3

Design and Implementation

Show me your flowcharts and conceal your tables, and I shall continue to be mystijied. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.

Frederick P. Brooks, Jr., ***The Mythical Man Month***

As the quotation from Brooks's classic book suggests, the design of the data structures is the central decision in the creation of a program. Once the data structures are laid out, the algorithms tend to fall into place, and the coding is comparatively easy.

This point of view is oversimplified but not misleading. In the previous chapter we examined the basic data structures that are the building blocks of most programs. In this chapter we will combine such structures as we work through the design and implementation of a modest-sized program. We will show how the problem influences the data structures, and how the code that follows is straightforward once we have the data structures mapped out.

One aspect of this point of view is that the choice of programming language is relatively unimportant to the overall design. We will design the program in the abstract and then write it in C, Java, C++, Awk, and Perl. Comparing the implementations demonstrates how languages can help or hinder, and ways in which they are unimportant. Program design can certainly be colored by a language but is not usually dominated by it.

The problem we have chosen is unusual, but in basic form it is typical of many programs: some data comes in, some data goes out, and the processing depends on a little ingenuity.

Specifically, we're going to generate random English text that reads well. If we emit random letters or random words, the result will be nonsense. For example, a program that randomly selects letters (and blanks. to separate words) might produce this:

```
xptmxgn xusaja  afqnzgxl  lhidlwcd rjdjuvpydrlwnjy
```

which is not very convincing. If we weight the letters by their frequency of appearance in English text, we might get this:

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

which isn't a great deal better. Words chosen from the dictionary at random don't make much more sense:

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

For better results, we need a statistical model with more structure. such as the frequency of appearance of whole phrases. But where can we find such statistics?

We could grab a large body of English and study it in detail, but there is an easier and more entertaining approach. The key observation is that we can use any existing text to construct a statistical model of the language ***as used*** in ***that text,*** and from that generate random text that has similar statistics to the original.

3.1 The Markov Chain Algorithm

An elegant way to do this sort of processing is a technique called a ***Markov chain algorithm.*** If we imagine the input as a sequence of overlapping phrases, the algorithm divides each phrase into two parts, a multi-word ***prefix*** and a single ***suffix*** word that follows the prefix. A Markov chain algorithm emits output phrases by randomly choosing the suffix that follows the prefix, according to the statistics of (in our case) the original text. Three-word phrases work well—a two-word prefix is used to select the suffix word:

```
set w1 and w2 to the first two words in the text
print w1 and w2
loop:
    randomly choose w3, one of the successors of prefix w1 w2 in the text
    print w3
    replace w1 and w2 by w2 and w3
    repeat loop
```

To illustrate, suppose we want to generate random text based on a few sentences paraphrased from the epigraph above, using two-word prefixes:

```
Show your flowcharts and conceal your tables and I will be
mystified.  Show your tables and your flowcharts will be
obvious.  (end)
```

These are some of the pairs of input words and the words that follow them:

Input prefix:	*Suffix words that follow:*
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.	(*end*)

A Markov algorithm processing this text will begin by printing **Show your** and will then randomly pick either **flowcharts** or **tables.** If it chooses the former, the current prefix becomes **your flowcharts** and the next word will be **and** or **will.** If it chooses **tables,** the next word will be **and.** This continues until enough output has been generated or until the end-marker is encountered as a suffix.

Our program will read a piece of English text and use a Markov chain algorithm to generate new text based on the frequency of appearance of phrases of a fixed length. The number of words in the prefix, which is two in our example, is a parameter. Making the prefix shorter tends to produce less coherent prose; making it longer tends to reproduce the input text verbatim. For English text, using two words to select a third is a good compromise; it seems to recreate the flavor of the input while adding its own whimsical touch.

What is a word? The obvious answer is a sequence of alphabetic characters, but it is desirable to leave punctuation attached to the words so **"words"** and **"words."** are different. This helps to improve the quality of the generated prose by letting punctuation, and therefore (indirectly) grammar, influence the word choice, although it also permits unbalanced quotes and parentheses to sneak in. We will therefore define a "word" as anything between white space, a decision that places no restriction on input language and leaves punctuation attached to the words. Since most programming languages have facilities to split text into white-space-separated words, this is also easy to implement.

