Exhibit 8

Second Edition

Robert L. Kruse

DATA STRUCTURES AND PROGRAM DESIGN

In this new edition, Robert L. Kruse explores top-down, structured problem solving, the process of data abstraction and structuring, and the comparative study of algorithms as fundamental tools of program design. The author includes all topics from ACM C2 with additional content in data structures, algorithm development, analysis, and software engineering.

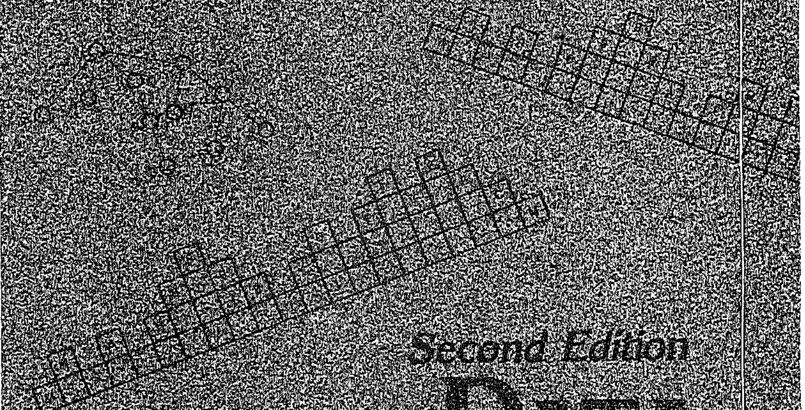
Among its NEW features, the book

- provides greatly expanded emphasis on the process of data abstraction, information hiding, and abstract data types;
- includes an exposition of the principles of software engineering;
- uses preconditions and postconditions, when appropriate, both to verify the correctness of algorithms and to formulate algorithms;
- adds new sections on graphs and graph algorithms, on simulation as a use of queues, and extends several other sections;
- uses a two-color format to highlight and distinguish different elements in the text, programs, and diagrams;
- concludes each of the first ten chapters with a section of Pointers and Pitfalls giving helpful hints for problems of algorithm design and Review Questions;
- uses notes in the margins to highlight key words and point out all important ideas.

PRENTICE-HALL, INC., Englewood Cliffs, N.J. 07632

Kruse DATA STRUCTURES AND PROGRAM DESIGN Second Edition PRENTICE HALL

Robert L. Kruse



Second Edition

DATA STRUCTURES & PROGRAM DESIGN

sample hash function

```
function Hash(x: keytype): integer;
var
  i: 1..8;
  h: integer;
begin
  h := 0;
  for i := 1 to 8 do
    h := h + ord(x[i]);
  Hash := h mod hashsize
end;
```

We have simply added the integer codes corresponding to each of the eight characters. 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.

6.5.3 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 for the desired key or an empty location. Hence this method searches in a straight line, and it is therefore called ***linear probing***. The array should be considered circular, so that when the last location is reached, the search proceeds to the first location of the array.

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. For consider the example

example of clustering

in Figure 6.11, where the occupied positions are shown in color. Suppose that there 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, and hence the performance of the hash table starts to degenerate toward that of sequential search.

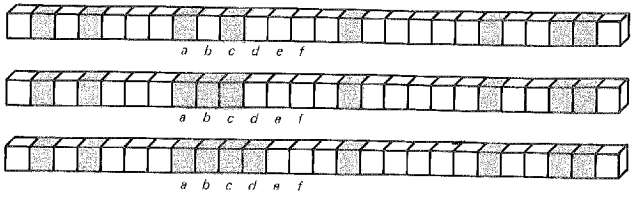


Figure 6.11. Clustering in a hash table

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 $h + i^2$ (mod hashsize) for $i = 1, 2, \ldots$. That is, the increment function is i^2.

This method substantially reduces clustering, but it is not obvious that it will probe all locations in the table, and in fact it does not. If hashsize is a power of 2, then relatively few positions are probed. Suppose that hashsize is a prime. If we reach the same location at probe i and at probe j, then

$$h + i^2 \equiv h + j^2 \text{ (mod hashsize)}$$

so that

$$(i - j)(i + j) \equiv 0 \text{ (mod hashsize)}.$$

Since hashsize is a prime, it must divide one factor. It divides $i - j$ only when j differs from i by a multiple of hashsize, so at least hashsize probes have been made. Hashsize divides $i + j$, however, when $j =$ hashsize $- i$, so the total number of distinct positions that will be probed is exactly

number of distinct probes

$$(\text{hashsize} + 1) \textbf{ div } 2.$$

8. Deletions

Up to now we have said nothing about deleting items 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 that some item whose hash address is the now-deleted position is actually stored elsewhere in the table. If we now try to retrieve that item, then the now-empty position will stop the search, and it is impossible to find the item, 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 item 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.

6.5.4 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,

linked storage

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 list headers. An example appears in Figure 6.12.

