erfüllt sein:

- Die Schlüssel sind nicht-negative ganze Zahlen.
- Zu jedem Zeitpunkt gilt: Max. Schlüssel - min. Schlüssel $\leq C$.
- Wenn durch `extractMin` ein Element mit Schlüssel k ausgegeben wurde, so werden danach nur noch Elemente mit Schlüssel $\geq k$ eingefügt. *(Monotonie des Heaps)*

Interne Repräsentation Zur Abspeicherung eines Heaps werden folgende elementare Datenstrukturen benötigt:

Buckets	`b[0..B]`
Schranken der Buckets	`u[0..B+1]`
Bucket-Nummern der Elemente	`b_no[x]`, wobei `x` ein beliebiges Element des Heaps ist

Hierbei sei $B = \lceil \log(C + 1) \rceil + 1$.

Beachtenswert ist (im Vergleich zu 2–level–Buckets), daß jedes Element des Heaps auf seinen Bucket verweist (Rückwärtsverzeigerung durch `b_no[.]`). Dies war bei 2–level–Buckets nicht erforderlich.

Im folgenden bezeichne für jedes Element x des Heaps „key(x)“ seinen Schlüssel.

Invarianten Während der Lebenszeit des Heaps müssen mehrere Invarianten erfüllt bleiben:

1. Einordnung der Elemente in die Buckets:
   $\forall 0 \leq i \leq B:$ `u[i]` $\leq$ key(x) $<$ `u[i+1]` für alle Elemente x in `b[i]`

2. Exponentielles Wachstum der Bucketgrößen:
   - `u[0]` = `u[1]`-1
   - $\forall\, 1 \leq i < B: \quad 0 \leq$ `u[i+1]`-`u[i]` $\leq 2^{i-1}$

3. Monotonie der Schlüssel: Es werden immer nur Elemente x mit key(x) $\geq$ `u[0]` eingefügt

Initialisierung Die Initialisierung der internen Datenstrukturen erfolgt in der Prozedur `Initialize`. Hierbei werden leere Buckets erzeugt und die Bucketgrenzen `u[i]` entsprechend der Invariante 2 gesetzt.

```
Initialize
// Alle Buckets sind leer
for (i = 1; i++; i ≤ B) {
    b[i] = empty;
}
// Bucketgrenzen setzen
u[0] = 0; u[1] = 1;
for (i = 2; i++; i ≤ B) {
    u[i] = u[i-1] + 2^(i-2)
}
u[B+1] = ∞;
```

Abbildung 2.10 zeigt einen leeren Radix–Heap.

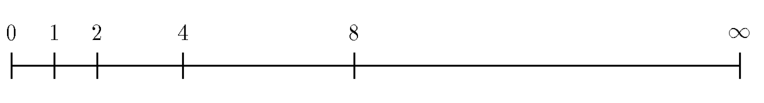


Abbildung 2.10: Ein leerer Radix–Heap

Ausgehend von der Startsituation in Abbildung 2.11 werden nun die Zugriffsprozeduren auf den Radix–Heap demonstriert.

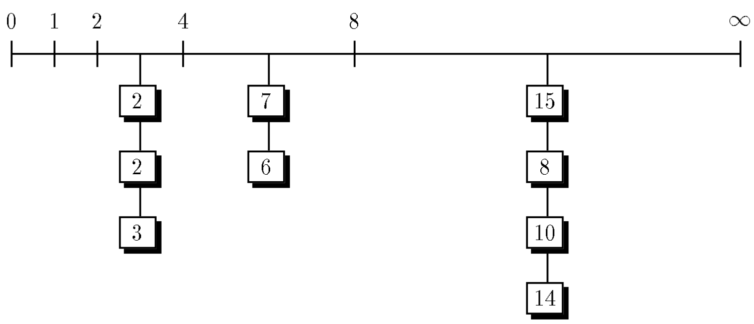


Abbildung 2.11: Ausgangszustand des Beispiel–Heaps

Einfügen eines neuen Elements Um ein neues Element einzufügen, wird linear nach dem richtigen Bucket gesucht, wobei die Suche beim größten Bucket beginnt.

Insert(x)
i = B;
while (u[i] > key(x)) $i--$;
Füge x in b[i] ein;

Offensichtlich bleiben dabei alle Invarianten erfüllt. Wegen Invariante 3 ist die Terminierung der **while**–Schleife gesichert. Abbildung 2.12 zeigt das Vorgehen beim Einfügen graphisch. Die gestrichelten Linien symbolisieren den Suchverlauf beim Einfügen des Elements mit dem Schlüssel „4“. Es ist unerheblich, ob die Elemente am Anfang oder am Ende in die Buckets eingefügt werden, da durch entsprechende Wahl der Datenstruktur (doppelt-verkettete Liste ggf. mit Zeiger auf das Listenende) diese Operation immer in konstanter Zeit durchgeführt werden kann. In einer praktischen Implementierung ist es wahrscheinlich vorzuziehen, die Elemente am Listenanfang einzufügen. In Abbildung 2.12 wird das Element nur aus ästhetischen Gründen am Ende eingefügt.

Erniedrigen eines Schlüssels Der Schlüssel eines Elements kann durch die Prozedur `decreaseKey` erniedrigt werden. Hierbei wird analog zur Prozedur `insert` linear nach dem Bucket gesucht, in den das Element nun mit dem veränderten Schlüssel eingeordnet werden muß. Der einzige Unterschied besteht darin, daß die Suche in diesem Fall beim aktuellen Bucket des Elements beginnen kann, da der Schlüssel erniedrigt wird und das Element deshalb nie in einen größeren Bucket wandern kann. Aus diesem Grund war es notwendig, zu jedem Element x die zugehörige Bucketnummer in `b_no[x]` zu speichern.

decreaseKey(x)
i = b_no[x];
Entferne x aus b[i];
while (u[i] > key(x)) i−−;
Füge x in b[i] ein;

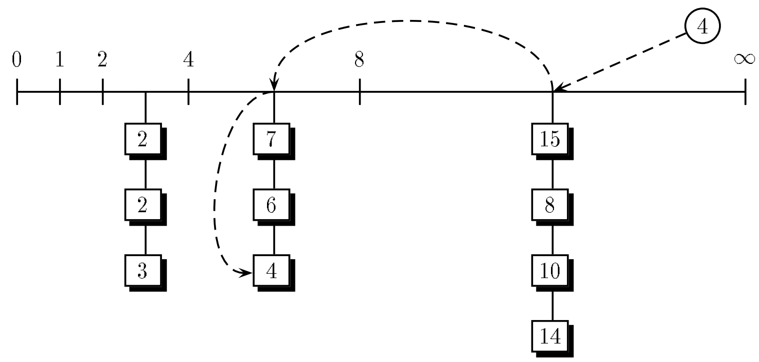


Abbildung 2.12: Einfügen eines Elements mit Schlüssel „4“

Abbildung 2.13 zeigt, wie der Schlüssel des zuvor eingefügten Elements[5] „4“ auf „0“ erniedrigt wird. Wieder symbolisieren die gestrichelten Linien den Verlauf des Suchvorgangs.

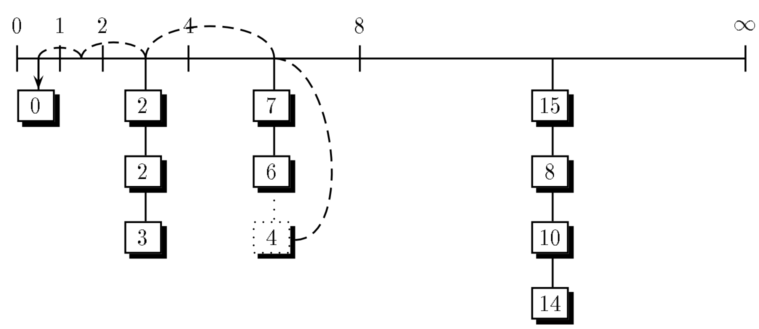


Abbildung 2.13: Erniedrigen des Elements „4“ auf „0“

Wenn man das Vorgehen bei `insert` und `decreaseKey` betrachtet, kann man sich intuitiv klarmachen, daß die gesamte Laufzeit von l `decreaseKey`–Operationen, angewandt auf ein bestimmtes Element x, $\mathcal{O}(l + B)$ betragen wird. Dies ergibt sich dadurch, daß für dieses Element bei der linearen Suche jeder Bucket höchstens einmal durchlaufen wird. Eine genaue Laufzeitanalyse wird später zeigen, daß die amortisierte Laufzeit von `decreaseKey` $\mathcal{O}(1)$ beträgt. Dies kann man aber bereits durch die obige einfache Argumentation intuitiv einsehen.

Diese Argumente können allerdings nicht mehr aufrechterhalten werden, wenn man statt der linearen Suche binäre Suche einsetzt, obwohl man sich davon vielleicht intuitiv eine Verbesserung der Effizienz versprechen könnte. Man kann sich jedoch überlegen, daß man für die binäre Suche einen deutlich schlimmeren worst–case konstruieren kann als für die

[5] Im folgenden wird das Element mit dem Schlüssel x kurz als Element x bezeichnet.

lineare Suche. Man stelle sich hierzu vor, daß die Elemente immer im größten Bucket eingefügt werden und dann bei jedem Aufruf von `decreaseKey` nur einen Bucket nach unten wandern. In diesem Fall werden bei der binären Suche viele Buckets mehrfach untersucht. Bei einer praktischen Implementierung sollte man also nur dann binären Suche einsetzen, wenn man zusätzliche Kenntnisse über die Verteilung der Eingabedaten verfügt und solche Fälle ausschließen kann.

Auslesen des kleinsten Elements Von den üblichen Zugriffsprozeduren für Priority-Queues fehlt nun nur noch `extractMin`. Hierbei wird zuerst nach dem kleinsten nicht-leeren Bucket gesucht und der kleinste dort enthaltene Schlüssel k festgestellt. Wenn der kleinste Bucket bereits ein Element enthält, so kann dieses ausgegeben werden und es ist sonst nichts weiter zu tun. Abbildung 2.14 zeigt diesen Fall.

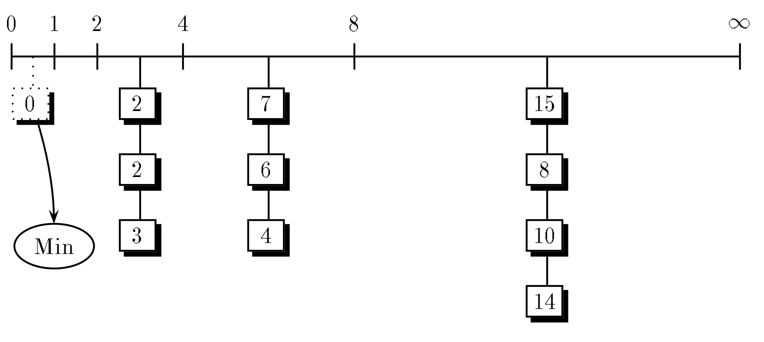


Abbildung 2.14: `extractMin`: einfacher Fall

Andernfalls wird `u[0]` auf k gesetzt und die Bucketgrenzen gemäß der Invarianten neu gesetzt. Dies wird in den Abbildungen 2.15 und 2.16 gezeigt. Danach wird ein minimales Element aus dem kleinsten Bucket, der nun nicht mehr leer sein kann, ausgelesen und entfernt. Von diesen „Aufräumarbeiten“ sind nur der kleinste nicht-leere Bucket und die davorliegenden leeren Buckets betroffen.

<u>extractMin</u>
i = 0;
while (b[i] == empty) i++;
Bestimme kleinsten Schlüssel k in b[i];
// Neuer Heap beginnt bei k → Grenzen u[.] neu setzen
u[0] = k; u[1] = k+1;
for (j=2; j++; j $\leq$ i) {
    u[j] = min{u[j-1]+2^{j-2}; u[i+1]}
}
Verteile alle Elemente von b[i] auf b[0],...,b[i-1];
Entferne ein beliebiges Element von b[0] und gib es aus;

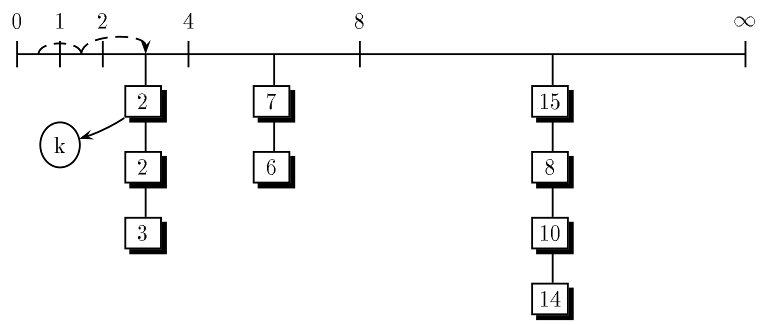


Abbildung 2.15: `extractMin`: Suche nach dem kleinsten Schlüssel

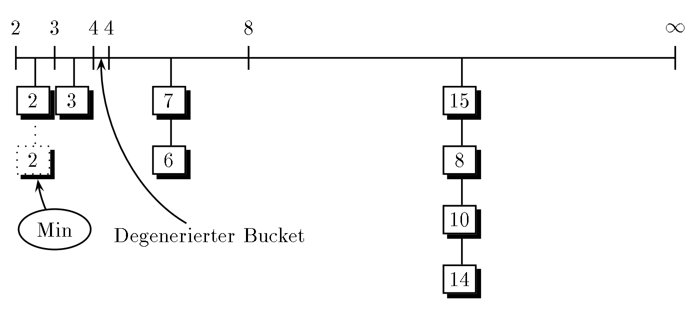


Abbildung 2.16: `extractMin`: Reorganisation des Heaps

Wie im Beispiel von Abbildung 2.16 zu sehen ist, können durch die Reorganisation des Heaps durchaus „degenerierte" Buckets entstehen, bei denen gilt „`u[i] = u[i+1]`" und die somit nach Invariante 1 keine Elemente enthalten können. Dies wird durch die Invariante 2 allerdings nicht verboten.

In Lemma 2.11 wird später gezeigt, daß bei der Reorganisation des Heaps *alle* Elemente aus dem Bucket i auf kleinere Buckets verteilt werden. Dies spielt bei der Laufzeitanalyse eine entscheidende Rolle.

Amortisierte Laufzeitanalyse

Die Zugriffsprozeduren `insert`, `decreaseKey` und `extractMin` benötigen wegen der **while**–Schleife im worst–case bei einem Durchlauf $\mathcal{O}(\log C)$ Zeit, da nicht ausgeschlossen werden kann, daß alle Buckets betrachtet werden müssen. Allerdings wurde bereits aus den Überlegungen im vorigen Abschnitt zum Verhalten von `insert` und `decreaseKey` deutlich, daß der Zeitbedarf mehrerer solcher Operationen nicht einfach aus der Summe der einzelnen worst–case–Laufzeiten errechnet werden kann. Es ist also eine amortisierte Laufzeitanalyse erforderlich.

Hierzu betrachten wir folgende Potentialfunktion:

Potential eines Elementes x	$\phi(x) = \texttt{b\_no[x]}$
Potential des gesamten Heaps H	$\Phi(H) = c \cdot \sum_{x \in H} \phi(x)$

wobei c geeignet gewählt sei.

Nun untersuchen wir die einzelnen Prozeduren:

Laufzeit von `Initialize` Die Laufzeit ergibt sich unmittelbar aus der Anzahl der Schleifendurchläufe der **for**–Schleifen.

tatsächliche Laufzeit	$\mathcal{O}(B)$
Potentialänderung	(Potential danach $\Phi(H) = 0$)
amortisierte Laufzeit	$\mathcal{O}(B)$

Laufzeit von `Insert` Die lineare Suche benötigt maximal linearen Aufwand in der Anzahl B der Buckets und führt zu einer maximalen Potentialerhöhung in derselben Größenordnung.

tatsächliche Laufzeit	$\mathcal{O}(B)$
Potentialänderung	nimmt um $\leq cB$ zu
amortisierte Laufzeit	$\mathcal{O}(B)$

Laufzeit von `decreaseKey` Die Laufzeit ist linear in der Anzahl der Buckets, die bei der Suche nach einem neuen Bucket für das Element durchlaufen werden. Gleichzeitig nimmt aber das Potential um einen Betrag derselben Größenordnung ab, so daß die amortisierte Laufzeit konstant ist.

tatsächliche Laufzeit	$c_1(\phi_{\text{vorher}}(x) - \phi_{\text{nachher}}(x)) + c_2$		
Potentialänderung	nimmt um	$c(\phi_{\text{vorher}}(x) - \phi_{\text{nachher}}(x))$	ab
amortisierte Laufzeit	$\mathcal{O}(1)$, falls $c \geq c_1$		

Laufzeit von `extractMin` Zur Analyse von `extractMin` benötigen wir das folgende Lemma.

Lemma 2.11 *Bei der Reorganisation der Heaps durch* `extractMin` *werden alle Elemente des Buckets i, der das minimale Element k des Heaps enthält, auf kleinere Buckets verteilt (falls $i \geq 1$)*

Beweis: Für $i = 1$ ist die Behauptung offensichtlich erfüllt, da in diesem Fall alle Elemente des Buckets 1 in Bucket 0 verschoben werden. Es gelte also nun $i > 1$.

(Im folgenden bezeichnen die mit „vorher“ bzw. „nachher“ gekennzeichneten Größen den Zustand vor bzw. nach der Reorganisation der Heaps.)

Es genügt zu zeigen:

$$u[i]_{nacher} \geq \min\{k + C + 1, u[i+1]_{vorher}\}. \tag{2.5}$$

Denn: Die Schlüssel aller Elemente im Bucket i sind kleiner gleich $k + C$ (nach Voraussetzung) und auch kleiner als $u[i+1]_{vorher}$ (nach Invariante 1).

Bei der Zuweisung an `u[i]` im letzten Durchlauf der **for**–Schleife sind zwei Fälle zu unterscheiden, je nach Ergebnis der Berechnung des Minimums:

1. Fall: $u[i]_{nachher} = u[i+1]_{vorher}$

Hier ist nichts mehr zu zeigen.

2. Fall: $u[i]_{nachher} = u[i-1]_{nachher} + 2^{i-2}$

Wenn dieser Fall bei der Berechnung des Minimums zutrifft, so muß er auch bei allen vorangegangenen Durchläufen der **while**-Schleife gegolten haben, da immer größere Bucketgrenzen betrachtet werden.

In diesem Fall gilt (für $i \geq 3$; Aussage folgt analog für kleinere i):

$$\begin{aligned}
u[i]_{nachher} &= 2^{i-2} + u[i-1]_{nachher} = 2^{i-2} + 2^{i-3} + u[i-2]_{nachher} \\
&= \ldots = \\
&= \sum_{j=0}^{i-2} 2^j + u[1]_{nachher} = 2^{i-1} - 1 + u[1]_{nachher} = 2^{i-1} + u[0]_{nachher} \\
&= 2^{i-1} + k.
\end{aligned}$$

Für $i < B$ folgt aus den Invarianten 1 und 2:

$$u[i]_{nachher} = 2^{i-1} + k \geq 2^{i-1} + u[i]_{vorher} \geq u[i+1]_{vorher}.$$

Für $i = B$ gilt andererseits nach Definition von B:

$$u[i]_{nachher} = 2^{B-1} + k \geq C + 1 + k.$$

In jedem Fall gilt also die behauptete Ungleichung (2.5). ■

Sei nun i die Nummer des kleinsten nicht-leeren Buckets beim Aufruf von `extractMin`. Dieser Bucket enthalte $\tilde{b}$ Elemente.

Die tatsächliche Laufzeit beträgt $c_3 \cdot i$ für die Suche nach dem ersten nicht-leeren Bucket und $c_4 \cdot \tilde{b}$ für die Bestimmung des minimalen Elements in diesem Bucket.

Bei der Reorganisation des Heaps wird mindestens ein Element in Bucket 0 verschoben, also um eine Distanz von i Buckets. Außerdem wandern die übrigen $\tilde{b} - 1$ Elemente um mindestens einen Bucket nach links (Dies gilt strenggenommen nur, wenn $i > 0$. Für $i = 0$ fällt aber ohnehin nur konstanter Aufwand an, so daß die gewonnen Zeitschranken auch in diesem Fall zutreffen.)

Insgesamt ergibt sich daraus folgende Laufzeit:

tatsächliche Laufzeit	$c_3 i + c_4 \tilde{b} + c_5$
Potentialänderung	nimmt um $\geq c(i + \tilde{b} - 1)$ ab
amortisierte Laufzeit	$\mathcal{O}(1)$, falls $c \geq c_3, c_4$

Mit $B = \mathcal{O}(\log C)$ erhält man unmittelbar folgenden Satz:

Theorem 2.12 *Ausgehend von einem leeren Heap beträgt die* worst–case–*Laufzeit von k* `Insert`- *und l* `decreaseKey`-/`extractMin`–*Operationen in einem Radix–Heap $\mathcal{O}(k \log C + l)$.*

Bemerkung 2.13 *Wenn man die Radix–Heaps zu 2–level–Radix–Heaps erweitert und diese mit für beschränkte Schlüssel modifizierten Fibonacci–Heaps kombiniert, so erhält man als worst–case–Laufzeit $\mathcal{O}(k\sqrt{\log C} + l)$. Dies stellt das momentan beste bekannte Ergebnis für Priority–Queues dar und spielt deshalb in der Theorie eine wichtige Rolle. In der Praxis ist diese Vorgehensweise allerdings uninteressant, da der Verwaltungsaufwand sehr hoch ist und sich deshalb die gute asymptotische Laufzeit erst bei Problemgrößen bemerkbar machen würde, die auf heutigen Rechnern nicht bearbeitet werden können.*

Einsatz von Radix–Heaps in der Praxis

B. V. Cherkassky, A. V. Goldberg und T. Radzik [?] haben in einer Studie über Algorithmen zum Single–Source–Shortest–Path–Problem unter anderem mehrere Varianten des Algorithmus von Dijkstra untersucht. Diese Varianten unterschieden sich nur im Hinblick auf die Datenstruktur für die Priority–Queue. Die dabei erzielten Ergebnisse lassen Rückschlüsse auf den Einsatz von Radix–Heaps in der Praxis zu.

