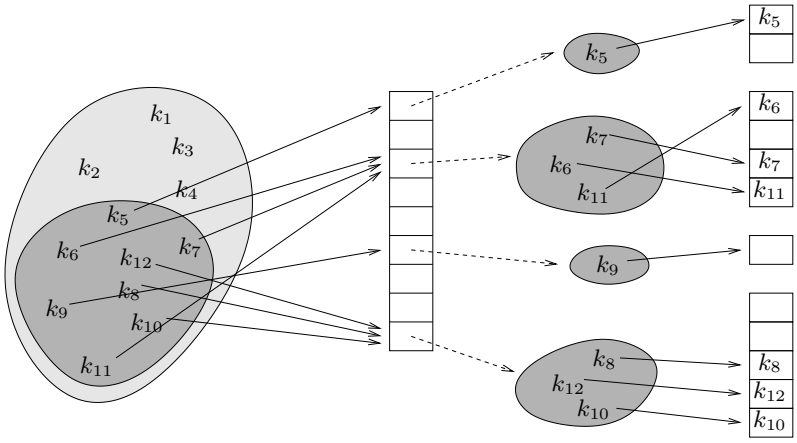
Skript zur Vorlesung:

Komplexe Datenstrukturen



Wintersemester 2000/01

Dr. Andreas Jakoby

Institut für Theoretische Informatik
Medizinische Universität zu Lübeck

Inhaltsverzeichnis

1 Definitionen und Berechnungsmodelle

Diese Vorlesung basiert weitgehend auf einer Vorlesung, die im Winter 1999/2000 an der Universität von Toronto von Faith Fich gehalten wurde. Ich möchte mich an dieser Stelle bei ihr für die zur Verfügungstellung ihrer Unterlagen bedanken.

1.1 Abstrakte Datentypen vs. Datenstrukturen

Ein abstrakter Datentyp (ADT) besteht aus

1. einer Kollektion mathematischer Objekte
2. Operationen, die auf diesen Objekten ausgeführt werden können
3. Bedingungen, die erfüllt sein müssen

Ein ADT ist somit eine Spezifikation. Beispiele für einen ADT sind

1. der Stack
   - eine Sequenz von Elementen eines Universums, zum Beispiel eine Sequenz von Zahlen
   - die Operationen: `push`, `pop` und *newstack*
2. ein statisches Wörterbuch (static dictionary)
   - eine Menge von Elementen eines Universums
   - die Operation: *ist ein Element von*
3. ein dynamisches Wörterbuch (dynamic dictionary)
   - eine Menge von Elementen eines Universums
   - die Operationen *ist ein Element von*, *insert*, *delete*

Eine Datenstruktur ist eine Implementation eines ADT. Sie besteht daher aus einer Repräsentation der Objekte auf einem Computer sowie aus den dazugehörigen Algorithmen.

Beispiele für Datenstrukturen für statische oder dynamische Wörterbücher sind: binäre Suchbäume, AVL-Bäume, sortierte Arrays, Hashtabellen.

1.2 Berechnungsmodelle

Je nach dem benutzten Berechnungsmodell können Datenstrukturen mehr oder weniger effizient sein. Im wesentlichen unterscheiden wir die folgenden Modelle:

1. **Comparison Trees**: Die einzigen erlaubten Operationen sind Vergleiche zwischen den Elementen.
   Vorteile: einfach zu analysieren, gute untere Schranken
   Nachteile: sehr schwach, Hashing oder Bucket Sort können nicht implementiert werden

2. **Unit Cost RAM**: Der Speicher besteht aus einem unbegrenzten Array von Registern, die über einen positiven Integerwert adressiert werden können. Jedes Register kann einen beliebigen (unbegrenzten) Wert speichern. Jede Operation (wie die direkte und indirekte Adressierung, bedingte Sprünge, Addition, Multiplikation) kostet eine Einheit.
   Vorteile: einfach und universell
   Nachteile: zu stark, eine ganze Datenstruktur passt in ein Wort (siehe Beispiel unten)

3. **RAM mit konstanter Wortgröße**: Jedes Register kann nur ein Wort konstanter Größe speichern. Jede Operation kostet eine Einheit.
   Vorteile: Reale Computer haben eine feste Wortgröße
   Nachteile: zu schwach, es kann nur ein begrenzter Speicherbereich adressiert werden. Die Maschine ist somit nicht mächtiger als ein endlicher Automat.

4. **log-Cost RAM**: Die Kosten jeder einzelnen Operation hängen von der Anzahl der Bits der Operanden ab.
   Vorteile: Realistisch für sehr große Operanden
   Nachteile: schwer zu analysieren und stellenweise schwach: Addition nicht allzu großer Zahlen und das Folgen von Pointern ist nicht in konstanter Zeit möglich.

5. **Word Based RAM**: (oder Word Level RAM, Transdichotomous Modell, Conservative Modell, Random Access Computer (RAC)) Algorithmen für b-Bit Integerwerte als Eingabe können O(b)-bit Wörter verwenden. Somit kann eine konstante Anzahl von Eingabewerten in einem Wort gespeichert werden. Eine Operation kostet eine Einheit.
   Vorteile: einfach zu analysieren, lässt beschränkten Parallelismus zu (auf dem Wort-Level), gebräuchliches Modell in der heutigen Datenstrukturliteratur.

Zum Beweis von unteren Schranken erlauben wir eine beliebige Menge von Operationen, so lange sie nur eine konstante Anzahl von Register verändern oder benutzen. Für die oberen Schranken erlauben wir die Basisoperationen: Addition, Subtraktion, Multiplikation, Division sowie das bitweise And, Or, Not, Left-Shift und Right-Shift. Ferner wird für einige Algorithmen der Befehl „Random“ erlaubt.

Zu Beginn einer Berechnung sind alle Register mit 0 initialisiert. Alternativ kann man davon ausgehen, daß alle Register mit zufälligen Werten initialisiert sind.

1.3 Simulation von 0-initialisierten Speichern bei nicht initialisierten Speichern

Sei M eine RAM mit nicht initialisiertem Speicher bestehend aus überlappten unendlichen Arrays A und B. Zudem sei c ein Zähler.

A gibt den Inhalt der Speicherzellen an, in welche der Algorithmus schon geschrieben hat. Speicherstellen, auf welche noch nicht zugegriffen wurde, enthalten eine zufälligen unnützen Wert (Garbage). Um zwischen den Speicherstellen zu unterscheiden, auf die schon zugegriffen wurde, und solchen, die noch ihren ursprünglichen Wert speichern, werden wir eine Liste aller Speicherstellen verwalten, in die schon einmal geschrieben wurde. Diese wird in B verwaltet. c gibt die Länge dieser Liste an. Will der Algorithmus auf eine Speicherstelle zugreifen, so durchsuchen wir zunächst diese Liste. Wird die Adresse nicht gefunden, so fügen wir sie an das Ende der Liste an und schreiben eine 0 an die entsprechende Stelle.

Ein Problem bei dieser Lösung ist, dass sie die Laufzeit eines Algorithmuses quadrieren kann. Daher verwenden wir ein drittes Array C. Wenn $A[i]$ initialisiert wurde, speichern wir an $C[i]$ die erste Stelle in B, welche den Wert i enthält. Um zu testen, ob $B[i]$ schon initialisiert wurde, genügt es zu testen, ob $B[C[i]] = i$ und $C[i] \leq c$ ist. Diese Tests können in konstanter Zeit ausgeführt werden.

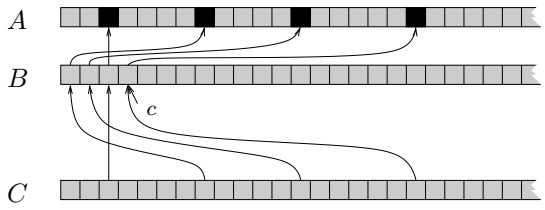


Abbildung 1: Speicher einer RAM mit nicht initialisiertem Speicher zur Simulation von 0-initialisiertem Speicher.

1.4 Ein Suchalgorithmus für eine Unit Cost RAM in konstanter Zeit

Sei $S := \{y_1, \ldots, y_n\} \subseteq [0, 2^b - 1]$. Wir repräsentieren S durch ein einzelnes $n(b+1)$-Bit Wort Y.

Sei $x \in [0, 2^b - 1]$. Wir repräsentieren x durch einen b-Bit String. Multipliziere x mit $(0^b 1)^n$. Wir erhalten den $n(b+1)$-Bit String $(0x)^n$. Z sei das bitweise XOR von $(0x)^n$ mit Y. Für Z gilt nun, dass es in Z eine Bitsequenz $Z[i \cdot (b+1), i \cdot (b+1)+b] = 0^{b+1}$ für $i = 1, 2, \ldots, n$ gibt, genau dann wenn eine Zeichenkette $y_i = x$ ist. Zudem gilt, dass Z an jeder $(b+1)$-ten Stelle $Z[0], Z[b+1], Z[2b+2], \ldots$ gleich 0 ist.

Subtrahieren wir Z von $(10^b)^n$, so ist das Ergebnis Z' nur dann an einer der $(b+1)$-ten Stelle $Z'[0], Z'[b+1], Z'[2b+2], \ldots$ gleich 1, wenn es eine Bitsequenz $Z[i \cdot (b+1), i \cdot (b+1)+b] = 0^{b+1}$ gab. Sei Z'' das bitweise AND von $(10^b)^n$ und Z', so ist Z'' genau dann größer 0, wenn $x \in S$ ist.

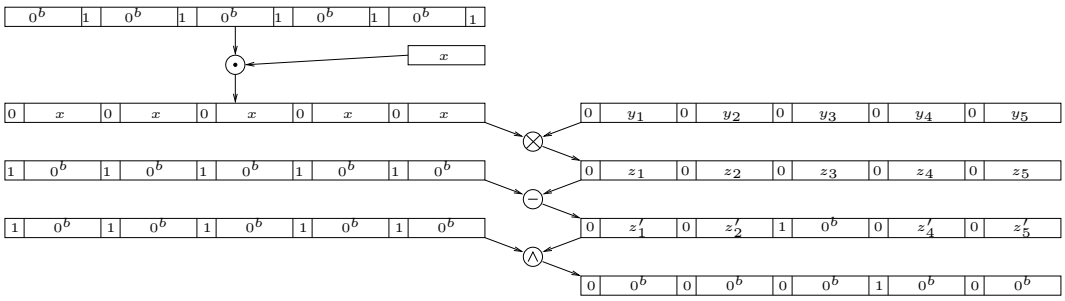


Abbildung 2: Statisches Wörterbuch auf einer Unit-Cost RAM.

1.5 Zeit- und Platzkomplexitätsmaße

Wir messen die Zeit, die benötigt wird, um eine Operation auszuführen, sowie den benötigten Platz, um die Repräsentation einer Datenstruktur zu speichern. Der Platz ergibt sich somit aus dem Index des größten Registers, welches einen Wert ungleich 0 speichert.

Für die Zeit unterscheiden wir folgende Komplexitätsmaße:

1.5.1 Worst-Case Zeitkomplexität

Das Worst-Case Zeitmaß eines Algorithmus auf einer Datenstruktur der Länge n definieren wir wie folgt:

$$T(n) := \max_{\text{Instanzen } S \text{ der Datenstruktur der Laenge } n} \text{Zeit } t(S)\text{, die der Algorithmus auf } S \text{ benötigt.}$$

Zum Beispiel wird für die Suche eines Elements in einem 2-3 Baum eine Worst-Case Zeit von $O(\log n)$ benötigt, wenn n die Anzahl der Elemente im 2-3 Baum angibt.

1.5.2 Randomisierte Zeitkomplexität

Ein randomisierter Algorithmus kann zur Berechnung Zufallszahlen (Münzwürfe) zur Hilfe nehmen. Seine Laufzeit kann vom aktuellen Ergebnis der Münzwürfe abhängen. Wir betrachten daher die erwartete Laufzeit eines Algorithmus auf einer Instanz S:

$$E(t(S)) := \sum_{\text{Sequenz von } r \text{ Ergebnissen von Muenzwuerfen}} (\text{Zeit } t(S,r)\text{, die der Algorithmus auf } S \text{ und } r \text{ benötigt}) \cdot \text{Prob}(r)\ .$$

Für die randomisierte Zeitkomplexität ergibt sich somit:

$$T_{\text{rand}}(n) := \max_{\text{Instanzen } S \text{ der Datenstruktur der Laenge } n} E(t(S))\ .$$

1.5.3 Amortisierte Zeitkomplexität

Bei der amortisierten Zeitkomplexität betrachten wir nicht eine einzelne Operation losgelöst von ihrem Kontext, sondern vielmehr den mittleren Zeitaufwand einer Operation innerhalb einer Sequenz von Operationen.

$$T_{\text{amort}}(n) := \sup_{\text{Sequenz } \sigma \text{ von Operationen startend von einem fixen Startzustand}} \frac{\text{Zeit um } \sigma \text{ auszuführen}}{\text{Länge von } \sigma}\ .$$

Eine einfache Überlegung zeigt, dass die amortisierten Kosten eines Algorithmus immer kleiner gleich dessen Worst-Case Kosten sind, da die amortisierten Kosten ein average Maß für eine einzelne Operation angibt.

2 Implementierung eines Stacks durch Array

In diesem Kapitel wollen wir das amortisierte Zeitverhalten zweier Datenstrukturen untersuchen, welche einen Stack mit Hilfe eines Arrays implementieren. Die betrachteten Operationen sind hierbei `push` für das Hinzufügen eines Elements auf den Stack, sowie `pop`, um das oberste Element vom Stack zu entfernen.

2.1 Implementierung ohne Speicherfreigabe nach pop-Sequenz

Unser Algorithmus speichert die Elemente des Stacks in der durch den Stack vorgegebenen Reihenfolge in einem Array. Wann immer ein `push`-Befehl einen Überlauf des Arrays zur Folge hat – das Array hat bereits im Schritt zuvor seine Fassungsgrenze erreicht – so reservieren wir ein neues doppelt so großes Array (beachte, dass diese Operation in dem von uns gewählten Modell nur eine konstante Anzahl von Schritte benötigt) und kopiere die alten Elemente in das neue Array. Zu Beginn gehen wir davon aus, dass das Array leer ist und die Länge 1 hat.

Sei σ eine Sequenz von n Operationen und betrachten wir den Fall, dass jedes `pop` und jedes einfache `push` in genau einer Zeiteinheit ausgeführt werden kann, so erkennt man schnell, dass σ genau dann die maximalen Kosten verursacht, wenn es zu einer maximalen Anzahl von Arrayüberläufen führt – wenn σ eine reine `push`-Sequenz ist. Wir erhalten daher für die Operationen folgende Kostenfunktion:

$$\text{Kosten für das } i\text{-te } \texttt{push} \;:=\; \begin{cases} 1, & \text{wenn } i-1 \text{ keine Potenz von 2 ist} \\ i+c, & \text{wenn } i-1 \text{ eine Potenz von 2 ist,} \end{cases}$$

wobei c die Kosten sind, die durch die Reservierung des neuen und die Freigabe des alten Arrays entstehen. Für die Sequenz σ ergibt sich somit:

$$\begin{aligned} \sum_{i=1}^{n} \text{Kosten für das } i\text{-te } \texttt{push} &= \sum_{1\le i\le n,\ \forall j\in\mathbb{N}: i-1\neq 2^j} 1 \;+\; \sum_{1\le i\le n,\ \exists j\in\mathbb{N}: i-1=2^j} (i+c) \\ &= n + \sum_{1\le i\le n,\ \exists j\in\mathbb{N}: i-1=2^j} (i-1+c) \\ &= n + \sum_{j=1}^{\lfloor \log_2(n-1)\rfloor} (2^j + c) \\ &= n + \lfloor \log_2(n-1)\rfloor \cdot c + 2^{\lfloor \log_2(n-1)\rfloor+1} - 1 \\ &< 3n + \lfloor \log_2(n-1)\rfloor \cdot c\ . \end{aligned}$$

Vernachlässigen wir also die Kosten für das Reservieren des Speicherplatzes (deren Anteil an der amortisierten Zeit $\frac{\lfloor \log_2(n-1)\rfloor \cdot c}{n}$ für wachsende Sequenzlängen gegen 0 geht), so erhalten wir für die amortisierten Kosten einen Wert kleiner 3.

2.2 Implementierung mit Speicherfreigabe nach pop-Sequenz

Der erste (naive) Algorithmus arbeitet wie folgt:

- Wie im obigen Algorithmus verdoppeln wir die Arraygröße, wenn wir durch einen `push`-Befehl einen Überlauf des Arrays erhalten. Wir benötigen daher für ein `push` auf einen Stack S die folgende Zeit:

$$t_{\texttt{push}}(S) \;:=\; \begin{cases} 1, & \text{wenn } |S| \text{ keine Zweierpotenz ist} \\ |S|+1+c, & \text{wenn } |S| \text{ eine Zweierpotenz ist} \end{cases}$$

$|S|$ bezeichnet hierbei die Anzahl der Elemente im Stack S vor dem `push`-Befehl. Wie im obigen Algorithmus sei c die Zeit, die benötigt wird, um ein neues Array zu reservieren.

- Wir halbieren die Arraygröße, wenn das Array nach einem `pop`-Befehl nur noch halb gefüllt ist, d.h. wir reservieren ein Array halber Größe und speichern die Elemente, die sich noch im Stack befinden neu in das verkleinerte Array. Für ein `pop` erhalten wir folgendes Zeitverhalten:

$$t_{\texttt{pop}}(S) := \begin{cases} 1, & \text{wenn } |S|-1 \text{ keine Zweierpotenz ist} \\ |S|+c, & \text{wenn } |S|-1 \text{ eine Zweierpotenz ist} \end{cases}$$

$|S|$ bezeichnet hierbei die Anzahl der Elemente im Stack S vor dem `pop`-Befehl.

Betrachten wir nun die folgende Befehlssequenz der Länge $n = 2^k + 1$:

1. führe $2^{k-1}+1$ `push`-Befehle aus
2. wiederhole 2^{k-2} mal ein `pop-push`-Paar

Wir erhalten somit folgendes Platz-/Zeitverhalten für den zweiten Schritt: Da vor dem ersten `pop` das Array eine Größe von 2^k hat und $2^{k-1}+1$ Elemente speichert, benötigen wir für ein `pop` $2^{k-1}+c$ Schritte. Nach dem `pop` hat das Array eine Größe von 2^{k-1} und speichert auch 2^{k-1} Elemente. Ein `push` benötigt daher $2^{k-1}+1+c$ Schritte und resultiert in einem Array der Größe 2^k, welches $2^{k-1}+1$ Elemente speichert. Dieses entspricht jedoch der Konstellation vor dem `pop-push`-Paar. Eine Sequenz von 2^{k-2} `pop-push`-Paaren benötigt daher eine Zeit von

$$2^{k-2} \cdot (2^k + 1 + c) = \frac{(n-1)^2 + (n-1)\cdot(1+c)}{4} \in \Theta(n^2) \ .$$

Die amortisierte Laufzeit ist somit zumindest linear in der Länge der Befehlssequenz. Das Worst-Case Verhalten dieses Verfahrens ist ebenfalls linear. Für die amortisierte Laufzeit erhalten wir daher $\Theta(n)$.

Im Folgenden wollen wir ein verbessertes Verfahren untersuchen:

- Wie zuvor verdoppeln wir die Arraygröße, wenn wir durch einen `push`-Befehl einen Überlauf des Arrays erhalten. Die Ausführungszeit eines `push`-Befehls auf einen Stack S und einem Array A der Größe $|A|$ ist somit:

$$t_{\texttt{push}}(S,A) := \begin{cases} 1, & \text{wenn } |S| < |A| \text{ keine Zweierpotenz ist} \\ |S|+1+c, & \text{wenn } |S| = |A| \text{ eine Zweierpotenz ist.} \end{cases}$$

- Wir halbieren die Arraygröße, wenn das Array nach einem `pop`-Befehl nur noch zu einem Viertel gefüllt ist. Für ein `pop` auf einen Stack S und einem Array A der Größe $|A|$ erhalten wir folgendes Zeitverhalten:

$$t_{\texttt{pop}}(S,A) := \begin{cases} 1, & \text{wenn } |S|-1 > \frac{|A|}{4} \text{ keine Zweierpotenz ist} \\ |S|+c, & \text{wenn } |S|-1 = \frac{|A|}{4} \text{ eine Zweierpotenz ist.} \end{cases}$$

Da sich die Bestimmung der amortisierten Zeit dieses Verfahrens schwieriger gestaltet, sollen im Folgenden zwei Verfahren vorgestellt werden, die uns diese Analyse erleichtern sollen:

2.2.1 Die Bilanzmethode (Accounting Method)

Zu Beginn weisen wir jedem Befehl einen Betrag zu, der eine (geschätzte) obere Grenze der amortisierten Kosten dieses Befehls entsprechen soll. Dieser Betrag wird beim Ausführen dieses Befehls auf ein Guthabenkonto verbucht. Zur Vereinfachung werden wir dieses Konto in eine entsprechend gewählten Datenstruktur speichern. Mit Hilfe dieses Guthabenkontos zahlen wir nach jedem Schritt die Kosten des eben ausgeführten Befehls. Wir müssen sicherstellen, d.h. beweisen, dass unser Konto nie einen negativen Betrag anzeigt. Dieses bedeutet:

$$\sum \text{Aktuelle Kosten der Operationen} \quad \leq \quad \sum \text{Kredit für die Operationen} .$$

Für den oben angegebenen Algorithmus wählen wir:

- für einen `push`-Befehl einen Kredit von 3\$ und
- für einen `pop`-Befehl einen Kredit von 2\$.

Wir wollen im Folgenden über eine vollständige Induktion zeigen, dass diese Beträge eine obere Schranke für die amortisierten Kosten darstellen. Hierfür speichern wir an jeder Arrayposition den Betrag, den wir über eine auf diese Position ausgeführten letzten Befehl erhalten. Man beachte, dass über diese Organisation des Guthabenkontos Beträge verloren gehen können. Ist die Summe der gespeicherten Beträge jedoch weiterhin größer gleich Null, so geben die gefundenen Befehlsbeträge weiterhin eine obere Schranke für die amortisierte Zeit an. Im weiteren werden wir davon ausgehen, dass keine Kosten für das Reservieren eines neuen Arrays anfallen, d.h. $c = 0$.

Wir wollen nun die folgende Invariante untersuchen:

> In einem Array A der Größe 2^k sind zumindest die unteren 2^{k-2} Positionen besetzt. Für jedes Element, welches in der oberen Hälfte gespeichert ist, weist unser Konto 2\$ Guthaben auf. Das Konto weist für jede freie Arrayposition im zweiten Viertel von unten einen Guthabensbetrag von 1\$ auf.

Der erste Teil der Invariante, dass ein Array der Größe 2^k sind zumindest 2^{k-2} Elemente speichert, folgt unmittelbar aus der Konstruktion unseres Verfahrens.

Betrachten wir nun den Rest unserer Invariante:

Für einen leeren Stack (bevor ein `push` oder ein `pop` ausgeführt wurde) gilt die Behauptung unmittelbar. Nach dem ersten `push` speichert das Array ein Element. Zudem können wir ein Guthaben von 2\$ verbuchen, da wir das Array ja noch nicht vergrößern müssen.

Nehmen wir nun an, dass die Invariante auch nach den ersten $i - 1$ Operationen gültig ist. Für die i-te Operation müssen wir nun zwischen den folgenden Fällen unterscheiden:

1. die i-te Operation ist ein `push`, welches keinen Arrayüberlauf zur Folge hat,
2. die i-te Operation ist ein `pop`, welches keine Arrayverkleinerung zur Folge hat,
3. die i-te Operation ist ein `push`, welches einen Arrayüberlauf zur Folge hat, und
4. die i-te Operation ist ein `pop`, welches eine Arrayverkleinerung zur Folge hat.

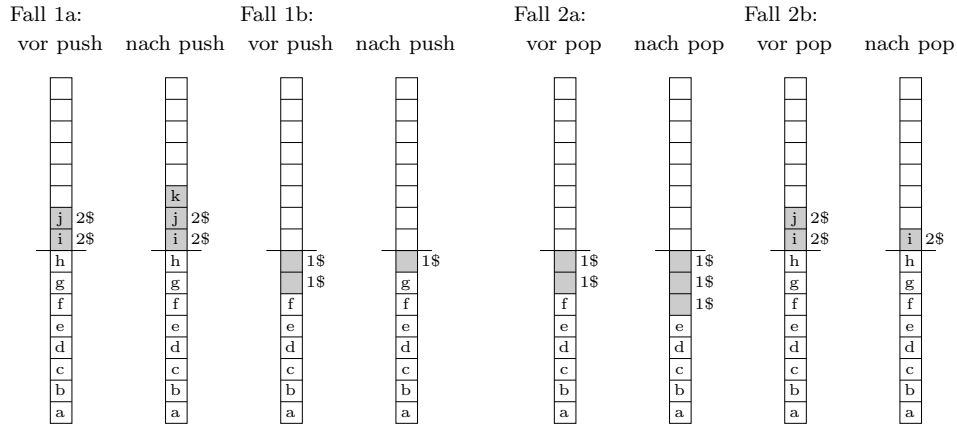


Abbildung 3: Kontoänderung bei `push` (links) und `pop` (rechts) ohne eine Vergrößerung oder Verkleinerung der Arraylänge.

Für den ersten und zweiten Fall folgt die Behauptung unmittelbar (siehe Abbildung 3)

Betrachten wir zunächst den `push`-Befehl. Hier müssen wir zwei Unterfälle unterscheiden. Im Fall 1a wird ein Element in der oberen Hälfte eingetragen. Dieses kostet 1$, erhöht jedoch den Betrag für die obere Hälfte um 2$. Schreibt ein `push` jedoch ein Element in das zweite Viertel von unten (Fall 1b), so können wir den Zugewinn von 3$ vernachlässigen und die Kosten für das `push` über einen in diesem Viertel gespeicherten Dollar zahlen.

Die Invariante gilt somit nach einem `push`, welcher keinen Arrayüberlauf zur Folge hat.

Kommen wir nun zum zweiten Fall. Wie zuvor, müssen wir zwei Unterfälle betrachten. Im Fall 2a wird ein Element aus dem zweiten Viertel von unten entfernt. Wir können somit nach dem Abzug der Kosten von 1$ unseren Kontostand für dieses Viertel um einen Dollar erhöhen. Löschen wir jedoch ein Element aus der oberen Hälfte, so können wir das Guthaben von 2$, welches wir für dieses Element speichern im Folgenden vernachlässigen und zahlen die Operationskosten von den 2$, die wir für das `pop` erhalten. Den verbleibenden Dollar werden wir im Weiteren vernachlässigen.

Die Invariante gilt somit auch nach einem `pop`, welches keine Arrayverkleinerung zur Folge hat.

Betrachten wir nun den Fall, dass die i-te Operation ein `push` ist, welches zu einem Arrayüberlauf führt. Das Array speichert somit vor diesem `push` $|A| = 2^k$ Elemente und weist ein Guthaben von $2 \cdot \frac{|A|}{2}\$ = |A|\$$ in der oberen Hälfte auf. Zum Kopieren der alten Daten benötigen wir $|A|\$$, welche wir über dieses Guthaben begleichen können. Das Hinzufügen des neuen Elements kostet 1$. Somit können wir noch 2$ für das neue (und jetzt einzige) Element in der neuen oberen Hälfte verbuchen (siehe Abbildung 4).

Dieses bedeutet, dass die Invariante nach einem `push` mit Arrayüberlauf gültig ist.

