2

HASHING TECHNIQUES

Unit Structure:

2.1 INTRODUCTION

The ideal hash table data structure is an array of some fixed size, containing the keys. Whereas, a key is a string. We will refer to the table size as *H_SIZE*, with the understanding that this is part of a hash data structure and not merely some variable floating around globally. The common convention is to have the table run from 0 to *H_SIZE-1.*

Each key is mapped into some number in the range 0 to *H_SIZE - 1* and placed in the appropriate cell. The mapping is called a *hash function*, which ideally should be simple to compute. Since there are a finite number of cells and a virtually inexhaustible supply of keys, this is clearly impossible, and thus we seek a Hash function that distributes the keys evenly among the cells. Figure 2.1 is an example, here *Raj* hashes to 3, Madhurya hashes to 4, *Rahul* hashes to 6, and *Priya* hashes to 7.

0	
1	
2	
3	Raj
4	Madhurya
5	
6	Rahul
7	Priya
8	
9	

Figure 2.1 An ideal hash table

This is the basic idea of hashing. The only remaining problems deal with choosing a function, deciding what to do when two keys hash to the same value (this is known as a *collision*), and deciding on the table size.

2.2. HASH FUNCTION

If the input keys are integers, then simply returning *key mod H_SIZE* (Table size) is generally a reasonable strategy, unless *key* happens to have some undesirable properties. In this case, the choice of hash function needs to be carefully considered. For instance, if the table size is 10 and the keys all end in zero, then the standard hash function is obviously a bad choice. When the input keys are random integers, then this function is not only very simple to compute but also distributes the keys evenly.

One option is to add up the ASCII values of the characters in the string. In Figure 2.2 we declare the type *INDEX*, which is returned by the hash function. The routine in Figure 2.3 implements this strategy and uses the typical C method of stepping through a string.

The hash function depicted in Figure 2.3 is simple to implement and computes an answer quickly. However, if the table size is large, the function does not distribute the keys well. For instance, suppose that *H_SIZE* = 10,007 (10,007 is a prime number). Suppose all the keys are eight or fewer characters long. Since a *char* has an integer value that is always at most 127, the hash function can only assume values between 0 and 1016, which is 127 * 8. This is clearly not an equitable distribution!

```
typedef unsigned int INDEX;
```

Figure 2.2 Type returned by hash function

```
INDEX
hash( char *key, unsigned int H_SIZE )
{
unsigned int hash_val = 0;
/*1*/       while( *key != '\0' )
/*2*/           hash_val += *key++;
/*3*/       return( hash_val % H_SIZE );
}
```

Figure 2.3 A simple hash function

Another hash function is shown in Figure 2.4. This hash function assumes *key* has at least two characters plus the NULL terminator. 27 represents the number of letters in the English alphabet, plus the blank, and 729 is 27^2. This function only examines the first three characters, but if these are random, and the table size is 10,007, as before, then we would expect a reasonably equitable distribution. Although there are $26^3 = 17,576$ possible combinations of three characters (ignoring blanks), a check of a reasonably large on-line dictionary reveals that the number of different combinations is actually only 2,851. Even if none of *these* combinations collide, only 28 percent of the table can actually be hashed to. Thus this function, although easily computable, is also not appropriate if the hash table is reasonably large.

Figure 2.5 shows a third attempt at a hash function. This hash function involves all characters in the key and can generally be expected to distribute well (it computes $\sum_{i=0}^{KeySize - 1} key[key\_size - i - 1] 32^{i,}$ and brings the result into proper range). The code computes a polynomial function (of 32) by use of Horner's rule. For instance, another way of computing $h_k = k_1 + 27k_2 + 27^2k_3$ is by the formula $h_{k =} ((k_3) * 27 + k_2) * 27 + k_1$. Horner's rule extends this to an *n*th degree polynomial.