Die Radix–Heaps schnitten dabei im Vergleich zu anderen Varianten mit Buckets recht gut ab. Zwar wurden sie meistens von den 2–level–Buckets knapp geschlagen, aber die Laufzeiten lagen immer in derselben Größenordnung wie die der schnellsten Variante.

Von allen Bucket-Varianten zeigten nur 2–level-Buckets und Radix–Heaps ein gleichmässiges gutes Verhalten bei *allen* untersuchten Problemklassen. Beide Datenstrukturen waren bis auf einzelne Ausnahmen schneller als Implementierungen mit Priority–Queues ohne Buckets (Heaps mit Verzweigungsgrad 3, Fibonacci-Heaps).

Insgesamt erwies sich also, daß es sich durchaus lohnen kann, Radix–Heaps in der Praxis zu verwenden. Im Einzelfall wird man allerdings abwägen müssen, ob 2–level–Buckets vielleicht vorteilhafter sind.

Für die Radix–Heaps spricht die geringere Anzahl von Buckets und der dadurch reduzierte Speicherbedarf. Dabei darf man aber nicht vergessen, daß bei den Radix–Heaps zusätzlich für jedes Heap-Element die zugehörige Bucket–Nummer verwaltet werden muß. Dies könnte jedoch vor allem bei größeren Werten für C in der Hintergrund treten.

2.3 Mengen / Union-Find-Strukturen

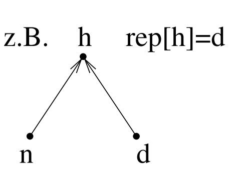
Ausarbeitung: *Gerhard Müller*

Gegeben: Datensätze, partitioniert in (disjunkte) Mengen, wobei jede Menge durch einen in ihr enthaltenen Datensatz repräsentiert werde.

Ziel: Datenstruktur, die die folgenden drei Operationen möglichst effizient realisiert:

MakeNewSet(x)	$\rightarrow$	Füge neuen Datensatz x ein; der bildet eine neue (einelementige) Menge
Find(x)	$\rightarrow$	Gebe Repräsentanten der Menge aus, zu der x gehört
Union(x, y)	$\rightarrow$	Vereinige die beiden Mengen mit Repräsentanten x und y, wobei x Repräsentant der neuen Menge sei.

Realisierung: Jede Menge wird durch einen zur Wurzel hin gerichteten Baum (sog. intree) dargestellt, an der Wurzel notieren wir den Repräsentanten der Menge.



2.3.1 Grundversion

MakeNewSel(x):	mache x zur Wurzel eines neuen (einelementigen) Baumes
Find(x):	Laufe von x zur Wurzel r, gebe Repräsentanten von r aus
Union(x, y):	Hänge Wurzel des kleineren der beiden Bäume ("kleiner" $\hat{=}$ Anzahl der Kinder im Baum) an die Wurzel des größeren an.

Lemma 2.14 *Für jeden in der Union-Find-Struktur enthaltenen Baum T gilt*

$$size(T) \geq 2^{height(T)}$$

wobei $size(T) \hat{=}$ Anzahl der Knoten in T und $height(T)$ die Höhe von T sei.

Beweis: Induktion über die Anzahl k der bereits ausgeführten Operationen

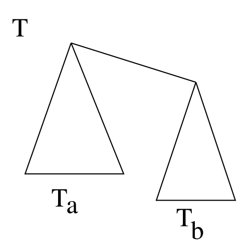
$k = 0$: Die Datenstruktur ist leer $\surd$

$k - 1 \to k$: Fallunterscheidung nach Art der k-ten Operation:

a) **Find**: Bäume bleiben unverändert $\surd$

b) **MakeNewSet**: Behauptung gilt für neuen Baum, die übrigen bleiben unverändert $\surd$

c) **Union**: Seien T_a und T_b die beiden Bäume, die vereinigt werden sollen, wobei o.E. T_b an T_a angehängt werde



Für den neuen Baum T gilt:

$$\text{height}(T) = \max\{\text{height}(T_a), \text{height}(T_b) + 1\}.$$

$$\begin{aligned}
\text{size}(T) &= \text{size}(T_a) + \text{size}(T_b) \\
&\geq \begin{cases} \text{size}(T_a) \overset{\text{I.A.}}{\geq} 2^{\text{height}(T_a)} = 2^{\text{height}(T)}, \\ \qquad\qquad \text{falls height}(T_b) < \text{height}(T_a) \\ 2\text{size}(T_b) \overset{\text{I.A.}}{\geq} 2 \cdot 2^{\text{height}(T_b)} = 2^{\text{height}(T)}, \\ \qquad\qquad \text{falls height}(T_b) \geq \text{height}(T_a). \end{cases}
\end{aligned}$$

Aus Lemma 2.14 folgt unmittelbar:

Theorem 2.15 ***MakeNewSet*** *und* ***Union*** *haben worst case Laufzeit $\mathcal{O}(1)$, die worst case Laufzeit von* ***Find*** *ist $\mathcal{O}(\log n)$, wobei n die maximale Anzahl Elemente pro Menge sei.* ∎

2.3.2 Path Compression

Idee: Ändere **Find**(x) so ab, daß alle Knoten auf dem Pfad von x zur Wurzel r *direkt* unter die Wurzel gehängt werden. (Dadurch erhöht sich die Laufzeit von **Find** nur um einen konstanten Faktor!)

Theorem 2.16 *Ausgehend von einer leeren Datenstruktur benötigen k beliebige Operationen, darunter genau n **MakeNewSet**, höchstens $\mathcal{O}(k \log^* n)$ Schritte.*

Zur Erinnerung:

$$\log^* n = \min\{i > 1 | \underbrace{\log\log\ldots\log}_{i \text{ mal}} n \leq 1\}$$

(Für alle "normalen" Probleme ist $\log^* n$ immer ≤ 6, log immer zur Basis 2, falls nichts anderes angegeben)

Beweis: Für den Beweis des Satzes führen wir eine amortisierte Laufzeitanalyse durch. Dazu definieren wir uns zunächst eine geeignete Potentialfunktion:

- Definiere $a_0, a_1, a_2, \ldots$ durch $\quad a_0 = 0, \quad a_{i+1} = 2^{a_i}$.
  (Die Funktion $n \mapsto a_n$ ist (cum grano salis) die Umkehrung von $\log^*$.)
- Ordne jedem Element x drei Werte zu:

  rank(x) = Höhe des Unterbaums mit Wurzel x — in der Datenstruktur *ohne* Path Compression;

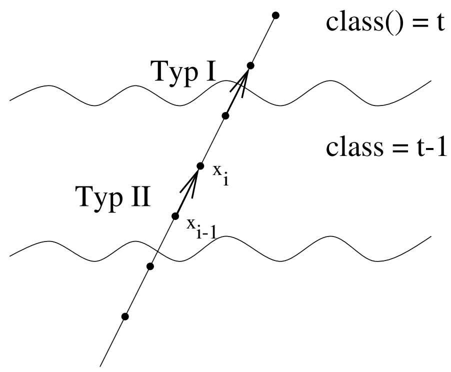
  class$(x) = i \iff a_i \leq \text{rank}(x) < a_{i+1}$
  (Anders ausgedrückt: class$(x) = \log^*(\text{rank}(x))$. (cum grano salis))

  dist(x) = Länge des kürzesten Pfades von x zu einem Knoten y mit class$(y) >$ class(x) bzw. zur Wurzel — in der Datenstruktur *mit* Path Compression;
- Potential: $\Phi = c \cdot \sum_x \text{dist}(x)$.

Hierbei meint „in der Datenstruktur ohne Path Compression", daß man sich nebenher mitlaufend noch eine Union-Find-Struktur, bei der bei **Find**s nichts getan wird, vorstellt und den Rang dort mißt. Das heißt insbesondere, daß sich rank und class eines Knotens durch **Find** nicht ändern.

Wieder überlegen wir uns, daß die Operation **MakeNewSet** konstante Zeit braucht und das Potential nicht ändert.

Betrachten wir eine **Find**-Operation angewendet auf einen Knoten x_0. Seien $x_0, x_1, \ldots, x_r$ die Knoten auf dem Pfad von x_0 zur Wurzel x_r. Auf diesem Pfad unterscheiden wir Kanten danach, ob ihre Endknoten gleiche oder unterschiedliche class-Werte haben, d.h., ob die Kante eine Klassengrenze überquert oder nicht:



(Wegen der Monotonie des Ranges entlang eines solchen Pfades gibt es höchstens $\log^*(n)$ Klassengrenzen überquerende Kanten.) Sei nun x_i ein Knoten mit $\text{class}(x_i) = \text{class}(x_{i+1})$ und $i+1 < r$. Vor Ausführung der path compression ist $\text{dist}(x_i) = \text{dist}(x_{i+1}) + 1 \geq 2$, nachher ist $\text{dist}(x_i) = 1$, also verringert sich der dist-Wert aller auf dem Pfad liegender Knoten, außer $\log^*(n) + 1$ vieler, um mindestens c. Da nur für die auf dem Pfad liegenden Knoten Arbeit anfällt, sind die amortisierten Kosten bei geeignet gewähltem c nur $\mathcal{O}(\log^*(n))$.

Ein **Union** hat zwar konstante tatsächliche Laufzeit, erhöht aber das Potential. Diesmal schätzen wir nicht die Potentialerhöhung einer einzelnen **Union**-Operation ab, sondern die, die von allen vorkommenden **Union**s zusammen bewirkt wird. Der dist-Wert eines festen Knotens x erhöht sich bei einem **Union** genau dann, wenn der Baum, in dem sich x befindet, an die Wurzel eines anderen Baumes, die vorher denselben class-Wert wie x hatte (der class-Wert dieser Wurzel kann sich durch das Anhängen erhöhen), angehängt wird. Da die rank-Werte zur Wurzel hin streng monoton steigen, kann dies nicht öfter als $a_{\text{class}(x)+1}$ mal passieren. Wir summieren nun diese maximal mögliche Potentialerhöhung aller Knoten auf, wobei wir außen nach gleichen class-Werten und innen nach gleichen rank-Werten gruppieren und dabei ausnützen, daß es höchstens $n/2^j$ Knoten x mit $\text{rank}(x) = j$ gibt, da ja unter jedem Knoten mit rank j ein Baum mit mindestens 2^j Knoten hängen muß:

$$\sum_{i=0}^{\log^*(n)} \sum_{j=a_i}^{a_{i+1}-1} \frac{n}{2^j} a_{i+1} = n \cdot \sum_{i=0}^{\log^*(n)} a_{i+1} \sum_{j=a_i}^{a_{i+1}-1} \frac{1}{2^j} \leq n \cdot \sum_{i=0}^{\log^*(n)} \frac{a_{i+1}}{2^{a_i - 1}} \leq 2n(\log^*(n) + 1)$$

Diese Potentialerhöhung ist im Rahmen der Kosten, die wir im Satz 2.16 zeigen wollen, da ja $n \leq k$ ist. ■

Kapitel 3

Grundlegende algorithmische Prinzipien

Ausarbeitung: *Christian Kettner*

3.1 Greedy Algorithmen

Die Klasse der Greedy-Algorithmen ist gekennzeichnet durch das 'Prinzip des Gierigen', das bedeutet, man wählt jeden Schritt so, daß der momentane Gewinn maximal ist.

Betrachten wir ein Beispiel. Zu einem gegebenen zusammenhängenden Graphen $G = (V, E)$ mit Kantengewichtsfunktion $w : E \to \mathbb{R}$ ist ein minimaler Spannbaum $T = (V_T, E_T)$ gesucht, also ein Baum T mit $V_T = V$, $E_T \subseteq E$ und $w(T) := \sum_{e \in E_T} w(e)$ ist minimal.

Der nach Kruskal benannte Greedy-Algorithmus zu diesem Problem geht wie folgt vor :

1. Gehe zunächst von der leeren Menge aus.
2. Nimm in jedem weiteren Schritt diejenige kürzeste Kante des Graphen zur Menge hinzu, die mit den bisher gewählten Kanten keinen Kreis bildet.

Implementierung dieses Algorithmus mit Hilfe von Union-Find-Strukturen :

> Sortiere die Kanten in $E = \{e_1, e_2, \ldots, e_m\}$ so, daß $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$;
> $E_T := \emptyset$;
> Initialisiere eine Union-Find-Struktur;
> **for all** $v \in V$ MakeNewSet(v);
> **for** ($i = 1$; $i \leq m$; $i++$) {
>   sei $e_i = \{x, y\}$;
>   **if** (Find(x) $\neq$ Find(y)) **then** {
> $E_T = E_T \cup e_i$;
>   Union(Find(x), Find(y));
>   }
> }
> Gib E_T aus;

Theorem 3.1 *Sei $G = (V, E)$ ein zusammenhängender Graph mit n Knoten und m Kanten und sei $w : E \to \mathbb{R}$ eine beliebige Gewichtsfunktion, dann bestimmt der Algorithmus von Kruskal in $O(m \cdot \log n)$ Schritten einen minimalen Spannbaum.*

Beweis:

Laufzeit : Unter Berücksichtigung der schon bekannten Laufzeit des Mergesort-Algorithmus und der in Theorem 2.15 bewiesenen Laufzeiten der Operationen in Union-Find-Strukturen ergeben sich folgende Werte :

Kanten sortieren	$O(m \cdot \log m)$	$=$	$O(m \cdot \log n)$
Initialisierung			$O(n)$
Schleife			$O(m \cdot \log n)$
Ausgabe			$O(n)$
insgesamt			$O(m \cdot \log n)$

Korrektheit : Seien $t_1 \dots t_r$ die vom Algorithmus in dieser Reihenfolge ausgewählten Kanten. Wir zeigen durch Induktion über i : $\forall\, 1 \leq i \leq r\ \exists$ minimaler Spannbaum T_i, der $t_1 \dots t_i$ enthält.

$i = 0$: Offensichtlich.

$i \to i+1$: Falls $t_{i+1} \in T_i$, so setzen wir einfach $T_{i+1} := T_i$.

Nehmen wir also an, daß $t_{i+1} \notin T_i$. Betrachte $T_i \cup \{t_{i+1}\}$: Da T_i den Graphen G bereits vollständig aufspannt, muß die Vereinigung einen Kreis C enthalten mit $t_{i+1} \in C$.

Sei nun $C^* := C \setminus \{t_{i+1}\}$. Man sieht leicht ein, daß $T_i \setminus \{e\} \cup \{t_{i+1}\}$ für jede Kante e in C^* ein Spannbaum ist. Um nun den Baum T_{i+1} zu konstruieren, werden wir die Kante e so wählen, daß zum einen $w(e) = w(t_{i+1})$ gilt (denn dann folgt $w(T_i \setminus \{e\} \cup \{t_{i+1}\}) = w(T_i)$) und zum anderen $e \notin \{t_1, \dots, t_i\}$ gilt (denn dann enthält $T_i \setminus \{e\} \cup \{t_{i+1}\}$ weiterhin alle bereits ausgewählten Kanten t_i).

Da der Algorithmus die Kante t_{i+1} ausgewählt hat, können nicht alle Kanten von C^* zur Menge $\{t_1, \dots, t_i\}$ gehören. Wir wählen eine Kanten $e_0 \in C^* \setminus \{t_1, \dots, t_i\}$ beliebig aus.

Zuerst schließen wir aus, daß e_0 länger ist als t_{i+1}: Angenommen doch. Dann wäre $T_i \setminus \{e_0\} \cup \{t_{i+1}\}$ ein Spannbaum mit einem echt kleineren Gewicht als T_i. – Was nicht geht, da T_i nach Annahme ein *minimaler* Spannbaum ist.

Als nächtens schließen wir aus, daß e_0 kürzer ist als t_{i+1}: Angenommen doch. Dann wäre e_0 vom Algorithmus vor t_{i+1} betrachtet worden – und, da e_0 zusammen mit $t_1, \dots, t_i$ ja keinen Kreis erzeugt, auch ausgewählt worden. Im Widerspruch zu unserer Annahme, daß $e_0 \notin \{t_1, \dots, t_i\}$ ist.

Für die Kante e_0 gilt also $w(e_0) = w(t_{i+1})$ und $T_{i+1} := T_i \setminus \{e_0\} \cup \{t_{i+1}\}$ ist der gesuchte minimale Spannbaum.

Damit ist Theorem 3.1 bewiesen. ∎

3.2 Divide and Conquer

Die Idee, die hinter *Divide and Conquer Verfahren* (auf deutsch „teile und herrsche") steht, ist es, ein Problem in zwei oder mehrere Teilprobleme aufzuteilen, diese rekursiv zu lösen und mit diesen Ergebnissen eine Lösung des Ausgangsproblems zu erzeugen.

Ein erstes Beispiel für solch ein Verfahren haben wir mit dem Algorithmus MERGESORT bereits in Abschnitt 1.1 kennengelernt. Jetzt wollen wir uns noch ein zweites Beispiel betrachten.

Im Folgenden geht es um ein Verfahren zur schnellen Multiplikation zweier $n \times n$-Matrizen in einer Umgebung, in der Multiplikationen sehr teuer sind im Verhältnis zu Additionen (z.B. wenn die Wortlänge der Zahlen größer ist als die Wortlänge des Prozessors). Nach dem Standardverfahren wird

$$\begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{nn} \end{pmatrix} = \begin{pmatrix} c_{11} & \dots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nn} \end{pmatrix}$$

durch n^3 Multiplikationen und $n^2 \cdot (n-1)$ Additionen berechnet :

$$\forall i, j \in \{1, \dots n\} \ : c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Strassen gelang es, die Zahl der Multiplikationen auf Kosten der Zahl der Additionen zu verringern. Er entdeckte ein Verfahren, mit dem die Multiplikation von 2x2-Matrizen durch Speichern von Zwischenwerten mit sieben Multiplikationen und 15 Additionen erfolgt (im Standardverfahren benötigt man acht Multiplikationen und vier Additionen).

Die Vorgehensweise von Strassen zur Berechnung von

$$C = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

ist die folgende. Wir berechnen zunächst

$$\begin{aligned}
m_1 &= (a_{21} + a_{22} - a_{11})(b_{22} - b_{12} + b_{11}) \\
m_2 &= a_{11}b_{11} \\
m_3 &= a_{12}b_{21} \\
m_4 &= (a_{11} - a_{21})(b_{22} - b_{12}) \\
m_5 &= (a_{21} + a_{22})(b_{12} - b_{11}) \\
m_6 &= (a_{12} - a_{21} + a_{11} - a_{22})b_{22} \\
m_7 &= a_{22}(b_{11} + b_{22} - b_{12} - b_{21})
\end{aligned}$$

und setzen dann

$$C = \begin{pmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 - m_7 & m_1 + m_2 + m_4 + m_5 \end{pmatrix}.$$

Man überzeugt sich leicht, daß hiermit zum einen wirklich das richtige Ergebnis berechnet wird und zum anderen auch wirklich nur 7 Multiplikationen verwendet werden. (Die Anzahl der Additionen läßt sich bei geeigneter Speicherung von Zwischenergebnissen von den leicht einzusehenden 18 noch auf 15 reduzieren.)

Dieses Verfahren kann nun rekursiv bei der Multiplikation beliebiger Matrizen verwendet werden. Zunächst betrachten wir den Fall, daß $n = 2^k$ eine Zweierpotenz ist. Wir zerlegen die $n \times n$-Matrizen in jeweils vier $2^{k-1} \times 2^{k-1}$-Matrizen :

$$A \cdot B = C \quad \rightarrow \quad \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Mit dem Verfahren von Strassen benötigt man nun 18 Additionen und sieben Multiplikationen von $2^{k-1} \times 2^{k-1}$-Matrizen. Die Multiplikationen können rekursiv weiter heruntergebrochen werden, bis man schließlich bei der Multiplikation von reellwertigen 2×2-Matrizen angelangt. Sei $f(n)$ die Anzahl benötigter Multplikationen in $\mathbb{R}$ bei der Multiplikation zweier nxn-Matrizen, so gilt in diesem Fall :

$$f(n) = f(2^k) = 7 \cdot f(2^{k-1}) = \ldots = 7^{k-1} \cdot f(2) = 7^k = 7^{\log n} = n^{\log 7} \leq n^{2.81}$$

Wenn n keine Zweierpotenz ist, dann besteht die Möglichkeit, die Matrizen erst zu erweitern, bevor das Verfahren von Strassen angewendet wird :

$$A \quad \rightarrow \quad \underbrace{\left(\begin{array}{ccc|c} 1 & & & \\ & \ddots & & 0 \\ & & 1 & \\ \hline & 0 & & A \end{array}\right)}_{2^{\lceil \log n \rceil}} \qquad \text{analog für } B \text{ und } C$$

Die Anzahl benötigter Multiplikationen ist dann :

$$f(n) = f(2^{\lceil \log n \rceil}) = 7^{\lceil \log n \rceil} \leq 7^{\log n + 1} \leq 7 \cdot n^{2.81} = O(n^{2.81}).$$

Hieraus folgt

Theorem 3.2 *Zur Berechnung des Produkts zweier $n \times n$-Matrizen genügen $O(n^{2.81})$ Multiplikationen.*

Bemerkung 3.3 *Es gibt sogar ein Verfahren mit $\mathcal{O}(n^{2.376})$ Multiplikationen (Coppersmith, Winograd [?]). Dieses ist allerdings – im Gegensatz zum Verfahren von Strassen, das sich im Prinzip noch erfolgreich implementieren läßt – nur von theoretischem Interesse.*

3.3 Dynamische Programmierung

Die Grundidee der dynamischen Programmierung ist, die Lösung eines Problems rekursiv zu definieren und dies zu verwenden um schrittweise eine Tabelle aufzufüllen. Man beginnt dabei mit den Elementen am Rekursionsende und nimmt die schon berechneten Elemente als Grundlage zur Berechnung der weiteren Elemente.

Als ein erstes, einfaches Beispiels betrachten hierfür die Berechnung von Binomialkoeffizienten.