Because of the method, all words, all two-word phrases, and all three-word phrases in the output must have appeared in the input, but there should be many four-word and longer phrases that are synthesized. Here are a few sentences produced by the program we will develop in this chapter, when given the text of Chapter VII of ***The Sun Also Rises*** by Ernest Hemingway:

> 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."

We were lucky here that punctuation came out correctly; that need not happen.

3.2 Data Structure Alternatives

How much input do we intend to deal with? How fast must the program run? It seems reasonable to ask our program to read in a whole book, so we should be prepared for input sizes of $n = 100{,}000$ words or more. The output will be hundreds or perhaps thousands of words, and the program should run in a few seconds instead of minutes. With 100,000 words of input text, n is fairly large so the algorithms can't be too simplistic if we want the program to be fast.

The Markov algorithm must see all the input before it can begin to generate output. so it must store the entire input in some form. One possibility is to read the whole input and store it in a long string, but we clearly want the input broken down into words. If we store it as an array of pointers to words, output generation is simple: to produce each word, scan the input text to see what possible suffix words follow the prefix that was just emitted, and then choose one at random. However, that means scanning all 100,000 input words for each word we generate; 1,000 words of output means hundreds of millions of string comparisons. which will not be fast.

Another possibility is to store only unique input words, together with a list of where they appear in the input so that we can locate successor words more quickly. We could use a hash table like the one in Chapter 2, but that version doesn't directly address the needs of the Markov algorithm, which must quickly locate all the suffixes of a given prefix.

We need a data structure that better represents a prefix and its associated suffixes. The program will have two passes, an input pass that builds the data structure representing the phrases, and an output pass that uses the data structure to generate the random output. In both passes, we need to look up a prefix (quickly): in the input pass to update its suffixes, and in the output pass to select at random from the possible suffixes. This suggests a hash table whose keys are prefixes and whose values are the sets of suffixes for the corresponding prefixes.

For purposes of description, we'll assume a two-word prefix, so each output word is based on the pair of words that precede it. The number of words in the prefix doesn't affect the design and the programs should handle any prefix length, but selecting a number makes the discussion concrete. The prefix and the set of all its possible suffixes we'll call a state, which is standard terminology for Markov algorithms.

Given a prefix, we need to store all the suffixes that follow it so we can access them later. The suffixes are unordered and added one at a time. We don't know how many there will be, so we need a data structure that grows easily and efficiently. such as a list or a dynamic array. When we are generating output, we need to be able to choose one suffix at random from the set of suffixes associated with a particular prefix. Items are never deleted.

What happens if a phrase appears more than once? For example, 'might appear twice' might appear twice but 'might appear once' only once. This could be represented by putting 'twice' twice in the suffix list for 'might appear' or by putting it in once, with an associated counter set to 2. We've tried it with and without counters;

without is easier. since adding a suffix doesn't require checking whether it's there already, and experiments showed that the difference in run-time was negligible.

In summary, each state comprises a prefix and a list of suffixes. This information is stored in a hash table, with prefix as key. Each prefix is a fixed-size set of words. If a suffix occurs more than once for a given prefix, each occurrence will be included separately in the list.

The next decision is how to represent the words themselves. The easy way is to store them as individual strings. Since most text has many words appearing multiple times, it would probably save storage if we kept a second hash table of single words, so the text of each word was stored only once. This would also speed up hashing of prefixes, since we could compare pointers rather than individual characters: unique strings have unique addresses. We'll leave that design as an exercise; for now, strings will be stored individually.

