$n \lg n$. If the keys are long but there are relatively few of them, then k is large and n relatively small, and other methods (such as mergesort) will outperform radix sort; but if k is small and there are a large number of keys, then radix sort will be faster than any other method we have studied.

racter.

n the function
to 26 for 'z' or
on returns 27.
for ASCII, not

Exercises 8.5

E1. Trace the action of radix sort on the list of 14 names used to trace other sorting methods:

Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan

E2. Trace the action of radix sort on the following list of seven numbers considered as two-digit integers:

26 33 35 29 19 12 22

E3. Trace the action of radix sort on the preceding list of seven numbers considered as six-digit binary integers.

st. A project
vay that will

Programming Projects 8.5

P1. Design, program, and test a version of radix sort that is implementation independent, with alphabetic keys.

P2. The radix-sort program presented in the book is very inefficient, since its implementation-independent features force a large amount of data movement. Design a project that is implementation dependent and saves all the data movement. In Rethread you need only link the rear of one queue to the front of the next. Compare the performance of this version with that of other sorting methods for linked lists.

list has been

are empty. */

8.6 Hashing

8.6.1 Sparse Tables

1. Index Functions

We can continue to exploit table lookup even in situations where the key is no longer an index that can be used directly as in array indexing. What we can do is to set up a one-to-one correspondence between the keys by which we wish to retrieve information and indices that we can use to access an array. The index function that we produce will be somewhat more complicated than those of previous sections,

s the number
me for all our
of a key. The

The only difficulty arises when the number of possible keys exceeds the amount of space available for our table. If, for example, our keys are alphabetical words of eight letters, then there are $26^8 \approx 2 \times 10^{11}$ possible keys, a number likely greater than the number of positions that will be available in high-speed memory. In practice, however, only a small fraction of these keys will actually occur. That is, the table is ***sparse***. Conceptually, we can regard it as indexed by a very large set, but with relatively few positions actually occupied. In C, for example, we might think in terms of conceptual declarations such as

```
typedef sparse table of Entry type SparseTable[MAXENTRY];
```

Even though it may not be possible to implement a declaration such as this directly, it is often helpful in problem solving to begin with such a picture, and only slowly tie down the details of how it is put into practice.

2. Hash Tables

index function not one to one

The idea of a ***hash table*** (such as the one shown in Figure 8.12) is to allow many of the different possible keys that might occur to be mapped to the same location in an array under the action of the index function. Then there will be a possibility that two records will want to be in the same place, but if the number of records that actually occur is small relative to the size of the array, then this possibility will cause little loss of time. Even when most entries in the array are occupied, hash methods can be an effective means of information retrieval.

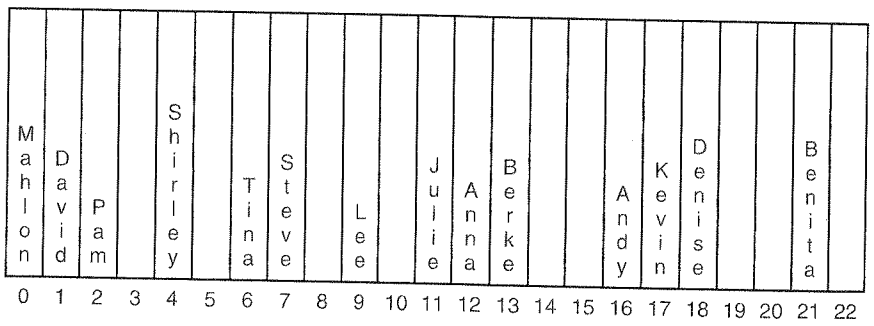


Figure 8.12. A hash table

hash function

We begin with a ***hash function*** that takes a key and maps it to some index in the array. This function will generally map several different keys to the same index. If the desired record is in the location given by the index, then our problem is solved; otherwise we must use some method to resolve the ***collision*** that may have occurred between two records wanting to go to the same location. There are thus two questions we must answer to use hashing. First, we must find good hash functions, and, second, we must determine how to resolve collisions.

collision

Before approaching these questions, let us pause to outline informally the steps needed to implement hashing.