Die Rekursion sieht wie folgt aus:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Man die Binomialkoeffizienten daher mit Hilfes des *Pascalschen Dreieckes*

$$\begin{array}{ccccccccc}
\vdots & & \vdots & & \vdots & & \vdots & & \vdots \\
\binom{5}{1} & & \binom{5}{2} & & \binom{5}{3} & & \binom{5}{4} & & \binom{5}{5} \\
& \binom{4}{1} & & \binom{4}{2} & & \binom{4}{3} & & \binom{4}{4} & \\
& & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3} & & \\
& & & \binom{2}{1} & & \binom{2}{2} & & & \\
& & & & \binom{1}{1} & & & &
\end{array}$$

leicht wie folgt berechnen: Beginnend bei $\binom{1}{1}$ berechnet man die darüberliegenden Felder jeweils als Summe der beiden direkt darunter liegenden Elemente bis man beim gesuchten Wert $\binom{n}{k}$ angelangt. Insgesamt müssen $O(n \cdot k)$ Felder berechnet werden, wozu also $O(n \cdot k)$ Additionen ausreichend sind.

Als zweites Beispiel wollen uns nun noch ausführlicher mit dem sogenannten *Rucksack-problem* (engl. knapsack problem) beschäftigen :

Es sind n Objekte mit Gewichten $w_1, \ldots w_n \in \mathbb{N}$ und Profiten $p_1, \ldots p_n \in \mathbb{N}$ gegeben, außerdem ein Rucksack mit Kapazität $B \in \mathbb{N}$.

Gesucht ist eine optimale Packung des Rucksacks, d.h. eine Teilmenge $I \subseteq \{1, \ldots n\}$ mit $(\sum_{i \in I} w_i) \leq B$ und $(\sum_{i \in I} p_i) = max\{(\sum_{i \in I'} p_i) \mid I' \subseteq \{1, \ldots n\} \ \wedge \ (\sum_{i \in I'} w_i) \leq B\}$.

Um dieses Problem nach dem Schema der dynamischen Programmierung zu lösen, brauchen wir ein Array f, das rekursiv berechnet werden kann und aus dem man die optimale Packung ablesen kann : sei $f[i, t]$ das minimale Gewicht, wenn nur die ersten i Objekte zum Packen des Rucksacks zur Verfügung stehen und der Profit der Packung mindestens t betragen soll, wobei $f[i, t] = \infty$ falls $(\sum_{j=1}^{i} p_j) < t$.

Proposition 3.4

$$f[i,t] = min\{f[i-1,t], w_i + f[i-1,t-p_i]\}$$

Beweis: Es gibt nur zwei Möglichkeiten : das i-te Element gehört zu einer Teilmenge mit minimalem Gewicht und Profit mindestens t (dann ist $f[i,t] = f[i-1,t-p_i] + w_i$) oder nicht (dann ist $f[i,t] = f[i-1,t]$). ∎

Nun formulieren wir den Algorithmus zur rekursiven Berechnung des Arrays f der Größe nx$(\sum_{i=1}^{n} p_i)$:

$p := \sum_{i=1}^{n} p_i$
for $(t = 1;\ t \leq p_1;\ t++\ )\ f[1,t] = w_1;$
for $(t = p_1;\ t \leq p;\ t++\ )\ f[1,t] = \infty;$
for $(i = 2;\ i \leq n;\ i++\ )\ \{$
  for $(t = 1;\ t \leq p;\ t++\ )\ \{$
    if $(t \leq p_i)$ **then** $f[i,t] = \min\{f[i-1,t], w_i\ \}$
    else $f[i,t] = \min\{f[i-1,t], f[i-1,t-p_i] + w_i\ \}$
  $\}$
$\}$
return $\max\{t \mid f[n,t] \leq B\};$

Theorem 3.5 *Der Algorithmus ist korrekt und hat Laufzeit $O(n^2 \cdot p_{max})$, wobei $p_{max} = max\{p_i \mid 1 \leq i \leq n\}$.*

Beweis:

Korrektheit : folgt aus Proposition 3.4

Laufzeit:

Initialisierung	$O(n+p)$	$=$	$O(n + n \cdot p_{max})$
Schleife	$O(n \cdot p)$	$=$	$O(n^2 \cdot p_{max})$
Maximumsuche			$O(p)$
insgesamt			$O(n^2 \cdot p_{max})$

Damit ist Theorem 3.5 bewiesen. ∎

Bemerkung 3.6 *Obiger Algorithmus läßt sich leicht so modifizieren, daß nicht nur das optimale Gewicht, sondern auch die zugehörige Teilmenge der Objekte berechnet wird.*

Kapitel 4

Sortieren und Selektieren

Ausarbeitung: *Armin Amon*

4.1 Sortieren

Das Sortieren von Daten ist eines der grundlegenden algorithmischen Probleme. In diesem Kapitel wollen wir einige der dabei verwendeten Verfahren vorstellen. Wobei wir allerdings einige klassische Verfahren (wie beispielsweise HeapSort) nicht betrachten, da diese ja schon im Grundstudium behandelt wurden.

Was wissen wir über die benötigte Laufzeit bzw. die Anzahl der Schlüsselvergleiche? MergeSort (vgl. Abschnitt 1.1) und HeapSort benötigen jeweils $\mathcal{O}(n \log n)$. – Die gleiche Schranke erhält man übrigens, wenn man die im Kapitel **??** vorgestellten *Priority Queues*, zum Sortieren verwendet: man benötigt n *Inserts* und danach n *ExtractMins*. Allerdings sind die Konstanten, die sich hinter dem Groß Oh verbergen hier wesentlich größer, so daß man dieses Vorgehen höchstens in Ausnahmefällen verwenden sollte.

Zunächst wollen wir die Frage angehen, ob es vielleicht auch einen Algorithmus gibt, der das Sortieren von n Zahlen in weniger als $\mathcal{O}(n \log n)$ Schritten bewältigt. Die Antwort ist hier leider nein. Zumindest, wenn man das Sortieren mit Hilfe von "if"-Abfragen vornehmen will.

Theorem 4.1 *Jeder Algorithmus, der n Elemente mit Hilfe von "if"-Abfragen sortiert, benötigt mit $\Omega(n \log n)$ Vergleiche.*

Beweis: Für n Elemente gibt es $n!$ viele verschiedene Ordnungen. Um zu zeigen, daß <u>jeder</u> Algorithmus mindestens $\Omega(n \log n)$ Vergleiche durchführen muß, verwenden wir die Idee des sogenannten *Gegenspieler-Beweises*.

Jeder Frage "$x < y$?" des Algorithmus beantwortet ein (imaginärer) Gegenspieler so, daß möglichst viele Ordnungen mit allen bisher gegebenen Antworten konsistent bleiben. — Und so lange dies mehr als eine sind, kennt der Algorithmus die genaue Reihenfolge der Daten noch nicht, muß also weiterfragen.

Schauen wir uns dies im Detail an: Zu jedem Zeitpunkt beantwortet der Gegenspieler die Anfrage so, daß von den derzeit noch konsistenten Ordnungen mindestens die Hälfte übrig bleiben. Dann gilt also:

Nach dem 1. Vergleich bleiben $\frac{n!}{2}$ verschiedene Ordnungen übrig.

Nach dem 2. Vergleich bleiben noch mindestens $\frac{n!}{4}$ verschiedene Ordnungen übrig.

$$\vdots$$

Nach dem k.ten Vergleich bleiben noch mindestens $\frac{n!}{2^k}$ verschiedene Ordnungen übrig.

D.h., jeder Algorithmus muß (im schlimmsten Fall) mindestens k_0 Vergleiche durchführen, wobei k_0 das kleinste k ist, so daß $\frac{n!}{2^k} < 2$.

Wir schätzen nun noch die Größenordnung von k_0 ab. Dazu überlegen wir uns zunächst, daß gilt

$$n! = \underbrace{n}_{\geq \frac{n}{2}} \underbrace{n-1}_{\geq \frac{n}{2}} \underbrace{n}_{\geq \frac{n}{2}} \cdots \underbrace{\frac{n}{2}}_{\geq \frac{n}{2}} \underbrace{(\frac{n}{2}-1)\cdots 1}_{\geq 1} \geq (\frac{n}{2})^{\frac{n}{2}}.$$

Damit folgt

$$\frac{n!}{2^k} < 2 \quad \Leftrightarrow \quad k > \log \frac{n!}{2} = \log n! - 1 \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} - 1 = \frac{n}{2}(\log n - 1) - 1.$$

D.h., es müssen mindestens $\frac{n}{2}(\log n - 1) - 1 = \Omega(n \log n)$ Vergleiche vorgenommen werden. [Nebenbemerkung: Verwendet man die *Stirling'sche Formel* $n! \approx \sqrt{2\pi n}\frac{n^n}{e^n}$, so erhält man statt $\frac{1}{2}$ eine leicht bessere (=größere) Konstante.] ■

Bemerkung 4.2 *Wie wir bereits gesehen haben, benötigt* BUCKETSORT *nur* $\mathcal{O}(n)$ *Operationen. – Dies ist kein Widerspruch zu Satz 4.1, da wir bei* BUCKETSORT *zum einen eine Annahme über die zu sortierenden Zahlen machen (die Zahlen müssen aus einem bestimmten Interval sein) und zum anderen dort nicht mit Hilfe von "if"-Abfragen sortieren.*

4.1.1 Radix Sort

In diesem Abschnitt wollen wir uns eine einige Verallgemeinerungen des Bucket Sort anschauen, die sogenannten Radix Sorts. Dazu machen wir die folgende

Annahme: Die zu sortierende Elemente s_i sind d-stellige Zahlen bezüglich der Basis r. Also

$$s_i = \sum_{j=1}^{d} a_{ij} r^{d-j} \qquad \text{mit} \qquad a_{ij} \in \{0, \cdots, r-1\}.$$

Dies ist keine restriktive Annahme: Beispielsweise erfüllen Dezimalzahlen ($r = 10$) aber auch ASCII Strings ($r = 256$) diese Voraussetzung.

Grundsätzlich unterscheidet man bei RadixSorts zwei Strategien:

Forward Radix Sorts: die zu sortierenden Zahlen s_i werden von links nach rechts abgearbeitet (d.h., die höchstwertigste Stelle a_{i1} wird zuerst gelesen) und

Backward Radix Sorts: hier werden entsprechend jeweils die niederwertigsten Stellen a_{id} zuerst gelesen.

Backward Radix Sort

Die Vorgehensweise ist ganz einfach: beginnend mit der letzten Stelle sortieren die Elemente jeweils in Buckets ein (wobei wir die Elemente innerhalb der Buckets jeweils hinten anhängen) und hängen dann die Listen der Buckets wieder aneinander (gemäß aufsteigender Bucketnummern) und wiederholen dieses Vorgehen dann mit der nächst größeren Stelle.

Betrachten wir ein Beispiel:

Beispiel 4.3 *Zu Sortieren seinen die 10 Dezimalzahlen 179, 208, 306, 093, 859, 055, 009, 271, 033, 984:*

(1) Sortiere nach 3. Stelle:

0	1	2	3	4	5	6	7	8	9
	271		093	984	055	306		208	179
			033						859
									009

$\Longrightarrow$ *271, 093, 033, 984, 055, 306, 208, 179, 859, 009*

(2) Sortiere nach 2. Stelle:

0	1	2	3	4	5	6	7	8	9
306			033		055		271	984	093
208					859		179		
009									

$\Longrightarrow$ *306, 208, 009, 033, 055, 859, 271, 179, 984, 093*

(3) Sortiere nach 1. Stelle:

0	1	2	3	4	5	6	7	8	9
009	179	208	306					859	984
033		271							
055									
093									

$\Longrightarrow$ $\boxed{009,\ 033,\ 055,\ 093,\ 179,\ 208,\ 271,\ 306,\ 859,\ 984}$

Formal können wir den Algorithmus wie folgt formulieren:

for $(j = d;\ j \geq 1;\ j--)$
{
Verteile Elemente s_i bezüglich ihrer j-ten Stelle a_{ij} in Buckets (Behalte dabei innerhalb der Buckets die derzeitige Reihenfolge bei).
Sammle die Elemente in aufsteigender Bucketnummer wieder ein.
}

Theorem 4.4 *Backward Radix Sort benötigt für n d-stellige Daten bezüglich Basis r Laufzeit $\mathcal{O}(d(n + r))$.* ■

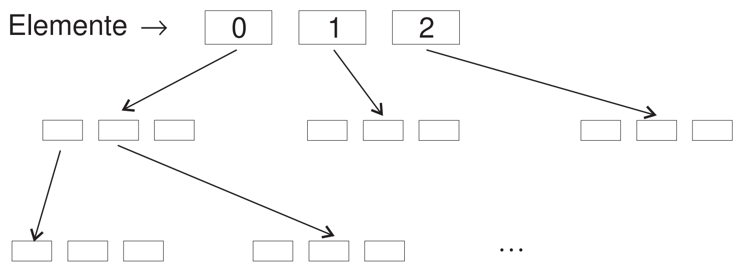
Bemerkung 4.5 *Ein Nachteil des Backward Radix Sort ist, daß er immer alle Stellen der Zahlen liest — auch dann, wenn die bereits die führende Stelle ausreichen würde, um die Zahlen zu sortieren.*

Standard Forward Radix Sort

Als naheliegende Idee ergibt sich daher, die Strings doch lieber von links zu lesen. Andererseits verkompliziert dies den Algorithmus: man darf hier Buckets nicht einfach wieder aneinanderhängen. Davon kann man sich an Hand des folgenden kleinen Beispiels leicht überzeugen: sortiert man zunächst nach der ersten und dann nach der zweiten Stelle, erhält man wieder die Ausgangsreihenfolge und die anschließende Sortierung nach der dritten Stelle führt zu einem falschen Ergebnis:

$$\begin{array}{c}200\\011\\020\end{array} \rightarrow \begin{array}{ccc}\boxed{0}&\boxed{1}&\boxed{2}\\011&&200\\020&&\end{array} \rightarrow \begin{array}{c}011\\020\\200\end{array} \rightarrow \begin{array}{ccc}\boxed{0}&\boxed{1}&\boxed{2}\\200&011&020\end{array} \rightarrow \begin{array}{c}200\\011\\020\end{array} \rightarrow \begin{array}{ccc}\boxed{0}&\boxed{1}&\boxed{2}\\200&011&\\020&&\end{array} \rightarrow \begin{array}{c}200\\020\\011\end{array}$$

Man muß daher die einzelnen Buckets (so sie noch mehr als ein Element enthalten) getrennt weiter sortieren:



Als Abschätzung für die Laufzeit erhält man daher nur eine vergleichsweise miserable Schranke von $\mathcal{O}(\sum_{i=1}^{d} r^i + d_n)$. Allerdings wird der Algorithmus im "Normalfall" wesentlich schneller sein.

Forward Radix Sort with Groups

Abschließend wollen wir nun noch eine neue Variante des Forward Radix Sort vorstellen, dessen Laufzeit (sowohl was die worst case Schranke an geht und auch die empirisch beobachtete Laufzeit in der Praxis) wesentlich besser ist. Dieses Verfahren stammt von Anderson und Nielsen [**?**].

Idee: Ordne jedem Element eine Gruppe zu ($\rightarrow group[x]$), dabei soll immer gelten:

$$group[x] \leq \text{ Rang des Elements in der sortierten Liste.}$$

Wir beginnen mit $group[x] = 1 \forall x$.

Gilt im Laufe des Algorithmus nun für ein $1 \leq i_0 \leq n$

$$|\{x | group[x] = i_0\}| = 1,$$

so brauchen wir das entsprechende x nicht weiter zu betrachten, da wir bereits wissen, daß dieses Element in der sortierten Liste an der i_0-ten Stelle kommt.

Beispiel 4.6 *Zu Sortieren seien wieder die Zahlen 179, 208, 306, 093, 859, 055, 009, 271, 033, 984.*

(1) Sortiere nach 1. Stelle:

0	1	2	3	4	5	6	7	8	9
093;1	179;1	208;1	306;1					859;1	984;1
055;1		271;1							
009;1									
033;1									

$$\Longrightarrow \quad \underbrace{\mathit{093,\ 055,\ 009,\ 033}}_{1}, \underbrace{\mathit{179}}_{5}, \underbrace{\mathit{208,\ 271}}_{6}, \underbrace{\mathit{306}}_{8}, \underbrace{\mathit{859}}_{9}, \underbrace{\mathit{984}}_{10}$$

Die Zahlen unter den Elementen geben die neuen Gruppen der Elemente an. Betrachte beispielsweise das Element 306: da in den Buckets 0,1 und 2 zusammen 7 Elemente waren (und wir die Elemente nach der führenden Stelle einsortiert hatten) wissen wir, das 306 höchstens das 8te größte Element sein kann. Entsprechend bekommt 306 die Gruppe 8 zugewiesen.

Die Gruppe 5, 8, 9 und 10 enthalten jeweils nur genau ein Element. Die darin enthaltenen Elemente müssen wir daher in der nächsten Iteration nicht mehr betrachten. Die übrigen sortieren wir (in dieser Reihenfolge) wiederum in die Buckets ein:

(2) Sortiere nach 2. Stelle:

0	1	2	3	4	5	6	7	8	9
009;1			033;1		055;1		271;6		093;1
208;6									

Nun gilt es wieder die Gruppen neu zuzuordnen. Dazu gehen wir der Reihe nach die Buckets durch (kleinster Bucket zuerst) und sortieren die Elemente in Gruppen ein wobei wir die Gruppe für Elemente in unterschiedlichen Buckets entsprechend erhöhen. Betrachten wir als Beispiel diesmal das Element 206: es gehörte zur 6. Gruppe. Da aber ein anderes Element der 6. Gruppe (208) in einem kleineren Bucket war, kommt 206 diesmal in die 7te Gruppe. Entsprechend werden die vier Elemente der ersten Gruppe auf vier verschiedene Gruppen verteilt, da sie sich jeweils in verscheidenen Buckets befinden. Wir erhalten daher:

Gr.:	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
	009				*179*	*208*		*306*	*859*	*984*
		033					*271*			
			055							
				093						

D.h., jede Gruppe enthält nur noch genau ein Element und wir sind fertig.

Den Algorithmus können wir daher wie folgt formulieren:

for $(i = 1;\ i \leq n;\ i++)$
{
  Füge Element s_j in Gruppe 1 ein;
  $j = 0$
  repeat
  {
    for all *groups* (Gruppen sind hintereinander gehängt)

Falls Anzahl Elemente in Gruppe ≥ 2, sortiere die Elemente der Gruppe gemäß dem j-ten Buchstaben in Buckets (hinten) ein.
for all *buckets*
Sammle Elemente wieder ein: Falls sich das k-te Element der t-ten Gruppe von dem $(k-1)$-ten Element der t-ten Gruppe an der Stelle j unterscheidet (die beiden Elemente also in unterschiedlichen Buckets einsortiert waren), füge es (und alle weiteren der gleichen Gruppe) in eine neue $t+k-1$-te Gruppe ein.
$j = j + 1$
}
until $j = d$
}

Bei der Implementation des Algorithmus sind noch einige Details hinsichtlich der zu verwendenden Datenstrukturen zu beachten, auf diese wollen wir an dieser Stelle aber verzichten.

4.1.2 Quicksort

Ausarbeitung: *Wolfgang Mayerle*

Wie bei vielen anderen Sortierverfahren (Bubblesort, Insertionsort, Mergesort, usw.) ist auch bei Quicksort die Aufgabe, die Elemente eines Array $a[1\,..\,n]$ zu sortieren. Quicksort ist zudem ein *Divide-and-Conquer*-Verfahren:

Divide: Wähle ein Pivot-Element p (z. B. das letzte) und partitioniere $a[l\,..\,r]$ gemäß p in zwei Teile $a[l\,..\,i-1]$ und $a[i+1\,..\,r]$ (durch geeignetes Vertauschen der Elemente), wobei anschließend $a[i] = p$.

Conquer: Sortiere $a[l\,..\,i-1]$ und $a[i+1\,..\,r]$ rekursiv.

Neben den zu sortierenden Elementen stehe in $a[0]$ zusätzlich ein Element mit minimalem Schlüssel (Erklärung s. u.). Hier zuerst einmal der Algorithmus:

quicksort(a, l, r)
{
if $r \geq l$ {
$p := a[r]$; /* Pivot-Wahl */
$i := l - 1$; $j := r$;
repeat
repeat $i := i + 1$ **until** $a[i] \geq p$;
repeat $j := j - 1$ **until** $a[j] \leq p$;
if $j > i$ **then** vertausche $a[i]$ und $a[j]$;
until $j \leq i$;
vertausche $a[i]$ und $a[r]$;
quicksort(a, l, $i - 1$);

quicksort(a, $i+1$, r);
    }
}

Zu Beginn der äußeren repeat-Schleife sind stets zwei Invarianten erfüllt:

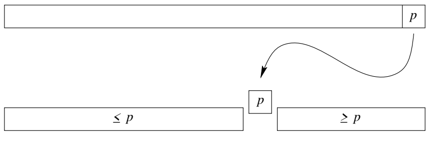
1. $a[1 .. i] \leq p$ und
2. $a[j .. n] \geq p$.

Die beiden Indizes i und j wandern von je einem Ende des Array in die entgegengesetzte Richtung und sind Zeugen, daß "hinter" ihnen zu Beginn nur Elemente kleiner (bei i) oder größer (bei j) als das Pivot sind. Jeder Index hält an der Position, an der für ihn die Bedingung ungültig ist. D. h. insbesondere, daß ein Index nicht hält, falls für ihn nie die Bedingung verletzt ist. Im Falle von i kann das nicht eintreten, da spätestens p die Bedingung verletzt (es ist hierfür *notwendig*, daß i bei Gleichheit von $a[i]$ mit p hält, sonst funktioniert der Algorithmus nicht!). Für j müssen wir künstlich einen Begrenzer einführen: Wir müssen dafür sorgen, daß in $a[0]$ ein minimales Element steht.

