3 Design und Implementierung

> *Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.*
>
> Frederick P. Brooks, *Jr., The Mythical Man Month*

Wie das Zitat von Brooks aus seinem bekannten Buch andeutet, ist der Entwurf der Datenstruktur die zentrale Entscheidung bei der Entwicklung eines Programms. Stehen die Datenstrukturen erst einmal fest, scheinen sich die Algorithmen fast von selbst zu ergeben, und ihre Programmierung ist vergleichsweise einfach.

Diese Sichtweise ist natürlich stark vereinfacht, aber deshalb nicht unrichtig. Im vorigen Kapitel haben wir die grundlegenden Datenstrukturen untersucht, die die Grundsteine für die meisten Programme darstellen. In diesem Kapitel werden wir diese Strukturen kombinieren, während wir uns mit dem Design und der Implementierung eines kleineren Programms befassen. Wir werden zeigen, wie sich das zu lösende Problem sich die Datenstrukturen auswirkt und wie sich der Code klar ergibt, wenn wir die Datenstrukturen festgelegt haben.

Ein Aspekt dieser Sichtweise ist, daß die Wahl der Programmiersprache relativ unwichtig in Hinblick auf das allgemeine Design wird. Wir werden das Programm zunächst abstrakt entwerfen und dann in C, Java, C++, Awk und Perl implementieren. Der Vergleich der Implementierungen demonstriert, wie Sprachen helfen können oder eher hinderlich sind, und zeigt auch Dinge, bei denen sie unwichtig sind. Ein Programmdesign kann natürlich durch eine Sprache beeinflußt sein, ist aber normalerweise nicht völlig von ihr abhängig.

Das Problem, das wir gewählt haben, ist ungewöhnlich, aber in seiner Grundform typisch für viele Programme: Einige Daten gehen rein, einige Daten kommen raus, und die Verarbeitung erfordert etwas Geschick.

Konkret wollen wir ein Programm entwickeln, das zufälligen englischen Text erzeugt, der sich gut lesen läßt. Wenn wir nur zufällig Buchstaben oder Wörter ausgeben, ist das Ergebnis Unfug. Zum Beispiel könnte ein Programm, das zufällig Buchstaben (und Leerzeichen zum Separieren einzelner Wörter) auswählt, folgenden Text produzieren:

```
xptmxgn xusaja afqnzgxl lhidlwcd rjdjuvpydrlwnjy
```

Das ist nicht sehr überzeugend. Wenn wir die Buchstaben nach der Häufigkeit ihres Vorkommens in englischen Texten gewichten, könnten wir

```
idtefoae tcs trder jcii ofdslnqetacp t ola
```

erhalten, was auch nicht viel besser ist. Auch Wörter, die zufällig aus einem Wörterbuch ausgesucht werden, ergeben nicht viel mehr Sinn:

```
polydactyl equatorial splashily jowl verandah circumscribe
```

Für bessere Ergebnisse benötigen wir ein statistisches Modell mit mehr Struktur, das z.B. die Häufigkeit des Vorkommens ganzer Wortgruppen beschreibt. Wo aber finden wir solche Statistiken?

Wir könnten uns einen langen englischen Text vornehmen und ihn genau studieren, es gibt jedoch einen leichteren und auch unterhaltsameren Ansatz. Die wichtigste Beobachtung ist, daß wir einen existierenden Text nutzen können, um ein statistisches Modell aus diesem zu konstruieren, das zu der Sprache paßt, *in der der Text geschrieben ist*, und mit dem wir dann zufälligen Text generieren können, der dem ursprünglichen Text ähnelt.

3.1 Der Markov-Chain-Algorithmus

Eine elegante Möglichkeit für ein solches Verarbeiten von Text ist eine Technik, die *Markov-Chain-Algorithmus* genannt wird. Wenn wir die Eingabe als Folge einander überlappender Wortgruppen auffassen, teilt der Algorithmus jede Wortgruppe in zwei Teile, ein aus mehreren Wörtern bestehendes *Präfix* und ein aus einem Wort bestehendes *Suffix*, das auf das Präfix folgt. Der Markov-Chain-Algorithmus gibt Wortgruppen aus, indem er zufällig das Suffix auswählt, das einem bestimmten Präfix folgt, und zwar entsprechend der statistischen Daten (in unserem Fall) des Originaltextes. Wortgruppen mit drei Wörtern funktionieren gut – ein aus zwei Wörtern bestehendes Präfix dient zur Wahl des Suffix-Wortes:

Setze w_1 und w_2 als die ersten beiden Wörter im Text.

Gib w_1 und w_2 aus.

Schleife:

> Wähle das Wort w_3 zufällig aus einem der Nachfolger
> vom Präfix $w_1\ w_2$ im Originaltext.

Gib w_3 aus.
Ersetze w_1 durch w_2 und w_2 durch w_3.
Wiederhole die Schleife.

Zur Illustration wollen wir annehmen, daß wir einen zufälligen Text, der auf den wenigen Sätzen des Anfangszitats dieses Kapitels basiert, mit Hilfe von Zwei-Wörter-Präfixen generieren wollen:
Show your flowcharts and conceal your tables and I will be
mystified. Show your tables and your flowcharts will be
obvious. (*Ende*)
Hier sehen Sie einige der Eingabepaare, die aus den ersten Wörtern und dem unmittelbar folgenden bestehen:

Eingabepräfix:	**Suffix-Wörter, die folgen:**
`Show your`	`flowcharts tables`
`your flowcharts`	`and will`
`flowcharts and`	`conceal`
`flowcharts will`	`be`
`your tables`	`and and`
`will be`	`mystified. Obvious.`
`be mystified.`	`Show`
`be obvious.`	`(`*Ende*`)`

Bei einem Vorgehen nach dem Markov-Chain-Algorithmus würde zuerst »`Show your`« ausgegeben und anschließend zufällig »`flowcharts`« oder »`tables`« gewählt werden. Wird das erstere ausgewählt, wird das »`your flowcharts`« zum aktuellen Präfix, und das nächste Wort ist entweder »`and`« oder »`will`«. Wird jedoch »`tables`« ausgewählt, ist das nächste Wort auf alle Fälle »`and`«. Dies wird fortgesetzt, bis genügend Text ausgegeben wurde oder die (*Ende*)-Markierung als Suffix ausgewählt wird.

Unser Programm wird einen englischen Text einlesen und den Markov-Chain-Algorithmus benutzen, um einen neuen Text zu generieren, der auf den Häufigkeiten basiert, mit denen Wortgruppen fester Länge auftreten. Die Anzahl der Wörter im Präfix, die in unserem Beispiel zwei war, ist ein Parameter. Wird ein kürzeres Präfix genommen, entsteht nicht so gut zusammenhängende Prosa, während längere Präfixe dazu führen können, daß der Eingabetext wortwörtlich reproduziert wird. Für einen englischen Text ist die Wahl eines Zwei-Wort-Präfixes, um ein drittes Wort zu wählen, ein guter Kompromiß; hierbei bleibt der Stil des Eingabetextes erhalten, während dennoch eine eigene Kreation entsteht.

Was ist aber ein Wort? Die offensichtliche Antwort lautet: eine Folge alphanumerischer Zeichen. Es ist jedoch wünschenswert die an Wörter gebundenen Interpunktionen beizubehalten, so daß »`words`« und »`words.`« verschieden voneinander sind. Hierdurch wird die Qualität des erzeugten Ausgabetextes erhöht, da die Interpunktionen, und damit (indirekt) die Grammatik, die Wahl des nächsten Wortes mit beeinflussen, obwohl dadurch auch einzelne Anführungszeichen oder Klammern entstehen können.

Wir werden deshalb ein Wort als all das definieren, was zwischen zwei Whitespaces[1] zu finden ist – eine Entscheidung, die keine Einschränkungen bezüglich der Eingabesprache macht und Interpunktionszeichen an ihren Wörtern läßt. Da die meisten Programmiersprachen Mittel haben, um Text in durch Whitespaces separierte Wörter zu zerlegen, läßt sich dies auch einfach implementieren.

Wegen der verwendeten Methode müssen alle Wörter und auch alle Wortgruppen, die aus zwei oder auch drei Wörtern bestehen, in der Eingabe vorgekommen sein, es sollte jedoch viele Wortgruppen aus vier oder fünf Wörtern geben, die künstlich erzeugt worden sind. Hier sehen Sie einige Sätze, die von dem Programm, das wir in diesem Kapitel entwickeln, erzeugt wurden, wobei als Eingabetext das Kapitel VII des Buches *»The Sun Also Rises«* von Ernest Hemingway verwendet wurde:

> As I started up the undershirt onto his chest black, and big stomach muscles bulging under the light. "You see them?" Below the line where his ribs stopped were two raised white welts. "See on the forehead." "Oh, Brett, I love you." "Let's not talk. Talking's all bilge. I'm going away tomorrow." "Tomorrow?" "Yes. Didn't I say so? I am." "Let's have a drink, then."

Wir waren freudig überrascht, daß die Interpunktion stimmte; denn das muß nicht so sein.