Kommen wir nun zum letzten Fall, dem `pop` mit einhergehender Verkleinerung des Arrays. Vor dieser Operation sind in dem Array $|A|/4 + 1$ Elemente gespeichert. Zudem weist unser Konto für das zweite Viertel von unten ein Guthaben von $|A|/4 - 1\$$ auf. Der `pop`-Befehl kostet 1$ und erhöht unser Guthaben zudem auf $|A|/4\$$. Dieses Guthaben können wir nun einsetzen, um die verbleibenden $|A|/4$ noch im Array gespeicherten Elemente in das neue Array der Länge $|A|/2$ zu übertragen. Das neue Array weist nach der Operation kein Guthaben auf, was auch nicht in der Invariante gefordert wird.

Wir können somit schließen, dass die Invariante nach einem `pop` mit Arrayverkleinerung gültig ist.

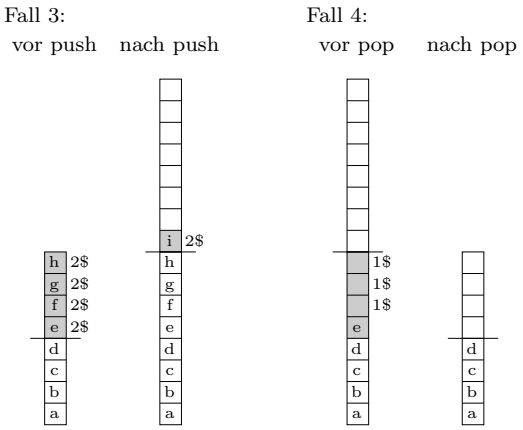


Abbildung 4: Kontoänderung bei `push` (links) und `pop` (rechts) bei Vergrößerung oder Verkleinerung der Arraylänge.

Zusammenfassend gönnen wir somit sagen, dass die amortisierte Zeit für ein `push` nach oben durch 3 und für ein `pop` durch 2 beschränkt ist.

Im Folgenden wollen wir noch eine weitere Methode zur Begrenzung der amortisierten Kosten vorstellen.

2.2.2 Die Potentialmethode

Bei der Potentialmethode versuchen wir mit Hilfe einer Potentialfunktion Φ Schwankungen der Kostenfunktion auszugleichen. Die Potentialfunktion ist dabei eine Abbildung vom Zustandsraum der Datenstruktur auf die ganzen Zahlen. Ein Beispiel für eine solche Funktion für einen Stack S und ein Array $|A|$ ist:

$$\Phi(S, A) \;=\; |\; 2 \cdot |S| \;-\; |A|\; | \;.$$

Der eigentliche Ausgleich der Schwankungen der Kostenfunktion erfolgt über die Potentialdifferenz

$$\Delta\Phi(D_{i-1}, D_i) \;:=\; \Phi(D_i) - \Phi(D_{i-1})$$

einer Operation auf einer Instanz D_{i-1} der Datenstruktur.

Hier bezeichnen D_{i-1} und D_i den Zustand der Datenstruktur vor und nach der Operation. Bezeichnen wir nun mit c_i die für die i-te Operation anfallenden Kosten, dann untersuchen wir nun die durch die Potentialdifferenz bereinigte Kostenfunktion:

$$\hat{c}_i \;:=\; c_i \;+\; \Delta\Phi(D_{i-1}, D_i) \;.$$

Für die Summe der Kosten gilt nun:

$$\sum_{i=1}^{n} c_i \quad = \quad \sum_{i=1}^{n} (\hat{c}_i - \Delta\Phi(D_{i-1}, D_i)) \quad = \quad \sum_{i=1}^{n} \hat{c}_i \;-\; \sum_{i=1}^{n} (\Phi(D_i) - \Phi(D_{i-1}))$$

$$= \sum_{i=1}^{n} \hat{c}_i \; - \; \Phi(D_n) \; + \; \Phi(D_0)$$

Für $\Phi(D_n) \leq \Phi(D_0)$ ist daher $\sum_{i=1}^{n} \hat{c}_i$ eine obere Schranke für Kosten der vorliegenden Sequenz. Für die Potentialmethode sollten wir die Potentialfunktion Φ so wählen, dass die Summe $\sum_{i=1}^{n} \hat{c}_i$ einfach zu berechnen ist.

Für das vorliegende Beispiel gilt nun $\Phi(S_0, A_0) = 1$. Im Folgenden soll nun gezeigt werden, dass $\hat{c}_i \leq 3$ für jede mögliche Sequenz σ und alle Positionen i in dieser Sequenz ist. Um dieses zu beweisen, müssen wir die folgenden vier Fälle analysieren:

1. Die i-te Operation ist ein `push`, welche keinen Arrayüberlauf zur Folge hat.

   In diesem Fall ist $c_i = 1$. Zudem gilt für $2 \cdot |S_{i-1}| \geq |A_{i-1}|$, d.h. das `push` fügt ein Element in die obere Hälfte des Arrays ein:

   $$\begin{aligned} \Delta\Phi(S_{i-1}, A_{i-1}, S_i, A_i) &= | \, 2 \cdot |S_i| - |A_i| \, | \; - \; | \, 2 \cdot |S_{i-1}| - |A_{i-1}| \, | \\ &= 2 \cdot (|S_{i-1}| + 1) - |A_{i-1}| - 2 \cdot |S_{i-1}| + |A_{i-1}| \;\; = \;\; 2 \; . \end{aligned}$$

   Fügt das `push` das Element in das zweite Viertel von unten ein, d.h. $2 \cdot |S_{i-1}| < |A_{i-1}|$, so gilt:

   $$\begin{aligned} \Delta\Phi(S_{i-1}, A_{i-1}, S_i, A_i) &= | \, 2 \cdot |S_i| - |A_i| \, | \; - \; | \, 2 \cdot |S_{i-1}| - |A_{i-1}| \, | \\ &= |A_{i-1}| - 2 \cdot (|S_{i-1}| + 1) - |A_{i-1}| + 2 \cdot |S_{i-1}| \;\; = \;\; -2 \; . \end{aligned}$$

   Wir erhalten somit: $\hat{c}_i = 1 \pm 2 \leq 3$.

2. Die i-te Operation ist ein `push` mit Arrayüberlauf.

   In diesem Fall ist $c_i = |A_{i-1}| + 1 = |S_{i-1}| + 1$ sowie $|A_i| = 2 \cdot (|S_i| - 1)$. Für die Potentialdifferenz erhalten wir:

   $$\Delta\Phi(S_{i-1}, A_{i-1}, S_i, A_i) \;\; = \;\; | \, 2 \cdot |S_i| - |A_i| \, | \; - \; | \, 2 \cdot |S_{i-1}| - |A_{i-1}| \, | \;\; = \;\; 2 - |S_{i-1}|.$$

   Es gilt daher $\hat{c}_i = |S_{i-1}| + 1 + 2 - |S_{i-1}| = 3$.

3. Die i-te Operation ist ein `pop`, welches zu keiner Arrayverkleinerung führt.

   In diesem Fall ist wiederum $c_i = 1$. Wie im ersten Fall müssen wir hier wieder zwischen den Fällen unterscheiden, dass die Operation sich auf die obere Hälfte oder das zweite Viertel von unten auswirkt. Wir erhalten für den ersten Fall, d.h. $2 \cdot |S_i| \geq |A_i|$:

   $$\begin{aligned} \Delta\Phi(S_{i-1}, A_{i-1}, S_i, A_i) &= | \, 2 \cdot |S_i| - |A_i| \, | \; - \; | \, 2 \cdot |S_{i-1}| - |A_{i-1}| \, | \\ &= 2 \cdot |S_i| - |A_{i-1}| - 2 \cdot (|S_i| + 1) + |A_{i-1}| \;\; = \;\; -2 \end{aligned}$$

   und für den zweiten Fall, d.h. $2 \cdot |S_i| < |A_i|$:

   $$\begin{aligned} \Delta\Phi(S_{i-1}, A_{i-1}, S_i, A_i) &= | \, 2 \cdot |S_i| - |A_i| \, | \; - \; | \, 2 \cdot |S_{i-1}| - |A_{i-1}| \, | \\ &= |A_{i-1}| - 2 \cdot |S_i| - |A_{i-1}| + 2 \cdot (|S_i| + 1) \;\; = \;\; 2 \; . \end{aligned}$$

   Daher gilt wiederum $\hat{c}_i = 1 \mp 2 \leq 3$.

4. Die i-te Operation ist ein `pop` und führt zu keiner Arrayverkleinerung.

   In diesem Fall ist $c_i = |S_{i-1}| = \frac{|A_{i-1}|}{4} + 1$ sowie $|A_i| = 2|S_i|$. Für die Potentialdifferenz erhalten wir:

   $$\Delta\Phi(S_{i-1}, A_{i-1}, S_i, A_i) \;\; = \;\; | \, 2 \cdot |S_i| - |A_i| \, | \; - \; | \, 2 \cdot |S_{i-1}| - |A_{i-1}| \, | \;\; = \;\; 4 - 2 \cdot |S_{i-1}|.$$

   Es gilt daher $\hat{c}_i = 4 - |S_{i-1}|$. Da der Stack vor dieser Operation nicht leer ist, ist folglich $\hat{c}_i \leq 3$.

Für die Summe der $\hat{c}_i$ erhalten wir somit unabhängig von der konkreten Befehlsfolge:

$$\sum_{i=1}^{n} \hat{c}_i \leq 3 \cdot n \quad \text{und somit} \quad \sum_{i=1}^{n} c_i \leq 3 \cdot n + 1 \ .$$

Als obere Schranke für die amortisierte Zeit erhalten wir $T_{\text{amort}}(n) \leq 3$.

3 Das statische Wörterbuch

Unter einem statischen Wörterbuch (static dictionary) verstehen wir eine Menge $S \subseteq U$ von Elementen eines Universums U , auf welchen die Operation „ x ist ein Element von S “ existiert. Diese Operation werden wir im Folgenden als *membership*-Operation bezeichnen. Wie der Name schon zu erkennen gibt, ist S fest, und kann nicht verändert werden. Zur Vereinfachung sei $n := |S|$ und $m := |U|$.

Im Folgenden wollen wir Verfahren vorstellen, mit deren Hilfe statische Wörterbücher implementiert werden können.

3.1 Sortierte Listen

Liegt die Menge $S \subseteq U$ in Form einer sortierten Liste vor, so kann mit Hilfe der binären Suche in dieser Liste die Frage $x \in S$ in $\lceil \log_2(n+1) \rceil$ Schritten entschieden werden. Steht uns auf der anderen Seite nur ein Comparison Tree zur Verfügung, so zeigt uns eine einfache informationstheoretische Überlegung:

Beobachtung 1 *Jedes Comparison Tree Verfahren zur Lösung des* membership-*Problems benötigt* $\Omega(\log n)$ *Schritte.*

Beweis: Wir wählen $S := \{1, 3, \ldots, 2 \cdot n - 1\}$ und $U := \{0, 1, 2, \ldots, 2 \cdot n\}$. So benötigt ein binärer Suchbaum, welcher das *membership*-Problem entscheidet, $2 \cdot n$ Blätter. Sollte dieses nicht der Fall sein, so existieren zwei Elemente $y < y'$ in U , auf deren Eingabe der Comparison Tree Algorithmus im gleichen Blatt des Suchbaumes endet. Somit endet dieser Algorithmus auch für alle Eingaben z mit $y \leq z \leq y'$ und daher auch $y + 1$ in diesem Blatt. Der Comparison Trees Algorithmus generiert folglich auf y und $y + 1$ die gleiche Antwort. Da aber nur einer dieser Werte in S ist, löst dieser Algorithmus nicht das *membership*-Problem.

Beachten wir nun, dass ein Binärbaum der Tiefe k höchstens 2^k Blätter hat, so folgt die Behauptung unmittelbar. ∎

Diese untere Schranke wollen wir nun auf Verfahren verallgemeinern, welche auf dem Vergleich der Eingabe mit indizierten Elementen aus S beruhen.

3.1.1 Das Cell-Probe Modell

Das nachfolgende Ergebnis stammt von A. Yao [Yao81]:

Theorem 1 *Sei S eine n-elementige Teilmenge von U, welche in einer sortierten Tabelle der Länge n gespeichert ist. Ist $m = |U| \geq 2n-1$, dann benötigt ein Vergleichsverfahren zumindest $\lceil \log_2(n+1) \rceil$ Proben von T, um das n-t kleinste Element aus U zu finden.*

Beweis: Das Theorem soll mit Hilfe eine Adversary-Arguments bewiesen werden. Nehmen wir also an, dass ein Algorithmus existiert, welcher auf jede Eingabe weniger als $\lceil \log_2(n+1) \rceil$ Schritte benötigt, um das n-t kleinste Element aus U in S zu finden. Mit anderen Worten, sei $U := \{x_0, \ldots, x_{m-1}\}$ mit $x_0 < x_1 < \ldots < x_{m-1}$, dann soll der Algorithmus $x_{n-1} \in S$ entscheiden. Hierzu konstruiert der Adversary S, so dass der Algorithmus die falsche Antwort ausgibt. Der Beweis – und damit auch die Konstruktion von S – erfolgt durch Induktion über n.

Für der Basisfall, $n = 2$, müssen wir zeigen, dass jeder Algorithmus zumindest $\lceil \log_2(n+1) \rceil = 2$ Proben aus S ansehen muss. Für $n = 2$ sind daher drei Tabellen möglich: $[x_0, x_1]$, $[x_0, x_2]$ und $[x_1, x_2]$. Der Algorithmus hat somit die Möglichkeit entweder nur in der Tabellenposition $T[0]$ oder $T[1]$ anzusehen. Im ersten Fall wählen wir $T[0] = x_0$, S ist somit $\{x_0, x_1\}$ oder $\{x_0, x_2\}$. Entscheidet sich unser Verfahren nun für $x_1 \in S$, so wählen wir $S = \{x_0, x_2\}$ und unser Verfahren generiert eine falsche Antwort. Antwortet unser Algorithmus jedoch mit $x_1 \notin S$, so wählen wir $S = \{x_0, x_1\}$. Der Fall, dass der Algorithmus $T[1]$ testet, verläuft analog.

Nehmen wir nun im Folgenden an, dass die Aussage von Theorem 1 für alle $\ell < n$ gültig ist. Es soll nun gezeigt werden, dass dann diese Aussage auch für alle Arrays der Größe n gültig ist.

Sei $T[p-1]$ die erste Abfrage unseres Verfahrens, dann unterscheiden wir die folgenden Fälle:

1. $p \leq n/2$: In diesem Fall wählen wir $\{x_0, \ldots, x_{p-1}\} \subset S$ und somit $T[i] := x_i$ für alle $i \leq p-1$. Die ersten p Felder des Arrays sind somit festgelegt. Für die verbleibenden $n' := n - p \geq n/2$ Elemente in $S' := S \setminus \{x_0, \ldots, x_{p-1}\} \subset U' := U \setminus \{x_0, \ldots, x_{p-1}\}$ gilt $|U'| \geq 2n - 1 - p > 2(n-p) - 1 = 2n' - 1$ und das n'-t kleinste Element in U' ist x_{n-1}. Wir haben somit das Problem, das Element x_{n-1} in S zu finden, auf das Problem reduziert, x_{n-1} in S' zu finden. Aus der Induktionshypothese geht jedoch hervor, dass dieses mindestens $\lceil \log_2(n'+1) \rceil \geq \lceil \log_2(n/2-1) \rceil \geq \lceil \log_2(n-1) \rceil - 1$ Tests benötigt. Zuzüglich des ersten Schritts benötigt das Verfahren also zumindest $\lceil \log_2(n'+1) \rceil$ Proben.

2. $p > n/2$: In diesem Fall wählen wir $\{x_{m-(n-p)-1}, \ldots, x_{m-1}\} \subset S$ und somit $T[p-1] := x_{m-(n-p)-1}, \ldots,$ $T[n-1] = x_{m-1}$. Sei $n' := p - 1 \geq n/2$, $S' := S \setminus \{x_{m-(n-p)-1}, \ldots, x_{m-1}\}$ und
$$U' := U \setminus (\{x_0, \ldots, x_{n-p} \cup \{x_{m-(n-p)-1}, \ldots, x_{m-1}\}) ,$$
dann ist $|S'| := p-1$, $|U'| \geq 2n-1-2(n-p+1) = 2(p-1)-1$ und x_{n-1} das $p-1$-t kleinste Element in U'. Nach der Induktionshypothese benötigt jedes Verfahren jedoch $\lceil \log_2(n'+1) \rceil = \lceil \log_2 p \rceil$ Tests, um $x_n \in S'$ korrekt zu entscheiden. Aus der Bedingung $p > n/2$ folgt jedoch, dass $\lceil \log_2 p \rceil \geq \lceil \log_2(n+1) \rceil - 1$ (beachte, dass entweder n eine Potenz von 2 ist und daher $\lceil \log_2 p \rceil = \lceil \log_2 n \rceil$ oder n keine Potenz von 2 ist und daher $\lceil \log_2 n \rceil = \lceil \log_2(n+1) \rceil$). Die Behauptung ergibt sich unmittelbar.

∎

Wählen wir an Stelle der *sortierten* Tabellen, Tabellen, in denen S bezüglich einer festen Permutation gespeichert ist, so können wir obige Aussage auf derartige Tabellen verallgemeinern.

Können wir zudem Vergleiche des Positionsindex zur Hilfe nehmen, so ist eine Wörterbuchabfrage oft effizienter möglich. Betrachten wir beispielsweise eine *zyklische Repräsentation* einer Menge $S \subset U$

mit $|U| = |S| + 1$ und sei $\{x_p\} := U \setminus S$, dann speichern wir S in der Tabelle wie folgt

$$T[i] := x_{p+i+1} \quad \text{für} \quad 0 \leq i \leq |U| - p - 2 \quad \text{und} \quad T[|U| - p + i] := x_i \quad \text{für} \quad 0 \leq i < p \; .$$

Ein Beispiel hierfür ist in Abbildung 5 zu finden.

x_5	x_6	x_7	x_8	x_9	x_0	x_1	x_2	x_3

Abbildung 5: Zyklische Repräsentation eines Wörterbuchs.

Um das *membership*-Problem zu lösen, genügt nun eine Anfrage, da genau dieses Element aus U nicht in S ist, welches dem Element $T[0]$ vorausgeht.

Datenstrukturen, welche über die Position, an der ein Element gespeichert ist, Informationen über die gesamte Menge enthalten, nennen wir *implizite Datenstrukturen.*

3.1.2 Implizite Datenstrukturen

In einer impliziten Datenstruktur speichern wir eine Menge S in einer Tabelle der Größe $|S|$, jedes Element aus S an einer eigenen Position. Jede Menge S wird nach einer bestimmten festen Permutation π_S gespeichert. Eine zyklische Repräsentation eines Wörterbuchs ist daher eine implizite Datenstruktur.

Es soll nun gezeigt werden, dass implizite Datenstrukturen immer helfen können, einen effizienten Zugriff zu gewährleisten. Hierzu werden wir den folgenden Satz aus der Ramsey-Theorie anwenden (siehe [GrRS90]).

Theorem 2 (Ramsey's Theorem) *Für alle $k, r, t \in \mathbb{N}$ existiert ein $R(k, r, t) \in \mathbb{N}$, so dass für alle $m > R(k, r, t)$ und alle Funktionen $f : \{0, \ldots, m-1\}^r \to \{0, \ldots, t-1\}$ die folgende Aussage gültig ist: Es existiert eine Menge $M \subseteq \{0, \ldots, m-1\}$ der Größe k, so dass für alle Teilmengen $M' \subseteq M$ aus M der Wert von $f(M')$ der gleiche ist.*

Versuchen wir uns diesen Satz an folgendem Beispiel mit $k = 3$ und $r = t = 2$ zu illustrieren: Betrachten wir eine Menge von 6 Menschen. Für jedes Paar in dieser Menge gilt, dass sich die beiden Personen entweder kennen oder sich fremd sind. Da $R(3, 2, 2) \leq 6$ ist, gibt es in einer Gruppe von 6 Personen eine Dreiergruppe, so dass in dieser Gruppe jeder jeden kennt, oder keiner keinen kennt. Dieses Beispiel ist in Abbildung 6 illustriert.

r stellt also die Gruppengröße dar, zwischen deren Mitgliedern wir eine Beziehung f kennen. Hierbei gibt es t verschiedene Beziehungsformen. k ist hingegen die Größe einer Zielgruppe, deren Untergruppen der Größe r jeweils die gleiche Beziehungsform zueinander haben.

Mit Hilfe von Ramsey's Theorem können wir nun den folgenden Satz beweisen:

Theorem 3 *Jede implizite Datenstruktur zum Speichern einer Menge S der Größe n aus einem Universum U der Größe m hat für ein hinreichend großes m eine worst-case Zeitkomplexität von $\lceil \log_2(n+1) \rceil$ zur Beantwortung von* membership*-Frage.*

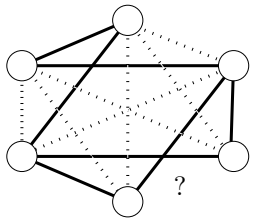


Abbildung 6: Ein Beispiel für Ramsey's Theorem. Die Kante, welche durch ein Fragezeichen ersetzt ist, kann weder durchgezogen noch gestrichelt sein, ohne dass ein durchgezogenes oder gestricheltes Dreieck entsteht.

Beweis: Sei $m \geq R(2n-1, n, n!)$. Die Parameter der Ramseyfunktion R ergeben sich wie folgt: die Mengen S, die wir in einem Wörterbuch speichern wollen, haben eine Größe von n und die Anzahl der Permutationen von einer n-elementigen Menge ist $n!$. Sollte uns die vorhandene Permutation keine Erkenntnisse über S geben, so liegt keine sinnvolle implizite Datenstruktur vor. Nach Theorem 1 benötigen wir aber dann $\lceil \log_2(n+1) \rceil$ Schritte, um eine *membership*-Frage zu beantworten, wenn dass Universum $2n-1$ Elemente umfasst.

In einer festen impliziten Datenstruktur wird jede Menge S nach einer festen Permutation π_S in der Tabelle T gespeichert. Wir wählen daher die Funktion f so, dass $f(S) := \pi_S$. Nach Ramsey's Theorem existiert nun bei einem Universum der Größe m eine Menge von $2n-1$ Elementen, dessen Teilmengen der Größe n alle nach der gleichen Permutation in T gespeichert werden. Aus Theorem 1 wissen wir somit, dass wir zumindest $\lceil \log_2(n+1) \rceil$ Proben aus T ansehen müssen, um eine *membership*-Frage im worst-case beantworten zu können. ∎

3.2 Hashing

Über Hashingverfahren kann oft effizienter auf Einträge eines Wörterbuches zugegriffen werden als über sortierte Listen. Ein einfaches mit dem Hashing verwandtes Verfahren stellt hierbei der *charakteristische Vektor* dar, der wie folgt definiert ist:

$$T[i] := \begin{cases} 1, & \text{wenn } x_i \in S \\ 0, & \text{ansonsten.} \end{cases}$$

Die *membership*-Frage kann auf einer derartigen Datenstruktur in einem Schritt entschieden werden. Ein Nachteil dieser Datenstruktur ist jedoch die Größe der Tabelle auch bei kleinen Mengen S.

Ein Hashverfahren wird über eine Funktion, der *Hashfunktion* $h : U \to \{0, \ldots, r-1\}$ angegeben. Ein Element $x \in S$ wird danach an der Tabellenposition $h(x)$ abgelegt. Man beachte, dass Hashfunktionen zunächst nicht ausschließen, dass es zu Kollisionen kommt, d.h. es ist möglich, dass es zwei Elemente $x \neq y$ in S gibt, für welche $h(x) = h(y)$ ist. Um eine *membership*-Frage zu beantworten, müssen wir jetzt nur testen, ob ein x an der Position $h(x)$ abgespeichert ist. Um eine *membership*-Frage einfach beantworten zu können, müssen wir nun garantieren, dass die Hashfunktion Konflikte ausschließt. Sie muss also für unser S eindeutig sein. Wir nennen daher eine Hashfunktion **perfekt** für S, wenn für alle Paare $x, y \in S$ mit $x \neq y$ gilt: $h(x) \neq h(y)$.

Ein Beispiel für eine Hashfunktion ist $h(x) := 2x \bmod 5$. Ist nun $S = \{2, 9\} \subset \{0, \ldots, 25\} =: U$, so erhalten wir $h(2) = 4$ und $h(9) = 3$. Wir speichern folglich $T[4] := 2$ und $T[3] := 9$.

3.2.1 Perfekte Hashfunktionen

Beispiele für perfekte Hashfunktionen sind:

1. die identische Funktion: $h : U \to U$ mit $h(x) = x$ für alle $x \in U$. Diese Funktion ist eine perfekte Hashfunktion für alle $S \subseteq U$, jedoch benötigt sie wie der charakteristische Vektor sehr große Hashtabellen.

2. Funktionen aus einer *universellen Familie von Hashfunktionen*. Eine Familie $\mathcal{H}$ von Funktionen $h : U \to \{0, \ldots, r-1\}$ nennen wir eine **universellen Familie von Hashfunktionen**, wenn für alle $x, y \in U$ mit $x \neq y$ gilt:

$$\text{Prob}_{h \in \mathcal{H}}[h(x) = h(y)] \quad = \quad \frac{|\{h \in \mathcal{H} | h(x) = h(y)\}|}{|\mathcal{H}|} \quad \in \quad O\left(\frac{1}{r}\right) .$$

Die Beobachtung, dass jede universelle Familie von Hashfunktionen eine perfekte Hashfunktion für jedes $S \subset U$ mit $|S| = n$ umfasst, stammt von Carter und Wegman [CaWe79].

Theorem 4 *Sei $\mathcal{H}$ eine universelle Familie von Funktionen $h : U \to \{0, \ldots, r-1\}$ für ein hinreichend großes r. Dann existiert für jede Menge $S \subset U$ der Größe n eine perfekte Hashfunktion $h \in \mathcal{H}$ für S in $\mathcal{H}$.*

Beweis: Für ein festes $S \subset U$ mit $|S| = n$ sei $C(h)$ die Anzahl der Kollisionen für $h \in \mathcal{H}$, d.h.

$$C(h) \quad := \quad | \{ (x, y) \mid x, y \in S,\ x < y,\ h(x) = h(y) \} | .$$

Ist h eine perfekte Hashfunktion, dann ist $C(h) = 0$. Es gilt:

$$C(h) \quad := \sum_{x<y,\ \ x,y \in S} C_{xy}(h) \quad \text{wobei} \quad C_{xy}(h) \quad := \quad \begin{cases} 1 & \text{für } h(x) = h(y) \\ 0 & \text{ansonsten.} \end{cases}$$

Aus der Definition einer universellen Familie folgt nun für den Erwartungswert $E[C]$ für eine entsprechend gewählte Konstante c (implizit durch die O-Notation gegeben):

$$\begin{aligned} E[C] \quad &= \quad \sum_{x<y,\ \ x,y \in S} E[C_{xy}] \quad = \quad \sum_{x<y,\ \ x,y \in S} \text{Prob}_{h \in \mathcal{H}}[C_{xy}(h)] \\ &= \quad \sum_{x<y,\ \ x,y \in S} \text{Prob}_{h \in \mathcal{H}}[h(x) = h(y)] \\ &\leq \quad \sum_{x<y,\ \ x,y \in S} \frac{c}{r} \quad = \quad \binom{n}{2} \frac{c}{r} . \end{aligned}$$

Wählen wir $r \geq c \cdot \binom{n}{2}$, dann ist $E[C] < 1$. Da $C(h)$ nicht negativ sein kann, existiert eine Funktion h mit $C(h) < 1$. Die Anzahl der Kollisionen ist jedoch immer ganzzahlig und daher für ein $h \in \mathcal{H}$ gleich Null. Diese Funktion ist folglich eine perfekte Hashfunktion für S. ∎

Wählen wir nun $r \geq 2c \cdot \binom{n}{2}$, so gilt $E(C) < \frac{1}{2}$. Somit ist bei dieser Wahl von r zumindest die Hälfte aller Funktionen in $\mathcal{H}$ eine perfekte Hashfunktion für S. Können wir nun Funktionen uniform zufällig aus $\mathcal{H}$ wählen, so bestimmt der folgende Algorithmus eine perfekte Hashfunktion für eine Menge S:

1. Wähle eine zufällige Funktion $h \in \mathcal{H}$.
2. Teste, ob h allen Elementen $x \in S$ eine unterschiedliche Tabellenposition zuweist.
3. Weist h zwei Werten $x, y \in S$ mit $x \neq y$ die gleiche Position zu, beginne von vorne.