Falls beide Indizes halten und sich noch nicht begegnet sind, also $j > i$, werden durch das Vertauschen die Invarianten wiederhergestellt. Es kann auch sein, daß vor dem Vertauschen nur eine oder sogar gar keine Invariante verletzt war: Falls ein Index auf ein Duplikat von p trifft, müsste er strenggenommen gar nicht halten. (Man läßt ihn jedoch halten, weil der ansonsten notwendige Zusatzaufwand, um das Pivot von einem Duplikat zu unterscheiden, den geringen Vorteil von wenigen eventuell eingesparten Schlüsselvergleichen in der Praxis mehr als aufwiegen würde.)

Und falls sich die beiden Indizes dann schließlich einmal begegnen (zusammenfallen oder überschneiden), wissen wir, daß alle Elemente in $a[1 .. i-1]$ kleinergleich und alle Elemente in $a[i+1 .. n]$ größergleich dem Pivot sind. Außerdem wissen wir, daß $i \in \{1, \ldots, n\}$ (für j gilt diese Bedingung i. a. nicht!) und daß $a[i] \geq p$. Wir erreichen also mit einer abschließenden Vertauschung folgende Partitionierung des Array:

Im wesentlichen geschieht in einem Rekursionsschritt eine Partitionierung des Array in Elemente $\leq p$, p und Elemente $\geq p$.



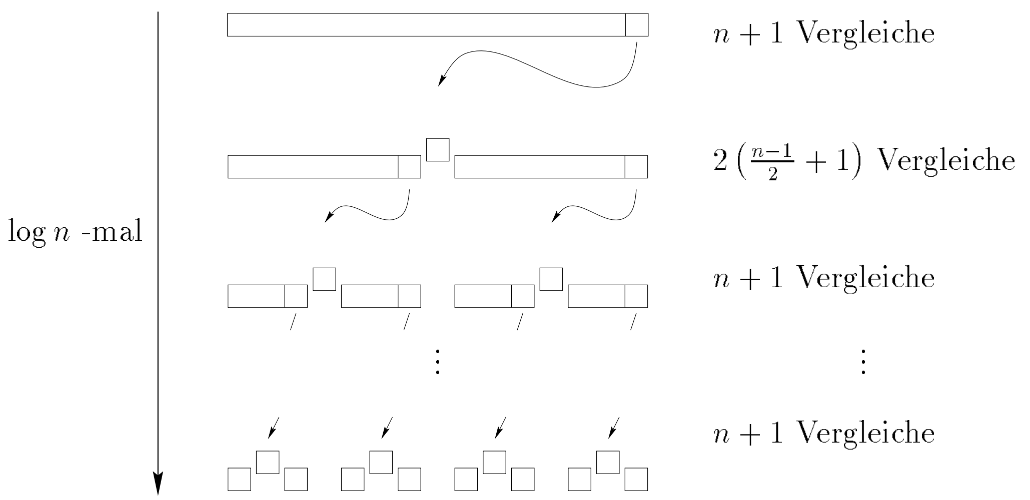
Bemerkung 4.7 *Der oben formulierte Algorithmus benötigt pro Durchlauf in der Regel* $n+1$ *Schlüsselvergleiche. Lediglich wenn die beiden Indizes zusammenfallen (was nur bei einem Pivotduplikat passieren kann), benötigt er* n *Vergleiche.*

Durch geschicktes Einstreuen von if-Abfragen kann man auch in jedem Fall mit $n-1$ *Schlüsselvergleichen auskommen. Man verhindert eine Überlappung von den von* i *und* j *besuchten Elementen, indem man z. B. die äußere repeat-Schleife vorzeitig abbricht, falls unmittelbar nach einer Indexfortschaltung* $i = j$ *ist. Eine Fallunterscheidung je nach Abbruchursache zeigt, daß* $i \in \{1, \ldots, n\}$ *und* $a[i] \geq p$ *immer gilt. Die Vertauschung am Schluß von* $a[i]$ *mit dem Pivotelement liefert dann ebenfalls die gewünschte Partitionierung.*

Wir werden im folgenden (u. a.) beide Möglichkeiten analysieren.

Best-Case

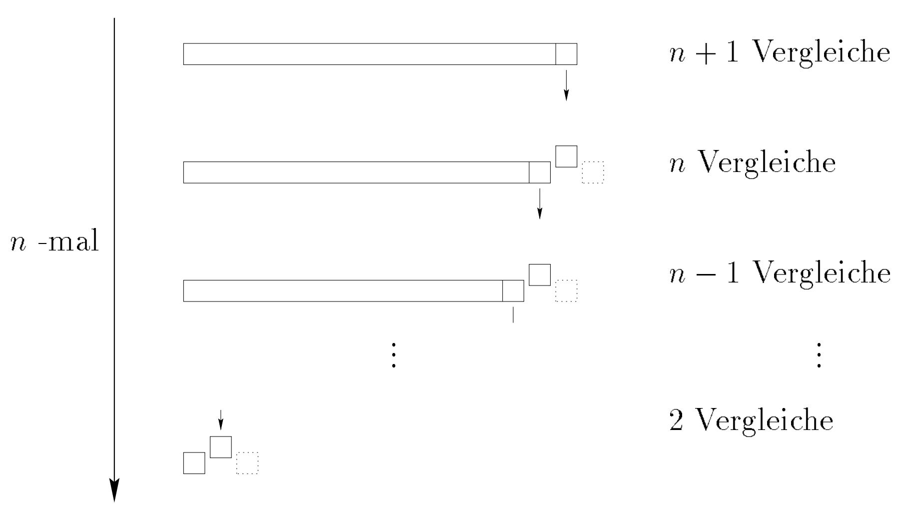
Quicksort läuft natürlich am schnellsten, falls die Partitionierung möglichst ausgewogen gelingt, im Idealfall also immer zwei gleichgroße Teilintervalle entstehen – das Pivot ist dann stets der Median.



Die Anzahl Schlüsselvergleiche insgesamt beläuft sich also auf $\sum_{i=1}^{\log n}(n+1) = \log n(n+1) \approx n \log n$.

Worst-Case

Quicksort kann jedoch unter Umständen auch ein schlechtes Pivot erwischen. Der schlimmste Fall tritt dann ein, wenn die Partitionierung stets ein leeres Teilintervall zur Folge hat, das Pivot also immer das Maximum oder Minimum ist. Ein solches Beispiel liefert ein sortiertes Array.



Die Gesamtanzahl der Schlüsselvergleiche beläuft sich hier auf $\sum_{i=2}^{n+1} i \approx \frac{n^2}{2}$.

Average-Case Analyse

Da die Laufzeit von Quicksort sehr stark von den Eingabedaten abhängt und sich demnach keine scharfe allgemeingültige Schranke angeben läßt, kann man die Frage stellen, wie lange der Algorithmus *im Mittel* zum Sortieren von n Elementen braucht.

Um jedoch überhaupt eine derartige Analyse durchführen zu können, muß man zuerst die genaue Bedeutung von "im Mittel" festlegen. Eine naheliegende Annahme ist, daß alle möglichen Permutationen der Eingabedaten mit gleicher Wahrscheinlichkeit auftreten.

Bemerkung 4.8 *Auch wenn diese Annahme auf den ersten Blick sinnvoll erscheint – in der Praxis wird man sie eher selten antreffen. Ein Grund hierfür ist, daß Daten oftmals schon teilweise sortiert sind. Und genau für diesen Fall scheint Quicksort ja dummerweise eine Schwäche zu haben. Wir werden jedoch später noch zwei Lösungen zu diesem Problem kennenlernen.*

Zur Vereinfachung der Berechnung gehen wir weiter davon aus, daß alle n Schlüssel verschieden sind, d. h. insbesondere, daß es stets $n!$ mögliche Permutationen der Eingabedaten gibt.

Theorem 4.9 *Die durchschnittliche Anzahl von Schlüsselvergleichen C_n, um n verschiedene Schlüssel zu sortieren, beträgt unter der Annahme, daß jede der $n!$ möglichen Reihenfolgen gleich wahrscheinlich sind,*

$$C_n = 2(n+1)(H_{n+1} - 1) = 2n \ln n - 0.846..\, n + o(n) \approx 1.386 n \log n,$$

wobei $H_n = \sum_{k=1}^{n} \frac{1}{k}$ die n-te Harmonische Zahl ist.

Beweis: Der Beweis erfolgt in zwei Schritten: Zunächst wird eine Rekursionsgleichung aufgestellt, welche wir dann im zweiten Schritt lösen werden.

Aufstellen einer Rekursionsgleichung

$$C_n = n + 1 + \frac{1}{n} \sum_{j=1}^{n} (C_{j-1} + C_{n-j})$$

Im Divide-Schritt benötigen wir $n+1$ Schlüsselvergleiche zur Partitionierung des Array. Da das j-größte Element nach Annahme mit Wahrscheinlichkeit $\frac{1}{n}$ an unserer festen Pivot-Stelle steht, ergibt sich für den Conquer-Schritt als durchschnittliche Anzahl von Schlüsselvergleichen der Term um das Summenzeichen in obiger Gleichung.

Wir gehen hier stillschweigend davon aus, daß auch in den Teilintervallen immer noch alle Reihenfolgen gleichwahrscheinlich sind – wir verlassen uns hier auf unsere Intuition. Der interessierte Leser möge sich den formalen Beweis selbst herleiten.

Lösen der Rekursionsgleichung

Unter Berücksichtigung der Symmetrie sowie einer Indexverschiebung erhalten wir

$$C_n = n + 1 + \frac{2}{n} \sum_{j=0}^{n-1} C_j.$$

Wir multiplizieren diese Gleichung mit n

$$nC_n = n^2 + n + 2 \sum_{j=0}^{n-1} C_j,$$

betrachten die identische Gleichung für $n-1$ statt n

$$(n-1)C_{n-1} = (n-1)^2 + n - 1 + 2 \sum_{j=0}^{n-2} C_j$$

und subtrahieren die letzte Gleichung von der vorletzten:

$$\begin{aligned} & nC_n - (n-1)C_{n-1} = 2n - 1 + 1 + 2C_{n-1} \\ \iff \quad & nC_n = 2n + (n+1)C_{n-1}. \end{aligned}$$

Teilt man diese Gleichung nun durch $n(n+1)$, und wendet sie iterativ an, so kommt man zu

$$\begin{aligned} \frac{C_n}{n+1} &= \frac{2}{n+1} + \frac{C_{n-1}}{n} \\ &= \frac{2}{n+1} + \frac{2}{n} + \frac{C_{n-2}}{n-1} \\ &= \cdots = \frac{2}{n+1} + \frac{2}{n} + \cdots + \frac{2}{3} + \frac{C_1}{2} \\ &= 2 \sum_{k=2}^{n+1} \frac{1}{k} \\ &= 2(H_{n+1} - 1), \end{aligned}$$

da $C_1 = 2$ ist.

Daraus folgt zusammen mit der Näherung $H_n = \ln n + 0.57721.. + o(1)$ (ein Beweis hierfür findet sich z.B. in [?]) schließlich

$$\begin{aligned} C_n = 2(n+1)(H_{n+1} - 1) &= 2n \ln n - 0.846..\, n + o(n) \\ &\approx 2 \ln 2 n \log n \approx 1.386 n \log n, \end{aligned}$$

womit der Satz bewiesen ist. ■

Die durchschnittliche Laufzeit ist also erfreulicherweise nicht nur von derselben Größenordnung wie der Best-Case, sondern auch nur um ca. 40% schlechter. Man kann also auch sagen, daß für große n Quicksort im Schnitt um ungefähr 40% mehr Vergleiche benötigt als Mergesort.

Verbesserungsmöglichkeiten

Es gibt eine Reihe von Möglichkeiten (die sogar unabhängig voneinander sind) mit denen man den oben vorgestellten Quicksort verbessern kann:

Einsparen zweier Schlüsselvergleiche pro Durchlauf: siehe Bemerkung 4.7 und Seite 63.

Hybridverfahren: Stellt man dem "schnellen" Quicksort z. B. ein langsames Insertionsort ($C_n = \frac{1}{4}n(n-1)$) gegenüber, so fällt auf, daß bei kleiner Schlüsselzahl Insertionsort schneller ist. Es bietet sich also an, zu Beginn von Quicksort die Intervallgröße zu testen und dann entweder wie gewohnt zu verfahren oder eben Insertionsort aufzurufen. Für dieses Verfahren kommt man dann zu folgender Formel:

$$C_n = \begin{cases} n + 1 + \frac{1}{n}\sum_{j=1}^{n}(C_{j-1} + C_{n-j}) & \text{für } n > m \\ \frac{1}{4}n(n-1) & \text{für } n \leq m \end{cases}$$

Diese Gleichung läßt sich im wesentlichen wie oben auflösen, und man erhält nach einigem Umformen

$$C_n = 2(n+1)H_{n+1} + (n+1)\underbrace{\left(\frac{m^2 + 5m + 12}{6(m+2)} - 2H_{m+2}\right)}_{f(m)}.$$

Bei genauerer Betrachtung kommt man zu dem Schluß, daß C_n möglichst klein wird, wenn $f(m)$ minimal ist, und daß $f(m)$ genau ein Minimum für ein kleines m hat. Eine Wertetabelle liefert dann schließlich $m_{min} = 10$.

Median-Of-3-Quicksort: Eine Möglichkeit, um den peinlichen Worst-Case bei vollständig vorsortierten Daten zu vermeiden, wäre, das Pivot nicht am Intervallrand zu nehmen, sondern in der Intervallmitte. Noch etwas flexibler ist die hier vorgestellte Variante: Wir ersetzen die Pivot-Wahl in obigem Algorithmus durch die folgende:

/* Pivot-Wahl: */
Bestimme $i \in \{l, \frac{l+r}{2}, r\}$ mit $a[i] = median(a[l], a[\frac{l+r}{2}], a[r])$;
Vertausche $a[i]$ mit $a[r]$; $p := a[r]$;

Mit beträchtlichem Aufwand läßt sich selbst für diese Version ein Ausdruck für die durchschnittliche Anzahl von Vergleichen finden (für $n \geq 6$):

$$C_n = \frac{12}{7}(n+1)\left(H_{n+1} - \frac{23}{14}\right) = \frac{12}{7}n \ln n - 1.83..n + o(n) \approx 1.188n \log n$$

Man kann jedoch auch bei dieser Version leicht einen Worst-Case herbeiführen, indem man etwa als Datenfolge einen "Berg" betrachtet, also die Schlüssel vom ersten bis zum mittleren ansteigen und anschließend wieder ebenso schnell abfallen.

Randomized Quicksort: Hier liegt die Idee zugrunde: "Wenn meine Eingangsdaten nicht zufällig permutiert sind, dann würfle ich sie halt selbst durcheinander." Spinnt man den Gedanken noch ein wenig weiter, kommt man zu dem Schluß, daß es sogar genügt, nur das Pivot-Element zufällig zu wählen. Der Sortieralgorithmus sorgt somit selbst für ein gleichmäßiges Auftreten aller möglicher Permutationen, und hat damit *ohne Annahmen über die zu sortierenden Daten* eine durchschnittliche Laufzeit von $\mathcal{O}(n \log n)$. Für eine alternative Analyse siehe folgenden Abschnitt.

Speicherplatzminimierung: Neben allen Optimierungen, die sich mit der Laufzeit von Quicksort beschäftigen, gibt es auch eine sehr elegante, die den zusätzlich benötigten Speicherplatz logarithmisch beschränkt. Man muß lediglich dafür sorgen, daß von den zwei rekursiv zu untersuchenden Teilintervallen *zuerst das kleinere* behandelt wird. Im Beweis geht man von einer Realisierung der Rekursion mittels eines Stack aus, auf dem die Intervallgrenzen gespeichert werden. Ausgehend davon, daß jedes Intervall höchstens so groß ist wie das darunterliegende, kann man zeigen, daß jedes Intervall höchstens *halb* so groß wie das zweitdarunterliegende ist. Damit hat man die Stackhöhe logarithmisch beschränkt.

Randomized Quicksort

Wir ersetzen die Pivot-Wahl des Originalalgorithmus durch die folgende:

/* Pivot-Wahl: */
Vertausche $a[\text{RANDOM}(l, r)]$ mit $a[r]$; $p := a[r]$;

Aus beweistechnischen Gründen (s. u.) ändern wir die repeat-Schleife so, daß wir in der Partitionierungsphase mit $n-1$ Schlüsselvergleichen auskommen.

Theorem 4.10 *Die erwartete Anzahl von Vergleichen E_n, die Randomized Quicksort zum sortieren n verschiedener Zahlen benötigt, beträgt*

$$E_n = 2(n+1)H_n - 4n = 2n \ln n - 2.846..n + o(n) \approx 1.386 n \log n.$$

Der Beweis geht analog zum Beweis von Satz 4.9. Es gibt jedoch auch eine weitere, elegante Methode, welche wir hier betrachten wollen.

Beweis: Sei s_i das i-t-größte der zu sortierenden Elemente. Wir definieren weiter eine Zufallsvariable X_{ij} wie folgt:

$$X_{ij} = \begin{cases} 1 & s_i \text{ wird (irgendwann) mit } s_j \text{ verglichen} \\ 0 & sonst \end{cases}$$

(Hier verwenden wir, daß in jeder Rekursion nur $n-1$ Schlüsselvergleiche stattfinden und daher insbesondere je zwei Schlüssel höchstens einmal miteinander verglichen werden.)

Wir erhalten also für die erwartete Anzahl von Vergleichen

$$E_n = Exp\left(\sum_{i=1}^{n} \sum_{j>i} X_{ij}\right) = \sum_{i=1}^{n} \sum_{j>i} Exp(X_{ij}) = \sum_{i=1}^{n} \sum_{j>i} P_{ij},$$

wobei P_{ij} die Wahrscheinlichkeit für $X_{ij} = 1$ ist.

Wir müssen jetzt nur noch die Frage beantworten, welchen Wert P_{ij} hat, also wann s_i mit s_j verglichen wird.

s_i wird mit s_j verglichen.

$\iff$ s_i oder s_j ist Pivot-Element *und* beide liegen zu diesem Zeitpunkt im gleichen Teilarray.

$\iff$ Keines der Elemente $s_{i+1}, \ldots, s_{j-1}$ wird *vor* s_i oder s_j verglichen (denn das würde zu einer Trennung führen).

$\iff$ Das erste Pivot-Element aus der Menge $\{s_i, \ldots, s_j\}$ ist s_i oder s_j.

Damit bekommen wir $P_{ij} = \frac{2}{j-i+1}$ und damit

$$\begin{aligned}
E_n &= \sum_{i=1}^{n} \sum_{j>i} P_{ij} = \sum_{i=1}^{n} \sum_{j>i} \frac{2}{j-i+1} = 2 \sum_{i=1}^{n} \left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-i+1} \right) \\
&= 2 \sum_{i=1}^{n} (H_{n-i+1} - 1) = 2 \sum_{i=1}^{n} H_{n-i+1} - 2n = 2(\underbrace{H_1 + H_2 + \cdots + H_n}_{\overset{s.u.}{=} H_n(n+1)-n}) - 2n \\
&= 2(H_n(n+1) - n) - 2n = 2(n+1)H_n - 4n \\
&= 2(n+1)(\ln n + 0.57721.. + o(1)) - 4n \\
&= 1.386..n \log n - 2.846..n + o(n)
\end{aligned}$$

mit

$$\begin{aligned}
H_1 + H_2 + \cdots + H_n &= nH_n - \sum_{i=2}^{n} (i-1)\frac{1}{i} = nH_n + \sum_{i=2}^{n} \left(\frac{1}{i} - 1 \right) \\
&= nH_n + H_n - 1 - (n-1) = H_n(n+1) - n.
\end{aligned}$$

(Um den Beweis der letzten Hilfsbehauptung einzusehen, schreibe man die harmonische Summe n-mal untereinander auf, so daß man eine Art Matrix erhält. nH_n ist dann die Summe aller Elemente und $H_1 + H_2 + \cdots + H_n$ entspricht dann z. B. der Dreiecksmatrix links oben samt Diagonale.) ■

Zusammenfassung

Wir haben gesehen, daß Quicksort trotz seiner Einfachheit auch einige Risiken bei der Implementierung birgt und sich nur unter Zuhilfenahme einiger Tricks mathematisch analysieren läßt. Der Lohn der Mühe ist ein Sortierverfahren, das im Schnitt nur um etwa 40% mehr Schlüsselvergleiche benötigt (20% in Kombination mit *Median-Of-3*) als ein optimales Verfahren (z. B. Mergesort), im Gegenzug jedoch nur logarithmisch viel Speicherplatz in Anspruch nimmt (Mergesort benötigt linear viel Zusatzspeicher).

Wir haben ebenfalls beobachtet, daß die Einsparung zweier Vergleiche ($n-1$ statt $n+1$ pro Rekursionsschritt) keine Auswirkung auf den asymptotisch größten Summanden in der Anzahl der Vergleiche hat.

	Schlüsselvergleiche
$n+1$	$1.386n \log n - 0.846n$
$n-1$	$1.386n \log n - 2.846n$

In vielen gängigen Bibliotheken findet sich jedoch meist allenfalls eine *Median-Of-3*-Implementierung, was man anhand geeigneter Testdaten leicht überprüfen kann. Hier ein Beispiel: Die Funktion `qsort()` der GNU C++ Library führt auf einer HP 712/100 für $n = 50.000$ zu folgenden Laufzeiten:

Zufällig erzeugt:	1,08	Sekunden
Aufsteigend vorsortiert:	0,46	Sekunden
Absteigend vorsortiert:	0,44	Sekunden
Auf-/Absteigend vorsortiert:	11,22	Minuten (!)

Der scheinbare Widerspruch der Laufzeiten bei zufälligen Daten im Vergleich zu sortierten (130% langsamer statt den theoretischen 40% oder 20%) liegt vermutlich in der hier nicht vernachlässigbaren Laufzeit der zusätzlich nötigen Vertauschungen verborgen. So sind bei aufsteigend sortierten Daten fast keine Vertauschungen, bei absteigend sortierten um die n, und bei zufälligen Daten ca. ein Viertel der Anzahl Vergleiche notwendig. Setzt man an, daß eine Vertauschoperation ungefähr dreimal so teuer als ein Vergleich ist, so kommt man mit 145% bzw. 110% (für den Median-Of-3-Fall) dem tatsächlich gemessenen Wert schon recht nahe.