3. Algorithm Outlines

keys in table

First, an array must be declared that will hold the hash table. With ordinary arrays the keys used to locate entries are usually the indices, so there is no need to keep them within the array itself, but for a hash table, several possible keys will correspond to the same index, so one field within each record in the array must be reserved for the key itself.

initialization

Next, all locations in the array must be initialized to show that they are empty. How this is done depends on the application; often it is accomplished by setting the key fields to some value that is guaranteed never to occur as an actual key. With alphanumeric keys, for example, a key consisting of all blanks might represent an empty position.

insertion

To insert a record into the hash table, the hash function for the key is first calculated. If the corresponding location is empty, then the record can be inserted, else if the keys are equal, then insertion of the new record would not be allowed, and in the remaining case (a record with a different key is in the location), it becomes necessary to resolve the collision.

retrieval

To retrieve the record with a given key is entirely similar. First, the hash function for the key is computed. If the desired record is in the corresponding location, then the retrieval has succeeded; otherwise, while the location is nonempty and not all locations have been examined, follow the same steps used for collision resolution. If an empty position is found, or all locations have been considered, then no record with the given key is in the table, and the search is unsuccessful.

8.6.2 Choosing a Hash Function

method

The two principal criteria in selecting a hash function are that it should be easy and quick to compute and that it should achieve an even distribution of the keys that actually occur across the range of indices. If we know in advance exactly what keys will occur, then it is possible to construct hash functions that will be very efficient, but generally we do not know in advance what keys will occur. Therefore, the usual way is for the hash function to take the key, chop it up, mix the pieces together in various ways, and thereby obtain an index that (like the pseudorandom numbers generated by computer) will be uniformly distributed over the range of indices.

Note, however, that there is nothing random about a hash function. If the function is evaluated more than once on the same key, then it gives the same result every time, so the key can be retrieved without fail.

It is from this process that the word *hash* comes, since the process converts the key into something that bears little resemblance to the original. At the same time, it is hoped that any patterns or regularities that may occur in the keys will be destroyed, so that the results will be uniformly distributed.

Even though the term *hash* is very descriptive, in some books the more technical terms ***scatter-storage*** or ***key-transformation*** are used in its place.

1. Truncation

Ignore part of the key, and use the remaining part directly as the index (considering non-numeric fields as their numerical codes). If the keys, for example, are eight-digit integers and the hash table has 1000 locations, then the first, second, and fifth digits from the right might make the hash function, so that 62538194 maps to 394. Truncation is a very fast method, but it often fails to distribute the keys evenly through the table.

2. Folding

Partition the key into several parts and combine the parts in a convenient way (often using addition or multiplication) to obtain the index. For example, an eight-digit integer can be divided into groups of three, three, and two digits, the groups added together, and truncated if necessary to be in the proper range of indices. Hence 62538194 maps to 625 + 381 + 94 = 1100, which is truncated to 100. Since all information in the key can affect the value of the function, folding often achieves a better spread of indices than does truncation by itself.

3. Modular Arithmetic

Convert the key to an integer (using the above devices as desired), divide by the size of the index range, and take the remainder as the result. This amounts to using the C modulus operator %. The spread achieved by taking a remainder depends very much on the modulus (in this case, the size of the hash array). If the modulus is a power of a small integer like 2 or 10, then many keys tend to map to the same index, while other indices remain unused. The best choice for modulus is often, but not always, a prime number, which usually has the effect of spreading the keys quite uniformly. (We shall see later that a prime modulus also improves an important method for collision resolution.) Hence, rather than choosing a hash table size of 1000, it is often better to choose either 997 or 1009; $2^{10} = 1024$ would usually be a poor choice. Taking the remainder is usually the best way to conclude calculating the hash function, since it can achieve a good spread at the same time that it ensures that the result is in the proper range.

prime modulus

4. C Example

As a simple example, let us write a hash function in C for transforming a key consisting of alphanumeric characters into an integer in the range

```
0..HASHSIZE-1.
```

That is, we shall begin with the type

```
typedef char *Key;
```

We can then write a simple hash function as follows:

```
/* Hash: determine the hash value of key s.
   Pre:  s is a valid key type.
   Post: s has been hashed, returning a value between 0 and HASHSIZE - 1 */
int Hash(Key s)
{
   unsigned h = 0;
   while (*s)
      h += *s++;
   return h % HASHSIZE;
}
```

We have simply added the integer codes corresponding to each of the characters in the string. There is no reason to believe that this method will be better (or worse), however, than any number of others. We could, for example, subtract some of the codes, multiply them in pairs, or ignore every other character. Sometimes an application will suggest that one hash function is better than another; sometimes it requires experimentation to settle on a good one.