Die erwartete Anzahl von Versuchen bei $r \geq 2c \cdot \binom{n}{2}$ ist kleiner 2, daher ist die erwartete Laufzeit dieses Algorithmus $O(n)$, sofern die Funktion in linearer Zeit aus $\mathcal{H}$ gewählt werden kann, und zudem die Berechnung von $h(x)$ in konstanter Zeit erfolgt.

3.2.2 Universelle Familien von Hashfunktionen

Beispiele für universelle Familien sind:

1. Ein erstes Beispiel für eine universelle Familie stellt die Menge aller Funktionen

   $$f : U \to \{0, \ldots, r-1\}$$

   dar. So ist einfach einzusehen, dass für alle $x \in U$ und $z \in \{0, \ldots, r-1\}$ gilt

   $$\frac{|\{h | h : U \to \{0, \ldots, r-1\} \text{ und } h(x) = z\}|}{|\{h | h : U \to \{0, \ldots, r-1\}\}|} \quad = \quad \frac{1}{r} .$$

   Folglich ist die Wahrscheinlichkeit, dass eine zufällig gewählte Funktion zwei Elemente $x, y \in U$ auf den gleichen Wert abbildet, auch gleich $\frac{1}{r}$.

2. Ist die Größe m des Universums eine Primzahl, und teilt r die Zahl $m-1$, so ist

   $$\mathcal{H}_r \quad := \quad \{ \, (ax \bmod p) \bmod r \mid a \in \mathbb{Z}_p \setminus \{0\} \, \}$$

   eine universelle Familie (siehe hierzu [FrKS84]).

3. Eine verwandte universelle Familie (siehe hierzu [CoLR90], Seite 231) erhalten wir durch die Unterteilung der Binärdarstellung einer Eingabe x in Blöcke $x_0, \ldots, x_k$ der Länge $\lfloor \log_2 r \rfloor$. Für eine Primzahl r erhalten wir dann eine universelle Familie

   $$\mathcal{H} \quad := \quad \{ \, (\sum_{i=1}^{m} a_i x_i) \bmod r \mid a_0, \ldots, a_r \in \{0, \ldots, r-1\} \, \}.$$

   Man beachte, dass $|\mathcal{H}| = r^{m+1}$.

4. Die letzte universelle Familie $\mathcal{H} \quad := \quad \{ \, h_a \mid 0 < a < 2^k \text{ für ungerades } a \, \}$ und

   $$h_a \; : \; \{0, \ldots, 2^k - 1\} \leftarrow \{0, \ldots, 2^b\} \quad \text{mit} \quad h_a(x) \quad := \quad (ax \bmod 2^k) \operatorname{div} 2^{k-b} .$$

   Für diese universelle Familie gilt $|U| = 2^k$ und $|T| = 2^b$. Man beachte, dass die Funktionen h_a genau b Bits aus der Binärdarstellung von ax. Die Operationen mod und div lassen sich als einfache Shift-Operationen implementieren. Daher ist die Berechnung einer Funktion aus dieser Menge einfacher als die Berechnung einer Funktion aus dem zweiten Beispiel. Letzte universelle Familie wurde in [DHKP97] ausgiebig untersucht und im Folgenden wollen wir zeigen, dass es sich hierbei tatsächlich um eine universelle Familie handelt.

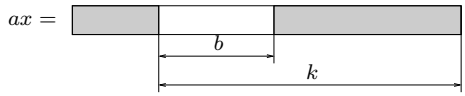


Abbildung 7: Wahl des Blocks in der Binärdarstellung von ax, der den Wert von $h_a(x)$ angibt.

Theorem 5 *Für alle Paare* $x, y \in U$ *mit* $x \neq y$ *gilt*

$$Prob_{h_a \in \mathcal{H}}[h(x) = h(y)] \;\leq\; 2^{1-m} \;.$$

Beweis: Die für die Analyse einer Funktion h_a wichtigen Blöcke der Binärdarstellung von ax sind in Abbildung 7 illustriert. Sei $x, y \in U = \{0, \ldots, 2^k - 1\}$ mit $x > y$, dann gilt

$$ax \text{ div } 2^k \;\geq\; ay \text{ div } 2^k$$

und für $h_a(x) = h_a(y)$

$$| \, ax \bmod 2^k \;-\; ay \bmod 2^k \, | \;=\; \left\{ \begin{array}{ll} a(x-y) \bmod 2^k & \text{für } ax \bmod 2^k > ay \bmod 2^k \\ a(y-x) \bmod 2^k & \text{für } ax \bmod 2^k < ay \bmod 2^k \end{array} \right\} \;<\; 2^{k-b} \;. \quad (1)$$

Man beachte, dass bei der Umformung des linken Ausdrucks in Gleichung 1 die höherwertigen Bitstellen in $a(x-y) \bmod 2^k$ von $a(x-y)$ nicht ins Gewicht fallen. Aus $0 \leq x, y < 2^k$ folgt zudem

$$(x-y) \bmod 2^k \;\neq\; 0 \quad \text{und daher für ungerade } a \quad a(x-y) \bmod 2^k \;\neq\; 0 \;.$$

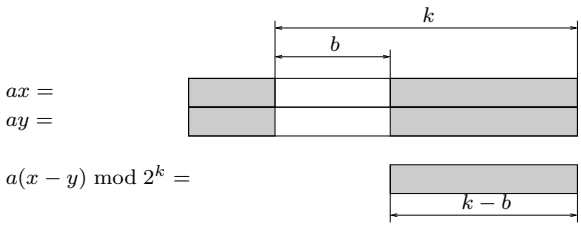


Abbildung 8: Blöcke in der Binärdarstellung von $a(x-y) \bmod 2^k$.

Daher gilt:

$$\begin{array}{rcl} \text{Prob}_{h_a \in \mathcal{H}}[h(x) = h(y)] & \leq & \text{Prob}_{a \in \{0,\ldots,2^k-1\},\ a \text{ ungerade}}[1 \leq a(x-y) \bmod 2^k \leq 2^{k-b} - 1] \\ & + & \text{Prob}_{a \in \{0,\ldots,2^k-1\},\ a \text{ ungerade}}[1 \leq a(y-x) \bmod 2^k \leq 2^{k-b} - 1] \;. \end{array}$$

Die beiden Wahrscheinlichkeiten ergeben sich für die beiden symmetrischen Fälle in Gleichung 1. Im Folgenden wollen wir uns auf den ersten Fall beschränken und zeigen, dass

$$\text{Prob}_{a \in \{0,\ldots,2^k-1\},\ a \text{ ungerade}}[1 \leq a(x-y) \bmod 2^k \leq 2^{k-b} - 1] \;\leq\; \frac{1}{2^b} \;.$$

Wir beginnen unsere Analyse mit der Betrachtung des Sonderfalls, dass $x - y$ ungerade ist. Da $x - y$ ungerade ist, ist der größte gemeinsame Teiler von $x - y$ und 2^k gleich 1, daher existieren Zahlen

$u, v \in \mathbb{N}$ mit $u(x-y) = v2^k + 1$ (siehe beispielsweise [Rose93], S. 75). Sei $w := u \bmod 2^k$, so gilt $w(x-y) \bmod 2^k = 1$. Da a ungerade ist, gilt ferner, dass auch $a(x-y)$ und $a(x-y) \bmod 2^k$ ungerade sind.

Aus $1 \leq a(x-y) \bmod 2^k \leq 2^{k-b} - 1$ folgt:

$$a(x-y) \bmod 2^k \in \{1, 3, \ldots, 2^{k-b} - 1\}$$

und daher

$$\begin{aligned} a &= [\underbrace{(w(x-y) \bmod 2^k)}_{=1} \cdot \underbrace{(a \bmod 2^k)}_{=a}] \bmod 2^k = wa(x-y) \bmod 2^k \\ &\in \{w \bmod 2^k,\ 3w \bmod 2^k,\ \ldots,\ (2^{k-b}-1)w \bmod 2^k\} . \end{aligned}$$

Da a jedoch uniform aus $\{1, 3, \ldots, 2^k - 1\}$ gezogen werden konnte, folgt

$$\begin{aligned} &\mathrm{Prob}_{a \in \{0,\ldots,2^k-1\},\ a \text{ ungerade}}[1 \leq a(x-y) \bmod 2^k \leq 2^{k-b} - 1] \\ &\quad \leq \frac{|\{w \bmod 2^k,\ 3w \bmod 2^k,\ \ldots,\ (2^{k-b}-1)w \bmod 2^k\}|}{|\{1, 3, \ldots, 2^k - 1\}|} \leq \frac{2^{k-b-1}}{2^{k-1}} = \frac{1}{2^b} . \end{aligned}$$

Im Allgemeinen gibt es jedoch $s, z \in \mathbb{N}$, so dass $x - y = z \cdot 2^s$ für ein ungerades z ist. Für s gilt $0 \leq s < k$. Die Blöcke in der Binärdarstellung von $a(x-y)$ sind in Abbildung 9 illustriert. Hierbei

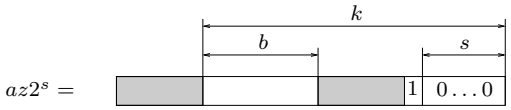


Abbildung 9: Blöcke in der Binärdarstellung von $a(x-y) = az2^s$.

müssen wir zwei Fälle unterscheiden:

1. $s \geq k - b$: Da z ungerade ist, gilt

$$az2^s \bmod 2^k = \begin{cases} (az \bmod 2^{k-s}) \cdot 2^s \geq 2^s > 2^{k-b} - 1 & \text{für } s < k \\ 0 & \text{für } s \geq k . \end{cases}$$

   Somit gilt $\mathrm{Prob}_{a \in \{0,\ldots,2^k-1\},\ a \text{ ungerade}}[1 \leq a(x-y) \bmod 2^k \leq 2^{k-b} - 1] = 0$.

2. $s < k - b$: Dieser Fall stellt eine Verallgemeinerung des einfachen Falls mit ungeradem $x - y$ dar, nur dass wir uns in diesem Fall auf z konzentrieren. Wie im obigen Fall können wir auf die Existenz zweier Zahlen $u, v \in \mathbb{N}$ folgern, so dass $u \cdot z = v2^{k-s} + 1$. Sei $w := u \bmod 2^{k-s}$, dann ist $wz \bmod 2^{k-s} = 1$. Da z ungerade ist, gilt jetzt wiederum:

$$az2^s \bmod 2^k \in \{2^s,\ 3 \cdot 2^s\ , \ldots, (2^{k-m-s} - 1) \cdot 2^s\}$$

   und daher

$$\begin{aligned} a2^s \bmod 2^k &= awz2^s \bmod 2^k \\ &\in \{(w2^s) \bmod 2^k,\ (3w \cdot 2^s) \bmod 2^k\ , \ldots,\ ((2^{k-m-s} - 1)w \cdot 2^s) \bmod 2^k\} . \end{aligned}$$

   Eine einfache Umformung führt somit zu

$$a \bmod 2^{k-s} \in \{w \bmod 2^{k-s},\ 3w \bmod 2^{k-s}\ , \ldots,\ (2^{k-m-s} - 1)w \bmod 2^{k-s}\} .$$

Da a uniform aus $\{1,3,\ldots,2^k-1\}$ gewählt werden konnte, wählen wir $a \bmod 2^{k-s}$ uniform aus $\{1,3,\ldots,2^{k-s}-1\}$. Somit ergibt sich über die Wahlmöglichkeiten von $a \bmod 2^{k-s}$:

$$\text{Prob}_{a\in\{0,\ldots,2^k-1\},\ a \text{ ungerade}}[1 \le a(x-y) \bmod 2^k \le 2^{k-b}-1] \quad \le \quad \frac{2^{k-s-m-1}}{2^{k-s-1}} \quad = \quad \frac{1}{2^m} \ .$$

∎

3.2.3 Deterministische Konstruktion guter Hashfunktionen

Im Folgenden wollen wir ein Polynomialzeit-Verfahren vorstellen, welches eine Hashfunktionen h_a beziehungsweise ein dazugehöriges a findet, welche(s) die am Ende des Beweises von Theorem 4 gezeigte Güte für die erwartete Anzahl $E_{h_a\in\mathcal{H}}[C(h)] \le \binom{n}{2}2^{1-b}$ von Kollisionen erreicht. In anderen Worten, für das zu konstruierende a soll gelten:

$$C((ax \bmod 2^k) \text{ div } 2^{k-b}) \quad \le \quad \frac{\binom{n}{2}}{2^{b-1}} \ .$$

Der nachfolgende Satz stammt aus [Ram96a]:

Theorem 6 *Gegeben sei eine Menge $S \subseteq \{0,\ldots,2^k-1\}$ der Größe n. Dann kann eine Funktion $h_a \in \mathcal{H}$ deterministisch in Zeit $O(k\cdot n^2)$ konstruiert werden, so dass $C(h_a) \le \binom{n}{2}\cdot 2^{1-b}$.*

Beweis: Die Konstruktion von $a = a_{k-1}a_{k-2}\ldots a_1a_0$ erfolgt von der niederwertigsten Stelle ausgehend zur höchstwertigsten Stelle. Da a ungerade ist, gilt $a_0 = 1$. Sei nun $\alpha = a'_{i-1}\ldots a'_0$ ein Suffix der Länge i von a, so dass

$$E_{a\in\{0,\ldots,2^k-1\}}[\ C(h_a) \mid \alpha \text{ ist eine Suffix von } a] \quad \le \quad \frac{\binom{n}{2}}{2^{b-1}} \ .$$

Da

$$\begin{aligned} E_a[\ C(h_a) \mid \alpha \text{ ist eine Suffix von } a] \quad = \quad & \frac{1}{2}E_a[\ C(h_a) \mid 0\alpha \text{ ist eine Suffix von } a] \\ + \quad & \frac{1}{2}E_a[\ C(h_a) \mid 1\alpha \text{ ist eine Suffix von } a] \end{aligned}$$

gilt entweder

$$E_{a\in\{0,\ldots,2^k-1\}}[\ C(h_a) \mid 0\alpha \text{ ist eine Suffix von } a] \quad \le \quad \frac{\binom{n}{2}}{2^{b-1}}$$

oder

$$E_{a\in\{0,\ldots,2^k-1\}}[\ C(h_a) \mid 1\alpha \text{ ist eine Suffix von } a] \quad \le \quad \frac{\binom{n}{2}}{2^{b-1}} \ .$$

Es genügt daher $E_{a\in\{0,\ldots,2^k-1\}}[\ C(h_a) \mid 0\alpha \text{ ist eine Suffix von } a]$ zu bestimmen, um den nächsten Wert a'_i von a zu bestimmen. Ist dieser Wert kleiner $\binom{n}{2}2^{1-b}$ so wählen wir $a'_i = 0$ und $a'_i = 1$ ansonsten. Nach k Iterationen haben wir somit ein entsprechendes a bestimmt.

Ein Problem stellt jedoch die Bestimmung der einzelnen Werte von $E_a[\ C(h_a) \mid \alpha \text{ ist eine Suffix von } a]$ dar. Sei nun

$$\delta_{xy}(\alpha) \quad := \quad \text{Prob}_a[h_a(x) = h_a(y) | \alpha \text{ ist eine Suffix von } a] \ ,$$

dann gilt

$$E_a[\ C(h_a) \mid \alpha \text{ ist eine Suffix von } a] \quad = \quad \sum_{x,y \in \{0,\ldots,2^k-1\},\ \ x \neq y} \delta_{xy}(\alpha)\ .$$

Der Beweis, dass $\delta_{xy}(\alpha)$ in konstanter Zeit berechnet werden kann, ist in [Ram96a, Ram96b] zu finden. Die hier zum Einsatz kommende Technik ist sehr mit der Technik verwandt, die wir im letzten Beweis angewendet haben. Daher soll hier nicht weiter darauf eingegangen werden. ∎

3.2.4 Konstruktion von perfekten Hashfunktionen für Tabellen linearer Größe

Im Beweis, dass jede universelle Familie eine perfekte Hashfunktion beinhaltet, gibt uns eine quadratische obere Schranke für die Größe der benutzten Hashtabelle. Dieses führt jedoch dazu, dass die Hashtabelle nur sehr spärlich gefüllt ist. In diesem Abschnitt soll nun ein Hashverfahren vorgestellt werden, welches mit einer Hashtabelle linearer Größe auskommt. Die folgende Konstruktion geht auf die Arbeit [FrKS84] zurück.

Betrachten wir zunächst noch einmal die Schranke, die wir im Beweis von Theorem 4: Für jede universelle Familie $\mathcal{H}$ von Funktionen $h : U \to \{0,\ldots,r-1\}$ und für jede Menge $S = \{0,\ldots,n-1\}$ gilt:

$$\exists h \in \mathcal{H}\ :\ \ E[C(h)] \quad \in \quad O\left(\ \binom{n}{2} \cdot \frac{1}{r}\ \right)\ .$$

Beschränken wir uns auf die Funktion h, deren Bildbereich linear in n ist, so bedeutet diese Schranke, dass die erwartete Anzahl von Kollisionen ebenfalls linear ist. Betrachten wir nun das Verfahren von Raman im letzten Unterkapitel, so sehen wir, dass eine solche Hashfunktion auch effizient konstruiert werden kann.

Wir wollen nun die Anzahl der Elemente aus S untersuchen, die aus einem Wert abgebildet werden. Sei also:

$$N_i(h) \quad := \quad \mid \{\ x \in S \mid h(x) = i\ \}\ .$$

Somit gilt $\sum_{i=0}^{n} N_i(h) = n$. Da zwei Elemente aus S nur dann auf eine Position über h abgebildet werden, wenn diese kollidieren, gilt:

$$C(h) \quad = \quad \sum_{i=0}^{r-1} \binom{N_i(h)}{2} \quad = \quad \frac{1}{2} \sum_{i=1}^{r-1} N_i(h)^2 \quad - \quad \frac{n}{2}\ .$$

Somit gilt

$$E_{h \in \mathcal{H}}[\textstyle\sum_{i=1}^{r-1} N_i(h)^2] \quad = \quad 2 \cdot E_{h \in \mathcal{H}}[C(h)] + n \quad \in \quad \Theta(n)\ .$$

Die Funktion h an sich ist im Allgemeinen keine perfekte Hashfunktion. Erlauben wir aber die Benutzung eines weiteren Hashschritts, so können wir ein perfektes Hashingverhalten erreichen. Man beachte hierzu, dass es für jeden Wert $i \in \{0,\ldots,r-1\}$ ausreicht, einen Bucket der Größe $N_i(h)^2$ zur Verfügung zu stellen, um die Elemente in $N_i(h)$ perfekt abzubilden (siehe Theorem 4). Hierzu benutzen wir für jeden Wert i eine entsprechend gewählte Funktion h_i (für jedes nicht leere Bucket i). Dieses ist in Abbildung 10 illustriert.

Der Platzbedarf dieses Verfahrens ist

$$r \quad + \quad \sum_{i=1}^{r-1} N_i(h)^2 \quad \in \quad \Theta(n)\ .$$

Der Test, ob ein $x \in U$ ein Element aus S ist, kann wie folgt erfolgen:

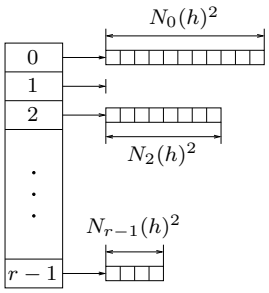


Abbildung 10: Ein Hashverfahren, welches eine Datenstruktur linearer Größe benötigt.

1. Bestimme $i = h(x)$;
2. bestimme $y = T_i[h_i(x)]$;
3. $x \in S$ genau dann, wenn $x = y$.

Zusammenfassend erhalten wir:

Theorem 7 *Es existiert eine Datenstruktur der Größe* $\Theta(n)$ *für Mengen der Größe* n *, auf der eine RAM, welche die Operationen* ADD, SUB, MULT *sowie* SHIFT *bzw.* DIV *benutzt, das* membership*-Problem in konstanter Zeit lösen kann.*

3.2.5 Hashing auf RAMs ohne *SHIFT* und *DIV*

Im Folgenden wollen wir zeigen, dass die Operationen *SHIFT* bzw. *DIV* für die Implementierung effizienter Hashingverfahren notwendig sind. Das nachfolgende Ergebnis ist in [FiMi95] zu finden.

Aber zuvor sollen noch einige hilfreiche Beobachtungen hergeleitet werden. Sei q ein Polynom vom Grade d , dessen Koeffizienten alle Intergerwerte sind, dann gilt:

Fakt 1 *Für alle* $y \in \mathbb{R}$ *gilt* $| \{ x \in \mathbb{R} \mid q(x) = y \} | \leq d$.

Fakt 2 *Für alle endlichen Mengen* $Y \subset \mathbb{R}$ *gilt* $| \{ x \in \mathbb{R} \mid q(x) \in Y \} | \leq d \cdot |Y|$.

Fakt 3 *Sei* $Z \subset \mathbb{Z}$ *, so dass für alle Paare* $z_1, z_2 \in Z$ *der Abstand zwischen* z_1 *und* z_2 *größer gleich* $d \cdot (s+1)$ *ist, dann gilt*

$$| \{ z \in Z \mid q(z) \in \{0, \ldots, s\} \} | \leq d .$$

Beweis: Der Beweis erfolgt über die Widerspruchsannahme, dass

$$| \{ z \in Z \mid q(z) \in \{0, \ldots, s \} | \geq d+1 .$$

Seien $z_0, z_1, \ldots, z_d \in Z \cap \{\, x \mid q(z) \in \{0, \ldots, s\} \,\}$ mit $z_0 < z_1 < \ldots < z_d$. Aus Fakt 2 und der Wahl der Abstände zwischen den Werten z_i folgt nun unmittelbar, dass für jedes Paar z_i, z_{i+1} ein $x_i \in \mathbb{Z}$ mit $z_i \leq x_i \leq z_{i+1}$ existiert, so dass

$$q(x_i) \quad \notin \quad \{0, \ldots, s\} \; .$$

ist. Man beachte, dass $q(x_i) \in \mathbb{Z}$, da q nur ganzzahlige Koeffizienten aufweist. Somit hat q zumindest $2 \cdot (2d + 2)$ Schnittpunkte mit den Geraden $g_o(x) = s + \frac{1}{2}$ und $g_u(x) = -\frac{1}{2}$ (wobei zumindest einer der Schnittpunkte zwischen $-\infty$ und z_0, jeweils zumindest zwei zwischen zwei Werten z_i, z_{i+1} und schließlich zumindest einer zwischen z_d und ∞ liegt. Dieses Verhalten ist in Abbildung 11 illustriert.

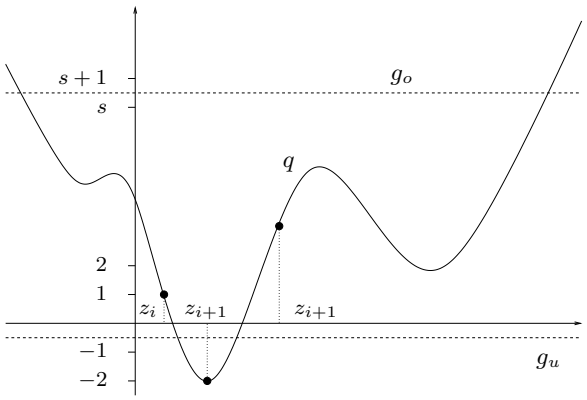


Abbildung 11: Der Kurvenverlauf eines ganzzahligen Polynoms.

Somit muss q zumindest eine der beiden Geraden g_o oder g_u $d+1$ mal schneiden. Dieses widerspricht jedoch Fakt 1. Es gibt also maximal d Elemente in Z, welche von q auf Elemente aus $\{0, \ldots, s\}$ abgebildet werden. ∎

Mit Hilfe der oben angegebenen Beobachtungen wollen wir nun folgendes Theorem zeigen:

Theorem 8 *Jedes statische Wörterbuch, welches eine n-elementige Teilmenge $S \subseteq U = \{0, \ldots, m-1\}$ mit Hilfe einer Datenstruktur bestehend aus $r < \frac{m}{n^2}$ Worten verwaltet, benötigt zur Beantwortung einer* membership*-Anfrage eine worst-case Zeit von $\Omega(\log \min\{n, \log(m/n)\})$ auf einer RAM, welche die Operationen* ADD, SUB *und* MULT *jedoch nicht* SHIFT *oder* DIV *benutzt.*

Beweis: Der Beweis erfolgt über ein Adversary-Argument. Wir wählen hierfür ein $U' \subseteq U$ und ein S und bestimmen eine Sequenz von Teilmengen aus

$$U_t \subseteq U_{t-1} \subseteq \; \ldots \; \subseteq U_2 \subseteq U_0 = U' \; ,$$

so dass sich eine RAM auf allen Elementen auf U_i in den ersten i Schritten gleich verhält. Der Adversary schränkt hierfür die Menge U' immer weiter ein. Beinhaltet U_t nun Elemente aus S sowie dem Kompliment von S, so muss die RAM auf einem der Elemente einen Fehler machen, da das gleiche Verhalten auch ein gleiches Akzeptanzverhalten einschließt. Auf die für diesen Beweis nötigen technischen Definitionen soll nun im Folgenden eingegangen werden.

Hierfür wollen wir uns auf ein Teiluniversum

$$U' \quad := \quad \{0, \; g, \; 2g, \; \ldots, \; (2n-1) \cdot g\}$$

sowie

$$S \;:=\; \{0,\ 2g,\ \ldots,\ (2n-2)\cdot g\}$$

mit

$$g \;:=\; (r+1)\cdot 2^t$$

beschränken. Es soll gezeigt werden, dass t eine untere Schranke für die Suchzeit im worst-case ist. Ist $(2n-1)\cdot g \le m-1$, so folgt aus der Wahl von g

$$2^t \le \frac{m-1}{(2n-1)\cdot(r+1)} \quad \text{und somit} \quad t \in \Omega\left(\log\frac{m}{n\cdot s}\right)\ .$$

Aus der Vorbedingung $s < \frac{m}{n^2}$ folgt schließlich $t \in \Omega(\log n)$.

Wie oben schon dargestellt, werden wir einen Adversary konstruieren, welchen beginnend mit der Menge $U_0 := U'$ die Menge der Kandidaten, auf welche eine Suchfrage zutreffend ist, Schritt für Schritt weiter einschränkt. Hierbei seien die Mengen U_i konstruiert, auf denen die RAM in den ersten i Schritten immer gleich reagiert. Diese sind also für die RAM nicht zu unterscheiden. Von großer Bedeutung wird hierbei sein, dass für wachsendes i die Werte von $|U_i|$ und $|U_i \cap S|$ nicht zu schnell schrumpfen.