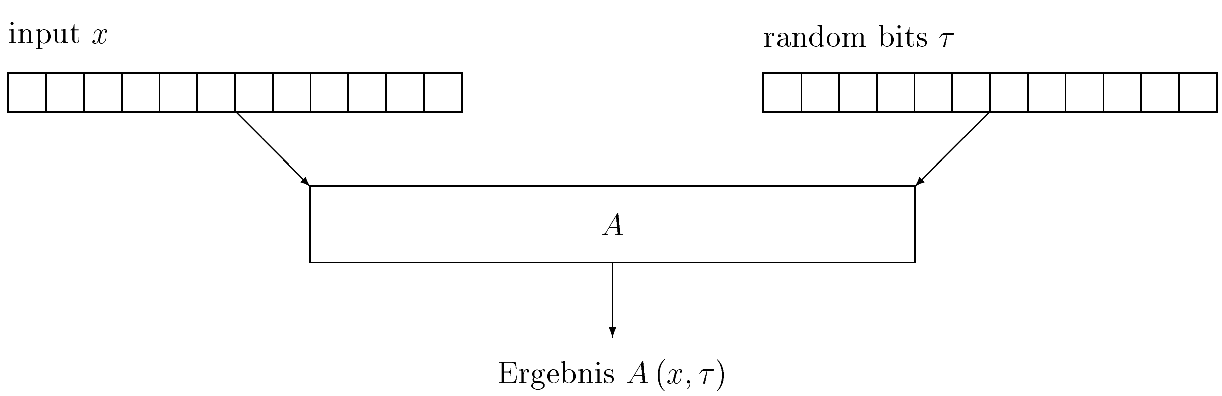
Bei unserem Beispiel schlagen die Vertauschungen also offensichtlich ganz schön zu Buche. Bei komplexeren Datenstrukturen sollte jedoch die Zeit, um zwei Zeiger zu vertauschen neben der Zeit, zwei Zeiger zu dereferenzieren und die Inhalte (byteweise) zu vergleichen, vernachlässigbar klein sein – selbst wenn der byteweise Vergleich nicht vollständig, sondern nur bis zur ersten nicht übereinstimmenden Stelle läuft.

4.1.3 Exkurs: Randomisierte Algorithmen

Ausarbeitung: *Markus Neuhauser*

Idee: Dem Algorithmus wird erlaubt zu „würfeln."

Formale Definition: Algorithmus A erhält zusätzlich zur Eingabe x einen String $\tau \in \{0,1\}^k$ von Zufallsbits, wobei alle Strings gleichwahrscheinlich sind (und k eine hinreichend große Zahl ist, bspw. $k = t(|x|) =$ Laufzeit des Algorithmus bei Eingabe x). Das Ergebnis und/oder Laufzeit hängt nun also von x *und* τ ab!



D. h., man kann nur noch Aussagen der Form *Die Wahrscheinlichkeit, daß der Algorithmus korrekt/schnell/gut ist, ist „hoch“* zeigen. Diese Ergebnisse sind dafür zuweilen wesentlich besser als vergleichbare Aussagen für deterministische Verfahren.

Für eine tatsächliche Implementierung eines solchen randomisierten Algorithmus stellt sich natürlich die Frage, wie man an die verlangten *zufälligen Bits* kommt. Für den Zweck dieser Vorlesung machen wir dafür die folgende

Pragmatische Annahme: Algorithmus hat Zugriff auf Funktion RANDOM(ℓ, r), die eine zufällige (gleichverteilte) Zahl $\in \{\ell, \ell+1, \ldots, r\}$ liefert.

Bemerkung 4.11 *Realisierungen solcher Funktionen liefern nur „fast“ gleichverteilte (sog. pseudorandom) Zahlen. Für praktische Zwecke ist dies aber i. a. ausreichend.*

4.2 Selektieren

Ausarbeitung: *Volker Heun*

Wie wir im Abschnitt über Quicksort gesehen haben, könnten wir die Anzahl der Vergleiche von Quicksort auf $O(n\log(n))$ beschränken, wenn mit jeweils linear vielen Vergleichen den *Median* einer Menge bestimmen könnten.

Wir betrachten dazu das folgende, etwas allgemeinere Problem:

Gegeben: Eine (Multi-)menge $S = \{a_1, \ldots, a_n\}$ und ein $k \in \mathbb{N}$.
Gesucht: Das Element mit Rang k, d.h., das k-kleinste Element der a_i.

Für $k = \lceil \frac{n}{2} \rceil$ entspricht dieses Problem genau der Bestimmung des Medians der Menge S.

Wieviele (Schlüssel-)Vergleiche sind zur Bestimmung des k-kleinsten Elementes nötig? Sicherlich genügen $\mathcal{O}(n\log n)$, da wir in dieser Zeit die Menge ja bereits vollständig sortieren können. Andererseits sieht man auch leicht ein (Wie?!), daß man mindestens $n-1$ Vergleiche durchführen muß. In diesem Abschnitt werden wir zwei Verfahren vorstellen, die in der Tat auch mit $\mathcal{O}(n)$ Vergleichen auskommen.

4.2.1 Der Algorithmus von Blum-Floyd-Pratt-Rivest-Tarjan

In diesem Abschnitt wollen wir einen Algorithmus vorstellen, der in jedem Falle nur linear viele Vergleiche zur Bestimmung des Elementes mit Rang k einer n-elementigen Menge benötigt. Dieser Algorithmus geht auf M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest und R.E. Tarjan aus dem Jahre 1973 zurück. Nach ihren Entwicklern wird er oft auch BFPRT-Algorithmus genannt.

Ohne Beschränkung der Allgemeinheit nehmen wir an, daß die Kardinalität der Menge S mindestens 16 ist. Andernfalls läßt sich ein Element mit Rang k einer so kleinen Menge sehr einfach finden.

Zuerst teilen wir die n Elemente in $\lfloor \frac{n}{5} \rfloor$ Gruppen zu je 5 Elementen. Die übrigen Elemente fassen wir gegebenfalls in einer zusätzlichen Gruppen zusammen. Im folgenden bezeichnen wir der Einfachheit halber alle $\lceil\frac{n}{5}\rceil$ Gruppen immer als 5-er Gruppen, auch wenn so eine Gruppe in Wirklichkeit weniger Elemente besitzt. Nun bestimmen wir von jeder 5-er Gruppe den Median ihrer Elemente. Bei diesen Medianbestimmungen wird gleichzeitig noch mitbestimmt, welche Elemente kleiner oder größer als der entsprechende Median sind. Anschließend bestimmen wir rekursiv aus den $\lceil\frac{n}{5}\rceil$ Medianen deren Median.

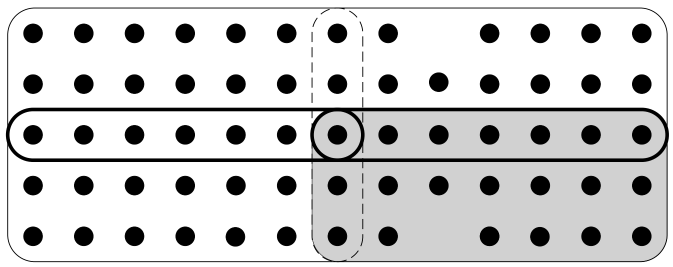


Abbildung 4.1: Die Aufteilung der Mediane vor dem Aufruf der zweiten Rekursion

Eine schematische Darstellung dieses Aufteilung ist in Abbildung 4.1 gegeben. Jede Spalte bildet hier eine der 5-er Gruppen. Dabei sind die Elemente, die kleiner als der Median sind, oberhalb und die Elemente, die größer als der Median sind, unterhalb des Medians der entsprechenden 5-er Gruppe angeordnet. Die Mediane selbst sind von links nach rechts aufsteigend geordnet dargestellt, wobei die einzelnen 5-er Gruppen (Spalten) bei dieser Darstellung mitpermutiert wurden. Innerhalb des Algorithmus kennen wir diese Reihenfolge natürlich nicht: dies dient nur der Veranschaulichung des Algorithmus!

Als nächstes bestimmen wir dann den Rang des Median der Mediane (in der Menge aller Elemente) und vergleichen ihn mit dem Rang k des gesuchten Elementes. Danach können wir ca. ein viertel der Elemente als Kandidaten für das Element mit Rang k verwerfen: Ist der Rang des Medians der Mediane beispielsweise größer als der gesuchte Rang, dann können wir alldiejenigen Elemente verwerfen, die größer als der Median sind. Das sind die drei Elemente in derselben 5-er Gruppe des Medians und jeweils die drei Elemente die größer sind als ein Median, der selbst größer als der Median der Mediane ist. Diese Elemente sind in Abbildung 4.1 grau hinterlegt. In der restlichen Menge suchen wir nun rekursiv weiter. Auch dies ist wieder ein Divide-and-Conquer Algorithmus, wobei wir hier bereits im Divide-Schritt von einem rekursiven Aufruf Gebrauch machen.

Der Algorithmus selbst ist schematisch in Abbildung 4.2 angegeben.

Theorem 4.12 *Der BFPRT-Selektionsalgorithmus bestimmt das k-te kleinste Element einer n-elementigen Menge mit $24n$ Vergleichen.*

Beweis: Die Korrektheit des Algorithmus ist unmittelbar klar.

Für die Analyse der Anzahl durchgeführter Vergleiche bezeichen wir mit S_n die maximal notwendige Anzahl von Vergleichen für eine n-elementige Menge.

Wie schon in der Analyse des QuickSort-Algorithmus werden wir als erstes eine Rekursion für S_n herleiten.

SELECT(S, k)
(1) Teile Elemente aus S in $\lceil \frac{n}{5} \rceil$ Gruppen auf, $\lfloor \frac{n}{5} \rfloor$ davon mit jeweils genau 5 Elementen;
(2) Bestimme den Median jeder Gruppe;
(3) Bestimme rekursiv den Median dieser Mediane, nenne ihn p;
(4) Partitioniere S:
  $S_0 := \{s \in S \mid s < p\}$,
  $S_1 := \{s \in S \mid s = p\}$,
  $S_2 := \{s \in S \mid s > p\}$,
(5) **if** $k \leq |S_0|$ **then**
  SELECT(S_0, k)
 else if $|S_0| < k \leq |S_0| + |S_1|$ **then**
  return p
 else
  SELECT($S_2, k - |S_0| - |S_1|$);

Abbildung 4.2: Der BFPRT-Selektionsalgorithmus

Zunächst müssen wir aus lauter 5-er Gruppen den Median bestimmen. Hierfür genügen 7 Vergleiche pro Gruppe ($\Rightarrow$ Übung). Dann wird der BFPRT-Algorithmus rekursiv zum Bestimmen des Medians der Mediane aufgerufen ($S_{\lceil n/5 \rceil}$ Vergleiche). Dann werden die Elemente nach dem Median der Mediane partitioniert ($n-1$ Vergleiche). Schließlich wird rekursiv weitergesucht. Hierfür müssen wir nun die Größe der Menge abschätzen in der weitergesucht wird.

Ohne Beschränkung der Allgemeinheit nehmen wir an, daß der Rang des Medians der Mediane größer als der gesuchte Rang k war. Hierzu betrachten wir noch einmal Abbildung 4.1. In dieser Abbildung sind in jeder Spalte die Elemente aufsteigend sortiert, und die Median sind von links nach rechts aufsteigend sortiert. Dann wissen wir, daß die grau unterlegten Elemente alle größer als der Median der Mediane ist, und somit ebenfalls einen Rang größer als k haben.

Wie groß ist nun der graue Bereich mindestens? Es gibt mindestens

$$\left\lceil \frac{\lceil \frac{n}{5} \rceil}{2} \right\rceil = \left\lceil \frac{n}{10} \right\rceil$$

Mediane die größer oder gleich als der Median der Mediane sind. Zu jedem Median (bis auf höchstens denjenigen, der nicht zu einer echten 5er Gruppe gehört) wissen wir von noch zwei Elementen, daß sie größer als diese Mediane sind. Damit ist die Anzahl der Element im grauen Bereich mindestens

$$3 \left\lceil \frac{n}{10} \right\rceil - 2.$$

Damit bleiben also im weißen Bereich, den wir jetzt noch zu durchsuchen haben höchstens

$$n - (3 \left\lceil \frac{n}{10} \right\rceil - 2) \leq \left\lfloor \frac{7n}{10} + 2 \right\rfloor$$

Elemente übrig. Damit erhalten wir die folgende Rekursionsgleichung:

$$S_n \;\leq\; S_{\lceil \frac{n}{5} \rceil} + S_{\lfloor \frac{7n}{10} + 2 \rfloor} + 7 \left\lceil \frac{n}{5} \right\rceil + n - 1.$$

Wir zeigen nun mittels vollständiger Induktion, daß $S_n \leq 24(n-3)$ für $n \geq 4$ ist.

Für die Induktionsverankerung betrachten wir zunächst die Fälle $4 \leq n \leq 10$. Verwenden wir hierfür einen (optimalen) Sortieralgorithmus, benötigen wir die folgende Anzahl Vergleiche:

n	4	5	6	7	8	9	10	11	12	13	14	15
# Vergleiche	5	7	10	13	16	19	22	26	30	$\leq$34	$\leq$38	$\leq$42

Man überzeugt sich leicht, daß diese Zahlen jeweils kleiner als $24(n-3)$ sind.

Für den Induktionsschritt sei nun ein $n \geq 16$ gegeben. Als erstes bemerken wir daß für $n \geq 16$ alle rekursiv aufgerufenen Teilprobleme mindestens die Größe 4 haben, also ist der Induktionsanfang vollständig gelegt.

$$\begin{aligned}
S_n &\leq S_{\lceil \frac{n}{5} \rceil} + S_{\frac{7n}{10}+2} + 7\left\lceil \frac{n}{5} \right\rceil + n - 1 \\
&\quad \text{da } \left\lceil \tfrac{n}{5} \right\rceil \geq 4 \text{ und } \left\lfloor \tfrac{7n}{10} + 2 \right\rfloor \geq 13 \text{ folgt} \\
&\overset{\text{I.V.}}{\leq} \left(24\left\lceil \frac{n}{5} \right\rceil - 72\right) + \left(24 \cdot \frac{7n}{10} - 24\right) + 7\left\lceil \frac{n}{5} \right\rceil + n - 1 \\
&\quad \text{da } \left\lceil \tfrac{n}{5} \right\rceil \leq \tfrac{n+4}{5} \\
&\leq 24\left(\frac{n}{5} + \frac{4}{5}\right) + \frac{7 \cdot 24n}{10} + 7\left(\frac{n}{5} + \frac{4}{5}\right) + n - 97 \\
&= 24n - \frac{361}{5} \\
&\leq 24(n-3)
\end{aligned}$$

■

Bemerkung 4.13 *Implementiert man den Algorithmus geschickter, so kann man die Anzahl der Vergleiche noch auf $16 \cdot n$ senken: Man kann auf den Partitionierungsschritt verzichten und stattdessen, wie in der Analyse, den Rang des Medians der Median nur so weit abschätzen, daß nur noch $\frac{7}{10}n+2$ Elemente für den gesuchten Rang in Frage kommen. Dazu muß man nur den Median der Mediane mit den anderen Medianen Vergleichen, da wir die Ergebnisse der Vergleiche der Mediane mit den Elementen in ihren 5-er Gruppen ja bereits kennen. Damit kann man in der Rekursionsgleichung den Summanden $n-1$ durch $\lceil n/5 \rceil - 1$ ersetzten.*

Aus der Komplexität der Selektion ergibt sich für die Komplexität der Medianbestimmung unmittelbar das folgende Korollar:

Korollar 4.14 *Der BFPRT-Algorithmus benötigt zur Bestimmung des Medians einer n-elementigen Menge maximal $24n$ Vergleiche.*

Bemerkung 4.15 *Der bislang asymptotisch beste Medianalgorithmus wurde 1995 von Dor und Zwick [DZ95] entwickelt. Er kommt für die Bestimmung des Medians einer n-elementigen Menge mit $2.95n$ Vergleichen aus.*

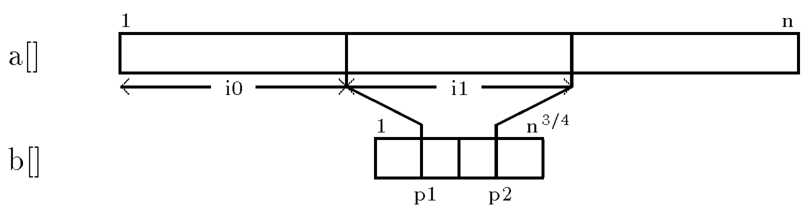
Ebenfalls von Dor und Zwick [DZ96] wurde eine untere Schranke für das Medianproblem gezeigt: Jeder Algorithmus zur Bestimmung des Median einer Menge aus n Elementen benötigt im schlimmsten Fall mindestens $(2+\epsilon)n$ Vergleiche.

4.2.2 Randomisierte Median-Bestimmung

Ausarbeitung: *Peter Trunk*

In diesem Abschnitt stellen wir einen randomisierten Algorithmus vor, der für die Bestimmung des Medians von n Elementen mit sehr hoher Wahrscheinlichkeit lediglich $\frac{3}{2}n$ Vergleiche benötigt.

Idee: Wir ziehen (zunächst) nicht alle n Elemente von $S = a[] = \{a_1, \ldots, a_n\}$ zur Median-Bestimmung heran, sondern wählen eine Teilmenge $b[]$, die $n^{\frac{3}{4}}$ Elemente umfaßt, aus $a[]$ zufällig aus (der Zufall sei gleichverteilt, d.h. jedes Element aus $a[]$ hat die gleiche Chance in $b[]$ aufgenommen zu werden). Anschließend sortieren wir die Elemente in $b[]$ mit einem $n \log n$-Sortierverfahren (z.B. MergeSort), wobei wir $\mathcal{O}(n^{\frac{3}{4}} \log n^{\frac{3}{4}})$ Zeit aufwenden (weil $b[]$ nur $n^{\frac{3}{4}}$ Elemente enthält). Dann wählen wir zwei Pivotelemente p_1 und p_2 aus $b[]$ und „hoffen", daß sich der Median von $a[]$ zwischen p_1 und p_2 befindet. Um dies zu überprüfen, laufen wir in $\mathcal{O}(n)$ durch $a[]$ und zählen dabei mit, wie viele Elemente vor p_1 und wie viele Elemente zwischen p_1 und p_2 liegen. Falls alles „glatt geht", sortieren wir die Elemente zwischen p_1 und p_2 und haben damit den Median. Ansonsten, falls der Zufall also nicht „auf unserer Seite war", fangen wir von vorne an (solange bis es einmal klappt.)



Hier ist der Algorithmus nochmal etwas genauer in Pseudo-Code-Notation:

1. Wähle $n^{\frac{3}{4}}$ Elemente aus $a[]$ zufällig aus:

   for $(i = 1;\ i \le \lceil n^{\frac{3}{4}} \rceil;\ i++)$
   $\quad b[i] = a[\mathrm{RANDOM}(1, n)];$

2. Sortiere die Elemente in $b[]$ (z.B. mit MergeSort)

3. Setze $p_1 = b[\max\{\lfloor \frac{1}{2} n^{\frac{3}{4}} - \sqrt{n} \rfloor, 1\}]$ und $p_2 = b[\min\{\lceil \frac{1}{2} n^{\frac{3}{4}} + \sqrt{n} \rceil, \lceil n^{\frac{3}{4}} \rceil\}]$

   $i_0 = i_1 = 0;$
   for $(j = 1;\ j \le n;\ j++)$
   $\quad$ **if** $p_1 \le a[j] \le p_2$ **then** $i_1++;\ b[i_1] = a[j];$
   $\quad$ **else if** $a[j] < p_1$ **then** $i_0++;$

4. Falls $i_0 \ge \lceil \frac{n}{2} \rceil$ oder $i_0 + i_1 < \lceil \frac{n}{2} \rceil$ oder $i_1 \ge 4n^{\frac{3}{4}}$ gehe zurück zu (1)

   Sonst: Sortiere $b[]$ und gib den Median $b[\lceil \frac{n}{2} \rceil - i_0]$ zurück.

Bemerkung 4.16 *Laufzeit für das Sortieren:* $\mathcal{O}(n^{\frac{3}{4}} \log n^{\frac{3}{4}})$

Theorem 4.17 *Mit einer Wahrscheinlichkeit von* $1 - \mathcal{O}(n^{-\frac{1}{4}})$ *findet der Algorithmus den Median in Laufzeit* $\mathcal{O}(n)$ *mit* $\frac{3}{2}n + o(n)$ *Schlüsselvergleichen.*

Beweis:

Korrektheit: (klar)

Laufzeit (pro Iteration): $\mathcal{O}(n^{\frac{3}{4}}) + \mathcal{O}(n^{\frac{3}{4}} \log n^{\frac{3}{4}}) + \mathcal{O}(n) + \mathcal{O}(n^{\frac{3}{4}} \log n^{\frac{3}{4}}) = \mathcal{O}(n)$.

Schlüsselvergleiche: Pro Iteration kommt man sicherlich mit $\mathcal{O}(n^{\frac{3}{4}} \log n^{\frac{3}{4}})$ (für das Sortieren im 2. und 4. Schritt) plus $\mathcal{O}(2n)$ (für das Auswählen der Elemente im 3. Schritt) aus. Dies ergibt zusammen aber nur eine Schranke von $2n + o(n)$ für die Anzahl benötigter Schlüsselvergleich. Um diese auf $\frac{3}{2}n + o(n)$ zu reduzieren, überlegen wir uns noch, daß im 3. Schritt im Schnitt nur für jedes zweite Element wirklich zwei Verleiche notwendig sind. Entscheidet man daher zufällig, welchen Vergleich man zuerst auswählt, so sieht man, daß $\frac{3}{2}n + \mathcal{O}(n)$ Schlüsselvergleiche reichen.