We have used 32 instead of 27, because multiplication by 32 is not really a multiplication, but amounts to bit-shifting by five. In line 2, the addition could be replaced with a bitwise exclusive or, for increased speed.


```
INDEX
hash( char *key, unsigned int H_SIZE )
{
return ( ( key[0] + 27*key[1] + 729*key[2] ) % H_SIZE );
}
```

Figure 2.4 Another possible hash function -- not too good

```
INDEX
hash( char *key, unsigned int H_SIZE )
{
unsigned int hash_val = O;
/*1*/       while( *key != '\0' )
/*2*/           hash_val = ( hash_val << 5 ) + *key++;
/*3*/       return( hash_val % H_SIZE );
}
```

Figure 2.5 A good hash function

The hash function described in Figure 2.5 is not necessarily the best with respect to table distribution, but does have the merit of extreme simplicity (and speed if overflows are allowed). If the keys are very long, the hash function will take too long to compute. Furthermore, the early characters will wind up being left-shifted out of the eventual answer. A common practice in this case is not to use all the characters. The length and properties of the keys would then influence the choice. For instance, the keys could be a complete street address. The hash function might include a couple of characters from the street address and perhaps a couple of characters from the city name and ZIP code. Some programmers implement their hash function by using only the characters in the odd spaces, with the idea that the time saved computing the hash function will make up for a slightly less evenly distributed function.

The main programming detail left is collision resolution. If, when inserting an element, it hashes to the same value as an already inserted element, then we have a *collision* and need to resolve it. There are several methods for dealing with this. We will discuss two of the simplest: open hashing and closed hashing.*

*These are also commonly known as separate chaining and open addressing, respectively.

2.3. OPEN HASHING (SEPARATE CHAINING)

The first strategy, commonly known as either *open hashing,* or *separate chaining*, is to keep a list of all elements that hash to the same value. We assume that the keys are the first 10 perfect squares and that the hashing function is simply *hash*(*x*) = *x mod* 10. (The table size is not prime, but is used here for simplicity.) Figure 2.6 should make this clear.

To perform a *find*, we use the hash function to determine which list to traverse. We then traverse this list in the normal manner, returning the position where the item is found. To perform an *insert*, we traverse down the appropriate list to check whether the element is already in place (if duplicates are expected, an extra field is usually kept, and this field would be incremented in the event of a match). If the element turns out to be new, it is inserted either at the front of the list or at the end of the list, whichever is easiest. This is an issue most easily addressed while the code is being written. Sometimes new elements are inserted at the front of the list, since it is convenient and also because frequently it happens that recently inserted elements are the most likely to be accessed in the near future.

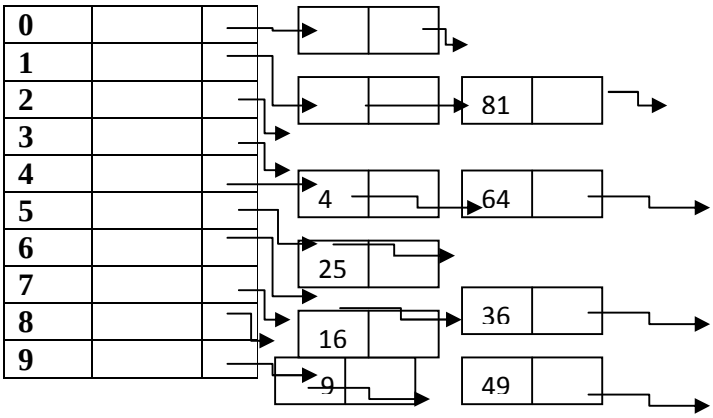

Figure 2.6 An open hash table

The type declarations required to implement open hashing are in Figure 2.7. The hash table structure contains the actual size and an array of linked lists, which are dynamically allocated when the table is initialized. The *HASH_TABLE* type is just a pointer to this structure.

```
typedef struct list_node *node_ptr;
struct list_node
{
element_type element;
node_ptr next;
};
typedef node_ptr LIST;
typedef node_ptr position;
/* LIST *the_list will be an array of lists, allocated later */
/* The lists will use headers, allocated later */
struct hash_tbl
{
unsigned int table_size;
LIST *the_lists;
};
typedef struct hash_tbl *HASH_TABLE;
```