Von weiterer Bedeutung werden noch die Intervalle in U' sein, welche vollständig in einer aktuell betrachteten Teilmenge $U'' \subseteq U'$ enthalten sind. Wir sagen, dass U'' aus ℓ *Blöcken* besteht, wenn U'' aus ℓ vollständigen Intervallen aus U' besteht. Ferner muss für jedes Paar solcher Intervalle zumindest ein Element aus $U' \setminus U''$ zwischen diesen Intervallen liegen. Diese Eigenschaft ist in Abbildung 12 illustriert.

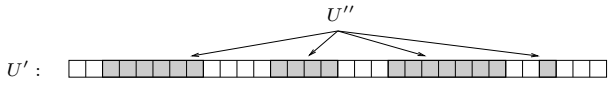


Abbildung 12: Zwei Mengen U' und U'', wobei U'' aus vier Blöcken besteht.

Kommen wir nun zur Beschreibung der RAM. Hierbei müssen wir jedoch darauf achten, dass diese Beschreibung keine RAM ausschließt bzw. jede RAM leicht in eine neue RAM überführt werden kann, die dieser Beschreibung genügt und ohne dass diese Transformation zu einem signifikanten Effizienzverlust führt.

Seien $M[0], M[1], M[2], \ldots$ die Register des Speichers, dann enthält zu Beginn $M[0]$ die Eingabe x und $M[1], \ldots, M[r]$ die Datenstruktur, in welcher die Menge S gespeichert ist. Für alle Speicherpositionen $i > r$ sei $M[i] = 0$. Zum verwalten der Parameter der einzelnen Operationen verfügt die RAM über zwei zusätzliche Register, sogenannten *Accumulatoren*, welche wir mit *acc* und *acc'* bezeichnen. Der Befehlssatz der RAM besteht aus den folgenden Operationen:

1. *HALT, ACCEPT, REJECT*
2. *LOAD*: lädt eine Konstante a in den Accumulatoren *acc*: $acc \leftarrow a$.
3. *ADD, SUB*: addiert *acc* und *acc'* bzw. subtrahiert *acc'* von *acc*: $acc \leftarrow acc + acc'$ bzw. $acc \leftarrow acc - acc'$
4. *MULT*: multipliziert *acc* und *acc'*: $acc \leftarrow acc \cdot acc'$.
5. *READ*: liest ein Register des Speichers: $acc \leftarrow M[acc']$

6. *WRITE*: schreibt den Wert eines Accumulators in den Speicher: $M[\,acc'\,] \leftarrow acc$.
7. *JUMP*: springt bei Zutreffen einer Bedingung an eine angegebene Befehlszeile, und setzt die Ausführung des Programms von dieser Zeile an fort: If *acc¿0* then go to line L.

Diese Befehle stehen auch bei vertauschten Accumulatoren zur Verfügung.

Mit c_t werden wir den Wert des Befehlszählers und mit p_t und p'_t die Werte der Accumulatoren zu Beginn des t-ten Schritts bezeichnet. Da wir das Verhalten der RAM auf verschiedene Eingaben x untersuchen wollen, sind diese Werte Funktionen von x. Zu Beginn einer Berechnung sei für alle $x \in U'$ $p_0(x) = p'_0(x) = 0$ und $c_0(x)$ ein konstanter Wert.

Um ein gleiches verhalten einer RAM auf zwei verschiedene Eingaben zu untersuchen, müssen wir noch die Speicherzugriffe der RAM beschreiben. Mit Q_t bezeichnen wir die Menge der Funktionen, welche – in Abhängigkeit von x – die bis zum t-ten Schritt von der RAM beschriebenen Speicherzellen angibt. Die Menge $\{\ q(x) \mid q \in Q_t\ \}$ ist somit die Menge der von der RAM auf Eingabe $x \in U'$ bis zum t-ten beschriebenen Speicherstellen. Zu Beginn sei $Q_0 := \{\ x \to 0\ \}$. Um den Wert der jeweiligen Speicherstellen nach dem Schreiben zu bezeichnen, verwenden wir die Funktion $v_{q,t}$, wobei für jedes $q \in Q_t$ $v_{q,t}(x) := M[q(x)]$ ist. Somit ist $v_{x\to 0,\ 0}(x) = x$.

Betrachten wir zunächst das folgende kleine Beispiel

1. *WRITE* 17 *TO* $2x$
2. *WRITE* x^2 *TO* 5

Für die oben beschriebenen Mengen erhalten wir $Q_0 := \{\ x \to 0\ \}$, $Q_1 := \{\ x \to 0,\ x \to 2x\ \}$, und $Q_2 := \{\ x \to 0,\ x \to 2x,\ x \to 5\ \}$. Zudem gilt $v_{x\to 0,\ 0}(x) = v_{x\to 0,\ 1}(x) = v_{x\to 0,\ 2}(x) = x$, $v_{x\to 2x,\ 1}(x) = v_{x\to 2x,\ 2}(x) = 17$ und $v_{x\to 5,\ 2}(x) = x^2$.

Mit Hilfe der oben angegebenen Funktionen und Mengen können wir nun definieren, was wir unter dem Ausdruck: „*Die RAM verhält sich im Schritt i auf alle Elemente $x \in U_i$ gleich.*“ verstehen. Diese Aussage ist erfüllt, wenn sich die RAM im Schritt $i-1$ auf alle Elemente $x \in U_i$ gleich verhält, und

(A) die Werte $c_i(x)$ für alle $x \in U_i$ gleich sind, d.h. es existiert ein Wert c, so dass für alle $x \in U_i$ $c_i(x) = c$.

(B) die Werte der Accumulatoren durch p_i und p'_i beschrieben werden, d.h. für alle $x \in U_i$ hat *acc* den Wert $p_i(x)$ und *acc'* den Wert $p_i(x)'$.

(C) für alle $x \in U_i$ und alle $q \in Q_i$ hat $M[q(x)]$ den Wert $v_{q,i}(x)$.

(C) für alle $x \in U_i$ und alle noch nicht beschriebenen Speicherstellen $M[j]$ – d.h. $j \notin \{\ q(x) \mid q \in Q_i\ \}$ – hat $M[j]$ noch den Initialwert.

Der Adversary beeinflußt nun über seine Antworten die Werte von c_i, U_i, p_i, p'_i, Q_i und $v_{q,i}$ für alle $q \in Q_i$. Dabei muss er dafür Sorge tragen, dass sich die RAM in allen Schritten $j \leq i$ auf alle Elemente $x \in U_i$ gleich verhält, zudem soll er die folgenden technischen Einschränkungen garantieren:

Lemma 1 *Die oben beschriebenen Werte können gewählt werden, so dass für alle $0 \leq i \leq t$ gilt:*

(a) U_i *besteht aus höchstens* 2^{2i} *Blöcken,*

(b) $|U_i \cap S| \geq \frac{n}{2^i}$,

(c) für alle $q \in Q_i$ *sind* q , $v_{q,t}$, p_t *und* p'_t *Polynome vom Grad* $\leq 2^i$,

(d) $|Q_i| \leq i+1$ *und*

(e) für alle $q, q' \in Q_i$ *mit* $q \neq q'$ *und alle* $x \in U_i$ *gilt* $q(x) \neq q'(x)$.

Beweis: Der Beweis erfolgt über eine Induktion über i .

Zu Beginn, d. h. für $i = 0$, ist $U_0 = U'$, $Q_0 = \{\ x \to 0\ \}$, $v_{x \to 0,\ 0} = x$, $p_0 = p'_0 = 0$ und c_0 die Startposition für alle Eingaben. Diese Werte erfüllen die Behauptungen des obigen Lemmas.

Nehmen wir nun an, dass die Behauptungen für alle Schritte $j \leq i$ erfüllt sind. Über eine Fallanalyse über alle möglichen Typen von Operationen soll nun gezeigt werden, dass Behauptungen auch noch nach dem $(i+1)$-ten Schritt erfüllt sind, oder vielmehr, dass der Adversary seine Antwort so wählen kann, dass die Behauptung nach dem $(i+1)$-ten Schritt erfüllt ist.

1) *HALT, ACCEPT, REJECT*
2) *LOAD*: *acc*: $acc \leftarrow a$
3) *ADD, SUB*: $acc \leftarrow$ *acc* + *acc'* bzw. $acc \leftarrow$ *acc* − *acc'*
4) *MULT*: $acc \leftarrow$ *acc* · *acc'*
5) *READ*: $acc \leftarrow M[$ *acc'*$]$
6) *WRITE*: $M[$ *acc'*$] \leftarrow$ *acc*
7) *JUMP*: If $acc > 0$ then go to line L .

Im ersten Fall, ist das $(i+1)$-te Kommando ein *HALT, ACCEPT* oder *REJECT* werden die Werte von c_i , U_i , p_i , p'_i , Q_i und $v_{q,i}$ unverändert gelassen. Die Bedingung gilt daher trivialer Weise auch nach diesem Schritt.

Handelt es sich bei dem $(i+1)$-te Kommando um ein *LOAD* ($acc \leftarrow a$), so verändert sich nur $p_{i+1}(x)$ und c_{i+1} . Wir erhalten

$$\begin{array}{rcl} c_{i+1}(x) & := & c_i(x)+1, \\ \forall q \in Q_{i+1} \ : \ v_{q,i+1} & := & v_{q,i}, \end{array} \qquad \begin{array}{rcl} U_{i+1} & := & U_i, \\ p'_{i+1} & := & p'_i, \end{array} \qquad \begin{array}{rcl} Q_{i+1} & := & Q_i, \\ p_{i+1} & := & a\ . \end{array}$$

Die Bedingung gilt daher auch nach diesem Kommando.

Bei *ADD* bzw. *SUB* erhalten wir

$$\begin{array}{rcl} c_{i+1}(x) & := & c_i(x)+1, \\ \forall q \in Q_{i+1} \ : \ v_{q,i+1} & := & v_{q,i}, \end{array} \qquad \begin{array}{rcl} U_{i+1} & := & U_i, \\ p'_{i+1} & := & p'_i, \end{array} \qquad \begin{array}{rcl} Q_{i+1} & := & Q_i, \\ p_{i+1} & := & p_i \pm p'_i\ . \end{array}$$

Nach Ausführen dieses Befehls ist p_{i+1} ein Polynom, dessen Grad dem maximalen Grad von p_i und p'_i entspricht. Die Behauptung des Lemmas sind somit weiterhin erfüllt.

Für die Multiplikation *MULT* gilt analog

$$\begin{array}{rcl} c_{i+1}(x) & := & c_i(x)+1, \\ \forall q \in Q_{i+1} \ : \ v_{q,i+1} & := & v_{q,i}, \end{array} \qquad \begin{array}{rcl} U_{i+1} & := & U_i, \\ p'_{i+1} & := & p'_i, \end{array} \qquad \begin{array}{rcl} Q_{i+1} & := & Q_i, \\ p_{i+1} & := & p_i \cdot p'_i\ . \end{array}$$

Der Grad von p_{i+1} ist folglich die Summe der Grade von p_i und p'_i . Da nach Voraussetzung die Grade von p_i und p'_i durch 2^1 nach oben beschränkt sind, ist der Grad von p_{i+1} höchstens 2^{i+1} . Die Aussage von Lemma 1 ist daher auch weiterhin erfüllt.

Da in einer *READ*-Operation sich nur der Wert eines Accumulators ändert, gilt:

$$c_{i+1}(x) \;:=\; c_i(x)+1, \qquad Q_{i+1} \;:=\; Q_i, \qquad \forall q \in Q_{i+1} \;:\; v_{q,i+1} \;:=\; v_{q,i}, \qquad p'_{i+1} \;:=\; p'_i \;.$$

Für die Analyse der Werte des veränderten Accumulators und U_{i+1} unterscheiden wir die folgenden Fälle:

1. $p'_i \in Q_i$:
   Da in diesem Fall $M[p'_i(x)] = v_{p'_i,i}(x)$ für alle $x \in U_i$ ist, gilt $p_{i+1} := v_{p'_i,i}$ und $U_{i+1} = U_i$.

2. $p'_i \notin Q_i$ und $p'_i = a$ ist eine Konstante:
   In diesem Fall wählen wir $U_{i+1} := \{\, x \in U_i \mid \forall q \in Q_i \;:\; q(x) \neq a \,\}$. Somit ist $p_{i+1}(x)$ gleich dem Initialwert der Speicherstelle $M[a]$. Die Bedingungen c-e sind somit auch nach diesem Schritt weiterhin erfüllt.

   Zur Analyse der ersten und zweiten Bedingung betrachten wir die Mengen $M_q := \{\, x \mid q(x) = a \,\}$ für alle $q \in Q_i$. Aus Fakt 1 und der Bedingung, dass der Grad eines Polynoms $q \in Q_i$ durch 2^i beschränkt ist, folgt $|M_q| \leq 2^i$. Die Bedingung $|Q_i| \leq i+1$ impliziert somit:

   $$|U_{i+1}| \;=\; |U_i \setminus \bigcup_{q \in Q_i} M_q| \;\geq\; |U_i| - \sum_{q \in Q_i} |M_q| \;\geq\; |U_i| - (i+1)2^i$$

   Die Anzahl der Blöcke in U_{i+1} erhöht sich von höchstens 2^{2i} auf maximal $2^{2i} + (i+1)2^i \leq 2^{2(i+1)}$.

   Betrachten wir nun die Größe von $U_{i+1} \cap S$. Analog zu unserer obigen Beobachtung gilt:

   $$|U_{i+1} \cap S| \;\geq\; |(U_i \cap S) \setminus \bigcup_{q \in Q_i} M_q| \;\geq\; |U_{i+1} \cap S| - \sum_{q \in Q_i} |M_q| \;\geq\; \frac{n}{2^i} - (i+1)2^i \;.$$

   Folglich ist für $n \geq (i+1)2^{2i+1}$ $\quad |U_{i+1} \cap S| \geq n \cdot 2^{-(i+1)}$. Daher sind die Bedingungen a und b aus Lemma 1 (für $i \leq \frac{\log_2 n}{3}$) erfüllt.

3. $p'_i \notin Q_i$ und p'_i ist keine Konstante:
   In diesem Fall geht unser Adversary davon aus, dass p'_i eine noch nicht benutzte Speicherstelle adressiert. Wir wählen $p_{i+1} = 0$ und

   $$U_{i+1} \;:=\; \{\, x \in U_i \mid p'_i(x) \notin \{0,\ldots,r\} \text{ und } \forall q \in Q_i \;:\; p'_i(x) \neq q(x) \,\} \;.$$

   Man beachte nun, dass der Abstand zwischen zwei Elementen aus U_i zumindest $2^i \cdot (r+1)$ ist. Da der Grad von p'_i zudem durch 2^i beschränkt ist, folgt aus Fakt 3:

   $$|\{\, x \in U_i \mid p'_i(x) \in \{0,\ldots,r\} \,\}| \;\leq\; 2^i \;.$$

   Auf der anderen Seite folgt aus der Bedingung c, dass $q - p'_i$ für alle $q \in Q_i$ ein Polynom vom Grad $\leq 2^i$ ist. Daher folgt aus Fakt 1:

   $$\begin{aligned} |U_{i+1}| \;&\geq\; |U_i| - |\{\, x \in U_i \mid p'_i(x) \in \{0,\ldots,r\} \,\}| - |Q_i| \cdot \max_{q \in Q_i} |\{\, x \in U_i \mid p'_i(x) - q(x) = 0 \,\}| \\ &\geq\; |U_i| - 2^i - (i+1)\cdot 2^i \;=\; |U_i| - (i+2)\cdot 2^i \;. \end{aligned}$$

   Es folgt unmittelbar, dass die Anzahl der Blöcke in U_{i+1} durch $2^{2i} + (i+2)\cdot 2^i \leq 2^{2(i+1)}$ beschränkt ist. Ferner gilt für $n \geq (i+2)2^{2i+1}$ bzw. $i \leq \frac{\log_2 n}{3}$

   $$|U_{i+1} \cap S| \;\geq\; |U_i \cap S| - (i+2)\cdot 2^i \;\geq\; \frac{n}{2^{i+1}} \;.$$

Die Behauptungen des Lemmas gelten auch nach dem $(i+1)$-ten Kommando, wenn dieses ein *LOAD*-Befehl ist.

Betrachten wir nun den Fall, dass die $(i+1)$-te Operation eine *WRITE*-Instruktion $M[p_i(x)] \leftarrow p'_i(x)$ ist. Nach diesem Befehl gilt trivialer Weise:

$$\begin{array}{rclcrclcrcl} c_{i+1}(x) & := & c_i(x)+1, & & p'_{i+1} & := & p'_i, & & p_{i+1} & := & p_i \cdot p'_i \\ Q_{i+1} & := & Q_i \cup \{p_i\}, & \forall q \in Q_i & : & v_{q,i+1} & := & v_{q,i}, & v_{p_i,i+1}(x) & := & p'_i(x) \ . \end{array}$$

Für den Wert von U_{i+1} müssen wir an dieser Stelle wieder mehrere Fälle unterscheiden:

1. $p'_i \in Q_i$:
   In diesem Fall sei $U_{i+1} := U_i$. Die Behauptung folgt unmittelbar.

2. $p'_i \notin Q_i$:
   In diesem Fall wählen wir

   $$U_{i+1} \ := \ \{ \ x \in U_i \mid \forall q \in Q_i \ : \ \ p_i(x) \neq q(x) \ \} \ .$$

   Aus dieser Konstruktion sowie aus der Induktionsannahme folgt sofort, dass die Bedingung c, d und e nach dem *WRITE*-Kommando erfüllt sind.

   Betrachten wir nun die Mengen $M_q \ := \ \{ \ x \in U_i \mid p_i(x) - q(x) = 0 \ \}$ für alle $q \in Q_i$. Da der Grad des Polynoms $p_i(x) - q(x)$ durch 2^i beschränkt ist, folgt aus Fakt 1, dass $|M_q| \leq 2^i$. Da ferner $|Q_i| \leq i+1$ und jedes Element aus $\bigcup_{q \in Q_i} M_q$ die Anzahl der Blöcke um höchsten einen erhöht, ist die Anzahl der Blöcke höchstens

   $$2^{2i} + |\bigcup_{q \in Q_i} M_q| \ \leq \ 2^{2i} + |Q_i| \cdot \max_{q \in Q_i} |M_q| \ \leq \ 2^{2i} + (i+1) \cdot 2^i \ \leq 2^{2(i+1)} \ .$$

   Zudem gilt für $n \geq (i+1) \cdot 2^{2i+1}$ (bzw. $i \leq \frac{\log_2 n}{3}$)

   $$|U_{i+1} \cap S| \ \geq \ |U_i \cap S| - (i+1) \cdot 2^i \ \geq \ \frac{n}{2^i} - (i+1) \cdot 2^i \ \geq \ \frac{n}{2^{i+1}} \ .$$

   Somit gelten nach der *WRITE*-Instruktion weiterhin die Bedingungen a und b des Lemmas.

Es verbleibt noch die Analyse des bedingten Sprungkommandos *JUMP*. Bei diesem Kommando wird abhängig vom Wert $p_i(x)$ des Akkumulators *acc* ein Sprung ausgeführt oder nicht. Ist $p_i(x) > 0$, so setzen wir den Wert $c_{i+1}(x)$ des Befehlszählers auf L . Ist hingegen $p_i(x) \leq 0$, so fahren wir mit dem im Programmtext folgenden Befehl fort, d.h. $c_{i+1}(x) := c_i(x) + 1$. Da die einzigen Werte, die durch dieses Kommando verändert werden, der Wert des Befehlszählers und die Menge U_{i+1} ist, setzen wir

$$Q_{i+1} := Q_i, \quad p_{i+1} := p_i, \quad p'_{i+1} := p'_i, \quad \forall \ q \in Q_{i+1} \ : \ \ v_{q,i+1} := v_{q,i} \ .$$

Zur Analyse der Werte von c_{i+1} und U_{i+1} unterscheiden wir wieder zwischen den folgenden Fällen:

1. $p_i = a$ ist eine Konstante:
   In diesem Fall verhält sich der Sprungbefehl für alle Eingaben $x \in U_i$ gleich. Es gilt daher

   $$U_{i+1} \ := \ U_i, \qquad c_{i+1}(x) \ := \ \begin{cases} c_i(x)+1 & \text{für } a \leq 0, \\ L & \text{für } a > 0 \ . \end{cases}$$

   Die Behauptung des Lemmas folgt somit unmittelbar aus der Induktionsannahme.

2. p_i ist keine Konstante:
   In diesem Fall wählen wir U_{i+1} bezüglich des Schnitts mit S maximal, d.h.

$$U_{i+1} := \begin{cases} \{\ x \in U_i \mid p_i(x) > 0\} & \text{für } |\{\ x \in U_i \mid p_i(x) > 0\} \cap S| \ \geq\ n \cdot 2^{-(i+1)} \\ \{\ x \in U_i \mid p_i(x) \leq 0\} & \text{ansonsten} \end{cases}$$

   und entsprechend

$$c_{i+1} := \begin{cases} L & \text{für } |\{\ x \in U_i \mid p_i(x) > 0\} \cap S| \ \geq\ n \cdot 2^{-(i+1)} \\ c_i + 1 & \text{ansonsten.} \end{cases}$$

   Man beachte, dass diese Wahl immer so möglich ist, dass $|U_{i+1} \cap S| \geq \frac{n}{2^{i+1}}$ – dieses folgt unmittelbar aus der Überlegung, dass wir die $|U_i \cap S| \geq \frac{n}{2^i}$ Elemente aus $U_i \cap S$ auf die beiden Möglichkeiten $p_i(x) > 0$ und $p_i(x) \leq 0$ aufteilen müssen. Zumindest die Hälfte – und somit $\frac{n}{2^{i+1}}$ – dieser Elemente muss daher eine dieser Bedingungen erfüllen.

   Es verbleibt nunmehr die Anzahl der Blöcke in U_{i+1} zu analysieren. Da wir U_i abhängig von p_i in zwei Teilmengen aufteilen — $U_i' = \{\ x \in U_i \mid p_i(x) > 0\}$ und $U_i'' = \{\ x \in U_i \mid p_i(x) \leq 0\}$, wovon eine Menge dann U_{i+1} entspricht — erhöht sich die Anzahl der Blöcke höchsten um die Hälfte der Nullstellen des Polynoms p_i. Diese Aufteilung ist in Abbildung 13 illustriert.

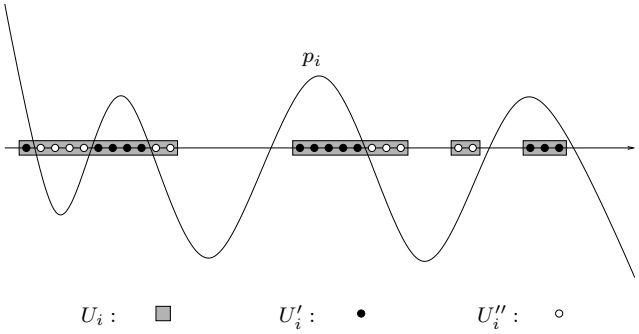


Abbildung 13: Aufteilung der Menge U_i in U_i' und U_i''.

   Die Anzahl der Blöcke in U_{i+1} ist durch $2^{2i} + 2^{i-1} \leq 2^{2(i+1)}$ beschränkt. Somit gelten auch in diesem Fall nach Ausführen des *WRITE*-Kommandos die Bedingungen a und b.

Zusammenfassend lässt sich somit sagen, dass die Aussage des Lemmas auch nach der $(i+1)$-ten Instruktion gültig ist, wenn $n \geq (i+2) \cdot 2^{2i+1}$ ist. Wählen wir für unsere Zeitschranke $t := \lceil \frac{\log_2 n}{3} \rceil - 1$, so gilt die Behauptung des Lemmas bis zum Ende der t-ten Instruktion. ∎

Aus Lemma 1 können wir nun schließen, dass U_t mit $t := \lceil \frac{\log_2 n}{3} \rceil - 1$ höchstens 2^{2t} Blöcke beinhaltet. Zudem gilt $|U_t \cap S| \geq n \cdot 2^{-t}$

$$|U_t \cap S| \ \geq\ n \cdot 2^{-t} \ >\ n \cdot 2^{-(\log_2 n)/3} \ \geq\ n^{2/3} \ >\ 2^{2 \cdot (\lceil (\log_2 n)/3 \rceil - 1)} \ =\ 2^{2t}\ .$$

Da also mehr Elemente in $U_t \cap S$ sind als Blöcke in U_t, muss es nach dem *Pigeon Hole Principle* zumindest einen Block in U_t geben, der zwei aufeinander folgende Elemente $x_1 := 2jg$ und $x_2 := 2(j+1)g$ von S umfasst. Da x_1 und x_2 jedoch Elemente eines Blocks sind, beinhaltet U_t auch das Element $x_0 := (2j+1)g \notin S$. Da sich die RAM auf allen Elementen aus U_t jedoch in den ersten t Schritten gleich verhält und somit vor allem das gleiche Akzeptanzverhalten aufweist, muss die t-zeitbeschränkte RAM auf einer der Eingaben x_0 oder x_1 eine fehlerhafte Ausgabe generieren.

Wir können somit schließen, dass eine RAM, welche die Operationen *ADD*, *SUB* und *MULT* jedoch nicht *SHIFT* oder *DIV* benutzt, zur korrekten Beantwortung einer *membership*-Anfrage eine worst-case Zeit von $\Omega(\log n)$ benötigt. ∎

3.3 Randomisiertes perfektes Hashing und dessen Derandomisierung

In Kapitel 3.2.4 haben wir ein perfektes Hashverfahren kennen gelernt, welches eine Hashtabelle linearer Größe benutzte. Das Bestimmen der für eine Menge $S \subset U \subset \{0,1\}^k$ mit $|S| = n$ geeigneten Hashfunktion benötigte auf einer RAM $O(kn^2)$ Schritte. Wir wollen nun im Folgenden ein im Erwartungswert $O(n)$ zeitbeschränktes randomisiertes Verfahren vorstellen, um eine perfekte Hashfunktion $h : U \to \{0, \ldots, r-1\}$ zu bestimmen. Hierbei soll die Tabellengröße r wieder linear in n sein. Im Anschluss soll dieses Verfahren derandomisiert werden. Diese Verfahren wurden in [TaYa79] bzw. [Pagh00] publiziert.

3.3.1 Ein randomisiertes Hashverfahren

Wir werden uns hier auf Universen polynomieller Größe beschränken, d.h. $|U| \in n^{O(1)}$. Daher können die Elemente aus U mit Hilfe binärer Strings logarithmischer Länge dargestellt werden.

Die Konstruktion unserer Hashfunktion wird in zwei Schritten erfolgen. Im ersten Schritt soll die Problemgröße von $|U| \in n^{O(1)}$ deterministisch auf ein Hashproblem über einem Universum U' quadratischer Größe reduziert werden. Dieses Hashproblem soll dann randomisiert gelöst werden.

Sei $|U| = 2^k = n^d$ für $n, k, d \in \mathbb{N}$, d.h. $n = 2^{k/d}$. Zur Repräsentation von S benutzen wir in unserem ersten Schritt einen n-nären Baum. Hierzu betrachten wir die n-nären Darstellung der Elemente $x \in U$. Um diese Repräsentation zu erhalten, zerlegen wir die binäre Repräsentation von x in Blöcke der Länge k/d. Jedes Element $x \in U$ besteht somit aus $d \in O(1)$ Blöcke $B_0(x), \ldots, B_{d-1}(x)$:

$$x = \underbrace{\underbrace{101\ldots11}_{k/d \text{ bits}}\underbrace{011\ldots10}_{k/d \text{ bits}}\ldots\underbrace{001\ldots01}_{k/d \text{ bits}}}_{d \text{ Blöcke } B_0(x),\ldots,B_{d-1}(x)}.$$

Diese Aufteilung kann auf einer RAM mit Hilfe der Division in $O(d) = O(1)$ für jedes $x \in U$ erfolgen. Zudem können wir den Wert jedes Blocks $B_i(x)$ in Zeit $O(1)$ bestimmen.