Nun müssen wir noch zeigen, daß mit einer Wahrscheinlichkeit von $1 - \mathcal{O}(n^{-\frac{1}{4}})$ reicht eine Iteration. Für den Beweis verwenden wir folgende Hilfsmittel aus der Stochastik:

Markov-Ungleichung:	$\text{Prob}[X \geq t] \leq \frac{\text{Exp}[X]}{t} \qquad (X \geq 0)$
Tschebyscheff-Ungleichung:	$\text{Prob}[\lvert X - \text{Exp}[X]\rvert \geq t] \leq \frac{\text{Var}[X]}{t^2}$
$0-1-$ Zufallsverteilung	$\text{Prob}[X_i = 1] = p; \quad \text{Prob}[X_i = 0] = 1 - p;$
Binomialverteilung:	$X = \sum_{i=1}^{n} X_i$, wobei X_i unabhängige und gleichverteilte 0-1 Zufallsvariablen $\text{Prob}[X = k] = \binom{n}{k} p^k (1-p)^{n-k}$ Exp[X] = np; Var[X] = npq; Schreibweise: $X \sim \text{B}(n; p)$

Ohne Einschränkung sei n gerade (dann gilt: $\lceil \frac{n}{2} \rceil = \frac{n}{2}$).

Frage: Mit welcher Wahrscheinlichkeit ist eine der Bedingungen in (4) erfüllt (was zur Folge hat, daß der Algoritmus zurück zu (1) geht und gewissermaßen von vorne anfängt), d.h. mit welcher Wahrscheinlichkeit braucht der Algorithmus mehr als eine Iteration ?

Wann ist die Bedingung $i_0 \geq \frac{n}{2}$ erfüllt ?

Genau dann, wenn bei der Auswahl in (1) höchstens $\max\{\lfloor \frac{1}{2} n^{\frac{3}{4}} - \sqrt{n} \rfloor, 1\} - 1$ Elemente der kleinsten $\frac{n}{2}$ Elemente ausgewählt wurden.

Setze

$$X_i = \begin{cases} 1 & , \quad b[i] \quad \text{gehört zu den kleinsten } \frac{n}{2} \text{ Elementen} \\ 0 & , \quad \text{sonst} \end{cases}$$

Es gilt: $\text{Prob}[X_i = 1] = \frac{1}{2}$

$X = \sum_{i=1}^{n^{\frac{3}{4}}} X_i$ ist die Anzahl der Elemente in $b[]$, die zu den kleinsten $\frac{n}{2}$ Elementen gehören.

X ist binomialverteilt mit den Parametern $n^{\frac{3}{4}}$ und $\frac{1}{2}$:

$X \sim \text{B}(n^{\frac{3}{4}}; \frac{1}{2}) \quad \Longrightarrow \quad \text{Exp}[X] = \frac{1}{2} n^{\frac{3}{4}} \quad \text{und} \quad \text{Var}[X] = \frac{1}{4} n^{\frac{3}{4}}$

Also gilt:

$$\begin{aligned}\mathrm{Prob}[X \leq \frac{1}{2}n^{\frac{3}{4}} - \sqrt{n}] &= \mathrm{Prob}[X \leq \mathrm{Exp}[X] - \sqrt{n}] = \mathrm{Prob}[X - \mathrm{Exp}[X] \leq -\sqrt{n}] \\ &\leq \mathrm{Prob}[|X - \mathrm{Exp}[X]| \geq \sqrt{n}] \leq \frac{\mathrm{Var}[X]}{n} = \frac{1}{4}n^{-\frac{1}{4}}\end{aligned}$$

Analog kann man zeigen, daß für die beiden anderen Bedingungen ($i_0 + i_1 < \frac{n}{2}$ und $i_1 \geq 4n^{\frac{3}{4}}$) Entsprechendes gilt. ■

4.2.3 Untere Schranke für das Median-Problem

Theorem 4.18 *Um den Median von n Elementen zu bestimmen, sind* $\geq \frac{3}{2}n-1$ *Vergleiche nötig.*

Beweis:

Idee: Gegenspieler-Beweis

Algorithmus fragt: $x < y$. Gegenspieler gibt konsistente Antworten, die den Algorithmus jedoch dazu zwingen, möglichst viele Fragen zu stellen.

Strategie des Gegenspielers: Partitioniere die Menge S aller Elemente in drei Teilmengen:

- T große Elemente
- C mittlere Elemente: Kandidaten für den Median
- B kleine Elemente

Es gilt: $\forall x \in T, y \in C, z \in B: \quad x > y > z$

Beobachtung: Solange $|B| < \lceil\frac{n}{2}\rceil - 1, |T| < \lceil\frac{n}{2}\rceil - 1$ und C noch mindestens zwei unverglichene Elemente enthält, ist der Median noch unbestimmt: Der Median muß in diesem Fall zur Menge C gehören, in C stehen aber noch mindestens zwei verschiedene Kandidaten zur Verfügung.

Die Strategie des Gegenspielers ist nun ganz einfach: in einer ersten Phase sorgt er dafür, daß die Menge C möglichst ungeordnet bleibt. D.h., daß jedes Element aus C mit höchstens einem anderen anderen Element aus C verglichen wurde. Dies erreicht er, in dem er gegebenenfalls ein Element aus C nach B oder T "verschiebt". (Dies kann man sich auch so vorstellen, daß der Gegenspieler in diesem Fall dem Algorithmus – ungefragt – mitteilt, daß das Element kleiner bzw. grösser als alle Elemente in $C \cup T$ bzw. $B \cup C$ ist.) In der zweiten Phase zwingt der Gegenspieler den Algorithmus dann, das kleinste (falls $|B| = \lceil\frac{n}{2}\rceil - 1$) bzw. größte (falls $|T| = \lceil\frac{n}{2}\rceil - 1$) der in C verbliebenen Elemente zu finden.

Initialisierung: $B = T = \emptyset, \quad C = S$

Erste Phase: (solange $|B| < \lceil\frac{n}{2}\rceil - 1, |T| < \lceil\frac{n}{2}\rceil - 1$)

1. Fall: $(x \in B \quad \text{und} \quad y \in B) \quad \text{oder} \quad (x \in T \quad \text{und} \quad y \in T)$
Anwort beliebig, aber konsistent.

2. Fall: $(x \in B \quad \text{und} \quad y \in T) \quad \text{oder} \quad (x \in T \quad \text{und} \quad y \in B)$
Anwort beliebig, aber konsistent.

3. Fall: $(x \in B \quad \text{und} \quad y \in C) \quad \text{oder} \quad (x \in C \quad \text{und} \quad y \in B) \quad \text{oder}$
$(x \in C \quad \text{und} \quad y \in T) \quad \text{oder} \quad (x \in T \quad \text{und} \quad y \in C)$
Anwort beliebig, aber konsistent.

4. Fall: $(x \in C \quad \text{und} \quad y \in C)$
Falls weder x noch y schon mit einem anderen Element in C verglichen wurde, so antworte beliebig. Ansonsten verschiebe x bzw. y nach B bzw. T und antworte entsprechend.

Beobachtung: Es gilt immer:
Anzahl der Vergleiche $\geq 2\underbrace{(|B| + |T|)}_{n-|C|}$ + Anzahl der Vergleiche innerhalb von C

Zweite Phase:

1. Fall: C enthält keine unverglichenen Elemente
entweder $|C| = 2$ und Anzahl der Vergleiche innerhalb von $C = 1$
$\Longrightarrow$ Anzahl der Vergleiche $\geq 2(n-2)+1$
oder $|C| = 1 \Longrightarrow$ Anzahl der Vergleiche $\geq 2(n-1)$

2.Fall: B oder T zu groß. O.B.d.A. sei B zu groß, also $|B| = \lceil \frac{n}{2} \rceil - 1$
Idee: Der Gegenspieler zwingt den Algorithmus ab jetzt das kleinste Element in C zu finden. Dazu sind insgesamt (also mit den bereits in der ersten Phase durchgeführten Vergleichen) mindestens $|C| - 1$ Vergleiche nötig. Insgesamt also:

$$\begin{aligned}
\text{Anz. d. Vgl.} \quad & = \quad 2(|B| + |T|) + |C| - 1 \\
& \stackrel{|B|+|T|=n-|C|}{=} \quad |B| + |T| + n - |C| + |C| - 1 \\
& = \quad \underbrace{|B|}_{=\lceil \frac{n}{2} \rceil - 1} + \underbrace{|T|}_{\geq 0} + n - 1 \geq \frac{3}{2}n - 1
\end{aligned}$$

■

Bemerkung 4.19 *Mit einer etwas aufwendigeren Analyse kann man die Schranke aus Satz 4.18 noch verbessern: Bent und John [?] haben gezeigt, daß jeder deterministischen Algorithmus zur Bestimmung des Medians mindestens $2n - o(n)$ Vergleiche benötigt.*

Kapitel 5

Elementare Graphenalgorithmen

Ausarbeitung: *Stefan Pfingstl*

5.1 Grundlagen

Ein Graph $G = (V, E)$ besteht aus einer Menge von Knoten V und einer Menge von Kanten E. Die Anzahl der Knoten bezeichnen wir mit n ($|V| = n$), die Anzahl der Kanten mit m ($|E| = m$).

Ist $E \subseteq \binom{V}{2}$, so ist G ein **ungerichteter Graph**, wobei

$$\binom{M}{k} := \{ X \subseteq M \mid |X| = k \}$$

bedeutet, und es gilt:

$$\left| \binom{M}{k} \right| = \binom{|M|}{k}$$

Ein Graph $D = (V, A)$ heißt **gerichtet**, wenn $A \subseteq V \times V$ ist.

Unter der **Nachbarschaft** $\Gamma(v) := \{u \mid \{u, v\} \in E\}$ eines Knotens v versteht man die Menge der direkten Nachbarn von v. Der **Grad** $\deg(v) := |\Gamma(v)|$ gibt die Anzahl der Nachbarknoten von v an.

Es gilt: $\sum_{v \in V} \deg(v) = 2|E|$.

Das **Komplement** $\overline{G} = (V, \binom{V}{2} \setminus E)$ eines Graphen $G = (V, E)$ besitzt die gleiche Knotenmenge V und hat als Kantenmenge alle Kanten des vollständigen Graphen ohne die Kantenmenge E.

Ein Graph $H = (V_H, E_H)$ heißt (schwacher) **Subgraph** eines Graphen $G = (V_G, E_G)$ (wir schreiben dies auch als $H \subseteq G$), wenn $V_H \subseteq V_G$ und $E_H \subseteq E_G \cap \binom{V_H}{2}$ ist. Gilt $E_H = E_G \cap \binom{V_H}{2}$, so spricht man von einem **induzierten Subgraphen**.

Ein **Kantenzug** ist eine Folge $e_1 := \{v_0, v_1\}, e_2 := \{v_1, v_2\}, \ldots, e_l := \{v_{l-1}, v_l\}$, wobei alle e_i verschieden sind. v_0 und v_l bezeichnen wir als Endknoten. Die Länge des Kantenzuges ist l (Anzahl der Kanten). Sind bei einem Kantenzug auch alle v_i verschieden, so sprechen wir von einem **Pfad**. Ein Kantenzug mit $v_l = v_0$ heißt **Zykel**. Ein Zykel, in dem alle v_i verschieden sind, heißt **Kreis**.

Ein Graph G ist **zusammenhängend**, wenn es für alle $u, v \in V$ einen Pfad gibt, der u und v verbindet. Ein (kanten-) maximaler zusammenhängender Subgraph heißt **(Zusammenhangs-) Komponente**. Ein Graph G ist **azyklisch**, wenn er keine Kreise enthält. Wir bezeichnen diesen dann als **Wald**. Ist dieser auch zusammenhängend, so sprechen wir von einem **Baum**.

Ein **(auf–) spannender Baum** ist ein azyklischer und zusammenhängender Subgraph T mit $V_T = V_G$.

Proposition 5.1 *Ist $G = (V, E)$ ein Wald mit $n = |V|$ Knoten, der aus c Zusammenhangskomponenten besteht, so gilt:*

$$|E| = |V| - c$$

Beweis: (mit Induktion über $m = |E|$)
$m = 0$: $\surd$
Sei $m \geq 1$:
Wähle eine beliebige Kante $e \in E$ und betrachte $G' = G \setminus \{e\}$.
$\Rightarrow c_{G'} = c_G + 1 \Rightarrow |E_G| = |E_{G'}| + 1 = (n - c_{G'}) + 1 = n - c_G$. ∎

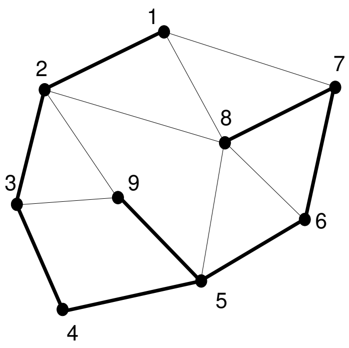
Korollar 5.2 *Jeder Baum mit n Knoten hat genau $n - 1$ Kanten.* ∎

Proposition 5.3 *G zusammenhängend $\Leftrightarrow$ G enthält einen spannenden Baum.*

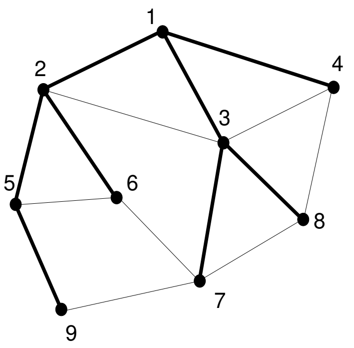
Beweis: „$\Rightarrow$“: Klar. „$\Leftarrow$“: Sie G also ein zusammenhängender Graph. Falls G bereits ein Baum ist, sind wir fertig. Ansonsten enthält G mindestens einen Kreis C. Entferne eine beliebige Kante e aus C. $G' := G - e$ ist dann sicherlich immer noch zusammenhängend. Wiederhole dieses Verfahren solange, bis G' ein Baum ist. ∎

Um die Knoten und Kanten eines Graphen zu durchlaufen, gibt es zwei bekannte Verfahren: Tiefen- und Breitensuche. Diese sollten vom Grundstudium bereits bekannt sein, und wir verzichten an dieser Stelle daher auf die formale Darstellung. Anschaulich kann man sich die Vorgehensweise an Hand der folgenden beiden Bilder nochmals in Erinnerung rufen. Die Nummern an den Knoten geben dabei an, in welcher Reihenfolge die Knoten durchlaufen wurden.

Tiefensuche:



Breitensuche:



Theorem 5.4 *Breiten– und Tiefensuche haben Laufzeit*

$$O(|V| + |E|).$$

■

Korollar 5.5 *Die Zusammenhangskomponenten eines Graphen $G = (V, E)$ lassen sich in Zeit*

$$O(|V| + |E|)$$

bestimmen. ■

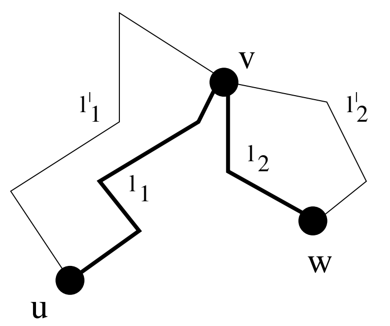
5.2 Das kürzeste–Wege–Problem

Gegeben sei ein Graph $G = (V, E)$, eine Längen- (oder Gewichts-)funktion $l : E \rightarrow \mathbb{R}^+$ und zwei Knoten s und t.

Gesucht ist ein kürzester Pfad, der s und t verbindet.

Lemma 5.6 *Führt ein kürzester Pfad von u nach w über v, so sind die Teilpfade von u nach v und von v nach w auch kürzeste Pfade.*

Beweis: (durch Widerspruch)



Sei l_1 (l_2) die Länge des Pfades von u nach v (v nach w). Angenommen, es gibt einen kürzeren Pfad von u nach v (v nach w) der Länge l_1' (l_2') mit $l_1' < l_1$ ($l_2' < l_2$). Dann ist aber der Pfad von u nach w mit der Länge $l_1' + l_2$ ($l_1 + l_2'$) kürzer als der ursprüngliche Pfad $l_1 + l_2$, d.h. der ursprüngliche Pfad war nicht minimal. ∎

5.2.1 Der Algorithmus von Dijkstra

Eingabe: Graph $G = (V, E); l : E \to \mathbb{R}^+; s, t \in V$
Ausgabe: kürzester Pfad von t nach s

Algorithmus:

/* Initialisierung */
$W := s;$

$$\rho[v] := \begin{cases} 0 & \text{falls} \quad v = s \\ l(s,v) & \text{falls} \quad v \in \Gamma(s) \\ \infty & \text{sonst} \end{cases}$$

$$pred[v] := \begin{cases} s & \text{falls} \quad v \in \Gamma(s) \\ \textbf{nil} & \text{sonst} \end{cases}$$

/* Hauptschleife */
while $t \notin$ W **{**
  Wähle $x_0 \in V \setminus W$, so daß $\rho[x_0] = \min\{\rho[v] \mid v \in V \setminus W\}$;
  $W := W \cup \{x_0\}$;
  for all $v \in \Gamma(x_0) \cap (V \setminus W)$, so daß $\rho[v] > \rho[x_0] + l(x_0, v)$ **{**
    $\rho[v] := \rho[x_0] + l(x_0, v)$;
    $pred[v] := x_0$;
  } /* for */
} /* while */

/* Ausgabe */
return pred[t], pred[pred[t]], ..., s;

Beschreibung des Algorithmus:

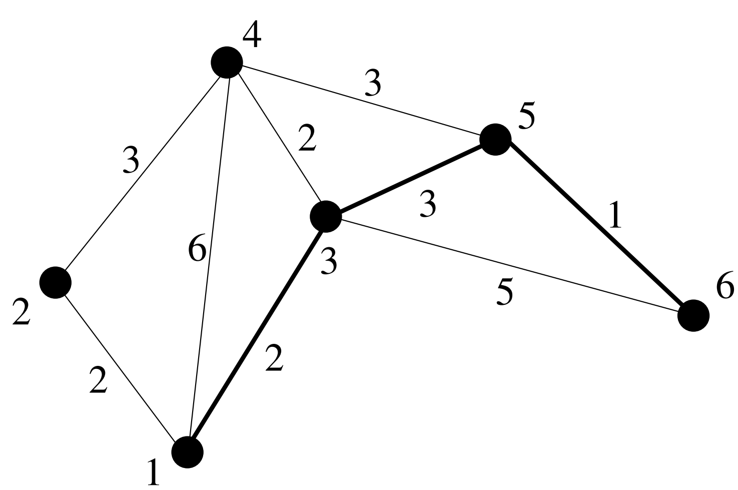
Variablen:

x_0	aktueller Knoten
W	Menge der Knoten, zu denen ein optimaler Pfad bekannt ist.
ρ[v]	Array, um den Abstand zum Startknoten zu speichern.
pred[v]	Array, um den Vorgänger von v (auf dem optimalen Weg) zu speichern.

Der Algorithmus arbeitet in 2 Schritten:

- In der Initalisierungsphase werden W, ρ[v] und pred[v] vorbelegt.
- Hauptschleife (solange der Zielknoten nicht in W enthalten ist)
  - Suche einen Knoten x_0 aus (V \ W), mit minimalen Abstand von Startknoten s. (Die Menge (V \ W), ist die Menge der Knoten, zu denen noch kein optimaler Pfad gefunden wurde.)
  - Füge den Knoten x_0 in W ein.
  - Aktualisiere die Arrays ρ[v] und pred[v].

Ein kleines Beispiel:



Startknoten s=1, Zielknoten t=6

Ablauf:

		v	1	2	3	4	5	6
Initialisierung	$W = \{1\}$							
		$\rho[v]$	0	2	2	6	∞	∞
		pred[v]	nil	1	1	1	nil	nil
1. Iteration	$x_0 = 2$							
	$W = \{1, 2\}$							
		$\rho[v]$	0	2	2	5	∞	∞

		pred[v]	nil	1	1	2	nil	nil
2. Iteration	$x_0 = 3$							
	$W = \{1, 2, 3\}$							
		$\rho[v]$	0	2	2	4	5	7
		pred[v]	nil	1	1	3	3	3
3. Iteration	$x_0 = 4$							
	$W = \{1, 2, 3, 4\}$							
		$\rho[v]$	0	2	2	4	5	7
		pred[v]	nil	1	1	3	3	3
4. Iteration	$x_0 = 5$							
	$W = \{1, 2, 3, 4, 5\}$							
		$\rho[v]$	0	2	2	4	5	6
		pred[v]	nil	1	1	3	3	5
5. Iteration	$x_0 = 6$							
	$W = \{1, 2, 3, 4, 5, 6\}$							
		$\rho[v]$	0	2	2	4	5	6
		pred[v]	nil	1	1	3	3	5

Ausgabe: 5,3,1

5.2.2 Die Algorithmen von Floyd-Warshall und Bellman-Ford

Ausarbeitung: *Franz Indra*

Die Algorithmen von Floyd-Warshall bzw Bellman-Ford benutzen dynamische Programmierung. Im folgenden sei die Menge der Knoten V $= \{1, \ \ldots, \ n\}$.

Der **Algorithmus von Floyd-Warshall** löst das "all pairs shortest path"–Problem.

$F_k[i, j]$ gibt dabei die Länge eines kürzesten i-j-Pfades an, der als **innere Knoten** (das sind die Knoten des Pfades außer i und j) nur Knoten aus $\{1, \ \ldots, \ k\}$ verwendet.

Initialisierung:

$$\forall\, i,\ j\ \in\ \mathrm{V}: \qquad F_0[i,j] := \begin{cases} l(\{i,j\}) & \text{falls } \{i,j\} \in \mathrm{E} \\ 0 & \text{falls } i = j \\ \infty & \text{sonst} \end{cases} . \tag{5.1}$$

Rekursion:
for $(k := 1;\ k \leq n;\ k++)$
$\quad \forall\, i,\ j\ \in\ \mathrm{V} : F_k[i,j] := \min\{F_{k-1}[i,j], F_{k-1}[i,k] + F_{k-1}[k,j]\};$

Ausgabe: $F_n[i,j]$ gibt die Länge eines kürzesten i-j-Pfades an.