8.6.3 Collision Resolution with Open Addressing

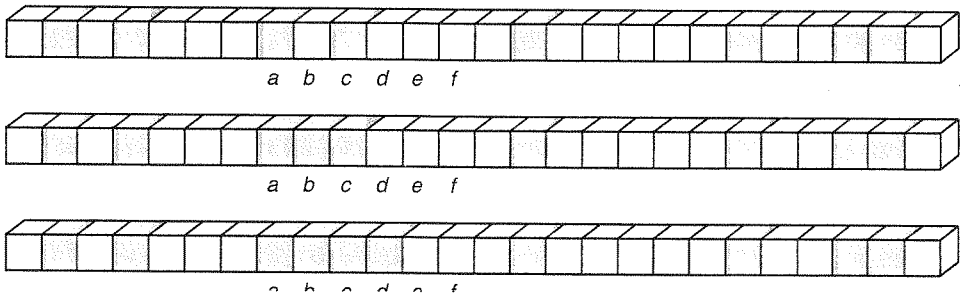
1. Linear Probing

The simplest method to resolve a collision is to start with the hash address (the location where the collision occurred) and do a sequential search through the table for the desired key or an empty location. Hence this method searches in a straight line, and it is therefore called ***linear probing***. The table should be considered circular, so that when the last location is reached, the search proceeds to the first location of the table.

2. Clustering

The major drawback of linear probing is that, as the table becomes about half full, there is a tendency toward ***clustering***; that is, records start to appear in long strings of adjacent positions with gaps between the strings. Thus the sequential searches needed to find an empty position become longer and longer. Consider the example in Figure 8.13, where the occupied positions are shown in color. Suppose that there

example of clustering



are n locations in the array and that the hash function chooses any of them with equal probability $1/n$. Begin with a fairly uniform spread, as shown in the top diagram. If a new insertion hashes to location *b*, then it will go there, but if it hashes to location *a* (which is full), then it will also go into *b*. Thus the probability that *b* will be filled has doubled to $2/n$. At the next stage, an attempted insertion into any of locations *a*, *b*, *c*, or *d* will end up in *d*, so the probability of filling *d* is $4/n$. After this, *e* has probability $5/n$ of being filled, and so as additional insertions are made the most likely effect is to make the string of full positions beginning at location *a* longer and longer. Hence the performance of the hash table starts to degenerate toward that of sequential search.

instability

The problem of clustering is essentially one of instability; if a few keys happen randomly to be near each other, then it becomes more and more likely that other keys will join them, and the distribution will become progressively more unbalanced.

3. Increment Functions

If we are to avoid the problem of clustering, then we must use some more sophisticated way to select the sequence of locations to check when a collision occurs. There are many ways to do so. One, called ***rehashing***, uses a second hash function to obtain the second position to consider. If this position is filled, then some other method is needed to get the third position, and so on. But if we have a fairly good spread from the first hash function, then little is to be gained by an independent second hash function. We will do just as well to find a more sophisticated way of determining the distance to move from the first hash position and apply this method, whatever the first hash location is. Hence we wish to design an increment function that can depend on the key or on the number of probes already made and that will avoid clustering.

rehashing

4. Quadratic Probing

If there is a collision at hash address h, this method probes the table at locations $h+1$, $h+4$, $h+9$, ..., that is, at locations $h + i^2$ (% HASHSIZE) for $i = 1, 2, \ldots$. That is, the increment function is i^2.