Mit Hilfe dieser Blockaufteilung, können wir S in einem n-nären Suchbaum darstellen. Jedes Element x aus S beschreibt hierbei einen eineindeutigen Pfad π_x von einem Blatt zur Wurzel des Baumes, wobei der i-te Block von x angibt, welcher Sohn des i-te Knotens v in diesem Pfad π_x der Nachfolger von v in π_x ist. Um den Baum möglichst kompakt zu gestalten, entfernen wir die Knoten, welche zu keinem Pfad π_x für ein $x \in S$ gehören. Ein Beispiel für einen solchen 4-nären Suchbaum ist in Abbildung 14 zu sehen.

Da $|U| = n^d$, ist die Tiefe des n-nären Suchbaums jeder Teilmenge von U gleich d. Der Suchbaum für S hat höchstens $d \cdot n \in O(n)$ Knoten. Um eine *membership* Anfrage zu beantworten, genügt es, somit dem Pfad zu folgen der durch eine Eingabe x angegeben wird. Endet dieser in einem Blatt, so ist $x \in S$, endet der Pfad an einem inneren Knoten, welcher keinen durch x angegebenen Nachfolger mehr im Baum hat, so ist $x \notin S$. Die *membership* Anfrage kann somit in konstanter Zeit beantwortet werden. Auch kann der Suchbaum in der Zeit $O(d \cdot n)$ konstruiert werden.

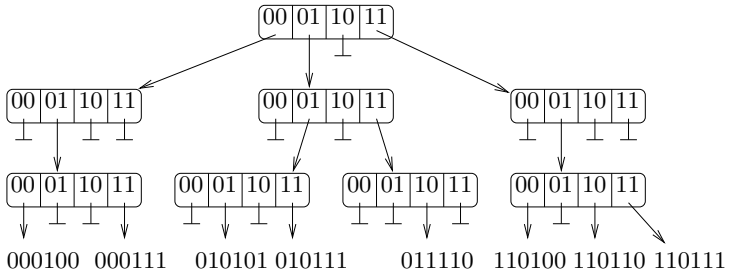


Abbildung 14: 4-närer Suchbaum für $S = \{000100, 000111, 010101, 010111, 011110, 1101000, 110110, 110111\}$.

Um den Suchbaum für S zu speichern, benutzen wir eine Tabelle in der wir die $d \cdot n$ Knoten des Baumes abspeichern können. Man beachte hierbei, dass jeder Baumknoten wiederum aus einer Tabelle von bis zu n Adressen besteht. Da aber jede Adresse eine Tabellenposition ist, und die Tabelle über maximal $d \cdot n^2$ Einträge verfügt, müssen wir folglich in jeder Speicherzelle nur $O(\log n)$ Bits speichern. Diese können in dem von uns gewählten Maschinenmodell an einer Speicherzelle abgelegt werden. Wollen wir einem Pfad im Baum folgen, so können wir mit der an einer Speicherstelle gespeicherten Adresse zu der Speicherstelle springen, an der die für den nachfolgenden Knoten relevante Teiltabelle beginnt. Der entsprechende Block in der Eingabe x enthält dann die relativen Koordinaten der Speicherzelle, an dem die Adresse des nächsten Knotens zu finden ist. Folgen wir auf diese Weise einem Suchpfad im Baum, so können wir eine *membership* Anfrage wiederum in konstanter Zeit beantworten. Wie der Suchbaum kann auch diese Tabelle in Zeit $O(d \cdot n)$ konstruiert werden.

Ein Beispiel für einen solchen Tabelle ist in Abbildung 15 abgebildet.

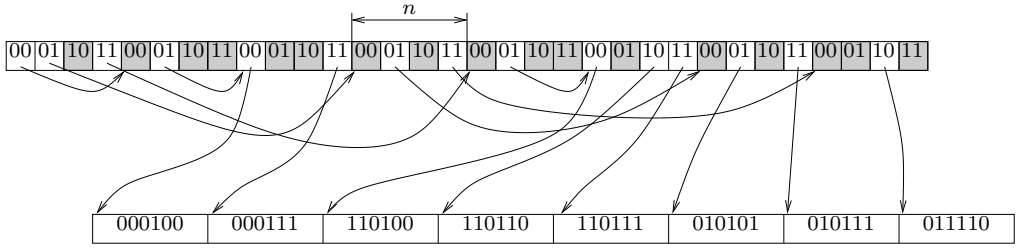


Abbildung 15: Tabelle zum Speichern des Suchbaumes aus Abbildung 15.

Betrachten wir nun einmal die Implementierung der Tabelle als unsere eigentliche Aufgabe. Es ist einfach zu erkennen, dass die Tabelle nur $O(d \cdot n)$ Nicht-NIL Einträge beinhaltet, wenn wir die Blätter, welche die Elemente aus S speichern, nur durch eine Zelle darstellen. Man beachte, dass die Elemente aus U durch binäre Zeichenketten logarithmischer Länge dargestellt und daher auch in einer Speicherzelle gespeichert werden können.

Stellen wir Adressen der Speicherzellen als ein neues Universum U' dar und die Menge der nicht leeren Speicherstellen als eine neue Menge S', so erhalten wir ein neues Problem für ein statisches Wörterbuch. Man beachte hierbei, dass die Beantwortung einer *membership*-Frage (ist die adressierte Speicherstelle nicht leer) nicht ausreicht, um das obige Problem zu lösen, vielmehr müssen wir auch den Inhalt einer nichtleeren Speicherstelle erfahren. Das im Folgenden konstruierte Wörterbuch kann diese Aufgabe jedoch ohne zusätzlichen Aufwand bei der Lösung des *membership*-Problems lösen.

Im Folgenden werden wir uns auf das *membership*-Problem konzentrieren, merken uns jedoch, dass wir an den entsprechenden Stellen der Tabelle immer ein Keyword und eine Zusatzinformation speichern. Für unser oben beschriebenes Problem bedeutet dieses, dass jeder Eintrag im Wörterbuch die aktuelle Adresse in der oben beschriebenen Tabelle sowie einen Zeiger auf die Teiltabelle des entsprechenden Nachfolgerknotens speichert. Zur Beantwortung einer *membership*-Frage im Startproblem S, U benötigen wir somit die Beantwortung von d *membership*-Frage im neuen Problem S', U'. Man beachte, dass $|U'| = m' \in \Theta(n^2)$ und $|S'| = n' \in \Theta(n)$. Erlauben wir eine lineare aufblähung der Problemstellung — dieses kann durch eine Vergrößerung des Universums geschehen — so gilt $k' = \lceil \log_2 |U'| \rceil \geq 2 \log_2 n' + \ell$ für eine Konstante ℓ, die wir im Folgenden noch festlegen müssen.

In einem ersten Schritt stellen wir U' als eine $2^\alpha \times 2^\beta$-Matrix $M = (a_{i,j})$ mit $\alpha + \beta = k'$ dar, deren Einträge wie folgt bestimmt werden:

$$a_{i,j} \quad := \quad \begin{cases} x \in S' & \text{wenn } x_0 \dots x_{\alpha-1} = j \text{ und } x_\alpha \dots x_{\alpha+\beta-1} = i \\ \text{leer} & \text{sonst.} \end{cases}$$

Ist $a_{i,j} = x$ für ein $x \in S'$ so entspricht die Zeilennummer i den α höchstwertigsten Bits der Binärdarstellung von x und die Spaltennummer j die β niederwertigsten Bits der Binärdarstellung von x. Für α und β wählen wir die folgenden Werte:

$$\alpha \; := \; \lceil \log_2 n' \rceil + 1 \quad \text{und} \quad \beta \; := \; \lceil \log_2 n' \rceil + k - 1 \; .$$

Die Aufteilung der Binärdarstellung sowie die Darstellung von S' mit Hilfe der Matrix M ist in Abbildung 16 illustriert.

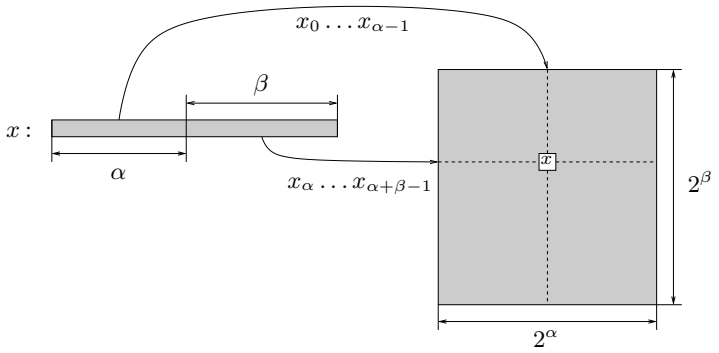


Abbildung 16: Speichern einer Menge S' in einer Matrix.

Da die Größe der Matrix in $\Theta(n^2)$ und die Größe von S in $\Theta(n)$ liegen, ist die Matrix M nur sehr dünn besetzt. Um ein perfektes Hashverfahren zu bekommen, permutieren wir die einzelnen Spalten und anschließend die Zeilen, so dass die resultierende Matrix nur einen nicht leeren Eintrag pro Spalte hat. Dieses ist möglich, da die Matrix $2^\alpha = 2^{\lceil \log_2 n' \rceil + 1} > 2n'$ Spalten besitzt. Eine perfekte Hashfunktion erhalten wir dann, indem wir jeden Eintrag an die Position der Tabelle eintragen, die durch die Spalten in der permutierten Matrix angegeben wird.

Da wir $2^\alpha + 2^\beta \in O(n)$ viele Permutationen benötigen, müssen die Spalten- und Zeilenpermutationen so gewählt werden, dass jede Permutationfunktion auf konstantem Platz abgespeichert werden kann. Die Spaltenpermutationen werden wir im Folgenden mit $\pi_0, \dots, \pi_{2^\alpha - 1}$ und die Zeilenpermutationen mit $\rho_0, \dots, \rho_{2^\beta - 1}$ bezeichnet. Dieses Herangehen ist in Abbildung 17 dargestellt.

Die erste Permutation soll gewährleisten, dass nur wenige Konflikte pro Zeile auftreten. Aufbauend auf dieser Matrix soll die zweite Permutation die Spalten Konflikte aufheben. Wir definieren daher:

$$S'_i \quad := \quad \{ \; x \in S' \mid x[0;\; \alpha - 1] = i \; , \text{ d.h. der Präfix von } x \text{ der Länge } \alpha \text{ ist } i \; \} \; ,$$

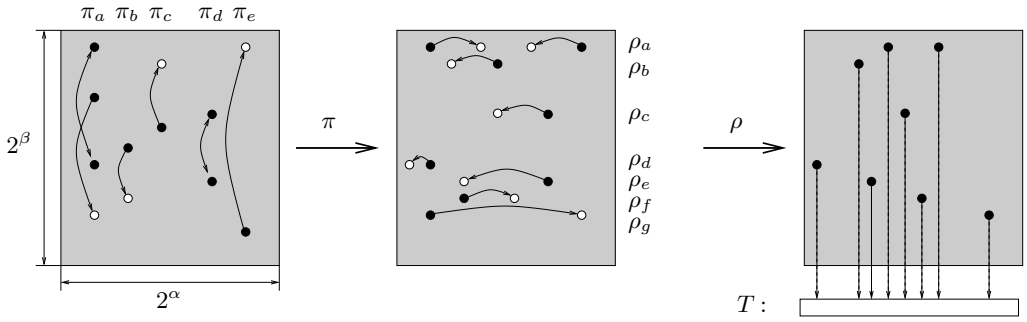


Abbildung 17: Spalten- und Zeilenpermutationen auf der Matrix M .

wobei j ein Binärstring der Länge α ist. Ferner werden wir mit $\boldsymbol{x[i]}$ die $(i+1)$-te Position in der Binärdarstellung von x adressieren, d.h. $x = x[0]x[1]\ldots x[|x|-1]$. Zudem sei $\boldsymbol{x[i;j]} := x[j]\ldots x[j]$.

Die Spaltenpermutationen werden wir Spalte für Spalte abhängig von den bisher gewählten Permutationen bestimmen. Sei also $\pi_0, \ldots, \pi_{i-1}$ eine Folge von bisher gewählten Permutationen, so bestimmen wir π_i abhängig von den auftretenden Zeilenkonflikten die durch die ersten i Spalten entstehen. Wir Definieren daher:

$$R'_i \;:=\; \{\ (x,y) \in S'_j \times S'_i \mid j < i \ \text{ und } \ \pi_i(x[\alpha;\ \alpha+\beta-1]) \;=\; \pi_j(y[\alpha;\ \alpha+\beta-1])\ \}\ .$$

Im Folgenden wollen wir zeigen, dass wir auf eine sehr einfache Form der Permutationen einschränken können, die über ein bitweises XOR der Zeichenkette $x[\alpha;\ \alpha+\beta-1]$ mit einem Zufallsstring b'_i berechnet werden kann,

$$\pi_i(x[\alpha;\ \alpha+\beta-1]) \;:=\; x[\alpha;\ \alpha+\beta-1] \ \otimes\ b'_i\ .$$

Zum Speichern einer Permutation wird daher nur die Zeichenkette b'_i benötigt. Da die Länge von b'_i logarithmisch in n beschränkt ist, können wir b'_i in einer Speicherzelle ablegen. Der Speicheraufwand zum Speichern aller 2^α Spaltenpermutationen beträgt $\Theta(n)$.

Die oben beschriebenen Permutationen vertauschen die Einträge zweier Positionen einer Spalte bzw. Zeile. Ein Beispiel ist in Abbildung 18 dargestellt.

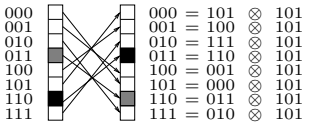


Abbildung 18: Beispiel einer XOR-Permutationen.

Wir wollen nun den Erwartungswert $E[|R'_i|]$ der Anzahl der Zeilenkollisionen nach einer Spaltenpermutation betrachten, wobei der Permutationsstring b'_i uniform aus $\{0,1\}^\beta$ gewählt wird. Wenn ein Element $y \in S'_j$ mit $j < i$ auf eine Zeile r durch die Permutation π_j geschoben wird, dann ist die Wahrscheinlichkeit $2^{-\beta}$, dass auch ein Element $x \in S'_i$ auf die Zeilenposition r geschoben wird. Es gilt somit

$$E[|R'_i|] \;=\; \frac{1}{2^\beta} \cdot |S'_i| \cdot \sum_{j<i} |S'_j|\ .$$

Da wir den Permutationsstring uniform aus $\{0,1\}^\beta$ wählen, folgt aus der Markhov-Ungleichung:

$$\text{Prob}[|R'_i| \geq (1+\varepsilon) \cdot E[|R'_i|]] \quad \leq \quad \frac{1}{1+\varepsilon}$$

bzw.

$$\text{Prob}[|R'_i| \leq (1+\varepsilon) \cdot E[|R'_i|]] \quad \geq \quad \frac{1}{1+\varepsilon} \; .$$

Somit können wir im Erwartungswert mit einer konstanten Anzahl von Versuchen (abhängig von ε) ein b'_i finden, so dass

$$|R'_i| \;\leq\; \frac{1+\varepsilon}{2^\beta} \cdot |S'_i| \cdot \sum_{j<i} |S'_j| \tag{2}$$

ist. Wie oben schon erwähnt, wählen wir für die Zeilenpermutationen den gleichen Ansatz wie für die Spaltenpermutationen:

$$\rho_i(x[0;\ \alpha-1]) \quad := \quad x[0;\ \alpha-1] \ \otimes \ b''_i \; ,$$

wobei wir b''_i aus $\{0,1\}^\alpha$ wählen. Im Unterschied zu den Spaltenpermutationen bestimmen wir die Zeilenpermutationen sortiert in der Reihenfolge der Anzahl ihrer Elemente. Sei also

$$S''_i \quad := \quad \{\ x \in S' \mid \pi_{x[0;\ \alpha-1]}(x[\alpha;\ \alpha+\beta-1]) = i\ \}$$

die Menge der Elemente aus S, die nach Ausführen aller Spaltenpermutationen in Zeile i sind. Sei $r_0, \ldots, r_{2^\beta-1}$ die Folge der Zeilenindizes, so dass für alle $i \in \{1, \ldots, 2^\beta - 1\}$:

$$|S''_{r_{i-1}}| \quad \geq \quad |S''_{r_i}| \; .$$

Wir bestimmen $\rho_0, \ldots, \rho_{2^\alpha-1}$ in der Reihenfolge $\rho_{r_0}, \rho_{r_1}, \rho_{r_2}, \ldots$. Man beachte, dass die entsprechende Folge der Zeilenindizes mit Hilfe von Bucket-Sort in linearer Zeit bestimmt werden kann.

Unter der Annahme, dass $\rho_{r_0}, \rho_{r_1}, \ldots, \rho_{r_{i-1}}$ schon festgelegt sind, bestimmen wir ρ_{r_i} wie folgt:

1. $|S''_{r_i}| = 1$, d.h. in Zeile r_i gibt es nur ein Element.

   In diesem Fall müssen wir den Permutationsstring nicht randomisiert bestimmen. Stattdessen wählen wir eine Spalte s_i der Matrix, in die wir bisher noch keinen Eintrag über die Permutationen $\rho_{r_0}, \rho_{r_1}, \ldots, \rho_{r_{i-1}}$ geschoben haben. Den Permutationsstring erhalten wir durch

   $$b''_i \quad := \quad x[0;\ \alpha-1] \ \otimes \ s_i \; ,$$

   wobei x das einzige Element in S''_{r_i} ist.

   Benutzen wir eine Liste der noch nicht besetzten Spalten, so kann der Index s_i für alle r_i mit $|S''_{r_i}| = 1$ in jeweils konstanter Zeit bestimmt werden. Die Liste dieser Spaltenindizes können wir wiederum in Zeit $O(2^\alpha + n) = O(n)$ vorweg bestimmen, und nach jedem Schritt in konstanter Zeit auf den jeweils aktuellen Stand bringen.

2. $|S''_{r_i}| \geq 2$, d.h. Zeile r_i hat zumindest zwei Elemente.

   In diesem Fall verfahren wir entsprechend dem Verfahren zur Bestimmung der Spaltenpermutationen. Sei also

   $$R''_i \quad := \quad \{\ (x,y) \in S''_{r_j} \times S''_{r_i} \mid j < i \ \text{ und } \ \rho_{r_i}(x[0;\ \alpha-1]) \ = \ \rho_{r_j}(y[0;\ \alpha-1])\ \}$$

   und

   $$\rho_{r_i}(x[0;\ \alpha-1]) \quad := \quad x[0;\ \alpha-1] \ \otimes \ b''_{r_i}$$

für ein uniform aus $\{0,1\}^\alpha$ gewähltes b''_{r_i}. Entsprechend der Analyse für die Spaltenpermutationen folgt für die Zeilenpermutationen, dass

$$\text{Prob}[|R''_i| \leq (1+\varepsilon) \cdot E[|R'_i|]] \quad \geq \quad \frac{\varepsilon}{1+\varepsilon} \; .$$

Nach einer im Erwartungswert konstanten Anzahl von Versuchen finden wir somit einen String b''_{r_i}, so dass

$$|R''_i| \quad \leq \quad \frac{1+\varepsilon}{2^\alpha} \cdot |S''_{r_i}| \cdot \sum_{j<i} |S''_{r_j}| \; . \tag{3}$$

Da $|S''_{r_i}| \leq |S''_{r_j}|$ für alle $j < i$ gilt, folgt

$$|S''_{r_i}| \cdot \sum_{j<i} |S''_{r_j}| \quad \leq \quad \sum_{j<i} |S''_{r_j}|^2 \quad \leq \quad 4 \cdot \sum_{j<i} \binom{|S''_{r_j}|}{2} \; . \tag{4}$$

$\binom{|S''_{r_j}|}{2}$ entspricht genau der Anzahl der Kollisionen in Zeile r_j, daher gilt

$$\sum_{j<i} \binom{|S''_{r_j}|}{2} \quad \leq \quad \sum_{j<2^\beta} \binom{|S''_{r_j}|}{2} \quad \leq \quad \sum_{i<2^\alpha} |R'_i| \; . \tag{5}$$

Fassen wir die Ungleichungen 2, 3, 4 und 5 zusammen, so erhalten wir:

$$\begin{aligned} |R''_i| \quad &\leq \quad \frac{1+\varepsilon}{2^\alpha} \cdot |S''_{r_i}| \cdot \sum_{j<i} |S''_{r_j}| \quad \leq \quad \frac{4 \cdot (1+\varepsilon)}{2^\alpha} \sum_{j<i} \binom{|S''_{r_j}|}{2} \quad \leq \quad \frac{4 \cdot (1+\varepsilon)}{2^\alpha} \sum_{i<2^\alpha} |R'_i| \\ &\leq \quad \frac{4 \cdot (1+\varepsilon)}{2^\alpha} \sum_{i<2^\alpha} \frac{1+\varepsilon}{2^\beta} \cdot |S'_i| \cdot \sum_{j<i} |S'_j| \quad \leq \quad \frac{4 \cdot (1+\varepsilon)^2}{2^{\alpha+\beta}} \binom{n'}{2} \; , \end{aligned}$$

da $\sum_{i<2^\alpha} |S'_i| \cdot \sum_{j<i} |S'_j|$ durch die Anzahl aller sortierten Paare in S' beschränkt ist. Setzen wir die für α und β gewählten Werte ein, so ergibt dieses:

$$|R''_i| \; \leq \; \frac{4 \cdot (1+\varepsilon)^2}{2^{\alpha+\beta}} \binom{n'}{2} \quad < \quad \frac{4 \cdot (1+\varepsilon)^2}{(n')^2 \cdot 2^k} \cdot \frac{(n')^2}{2} \quad = \quad \frac{\cdot (1+\varepsilon)^2}{2^{k-1}} \; .$$

Für $k \geq 1 + 2\log_2(1+\varepsilon)$ ist somit $|R''_i| < 1$. Da jedoch nur eine ganzzahlige Anzahl von Kollisionen möglich ist, können wir im Erwartungswert mit einer konstanten Anzahl von Versuchen ein b''_i bestimmen, so dass wir nach der Zeilenpermutation keine Spaltenkollisionen mit schon gewählten Zeilenpermutation haben.

Anschließend sei noch erwähnt, dass $1 + 2\log_2(1+\varepsilon)$ für ein konstantes ε ebenfalls konstant ist. Die Permutationsstrings können somit in jeweils einer Speicherzelle gespeichert werden.

Im Folgenden wollen wir noch einmal die Laufzeit dieses Hashverfahrens sowie den Platzbedarf für die Datenstruktur im Detail betrachten.

Die Datenstruktur muss neben der eigentlichen Tabelle T der Größe $2^\alpha \in O(n)$ für die Elemente aus S' auch eine Tabelle T' für die Spaltenpermutationen π_i bzw. für b'_i mit $2^\beta \in O(n)$ Einträgen sowie eine Tabelle T'' für die Zeilenpermutationen ρ_i bzw. für b''_i mit $2^\alpha \in O(n)$ Einträgen umfassen. Der gesamte Speicheraufwand ist jedoch weiterhin durch $O(n)$ beschränkt. Eine *membership*-Anfrage $x \in S'$ kann in der Zeit $O(1)$ durch Testen von

$$T[x[0,\ \alpha-1] \otimes T''[x[\alpha;\ \alpha+\beta-1] \otimes T'[x[0,\ \alpha-1]]]] \quad = \quad x$$

beantwortet werden. Um zu testen, ob ein Element in S ist (unsere ursprüngliche Fragestellung), genügen d voneinander abhängige Tests bzw. *membership*-Anfragen auf S'. Da d konstant ist, ist dieses auch in konstanter Zeit möglich.

Der Aufbau der Datenstruktur erfolgt in zwei Phasen. In der ersten Phase wird die Menge S' aus S bestimmt. Sortieren wir zunächst S – dieses ist mit dem Bucket-Sort in der Zeit $d \cdot n$ mit n Buckets und d Iterationen möglich – so können wir die Menge S' in Zeit $O(n)$ generieren. Auf dieses soll jedoch nicht weiter eingegangen werden. Vermerkt sei hier nur, dass die sortierte Liste von S alle Pfade im Baum in der Reihenfolge von links nach rechts gehend umfasst, und dass ein einmal verlassener Teilpfad des Baums durch eine nachfolgendes Element in S auch nicht mehr betreten wird.

In der zweiten Phase wird ein perfektes Hashverfahren auf S' realisiert. Der Zeitbedarf der einzelnen Schritte wurde von uns bereits bei der Präsentation dieses Verfahrens diskutiert. Die Bestimmung der einzelnen Permutationen konnte in der Zeit $O(|S'_i|)$ bzw. $O(|S''_i|)$ ausgeführt werden. Zusammenfassend ergibt dieses einen Zeitaufwand von $O(\sum_i |S'_i| + \sum_i |S''_i|) = O(n)$. Zudem mussten wir die Mengen einmal S''_i nach ihrer Größe sortieren. Dieses bedeutet jedoch nur einen zusätzlichen additiven Zeitaufwand von $O(n)$.

Zusammenfassend gilt:

Theorem 9 *Sei $S \subseteq U$ eine Menge der Größe n und $|U|$ polynomiell in n. Dann kann eine perfekte Hashfunktion $h : U \to \{0, \ldots, r-1\}$ für S und ein $r \in O(n)$ randomisierte in Zeit $O(n)$ generiert werden. Zum Speichern von h wird ein Platz von $\Theta(n)$ benötigt. Eine* membership*-Anfrage für S kann mit Hilfe von h in konstanter Zeit beantwortet werden.*

3.3.2 Die Derandomisierung

Wir wollen nun zeigen, dass die im oben vorgestellte Verfahren verwendeten Permutationsstrings b'_i und b''_i deterministisch in der Zeit $O(|S'_i| \log n)$ bzw. $O(|S''_i| \log n)$ generiert werden können. Somit ist die Laufzeit des derandomisierten Hashverfahren $O(n) + \sum_i O(|S'_i| \log n) + \sum_i O(|S''_i| \log n) = O(n \log n)$. Es folgt somit:

Theorem 10 *Sei $S \subseteq U$ eine Menge der Größe n und $|U|$ polynomiell in n. Dann kann eine perfekte Hashfunktion $h : U \to \{0, \ldots, r-1\}$ für S und ein $r \in O(n)$ deterministisch in der Zeit $O(n \log n)$ generiert werden. Zum Speichern von h wird ein Platz von $\Theta(n)$ benötigt. Eine* membership*-Anfrage für S kann mit Hilfe von h in konstanter Zeit beantwortet werden.*

Im Folgenden wollen wir nun die deterministische Berechnung eines Permutationsstrings b'_i beschreiben. Die Bestimmung einer Zeilenpermutation b''_i erfolgt analog.

Anstelle des oben angegebenen Problems, ein b'_i zu bestimmen, so dass R'_i unter einer vorgegebenen Schranke liegt, wollen wir diese Aufgabenstellung wie folgt abstrahieren:

> Gegeben sei eine Tabelle, welche wir mit Hilfe von Zeichenketten aus $\{0,1\}^\beta$ adressieren sowie eine Menge $A \subseteq \{0,1\}^\beta$. Generiere eine Zeichenkette $b \in \{0,1\}^\beta$, so dass
>
> $$\sum_{x \in A} T[x \otimes b] \quad \leq \quad \frac{|A|}{2^\beta} \cdot \sum_{i \in \{0,1\}^\beta} T[i]$$
>
> in der Zeit $O(\beta \cdot |A|)$.