3.2 Datenstruktur-Alternativen

Wie viele Eingabedaten wollen wir verarbeiten können? Wie schnell muß das Programm laufen? Es scheint angemessen, von unserem Programm zu verlangen, daß es ein ganzes Buch einlesen kann. Das heißt, wir sollten auf Eingabegrößen von n = 100.000 Wörtern oder mehr vorbereitet sein. Die Ausgabe wird einige hundert oder vielleicht ein paar tausend Wörter lang sein, und das Programm sollte nur wenige Sekunden laufen und nicht Minuten. Mit über 100.000 Wörtern im Eingabetext ist n ziemlich groß, so daß der gewählte Algorithmus kein sehr einfacher sein kann, wenn wir ein schnell laufendes Programm haben wollen.

Der Markov-Chain-Algorithmus muß die gesamten Eingabedaten sehen, bevor er anfangen kann, Ausgabedaten zu generieren, so daß die gesamte Eingabe in irgendeiner Form gespeichert werden muß. Eine Möglichkeit wäre, den ganzen Eingabetext zu lesen und in einem einzigen langen String abzuspeichern. Es ist jedoch offensichtlich, daß wir die Eingabe lieber in Wörter zerlegen möchten. Wenn wir den Eingabetext als ein Array von Zeigern auf Wörter speichern, ist die Ausgabe einfach: Um ein Wort zu produzieren, durchsuche den Eingabetext, um zu sehen, welche Suffix-Wörter auf die aktuelle Präfix-Wortgruppe folgen können, und wähle eins zufällig aus. Dies würde

1. Leerzeichen, Tabulatoren und Zeilenumbrüche (A. d. Ü.)

jedoch bedeuten, daß wir alle 100.000 Eingabewörter für jedes Wort, das wir generieren, erneut durchsuchen müssen; tausend Ausgabewörter würden einhundert Millionen Stringvergleiche bedeuten, was wohl nicht besonders schnell sein wird.

Ein andere Möglichkeit ist, alle unterschiedlichen Eingabewörter nur einmal zu speichern, zusammen mit einer Liste, wo sie überall im Eingabetext vorkommen, so daß wir die aufeinanderfolgenden Wörter schneller lokalisieren können. Wir könnten eine Hashtabelle wie die aus Kapitel 2 benutzen, wobei diese Version nicht direkt den Anforderungen des Markov-Chain-Algorithmus entspricht, für den wir sehr schnell alle Suffix-Wörter für eine gegebene Präfix-Wortgruppe finden müssen.

Wir benötigen eine Datenstruktur, die ein Präfix und die dazugehörenden Suffixe besser repräsentiert. Das Programm wird in zwei Phasen ablaufen – in einer Eingabephase, in der die Datenstruktur für die Wortgruppen aufgebaut wird, und einer Ausgabephase, in der diese Datenstruktur benutzt wird, um den zufälligen Ausgabetext zu erzeugen. In beiden Phasen müssen wir (schnell) ein bestimmtes Präfix finden: in der Eingabephase, um die möglichen Suffixe zu aktualiseren, und in der Ausgabephase, um ein Wort zufällig aus allen möglichen auszuwählen. Hierzu empfiehlt sich eine Hashtabelle, deren Schlüssel die Präfixe sind und deren Werte eine Menge von Suffixen für das entsprechende Präfix sind.

Für die folgende Beschreibung wollen wir annehmen, daß ein Präfix aus zwei Wörtern besteht, so daß jedes Ausgabewort auf einem vorangehenden Wortpaar basiert. Die Anzahl der Wörter im Präfix ändert nicht das grundlegende Design, und das Programm sollte mit jeder beliebigen Präfixlänge umgehen können, aber durch die Wahl der Zahl zwei wird die Beschreibung einfacher. Das Präfix und die Menge aller zugehörigen Suffixe nennen wir einen *Zustand* (*state*) – dies entspricht der gebräuchlichen Terminologie für Markov-Algorithmen.

Wir müssen zu einem gegebenen Präfix alle Suffix-Wörter, die diesem Präfix folgen, so abspeichern, daß wir später darauf zugreifen können. Die Suffixe sind nicht sortiert und werden immer einzeln hinzugefügt. Wir wissen nicht, wie viele es sein werden, so daß wir eine Datenstruktur benötigen, die einfach und effizient wachsen kann, wie eine Liste oder ein dynamisches Array. Wenn wir den Ausgabetext generieren, müssen wir in der Lage sein, ein Suffix zufällig aus der Menge auszuwählen, die mit einem bestimmten Präfix assoziiert ist. Einträge werden nie gelöscht.

Was passiert, wenn eine Wortgruppe mehr als einmal vorkommt? Zum Beipiel »kommt zweimal vor« kommt zweimal vor, aber »kommt einmal vor« nur einmal. Dies könnte dadurch repräsentiert werden, daß »vor« zweimal in die Suffix-Liste für »kommt zweimal« eingefügt wird, oder aber es wird nur einmal eingefügt, und zwar mit einem zugehörigen Zähler, der den Wert 2 annimmt. Wir haben es mit und ohne Zähler versucht; ohne ist es leichter, da beim Hinzufügen eines Suffixes nicht überprüft werden muß, ob es schon einmal existiert, und Experimente haben gezeigt, daß die Laufzeitunterschiede unerheblich sind.

Zusammenfassend läßt sich sagen, daß jeder Zustand ein Präfix und eine Liste von Suffixen umfaßt. Diese Informationen werden in einer Hashtabelle abgelegt, mit dem Präfix als Schlüssel. Jedes Präfix ist eine Wortgruppe mit einer fest vorgegebenen Anzahl von Wörtern. Wenn ein Suffix mehr als einmal für ein bestimmtes Präfix vorkommt, wird jedes Vorkommen des Suffixes separat in die Liste der Suffixe aufgenommen.

Die nächste Entscheidung ist die, wie die einzelnen Wörter selbst repräsentiert werden. Die einfachste Möglichkeit wäre, sie als individuelle Strings zu speichern. Da der meiste Text jedoch viele Wörter enthält, die mehrmals vorkommen, würden wir wahrscheinlich Speicherplatz sparen, wenn wir eine zweite Hashtabelle für einzelne Wörter einführen würden. Dies würde auch das Hashen der Präfixe beschleunigen, da wir Zeiger anstelle von einzelnen Zeichen vergleichen könnten: Gleiche Wörter hätten die gleiche Adresse. Wir überlassen Ihnen diese Idee als Übung; im folgenden werden wir die Strings für die Wörter individuell speichern.

3.3 Erzeugen der Datenstruktur in C

Lassen Sie uns mit der C-Implementierung beginnen. Der erste Schritt ist die Definition einiger Konstanten.

```
enum {
    NPREF = 2,     /* Anzahl der Wörter im Präfix */
    NHASH = 4093,  /* Größe der Zustandshashtabelle */
    MAXGEN = 10000 /* maximale Anzahl generierter Wörter */
};
```

Die Deklaration definiert die Anzahl der Wörter für ein Präfix (`NPREF`), die Größe des Arrays für die Hashtabelle (`NHASH`) und eine Obergrenze für die Anzahl der generierten Wörter (`MAXGEN`). Wenn `NPREF` eine Kompilierungskonstante ist und keine Laufzeitvariable, wird die Speicherverwaltung einfacher. Die Arraygröße ist relativ groß gewählt worden, da wir erwarten, daß das Programm große Eingabedokumente zu verarbeiten hat, unter Umständen sogar ganze Bücher. Wir haben `NHASH = 4093` gewählt, so daß – wenn die Eingabe 10.000 verschiedene Präfixe (Wortpaare) hat – die durchschnittliche Listenlänge sehr kurz ist: zwei oder drei Präfixe. Je größer `NHASH` ist, desto kürzer wird die erwartete Länge für die Listen und desto schneller die Suche. Dieses Programm ist aber eigentlich nur ein Spielzeug, so daß die Performance nicht so kritisch ist. Machen wir aber das Array zu klein, wird das Programm die erwarteten Eingabemengen nicht in angemessener Zeit verarbeiten können. Wenn wir es andererseits zu groß machen, paßt es unter Umständen nicht in den Speicher.

Das Präfix kann als Array von Wörtern gespeichert werden. Die Elemente der Hashtabelle werden durch folgenden `State`-Datentyp repräsentiert, der eine `Suffix`-Liste mit dem Präfix assoziiert:

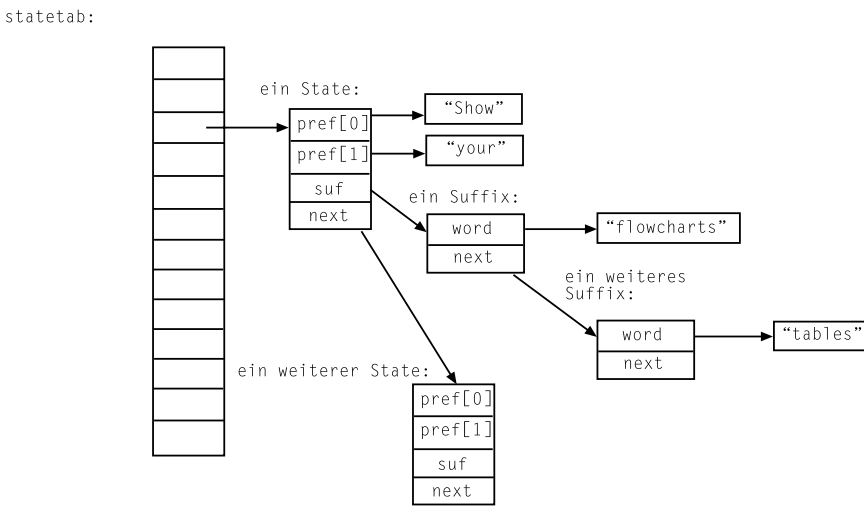
```
typedef struct State State;
typedef struct Suffix Suffix;

struct State { /* Präfix mit zugehöriger Suffix-Liste */
    char *pref[NPREF]; /* Präfix-Wörter */
    Suffix *suf;       /* Liste der Suffixe */
    State *next;       /* nächster Eintrag in der Hashtabelle */
};

struct Suffix { /* Liste der Suffixe */
    char *word;   /* Suffix */
    Suffix *next; /* nächster Eintrag in der Suffix-Liste */
};

State *statetab[NHASH]; /* Hashtabelle der Zustände */
```