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

1. Advantages of Linked Storage

space saving

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.

collision resolution

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

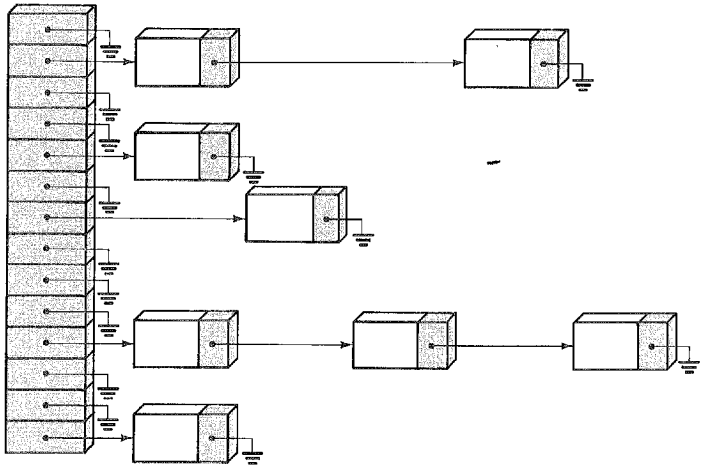


Figure 6.12. A chained hash table

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.

overflow

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.

deletion

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 Linked Storage

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

use of space

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.

small records

Suppose, for example, that the links take one word each and that the items 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 items. 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 items.

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 will mean that it will be only one third full, and therefore there will be relatively few collisions and the search for any given item will be faster.

3. Pascal Algorithms

A chained hash table in Pascal takes declarations like

declarations

```
type
  pointer   = ↑node;
  list      = record head: pointer end;
  hashtable = array [0..hashmax] of list;
```

The record type called node consists of an item, called info, and an additional field, called next, that points to the next node on a linked list.

The code needed to initialize the hash table is

initialization

```
for i := 0 to hashmax do H[i].head := nil;
```

We can even use previously written procedures 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 the procedure SequentialSearch (linked version) from Section 5.2, as follows:

retrieval

```
procedure Retrieve(var H: hashtable; target: keytype;
                   var found: Boolean; var location: pointer);
{finds the node with key target in the hash table H, and returns with location
  pointing to that node, provided that found becomes true}
begin
  SequentialSearch(H[Hash(target)], target, found, location)
end;
```

Our procedure for inserting a new entry will assume that the key does not appear already; otherwise, only the most recent insertion with a given key will be retrievable.

insertion

```
procedure Insert(var H: hashtable; p: pointer);
{inserts node p↑ into the chained hash table H, assuming no other node with
  key p↑.info.key is in the table}
var
  i: integer;                                  {used for index in hash table}
begin
  i := Hash(p↑.info.key);                {Find the index of the linked list for p↑.}
  p↑.next := H[i].head;                        {Insert p↑ at the head of the list.}
  H[i].head := p                       {Set the head of the list to the new item.}
end;
```

As you can see, both of these procedures are significantly simpler than are the versions for open addressing, since collision resolution is not a problem.

Exercises 6.5

E1. Write a Pascal procedure to insert an item into a hash table with open addressing and linear probing.

E2. Write a Pascal procedure to retrieve an item from a hash table with open addressing and **(a)** linear probing; **(b)** quadratic probing.

E3. Devise a simple, easy-to-calculate hash function for mapping three-letter words to integers between 0 and $n - 1$, inclusive. Find the values of your function on the words

PAL LAP PAM MAP PAT PET SET SAT TAT BAT

for $n = 11, 13, 17, 19$. Try for as few collisions as possible.

E4. Suppose that a hash table contains hashsize = 13 entries indexed from 0 through 12 and that the following keys are to be mapped into the table:

10 100 32 45 58 126 3 29 200 400 0.

(a) Determine the hash addresses and find how many collisions occur when these keys are reduced **mod** hashsize.

(b) Determine the hash addresses and find how many collisions occur when these keys are first folded by adding their digits together (in ordinary decimal representation) and then reducing **mod** hashsize.

perfect hash functions

(c) Find a hash function that will produce no collisions for these keys. (A hash function that has no collisions for a fixed set of keys is called ***perfect.***)

(d) Repeat the previous parts of this exercise for hashsize = 11. (A hash function that produces no collision for a fixed set of keys that completely fill the hash table is called ***minimal perfect.***)

E5. Another method for resolving collisions with open addressing is to keep a separate array called the ***overflow table,*** into which all items that collide with an occupied location are put. They can either be inserted with another hash function or simply inserted in order, with sequential search used for retrieval. Discuss the advantages and disadvantages of this method.

E6. Write an algorithm for deleting a node from a chained hash table.

E7. Write a deletion algorithm for a hash table with open addressing, using a second special key to indicate a deleted item (see part 8 of Section 6.5.3). Change the retrieval and insertion algorithms accordingly.

E8. With linear probing, it is possible to delete an item without using a second special key, as follows. Mark the deleted entry empty. Search until another empty position is found. If the search finds a key whose hash address is at or before the first empty position, then move it back there, make its previous position empty, and continue from the new empty position. Write an algorithm to implement this method. Do the retrieval and insertion algorithms need modification?