Gibt $T[j]$ die Anzahl der Elemente an, welche vor der Bestimmung von b'_i in den Spalten $k < i$ in Zeile j stehen, so ist $T[x \otimes b'_i]$ die Anzahl der Zeilenkonflikte von x nach der Permutation. Gilt

$$\sum_{x \in A} T[x \otimes b] \quad \leq \quad \frac{|A|}{2^\beta} \cdot \sum_{i \in \{0,1\}^\beta} T[i] \ ,$$

wobei A der Menge S'_i entspricht, so ist die Summe der Konflikte nach der Permutation durch $\frac{|A| \cdot n'}{2^\beta}$ beschränkt.

Betrachten wir nun das oben beschriebene neue Problem.

Zur Bestimmung von b erweitern wir zunächst T, so dass T durch alle Binärzeichenketten der Länge kleiner gleich β indiziert werden kann. Für ein $\gamma < \beta$ und eine binäre Zeichenketten $i \in \{0,1\}^\gamma$ definiere

$$D_i \quad := \quad \{ \ j \in\in \{0,1\}^\beta \mid j[0;\ \gamma - 1] \ = \ i \ \}$$

sowie

$$T[i] \quad := \quad T[i0] + T[i1] \quad = \quad \sum_{j \in D_i} T[j] \ .$$

T kann also naheliegend als einen Binärbaum mit 2^β Blättern dargestellt werden, indem wir dem i-ten Blatt den Wert von $T[\mathrm{bin}(i-1, \beta)]$ zuweisen, wobei $\mathrm{bin}(i-1, \beta)$ die Binärdarstellung von $i-1$ der Länge β ist. Ein interner Knoten v repräsentiert einen Tabelleneintrag $T[i]$, dessen Wert der Summe der Werte seiner Blätter des durch v adressierten Unterbaumes entspricht. Diese Baumdarstellung ist in Abbildung 19 illustriert.

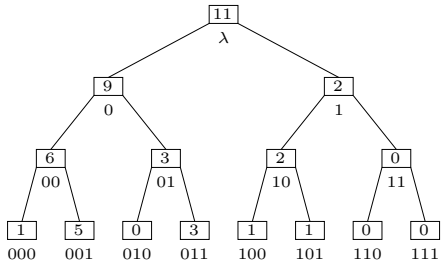


Abbildung 19: Repräsentation der erweiterten Tabelle T als Baum.

Die Konstruktion von b erfolgt sukzessiv Bit für Bit, indem wir die Folge der Zeichenketten $b_0 = \lambda, b_1, \ldots, b_\beta = b$ generieren. b_{i+1} entsteht dabei aus b_i, indem wir an b_i eine 0 bzw. eine 1 anhängen. Die Zeichenketten b_i ist somit ein String aus $\{0,1\}^i$. Für die jeweiligen Iterationsstufen soll hierbei gelten:

$$E_{d \in D_{b_i}}[\sum_{x \in A} T[x \otimes d]] \quad \leq \quad \frac{|A|}{2^\beta} \cdot \sum_{j \in \{0,1\}^\beta} T[j] \ . \tag{6}$$

Für $i = \beta$ gilt daher

$$E_{d \in D_{b_\beta}}[\sum_{x \in A} T[x \otimes d]] \quad = \quad \sum_{x \in A} T[x \otimes b_\beta] \quad \leq \quad \frac{|A|}{2^\beta} \cdot \sum_{j \in \{0,1\}^\beta} T[j] \ .$$

Womit b die erforderten Bedingungen erfüllt.

Betrachten wir zunächst einmal b_0, so gilt

$$\begin{aligned} E_{d\in D_{b_0}}[\sum_{x\in A} T[x\otimes d]] &= \frac{1}{|D_{b_0}|}\cdot\sum_{d\in D_{b_0}}\sum_{x\in A} T[x\otimes d] = \frac{1}{2^\beta}\cdot\sum_{x\in A}\sum_{d\in D_{b_0}} T[x\otimes d] \\ &= \frac{|A|}{2^\beta}\sum_{d\in\{0,1\}^\beta} T[x\otimes d] = \frac{|A|}{2^\beta}\sum_{i\in\{0,1\}^\beta} T[i]\ . \end{aligned}$$

Unter der Voraussetzung, dass die Ungleichung 6 für b_i erfüllt ist, wollen wir nun die Zeichenkette b_{i+1} konstruieren. Da $D_{b_i} = D_{b_i0}\cup D_{b_i1}$ sowie D_{b_i0} und D_{b_i1} disjunkt sind, folgt:

$$\begin{aligned} E_{d\in D_{b_i}}[\sum_{x\in A} T[x\otimes d]] &= \mathrm{Prob}_{d\in D_{b_i}}[d\in D_{b_i0}]\cdot E_{d\in D_{b_i0}}[\sum_{x\in A} T[x\otimes d]] \\ &\quad + \mathrm{Prob}_{d\in D_{b_i}}[d\in D_{b_i1}]\cdot E_{d\in D_{b_i1}}[\sum_{x\in A} T[x\otimes d]] \\ &= \frac{1}{2}\cdot\left(E_{d\in D_{b_i0}}[\sum_{x\in A} T[x\otimes d]] + E_{d\in D_{b_i1}}[\sum_{x\in A} T[x\otimes d]]\right)\ . \end{aligned}$$

Somit gilt für zumindest einen der Erwartungswerte auf der rechten Seite, d.h. für ein $\ell\in\{0,1\}$:

$$E_{d\in D_{b_i\ell}}[\sum_{x\in A} T[x\otimes d]] \leq \frac{|A|}{2^\beta}\cdot\sum_{j\in\{0,1\}^\beta} T[j]\ .$$

Gilt nun $E_{d\in D_{b_i0}}[\sum_{x\in A} T[x\otimes d]] \leq |A|\cdot 2^{-\beta}\cdot\sum_{j\in\{0,1\}^\beta} T[j]$, so wählen wir $b_{i+1} := b_i0$ und $b_{i+1} := b_i1$ ansonsten.

Es verbleibt somit noch zu zeigen, wie $E_{d\in D_{b_i0}}[\sum_{x\in A} T[x\otimes d]]$ für alle $i\leq\beta$ und $b_i\in\{0,1\}^i$ in Zeit $O(|A|)$ berechnet werden kann. Hierfür betrachten wir

$$\begin{aligned} E_{d\in D_{b_i}}[\sum_{x\in A} T[x\otimes d]] &= \sum_{x\in A} E_{d\in D_{b_i}}[T[x\otimes d]] = \sum_{x\in A}\frac{1}{2^{\beta-i}} T[x[0;\ i-1]\otimes b_i] \\ &= \frac{1}{2^{\beta-i}}\sum_{x\in A} T[x[0;\ i-1]\otimes b_i]\ . \end{aligned}$$

Mit Hilfe der erweiterten Tabelle T kann die Summe mit Hilfe von $|A|$ XOR-Operationen, $|A|$ Leseoperationen in T sowie $|A|-1$ Additionen berechnet werden. Der Zeitaufwand ist $O(|A|)$. Zur Berechnung des Erwartungswerts fehlt somit nur noch eine Division. Der Zeitaufwand zur Bestimmung einer Permutationzeichenkette ist daher $O(\beta\times|A|)$. Nach dem eine solche Permutationzeichenkette bestimmt worden ist, muss abschließend noch die Tabelle T modifiziert werden. Dieses erfordert weitere $O(\beta\times|A|)$ Schritte.

4 Das dynamische Wörterbuch

Eine Datenstruktur für ein dynamische Wörterbuch gestaltet aufgrund der Möglichkeit das Wörterbuch zu modifizieren um einiges komplexer als eine Datenstruktur für ein statisches Wörterbuch. Für ein dynamische, Wörterbuch benötigen wir neben der *membership*-Abfrage auch die Einfügeoperation *insert* und die Löschoperation *delete*.

4.1 Perfektes dynamisches Hashing

Das im Folgenden vorgestellte Verfahren ist aus der Arbeit [DKMM94] entnommen. Die diesem Verfahren zugrunde liegende Idee ist mit unserer Überlegung zu der dynamischen und speichereffizienten Realisierung eines Stacks (siehe Kapitel 2) verwandt. Wir arbeiten auf einer gegebenen Hashtabelle, bis diese zu einem bestimmten Grad gefüllt ist. Wird ein gegebener Threshold überschritten, so allokieren wir eine neue Tabelle der doppelten Größe. Wird eine gegebene untere Schranke unterschritten, so verwenden wir im weiteren eine Hashtabelle der halben Größe. Wir erreichen somit, dass die Hashtabelle immer linear in der Anzahl der zu speichernden Daten ist. Ein Problem ergibt sich allerdings, wenn es zu Konflikten bei der *insert*-Operation kommt. Um in diesem Fall nicht jedes mal eine neue Hashfunktion für die gesamte Menge zu generieren, wird im Folgenden wieder auf das doppelte Hashing zurückgegriffen. Wir erhalten eine Toplevel-Hashtabelle T der Größe r, deren Einträge Pointer auf Secondary-Hashtabelle $T_1, \ldots, T_r$ der Größe $r_1, \ldots, r_r$ sind.

Benutzen wir als Hashfunktionen wieder Funktionen einer universellen Familie, so folgt aus der Definition unmittelbar, dass die erwartete Anzahl von Kollisionen an einer beliebigen Toplevel-Position für $r \geq n$ in $O(1)$ liegt. Daher können wir *membership* sowie *insert* und *delete* in erwartet konstanter Zeit lösen.

Im worst-case hängt die Laufzeit einer *membership*-Anfrage bei linearen Secondary-Tabelle jedoch von dem maximalen r_i ab. Diese Situation ähnelt der Situation n Bälle in r Eimer zu werfen. Ist $r = n$, so ist die erwartete maximale Anzahl von Bällen in einem Eimer in $\Theta(\log n)$, für $r = 2n$ in $\Theta(\log n / \log\log n)$ und für $r = \Omega(n^2)$ ist mit sehr großer Wahrscheinlichkeit die maximale Anzahl der Bälle pro Eimer 1.

Um die Länge der Toplevel-Hashtabelle linear in n zu halten, speichern wir die Elemente in den Secondary-Tabelle mit Hilfe weiterer Hashfunktionen. Um ein dynamisches Anwachsen der einzelnen Tabellen zu erlauben, wählen wir die Toplevel-Hashtabelle doppelt so groß wie die Toplevel-Hashtabelle im statischen Fall (siehe Kapitel 3.2.4).

In Kapitel 3.2.4 haben wir gesehen, dass für eine universelle Familie $\mathcal{H}_{r(n)}$ von Hashfunktionen $h : U \to \{0, \ldots, r(n)-1\}$ mit $r(n) \in \Theta(n)$ ein $b(n) \in O(n)$ existiert, so dass für jede n elementige Teilmenge $S \subseteq U$ gilt

$$\text{Prob}_{h \in \mathcal{H}_{r(n)}} \left[\sum_{i=0}^{r(n)-1} (N_k(h))^2 \quad < \quad b(n) \right] \quad \geq \quad \frac{1}{2},$$

wobei $N_i(h) = |\{x \in S | h(x) = i\}|$ ist. Zudem existiert eine Konstante c, so dass

$$\text{Prob}_{h \in \mathcal{H}_{c \cdot n^2}} [h \text{ ist eine perfekte Hashfunktion für } S] \quad \geq \quad \frac{1}{2}$$

ist. Wie oben schon angedeutet, wählen wir die Hashfunktion so, dass ihre *Kapazität* auch für eine größere Menge S ausreicht. Anstelle des Parameters n benutzen wir daher zur Bestimmung der Tabellengröße $\overline{n} := 2 \cdots n$. Die Hashfunktionen der i-ten Untertabelle wird analog nicht bezüglich $N_i(h)$ sondern bezüglich ihrer *Kapazität* von $\overline{n}_i := 2 \cdots N_i(h)$ bestimmt.

Im Folgenden wollen wir nun zunächst auf das Generieren einer derartigen dynamischen Zwei-Level Hashtabelle eingehen.

4.1.1 Generieren einer dynamischen Zwei-Level Hashtabelle

Zur Bestimmung der Toplevel-Hashfunktion wählen wir uniform randomisierte Funktionen $h \in \mathcal{H}_{r(\overline{n})}$ und bestimmen $h(x)$ für alle $x \in S$. Wir beenden diesen Prozess sobald eine Hashfunktion gefunden

wurde, die

$$\sum_{i=0}^{r(\overline{n})} (N_i(h))^2 \quad \leq \quad b(\overline{n})$$

erfüllt. Da $n \leq \overline{n}$ ist, finden wir eine solche Hashfunktion im Erwartungswert in einer konstanten Anzahl von Zügen. Zur Auswertung der obigen Ungleichung benötigen wir pro Hashfunktion $O(n)$ Schritte. Somit können wir im Erwartungswert eine entsprechende Hashfunktion in Zeit $O(n)$ finden.

Analog suchen wir in einem zweiten Schritt eine perfekte Hashfunktion $h_i \in \mathcal{H}_{c \cdot \overline{n}_i}$ für jede der Secondary-Tabellen T_i bezüglich der Mengen $\{x \in S | h(x) = i\}$ jedoch mit einer Kapazität für $\overline{n}_i$-elementige Mengen. Wie oben gesehen, ist für eine entsprechende Wahl von c eine zufällig gewählte Hashfunktion $h_i \in \mathcal{H}_{c \cdot \overline{n}_i}$ perfekt für $\{x \in S | h(x) = i\}$. Somit kann eine solche Hashfunktion in erwarteter Zeit $O(N_i(h))$ ermittelt werden.

Für die erwartete Gesamtlaufzeit erhalten wir

$$O(n) \; + \; \sum_{i=0}^{r(\overline{n})-1} c \cdot \overline{n}_i \quad = \quad O(n) \; .$$

Um eine konstante amortisierte Zeit zu erhalten, dürfen wir folglich die Hashtabellen nicht zu häufig neu generieren. Zu den Tabellen speichern wir noch die Größe der aktuellen Mengen S und $\{x \in S | h(x) = i\}$ für alle $i \in \{0, \ldots, r(\overline{n}) - 1\}$ sowie die Werte von $\overline{n}, \overline{n}_0, \ldots, \overline{n}_{r(\overline{n})-1}$. Die Speichergröße ist durch

$$O(r(\overline{n})) \; + \; \sum_{i=0}^{r(\overline{n})-1} c \cdot \overline{n}_i^2 \quad \leq \quad O(r(\overline{n})) + 4 \cdot c \cdot b(\overline{n}) \quad \in \quad O(n)$$

beschränkt.

4.1.2 *delete* und *insert* in dynamischen Zwei-Level Hashtabelle

Im Folgenden sollen zwei Funktionen vorgestellt werden, mit deren Hilfe man die *delete*- und *insert*-Funktion effizient realisieren kann.

procedure delete(x)
1 `if` $T_{h(x)}(h_{h(x)}(x)) = x$ `then`
2 n:=n-1
3 `if` $n < \frac{\overline{n}}{4}$
4 `then` Generiere eine neue Hashtabelle für die gesamte Menge S ohne x
5 `else` i:=h(x); lösche den Eintrag $T_i(h_i(x))$; $n_i := n_i - 1$

```
procedure insert(x)
1    if  T_{h(x)}(h_{h(x)}(x)) ≠ x then
2        n:=n+1
3        if  n > n̄
4           then Generiere eine neue Hashtabelle für die gesamte Menge S zuzüglich x
5                return
6           else i := h(x); n_i := n_i + 1;
7                if T_i(h_i(x)) leer then T_i(h_i(x)) := x;  return;
8                if n_i > n̄_i
9                   then n̄_i := 2 · n̄_i
10                       if Σ_{j=0}^{r(n̄)-1} n̄_i > 4b(n̄)
11                          then Generiere eine neue Hashtabelle für ganz S zuzüglich x
12                               return
13               Generiere eine neue Hashtabelle für {x} ∪ {y ∈ S|h(y) = i} der Kapazität n̄_i
```

Zum Generieren einer neuen Hashtabelle aus einer alten Hashtabelle müssen wir neben dem Auswerten der Hashfunktion an sich auch noch die Daten aus der alten Hashtabelle aufsammeln. Dieses können wir jedoch dadurch vereinfachen, dass wir die Elemente aus S mit Hilfe mehrerer über Pointer verketteter Listen untereinander verknüpfen.

Um den zusätzlichen Speicher zur Verfügung zu stellen, den wir bei einer Verdoppelung der Kapazität einer Secondary-Tabellen T_i benötigen, allokieren wir beim Aufbau der Toplevel-Hashtabelle $\Theta(n)$ zusätzliche Speicherzellen.

4.1.3 Platz- und Zeitanalyse

Unter einer *Phase* verstehen wir das Zeitintervall zwischen dem Ende einer Neugenerierung einer Hashtabelle für die gesamte Menge S und dem Beginn der nächsten Neugenerierung. Im Folgenden wollen wir zunächst den benötigten Platz untersuchen: Im Verlauf einer Phase gilt $\overline{n}/4 \leq |S| \leq \overline{n}$. Die Größe der Toplevel-Hashtabelle ist somit in $O(r(\overline{n})) = O(r(|S|)) = O(|S|)$. Sei nun $\overline{n}'_i$ die Kapazität der Secondary-Tabellen T_i am Ende der untersuchten Phase, dann benötigen wir für T_i, sofern T_i in dieser Phase j-mal neu aufgebaut werden musste, zu Beginn der Phase $c \cdot (\overline{n}'_i/2^j)^2$ Speicherzellen. Allgemein wird für T_i ein Speicher der Größe

$$c \cdot (\overline{n}'_i/2^j)^2 \quad + \quad c \cdot (\overline{n}'_i/2^{j-1})^2 \quad + \quad \cdots \quad + \quad c \cdot (\overline{n}'_i/2^0)^2 \quad < \quad 2 \cdot c \cdot (\overline{n}'_i)^2$$

benötigt. Da die Werte von $\overline{n}_i$ innerhalb einer Phase monoton steigen (sie werden bei einem *delete* nicht verkleinert), ist der gesamte Speicherbedarf für alle Secondary-Tabellen durch

$$\sum_{i=0}^{r(\overline{n})-1} 2 \cdot c \cdot (\overline{n}'_i)^2 \quad \leq \quad 2 \cdot c \cdot \sum_{i=0}^{r(\overline{n})-1} (\overline{n}'_i)^2 \quad \leq \quad 2 \cdot c \cdot 4 \cdot b(\overline{n}) \quad \in \quad O(|S|)$$

beschränkt. Zusammenfassend können wir schließen, dass der Speicherbedarf innerhalb einer Phase linear in der Größe der aktuellen Menge S ist.

Bei der Zeitanalyse wollen wir zunächst einmal die *delete*-Operation betrachten. Ist die Größe von S größer als $\overline{n}/4$, so können wir ein Element aus S in konstanter Zeit löschen. Wird die Schranke $\overline{n}/4$ unterschritten – dieses kann jedoch frühstens nach $\overline{n}/4$ *delete*-Operationen geschehen – so benötigen wir für die Neugenerierung der gesamten Datenstruktur $O(|S|)$ Schritte. Verteilen wir diese Kosten – beispielsweise über die Bilanzmethode – auf die vorhergehenden $\overline{n}/4$ *delete*-Operationen, so erhalten wir amortisiert für jede Löschoperation einen konstanten Zeitbedarf.

Wegen der möglicherweisen Neugenerierung der Secondary-Tabellen erweist sich die Analyse der amortisierten Zeit von *insert*-Operationen als aufwendiger als die Analyse der *delete*-Operationen.

Der Einfachheit halber gehen wir davon aus, dass Zeile 4 der Prozedur *insert* alle $\overline{n}/2$ *insert*-Operationen ausgeführt wird. Man beachte, dass dieses aufgrund von *delete*-Operationen oder wegen des vorzeitigen Neubaus der ganzen Hashtabelle nicht unbedingt der Fall sein muss. Ein solcher Neubau stellt für den amortisierten Zeitbedarf kein großes Hindernis dar; er verschlechtert nur die Konstante. Die Analyse erfolgt für diesen Neubau analog zu der *delete*-Operation.

Im Folgenden wollen wir uns auf die Analyse der Neugenerierungen in Zeile 11 und 13 konzentrieren. Zu Beginn jeder Phase beinhaltet die betrachtete Menge S nur $\overline{n}/2$ Elemente, die Hashfunktion wurde jedoch uniform aus $\mathcal{H}_{r(\overline{n})}$ gezogen, folglich sollte mit der Wahrscheinlichkeit von zumindest $\frac{1}{2}$ auch nach dem Hinzufügen von $\overline{n}/2$ weiteren Elementen zu S

$$\sum_{i=0}^{r(\overline{n})} \left(N_i(h)\right)^2 \quad \leq \quad b(\overline{n})$$

sein. Somit ist die Anzahl der Neugenerierungen der gesamten Hashtabellen zwischen zwei regulären Neugenerierungen durch $O(1)$ beschränkt. Da die Kosten für eine Neugenerierung linear in n sind, ist der zu erwartende Gesamtaufwand für Neugenerierung der gesamten Hashtabellen zwischen zwei regulären Neugenerierungen in $O(n)$. Da zwischen zwei regulären Neugenerierungen $\overline{n}/2$ *insert*-Operationen liegen, sind die amortisierten Zeitkosten wiederum konstant.

Es verbleibt somit die Analyse der amortisierten Zeitkosten, welche sich durch die zusätzlichen Neugenerierungen der Secondary-Tabellen in Zeile 13 ergeben. Diese Zeile gelangt zur Ausführung, wenn entweder ein Konflikt in einer Secondary-Tabelle entsteht oder die Kapazität einer Secondary-Tabelle überschritten wird.

Man beachte, dass die Anzahl der Elemente im i-ten Bucket zu Beginn einer Phase $\overline{n}_i/2$ ist, und da wir h_i uniform aus $\mathcal{H}_{r_i(\overline{n}_i)}$ gewählt haben, ist diese Hashfunktion mit einer Wahrscheinlichkeit von zumindest $\frac{1}{2}$ eine perfekt Hashfunktion für dieses Bucket, auch wenn wir weitere $\overline{n}_i/2$ Elemente zu diesem Bucket hinzufügen. Die erwartete Anzahl von Neugenerierungen dieser Hashtabelle für die folgenden $\overline{n}_i/2$ *insert*-Operationen zu diesem Bucket ist somit 2. Da die Kosten für die Neugenerierung einer Secondary-Tabelle für das i-te Bucket $O(\overline{n}_i)$ sind, sind die amortisierten Kosten, welche durch einen Konflikt entstehen, konstant.

Wir müssen um die Kapazität der i-ten Secondary-Tabelle zu überschreiten $\overline{n}_i/2$ Elemente dem i-ten Bucket hinzufügen. Die hierauf folgende Neugenerierung der Secondary-Tabelle benötigt wiederum $O(2 \cdot \overline{n}_i) = O(\overline{n}_i) = O(n_i)$ Schritte. Da dieser Neugenerierung jedoch $\overline{n}_i/2$ *insert*-Operationen vorweg gehen, können wir über diese Operationen ein hinreichend großes Potential ansparen, so dass die amortisierten Kosten wiederum konstant sind.

4.2 Dynamisierung statischer Datenstrukturen

Dynamisierung ist eine generelle Technik, um aus einer statischen Datenstruktur eine dynamische zu erstellen. Da dieses eine Technik nicht nur für Wörterbücher ist, müssen wir allgemeine Anfragen an die Datenstruktur erlauben. Wir definieren daher für ein Universum U, eine Teilmenge $S \subseteq U$ und ein Element $x \in U$ die Anfragefunktion $Q(x, S)$ über

$$Q \quad : \quad U \quad \times \quad 2^U \quad \rightarrow \quad R \,,$$

wobei R die Menge der möglichen Antworten ist. Beispiele solcher Antwortmengen sind:

- für die *membership*-Anfrage: $R := \{\texttt{true}, \texttt{false}\}$, wobei

$$Q(x,S) \quad := \quad \begin{cases} \texttt{true} & \text{für } x \in S \\ \texttt{false} & \text{für } x \notin S \ . \end{cases}$$

- für die Vorgängerfunktion: $R := U \cup \{-\infty\}$. Für $U \subseteq \mathbb{N}$ gilt:

$$Q(x,S) \quad := \quad \begin{cases} \max\{\ y \in S \mid y < x\ \} & \text{für } x > \min S \\ -\infty & \text{für } x \leq \min S \ . \end{cases}$$

- für die *membership*-Anfrage bei der konvexen Hülle eine Punktmenge in der Ebene.

Die oberen Beispiele unterscheiden sich in einem wesentlichen Punkt von dem letzten Beispiel mit der konvexen Hülle; sie sind in einfachere Teilprobleme zerlegbar. Wir nennen eine Anfrage Q *zerlegbar*, wenn eine binäre Zusammensetzungsfunktion $f : R \times R \to R$ existiert, so dass für alle $S \subseteq U$, alle disjunkten Teilmengen $A, B \subseteq S$ mit $S = A \cup B$ und alle $x \in U$ gilt:

$$Q(x,S) \quad := \quad f(Q(x,A), Q(x,B)) \ .$$

Für die einfache *membership*-Anfrage ist diese Funktion f die Oderfunktion und für die Vorgängerfunktion das Maximum. Die *membership*-Anfrage bei der konvexen Hülle ist wie oben schon angedeutet nicht zerlegbar.

Wir werden im Folgenden zeigen, wie eine *effiziente* dynamische Datenstruktur aus einer statischen Datenstruktur generiert werden kann, wenn die benötigte Anfragefunktion zerlegbar ist.

4.2.1 Eine semidynamische Datenstruktur: Insertions Only

Wir beginnen unsere Analyse mit der Konstruktion einer semidynamischen Datenstruktur, welche nur die *insert*-Operation und nicht die *delete*-Operation zur Verfügung hat.

Theorem 11 *Sei Q eine zerlegbare Anfrage, für welche eine statische Datenstruktur mit Anfragezeit $q(n)$, Speicherplatzbedarf $s(n)$, Konstruktionszeit $c(n)$ und konstanter Zusammensetzungszeit existiert, wobei $q(n)$, $s(n)/n$ und $c(n)/n$ monoton wachsende Funktionen sind. Dann existiert eine semidynamische Datenstruktur für Q mit der Anfragezeit $q'(n) \in O(q(n) \cdot \log n)$, dem Speicherplatzbedarf $s'(n) \in O(s(n))$ und dem Zeitbedarf für eine* insert*-Operation von $t'(n) \in O(\frac{c(n) \cdot \log n}{n})$.*

Um S in einer semidynamischen Datenstruktur für Q zu speichern, zerlegen wir S in eine Folge von $\lfloor \log_2 n \rfloor + 1$ disjunkter Teilmengen $S = S_0 \cup S_i \cup \ldots \cup S_{\lfloor \log_2 n \rfloor}$, wobei für alle $i \in \{0, \ldots, \lfloor \log_2 n \rfloor\}$

$$|S_i| \quad = \quad \begin{cases} 2^i, & \text{wenn das } (i+1)\text{-te niederwertigste Bit der Binärdarstellung von } |S| \text{ 1 ist und} \\ 0, & \text{wenn das } (i+1)\text{-te niederwertigste Bit der Binärdarstellung von } |S| \text{ 0 ist.} \end{cases}$$

Die einzelnen Teilmengen S_i werden mit Hilfe einer statischen Datenstruktur verwaltet. Die genaue Zugehörigkeit eines Elements $x \in S$ zu einer Teilmenge S_i ergibt sich aus dem Zeitpunkt, an dem x zu der Menge S hinzugefügt wurde. Diese Zugehörigkeit wird aus der Betrachtung des Zeitbedarfs für eine *insert*-Operation, welche weiter unten folgen wird, deutlich.