Bildlich sieht die Datenstruktur so aus:



Wir benötigen eine Hashfunktion für Präfixe, die durch ein Array von Strings dargestellt sind. Es ist nicht weiter schwer, die in Kapitel 2 vorgestellte String-Hashfunktion so zu modifizieren, daß sie über alle Strings in dem Array läuft, womit letztendlich die Verkettung der Strings gehasht wird:

```
/* hash: Berechne den Hashwert eines Arrays von NPREF Strings */
unsigned int hash(char *s[NPREF])
{
    unsigned int h;
    unsigned char *p;

    int i;
```

```
    h = 0;
    for (i = 0; i < NPREF; i++)
        for (p = (unsigned char *) s[i]; *p != '\0'; p++)
            h = MULTIPLIER * h + *p;
    return h % NHASH;
}
```

Eine ähnliche Modifikation der `lookup`-Routine vervollständigt die Implementierung der Hashtabelle:

```
/* lookup: Suche nach prefix mit optionalem Hinzufügen */
/* Gibt einen Zeiger auf den gefundenen oder */
/* erzeugten Eintrag zurück, sonst NULL. */
/* Bei der Erzeugung wird nicht strdup aufgerufen, so daß */
/* Strings später nicht geändert werden dürfen. */
State* lookup(char *prefix[NPREF], int create)
{
    int i, h;
    State *sp;

    h = hash(prefix);
    for (sp = statetab[h]; sp != NULL; sp = sp->next) {
        for (i = 0; i < NPREF; i++)
            if (strcmp(prefix[i], sp->pref[i]) != 0)
                break;
        if (i == NPREF) /* gefunden */
            return sp;
    }
    if (create) {
        sp = (State *) emalloc(sizeof(State));
        for (i = 0; i < NPREF; i++)
            sp->pref[i] = prefix[i];
        sp->suf = NULL;
        sp->next = statetab[h];
        statetab[h] = sp;
    }
    return sp;
}
```

Beachten Sie, daß `lookup` keine Kopie des übergebenen Strings anlegt, wenn ein neuer `State` erzeugt wird; es werden nur die Zeiger in `sp->pref[]` gespeichert. Aufrufer von `lookup` müssen garantieren, daß die übergebenen Daten später nicht überschrieben werden. Wenn die Strings beispielsweise in einem Ein- und Ausgabepuffer liegen, muß vor einem Aufruf von `lookup` eine Kopie davon angelegt werden, da sonst im Anschluß eingelesene Eingabedaten die Daten, auf die in der Hashtabelle gezeigt wird, überschreiben könnten. Die Frage, wem eine Ressource gehört, die sich über eine Schnittstelle hinaus geteilt wird, taucht beim Programmieren sehr oft auf. Wir werden uns diesem Thema ausführlich im nächsten Kapitel widmen.

Als nächstes müssen wir die Hashtabelle beim Einlesen des Eingabetextes konstruieren:

```
/* build: Lies Eingabetext, und erzeuge die Präfix-Tabelle */
void build(char *prefix[NPREF], FILE *f)
{
    char buf[100], fmt[10];

    /* erzeuge einen Formatierungsstring, */
    /* nur "%s" könnte zu einem Überlauf führen */
    sprintf(fmt, "%%%ds", sizeof(buf)-1);
    while (fscanf(f, fmt, buf) != EOF)
        add(prefix, estrdup(buf));
}
```

Der merkwürdig anmutende Aufruf von `sprintf` vermeidet ein irritierendes Problem von `fscanf`, das sich ansonsten hervorragend für die Aufgabe eignet. Ein Aufruf von `fscanf` mit dem Formatierungsstring `"%s"` würde auch das nächste durch Whitespaces getrennte Wort aus der Eingabedatei in den Puffer `buf` lesen, aber es gibt keine Größenbeschränkung: Ein sehr langes Wort könnte zu einem Überlauf (*Overflow*) des Eingabepuffers führen und eine katastrophale Verwüstung anrichten. Wenn der Puffer 100 Bytes lang ist (was weit mehr ist, als wir für einen normalen Eingabetext jemals erwarten würden), können wir den Formatierungsstring `"%99s"` benutzen (der ein Byte Platz für die terminierende `'\0'` läßt), wodurch `fscanf` mitgeteilt wird, nach 99 Bytes mit dem Einlesen zu stoppen. Hierdurch wird ein zu langes Wort in Teilstücke zerlegt, was nicht sehr schön, aber sicher ist. Wir könnten auch folgendes deklarieren:

```
?  enum { BUFSIZE = 100 };
   char fmt[] = "%99s"; /* BUFSIZE-1 */
```

Dies würde jedoch zwei Konstanten für nur eine Entscheidung – die Größe des Puffers – bedeuten und führt dazu, daß die Beziehung der Konstanten bei Änderungen immer aufrecht erhalten werden muß. Dieses Problem kann aber ein für allemal gelöst werden, wenn der Formatierungsstring dynamisch durch einen Aufruf von `sprintf` erzeugt wird, weshalb wir diesen Weg gegangen sind.

Die zwei Argumente von `build` sind das Präfix-Array `prefix`, das die vorhergehenden `NPREF` Wörter im Eingabetext enthält, und ein `FILE`-Zeiger. `prefix` und eine Kopie des Eingabewortes werden an die Funktion `add` übergeben, die den neuen Eintrag in der Hashtabelle vornimmt und das Präfix aktualisiert:

```
/* add: Füge Wort zur Suffix-Liste hinzu, aktualisiere Präfix */
void add(char *prefix[NPREF], char *suffix)
{
    State *sp;

    sp = lookup(prefix, 1); /* Hinzufügen, wenn nicht vorhanden */
    addsuffix(sp, suffix);
```

```
    /* verschiebe die Wörter im Präfix */
    memmove(prefix, prefix+1, (NPREF-1)*sizeof(prefix[0]));
    prefix[NPREF-1] = suffix;
}
```

Der Aufruf von `memmove` ist ein Idiom für das Löschen aus einem Array. Er verschiebt die Elemente `1` bis `NPREF-1` in dem Array um eine Stelle an die Positionen `0` bis `NPREF-2`, wodurch das erste Wort des Präfixes gelöscht und am Ende des Arrays Platz für ein neues geschaffen wird.

Die `addsuffix`-Funktion fügt das neue Suffix hinzu:

```
/* addsuffix: Füge suffix zum Zustand *sp hinzu, */
/*            suffix darf sich später nicht verändern */
void addsuffix(State *sp, char *suffix)
{
    Suffix *suf;

    suf = (Suffix *) emalloc(sizeof(Suffix));
    suf->word = suffix;
    suf->next = sp->suf;
    sp->suf = suf;
}
```

Wir haben die Aktualisierung des Zustands in zwei Funktionen aufgespalten: `add` übernimmt die generelle Aufgabe, ein Suffix zu einem Präfix hinzuzufügen, während `addsuffix` die implementierungsspezifischen Schritte ausführt, um ein Wort zur Suffix-Liste hinzuzufügen. Die `add`-Routine wird von `build` benutzt, aber `addsuffix` wird nur intern von `add` benutzt; die Funktion ist ein Implementierungsdetail, das sich ändern könnte, und es scheint besser zu sein, es in eine separate Funktion auszulagern, auch wenn diese nur an einer Stelle aufgerufen wird.

3.4 Den Ausgabetext generieren

Wenn die Datenstruktur erzeugt worden ist, besteht der nächste Schritt darin, den Ausgabetext zu generieren. Die grundlegende Idee ist die gleiche wie zuvor: Das Präfix wird vorgegeben, und eines seiner Suffixe ist zufällig auszuwählen und auszugeben. Anschließend muß das Präfix noch aktualisiert werden. Dies ist die Hauptschleife des Algorithmus. Wir müssen aber noch herausfinden, wie wir anfangen und wieder aufhören. Das Anfangen ist leicht, wenn wir uns die Wörter des ersten Präfixes merken und mit diesen beginnen. Auch das Aufhören ist einfach. Wir brauchen ein Markierungswort, um den Algorithmus terminieren zu lassen. Nachdem der gesamte reguläre Eingabetext gelesen ist, können wir einen solchen Terminator hinzufügen – ein »Wort«, das garantiert nicht in irgendeinem Eingabetext vorkommt:

```
build(prefix, stdin);
add(prefix, NONWORD);
```

NONWORD sollte ein Wert sein, der niemals in einem regulären Eingabetext auftauchen kann. Da die Eingabeworte durch Whitespaces begrenzt werden, würde sich dafür ein »Wort« eignen, das aus einem Whitespace besteht, wie zum Beispiel ein Newline-Zeichen:

```
char NONWORD[] = "\n"; /* kann nicht als reguläres Wort vorkommen */
```

Es gibt ein weiteres Problem: Was passiert, wenn wir nicht genügend Eingaben haben, um anfangen zu können? Um diese Art von Problem zu lösen, gibt es zwei Ansätze: Entweder wird das Programm frühzeitig beendet, wenn nicht genügend Eingabedaten zur Verfügung stehen, oder es muß arrangiert werden, daß es immer genügend gibt, wodurch man sich nicht weiter um eine Überprüfung kümmern muß. Für unser Programm funktioniert der zweite Ansatz ganz gut.