Der Algorithmus ist offensichtlich korrekt, da ein kürzester i-j-Pfad mit inneren Knoten aus $\{1, \ldots, k\}$ entweder den Knoten k (genau einmal) enthält – dann ist seine Länge die Summe der Längen der beiden (kürzesten) Teilpfade von i nach k bzw. von k nach j, die nur noch $\{1, \ldots, k-1\}$ als innere Knoten besitzen –, oder den Knoten k nicht enthält – dann ist seine Länge gleich der eines kürzesten i-j-Pfades mit inneren Knoten aus $\{1, \ldots, k-1\}$. Der Algorithmus besitzt die Laufzeit $\mathcal{O}(n^3)$.

Der **Algorithmus von Bellman-Ford** löst das "single source shortest path"–Problem zu gegebenem Startknoten s.

$B_k[i]$ gibt dabei die Länge eines kürzesten s-i-Pfades an, der aus höchstens k Kanten besteht.

Initialisierung:

$$\forall\, i \in \mathrm{V}: \qquad B_1[i] := \begin{cases} l(\{s,i\}) & \text{falls } \{s,i\} \in \mathrm{E} \\ 0 & \text{falls } i = s \\ \infty & \text{sonst} \end{cases} . \tag{5.2}$$

Rekursion:
for $(k := 2;\ k \leq n-1;\ k++)$

$$\forall\, i \in \mathrm{V} : B_k[i] := \begin{cases} 0 & \text{falls } i = s \\ \min_{v \in \Gamma(i)}\{B_{k-1}[v] + l(\{v,i\})\} & \text{sonst} \end{cases} ;$$

Ausgabe: $B_{n-1}[i]$ gibt die Länge eines kürzesten s-i-Pfades an.

Der Algorithmus ist offensichtlich korrekt und besitzt die Laufzeit $\mathcal{O}\Big(n \cdot \sum_{i \in \mathrm{V}} \deg(i)\Big) = \mathcal{O}(nm)$.

5.2.3 Digraphen mit negativen Kantengewichten

Ein **gerichteter Graph** bzw. **Digraph** $\mathrm{D} = (\mathrm{V}, \mathrm{A})$ besteht aus einer Menge V von Knoten und einer Menge $\mathrm{A} \subseteq \mathrm{V} \times \mathrm{V}$ von gerichteten Kanten; $(u,v) \in \mathrm{A}$ bedeutet hier, daß der Graph D eine Kante von u nach v enthält.

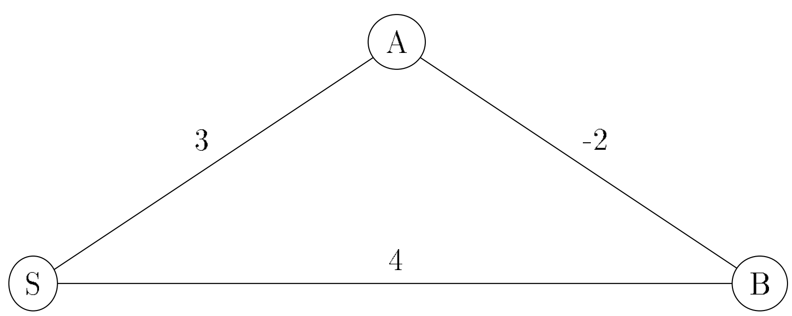
Bemerkung 5.7 *Bei nichtnegativen Kantenfunktionen l auf* A *funktionieren die vorgestellten Algorithmen (Dijkstra, Floyd-Warshall, Bellman-Ford) immer noch.*

Für beliebiges l sind die "shortest path"–Probleme nicht unbedingt sinnvoll. Es kann nämlich ein **negativer Kreis** auftreten, also ein gerichteter Kreis, dessen Kantengewichte eine negative Summe haben. Je öfter ein Kantenzug diesen Kreis durchläuft, desto kleiner wird seine Länge, d.h., es gibt dann keinen mit minimaler Länge.

Ziel ist daher, einen kürzesten gerichteten s-t-Pfad zu finden, der keinen von s aus erreichbaren negativen Kreis enthält, bzw. einen negativen Kreis zu entdecken. Im Spezialfall eines azyklischen Digraphen ergeben sich natürlich keine Schwierigkeiten.

Der **Algorithmus von Dijkstra** ist auf dieses erweiterte Problem nicht anwendbar.

Bei diesem Beispielgraphen (der nicht einmal einen negativen Kreis enthält) berechnet der Algorithmus von Dijkstra die minimale Entfernung von S nach A fälschlicherweise als 3 (statt 2).



Modifikation des Algorithmus' von Floyd-Warshall

$F_k[i,j]$ gibt nun die Länge eines kürzesten gerichteten i-j-Pfades an, der als innere Knoten nur Knoten aus $\{1, \ \ldots\ , \ k\}$ verwendet.

Lemma 5.8 *Enthält* D *einen negativen Kreis, so gibt es mindestens ein* $i \in \mathrm{V}$*, für das* $F_n[i,i] < 0$ *gilt.*

Beweis: Sei k der kleinste Knoten, zu dem es einen negativen Kreis in D gibt, dessen maximaler Knoten k ist. Wähle einen negativen Kreis C in D, der unter allen negativen Kreisen in D mit maximalem Knoten k die kleinste Länge hat.

Beobachtung: $\forall\ t \ < \ k$ gilt: $F_t[i,j]$ wird richtig berechnet. – Da der maximale Knoten jedes negativen Kreises in D nach Definition von k größer als t ist, folgt dies wie bisher.

Sei i der Nachfolger von k in C, d.h., (k,i) ist ein Teilstück von C. Aus der Wahl von C folgt: $F_{k-1}[i,k] = l\Big(\mathrm{C} \setminus (k,i)\Big)$ (wegen $(k,i) \in \mathrm{A}$ erhielte man mit einem i-k-Pfad kürzerer Länge einen Kreis mit maximalem Knoten k, der kürzer als C ist) und $F_{k-1}[k,i] = l\Big((k,i)\Big)$ (ein kürzerer k-i-Pfad ließe sich mit dem i-k-Teilstück von C wiederum zu einem Kreis mit maximalem Knoten k zusammenfügen, der kürzer als C ist). Daraus folgt: $F_k[i,i] \leq F_{k-1}[i,k] + F_{k-1}[k,i] = l(\mathrm{C}) < 0$, und dies ist ohne einen negativen Kreis nicht möglich. ■

Man kann also in den Algorithmus von Floyd-Warshall einen Test auf negative Kreise einbauen, indem man direkt vor der Ausgabe den folgenden Befehl einfügt:

$\forall\ i\ \in$ V: if $F_n[i,i] < 0$ then STOP: Es gibt einen negativen Kreis;

Modifikation des Algorithmus' von Bellman-Ford

$B_k[i]$ gibt nun die Länge eines kürzesten gerichteten s-i-Pfades an, der aus höchstens k Kanten besteht.

Jeder beliebige Kantenzug, der keinen Kreis enthält, besteht aus maximal $n-1$ Kanten.

In einem Graphen ohne negative Kreise gilt daher $\forall\ i\ \in$ V : $B_n[i] \geq B_{n-1}[i]$.

Gibt es dagegen einen (von s aus erreichbaren) negativen Kreis in D, so gibt es einen Knoten $i\ \in$ V, bei dem ein Kantenzug aus n Kanten mit Länge $B_n[i]$ diesen Kreis häufiger durchläuft als jeder Kantenzug aus max. $n-1$ Kanten der Länge $B_{n-1}[i]$. Demnach gilt dann $B_n[i] < B_{n-1}[i]$.

Man kann also in den Algorithmus von Bellman-Ford einen Test auf negative Kreise einbauen, indem man in der Rekursion auch $\forall\ i\ \in$ V : $B_n[i]$ berechnet und direkt vor der Ausgabe den folgenden Befehl einfügt:

$\forall\ i\ \in$ V: if $B_n[i] < B_{n-1}[i]$ then STOP: Es gibt einen negativen Kreis;

Algorithmus von Johnson

Für die Lösung des "all pairs shortest path"–Problem mit einer beliebigen (negativen) Längenfunktion l gibt es noch die Möglichkeit, es auf ein analoges "all pairs shortest path"–Problem mit einer anderen, aber positiven Längenfuntion zu übertragen. Dazu verwendet man eine sog. Rekalibrierung.

Definition 5.9 $f:\ \mathrm{V}\ \mapsto\ \mathbb{R}$ *heißt eine* **Rekalibrierung**, *falls* $\forall\ (u,v)\ \in\ \mathrm{A} : f(u) + l(u,v) \geq f(v)$ *gilt.*

Bemerkung 5.10 *Ist* f *eine Rekalibrierung, so ist* $l_f:\ \mathrm{A}\ \mapsto\ \mathbb{R}$ *mit* $\forall\ (u,v)\ \in\ \mathrm{A}$: $l_f(u,v) := l(u,v) + f(u) - f(v)$ *eine nichtnegative Längenfunktion. Für* l_f *gilt zusätzlich: Ist* $\mathrm{P} = (v_0,\ \ldots,\ v_k)$ *ein Pfad, so ist* $l_f(\mathrm{P}) = \sum_{i=0}^{k-1}\ l_f(v_i, v_{i+1}) = \sum_{i=0}^{k-1}\Big(\ l(v_i,v_{i+1}) + f(v_i) - f(v_{i+1})\ \Big) = \sum_{i=0}^{k-1}\ l(v_i,v_{i+1}) + f(v_0) - f(v_k) = l(\mathrm{P}) + f(v_0) - f(v_k)$, *d.h., die Pfadlängen unterscheiden sich nur durch eine additive Konstante, die nicht vom Pfad, sondern nur von Anfangs- und Endknoten abhängt.*

Also ist P *genau dann ein kürzester s-t-Pfad bzgl.* l, *wenn* P *ein kürzester s-t-Pfad bzgl.* l_f *ist.*

Der **Algorithmus von Johnson** findet zuerst eine Rekalibrierung f und löst dann das "all pair shortest path"–Problem bzgl. l_f (z.B. mit Dijkstra).

Eine Rekalibrierung kann man finden, indem man einen Knoten s hinzufügt und s mit jedem anderen Knoten durch eine Kante der Länge 0 verbindet. $d(v)$ gebe nun die Länge eines kürzesten s-v-Pfades an (da negative Kanten auftreten können, ist d nicht zwangsläufig konstant Null!). d wird mit einem "single source shortest path"–Algorithmus, z.B. Bellman-Ford, bestimmt. d ist eine Rekalibrierung, da $\forall\ (u,v)\ \in\ A : d(u)+l(u,v) \geq d(v)$ nach Definition von d gilt.

Zusammenfassung

In der folgenden Tabelle haben wir die Laufzeiten der vorgestellten Algorithmen noch einmal zusammengefaßt:

	$l \geq 0$	l beliebig
"single source"	Dijkstra: $\mathcal{O}\Big(m + n \cdot log(n)\Big)$	Bellman-Ford: $\mathcal{O}(nm)$
"all pairs"	Dijkstra: $\mathcal{O}\Big(nm + n^2 \cdot log(n)\Big)$ Floyd-Warshall: $\mathcal{O}(n^3)$	Johnson: $\mathcal{O}\Big(nm + n^2 \cdot log(n)\Big)$ Floyd-Warshall: $\mathcal{O}(n^3)$

5.3 Transitive Hülle

Ausarbeitung: *Florian Lindauer*

Sei gegeben ein Digraph $D = (V, A)$. Es gilt also $A \subseteq V \times V$. Die *transitive Hülle* von D ist ein Digraph $D^* = (V, A^*)$, der eine Kante (u,v) genau dann enthält, wenn in D ein gerichteter Pfad von u nach v existiert:

$$(u,v) \in A^* \iff \exists\ r \geq 2 : \exists (w_\rho)_{\rho=1,\ldots,r} : w_1 = u,\ w_r = v,\ (w_{i-1}, w_i) \in A\ \ \forall i \in \{2, \ldots, r\}$$

Es gilt offenbar $A \subseteq A^*$.

Zur Bestimmung der transitiven Hülle eines Digraphen:

1. Mittels Breitensuche: Ausgehend von jedem Knoten u führt man eine Breitensuche durch und fügt für alle erreichten Knoten v die Kante (u,v) in A^* ein.

   Dabei ergibt sich eine Laufzeit $\mathcal{O}(n \cdot (m+n)) = \mathcal{O}(nm + n^2)$. Diese Vorgehensweise eignet sich also nur für Digraphen mit wenigen Kanten; bei dicht besetzten Graphen liegt m in der Größenordnung von n^2.

2. Unter Verwendung von Matrixmultiplikation: Man betrachtet die Adjazenzmatrix $M = (m_{ij});\ \ i,j \in \{1,...,n\}$ wobei $m_{ij} = \begin{cases} 1 & \text{falls } (i,j) \in A \\ 0 & \text{sonst} \end{cases}$

   Es gilt $(M^2)_{ij} = \sum_{k=1}^{n} m_{ik} m_{kj} > 0$ wenn es einen gerichteten i-j-Pfad über ein k gibt, also einen Pfad der Länge zwei. Entsprechend verallgemeinert gilt für die Matrix M^t:

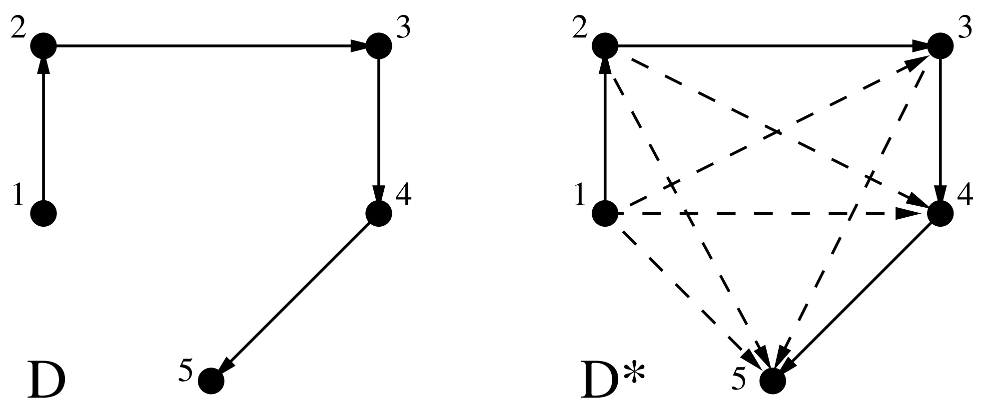


Abbildung 5.1: Beispiel zur transitiven Hülle

Proposition 5.11 $(M^t)_{ij} > 0 \Longleftrightarrow \exists$ *eine gerichtete i-j-Kantenfolge der Länge* t.

Beweis: Durch Induktion: Die Aussage gilt für $t \leq 2$. Gelte sie nun für ein beliebiges t: $(M^{t+1})_{ij} = \sum_{k=1}^{n}(M)_{ik}(M^t)_{kj}$. Also gilt auch für $t+1$: $(M^{t+1})_{ij} > 0$ genau dann, wenn es einen Zwischenknoten k, eine Kante von i nach k sowie einen gerichteten Kantenzug der Länge t von k nach j gibt. Zusammen ergibt sich ein Kantenzug der Länge $t+1$ von i nach j. ■

Korollar 5.12 $\forall i \neq j$ *gilt:*

$$(i,j) \in A^* \Longleftrightarrow (\sum_{t=1}^{n-1} M^t)_{ij} > 0 \Longleftrightarrow (\sum_{t=1}^{2^{\lceil \log n \rceil}} M^t)_{ij} > 0 \Longleftrightarrow [(I+M)^{2^{\lceil \log n \rceil}}]_{ij} > 0$$

Dabei ist I *die Einheitsmatrix, etwa für* $n = 3$*:* $I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

Beweis: Die ersten beiden Äquivalenzen sind leicht zu sehen: Durch die Aufsummierung bedeutet die Eigenschaft "> 0" für ein Knotenpaar (i,j), daß es einen Kantenzug einer Länge zwischen Eins und $n-1$ von i nach j gibt, d.h. insbesondere daß es einen gerichteten i-j-Pfad in D gibt, also $(i,j) \in A^*$. Durch das Hinzuaddieren weiterer Matrizen $M^n, ..., M^{2^{\lceil \log n \rceil}}$ ist die Eigenschaft "> 0" nicht betroffen: Aus Kantenzügen einer Länge größer als $n-1$ lassen sich immer auch Kantenzüge einer kleineren Länge ableiten, weshalb das entsprechende Matrixelement auch ohne die zusätzlichen Matrizen bereits "> 0" sein muss. Zur letzten Äquivalenz:

$$(I+M)^{2^{\lceil \log n \rceil}} = \sum_{t=0}^{2^{\lceil \log n \rceil}} \binom{2^{\lceil \log n \rceil}}{t} M^t I^{2^{\lceil \log n \rceil}-t} \quad (binomischeFormel)$$

Da $M^t I^{2^{\lceil \log n \rceil}-t} = M^t$ unterscheidet sich der Ausdruck vom dritten in Korollar 5.12 nur noch im Vorfaktor, der aber nichts an der Eigenschaft "> 0" ändert. ■

Eine etwas anschaulichere Erklärung für die Aussage $[(I+M)^{2^{\lceil \log n \rceil}}]_{ij} > 0 \Longleftrightarrow (i,j) \in A^*$: Die Matrix $(I+M)$ enthält für jeden Knoten eine Schleife (die von uns betrachteten Graphen sollen selbst keine Schleifen aufweisen). Deshalb enthält die Matrix $(I+M)^2$ alle Kantenzüge mit einer Länge ≤ 2 und entsprechend $(I+M)^{n-1}$ alle Kantenzüge bis zu einer Länge von $n-1$, was bereits für die Ermittlung der transitiven Hülle ausreicht. Dabei bedeutet die Sprechweise "eine Matrix enthält einen Kantenzug", daß ein i-j-Kantenzug durch die Eigenschaft eines Matrixelements $m_{ij} > 0$ als vorhanden angezeigt wird.

Statt nun mit $n-2$ Matrixmultiplikationen $(I+M)^{n-1}$ zu bestimmen, berechnet man durch sukzessives Quadrieren die Matrix $(I+M)^{2^{\lceil \log n \rceil}}$, wozu genau $\lceil \log n \rceil$ Matrixmultiplikationen erforderlich sind. Zum Beispiel aus Abbildung 5.1:

$$(I+M) = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}; \quad (I+M)^8 = \begin{pmatrix} 1 & 8 & 28 & 56 & 70 \\ 0 & 1 & 8 & 28 & 56 \\ 0 & 0 & 1 & 8 & 28 \\ 0 & 0 & 0 & 1 & 8 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Bei großen n können die Matrixelemente dabei sehr groß und die Multiplikation damit sehr teuer werden. Deshalb geht man über zu Booleschen Matrizen: von Interesse ist bei den betrachteten Matrizen ja nur die Eigenschaft "> 0" oder "$= 0$" eines Matrixelementes. Man kann also ohne weiteres nur noch mit den Elementen 0 und 1 arbeiten. Dabei gilt für Addition und Multiplikation:

$\oplus$	0	1
0	0	1
1	1	1

$\odot$	0	1
0	0	0
1	0	1

Die Addition entspricht also dem logischen 'oder', die Multiplikation 'und'.

Damit erhalten wir:

Theorem 5.13 *Sei $T(n)$ die Zeit, um die transitive Hülle eines Digraphen mit n Knoten zu bestimmen und $M(n)$ die Zeit, um zwei Boolesche $n \times n$-Matrizen zu multiplizieren. Dann gilt: $T(n) = \mathcal{O}(M(n) \log n)$.* ∎

Bemerkung 5.14 *Man kann zeigen: Falls $M(2n) \geq 4M(n)$ und $T(3n) \leq 27T(n)$ (was wegen $M(n) = \Omega(n^2)$ und $T(n) = \mathcal{O}(n^3)$ plausibel ist), so gilt*

$$T(n) = \mathcal{O}(M(n)) \qquad \textit{und} \qquad M(n) = \mathcal{O}(T(n)).$$

Zur Booleschen Matrixmultiplikation

1. Die Standardmatrixmultiplikation: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$ mit Boolescher Addition bzw. Multiplikation wie oben angegeben. Für die Laufzeit gilt: $\mathcal{O}(n^3)$

2. Matrixmultiplikation nach Strassen:

   Will man wie im Abschnitt "Divide and Conquer" gezeigt vorgehen, so stößt man auf das grundsätzliche Problem, daß die Boolesche Addition nicht invertierbar ist und deshalb die bei Strassen notwendigen Subtraktionen nicht durchgeführt werden können. Betrachtet man nun die Matrizen doch über $\mathbb{Z}$, so ist dieses Problem zwar gelöst, aber dafür werden die Zahlen wiederum i.a. viel zu groß.

   Man betrachtet deshalb die Matrizen über $\mathbb{Z}_{n+1}$ (Restklassenring modulo $n+1$): Hier stehen zum einen die notwendigen arithmetischen Operationen zur Verfügung und zum anderen gilt:

   $$(AB)_{ij} \mod (n+1) \neq 0 \qquad \Longleftrightarrow \qquad (AB)_{ij} > 0.$$

   Da die Größe der Zahlen in $\mathbb{Z}_{n+1}$ durch n beschränkt ist, sind die arithmetischen Operationen in $\mathcal{O}((\log n)^2)$ durchführbar. Damit ergibt sich für die Laufzeit insgesamt:

   $$\mathcal{O}(n^{\log 7}(\log n)^2) = \mathcal{O}(n^{2.81})$$

3. Der "Vier-Russen-Algorithmus"

   (vorgestellt 1970 von Arlazarov, Dinic, Kronrod, Faradzev [**?**])

   Wir nehmen der Einfachkeit halber hier an, n sei durch $\log n$ teilbar – dann lassen sich die zu multiplizierenden Matrizen A und B restlos in Spalten bzw. Zeilen der Breite/Höhe $\log n$ aufteilen:

   $$AB = \left[\begin{array}{c|c|c|c} A_1 & A_2 & \dots & A_{\frac{n}{\log n}} \end{array}\right] \left[\begin{array}{c} B_1 \\ \hline B_2 \\ \hline \dots \\ \hline \dots \\ \hline B_{n/\log n} \end{array}\right] = \sum_{i=1}^{n/\log n} \underbrace{A_i B_i}_{C_i}$$

   Es handelt sich also bei den A_i um $(n \times \log n)$-, bei den B_i um $(\log n \times n)$- und bei den C_i um $(n \times n)$-Matrizen. Kennt man die C_i, so kann man AB in Zeit $\mathcal{O}(\frac{n}{\log n} \cdot n^2) = \mathcal{O}(\frac{n^3}{\log n})$ berechnen.