Wir wollen zunächst die **amortisierten Kosten** einer solchen Datenstruktur analysieren. Später werden wir aus dieser vereinfachten Datenstruktur eine etwas aufwendigere zweite Datenstruktur entwickeln, welche auch ein effizientes worst-case Verhalten aufweist.

Speichern wir die oben vorgestellten Teilmengen mit Hilfe der gegebenen statischen Datenstruktur, so gilt für den Platzbedarf der semidynamischen Datenstruktur

$$\begin{aligned} s'(n) &= \sum_{i \text{ mit } b_i=1 \text{ und } |S|=b_{\lfloor \log_2 n \rfloor} \cdots b_1 b_0} s(2^i) \\ &\leq \sum_{i=0}^{\lfloor \log_2 n \rfloor} \frac{s(2^i)}{2^i} \cdot 2^i \leq \sum_{i=0}^{\lfloor \log_2 n \rfloor} \frac{s(n)}{n} \cdot 2^i < \frac{s(n)}{n} \cdot 2^{\lfloor \log_2 n \rfloor + 1} \in O(s(n)) \ , \end{aligned}$$

wobei die zweite Ungleichung aus der Annahme folgt, dass $\frac{s(n)}{n}$ keine fallende Funktion ist.

Um eine Anfrage Q an die Datenstruktur zu beantworten, wird eine Zeit von

$$\begin{aligned} q'(n) &= \sum_{i=0}^{\lfloor \log_2 n \rfloor} (q(|S_i|) + \text{Zeit zum Zusammensetzen zweier Resultate}) \\ &\leq \sum_{i=0}^{\lfloor \log_2 n \rfloor} (q(n) + O(1)) \in O(q(n) \cdot \log n) \ . \end{aligned}$$

Es verbleibt somit die Analyse der amortisierten Kosten für eine *insert*-Operation.

Bei einer *insert*-Operation eines Elements $x \in U \setminus S$ vereinigen wir die Teilmengen $S_0, \ldots, S_i$ von S mit $\{x\}$ zu einer neuen Menge S_{i+1}, wobei von dieser *insert*-Operation für diese Teilmengen gilt: $|S_j| = 2^j$ für alle $j \leq i$ und $|S_{i+1}| = 0$. Nach der *insert*-Operation gilt daher $|S_j| = 0$ für alle $j \leq i$ und $|S_{i+1}| = 2^{i+1}$. Die Kosten einer derartigen Operation sind daher durch $c(2^{i+1})$ beschränkt, sofern die Elemente aus $S_0 \cup S_1 \cup S_i$ in linearer Zeit aufgesammelt werden können. Dieses kann beispielsweise über eine mit Hilfe von Pointen verknüpfte Liste geschehen, über die die Elemente aus $S_0 \cup S_1 \cup S_i$ zusätzlich verbunden sind.

Um die amortisierte Zeit zu erhalten, betrachten wir eine Sequenz σ_n von n *insert*-Operationen. Man beachte, dass die Hälfte dieser *insert*-Operationen in der Zeit $c(1)$, ein Viertel in der Zeit $c(2)$, ein Achtel in der Zeit $c(4)$ oder allgemeiner ein Anteil von 2^{-i-1} der n Operationen in Zeit $c(2^i)$ ausgeführt werden können. Für die ganze Sequenz ergibt sich daher ein Zeitbedarf von

$$T(\sigma_n) = \sum_{i=0}^{\lfloor \log n \rfloor} \frac{c(2^i) \cdot n}{2^{i+1}} \leq \frac{n}{2} \cdot \sum_{i=0}^{\lfloor \log n \rfloor} \frac{c(2^i)}{2^i} \leq \frac{n}{2} \cdot \sum_{i=0}^{\lfloor \log n \rfloor} \frac{c(n)}{n} \in O(c(n) \cdot \log n) \ .$$

Für die amortisierte Zeit erhalten wir daher

$$t'_{\text{amort}}(n) \in O\left(\frac{c(n) \cdot \log n}{n}\right) \ .$$

Als Beispiel für eine Dynamisierung einer statischen Datenstruktur betrachten wir die Repräsentation einer Menge mit Hilfe einer sortierten Liste. Für den benötigten Speicherplatz gilt dann $s(n) \in O(n)$, für die Beantwortungszeit einer *membership*-Anfrage $q(n) \in O(\log n)$ und für die Konstruktionszeit $c(n) \in O(n \log n)$. Für die semidynamische Datenstruktur gilt daher ein Speicherplatzbedarf von $s'(n) \in O(n)$, eine Beantwortungszeit von $q'(n) \in O(\log^2 n)$ und für die amortisierte Zeit für eine *insert*-Operation $t'_{\text{amort}}(n) \in O(\log^2 n)$. Die amortisierte Zeit für eine *insert*-Operation kann noch auf $t'_{\text{amort}}(n) \in O(\log n)$ verbessert werden, wenn wir an Stelle der vollständigen Neukonstruktion durch eine Neusortierung von $S_0 \cup S_1 \cup S_i$ die einzelnen Listen durch Merge-Operationen zusammenführen.

Im Folgenden soll nun die oben vorgestellte Strategie so abgewandelt werden, dass eine *insert*-Operation auch im **worst-case** in der Zeit $O\left(\frac{c(n) \cdot \log n}{n}\right)$ ausgeführt werden kann. Hierzu verteilen wir die Kosten, welche durch das Zusammenfügen der einzelnen Teilmengen entstehen über die Zeit zwischen

zwei Zusammenfügeoperationen der gleichen Tiefe. Da wir jedoch nicht wie bei der Betrachtung der amortisierten Kosten eine Art Potential aufbauen und dann später bei Bedarf nutzen können, müssen wir das Zusammenfügen an sich über mehrere *insert*-Operationen ausführen. Wie zuvor teilen wir S wieder in Teilmengen der Größe $1, 2, 4, \ldots$. Im Unterschied zu unserer obigen Strategie benutzen wir jedoch eine Folge disjunkter Mengen $S_{i,1}, S_{i,2}, S_{i,3}, \ldots$ jeder Größe 2^i. Für S gilt somit

$$S \;=\; \bigcup_{i=0}^{\lfloor \log_2 n \rfloor} \bigcup_{j} S_{i,j} \;.$$

Wie zuvor benötigen wir eine statische Datenstruktur für jede nicht leere Teilmenge $S_{i,j}$. Zudem benutzen wir für jedes i eine Hilfsdatenstruktur für eine Menge S'_i der Größe 2^{i+1}, welche eine Menge $S_{i+1,j}$ *im Aufbau* darstellt. Im Folgenden werden wir noch zeigen, dass immer drei aktive Mengen $S_{i,1}, S_{i,2}, S_{i,3}$ für jede Größe 2^i ausreichen.

Es ist nun einfach zu sehen, dass wie im obigen Fall der Speicherplatz $s'(n)$ in $O(s(n))$ und der Zeitbedarf $q'(n)$ für eine Anfrage $Q(x, S)$ in $O(q(n) \cdot \log n)$ liegt. Bei der Analyse müssen wir hier nur darauf achten, dass wir für jedes i bis zu vier Teilmengen der Größe 2^i speichern und bis zu drei Teilmengen dieser Größe durchsuchen müssen. Wie wir jetzt gleich sehen werden, müssen wir die jeweiligen *Teilmengen im Aufbau* zur Beantwortung einer Anfrage $Q(x, S)$ nicht durchsuchen.

Um den Zeitbedarf einer *insert*-Operation zu analysieren, müssen wir den Aufbau der Datenstruktur und damit die Arbeitsweise der *insert*-Operation näher beschreiben. Die *insert*-Operation fügt das neue Element x in die Menge $S_{0,j}$ mit dem kleinsten Index j ein, welche leer ist. Analog identifizieren wir die Teilmenge S'_i, sobald die dazugehörige Datenstruktur fertig gestellt wurde, mit der Menge $S_{i+1,j}$ mit dem kleinsten Index j, für den $S_{i+1,j}$ noch leer ist:

$$S_{i+1,j} \;:=\; S'_i \qquad \text{und} \qquad S'_i \;:=\; \emptyset \;.$$

Mit der Konstruktion einer Datenstruktur für S'_i beginnen wir jedes Mal, wenn die Mengen $S_{i,1}$ und $S_{i,2}$ voll sind. Sobald die Menge S'_i fertig aufgebaut wurde, und deren Identifizierung mit einer Menge $S_{i+1,j}$ erfolgt ist, führen wir ferner die Operationen (bzw. Identifizierungen)

$$S_{i,1} \;:=\; S_{i,3}, \qquad \forall j > 3 \;:\; S_{i,j-1} \;:=\; S_{i,j} \qquad \text{und} \qquad S_{i,2} \;:=\; \emptyset$$

aus. Um eine Menge S'_i zu generieren, verteilen wir die Operationen um die zwei Mengen $S_{i,1}$ und $S_{i,2}$ auf 2^{i+1} *insert*-Operationen. Jede dieser Operationen hat somit einen anteiligen Berechnungsaufwand von $\frac{c(2^{i+1})}{2^{i+1}}$ Schritten dieser Zusammenfassungsoperation zu tragen. Da eine einzelne *insert*-Operation möglicherweise für jede der Teilmengen im Aufbau $S'_0, S'_1, S'_2, \ldots$ einige der Zusammenfassungsoperationszyklen ausführen muss (die Bearbeitung der einzelnen Zusammenfassungsoperatoren erfolgt hierbei von S'_0 nach $S'_{\lfloor \log n \rfloor}$), ergibt sich eine Gesamtlaufzeit für eine einzelne *insert*-Operation von

$$t'(n) \;\leq\; c(1) + \sum_{i=0}^{\lfloor \log n \rfloor - 1} \frac{c(2^{i+1})}{2^{i+1}} \;\leq\; \sum_{i=0}^{\lfloor \log n \rfloor} \frac{c(n)}{n} \;\in\; O\left(\frac{c(n) \cdot \log n}{n}\right) \;.$$

Hierbei haben wir bereits vorausgesetzt, dass keine Menge $S_{i,j}$ mit $j > 3$ benötigt wird. Der Beweis für die Korrektheit dieser Voraussetzung erfolgt in Lemma 2.

Ein Beispiel für die Funktionsweise der *insert*-Operation ist in Tabelle 1 illustriert. Wir werden hier wieder auf das Beispiel der sortierten Listen zurückgreifen, wobei wir für die Zusammenfassung zweier Teilmengen $S_{i,1}$ und $S_{i,2}$ die Merge-Operation verwenden, da dieses die Darstellung verständlicher macht.

Es verbleibt noch der Beweis, dass für jede Größe 2^i nur drei Teilmengen $S_{i,j}$ erforderlich sind.

Schritt	Element	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$	S'_0	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$	S'_1
$t=1$	2	[2]							
$t=2$	4	[2]	[4]		[2, −]				
$t=3$	1	[2]	[4]	[1]	[2, 4]				
		[1]				[2, 4]			
$t=4$	3	[1]	[3]		[1, −]	[2, 4]			
$t=5$	7	[1]	[3]	[7]	[1, 3]	[2, 4]			
		[7]				[2, 4]	[1, 3]		[1, −, −, −]

Tabelle 1: Eine Folge von 5 *insert*-Operationen 2, 4, 1, 3, 7.

Lemma 2 *Bei dem oben vorgestellten Verfahren sind für jede Größe* 2^i *nie mehr als drei aktive Mengen nötig.*

Beweis: Der Beweis erfolgt über eine Widerspruchsannahme. Sei die ℓ_1-te *insert*-Operation die erste Operation, bei der unser obiges Verfahren vier Teilmengen der Größe 2^i generiert. Zudem sei die ℓ_0-te *insert*-Operation die letzte Operation vor der die ℓ_1-te *insert*-Operation, bei der eine Menge $S_{i,2}$ generiert wird. Diese erfolgt durch eine Identifizierung:

$$S_{i,2} \;:=\; S'_{i-1} \qquad \text{gefolgt von} \qquad S'_{i-1} := \emptyset \;.$$

Um die dritte Menge $S_{i,3}$ zu generieren, müssen wir zunächst S'_{i-1} füllen, für welches wir 2^i *insert*-Operationen benötigen. Die Identifizierungen

$$S_{i,3} \;:=\; S'_{i-1} \qquad \text{gefolgt von} \qquad S'_{i-1} := \emptyset$$

erfolgen somit frühstens im Anschluss zu der $(\ell_0 + 2^i)$-ten *insert*-Operation. Um die vierte Teilmenge $S_{i,4}$ zu füllen, benötigen wir wiederum zumindest 2^i *insert*-Operationen. Somit gilt $\ell_1 \geq \ell_0 + 2^{i+1}$. Man beachte, dass bei der ℓ_0-ten, der $(\ell_0 + 2^i)$-ten und der $(\ell_0 + 2^{i+1})$-ten *insert*-Operation kein neuer Aufbau der Menge S'_{i-1} begonnen wird.

Der Algorithmus beginnt mit der Generierung der Menge S'_i, sobald die beiden Mengen $S_{i,1}$ und $S_{i,2}$ voll sind, also mit der ℓ_0-te *insert*-Operation. Da die Fertigstellung von S'_i genau 2^{i+1} *insert*-Operationen benötigt (die ℓ_0-te Operation wird hierbei mitgerechnet), erfolgt mit der $(\ell_0 + 2^{i+1} - 1)$-ten *insert*-Operation auch die Zuweisung

$$S_{i,1} \;:=\; S_{i,3} \qquad \text{gefolgt von} \qquad S_{i,2} := \emptyset \;.$$

Da die ℓ_0-te *insert*-Operation die letzte Operation vor der ℓ_1-ten *insert*-Operation ist, bei der eine Menge $S_{i,2}$ generiert wird, muss folglich $\ell_1 \leq \ell_0 + 2^{i+1} - 1$ sein – ein Widerspruch. ∎

4.2.2 Eine semidynamische Datenstruktur: Deletions Only

Die *delete*-Operation unterscheidet sich in einem Punkt wesentlich von der *insert*-Operation: Beim Einfügen eines Elements spielt es Dank der Möglichkeit Ergebnisse von Anfrageoperationen zu kombinieren, keine Rolle zu welcher Teilmenge wir ein Element hinzufügen. Die *delete*-Operation entfernt hingegen ein bestimmtes Element aus einer bestimmten Menge. Wir wollen daher hier nur die wesentlichen Ideen beschreiben, welche ein effizientes amortisiertes Zeitverhalten einer dynamisierten Datenstruktur mit der *delete*-Operation garantiert.

Anstelle ein Element bei einer *delete*-Operation zu löschen, markieren wir dieses Element nur als gelöscht. Somit ist der Speicherplatzbedarf solange asymptotisch gleich dem Speicherplatzbedarf der statischen Datenstruktur, solange wir in der Datenstruktur einen konstanten Anteil der Elemente der ursprünglichen Menge speichern. Ist der Zeitbedarf zum Beantworten einer Anfrage Q höchstens polynomiell in der Größe der Menge S, so ist auch der Zeitbedarf zur Berechnung der Antwort auf eine Sache Frage asymptotisch gleich dem entsprechenden Zeitbedarf für die statischen Datenstruktur.

Damit diese Eigenschaften erhalten bleiben, generieren wir immer dann eine neue Datenstruktur für S, wenn die Hälfte aller Elemente der ursprünglichen Menge gelöscht wurden. Es gilt somit

$$s'(n) \in O(s(n)), \qquad q'(n) \in O(q(n)) \qquad \text{und} \qquad t'_{\text{amort}}(n) \in O(c(n)/n + d(n)) ,$$

wobei $d(n)$ die Zeit angibt, in welcher das Element aus S, welches gelöscht werden soll, gefunden werden kann. Wir gehen hierbei wieder davon aus, dass wir die einzelnen Elemente der Menge zusätzlich zu der Datenstruktur noch über eine doppelte Pointerkette verbunden haben. Im Fall der sortierten Liste bedeutet dieses

$$s'(n) \in O(n), \qquad q'(n) \in O(\log n) \qquad \text{und} \qquad t'_{\text{amort}}(n) \in O(\log n) .$$

4.2.3 Deletions und Insertions: eine amortisierte Analyse

Wir wollen nun untersuchen, wie wir beide Operationen Deletions und Insertions kombinieren können. Betrachten wir zunächst eine statische Datenstruktur für eine Menge S der Größe n. Sei $d(n)$ die Zeit, die benötigt wird, um ein Element aus S zu löschen, ohne das dabei der von der Datenstruktur benötigte Speicherplatz oder die Zeit, die wir für eine Einfügeoperation oder zur Beantwortung einer *membership*-Anfrage benötigen, vergrößert wird. Eine Löschoperation aus einem Array, wo von dem zu löschenden Element die Position des Eintrags bekannt ist, können wir in $O(1)$ Schritten ausführen, indem wir den Eintag als gelöscht markieren. Müssen wir hingegen diesen Eintrag zuvor noch finden, so gilt $d(n) = \log n$. Wir gehen im Folgenden davon aus, dass $d(n)$ eine in n wachsende Funktion ist.

Theorem 12 *Existiert eine semidynamische Deletions-Only-Datenstruktur für eine zerlegbar Anfrage Q mit der Anfragezeit $q(n)$, dem Speicherplatzbedarf $s(n)$, der Konstruktionszeit $c(n)$ und dem Zeitbedarf $d(n)$ zum Löschen eines Eintrags, wobei $q(n)$, $s(n)/n$ und $c(n)/n$ monoton wachsende Funktionen sind. Dann existiert eine dynamischen Datenstruktur für Q mit der Anfragezeit $q'(n) \in O(q(8n) \cdot \log n)$, dem Speicherplatzbedarf $s'(n) \in O(s(8n))$, einem amortisiertem Zeitbedarf für eine* insert-*Operation von $t'_{\text{amort}}(n) \in O(\frac{c(n)\cdot \log n}{n})$ und einem amortisiertem Zeitbedarf für eine* delete-*Operation von $d'_{\text{amort}}(n) \in O(\frac{c(n)}{n} + d(n))$.*

Die Strategie, um eine derartige Datenstruktur zu realisieren, basiert auf einer Auflockerung der Größe der einzelnen Teilmengen S_i. Bei der Implementation einer semidynamischen Datenstruktur mit Insertion-Only wurden diese Mengen so gewählt, dass für alle i entweder die Teilmenge S_i leer oder voll, d.h. $|S_i| = 2^i$, war. Im Folgenden wollen wir für S_i die Fälle $|S_i| = 0$ oder $2^i/8 < |S_i| \leq 2^i$ erlauben. Wir speichern S_i mit Hilfe einer statischen Datenstruktur für 2^i Elemente, wobei diese zumindest zu einem Achtel gefüllt ist. Zudem gehen wir davon aus, dass die Größe von S_i bekannt ist, und die Elemente aus S_i als eine Liste zur Verfügung stehen. Dieses wird uns bei einem Neuaufbau der Datenstruktur behilflich sein. Da $|S_i| > 2^{i-3}$ ist, sind alle Teilmengen S_i mit $i \geq 3 + \log_2 n$ leer. Die Realisierung der Datenstruktur erfolgt analog zu der Implementation einer semidynamischen Datenstruktur für Insertion-Only. Wir können daher nun den Zeitbedarf für eine *membership*-Anfrage und den benötigten Speicherbedarf analysieren.

Somit gibt es höchstens $3+\log_2 n$ verschiedene nichtleere Teilmengen. Um eine *membership*-Anfrage zu beantworten, müssen wir zumindest jede dieser Mengen betrachten. Um eine solche Anfrage für ein S_i zu beantworten, benötigen wir $q(2^i)$ Schritte. Da für alle i die Größe der Mengen S_i durch $8n$ beschränkt ist, gilt für die resultierende Anfragezeit:

$$q'(n) \;\in\; O(q(8n)\log n)\ .$$

Der Speicherbedarf ergibt sich für monoton wachsende Funktionen $s(n)/n$ wie folgt:

$$\begin{aligned} s'(n) &= \sum_{i=0}^{3+\log_2 n} s(2^i) \;=\; \sum_{i=0}^{3+\log_2 n} \frac{s(2^i)}{2^i}\cdot 2^i \\ &\le \sum_{i=0}^{3+\log_2 n} \frac{s(2^{3+\log_2 n})}{2^{3+\log_2 n}}\cdot 2^i \;=\; \frac{s(8n)}{8n}\sum_{i=0}^{3+\log_2 n} 2^i \\ &\in \frac{s(8n)}{8n}\cdot O(n) \;=\; O(s(8n))\ . \end{aligned}$$

Wir nennen eine nicht leere Menge $S_i \neq \emptyset$ **deletion-sicher**, wenn $|S_i| \ge 2^{i-2}$ und **sicher**, wenn $2^{i-2} \le |S_i| \le 2^{i-1}$ ist. Eine Menge S_i ist somit deletion-sicher, wenn wir die Hälfte der Elemente aus dieser Menge streichen können, ohne dass wir die Datenstruktur neu aufbauen müssen. Eine Menge ist sicher, wenn wir auch die Größe der Menge verdoppeln können, ohne dass ein Neuaufbau nötig wird. Um die Funktionen *insert* und *delete* zu implementieren, verfahren wir wie folgt:

- Einfügen eines Elements x:

  Um ein Element x in unsere Datenstruktur einzufügen, bestimmen wir zunächst das kleinste k, so dass
  $$1 \;+\; |S_0| \;+\; |S_1| \;+\; \cdots \;+\; |S_k| \;\le\; 2^k$$
  ist. Dieses kann in $O(\log n)$ Schritten erfolgen. Anschließend generieren wir eine neue Datenstruktur für die Menge
  $$S'_k \;:=\; \{x\} \cup S_0 \cup S_1 \cup \cdots \cup S_k$$
  und entfernen die Datenstrukturen für die Mengen S_0 bis S_k. Dieses kann in Zeit $c(2^k)$ erfolgen. Man beachte, dass aus der Wahl von k folgt, dass $|S'_k| \ge 1 + \sum_{i\le k} |S_i| > 2^{k-1}$ und S'_k daher deletion-sicher ist.

- Entfernen eines Elements $x \in S_i$:

  Gilt für x, dass nach dem Entfernen die Menge S_i die untere Größenschranke noch nicht unterschreitet, d.h. $|S_i| > 2^{i-3}+1$, so können wir x einfach aus S_i entfernen. Der Zeitbedarf liegt daher in $d(2^i)$. Ansonsten müssen wir nach dem Entfernen die Mengen neu balancieren. Für das Entfernen muss ein Zeitbedarf von $d(2^i)$ miteinbezogen werden.

  Die Menge S_{i-1} ist vor dieser Operation entweder leer, oder es gilt $2^{i-4} < |S_{i-1}| \le 2^{i-1}$. Wir müssen bei einem Rebalancieren die folgenden drei Fälle unterscheiden:

  1. S_{i-1} ist klein, d.h. $|S_{i-1}| \le 2^{i-3}$:
     Wir entfernen x aus S_i und bilden eine neue Datenstruktur für $S'_{i-1} = S_{i-1} \cup S_i \setminus \{x\}$. Der Aufwand zum Generieren der neuen Datenstruktur beträgt $c(2^{i-1})$. Da ferner
     $$2^{i-3} \;= |S_i \setminus \{x\}| \;\le\; |S'_{i-1}| \;=\; |S_{i-1}| + |S_i \setminus \{x\}| \;\le\; 2^{i-2}\ ,$$
     ist die Menge S'_{i-1} nach dieser Operation sicher.

2. S_{i-1} hat eine mittlere Größe, d.h. $2^{i-3} < |S_{i-1}| \leq 2^{i-2}$:
   Wir entfernen x aus S_i und bilden eine neue Datenstruktur für $S'_i = S_{i-1} \cup S_i \setminus \{x\}$. Der Aufwand zum Generieren der neuen Datenstruktur beträgt $c(2^i)$. Zudem gilt

$$2 \cdot 2^{i-3} \;=\; 2^{i-2} \;\leq\; |S'_i| \;=\; |S_{i-1}| + |S_i \setminus \{x\}| \;\leq\; 2^{i-2} + 2^{i-3} \;<\; 2^i \;.$$

   Daher ist die Menge S'_i nach dieser Operation sicher.
3. S_{i-1} ist groß, d.h. $2^{i-2} < |S_{i-1}| \leq 2^{i-1}$:
   Nachdem wir x aus S_i entfernt haben, tauschen wir die Elemente aus $S_i \setminus \{x\}$ mit denen aus S_{i-1} , d.h. $S'_{i-1} = S_i \setminus \{x\}$ und $S'_i = S_{i-1}$. Dieses kann in $c(2^{i-1}) + c(2^i) \in O(c(2^i))$ Schritten erfolgen. Aus der Größenbeschränkung der Mengen S_{i-1} und S_i folgt unmittelbar, dass sowohl S'_{i-1} als auch S'_i sicher sind.

Der Aufwand zum Löschen eines Elements aus S_i beträgt daher $O(d(2^i) + c(2^i))$.

Wir wollen nun die amortisierten Zeiten für das Einfügen und Entfernen eines Elements analysieren:

Amortisierte Zeit für das Entfernen

Die Kosten zum Entfernen eines Elements aus S_i betragen (ohne die Rebalancierungskosten) $d(2^i) \leq d(n)$. Da nach dem Rebalancieren die Mengen S_i und S_{i-1} sicher oder leer sind, wird die Menge S_i höchstens alle 2^{i-3} Schritte erneut durch das Entfernen eines Elements aus dieser Menge rebalanciert. Wir können die Kosten zum Rebalancieren auf 2^{i-3} Schritte aufteilen, daher benötigen wir für diese Operation amortisiert $\frac{c(2^{i-1})+c(2^i)}{2^{i-3}} \in O(\frac{c(2^i)}{2^i})$ Schritte. Aufgrund der Monotonie ergibt sich eine amortisierte Laufzeit von $O(\frac{c(n)}{n})$. Es gilt:

$$d'_{\text{amort}}(n) \;\in\; O(c(n) + \frac{c(n)}{n}) \;.$$

Man beachte, dass S_i auch dann deletion-sicher ist, wenn es nach einer Löschoperation leer ist, und aufgrund einer Löschoperation aus S_{i+1} oder durch Hinzufügen von Elementen wieder neu aufgebaut wird.

Amortisierte Zeit für das Einfügen

Sei t ein Zeitpunkt, an dem wir die Menge S_k aufgrund einer Einfügeoperation neu aufbauen müssen, und $t' > t$ der nächste Schritt nach t ist, wo eine Einfügeoperation den Neuaufbau einer Menge S_ℓ mit $\ell \geq k$ zur Folge hat. Wir wollen nun die benötigte Anzahl an Operationen zwischen diesen beiden Schritten untersuchen. Man beachte, dass unmittelbar nach Schritt t gilt:

$$S_0 \;=\; S_1 \;=\; \cdots \;=\; S_{k-1} \;=\; \emptyset \;.$$

Lemma 3 $t' - t \geq 2^{k-2}$.