Wir können das Erzeugen der Datenstruktur und das Generieren des Ausgabetextes mit einem künstlichen Präfix beginnen, das garantiert, daß wir immer genügend Eingabedaten für das Programm haben. Vor jeder Schleife wird das Präfix so initialisiert, daß es nur aus NONWORD-Wörtern besteht. Dies hat den hübschen Vorteil, daß das erste Wort des Eingabetextes das erste *Suffix* des künstlichen Präfixes wird, so daß die Schleife zum Generieren des Ausgabetextes nur Suffixe ausgeben muß.

In dem Fall, daß die Ausgabe zu lang wird, können wir den Algorithmus terminieren, nachdem eine gewisse Anzahl von Wörtern ausgegeben wurde oder wir auf ein NONWORD als Suffix stoßen, je nachdem, was zuerst eintrifft.

Das Hinzufügen einiger NONWORD-Einträge an das Ende der Daten vereinfacht die Hauptschleife des Programms wesentlich; dies ist ein Beispiel für die Technik des Hinzufügens von *Wächtermarken*, um Grenzen zu markieren.

Als Faustregel gilt, daß Sie versuchen sollten, Unregelmäßigkeiten, Ausnahmen und Sonderfälle in den Daten selbst zu behandeln. Code ist schwerer zu verstehen, weshalb der Programfluß so einfach und regulär wie möglich sein sollte.

Die generate-Funktion arbeitet nach dem Algorithmus, den wir zuvor aufgezeigt haben. Sie produziert ein Wort pro Ausgabezeile, was von einer Textverarbeitung zu längeren Zeilen zusammengefaßt werden kann; in Kapitel 9 stellen wir einen einfachen Formatierer mit dem Namen fmt für diese Aufgabe vor.

Durch die Verwendung von NONWORD-Strings zum Anfang und zum Schluß beginnt und stoppt generate ordnungsgemäß:

```
/* generate: Erzeuge Ausgabetext, ein Wort pro Zeile */
void generate(int nwords)
{
    State *sp;
    Suffix *suf;
    char *prefix[NPREF], *w;
    int i, nmatch;

    for (i = 0; i < NPREF; i++) /* Setze auf Start-Präfix zurück */
        prefix[i] = NONWORD;
```

```
    for (i = 0; i < nwords; i++) {
        sp = lookup(prefix, 0);
        nmatch = 0;
        for (suf = sp->suf; suf != NULL; suf = suf->next)
            if (rand() % ++nmatch == 0) /* Wahrscheinlichkeit */
                w = suf->word;          /* = 1/nmatch */
        if (strcmp(w, NONWORD) == 0)
            break;
        printf("%s\n", w);
        memmove(prefix, prefix+1, (NPREF-1)*sizeof(prefix[0]));
        prefix[NPREF-1] = w;
    }
}
```

Beachten Sie die Vorgehensweise, mit der wir ein Wort zufällig aus der Suffix-Liste auswählen, wenn wir nicht wissen, wie viele Einträge die Liste enthält. Die Variable `nmatch` zählt die Anzahl der Suffix-Wörter beim Durchgehen der Liste. Der Ausdruck

```
rand() % ++nmatch == 0
```

inkrementiert `nmatch` und ist mit der Wahrscheinlichkeit `1/nmatch` wahr. Somit wird das erste Wort mit einer Wahrscheinlichkeit von 1 ausgewählt, das zweite mit einer Wahrscheinlichkeit von 1/2, das dritte ersetzt das vorher gewählte Wort mit einer Wahrscheinlichkeit von 1/3 und so weiter. Zu jedem Zeitpunkt wurde jedes der bis dahin gesehenen k Wörter mit einer Wahrscheinlichkeit von $1/k$ ausgewählt.

Zu Anfang setzen wir `prefix` auf den Startwert, der garantiert in die Hashtabelle eingefügt wurde. Das erste Suffix, das wir dazu finden, ist das erste Wort des Eingabedokuments, da es das einzige ist, das dem einmaligen Startpräfix folgen kann. Anschließend werden zufällig Suffixe ausgewählt. Die Schleife ruft `lookup` auf, um den Hashtabelleneintrag für das aktuelle Präfix zu finden, wählt dann ein Suffix zufällig aus, gibt das gewählte Wort aus und aktualisiert schließlich das aktuelle Präfix.

Wenn das gewählte Suffix `NONWORD` ist, sind wir fertig, da wir den Zustand gewählt haben, der dem Ende des Eingabetextes entspricht. Wenn das gewählte Suffix nicht `NONWORD` ist, geben wir es aus, entfernen das erste Wort vom Präfix mit einem Aufruf von `memmove`, setzen das letzte Wort des Präfixes gleich dem gewählten Suffix und setzen schließlich die Schleife fort.

Nun können wir alle Funktionen in einer `main`-Routine vereinen, die von der Standardeingabe liest und maximal eine vorgegebene Anzahl von Wörtern generiert:

```
/* main: Generierung eines zufälligen Textes nach */
/*       dem Markov-Chain-Algorithmus */
int main(void)
{
    int i, nwords = MAXGEN;
    char *prefix[NPREF];        /* aktueller Eingabe-Präfix */
```

```
    for (i = 0; i < NPREF; i++) /* initialisiere Start-Präfix */
        prefix[i] = NONWORD;
    build(prefix, stdin);
    add(prefix, NONWORD);
    generate(nwords);
    return 0;
}
```

Hiermit ist die C-Implementierung vollständig. Wir werden am Ende des Kapitels auf einen Vergleich der Programme in den verschiedenen Programmiersprachen zurückkommen. Die großen Stärken von C sind, daß der Programmierer die gesamte Kontrolle über die Implementierung hat und Programme, die in C geschrieben sind, meistens auch sehr schnell sind. Dafür muß der C-Programmierer auch mehr Arbeit erledigen, wie die Allozierung und das Freigeben von Speicher, das Erzeugen von Hashtabellen und verketteten Listen und dergleichen. C ist ein rassiermesserscharfes Werkzeug, mit dem man elegante und effiziente Programme erstellen kann oder ein fürchterliches Durcheinander.

Übung 3-1. Der Algorithmus zum Auswählen eines zufälligen Eintrags aus einer Liste von unbekannter Länge ist von einem guten Zufallsgenerator abhängig. Entwerfen Sie Experimente, und führen Sie diese aus, um zu bestimmen, wie die vorgestellte Methode in der Praxis funktioniert.

Übung 3-2. Wenn jedes Eingabewort in einer zweiten Hashtabelle gespeichert wird, wird der Text nur einmal gespeichert, was Speicherplatz sparen würde. Messen Sie für einige Dokumente aus, wieviel. Diese Organisation der Daten würde es uns erlauben, Zeiger statt Strings in den Hashketten für Präfixe zu vergleichen, was schneller sein sollte. Implementieren Sie diese Variante, und messen Sie die Geschwindigkeitsänderung und den Speicherbedarf.

Übung 3-3. Entfernen Sie die Anweisungen, die die `NONWORD`-Wächtermarken am Anfang und am Ende der Eingabedaten plazieren, und modifizieren Sie `generate` so, daß es auch ohne diese Daten ordnungsgemäß anfängt und stoppt. Stellen Sie sicher, daß für Eingabetexte mit keinem, einem, zwei, drei und vier Wörtern korrekte Ausgaben produziert werden. Vergleichen Sie die Implementierung zu der Version mit den Wächtermarken.

3.5 Java

Unsere zweite Implementierung des Markov-Chain-Algorithmus erfolgt in Java. Objektorientierte Sprachen wie Java erlauben es, den Schnittstellen zwischen den Komponenten eines Programms besondere Aufmerksamkeit zu schenken, die dann als unabhängige Datenelemente gekapselt Objekte oder Klassen genannt werden und mit denen Funktionen assoziiert sind, die Methoden genannt werden.

Java hat eine mächtigere Bibliothek als C, die auch eine Anzahl *Container-Klassen* umfaßt, um existierende Objekte auf verschiedene Weisen zu gruppieren. Ein Beispiel ist die Klasse `Vector`, die ein dynamisch wachsendes Array zur Verfügung stellt und jeden `Object`-Typ abspeichern kann. Ein weiteres Beispiel ist die Klasse `Hashtable`, mit der man Werte eines Typs mit Hilfe von Objekten eines anderen Typs als Schlüssel speichern und wiederfinden kann.

In unserer Applikation ist ein `Vector` von `String` die natürliche Wahl, um die Präfixe und Suffixe zu speichern. Wir können `Hashtable` mit Präfix-Vektoren als Schlüssel und Suffix-Vektoren als Werte benutzen. Die Terminologie für eine solche Konstruktion ist eine *map* (Abbildung) von Präfixen auf Suffixe; in Java benötigen wir keinen expliziten `State`-Typ, da in der Hashtabelle die Präfixe implizit mit den Suffixen assoziiert werden. Das Design unterscheidet sich von der C-Version, in der wir eine extra `State`-Struktur hatten, die sowohl das Präfix als auch die Suffix-Liste enthielt, und wo wir das Präfix gehasht haben, um den kompletten Zustand wiederzufinden.