Figure 2.7 Type declaration for open hash table

Notice that the the_lists field is actually a pointer to a pointer to a list_node structure. If typedefs and abstraction are not used, this can be quite confusing.

```
HASH_TABLE
initialize_table( unsigned int table_size )
{
HASH_TABLE H;
int i;
/*1*/       if( table size < MIN_TABLE_SIZE )
{
/*2*/           error("Table size too small");
/*3*/           return NULL;
}
/* Allocate table */
/*4*/       H = (HASH_TABLE) malloc ( sizeof (struct hash_tbl) );
/*5*/       if( H == NULL )
/*6*/           fatal_error("Out of space!!!");
/*7*/       H->table_size = next_prime( table_size );
/* Allocate list pointers */
/*8*/           H->the_lists = (position *) malloc ( sizeof (LIST) * H-
>table_size );
/*9*/       if( H->the_lists == NULL )
/*10*/           fatal_error("Out of space!!!");
/* Allocate list headers */
/*11*/      for(i=0; i<H->table_size; i++ )
{
/*12*/          H->the_lists[i] = (LIST) malloc ( sizeof (struct list_node)
);
/*13*/          if( H->the_lists[i] == NULL )
/*14*/              fatal_error("Out of space!!!");
else
/*15*/              H->the_lists[i]->next = NULL;
}
/*16*/      return H;
}
```

Figure 2.8 Initialization routine for open hash table

Figure 2.8 shows the initialization function, which uses the same ideas that were seen in the array implementation of stacks. Lines 4

through 6 allocate a hash table structure. If space is available, then H will point to a structure containing an integer and a pointer to a list. Line 7 sets the table size to a prime number, and lines 8 through 10 attempts to allocate an array of lists. Since a LIST is defined to be a pointer, the result is an array of pointers.

If our LIST implementation was not using headers, we could stop here. Since our implementation uses headers, we must allocate one header per list and set its next field to NULL. This is done in lines 11 through 15. Also lines 12 through 15 could be replaced with the statement

H->the_lists[i] = make_null();

Since we have not used this option, because in this instance it is preferable to make the code as self-contained as possible, it is certainly worth considering. An inefficiency of our code is that the malloc on line 12 is performed H->table_size times. This can be avoided by replacing line 12 with one call to malloc before the loop occurs:

H->the lists = (LIST*) malloc

(H->table_size * sizeof (struct list_node));

Line 16 returns H.

The call find(key, H) will return a pointer to the cell containing key. The code to implement this is shown in Figure 2.9.

Next comes the insertion routine. If the item to be inserted is already present, then we do nothing; otherwise we place it at the front of the list (see Fig. 2.10).*

```
position

find( element_type key, HASH_TABLE H )

{

position p;

LIST L;

/*1*/       L = H->the_lists[ hash( key, H->table_size) ];

/*2*/       p = L->next;

/*3*/       while( (p != NULL) && (p->element != key) )
```

```
/* Probably need strcmp!! */
/*4*/           p = p->next;
/*5*/       return p;
}
```

Figure 2.9 Find routine for open hash table

```
void insert( element_type key, HASH_TABLE H )
{
position pos, new_cell;
LIST L;
/*1*/     pos = find( key, H );
/*2*/     if( pos == NULL )
{
/*3*/         new_cell = (position) malloc(sizeof(struct list_node));
/*4*/         if( new_cell == NULL )
/*5*/             fatal_error("Out of space!!!");
else
{
/*6*/             L = H->the_lists[ hash( key, H->table size ) ];
/*7*/             new_cell->next = L->next;
/*8*/             new_cell->element = key; /* Probably need strcpy!! */
/*9*/             L->next = new_cell;
}
}
}
```

Figure 2.10 Insert routine for open hash table

The element can be placed anywhere in the list; this is most convenient in our case. The insertion routine coded in Figure 2.10 is somewhat poorly coded, because it computes the hash function twice. Redundant calculations are always bad, so this code should be rewritten if it turns out that the hash routines account for a significant portion of a program's running time.

The deletion routine is a straightforward implementation of deletion in a linked list, so we will not bother with it here. Any scheme could be used besides linked lists to resolve the collisions- a binary search tree or even another hash table would work.

We define the load factor of a hash table to be the ratio of the number of elements in the hash table to the table size. The effort required to perform a search is the constant time required to evaluate the hash function plus the time to traverse the list.