Quadratic probing substantially reduces clustering, but it is not obvious that it will probe all locations in the table, and in fact it does not. For some values of HASHSIZE, for example powers of 2, the function will probe relatively few positions in the array. When HASHSIZE is a prime number, however, quadratic probing reaches half the locations in the array.

proof

To prove this observation, suppose that HASHSIZE is a prime number. Also suppose that we reach the same location at probe i and at some later probe that we can take as $i + j$ for some integer $j > 0$. Suppose that j is the smallest such integer. Then the values calculated by the function at i and at $i + j$ differ by a multiple of HASHSIZE. In other words,

$$h + i^2 \equiv h + (i + j)^2 \text{ (\% HASHSIZE)}.$$

When this expression is simplified, we obtain

$$j^2 + 2ij = j(j + 2i) \equiv 0 \text{ (\% HASHSIZE)}.$$

This last expression means that HASHSIZE divides (with no remainder) the product $j(j+2i)$. The only way that a prime number can divide a product is to divide one of its factors. Hence HASHSIZE either divides j or it divides $j+2i$. If the first case occurred, then we would have made HASHSIZE probes before duplicating probe i. (Recall that j is the smallest positive integer such that probe $i+j$ duplicates probe i.) The second case, however, will occur sooner, when j = HASHSIZE $- 2i$, or, if this expression is negative, at this expression increased by HASHSIZE. Hence the total number of distinct positions that will be probed is exactly

(HASHSIZE + 1)/2.

It is customary to regard the table as full when this number of positions has been probed, and the results are quite satisfactory.

calculation

Note that quadratic probing can be accomplished without doing multiplications: After the first probe at position x, the increment is set to 1. At each successive probe, the increment is increased by 2 after it has been added to the previous location. Since

$$1 + 3 + 5 + \cdots + (2i - 1) = i^2$$

for all $i \geq 1$ (you can prove this fact by mathematical induction), probe i will look in position $x + 1 + 3 + \cdots + (2i - 1) = x + i^2$, as desired.

5. Key-Dependent Increments

Rather than having the increment depend on the number of probes already made, we can let it be some simple function of the key itself. For example, we could truncate the key to a single character and use its code as the increment. In C, we might write

```
increment = *key;
```

A good approach, when the remainder after division is taken as the hash function, is to let the increment depend on the quotient of the same division. An optimizing compiler should specify the division only once, so the calculation will be fast, and the results generally satisfactory.

In this method, the increment, once determined, remains constant. If HASHSIZE is a prime, it follows that the probes will step through all the entries of the array before any repetitions. Hence overflow will not be indicated until the array is completely full.

6. Random Probing

A final method is to use a pseudorandom number generator to obtain the increment. The generator used should be one that always generates the same sequence provided it starts with the same seed. (See Chapter 4.) The seed, then, can be spec-

7. C Algorithms

To conclude the discussion of open addressing, we continue to study the C example already introduced, which used alphanumeric keys of the type

```
typedef char *Key;
```

We set up the hash table with the declarations

```
/* declarations for a hash table with open addressing */
#define HASHSIZE 997
typedef char *Key;
typedef struct item {
   Key key;
} Entry;
typedef Entry HashTable[HASHSIZE];
```

initialization

The hash table must be created by setting the key field of each item to NULL. This is the task of the function CreateTable, whose specifications are:

```
void CreateTable(HashTable H);
```
precondition: None.

postcondition: The hash table H has been created and initialized to be empty.

There should also be a function ClearTable that returns a table that already has been created to an empty state. Its specifications follow, but its code (for the case of hash tables using open addressing) will be identical to that of CreateTable.

```
void ClearTable(HashTable H);
```
precondition: The hash table H has been created.

postcondition: The hash table H has been cleared and is empty.

Although we have started to specify hash-table operations, we shall not continue to develop a complete and general set of functions. Since the choice of a good hash function depends strongly on the kind of keys used, hash-table operations are usually too dependent on the particular application to be assembled into a set of functions.

To show how the code for further routines will be written, we shall continue to follow the example of the hash function already written in Section 8.6.2, page 357, and we shall use quadratic probing for collision resolution. We have shown that the maximum number of probes that can be made this way is (HASHSIZE + 1)/2, and we keep a counter pc to check this upper bound.