`Hashtable` stellt eine `put`-Funktion bereit, um ein Schlüssel/Wert-Paar zu speichern, und eine `get`-Methode, um den zu einem Schlüssel gehörenden Wert wiederzufinden:

```
Hashtable h = new Hashtable();
h.put(key, value);
Sometype v = (Sometype) h.get(key);
```

Unsere Implementierung besteht aus drei Klassen. Die erste Klasse, `Prefix`, hält die Wörter des Präfixes:

```
class Prefix {
    public Vector pref; // NPREF benachbarte Wörter der Eingabe
    ...
```

Die zweite Klasse, `Chain`, liest den Eingabetext, erstellt die Hashtabelle und generiert den Ausgabetext; hier sehen Sie die Variablen der Klasse:

```
class Chain {
    static final int NPREF = 2; // Größe des Präfix
    static final String NONWORD = "\n";
                    // "Wort", das nicht auftauchen kann
    Hashtable statetab = new Hashtable();
                    // Schlüssel = Prefix, Wert = Suffix-Vector
    Prefix prefix = new Prefix(NPREF, NONWORD);
                    // Start-Präfix
    Random rand = new Random();
    ...
```

Die dritte Klasse ist die öffentliche Schnittstelle; sie enthält `main` und instanziiert ein Objekt vom Typ `Chain`:

```
class Markov {
    static final int MAXGEN = 10000; // maximale Anzahl zu
                                     // erzeugender Wörter
```

```
    public static void main(String[] args) throws IOException
    {
        Chain chain = new Chain();
        int nwords = MAXGEN;

        chain.build(System.in);
        chain.generate(nwords);
    }
}
```

Wenn eine Instanz der Klasse `Chain` erzeugt wird, so wird dadurch wiederum eine Hashtabelle erzeugt und das Start-Präfix mit `NPREF NONWORD`-Wörtern initialisiert. Die `build`-Funktion benutzt die Bibliotheksklasse `StreamTokenizer`, um die Eingabedaten in Wörter zu zerlegen, die durch Whitespaces voneinander getrennt sind. Die drei Aufrufe vor der Schleife versetzen das `StremTokenizer`-Objekt in den gewünschten Zustand für unsere Definition für ein »Wort«.

```
// Chain build: Erzeuge Zustandstabelle aus Eingabestream
void build(InputStream in) throws IOException
{
    StreamTokenizer st = new StreamTokenizer(in);

    st.resetSyntax();           // entferne Default-Regeln
    st.wordChars(0, Character.MAX_VALUE); // alle Zeichen aktivieren
    st.whitespaceChars(0, ' '); // außer dem Leerzeichen
    while (st.nextToken() != st.TT_EOF)
        add(st.sval);
    add(NONWORD);
}
```

Die `add`-Funktion ermittelt den Suffix-Vektor für das aktuelle Präfix aus der Hashtabelle; gibt es noch keinen (der Vektor ist `null`), so erzeugt `add` einen neuen Vektor und eine neue `Prefix`-Instanz, die in der Hashtabelle abgelegt werden. In jedem Fall wird das neue Wort zum Suffix-Vektor hinzugefügt und das aktuelle Präfix aktualisiert, indem das erste Wort entfernt und das neue Wort am Ende hinzugefügt wird.

```
// Chain add: Füge word zum Suffix-Vektor hinzu, aktualisiere prefix
void add(String word)
{
    Vector suf = (Vector) statetab.get(prefix);

    if (suf == null) {
        suf = new Vector();
        statetab.put(new Prefix(prefix), suf);
    }
    suf.addElement(word);
    prefix.pref.removeElementAt(0);
    prefix.pref.addElement(word);
}
```

Beachten Sie, daß `add`, wenn `suf` gleich `null` ist, ein *neues* Präfix zur Hashtabelle hinzufügt statt `prefix` selbst. Dies ist deshalb erforderlich, da die `Hashtable`-Klasse Einträge nur als Referenz speichert. Würden wir keine Kopie machen, würden wir die Daten in der Tabelle überschreiben, wenn wir `prefix` aktualisieren. Das ist das gleiche Problem, das wir auch bei der C-Implementierung hatten.

Die `generate`-Funktion ähnelt der in C, ist aber etwas kompakter, da sie direkt über einen Index zufällig auf den Suffix-Vektor zugreifen kann und nicht erst eine Liste durchlaufen muß.

```
// Chain generate: Generiere Ausgabetext
void generate(int nwords)
{
    prefix = new Prefix(NPREF, NONWORD);
    for (int i = 0; i < nwords; i++) {
        Vector s = (Vector) statetab.get(prefix);
        int r = Math.abs(rand.nextInt()) % s.size();
        String suf = (String) s.elementAt(r);
        if (suf.equals(NONWORD))
            break;
        System.out.println(suf);
        prefix.pref.removeElementAt(0);
        prefix.pref.addElement(suf);
    }
}
```

Die beiden Konstruktoren von `Prefix` erzeugen neue Instanzen aus den übergebenen Daten. Der erste kopiert ein existierendes `Prefix`-Objekt, und der zweite erzeugt ein Präfix mit `n` Kopien eines Strings; dies benutzen wir, um `NPREF` Kopien von `NONWORD` für das Start-Präfix zu machen.

```
// Prefix constructor: Dupliziere existierendes Präfix
Prefix(Prefix p)
{
    pref = (Vector) p.pref.clone();
}

// Prefix constructor: n Kopien von str
Prefix(int n, String str)
{
    pref = new Vector();
    for (int i = 0; i < n; i++)
        pref.addElement(str);
}
```

`Prefix` verfügt auch über die beiden Methoden `hashCode` und `equals`, die implizit durch die Implementierung von `Hashtable` aufgerufen werden, um die Tabelle zu indizieren und zu durchsuchen. Nur die Notwendigkeit, eine Klasse mit diesen beiden Funktionen für die Benutzung von `Hashtable` zu haben, hat uns veranlaßt, aus `Prefix` ein eigenständige Klasse zu machen und nicht nur einen `Vector` zu benutzen, wie für die Suffixe.

Die `hashCode`-Methode erzeugt einen einzigen Hashwert, indem sie die Hashcodes für die einzelnen Elemente des Vektors kombiniert:

```
static final int MULTIPLIER = 31; // für hashCode()

// Prefix hashCode: Erzeuge Hashwert aus allen Präfix-Wörtern
public int hashCode()
{
    int h = 0;

    for (int i = 0; i < pref.size(); i++)
        h = MULTIPLIER * h + pref.elementAt(i).hashCode();
    return h;
}
```

`equals` führt einen elementeweisen Vergleich der Wörter in zwei Präfixen aus:

```
// Prefix equals: Vergleiche zwei Präfixe auf gleiche Wörter
public boolean equals(Object o)
{
    Prefix p = (Prefix) o;

    for (int i = 0; i < pref.size(); i++)
        if (!pref.elementAt(i).equals(p.pref.elementAt(i)))
            return false;
    return true;
}
```

Das Java-Programm ist wesentlich kleiner als das C-Programm und nimmt den Programmierer mehr Arbeit ab; wofür `Vector` und `Hashtable` die offensichtlichsten Beispiele sind. Im allgemeinen ist die Speicherverwaltung einfacher, da `Vector`-Objekte bei Bedarf wachsen und die Garbage-Collection sich um nicht mehr benötigten Speicher kümmert, auf den keine Referenz mehr zeigt, wodurch das Freigeben von Speicher nicht mehr nötig ist. Aber um die `Hashtable`-Klasse zu benutzen, müssen wir dennoch die Funktionen `hashCode` und `equals` schreiben; Java nimmt uns also nicht alle Arbeit ab.

Wenn wir die Art und Weise vergleichen, wie im C- und im Java-Programm die gleiche grundlegende Datenstruktur repräsentiert und auf ihr operiert wird, sehen wir, daß in der Java-Version eine bessere Trennung der Funktionalitäten vorliegt. Zum Beispiel wäre ein Wechsel von `Vector` auf ein normales Array einfach. In der C-Version weiß jeder, was alle anderen wie machen: Die Hashtabelle arbeitet mit Arrays, die an verschiedenen Plätzen gehalten werden, `lookup` kennt das Layout der `State`- und `Suffix`-Strukturen und überall ist die Größe des Präfix-Arrays bekannt.

```
% java Markov < jr_chemistry.txt | fmt
Wash the blackboard. Watch it dry. The water goes
into the air. When water goes into the air it
evaporates. Tie a damp cloth to one end of a solid or
```

```
liquid. Look around. What are the solid things?
Chemical changes take place when something burns. If
the burning material has liquids, they are stable and
the sponge rise. It looked like dough, but it is
burning. Break up the lump of sugar into small pieces
and put them together again in the bottom of a liquid.
```

Übung 3-4. Verändern Sie die Java-Version des Markov-Chain-Algorithmus so, daß ein Array anstelle eines `Vector`-Objekts für das Präfix in der `Prefix`-Klasse benutzt wird.