   Wir zeigen nun noch, daß auch die Matrizen C_i in jeweils Zeit $\mathcal{O}(n^2)$ berechnet werden können.

   Betrachte eine Zeile von A_i:

   $$\underbrace{[0010101]}_{1 \times \log n} \underbrace{[B_i]}_{\log n \times n} = \underbrace{[\dots]}_{1 \times n}$$

   Offenbar ist das Ergebnis die Summe der dritten, fünften und siebten Zeile von B_i. Berechnet man nun für alle denkbaren Zeilen der A_i, also für $2^{\log n} = n$ mögliche Binärzahlen mit $\log n$ Bit ein Feld von Zeilen

   $$\text{ROWSUM}_i[\sum_{j=1}^{\log n} t_j 2^{j-1}] = \sum_{j=1}^{\log n} t_j \cdot < \textit{j.te Zeile von } B_i >$$

wobei die t_j die einzelnen Bit darstellen, so gilt für die Zeile j von C_i:

$$(C_i)_j = \mathrm{ROWSUM}_i[Value((A_i)_j)].$$

Hier ist $Value((A_i)_j)$ der Wert der j.ten Zeile von A_i interpretiert als Binärzahl. Das Ablesen von C_i aus ROWSUM läßt sich so in Laufzeit $\mathcal{O}(n \cdot (\log n + n)) = \mathcal{O}(n^2)$ durchführen: n Zeilen in A_i, Übertragung des Ergebnisses (n Bit) sowie Berechnung von $Value((A_i)_j)$.

Jetzt muß es nur noch gelingen, das Feld ROWSUM_i ebenfalls in Laufzeit $\mathcal{O}(n^2)$ zu berechnen. Dies erreicht man mit folgendem Algorithmus:

$\mathrm{ROWSUM}_i[0]{=}0$;
for ($j = 1$; $j \le \log n$; $j{+}{+}$)
  for ($k = 0$; $k < 2^{j-1}$; $k{+}{+}$)
    $\mathrm{ROWSUM}_i[2^{j-1} + k] {=} \mathrm{ROWSUM}_i[k] +$ < *j.te Zeile von* B_i >

Man sieht: der Algorithmus greift für die Berechnung einer $(j+1)$-Bit-Zeilensumme (wobei die höchstwertigen $n-j-1$ Bit Null sind) auf die bereits vorher berechneten j-Bit-Zeilensummen zurück und muß lediglich noch das neu hinzugekommene Bit berücksichtigen, sprich die entsprechende Zeile von B_i addieren. Zur Laufzeit: Die verschachtelte Schleife wird $\sum_{j=1}^{\log n} 2^{j-1} = 2^{\log n} - 1 = n - 1$ mal durchlaufen, bei jedem Durchlauf wird in $\mathcal{O}(n)$ die entsprechende Zeile berechnet. Insgesamt ergibt sich also wie gefordert $\mathcal{O}(n^2)$.

Damit ergibt sich für die gesamte Berechnung des Matrixprodukts wie bereits oben angekündigt:

Theorem 5.15 *Der "Vier-Russen-Algorithmus" berechnet das Boolesche Produkt zweier $n \times n$-Matrizen in Laufzeit $\mathcal{O}(\frac{n^3}{\log n})$.*

Bemerkung 5.16 *Während der Algorithmus nach Strassen die theoretisch/asymptotisch günstigere Laufzeit aufweist, ist der "Vier-Russen-Algorithmus" in der Praxis erheblich schneller und deshalb vorzuziehen.*

5.4 Minimal spannende Bäume

Ausarbeitung: *Maximilian Fischer*

Wie bereits aus Kapitel **??** bekannt, kann man zu jedem zusammenhängenden Graphen $G = (V, E)$ einen azyklischen Subgraphen $T = (V, E_T)$ bestimmen, in dem von allen Knoten aus sämtliche anderen Knoten über einen eindeutigen Pfad erreichbar sind. Einen solchen Subgraphen bezeichnet man als *spannenden* Baum (bzw. Spannbaum).

Die Bestimmung eines spannenden Baumes kann beispielsweise mit Hilfe einer Tiefen- oder Breitensuche (vgl. **??**) erfolgen. Da das Ergebnis beider Verfahren sowohl von der Wahl des Startknotens als auch von der Abarbeitungsreihenfolge an Verzweigungen abhängt, ist klar, daß zu jedem zusammenhängenden Graphen im allgemeinen mehrere spannende Bäume existieren.

Für eine Vielzahl von Anwendungen ist es nun wünschenswert, nicht nur einen beliebigen spannenden Baum zu finden, sondern denjenigen, der hinsichtlich einer gegebenen Längenfunktion (bzw. Kostenfunktion) $\ell : E \to \mathbb{R}$ minimal (d.h. optimal) ist. Gesucht ist also ein spannender Baum T in G mit

$$\ell(T) = \min\{\ell(T') \mid T' \text{ ist ein spannender Baum}\}.$$

Dabei gilt $\ell(T) := \sum_{e\in E_T} \ell(e)$. Ein Baum mit dieser Eigenschaft heißt *minimal spannender Baum* (engl. minimum spanning tree, kurz MST). Als Anwendungsbeispiel für minimal spannende Bäume sei die Schleifenunterdrückung in vermaschten Rechnernetzen genannt, wobei hier z.B. hinsichtlich anfallender Leitungskosten optimiert werden kann.

Bemerkung 5.17 *Wie die folgenden Beispiele zeigen werden, ist auch der minimal spannende Baum eines Graphen im allgemeinen nicht eindeutig bestimmt.*

Bereits in Kapitel **??** wurde der Algorithmus von Kruskal vorgestellt, der in $\mathcal{O}(|E|\log|V|)$ einen minimal spannenden Baum berechnet. Die Funktionsweise dieses Algorithmus besteht darin, ausgehend von der isolierten Knotenmenge nach und nach die (hinsichtlich ℓ) „kürzesten“ Kanten hinzuzunehmen, durch deren Hinzunahme kein Zyklus entsteht. Es werden also mehrere Bäume gleichzeitig betrachtet (zu Beginn ist jeder Knoten ein eigener Baum), die schlußendlich zu einem einzigen Baum (dem minimal spannenden Baum) verschmelzen, womit der Algorithmus terminiert. Siehe dazu Abbildung 5.2.

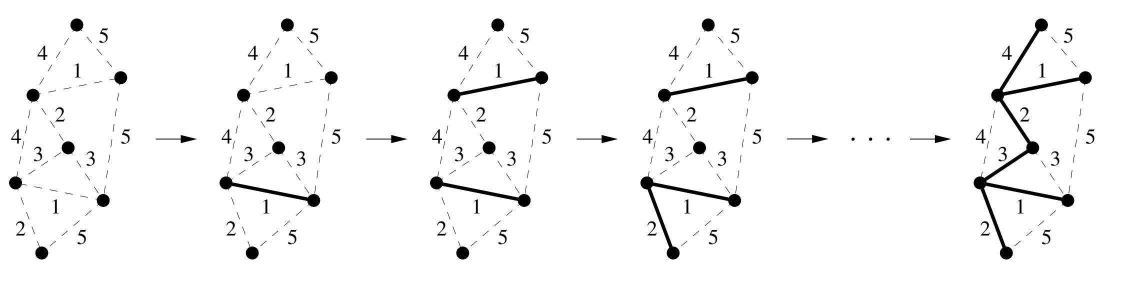


Abbildung 5.2: Beispiel zum Algorithmus von Kruskal

Hier soll nun als weiteres Verfahren zur Bestimmung minimal spannender Bäume der Algorithmus von Prim vorgestellt werden, der gerade für dichte Graphen erheblich bessere Laufzeiten erzielt als der zuvor angesprochene. Dieser Algorithmus entspricht in seiner Arbeitsweise weitgehend dem bereits aus **??** bekannten kürzeste-Wege-Verfahren nach Dijkstra, wie man leicht nachprüfen kann:

<u>Prim(G, ℓ)</u>
/* Initialisierung */
Wähle einen Knoten $s \in V$ beliebig;
$W := \{s\};\ T := (V, \emptyset);$

$$\rho[v] := \begin{cases} 0 & \text{falls } v = s \\ \ell(s,v) & \text{falls } v \in \Gamma(s) \\ \infty & \text{sonst;} \end{cases} \qquad \text{pred}[v] := \begin{cases} s & \text{falls } v \in \Gamma(s) \\ \textbf{nil} & \text{sonst;} \end{cases}$$

/* Hauptschleife */
while $(W \neq V)$
{
  Wähle $x_0 \in V \setminus W$ so daß $\rho[x_0] = \min\{\rho[v] \mid v \in V \setminus W\}$;
  $W := W \cup \{x_0\}$; $T := T + \{x_0, \text{pred}[x_0]\}$;
  for all ($v \in \Gamma(x_0) \cap (V \setminus W)$ mit $\rho[v] > \ell(x_0, v)$)
  {
    $\rho[v] := \ell(x_0, v)$;
    $\text{pred}[v] := x_0$;
  }
}
/* Ausgabe */
return T;

Ausgehend von einem (beliebigen) Startknoten $s \in V$ wird dabei jeweils die „kürzeste“ Kante, die einen der bereits besuchten Knoten (im ersten Schritt ist dies der Startknoten s) mit einem noch unbesuchten Knoten aus deren Nachbarschaft verbindet, zum minimalen Spannbaum hinzugenommen. Zyklen können also hier nicht entstehen und es wird zu jedem Zeitpunkt nur *ein* Baum betrachtet. Der Algorithmus terminiert, sobald alle Knoten besucht worden sind (siehe Abbildung 5.3).

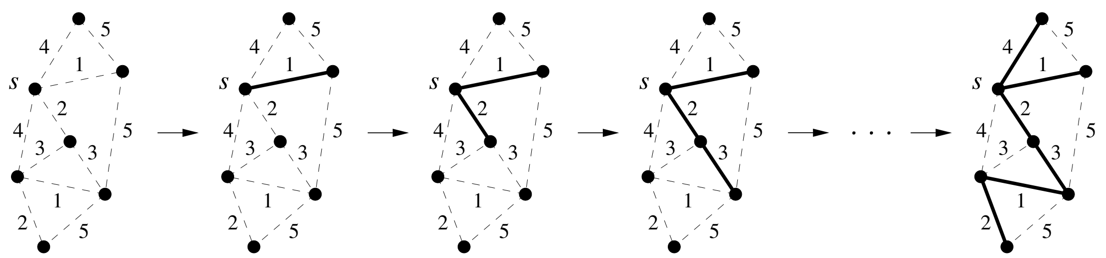


Abbildung 5.3: Arbeitsweise des Algorithmus von Prim

Theorem 5.18 *Der Algorithmus von Prim ist korrekt und hat Laufzeit* $\mathcal{O}(m + n \cdot \log n)$.

Beweis: *Laufzeit:* Der Laufzeitbeweis erfolgt völlig analog zum Algorithmus von Dijkstra (siehe **??**). Dabei wird wieder davon ausgegangen, daß eine Vorrangwarteschlange zur Speicherung der ρ-Werte verwendet wird. Es ist lediglich anzumerken, daß hier (im Gegensatz zum Dijkstra-Verfahren) weder k-level Buckets noch Radix Heaps eingesetzt werden können. Der Grund hierfür ist, daß deren spezielle Monotonie-Anforderungen bezüglich der Insert-Operation nicht immer gegeben sind.

Korrektheit: Wir zeigen, daß die folgenden Invarianten vor und nach jedem Durchlauf der **while**-Schleife gelten:

(1) T ist ein spannender Baum in dem auf die Knotenmenge $W \subseteq V$ beschränkten Graphen $G[W] = (W, E \cap \binom{W}{2})$

(d.h. T ist ein spannender Baum auf der Knotenmenge W)

(2) $\forall\, v \in V \setminus W$ gilt:

$\rho[v] = \min\{\ell(x,v) \mid \{x,v\} \in E,\ x \in W\}$ und $\rho[v] = \ell(v, \mathrm{pred}[v])$

(d.h. $\rho[v]$ ist die Länge einer kürzesten Kante, die einen Knoten des Baumes T mit dem Knoten v verbindet und $\{v, \mathrm{pred}[v]\}$ ist so eine Kante)

(3) es gibt einen minimal spannenden Baum T^* in G mit $T \subseteq T^*$

Unmittelbar einsichtig sind die folgenden beiden Beobachtungen:

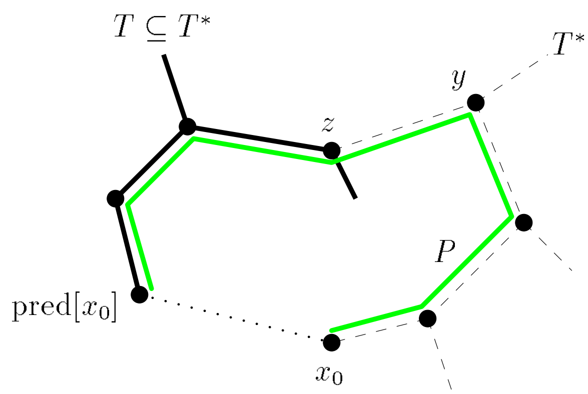
- (1)–(3) gelten nach der Initialisierung
- nach Abbruch der **while**-Schleife (bei $W = V$) ist T wegen (3) ein minimal spannender Baum

Um den Korrektheitsbeweis abzuschließen, muß nun noch die Invarianten-Eigenschaft der Punkte (1)–(3) nachgeprüft werden, wir müssen also zeigen: Gelten (1)–(3) zu Beginn eines Durchlaufs der **while**-Schleife, dann auch am Ende (bezüglich des neuen W).

Die Gültigkeit der Invarianten (1) und (2) folgt direkt aus der Wahl von x_0. Um (3) zu zeigen, sei nun T^* ein minimal spannender Baum, der den bisher bestimmten („alten") Baum T enthält.

1. Fall: Für $\{x_0, \mathrm{pred}[x_0]\} \in T^*$ gilt (3) trivialerweise.

2. Fall: Sei nun $\{x_0, \mathrm{pred}[x_0]\} \notin T^*$ (siehe Skizze).



Betrachte den Pfad P, der x_0 mit $\mathrm{pred}[x_0]$ in T^* verbindet (dieser ist eindeutig bestimmt, da T^* ein Baum ist). Wegen $x_0 \notin W$ und $\mathrm{pred}[x_0] \in W$ muß P eine Kante $\{y, z\}$ mit $y \notin W$ und $z \in W$ enthalten. Nach Wahl von x_0 gilt $\rho[y] \geq \rho[x_0]$. Weiter gilt nach Annahme (2):

$$\begin{aligned} \rho[y] &\leq \ell(y,z) \\ \rho[x_0] &= \ell(x_0, \mathrm{pred}[x_0]) \end{aligned}$$

Betrachte nun den Baum $\widetilde{T}$, der aus T^* entsteht, wenn die Kante $\{y, z\}$ entfernt und stattdessen die neue Kante $\{x_0, \mathrm{pred}[x_0]\}$ hinzugenommen wird, sei also $\widetilde{T} = (T^* - \{y,z\}) + \{x_0, \mathrm{pred}[x_0]\}$. Dann ist $\widetilde{T}$ ein spannender Baum in G (vgl. Skizze) und es gilt:

$$\begin{aligned} \ell(\widetilde{T}) &= \ell(T^*) - \underbrace{\ell(y,z)}_{\geq \rho[y]} + \underbrace{\ell(x_0, \mathrm{pred}[x_0])}_{=\rho[x_0]} \\ &\leq \ell(T^*) - \underbrace{\rho[y] + \rho[x_0]}_{\leq 0} \leq \ell(T^*) \end{aligned}$$

Da T^* nach der Annahme ein minimal spannender Baum war, ist also auch $\widetilde{T}$ ein minimal spannender Baum. Damit existiert mit $\widetilde{T}$ am Ende eines Durchlaufs der **while**-Schleife wieder ein minimal spannender Baum, der den „neuen" Baum T vollständig enthält. ∎

Bemerkung 5.19 *Der Algorithmus von Prim ist für „dichte" Graphen der schnellste bekannte deterministische Algorithmus zur Bestimmung eines minimal spannenden Baumes. Für „dünne" Graphen ist hingegen ein noch etwas schnelleres Verfahren nach Gabow, Galil, Spencer und Tarjan bekannt [**?**], das einen minimal spannenden Baum in* $\mathcal{O}(m \cdot \log \beta(m,n))$ *findet, wobei* $\beta(m,n) = \min\{i \mid \log^{(i)} \frac{m}{n} \leq 1\}$.

*In [**?**] wird ein randomisierter Algorithmus beschrieben, der einen minimalen Spannbaum in* $\mathcal{O}(m)$ *bestimmt.*

Kapitel 6

Wie funktioniert ...

6.1 `fgrep`

6.2 `compress`

Ausarbeitung: *Stephan Schierlinger*

Der Befehl compress basiert auf einem modifizierten LZ-Algorithmus. Im Jahr 1978 veröffentlichten die Herren Abraham Lempel und Jacob Ziv ein Verfahren, das Textstücke in einem Wörterbuch verwaltet und deren Erscheinen innerhalb eines zu komprimierenden Text durch den Index im Wörterbuch ausgedrückt wird.

1984 modifizierte Terry Welch, für die Hardware-Implementierung von Festplatten Controllern, den LZ78 zu einem Verfahren das als LZW bezeichnet wird. Er ging darin nicht von einem festen Wörterbuch aus das mit der komprimierten Datei übertragen werden mußte, sondern von einem minimalen Wörterbuch das auf allen Systemen verfügbar ist und das sich durch den eingegeben Text dynamisch vergrößert. Beim Entkompremieren wird dann das identische Wörterbuch aufgebaut.

Begonnen wird mit einem Wörterbuch von 4K Größe. Die Einträge 0-255 beinhalten die Zeichen eines Alphabets, die Einträge 256-4096 beziehen sich auf Zeichenketten. Der zu komprimierende Text wird zeichenweise durchgelesen und zu Strings zusammengefügt. Falls der String im Wörterbuch bereits enthalten ist wird das nächste Zeichen an den bisherigen String angefügt, andernfalls wird der Index-Eintrag des Strings ohne das letzte Zeichen in die Ausgabe-Datei geschrieben und der Teilstring als neuer Eintrag am Ende des Wörterbuchs angehängt. Der Algorithmus für LZW sieht folgendermaßen aus:

```
set w = NIL
loop
    read a character K
    if wK exists in the dictionary
         w = wK
    else
```

```
            output the code for w
            add wK to the dictionary
            w = K
    endloop
```

Jede Ausgabe wird an eine Datei angehängt die letztendlich das Ergebnis der Kompression ist und die nötige Information enthält um die ursprüngliche Information zurückzugewinnen.

Beispiel:

Verschlüsselt wird das Wort `barbapapa`. In der folgenden Tabelle wurden die Zeilen weggelassen, in denen die Eingabe bereits im Wörterbuch vorhanden ist.

Eingabe	Ausgabe	Wörterbuch-Index
ba	b	256 = ba
ar	a	257 = ar
rb	r	258 = rb
bap	256	259 = bap
pa	p	260 = pa
ap	a	261 = ap
pa`<END&gt`	260	—

Tabelle 6.1: Compress angewandt auf barbapapa

Komprimiert sieht das Ganze so aus: `bar<256>pa<260>` Zur Dekompression wird derselbe Algorithmus verwendet. Die Indizies der Eingabe werden durch ihren Eintrag im bisherigen Wörterbuch desjenigen der die Dekompression durchführt ersetzt. Außerdem kommen nun die jeweiligen Wörterbucheinträge zur Ausgabe.

Eingabe	Ausgabe	Wörterbuch-Index
ba	b	256 = ba
ar	a	257 = ar
rb	r	258 = rb
bap	ba	259 = bap
pa	p	260 = pa
ap	a	261 = ap
pa	pa	—

Tabelle 6.2: Decompress angewandt auf Kompression von barbapapa

Es ist anzumerken, daß UNIX eine leicht modifizierte Variante, genannt LZC verwendet, die mit 9 Bit langen Indizes für die Wörterbucheinträge beginnt und diese um ein Bit verlängert sobald das Wörterbuch voll ist. Jede dieser Varianten kontrolliert mit einem Monitor die Performance des Algorithmus, um bei einer Absenkung der Effizienz das Wörterbuch zu verwerfen. In der komprimierten Datei wird dies durch ein speziellen Eintrag signalisiert.

Ergänzende Literatur:

- K.Sayood, "Data Compression", Morgan Kaufmann, 1996
- T.Welch, "A Technique for High-Performance Data Compression", Computer, June 1984
- M.R.Nelson, "LZW Data Compression", Dr. Dobb's Journal, Octtober 1989

Quellen:

- Beiträge aus der Newsgroup comp.compression
- Pro-Seminar von Christian Naeger (14.1.97; http://wwwmayr.informatik.tu-muenchen.de/lehre/1996WS/proseminar)

Literaturverzeichnis

[DZ95] D. Dor and Uri Zwick. Selecting the median. In *Proc. of 6th SODA*, pages 28–37, 1995.

[DZ96] D. Dor and Uri Zwick. Median selection requires $(2+\epsilon)n$ comparisons. In *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Sciene*, pages 125–134, 1996.