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.11 Closed hash table with linear probing, after each insertion

In an unsuccessful search, the number of links to traverse is (excluding the final NULL link) on average. A successful search requires that about 1 + (1/2) links be traversed, since there is a guarantee that one link must be traversed (since the search is successful), and we also expect to go halfway down a list to find our match. This analysis shows that the table size is not really important, but the load factor is. The general rule for open hashing is to make the table size about as large as the number of elements expected.

2.4. CLOSED HASHING (OPEN ADDRESSING)

Open hashing has the limitation of requiring pointers. This will slow down the algorithm a bit because of the time required to allocate new cells, and also essentially requires the implementation of a second data structure. Closed hashing, also known as open addressing, is an alternative to resolving collisions with linked lists. In a closed hashing system, if a collision occurs, alternate cells are

tried until an empty cell is found. More formally, cells h0(x), h1(x), h2(x), . . . are tried in succession, where

hi(x) = (hash(x) + F(i))mod H_SIZE, with F(0) = 0.

The function, F, is the collision resolution strategy. Because all the data goes inside the table, a bigger table is needed for closed hashing than for open hashing. Generally, the load factor should be below $\lambda = 0.5$ for closed hashing.

The following are the three common collision resolution strategies.

Linear Probing

Quadratic Probing

Double Hashing

2.4.1 Linear Probing

In linear probing, F is a linear function of i, typically F(i) = i. This amounts to trying cells sequentially (with wraparound) in search of an empty cell. Figure 2.11 shows the result of inserting keys {89, 18, 49, 58, 69} into a hash table using the same hash function as before and the collision resolution strategy, F(i) = i.

The first collision occurs when 49 is inserted; it is put in spot 0 which is open. key 58 collides with 18, 89, and then 49 before an empty cell is found three away. The collision for 69 is handled in a similar manner. As long as the table is big enough, a free cell can always be found, but the time to do so can get quite large. Worse, even if the table is relatively empty, blocks of occupied cells start forming. This effect is known as **Primary Clustering.**

We will assume a very large table and that each probe is independent of the previous probes. These assumptions are satisfied by a random collision resolution strategy and are reasonable unless λ is very close to 1. First, we derive the expected number of probes in an unsuccessful search. This is just the expected number of probes until we find an empty cell. Since the fraction of empty cells is $1 - \lambda$, the number of cells we expect to probe is $1/(1 - \lambda)$. The number of probes for a successful search is equal to the number of probes required when the particular element was inserted. When an element is inserted, it is done as a result of an unsuccessful search. Thus we can use the cost of an

unsuccessful search to compute the average cost of a successful search.

The caveat is that ƛ changes from 0 to its current value, so that earlier insertions are cheaper and should bring the average down. For instance, in the table above, ƛ= 0.5, but the cost of accessing 18 is determined when 18 is inserted. At that point, ƛ = 0.2. Since 18, was inserted into a relatively empty table, accessing it should be easier than accessing a recently inserted element such as 69. We can estimate the average by using an integral to calculate the mean value of the insertion time, obtaining

$$I(ƛ) = 1/ ƛ \int_{0}^{ƛ} 1/ 1 – x \, .dx = 1 / ƛ \ln 1 / 1 - ƛ$$

These formulas are clearly better than the corresponding formulas for linear probing. Clustering is not only a theoretical problem but actually occurs in real implementations. Figure 2.12 compares the performance of linear probing (dashed curves) with what would be expected from more random collision resolution. Successful searches are indicated by an S, and unsuccessful searches and insertions are marked with U and I, respectively.