3.6 C++

Unsere dritte Implementierung ist in C++. Da C++ praktisch eine Obermenge von C ist, kann es so benutzt werden, als ob es sich um C mit einigen bequemeren Schreibweisen handeln würde. Unsere anfängliche C-Version von `markov` ist auch ein gültiges C++-Programm. Ein angemessenere Benutzung von C++ würde jedoch darin bestehen, Klassen für die Objekte im Programm zu definieren, und zwar mehr oder weniger so, wie wir es bei der Java-Version gemacht haben. Hierdurch könnten wir Implementierungsdetails verstecken. Wir haben uns dafür entschieden, sogar noch weiter zu gehen und die Standard Template Library (STL) zu benutzen, da die STL über eingebaute Mechanismen verfügt, die vieles von dem können, was wir brauchen. Der ISO-Standard für C++ beinhaltet die STL als Teil der Sprachspezifikation.

Die STL stellt Container wie Vektoren, Listen und Mengen zur Verfügung sowie eine Menge grundlegender Algorithmen für die Suche, das Sortieren, Einfügen und Löschen. Durch das Template-Sprachmittel von C++ arbeitet jeder Algorithmus auf einer Vielzahl von Containern, inklusive selbstdefinierter oder den normalen aus C bekannten Arrays. Container werden in C++ als Templates beschrieben, die für einen spezifischen Datentyp instanziiert werden. Beispielsweise ist `vector` ein Container, der für viele Typen benutzt werden kann, wie z.B. als `vector<int>` oder `vector<string>`. Alle `vector`-Operationen einschließlich der Standardalgorithmen zum Sortieren können für solche Datentypen benutzt werden.

Zusätzlich zum `vector`-Container, der der Java-Klasse `Vector` ähnelt, stellt die STL einen `deque`-Container zur Verfügung. Eine *Deque* (gesprochen »deck«) ist eine zweiendige *Queue*, die genau das kann, was wir für Präfixe benötigen: Sie hält `NPREF` Elemente und läßt uns das erste Element entfernen und ein neues am Ende hinzufügen, und zwar beides in $O(1)$-Zeit. Die STL-Deque ist mächtiger, als wir sie eigentlich brauchen, da sie das Entfernen und Hinzufügen an beiden Enden erlaubt, aber die Performance-Garantien legen ihre Verwendung nahe.

Die STL enthält auch einen expliziten `map`-Container, der auf balancierten Bäumen basiert, Schlüssel/Wert-Paare speichert und das Ermitteln des zu einem beliebigen Schlüssel gehörenden Wertes in einer Zeit von $O(\log n)$ bewerkstelligt. Die C++-Maps

sind vielleicht nicht ganz so effizient wie eine $O(1)$-Hashtabelle, aber es ist schön, daß kein zusätzlicher Code geschrieben werden muß, um sie zu nutzen. (Es existieren einige über den Standard hinausgehende C++-Bibliotheken, die einen `hash`- oder `hash_map`-Container bereitstellen, der vielleicht eine bessere Performance zeigt.)

Wir benutzen auch die eingebauten Vergleichsfunktionen, die in diesem Fall einen Stringvergleich für die einzelnen Strings im Präfix ausführen.

Ausgestattet mit diesen Komponenten, wird der Code sehr elegant. Hier die Deklarationen:

```
typedef deque<string> Prefix;
map<Prefix, vector<string> > statetab; // Präfix -> Suffixe
```

Die STL stellt ein Template für Deques zur Verfügung. Beachten Sie die Notation `deque<string>`, die das Template auf eine Deque für `string`-Objekte spezialisiert. Da dieser Typ mehrmals in unserem Programm vorkommt, haben wir einen `typedef` verwendet, um ihm den Namen `Prefix` zu geben. Der `map`-Typ, der die Präfixe zusammen mit ihren Suffixen speichert, kommt jedoch nur einmal vor, so daß wir ihm keinen eigenen Namen gegeben haben. Die `map`-Deklaration deklariert eine Variable `statetab`, eine Abbildung von Präfixen auf Vektoren von Strings. Dies ist noch bequemer als in C oder in Java, da wir keine Hashfunktion oder `equals`-Methode bereitstellen müssen.

Die `main`-Routine initialisiert `prefix`, liest den Eingabetext (aus der Standardeingabe, die in der C++-`iostream`-Bibliothek als `cin` bezeichnet wird), fügt das abschließende `NONWORD` hinzu und erzeugt den Ausgabetext genauso wie in den vorigen Versionen:

```
/* main: Generierung eines zufälligen Textes nach */
/*       dem Markov-Chain-Algorithmus */
int main(void)
{
    int nwords = MAXGEN;
    Prefix prefix; // aktueller Eingabe-Präfix

    for (int i = 0; i < NPREF; i++) // initialisiere Start-Präfix
        add(prefix, NONWORD);
    build(prefix, cin);
    add(prefix, NONWORD);
    generate(nwords);
    return 0;
}
```

Die Funktion `build` benutzt die `iostream`-Bibliothek, um immer ein Wort nach dem anderen einzulesen:

```
// build: Lies Eingabetext, und erzeuge Präfix-Tabelle
void build(Prefix& prefix, istream& in)
{
    string buf;
```

```
    while (in >> buf)
        add(prefix, buf);
}
```

Der String `buf` wächst bei Bedarf, um Eingabewörter beliebiger Länge aufnehmen zu können.

Die `add`-Funktion zeigt weitere Vorteile, die die Benutzung der Standard Template Library mit sich bringt:

```
// add: Füge Wort zur Suffix-Liste hinzu, aktualisiere Präfix
void add(Prefix& prefix, const string& s)
{
    if (prefix.size() == NPREF) {
        statetab[prefix].push_back(s);
        prefix.pop_front();
    }
    prefix.push_back(s);
}
```

Diese scheinbar einfachen Anweisungen bewirken eine ganze Menge. Der `map`-Container überlädt den Zugriffsoperator `operator[]` so, daß er sich wie eine `lookup`-Funktion verhält. Der Ausdruck `statetab[prefix]` führt eine Suche in `statetab` mit `prefix` als Schlüssel aus und gibt eine Referenz auf den gewünschten Eintrag zurück. Der Vektor wird dabei erzeugt, wenn er nicht schon existiert. Die `push_back`-Memberfunktionen von `vector` und `deque` fügen ein neues Element an das Ende des Vektors bzw. der Deque hinzu; `pop_front` entfernt das erste Element von einer Deque.

Das Generieren des Ausgabetexts ähnelt den vorherigen Versionen:

```
// generate: Erzeuge Ausgabetext, ein Wort pro Zeile
void generate(int nwords)
{
    Prefix prefix;
    int i;

    for (i = 0; i < NPREF; i++) // Setze auf Start-Präfix zurück
        add(prefix, NONWORD);

    for (i = 0; i < nwords; i++) {
        vector<string>& suf = statetab[prefix];
        const string& w = suf[rand() % suf.size()];
        if (w == NONWORD)
            break;
        cout << w << "\n";
        prefix.pop_front();     // aktualisiere Präfix
        prefix.push_back(w);
    }
}
```

Diese Version scheint vor allem besonders klar und elegant zu sein – der Code ist kompakt, die Datenstruktur sichtbar und die Algorithmen völlig transparent. Leider muß dafür auch ein Preis bezahlt werden: Diese Version läuft viel langsamer als die ursprüngliche C-Version, wobei sie aber nicht die langsamste ist. Wir werden bald auf Performance-Messungen zurückkommen.

Übung 3-5. Die große Stärke der STL ist die Einfachheit, mit der man mit verschiedenen Datenstrukturen experimentieren kann. Modifizieren Sie die C++-Version des Markov-Programms so, daß verschiedene Datenstrukturen benutzt werden, um das Präfix, die Suffix-Liste und die Zustandstabelle zu repräsentieren. Wie verändert sich die Performance für die unterschiedlichen Strukturen?

Übung 3-6. Schreiben Sie eine C++-Version, die nur eigene Klassen und den `string`-Datentyp benutzt, aber keine weiteren Bibliotheksmittel. Vergleichen Sie diese im Stil und in der Performance mit der STL-Version.

3.7 Awk und Perl

Um das Kapitel abzurunden, haben wir das Programm auch in zwei populären Skriptsprachen geschrieben, Awk und Perl. Diese verfügen über die notwendigen Mittel für unsere Applikation: Assoziative Arrays und Stringfunktionen.

Ein *assoziatives Array* ist eine bequeme Verpackung für eine Hashtabelle. Es sieht wie ein Array aus, aber seine Indizes sind beliebige Strings oder Zahlen oder durch Komma getrennte Listen von diesen. Es ist auch eine Art Map von einem Datentyp auf einen anderen. In Awk sind alle Arrays assoziative Arrays; Perl hat sowohl konventionell über Integer indizierte Arrays als auch assoziative Arrays, die »Hashes« genannt werden, ein Name, der die Implementierungstechnik der assoziativen Arrays in Perl andeutet.