With these conventions, let us write a function to insert a new entry newentry into the hash table H.

```
/* Insert: insert an item using open addressing and linear probing.
   Pre:  The hash table H has been created and is not full. H has no current entry with
         key equal to that of newitem.
   Post: The item newitem has been inserted into H.
   Uses: Hash. */
void Insert(HashTable H, Entry newitem)
{
    int pc = 0;                    /* probe count to be sure that table is not full      */
    int probe;                     /* position currently probed in H                     */
    int increment = 1;             /* increment used for quadratic probing               */
    probe = Hash(newitem.key);
    while (H[probe].key != NULL &&  /* Is the location empty?                            */
      strcmp(newitem.key, H[probe].key) &&  /* Duplicate key present?                     */
      pc <= HASHSIZE/2) {          /* Has overflow occurred?                             */
        pc++;
        probe = (probe + increment) % HASHSIZE;
        increment += 2;            /* Prepare increment for next iteration.              */
    }
    if (H[probe].key == NULL)
        H[probe] = newitem;        /* Insert the new entry.                              */
    else if (strcmp(newitem.key, H[probe].key) == 0)
        Error("The same key cannot appear twice in the hash table.");
    else
        Error("Hash table is full; insertion cannot be made.");
}
```

A function to retrieve the record (if any) with a given key will have a similar form and is left as an exercise. The retrieval function should return the full entry associated with a target key. Its specifications are:

```
int RetrieveTable(HashTable H, Key target);
```

precondition: The hash table H has been created.

postcondition: If an entry in H has key equal to target, then the function returns the index for the entry. Otherwise, the function returns −1.

8. Deletions

Up to now, we have said nothing about deleting entries from a hash table. At first glance, it may appear to be an easy task, requiring only marking the deleted location with the special key indicating that it is empty. This method will not work. The reason is that an empty location is used as the signal to stop the search for a target key. Suppose that, before the deletion, there had been a collision or two and

position will stop the search, and it is impossible to find the entry, even though it is still in the table.

special key

One method to remedy this difficulty is to invent another special key, to be placed in any deleted position. This special key would indicate that this position is free to receive an insertion when desired but that it should not be used to terminate the search for some other entry in the table. Using this second special key will, however, make the algorithms somewhat more complicated and a bit slower. With the methods we have so far studied for hash tables, deletions are indeed awkward and should be avoided as much as possible.

8.6.4 Collision Resolution by Chaining

Up to now we have implicitly assumed that we are using only contiguous storage while working with hash tables. Contiguous storage for the hash table itself is, in fact, the natural choice, since we wish to be able to refer quickly to random positions in the table, and linked storage is not suited to random access. There is, however, no reason why linked storage should not be used for the records themselves. We can take the hash table itself as an array of pointers to the records, that is, as an array of linked lists. An example appears in Figure 8.14.

linked storage

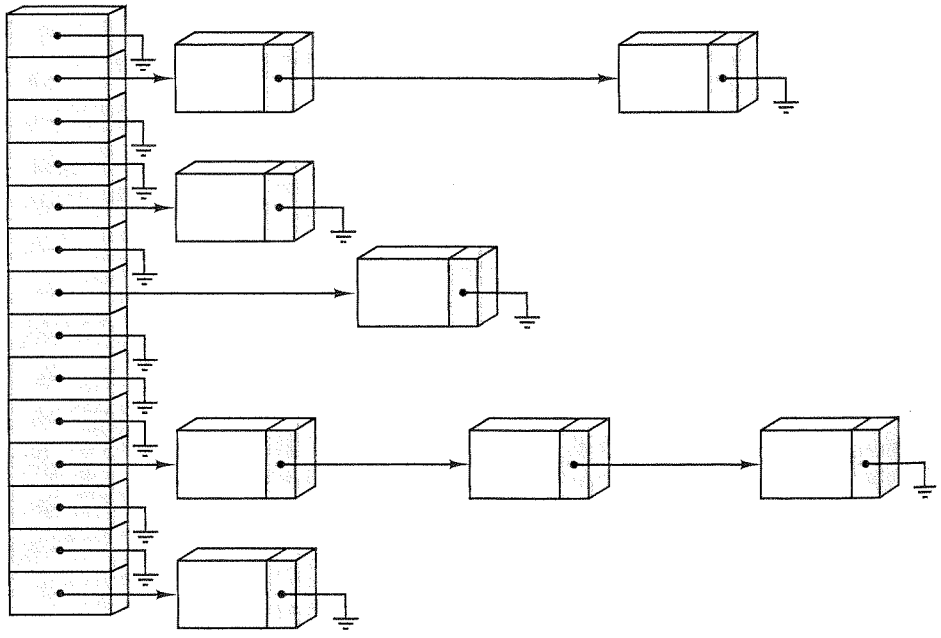


Figure 8.14. A chained hash table

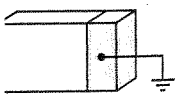
chaining

It is traditional to refer to the linked lists from the hash table as ***chains*** and call this method collision resolution by ***chaining***.