$$h(x) + i^2 = h(x) + j^2 \quad (mod\ H\_SIZE)$$

$$i^2 = j^2 \quad (mod\ H\_SIZE)$$

$$i^2 - j^2 = 0 \quad (mod\ H\_SIZE)$$

$$(i - j)(i + j) = 0 \quad (mod\ H\_SIZE)$$

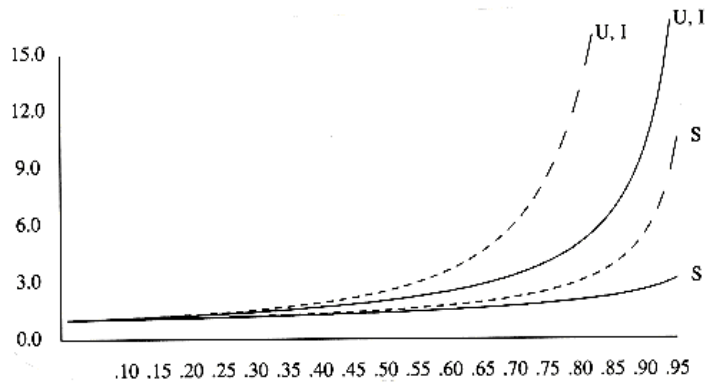

Figure 2.12 Number of probes plotted against load factor for linear probing (dashed) and random strategy. S is successful search,U is unsuccessful search, I is insertion

If ƛ = 0.75, then the formula above indicates that 8.5 probes are expected for an insertion in linear probing. If ƛ = 0.9, then 50 probes are expected, which is unreasonable. This compares with 4 and 10 probes for the respective load factors if clustering were not a problem. We see from these formulas that linear probing can be a bad idea if the table is expected to be more than half full. If ƛ = 0.5, however, only 2.5 probes are required on average for insertion and only 1.5 probes are required, on average, for a successful search.

2.4.2. Quadratic Probing

Quadratic probing is a collision resolution method that eliminates the primary clustering problem of linear probing. Quadratic probing is what you would expect-the collision function is quadratic. The popular choice is F(i) = i2. Figure 2.13 shows the resulting closed table with this collision function on the same input used in the linear probing example.

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 2.13 Closed hash table with quadratic probing, after each insertion

When 49 collides with 89, the next position attempted is one cell away. This cell is empty, so 49 is placed there. Next 58 collides at position 8. Then the cell one away is tried but another collision occurs. A vacant cell is found at the next cell tried, which is 22 = 4 away. 58 is thus placed in cell 2. The same thing happens for 69.

For quadratic probing, the situation is even more drastic: There is no guarantee of finding an empty cell once the table gets

more than half full, or even before the table gets half full if the table size is not prime. This is because at most half of the table can be used as alternate locations to resolve collisions.

Indeed, we prove now that if the table is half empty and the table size is prime, then we are always guaranteed to be able to insert a new element.

THEOREM 2.1.

If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.

PROOF:

Let the table size, H-SIZE, be an (odd) prime greater than 3. We show that the first [H-SIZE/2] alternate locations are all distinct. Two of these locations are h(x) + i2(mod H-SIZE) and h(x) + j2(mod H-SIZE), where 0 < i, j <[H-SIZE/2]. Suppose, for the sake of contradiction, that these locations are the same, but i ≠ j. Then

h(x) + i2 = h(x) + j2 (mod H_SIZE)
i2 = j2 (mod H_SIZE)
i2 - j2 = 0 (mod H_SIZE)(i - j)(i + j) = 0 (mod H_SIZE)

Since H_SIZE is prime, it follows that either (i - j) or (i + j) is equal to 0 (mod H_SIZE). Since i and j are distinct, the first option is not possible.

Since 0 < i, j <[H_SIZE/2], the second option is also impossible. Thus, the first [H_SIZE/2] alternate locations are distinct. Since the element to be inserted can also be placed in the cell to which it hashes (if there are no collisions), any element has [H_SIZE/2] locations into which it can go. If at most [H_SIZE/2] positions are taken, then an empty spot can always be found.

If the table is even one more than half full, the insertion could fail (although this is extremely unlikely). Therefore, it is important to keep this in mind. It is also crucial that the table size be prime.* If the table size is not prime, the number of alternate locations can be severely reduced. As an example, if the table size were 16, then the only alternate locations would be at distances 1, 4, or 9 away.

*If the table size is a prime of the form 4k + 3, and the quadratic collision resolution strategy f(i) = + i2 is used, then the entire table can be probed. The cost is a slightly more complicated routine.