Die Awk- und Perl-Implementierungen sind auf Präfixlängen von zwei Wörtern spezialisiert.

```
# markov.awk: Markov-Chain-Algorithmus für 2-Wort-Präfixe

BEGIN { MAXGEN = 10000; NONWORD = "\n"; w1 = w2 = NONWORD }

{ for (i = 1; i <= NF; i++) {   # Lies alle Wörter
        statetab[w1,w2,++nsuffix[w1,w2]] = $i
        w1 = w2
        w2 = $i
    }
}

END {
    statetab[w1,w2,++nsuffix[w1,w2]] = NONWORD # Füge Endemarke hinzu
    w1 = w2 = NONWORD
```

```
    for (i = 0; i < MAXGEN; i++) {   # Generierung
        r = int(rand()*nsuffix[w1,w2]) + 1 # nsuffix >= 1
        p = statetab[w1,w2,r]
        if (p == NONWORD)
            exit
        print p
        w1 = w2   # aktualisiere Präfix
        w2 = p
    }
}
```

Awk ist eine sogenannte *pattern action language* (Muster-Aktions-Sprache): Die Eingabedaten werden zeilenweise gelesen, wobei jede Zeile mit allen Mustern im Awk-Programm verglichen wird und für jede Übereinstimmung die entsprechende Aktion ausgeführt wird. Zusätzlich gibt es die beiden speziellen Muster `BEGIN` und `END`, deren Aktionen vor der ersten Zeile des Eingabetextes und nach der letzten ausgeführt werden.

Eine Aktion ist ein Block mit Anweisungen, die in geschweiften Klammern eingeschlossen sind. In der Awk-Version des Markov-Programms initialisiert der `BEGIN`-Block das Präfix und einige andere Variablen.

Der nächste Block hat kein Muster, so daß er standardmäßig für jede Zeile der Eingabe ausgeführt wird. Awk zerlegt automatisch jede Eingabezeile in Felder (durch Whitespaces getrennte Wörter), auf die über `$1` bis `$NF` zugegriffen wird; die Variable `NF` ist die Anzahl der Felder für die aktuelle Zeile. Die Anweisung

```
statetab[w1,w2,++nsuffix[w1,w2]] = $i
```

initialisiert die Map vom Präfix auf die Suffixe. Das Array `nsuffix` zählt die Anzahl der Suffixe, und das Element `nsuffix[w1,w2]` enthält die Anzahl der Suffixe, die mit dem Präfix `w1,w2` assoziiert sind. Die Suffixe selbst werden in dem Array-Elementen `statetab[w1,w2,1]`, `statetab[w1,w2,2]` und so weiter abgespeichert.

Wenn der `END`-Block ausgeführt wird, ist der gesamte Eingabetext gelesen worden. Zu diesem Zeitpunkt existiert zu jedem Präfix ein Element in `nsuffix`, das die Anzahl der Suffixe für das jeweilige Präfix enthält, die wiederum in `statetab` gefunden werden können.

Die Perl-Version ist ähnlich, benutzt aber anonyme Arrays statt eines dritten Index, um sich die Suffixe zu merken. Außerdem wird auch eine Mehrfachzuweisung zum Aktualisieren des Präfixes eingesetzt. In Perl werden spezielle Zeichen benutzt, um den Typ einer Variablen zu spezifizieren: `$` markiert eine skalare Variable, und `@` ein indiziertes Array, während eckige Klammern `[]` zum Indizieren von Arrays und geschweifte Klammern `{}` zum Indizieren von Hashes verwendet werden.

```
# markov.pl: Markov-Chain-Algorithmus für 2-Wort-Präfixe

$MAXGEN = 10000;
$NONWORD = "\n";
```

```
$w1 = $w2 = $NONWORD;   # Anfangszustand
while (<>) {   # Lies jede Zeile des Eingabetextes
    foreach (split) {
        push(@{$statetab{$w1}{$w2}}, $_);
        ($w1, $w2) = ($w2, $_);   # Mehrfachzuweisung
    }
}
push(@{$statetab{$w1}{$w2}}, $NONWORD); # Füge Endemarke hinzu
$w1 = $w2 = $NONWORD;
for ($i = 0; $i < $MAXGEN; $i++) {
    $suf = $statetab{$w1}{$w2};  # Array-Referenz
    $r = int(rand @$suf); # @$suf ist die Anzahl der Elemente
    exit if (($t = $suf->[$r]) eq $NONWORD);
    print "$t\n";
    ($w1, $w2) = ($w2, $t); # Aktualisiere Präfix
}
```

Wie im vorherigen Programm wir die Map in der Variablen `statetab` gespeichert. Das Herz des Programms ist die folgende Zeile:

```
push(@{$statetab{$w1}{$w2}}, $_);
```

In ihr wird ein neues Suffix an das Ende des (anonymen) Arrays hinzugefügt, das an der Position `statetab{$w1}{$w2}` gespeichert ist. In der Generierungsphase ist `$statetab{$w1}{$w2}` eine Referenz auf ein Array von Suffixen, und `$suf->[$r]` zeigt auf das `r`-te Suffix.

Sowohl das Awk- als auch das Perl-Programm sind verglichen mit den drei ersten Versionen sehr kurz. Es ist aber schwerer, sie so anzupassen, daß sie auch mit Präfixen anderer Länge umgehen können. Der Kern der C++-STL-Implementierung (die `add`- und die `generate`-Funktion) ist von vergleichbarer Länge, wirkt aber klarer. Nichtsdestotrotz sind Skriptsprachen oft eine gute Wahl zum experimentellen Programmieren, zum Erstellen von Prototypen und selbst für Produktionszwecke, wenn die Laufzeit keine entscheidende Rolle spielt.

Übung 3-7. Modifizieren Sie die Awk- und die Perl-Version so, daß sie mit Präfixen jeder beliebigen Länge umgehen können.

3.8 Performance

Wir haben mehrere Implementierungen zu vergleichen. Wir haben die Zeitmessungen für das *Book of Psalms* der *King James Bible* durchgeführt, die 42.685 Wörter umfaßt (5.238 verschiedene Wörter und 22.482 Präfixe). Dieser Text hat genügend sich wiederholende Wortgruppen (»Blessed is the ...«), so daß manche Suffix-Listen mehr als 400 Elemente haben, und es gibt einige hundert Listen mit Dutzenden von Suffixen, wodurch es sich um ein schönes Test-Datenset handelt.

> Blessed is the man of the net. Turn thee unto me, and raise me up, that I may tell all my fears. They looked unto him, he heard. My praise shall be blessed. Wealth and riches shall be saved. Thou hast dealt well with thy hid treasure: they are cast into a standing water, the flint into a standing water, and dry ground into watersprings.

Die Zeiten in der folgenden Tabelle geben die Anzahl der Sekunden wieder, die zum Generieren von 10.000 Wörtern benötigt wurden; ein Rechner ist ein 250 MHz MIPS R1000 unter Irix 6.4 und der andere ist ein 400 MHz Pentium II mit 128 MB Speicher, der unter Windows NT läuft. Die Laufzeit wird fast vollständig durch die Größe des Eingabetextes bestimmt; Die Generierung ist vergleichsweise sehr schnell. Die Tabelle enthält auch die ungefähren Programmgrößen in Quellcodezeilen.

	250 MHz R1000	**400 MHz Pentium II**	**Quellcodezeilen**
C	0,36 s	0,30 s	150
Java	4,9 s	9,2 s	105
C++/STL/`deque`	2,6 s	11,2 s	70
C++/STL/`list`	1,7 s	1,5 s	70
Awk	2,2 s	2,1 s	20
Perl	1,8 s	1,0 s	18

Die C- und die C++-Version wurden mit einem optimierenden Compiler übersetzt, während beim Ausführen der Java-Version ein Just-In-Time-Compiler aktiviert war. Die Zeiten für das C- und das C++-Programm unter Irix sind die schnellsten, die wir beim Ausprobieren von drei verschiedenen Compilern erhalten haben. Ähnliche Ergebnisse sind auf einer Sun SPARC und einem DEC Alpha-Rechner ermittelt worden. Die C-Version des Programms ist mit Abstand die schnellste; Perl kommt als zweites. Die Zeiten in der Tabelle sind lediglich eine Momentaufnahme unserer Erfahrungen mit einer bestimmten Menge von Compilern und Bibliotheken, so daß Sie unter Umständen in Ihrer Umgebung sehr stark abweichende Ergebnisse erhalten können.

Etwas stimmt offensichtlich mit der STL-`deque`-Version unter Windows ganz und gar nicht. Experimente haben gezeigt, daß die `deque`, die das Präfix repräsentiert, die meiste Laufzeit verschlingt, obwohl sie nie mehr als zwei Elemente enthält. Wir erwarten eigentlich, daß die zentrale Datenstruktur, die Map, die Laufzeit dominiert. Der Wechsel von einer `deque` auf eine `list` (die in der STL als doppelt-verkettete Liste implementiert ist) verbessert das Laufzeitverhalten dramatisch. Auf der anderen Seite machte der Wechsel vom `map`-Container auf einen (nicht standardisierten) `hash`-Container keinen Unterschied unter Irix; unter Windows war kein `hash`-Container verfügbar. Es ist ein Beweis für die fundamentale Schönheit des STL-Designs, daß diese Änderungen bloß erforderten, daß das Wort `deque` durch das Wort `list` bzw. das Wort `map` durch das Wort `hash` an zwei Stellen ersetzt und das Programm anschließend nur neu kompiliert

werden mußte. Wie man sieht, leidet die STL, die eine sehr neue Komponente von C++ ist, noch unter unausgereiften Implementierungen. Performance-Unterschiede zwischen Implementierungen mit der STL oder mit eigenen Datenstrukturen sind nicht vorhersagbar. Das gleiche gilt für Java, wo die Implementierungen auch noch ständigen Änderungen unterworfen sind.