Beweis: Sei $t' - t < 2^{k-2}$. Dann kann ein erneuter Neuaufbau einer Menge S_ℓ mit $\ell \geq k$ nicht durch alleiniges Einfügen von Elementen erfolgen, da diese $2^{k-2} - 1$ Elemente in den Mengen S_ℓ mit $\ell < k-2$ verwaltet werden können. Einer Menge S_ℓ mit $\ell \geq k$ wird durch eine Einfügeoperation jedoch nur dann neu aufgebaut, wenn zumindest 2^{k-1} Elemente in den Mengen $S_0, \ldots, S_{k-1}$ vorhanden sind.

Da S_k nach dem Neuaufbau in Schritt t deletion-sicher ist, und dieser Umstand auch nicht durch das Entfernen von Elementen aus den Mengen S_ℓ mit $\ell > k$ geändert werden kann, benötigen wir

zumindest 2^{k-3} Löschoperationen, bevor durch Löschen eines Elements wieder Elemente aus S_k in eine Menge S_ℓ mit $\ell < k-1$ gelangen. Nach diesem Löschen ist S_k jedoch wieder deletion-sicher, und wir benötigen weitere 2^{k-3} Löschoperationen, bevor ein Neuaufbau wieder stattfinden muss. Daher kann durch Entfernen von Elementen innerhalb der ersten $2^{i-2}-1$ Schritten nach dem Schritt t maximal einmal eine solche Operation erfolgen. Eine solche Rebalancierung führt dazu, dass bis zu 2^{k-2} Elemente in die Mengen $S_0, \ldots, S_{k-1}$ gelangen. Zum Zeitpunkt t' gilt:

$$1 \;+\; |S_0| \;+\; |S_1| \;+\; \cdots \;+\; |S_{k-1}| \;\;\leq\;\; 2^{k-2} + 2^{k-3} \;<\; 2^{k-1} \;.$$

Folglich führt eine Einfügeoperation nur zum Neuaufbau der Menge S_{k-1}. ∎

Aus diesem Lemma können wir schließen, dass die Menge S_k maximal $\frac{n}{2^{k-2}}$ mal neu aufgebaut wird. Da der Aufbau von S_k maximal $c(2^k)$ Schritte benötigt, ist die gesamte Laufzeit für die Menge aller Einfügeoperationen beschränkt durch:

$$\sum_{k=0}^{\log_2 n} c(2^k)\frac{n}{2^{k-2}} \;=\; n \sum_{k=0}^{\log_2 n} \frac{c(2^k)}{2^{k-2}} \;\leq\; 4n \sum_{k=0}^{\log_2 n} \frac{c(n)}{n} \;=\; \frac{4n \cdot c(n) \cdot \log_2 n}{n} \;\in\; O(c(n)\log_2 n)\;,$$

da $c(n)/n$ monoton wachsend ist. Für die amortisierte Zeit gilt daher:

$$t'_{\text{amort}}(n) \;\in\; O(\frac{c(n)\log_2 n}{n})\;.$$

Wenden wir nun die Technik der Dynamisierung an, wie wir sie bereits kennen gelernt haben, so können wir zeigen:

Theorem 13 *Existiert eine semidynamische Deletions-Only Datenstruktur für eine zerlegbar Anfrage Q mit der Anfragezeit $q(n)$, dem Speicherplatzbedarf $s(n)$, der Konstruktionszeit $c(n)$ und dem Zeitbedarf $d(n)$ zum Löschen eines Eintrags, wobei $q(n)$, $s(n)/n$ und $c(n)/n$ monoton wachsende Funktionen sind und sei f eine monoton wachsende Funktionen mit $f(n) > 2$ für alle n. Dann existiert eine dynamische Datenstruktur für Q mit der Anfragezeit $q'(n) \in O(f(n) \cdot q(n))$, dem Speicherplatzbedarf $s'(n) \in O(s(n))$, einem worst-case Zeitbedarf für eine* delete-*Operation von $d'(n) \in O(\frac{c(n)}{n} + d(n))$ und einem Zeitbedarf für eine* insert-*Operation von*

$$t'(n) \in \begin{cases} O(\frac{c(n)\log n}{n\log(f(n)/\log n)}), & \text{wenn } f(n)/\log n \text{ monoton wachsend ist und} \\ O(\frac{c(n)f(n)n^{1/f(n)}}{n}), & \text{wenn } f(n)/\log n \text{ monoton fallend ist.} \end{cases}$$

5 Die Vorgänger-Datenstruktur

Eine Vorgänger-Datenstruktur verwaltet eine Menge $S \subseteq U$, so dass für jedes $x \in U$ die *predecessor*-Anfrage $pred(x, S) := \max\{y \in S | y < x\}$ effizient gefunden wird. Existiert kein Element y mit $y < x$ in der Menge S, so sei $pred(x, S) = \emptyset$.

Mit Hilfe einer sortierten Liste, in der wir einzelne Elemente sowie deren Nachbarn ansehen können, und der *predecessor*-Anfrage können wir die folgenden Aufgaben lösen:

1. Für ein gegebenes x finde den Wert $y \in S$, der den kleinsten Abstand zu x hat.
2. Für zwei gegebene Werte x, x' mit $x < x'$, bestimme die Werte $y \in S$ mit $x \leq y \leq x'$.

Eine statische Vorgänger-Datenstruktur können wir mit folgenden Hilfsmitteln implementieren:

1. Eine sortierte Liste in einem Array und mit Hilfe der binären Suche.

   In dieser Repräsentation benötigen wir zur Lösung einer *predecessor*-Anfrage $O(\log n)$ Schritte. Die Repräsentation kann auf Platz $O(n)$ realisiert werden, wobei $n = |S|$ ist.
2. Ein balancierter Baum.

   Diese Repräsentation ist wieder auf Platz $O(n)$ realisierbar und eine *predecessor*-Anfrage benötigt wieder $O(\log n)$ Zeit.
3. Eine Tabelle mit direktem Zugriff.

   Da diese Tabelle allen Elementen aus U einen Wert für die *predecessor*-Anfrage zuordnen muss, benötigt sie $O(|U|)$ Platz. Eine *predecessor*-Anfrage kann jedoch in konstanter Zeit beantwortet werden.

Um eine dynamische Vorgänger-Datenstruktur zu implementieren, können wir uns der AVL-, B- oder rot-schwarz Bäume bedienen. Jede dieser Implementationen benötigt $O(n)$ Platz und eine *predecessor*-Anfrage kann in $O(\log n)$ Zeit beantwortet werden.

5.1 Baumrepräsentation der Menge S

Wir wählen zunächst die folgende Repräsentation der Menge $S \subseteq U$: Seien $m = \lceil \log_2 |U| \rceil$ und $n = |S|$. U kann dann als eine Menge von Binärworten der Länge m aufgefasst werden, d.h. $U \subseteq \{0,1\}^m$. Sei T ein Baum der Tiefe m, wobei jeder Knoten v maximal zwei Söhne hat: einen linken Sohn $v0$ und einen rechten Sohn $v1$. Geben wir nun der Wurzel den Namen λ, wobei λ den leeren String darstellt. Dann beschreiben die Namen der Knoten im Baum auch die Pfade von der Wurzel des Baums zu den jeweiligen Knoten. Passen die Elemente aus U jeweils in ein Wort, so benötigt ein Baum T_S zur Repräsentation von S $O(nm)$ Speicherzellen. Die Blätter des Baums stellen also die Elemente aus S dar. Neben dem Baum gehen wir davon aus, dass die Blätter auch in einer sortierten Zeigerkette angeordnet sind.

Für ein gegebenes x sei y der längste Präfix von x, so dass y ein Knoten von T_S ist. Man beachte, dass y auch einen Pfad von der Wurzel bis zum Knoten y beschreibt, und dass wir y in der Zeit $O(|y|)$ finden können. Ist $y = x$, so ist $x \in S$, und wir können mit Hilfe der Zeigerkette direkt $pred(x, S)$ bestimmen. Ist $y \neq x$, so finden wir den Knoten yz nicht in T_S, wobei $z = x[|y| + 1]$ das $(|y| + 1)$-ste Bit in dem Binärwort x ist. Um das Vorgängerproblem für diesen Fall zu lösen, verfahren wir wie folgt:

- Ist $y1$ ein Präfix von x, so folgen wir beginnend bei y den am weitesten rechts liegenden Pfad zu einem Blatt – wir wählen für jeden Knoten startend bei y immer den rechten Sohn, wenn dieses möglich ist. Existiert dieser nicht, so wählen wir den linken Sohn. Das Blatt in dem dieser Pfad endet, ist das Ergebnis von $pred(x, S)$. Dieses ist in Abbildung 20 illustriert.
- Ist $y0$ ein Präfix von x, so folgen wir beginnend bei y den am weitesten links liegenden Pfad zu einem Blatt – wir wählen für jeden Knoten startend bei y immer den linken Sohn, wenn dieses möglich ist. Existiert dieser nicht, so wählen wir den rechten Sohn. Das Ergebnis von $pred(x, S)$ ist der Vorgänger des Blatts, in dem dieser Pfad endet.

Dieser Algorithmus benötigt $O(m)$ Schritte.

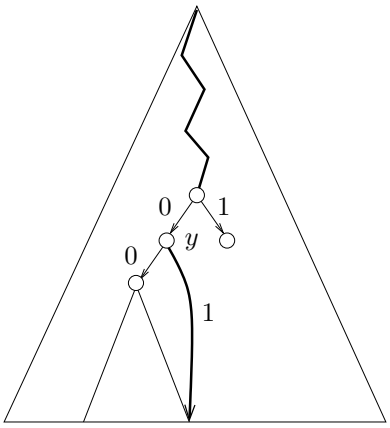


Abbildung 20: Suchweg für ein $x \notin S$ und $y1$ ist ein Präfix von x.

Um die Beantwortung einer Vorgängerfrage und die Suche nach einem maximalen Präfix von x, der auch der Präfix eines Elements in S ist, zu beschleunigen, fügen wir unserer Datenstruktur, die wir oben eingeführt haben, noch ein Wörterbuch hinzu (siehe Abbildung 21). Sei

$$S' := \{y \mid |y| = \lceil m/2 \rceil \text{ und } y \text{ ist der Präfix eines Elements aus } S\},$$

x' der Päfix von x der Länge $\lceil m/2 \rceil$ und x'' der entsprechende Suffix, d.h. $x = x'x''$. Ferner definiere $S_{x'} := \{y''|x'y'' \in S\}$. Wir benutzen nun ein Wörterbuch für S', welches neben den Elementen $y' \in S'$, für jedes solches Element auch einen Zeiger auf das maximale $y = \max(y') \in S$, welches y' als Präfix hat. Eine Vorgängerfrage kann nun wie folgt rekursive beantwortet werden:

- wenn $x' \in S'$, dann ist $pred(x, S) = pred(x'', S_{x'})$ und
- wenn $x' \notin S'$, dann ist $pred(x, S) = \max(pred(x', S'))$.

Die Zeit zur Beantwortung einer Vorgängerfrage reduziert sich durch diese rekursive Struktur zu $O(q \log m)$, wobei q die Zeit ist, die eine *membership*-Anfrage an das Wörterbuch benötigt. Durch diese Struktur vergrößert sich jedoch die Zeit, die wir für das Einfügen eines neuen Elements in die

Datenstruktur benötigen, da ein solches Element im worst-case eine Veränderung jedes Wörterbuchs auf jedem Level des Baumes zur Folge hat. Die resultierende Einfügezeit ist $O(mt_{\text{insert}})$, wobei t_{insert}, die benötigte Einfügezeit in ein Wörterbuch ist.

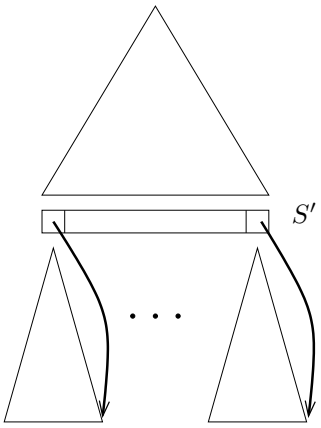


Abbildung 21: Kombination von Baum und Wörterbuch.

Wir wollen nun versuchen die Einfügezeit zu reduzieren. Hierfür wollen wir den Wert des maximalen Elements einer Menge S nicht mehr wie bisher in den Blättern des Baums abspeichern, sondern in einer separaten Datenstruktur, z.B. im Knoten selbst. Hierbei entstehende Konflikte können wir beispielsweise mit Hilfe eines Stacks umgehen. Das Vorgängerproblem können wir nun mit Hilfe der folgenden Prozedur lösen:

```
procedure pred(x, S)
1    if  S = ∅ then
2        return ∅
3    if  max_{y∈S} y < x then
4        return max_{y∈S} y
6    if  |S| = 1 then
7        return ∅
8    if  x' ∈ S' then
9        return pred(x'', S_{x'})
10   if  x' ∉ S' then
11       return pred(x', S')
```

Da bei den rekursiven Aufrufen der Prozedur pred die Länge der Zeichenketten für x und daher auch der Elemente in S verändert wird, entspricht der resultierende Wert von pred (x, S) nicht immer dem Wert des Vorgängers von x in S. Bei einer entsprechend gewählten Datenstruktur, können wir aber den jeweiligen Wert des Vorgängers von x aus S an der Stelle im Baum speichern, wo wir in Zeile 3 fündig werden. An Stelle von $\max_{y\in S} y$ in Zeile 4 müssen wir dann nur diesen Wert zurück geben. Wir bezeichnen diesen im Folgenden mit *value* $(\max_{y\in S} y)$. Um ein Element in unsere Datenstruktur einzufügen, können wir die folgende Prozedur benutzen:

```
procedure insert-pred(x, v, S)
1   if S = ∅ then
2       max_{y∈S} y := x und value(max_{y∈S} y) := v
3   else
4       if x > max_{y∈S} y then
5           h_x := max_{y∈S} y und h_v := value(max_{y∈S} y)
6           max_{y∈S} y := x und value(max_{y∈S} y) := v
7           x = h_x und v = h_v
8       sei x = x'x'' mit |x'| = ⌈|x|/2⌉
9       if x' ∉ S' then
10          füge x' in S' ein
11          max_{y∈S_{x'}} y := x'' und value(max_{y∈S_{x'}} y) := v
12      else insert-pred(x'', v, S_{x'})
```

Bis auf den rekursiven Aufruf in Zeile 12, die *membership*-Anfrage in Zeile 9 und das Einfügen eines Elements in ein Wörterbuch in Zeile 10 benötigt dieser Algorithmus konstante Zeit. Vielmehr ist sogar zu erkennen, dass wir bei der Ausführung der Rekursion maximal ein Element in ein Wörterbuch eintragen, Zeile 10 wird maximal einmal ausgeführt. Die Laufzeit ergibt sich somit wie folgt:

$$T(m) \;:=\; t_{\text{insert}} + T'(m) \quad \text{mit} \quad T'(m) \;:=\; O(t_{\text{mem}}) + T'(m/2) \;\in\; O(t_{\text{mem}} \log m) \;,$$

wobei t_{mem} die Zeit zur Beantwortung einer *membership*-Anfrage angibt. Um die genaue Zeit bestimmen zu können, müssen wir die Datenstruktur des Wörterbuchs genauer analysieren. Wir wollen hier zwei Möglichkeiten für eine solche Implementation ansprechen:

- Tabellen mit direktem Zugriff (siehe [Boas77] und [BoKZ77]): Die resultierende Datenstruktur wird Van-Emde-Boas-Baum genannt.

  Alle benötigten Operationen auf diese Art von Tabellen benötigen konstante Zeit. Der Platzbedarf ist jedoch $S(m) = O(2^{m/2}) + (2^{m/2}+1)S(m/2) \in O(2^m)$.

- Perfektes Hashing auf linearem Platz (siehe [Will83]): Die resultierende Datenstruktur wird X-Fast-Baum genannt.

  Jedes Element kommt maximal in einer Hashtabelle auf jedem Level des Baumes vor, daher ist der Platz beschränkt durch $O(nm)$. Zudem sind die amortisierten Kosten für das Einfügen und Löschen konstant und somit in $O(\log m)$ für die ganze Datenstruktur.

Beide Datenstrukturen erlauben es die Vorgängerfrage in $O(\log m)$ Zeit zu beantworten.

Um den Platzbedarf zu reduzieren, wollen wir eine zweistufige Datenstruktur betrachten. Hierfür teilen wir die Liste der sortierten Elemente aus S in Intervalle der Länge m ein. Sei $x_1, \ldots, x_k$ die Liste der jeweils ersten Elemente der Intervalle. Diese Liste verwalten wir mit Hilfe eines X-Fast-Baums. Der benötigte Platz hierfür ist $O(n)$. Für eine dynamische Datenstruktur erlauben wir Intervalle mit mindestens m und höchstens $2m$ Elementen. Wenn das Einfügen eines Elements zu einer Überschreitung der maximalen Intervallgröße führt, teilen wir diese Intervalle in zwei neue Intervalle und führen eine entsprechende Modifikation des X-Fast-Baums durch. Die Intervalle verwalten wir mit Hilfe von balancierten binären Suchbäumen, wie beispielsweise B- oder AVL-Bäumen. Jeder dieser Suchbäume kann auf einem Platz von $O(m)$ gespeichert werden. Der resultierende Platz liegt somit bei $O(n)$.

Die Beantwortung einer Vorgängerfrage kann in Zeit $O(\log m)$ erfolgen, indem wir zunächst die Vorgängerfrage mit Hilfe des X-Fast-Baums lösen und mit Hilfe dieses Ergebnisses noch die Vorgängerfrage in dem entsprechenden Intervall lösen. Der Zeitbedarf für das Einfügen eines Elements in die Datenstruktur hängt weitgehend von der Repräsentation der Intervalle ab.

5.2 Repräsentation der Menge S mit B-Bäumen

B-Bäume sind Suchbäume, die die folgenden Bedingungen erfüllen:

- alle Blätter haben die selbe Tiefe,
- jeder Knoten unterhalb der Wurzel speichert zwischen $B/2$ und B Schlüssel,
- die Wurzel speichert zwischen 1 und B Schlüssel und
- jeder Knoten, der k Schlüssel speichert und kein Blatt ist, hat $k+1$ Nachfolger.

Für einen Knoten v sei $x_1 < x_2 < \ldots < x_k$ die Menge der Schlüssel in v. Ist v kein Blatt, so separieren diese Schlüssel die Menge der Schlüssel, die in dem Teilbaum T_v mit der Wurzel v gespeichert sind wie folgt: alle Schlüssel kleiner x_1 befinden sich im ersten Teilbaum von T_v, alle Schlüssel zwischen x_1 und x_2 im zweiten Teilbaum, usw.

Eine Möglichkeit, um diese Schlüssel zu verwalten, ist eine Datenstruktur, welche die folgenden Operationen unterstützt: *rank*(y), *insert*(y), *delete*(y), *split* und *join*. Ein Ranking-Wörterbuch ist eine Datenstruktur, welche diese Operationen unterstützt. Hierbei sei *rank*$(y) := |\{x_i | x_i < y\}| + 1$. Suchen wir einen Wert $y \notin \{x_1, \ldots, x_k\}$, so gibt uns *rank*$(y)$ den Teilbaum, in dem wir die Suche fortsetzen müssen.

Durch die *insert*(y)-Operation kann es passieren, dass ein Knoten zu groß wird. In diesem Fall teilen wir diesen Knoten in zwei Teile mit Hilfe der *split*-Operation. Auf der anderen Seite können Knoten auch zu klein werden, wenn wir zu viele Schlüssel mit Hilfe der *delete*(y)-Operation löschen. Mit Hilfe der *join*-Operation können wir dann zwei Knoten vereinigen.

Lemma 4 *Gegeben sei ein Ranking-Wörterbuch, auf welchem die oben angegebenen Operationen in der Zeit $O(T)$ ausgeführt werden können. Dann gibt es eine dynamische Datenstruktur, in welcher Einfüge- und Löschoperationen sowie Vorgängerfragen in Zeit $O(\frac{T \log n}{\log B})$ bearbeitet werden können.*

Beweis: Wir implementieren jeden internen Knoten v mit Hilfe eines Ranking-Wörterbuchs R_v und eines Arrays P_v. Mit Hilfe von R_v verwalten wir die in v gespeicherten Schlüssel. P_v enthält Zeiger auf die Nachfolgeknoten von v.

Da jeder Knoten in einem B-Baum zwischen $B/2$ und B Schlüssel speichert, ist die Tiefe eines Baums, der n Schlüssel speichert, in $\Theta(\log_B n)$. Da jede Operation auf einem Ranking-Wörterbuch in $O(T)$ Schritten ausgeführt werden kann, können wir beginnend bei der Wurzel einen Schlüssel in $O(T \log_B n)$ Schritten finden. ∎

Wir wollen nun zunächst Ranking-Wörterbücher über einem kleinen Universum betrachten.

Lemma 5 *Ist die Wortlänge eines Computers zumindest $B(m+2)$ mit $m = \lceil \log |U| \rceil$, dann kann ein Ranking-Wörterbuch über dem Universum U, welches Platz für bis zu B Elemente bietet, so implementiert werden, dass das ganze Wörterbuch in ein Speicherwort passt und jede Operation in $O(1)$ Schritten ausgeführt werden kann.*

Beweis: Sei $S = \{x_1, \ldots, x_k\}$ mit $k \leq B$ und $x_i < x_{i+1}$ für alle $i < k$. Beachte, dass jedes Element aus U mit Hilfe eines Binärwortes der Länge m dargestellt werden kann. Sei nun $X \in \{0,1\}^{B(m+2)}$ ein Binärwort der Folgenden Gestalt:

$$X \;:=\; 00x_1 00x_2 00 \ldots 00x_k (001^m)^{B-k} \;.$$

Die $B-k$ Kopien der Zeichenketten 1^m dienen als Platzhalter, um der Zeichenkette X eine feste Länge zu geben.

Gegeben sei die Operation *rank*(y), so können wir die verbleibenden Operationen wie *insert*(y), *delete*(y), *split* und *join* mit Hilfe von Shiftoperationen in konstanter Zeit ausführen.

Wir wollen nun zeigen, wie das Ergebnis der Operation *rank*(y) für ein $y \in U$ in konstanter Zeit bestimmt werden kann. Zunächst berechnen wir

$$Z \;:=\; (00y)^B \;=\; \mathit{MULT}(y,\; (0^{m+1}1)^B)$$

und mit Hilfe des bitweisen *OR*

$$Z \;:=\; (01y)^B \;=\; \mathit{OR}((00y)^B,\; (01^{m+1})^B) \;.$$

Wir bestimmen nun

$$R \;:=\; \mathit{SUB}(Z', X) \;.$$

Die vor jedem y stehenden 01 in Z' bleiben hierbei für alle $x_i \leq y$ erhalten. Die verbleibenden Bitpaare werden zu 00:

$$R \;=\; (01\ \{0,1\}^m)^{rank(y)} (00\ \{0,1\}^m)^{B-rank(y)}$$

und mit Hilfe des bitweisen *AND*

$$R' \;:=\; (010^m)^{rank(y)} (0^{m+2})^{B-rank(y)} \;=\; \mathit{AND}(R,\; (110^m)^B) \;.$$

Der Wert von *rank*(y) entspricht nun der Anzahl der 1er in R'. Um diese zu bestimmen, multiplizieren wir R' mit $(0^{m+1}1)^B$. Den Wert von *rank*(y) finden wir dann in dem Bitintervall der Länge $m+2$ startend bei dem $(m+2)B-1$ niederwertigsten Bit des Ergebnisses. Ist die Zeichenkette von $\mathit{MULT}(R', (0^{m+1}1)^B)$ zu lang, um diese Bits anzugreifen, so teilen wir zunächst R' in zwei Teile, multiplizieren diese getrennt und addieren die beiden Teilergebnisse. ∎

Aus diesen beiden Lemmata können wir schließen:

Theorem 14 *Eine dynamische-Vorgänger-Datenstruktur über einem Universum der Größe $2^{O(w/B)}$ für Mengen der Größe n kann mit Hilfe von B-Bäumen und einer Wortlänge w so implementiert werden, dass Einfüge- und Löschoperationen sowie Vorgängeranfragen in der Zeit $O(\frac{\log n}{\log B})$ bearbeitet werden können. Die Darstellung dieser Datenstruktur benötigt $O(n)$ Wörter.*

Beweis: Um eine Menge von n Elementen in einem B-Baum zu speichern, benötigen wir einen B-Baum mit $\leq n$ Knoten, daher genügt es, Zeiger der Länge $\lceil \log n \rceil \in O(w/B)$ zur Darstellung der Kanten zu benutzen. Da jeder Knoten maximal $B+1$ Nachfolger hat, können wir jeden Knoten in $O(1)$ Wörtern speichern.

In Lemma 5 haben wir ferner gesehen, dass zur Implementation eines Ranking-Wörterbuches ein Wort genügt und dass jede nötige Operation in $O(1)$ Schritten ausgeführt werden kann. Die Zeitschranke von $O(\frac{\log n}{\log B})$ folgt unmittelbar aus Lemma 4, wobei wir $T \in O(1)$ setzen können. ∎

Literatur

[Boas77] Van Emde Boas, *Preserving Order in a Forest in Less than Logarithmic Time and Linear Space*, Information Processing Letters, vol. 6, 1977, S. 80-82.

[BoKZ77] Van Emde Boas, Kaas, Zijlstra, *Design and Implementation of an Efficient Priority Queue*, Mathematical System Theory, vol. 10, 1977, S. 99-127.

[CaWe79] Carter und Wegman, *Universal Classes of Hash Functions*, JCSS, vol. 18, 1979, S. 143-154.

[CoLR90] Cormen, Leiserson und Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.

[DHKP97] Dietzfelbinger, Hagerup, Katajainen und Penttonen, *A Reliable Randomized Algorithm for the Closest-Pair Problem*, Journal of Algorithms, vol. 25, no. 1, 1997, S. 19-51.

[FiMi95] F. Fich, P. B. Miltersen, *Tables Should be Sorted (on Random Access Machines)*, Proc. 4th Workshop on Algorithms and Data Structures, WADS'95, 1995, S. 163-174.

[FrKS84] Fredman, Komlós und Szemerédi, *Storing a Sparse Table with $O(1)$ Worst Case Access Time*, JACM, vol. 31, 1984, S. 538-544.

[GrRS90] R. Graham, J. Rothschild und B. Spencer, *Ramsey Theorie*, John Wiley & Sons, 1990.

[DKMM94] Dietzfelbinger, Karlin, Mehlhorn, Meyer auf der Heide, Rohnert, und Tarjan, *Dynamic Perfect Hashing: Upper and Lower Bounds*, SICOMP, vol. 23, 1994, S. 738-761.

[Pagh00] Pagh, *Faster Deterministic Dictionaries*, Proc. 11th ACM-SIAM Symposium on Discrete Algorithms, SODA'00, 2000, S. 487-493.

[Ram96a] R. Raman, *Priority queues: Small, Monotone, and Trans-Dichotomous*, Proc. 4th European Symposium on Algorithms, ESA'96, Springer-Verlag, 1996, S. 121-137.

[Ram96b] R. Raman, *Improved Data Structures for Predecessor Queries in Integer Sets*, TR 96-07, Kiung's College London, 1996, `ftp://ftp.dcs.kcl.ac.uk/pub/tech-reports`.

[Rose93] K. Rosen, *Elementary Number Theory and its Applications*, Addison-Wesley Publishing Company, 1993.

[TaYa79] Tarjan and Yao, *Storing a Sparse Table*, CACM, vol 22, no 11, 1979, S. 606-611.

[Will83] D. E. Willard, *Log-logarithmic Worst Case Range Queries are possible in Space $\Theta(n)$*, Information Processing Letters, vol. 17, 1983, S. 81-84.

[Yao81] A. Yao, *Should Tables be sorted?*, JACM, vol 28, no 3, 1981, S. 615-628.