3.3 Building the Data Structure in C

Let's begin with a C implementation. The first step is to define some constants.

```
enum {
    NPREF   = 2,     /* number of prefix words */
    NHASH   = 4093,  /* size of state hash table array */
    MAXGEN  = 10000 /* maximum words generated */
};
```

This declaration defines the number of words **(NPREF)** for the prefix, the size of the hash table array **(NHASH).** and an upper limit on the number of words to generate **(MAXGEN).** If **NPREF** is a compile-time constant rather than a run-time variable, storage management is simpler. The array size is set fairly large because we expect to give the program large input documents, perhaps a whole book. We chose **NHASH** = 4093 so that if the input has 10,000 distinct prefixes (word pairs). the average chain will be very short, two or three prefixes. The larger the size, the shorter the expected length of the chains and thus the faster the lookup. This program is really a toy, so the performance isn't critical, but if we make the array too small the program will not handle our expected input in reasonable time; on the other hand, if we make it too big it might not fit in the available memory.

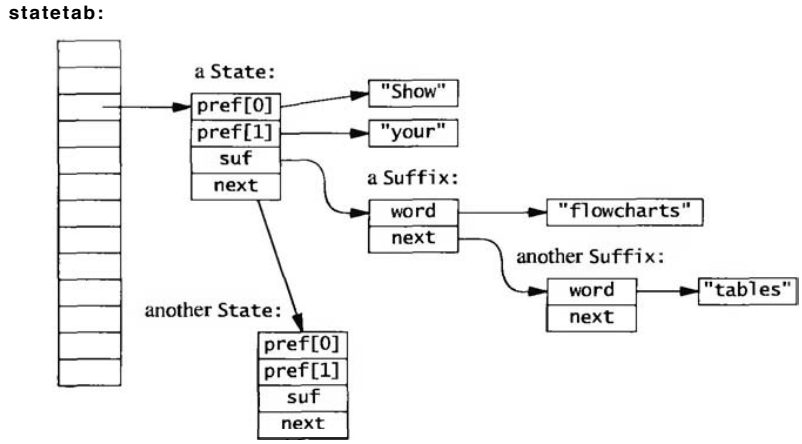
The prefix can be stored as an array of words. The elements of the hash table will be represented as a **State** data type, associating the **Suffix** list with the prefix:

```
typedef struct State State;
typedef struct Suffix Suffix;
struct State {  /* prefix + suffix list */
    char    *pref[NPREF];   /* prefix words */
    Suffix  *suf;           /* list of suffixes */
    State   *next;          /* next in hash table */
};
```

```
struct Suffix { /* list of suffixes */
    char    *word;          /* suffix */
    Suffix  *next;          /* next in list of suffixes */
};

State   *statetab[NHASH];   /* hash table of states */
```

Pictorially, the data structures look like this:



We need a hash function for prefixes, which are arrays of strings. It is simple to modify the string hash function fmm Chapter 2 to loop over the strings in the array, thus in effect hashing the concatenation of the strings:

```
/* hash: compute hash value for array of 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;
}
```

A similar modification to the lookup routine completes the implementation of the hash table:

```
/* lookup: search for prefix; create if requested. */
/*  returns pointer if present or created; NULL if not. */
/*  creation doesn't strdup so strings mustn't change later. */
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)      /* found it */
            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;
}
```

Notice that lookup doesn't make a copy of the incoming strings when it creates a new state; it just stores pointers in sp->pref[]. Callers of lookup must guarantee that the data won't be overwritten later. For example, if the strings are in an I/O buffer, a copy must be made before lookup is called; otherwise, subsequent input could overwrite the data that the hash table points to. Decisions about who owns a resource shared across an interface arise often. We will explore this topic at length in the next chapter.