Es gibt einige interessante Herausforderungen, wenn man ein Programm testen möchte, das eine umfangreiche zufällige Ausgabe generieren soll. Wie wissen wir, ob es überhaupt funktioniert? Wie wissen wir, daß es in jedem Fall funktioniert? Kapitel 6, in dem wir das Testen von Programmen diskutieren, enthält einige Vorschläge hierzu und beschreibt, wie wir das Markov-Programm getestet haben.

3.9 Was wir gelernt haben

Das Markov-Programm hat eine lange Geschichte. Die erste Version wurde von Don P. Mitchell geschrieben, von Bruce Ellis angepaßt und für humoristische Aktivitäten in den 80er Jahren eingesetzt. Es war danach vergessen, bis wir auf die Idee kamen, es in einem Universitätskurs als Veranschaulichung für das Entwerfen von Programmen einzusetzen. Anstatt das Original aus seiner Versenkung zu holen, haben wir es noch einmal ganz von vorne in C geschrieben, um unsere Erinnerungen über die ganzen Fragen aufzufrischen, die dabei auftauchen. Danach haben es noch in einigen anderen Sprachen geschrieben, wobei wir die jeweiligen Idiome der Sprachen benutzt haben, um die gleiche zugrundeliegende Idee auszudrücken.

Die grundlegende Idee ist über die ganze Zeit hinweg jedoch immer die gleiche geblieben. Die erste Version hat den gleichen Ansatz benutzt, den wie wir Ihnen hier gezeigt haben, obwohl sie eine zweite Hashtabelle für die einzelnen Wörter benutzt hat. Müßten wir das Programm noch einmal schreiben, würden wir wahrscheinlich nicht viel ändern. Das Design des Programms ist in seiner Datenstruktur verwurzelt. Die Datenstruktur definiert zwar nicht jede Einzelheit, gibt aber die allgemeine Lösung vor.

Einige Wahlen bei der Datenstruktur machen keinen großen Unterschied, wie beispielsweise der Einsatz von Listen oder dynamisch wachsender Arrays. Einige Implementierungen generalisieren besser als andere – der Perl- und der Awk-Code könnten leicht an Präfixe mit einem oder drei Wörtern angepaßt werden, sie jedoch hierfür zu parametrisieren, wäre sehr beschwerlich. Wie es sich für objekt-orientierte Sprachen gehört, reichen kleine Änderungen an den C++- und Java-Implementierungen, um die Datenstrukturen auch für andere Eingaben als englischen Text sinnvoll benutzen zu können, zum Beispiel für Programme (bei denen Whitespaces signifikant sind) oder für Musiknoten oder sogar für Mausklicks und Menü-Auswahlen zum Generieren von Testsequenzen.

Natürlich gibt es, auch wenn die Datenstrukturen in etwa gleich sind, doch große Unterschiede im Aussehen der Programme, in der Größe des Quellcodes und in ihrer Performance. Ganz grob läßt sich sagen, daß High-Level-Programmiersprachen zu langsameren Programmen führen als Low-Level-Sprachen, wobei es unklug wäre, dies anders als qualitativ zu verallgemeinern. Große Möglichkeiten der Sprachen, wie die C++-STL oder die assoziativen Arrays und das String-Handling der Skriptsprachen, können zu kompakterem Code und kürzeren Entwicklungszeiten führen. Diese haben aber ihren Preis, auch wenn die Performance-Nachteile für die meisten Programme keine große Rolle spielen, wie beim Markov-Programm, das nur wenige Sekunden läuft.

Weniger klar ist jedoch, wie der Verlust der Kontrolle und der Einsicht in den Code einzuschätzen ist, wenn die vom System zur Verfügung gestellte Funktionalität so groß wird, daß niemand mehr weiß, was genau im Hintergrund passiert. Dies ist der Fall bei der STL-Version; ihre Performance ist nicht vorhersagbar[1] und es gibt keinen leichten Ausweg aus dieser Situation. Eine unausgereifte Implementierung, die wir benutzt haben, benötigt eine Überarbeitung, bevor wir sie mit unserem Programm benutzen konnten. Aber nur die wenigsten von uns haben die Ressourcen und die Energie, solche Probleme zu verfolgen und zu beheben.

Dies ist eine immer wichtiger werdende und sich ausbreitende Sorge bei der Entwicklung von Software: Wenn Bibliotheken, Schnittstellen und Werkzeuge immer komplizierter werden, werden sie auch schlechter verstanden und schlechter kontrollierbar. Wenn alles funktioniert, können mächtige Programmierumgebungen sehr produktiv sein, wenn aber etwas schiefgeht, gibt es daraus meist keinen einfachen Ausweg. Unter Umständen merkt man nicht einmal, daß etwas nicht stimmt, wenn das Problem die Performance oder ein subtiler Logikfehler ist.

Das Design und die Implementierung des Markov-Programms hat uns eine Menge Lektionen für größere Programme gelehrt. Als erstes ist die Wahl einfacher Algorithmen und Datenstrukturen zu nennen, die einfachsten, die die Arbeit in angemessener Zeit für die erwartete Problemgröße erledigen können. Wenn jemand anderes diese schon geschrieben und für Sie in eine Bibliothek gepackt hat, ist das noch besser; zum Beispiel hat unsere C++-Implementierung davon profitiert.

Brooks Ratschlag befolgend finden wir es am besten, wenn man mit dem Design der Datenstrukturen beginnt, wobei man im Hinterkopf immer parat hat, welche Algorithmen auf diese angewandt werden können. Stehen die Datenstrukturen einmal fest, schreibt sich der Code fast von selbst.

1. Dies ist nur begrenzt richtig, da bei der Definition des Standards für alle wichtigen Operationen der STL auch die maximale Laufzeitkomplexität festgelegt wurde. (A. d. Ü.)

Es ist schwer, ein Programm vollständig zu entwerfen und dann zu programmieren; die Programmierung echter Programme geschieht durch Iterationen und Experimente. Der Akt des Programmierens zwingt einen, die Entscheidungen zu klären, über die zuvor (leichtfertig) hinweggegangen wurde. Dies war ganz bestimmt der Fall bei unserem Markov-Programm, bei dem viele Einzelheiten nach und nach geändert wurden. Soweit nur irgend möglich sollten Sie mit etwas ganz Einfachem anfangen und es dann so verbessern, wie Ihre Erfahrung damit es Ihnen diktiert. Wenn unser Ziel gewesen wäre, nur persönlich zu unserem Spaß ein Programm für den Markov-Chain-Algorithmus zu schreiben, hätten wir es mit großer Sicherheit in Awk oder Perl geschrieben – wenn auch nicht so aufpoliert wie die Programme, die Sie hier gesehen haben –, und dabei hätten wir es dann belassen.

Produktionscode bedeutet jedoch, einen wesentlich größeren Aufwand zu betreiben als bei Prototypen. Wenn wir die hier gezeigten Programme als *Produktionscode* ansehen (da sie aufpoliert und intensiv getestet wurden), erfordert Produktionsqualität eine oder zwei Größenordnungen mehr Aufwand als ein Programm für den persönlichen Gebrauch.

Übung 3-8. Wir haben eine Vielzahl von Markov-Programmen gesehen, die in den unterschiedlichsten Programmiersprachen programmiert wurden, einschließlich Scheme, Tcl, Prolog, Python, Generic Java, ML und Haskell; jedes mit seinen eigenen Problemen und Vorteilen. Implementieren Sie das Programm in Ihrer Lieblingssprache, und vergleichen Sie es hinsichtlich des Aussehens und der Performance.

Literatur

Die Standard Template Library wird in vielen Büchern beschrieben, wie beispielsweise *»Generic Programming and the STL«* von Matthew Austern (Addison-Wesley, 1998). Die definitive Referenz für C++ selbst ist *»The C++ Programming Language«* von Bjarne Stroustrup (3. Ausgabe, Addison-Wesley, 1997).[1] Für Java empfehlen wir *»The Java Programming Language«* von Ken Arnold und James Gosling (2. Auflage, Addison-Wesley, 1998). Die beste Beschreibung von Perl ist *»Programming Perl«* von Larry Wall, Tom Christiansen und Randal Schwartz (2. Auflage, O'Reilly, 1996).

Die Idee hinter *Entwurfsmuster* ist, daß es nur einige wenige Design-Konstrukte in den meisten Programmen gibt, genauso wie es auch nur einige wenige grundlegende Datenstrukturen gibt. Sehr salopp gesprochen ist es das Analogon für das Design zu unserer Diskussion über Code-Idiome im Kapitel 1. Die Standardreferenz ist *»Design Patterns: Elements of Reusable Object-Oriented Software«* von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides (Addison-Wesley, 1995).[2]

1. Deutsche Übersetzung: *»Die C++ Programmiersprache«* (Addison-Wesley, 1998)
2. Deutsche Übersetzung: *»Entwurfsmuster«* (Addison-Wesley, 1996)

Die unglaublichen Abenteuer des Markov-Programms, ursprünglich `shaney` genannt, sind der *»Computing Recreations«*-Kolumne der Juni-Ausgabe (1989) von *»Scientific American«* zu finden. Der Artikel wurde in *»The Magic Machine«* von A. K. Dewdney (W. H. Freeman, 1990) neu veröffentlicht.