even though it

ecial key, to be
this position is
ed to terminate
pecial key will,
it slower. With
deed awkward

iguous storage
able itself is, in
ndom positions
ere is, however,
hemselves. We
s, that is, as an



space saving

collision resolution

overflow

deletion

use of space

small records

1. Advantages of Chaining

There are several advantages to this point of view. The first, and the most important when the records themselves are quite large, is that considerable space may be saved. Since the hash table is a contiguous array, enough space must be set aside at compilation time to avoid overflow. If the records themselves are in the hash table, then if there are many empty positions (as is desirable to help avoid the cost of collisions), these will consume considerable space that might be needed elsewhere. If, on the other hand, the hash table contains only pointers to the records, pointers that require only one word each, then the size of the hash table may be reduced by a large factor (essentially by a factor equal to the size of the records), and will become small relative to the space available for the records, or for other uses.

The second major advantage of keeping only pointers in the hash table is that it allows simple and efficient collision handling. We need only add a link field to each record, and organize all the records with a single hash address as a linked list. With a good hash function, few keys will give the same hash address, so the linked lists will be short and can be searched quickly. Clustering is no problem at all, because keys with distinct hash addresses always go to distinct lists.

A third advantage is that it is no longer necessary that the size of the hash table exceed the number of records. If there are more records than entries in the table, it means only that some of the linked lists are now sure to contain more than one record. Even if there are several times more records than the size of the table, the average length of the linked lists will remain small and sequential search on the appropriate list will remain efficient.

Finally, deletion becomes a quick and easy task in a chained hash table. Deletion proceeds in exactly the same way as deletion from a simple linked list.

2. Disadvantage of Chaining

These advantages of chained hash tables are indeed powerful. Lest you believe that chaining is always superior to open addressing, however, let us point out one important disadvantage: All the links require space. If the records are large, then this space is negligible in comparison with that needed for the records themselves; but if the records are small, then it is not.

Suppose, for example, that the links take one word each and that the entries themselves take only one word (which is the key alone). Such applications are quite common, where we use the hash table only to answer some yes–no question about the key. Suppose that we use chaining and make the hash table itself quite small, with the same number n of entries as the number of entries. Then we shall use $3n$ words of storage altogether: n for the hash table, n for the keys, and n for the links to find the next node (if any) on each chain. Since the hash table will be nearly full, there will be many collisions, and some of the chains will have several entries. Hence searching will be a bit slow. Suppose, on the other hand, that we use open addressing. The same $3n$ words of storage put entirely into the hash table

3. C Algorithms

A chained hash table in C takes the simple declaration

```
typedef List HashTable[HASHSIZE];
```

where List refers to the linked implementation of lists studied in Chapter 5.

initialization

The code needed to create the hash table is

```
for(i = 0; i < HASHSIZE; i++)
   CreateList(H[i]);
```

To clear a chained hash table that has previously been created is a different task, in contrast to open addressing, where it was the same as creating the table. To clear the table, we must clear the linked list in each of the table positions. This task can be done by using the linked-list function ClearList.

We can even use the list processing functions to access the hash table. The hash function itself is no different from that used with open addressing; for data retrieval, we can simply use a linked version of SequentialSearch. The essence of RetrieveTable is

```
SequentialSearch(H[Hash(target)], target);
```

The details of converting this into a full function are left as an exercise.

Similarly, the essence of insertion is the one line

```
InsertList(0, newentry, H[newentry.key]);
```

Here we have chosen to insert the new entry as the first node of its list, since that is the easiest. As you can see, both insertion and retrieval are simpler than the versions for open addressing, since collision resolution is not a problem and we can make use of the previous work done for linked lists.

deletion

Deletion from a chained hash table is also much simpler than it is from a table with open addressing. To delete the entry with a given key, we need only use sequential search to find the entry where it is located within its chain in the hash table, and then we delete this entry from its linked list. The specifications for this function are:

```
Entry *DeleteTable(HashTable H, Key target);
```

precondition: The chained hash table H has been created and contains an entry with key equal to target.

postcondition: The entry with key equal to target has been deleted from H and its pointer has been returned.

Writing the corresponding function is left as an exercise.