Next we need to build the hash table as the file is read:

```
/* build: read input, build prefix table */
void build(char *prefix[NPREF], FILE *f)
{
    char buf[100], fmt[10];

    /* create a format string; %s could overflow buf */
    sprintf(fmt, "%%%ds", sizeof(buf)-1);
    while (fscanf(f, fmt, buf) != EOF)
        add(prefix, estrdup(buf));
}
```

The peculiar call to sprintf gets around an irritating problem with fscanf, which is otherwise perfect for the job. A call to fscanf with format %s will read the next white-space-delimited word from the file into the buffer, but there is no limit on size: a long word might overflow the input buffer, wreaking havoc. If the buffer is 100

bytes long (which is far beyond what we expect ever to appear in normal text), we can use the format %99s (leaving one byte for the terminal '\0'), which tells fscanf to stop after 99 bytes. A long word will be broken into pieces, which is unfortunate but safe. We could declare

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

but that requires two constants for one arbitrary decision—the size of the buffer—and introduces the need to maintain their relationship. The problem can be solved once and for all by creating the format string dynamically with sprintf, so that's the approach we take.

The two arguments to build are the prefix array holding the previous NPREF words of input and a FILE pointer. It passes the prefix and a copy of the input word to add, which adds the new entry to the hash table and advances the prefix:

```
/* add: add word to suffix list, update prefix */
void add(char *prefix[NPREF], char *suffix)
{
    State *sp;

    sp = lookup(prefix, 1);  /* create if not found */
    addsuffix(sp, suffix);
    /* move the words down the prefix */
    memmove(prefix, prefix+1, (NPREF-1)*sizeof(prefix[0]));
    prefix[NPREF-1] = suffix;
}
```

The call to memmove is the idiom for deleting from an array. It shifts elements 1 through NPREF-1 in the prefix down to positions 0 through NPREF-2, deleting the first prefix word and opening a space for a new one at the end.

The addsuffix routine adds the new suffix:

```
/* addsuffix: add to state. suffix must not change later */
void addsuffix(State *sp, char *suffix)
{
    Suffix *suf;

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

We split the action of updating the state into two functions: add performs the general service of adding a suffix to a prefix, while addsuffix performs the implementation-specific action of adding a word to a suffix list. The add routine is used by build, but addsuffix is used internally only by add; it is an implementation detail that might change and it seems better to have it in a separate function, even though it is called in only one place.

3.4 Generating Output

With the data structure built, the next step is to generate the output. The basic idea is as before: given a prefix, select one of its suffixes at random, print it, then advance the prefix. This is the steady state of processing; we must still figure out how to start and stop the algorithm. Starting is easy if we remember the words of the first prefix and begin with them. Stopping is easy, too. We need a marker word to terminate the algorithm. After all the regular input, we can add a terminator. a "word" that is guaranteed not to appear in any input:

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

NONWORD should be some value that will never be encountered in regular input. Since the input words are delimited by white space, a "word" of white space will serve, such as a newline character:

```
char NONWORD[] = "\n";  /* cannot appear as real word */
```

One more worry: what happens if there is insufficient input to start the algorithm? There are two approaches to this sort of problem, either exit prematurely if there is insufficient input, or arrange that there is always enough and don't bother to check. In this program, the latter approach works well.

We can initialize building and generating with a fabricated prefix, which guarantees there is always enough input for the program. To prime the loops, initialize the prefix array to be all NONWORD words. This has the nice benefit that the first word of the input file will be the first *suffix* of the fake prefix, so the generation loop needs to print only the suffixes it produces.

In case the output is unmanageably long, we can terminate the algorithm after some number of words are produced or when we hit NONWORD as a suffix, whichever comes first.

Adding a few NONWORDs to the ends of the data simplifies the main processing loops of the program significantly; it is an example of the technique of adding *sentinel* values to mark boundaries.

As a rule, try to handle irregularities and exceptions and special cases in data. Code is harder to get right so the control flow should be as simple and regular as possible.

The generate function uses the algorithm we sketched originally. It produces one word per line of output, which can be grouped into longer lines with a word processor; Chapter 9 shows a simple formatter called fmt for this task.

With the use of the initial and final NONWORD strings, generate starts and stops properly:

```
/* generate: produce output, one word per line */
void generate(int nwords)
{
    State *sp;
    Suffix *suf;
    char *prefix[NPREF], *w;
    int i, nmatch;

    for (i = 0; i < NPREF; i++) /* reset initial prefix */
        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) /* prob = 1/nmatch */
                w = suf->word;
        if (strcmp(w, NONWORD) == 0)
            break;
        printf("%s\n", w);
        memmove(prefix, prefix+1, (NPREF-1)*sizeof(prefix[0]));
        prefix[NPREF-1] = w;
    }
```

Notice the algorithm for selecting one item at random when we don't know how many items there are. The variable `nmatch` counts the number of matches as the list is scanned. The expression

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

increments `nmatch` and is then true with probability `1/nmatch`. Thus the first matching item is selected with probability 1. the second will replace it with probability 1/2, the third will replace the survivor with probability 1/3, and so on. At any time, each one of the k matching items seen so far has been selected with probability $1/k$.

At the beginning. we set the `prefix` to the starting value, which is guaranteed to be installed in the hash table. The first `Suffix` values we find will be the first words of the document. since they are the unique follow-on to the starting prefix. After that, random suffixes will be chosen. The loop calls `lookup` to find the hash table entry for the current `prefix`. then chooses a random suffix, prints it, and advances the prefix.

If the suffix we choose is `NONWORD`, we're done, because we have chosen the state that corresponds to the end of the input. If the suffix is not `NONWORD`, we print it, then drop the first word of the prefix with a call to `memmove`, promote the suffix to be the last word of the prefix, and loop.

Now we can put all this together into a `main` routine that reads the standard input and generates at most a specified number of words:

```
/* markov main: markov-chain random text generation */
int main(void)
{
    int i, nwords = MAXGEN;
    char *prefix[NPREF];        /* current input prefix */
    for (i = 0; i < NPREF; i++) /* set up initial prefix */
        prefix[i] = NONWORD;
    build(prefix, stdin);
    add(prefix, NONWORD);
    generate(nwords);
    return 0;
}
```

This completes our C implementation. We will return at the end of the chapter to a comparison of programs in different languages. The great strengths of C are that it gives the programmer complete control over implementation, and programs written in it tend to be fast. The cost, however, is that the C programmer must do more of the work, allocating and reclaiming memory, creating hash tables and linked lists, and the like. C is a razor-sharp tool, with which one can create an elegant and efficient program or a bloody mess.

Exercise 3-1. The algorithm for selecting a random item from a list of unknown length depends on having a good random number generator. Design and carry out experiments to determine how well the method works in practice. □

Exercise 3-2. If each input word is stored in a second hash table, the text is only stored once, which should save space. Measure some documents to estimate how much. This organization would allow us to compare pointers rather than strings in the hash chains for prefixes, which should run faster. Implement this version and measure the change in speed and memory consumption. □

Exercise 3-3. Remove the statements that place sentinel NONWORDs at the beginning and end of the data, and modify generate so it starts and stops properly without them. Make sure it produces correct output for input with 0, 1, 2, 3, and 4 words. Compare this implementation to the version using sentinels. □

3.5 Java

Our second implementation of the Markov chain algorithm is in Java. Object-oriented languages like Java encourage one to pay particular attention to the interfaces between the components of the program. which are then encapsulated as independent data items called objects or classes, with associated functions called methods.

Java has a richer library than C, including a set of *container* classes to group existing objects in various ways. One example is a Vector that provides a dynamically-growable array that can store any Object type. Another example is the Hashtable

class, with which one can store and retrieve values of one type using objects of another type as keys.

In our application, Vectors of strings are the natural choice to hold prefixes and suffixes. We can use a Hashtable whose keys are prefix vectors and whose values are suffix vectors. The terminology for this type of construction is a map from prefixes to suffixes; in Java, we need no explicit State type because Hashtable implicitly connects (maps) prefixes to suffixes. This design is different from the C version, in which we installed State structures that held both prefix and suffix list, and hashed on the prefix to recover the full State.

A Hashtable provides a put method to store a key-value pair, and a get method to retrieve the value for a key:

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

Our implementation has three classes. The first class, Prefix, holds the words of the prefix:

```
class Prefix {
    public Vector pref; // NPREF adjacent words from input
    ...
```

The second class, Chain, reads the input, builds the hash table, and generates the output; here are its class variables:

```
class Chain {
    static final int NPREF = 2; // size of prefix
    static final String NONWORD = "\n";
                    // "word" that can't appear
    Hashtable statetab = new Hashtable();
                    // key = Prefix, value = suffix Vector
    Prefix prefix = new Prefix(NPREF, NONWORD);
                    // initial prefix
    Random rand = new Random();
    ...
```

The third class is the public interface; it holds main and instantiates a Chain:

```
class Markov {
    static final int MAXGEN = 10000; // maximum words generated
    public static void main(String[] args) throws IOException
    {
        Chain chain = new Chain() ;
        int nwords = MAXGEN;

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

When an instance of class Chain is created, it in turn creates a hash table and sets up the initial prefix of NPREF NONWORDs. The build function uses the library function StreamTokenizer to parse the input into words separated by white space characters. The three calls before the loop set the tokenizer into the proper state for our definition of "word."

```
// Chain build: build State table from input stream
void build(InputStream in) throws IOException
{
    StreamTokenizer st = new StreamTokenizer(in);

    st.resetSyntax();                       // remove default rules
    st.wordChars(0, Character.MAX_VALUE); // turn on all chars
    st.whitespaceChars(0, ' ');           // except up to blank
    while (st.nextToken() != st.TT_EOF)
        add(st.sval);
    add(NONWORD);
}
```

The add function retrieves the vector of suffixes for the current prefix from the hash table; if there are none (the vector is null), add creates a new vector and a new prefix to store in the hash table. In either case, it adds the new word to the suffix vector and advances the prefix by dropping the first word and adding the new word at the end.

```
// Chain add: add word to suffix list, update 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);
}
```

Notice that if suf is null, add installs a new Prefix in the hash table, rather than prefix itself. This is because the Hashtable class stores items by reference, and if we don't make a copy, we could overwrite data in the table. This is the same issue that we had to deal with in the C program.

The generation function is similar to the C version, but slightly more compact because it can index a random vector element directly instead of looping through a list.

```
// Chain generate: generate output words
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);
    }
}
```

The two constructors of Prefix create new instances from supplied data. The first copies an existing Prefix, and the second creates a prefix from n copies of a string; we use it to make NPREF copies of NONWORD when initializing:

```
// Prefix constructor: duplicate existing prefix
Prefix(Prefix p)
{
    pref = (Vector) p.pref.clone();
}

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

Prefix also has two methods, hashCode and equals, that are called implicitly by the implementation of Hashtable to index and search the table. It is the need to have an explicit class for these two methods for Hashtable that forced us to make Prefix a full-fledged class. rather than just a Vector like the suffix.

The hashCode method builds a single hash value by combining the set of hashCodes for the elements of the vector:

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

// Prefix hashCode: generate hash from all prefix words
public int hashCode()
{
    int h = 0;
    for (int i = 0; i < pref.size(); i++)
        h = MULTIPLIER * h + pref.elementAt(i).hashCode();
    return h;
}
```

and equals does an elementwise comparison of the words in two prefixes:

```
// Prefix equals: compare two prefixes for equal words
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;
}
```

The Java program is significantly smaller than the C program and takes care of more details; Vectors and the Hashtable are the obvious examples. In general, storage management is easy since vectors grow as needed and garbage collection takes care of reclaiming memory that is no longer referenced. But to use the Hashtable class, we still need to write functions hashCode and equals, so Java isn't taking care of all the details.

Comparing the way the C and Java programs represent and operate on the same basic data structure, we see that the Java version has better separation of functionality. For example, to switch from Vectors to arrays would be easy. In the C version. everything knows what everything else is doing: the hash table operates on arrays that are maintained in various places, lookup knows the layout of the State and Suffix structures, and everyone knows the size of the prefix array.

```
% 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.
```

Exercise 3-4. Revise the Java version of markov to use an array instead of a Vector for the prefix in the State class. □