Standard deletion cannot be performed in a closed hash table, because the cell might have caused a collision to go past it. For instance, if we remove 89, then virtually all of the remaining finds will fail. Thus, closed hash tables require lazy deletion, although in this case there really is no laziness implied.

```
enum kind_of_entry { legitimate, empty, deleted };
struct hash_entry
{
element_type element;
enum kind_of_entry info;
};
typedef INDEX position;
typedef struct hash_entry cell;
/* the_cells is an array of hash_entry cells, allocated later */
struct hash_tbl
{
unsigned int table_size;
cell *the_cells;
};
typedef struct hash_tbl *HASH_TABLE;
```

Figure 2.14 Type declaration for closed hash tables

```
HASH_TABLE
initialize_table( unsigned int table_size )
{
HASH_TABLE H;
int i;
/*1*/       if( table_size < MIN_TABLE_SIZE )
{
/*2*/           error("Table size too small");
/*3*/           return NULL;
```

```
}
/* Allocate table */
/*4*/       H = (HASH_TABLE) malloc( sizeof ( struct hash_tbl ) );
/*5*/       if( H == NULL )
/*6*/           fatal_error("Out of space!!!");
/*7*/       H->table_size = next_prime( table_size );
/* Allocate cells */
/*8*/       H->the cells = (cell *) malloc
( sizeof ( cell ) * H->table_size );
/*9*/       if( H->the_cells == NULL )
/*10*/          fatal_error("Out of space!!!");
/*11*/      for(i=0; i<H->table_size; i++ )
/*12*/          H->the_cells[i].info = empty;
/*13*/      return H;
}
```

Figure 2.15 Routine to initialize closed hash table

As with open hashing, find(key, H) will return the position of key in the hash table. If key is not present, then find will return the last cell. This cell is where key would be inserted if needed. Further, because it is marked empty, it is easy to tell that the find failed. We assume for convenience that the hash table is at least twice as large as the number of elements in the table, so quadratic resolution will always work. Otherwise, we would need to test i before line 4. In the implementation in Figure 2.16, elements that are marked as deleted count as being in the table. Lines 4 through 6 represent the fast way of doing quadratic resolution. From the definition of the quadratic resolution function, f(i) = f(i - 1) + 2i -1, so the next cell to try can be determined with a multiplication by two (really a bit shift) and a decrement. If the new location is past the array, it can be put back in range by subtracting H_SIZE. This is faster than the obvious method, because it avoids the multiplication and division that seem to be required. The variable name i is not the best one to use; we only use it to be consistent with the text.

```
position

find( element_type key, HASH_TABLE H )
{
position i, current_pos;
```

```
/*1*/      i = 0;
/*2*/      current_pos = hash( key, H->table_size );
/* Probably need strcmp! */
/*3*/      while( (H->the_cells[current_pos].element != key ) &&
(H->the_cells[current_pos].info != empty ) )
{
/*4*/          current_pos += 2*(++i) - 1;
/*5*/          if( current_pos >= H->table_size )
/*6*/              current_pos -= H->table_size;
}
/*7*/      return current_pos;
}
```

Figure 2.16 Find routine for closed hashing with quadratic probing

The final routine is insertion. As with open hashing, we do nothing if key is already present. It is a simple modification to do something else. Otherwise, we place it at the spot suggested by the find routine. The code is shown in Figure 2.17.

2.4.3. Double Hashing

The last collision resolution method we will examine is double hashing. For double hashing, one popular choice is F(i) = i.h2(x). This formula says that we apply a second hash function to x and probe at a distance h2(x), 2h2(x), . . ., and so on. For instance, the obvious choice h2(x) = x mod 9 would not help if 99 were inserted into the input in the previous examples. Thus, the function must never evaluate to zero. It is also important to make sure all cells can be probed (this is not possible in the example below, because the table size is not prime). A function such as h2(x) = R - (x mod R), with R a prime smaller than H_SIZE, will work well. If we choose R = 7, then Figure 2.18 shows the results of inserting the same keys as before.

```
void insert( element_type key, HASH_TABLE H )
{
position pos;
pos = find( key, H );
if( H->the_cells[pos].info != legitimate )
```

```
{    /* ok to insert here */
H->the_cells[pos].info = legitimate;
H->the_cells[pos].element = key;
/* Probably need strcpy!! */
}
}
```

Figure 2.17 Insert routine for closed hash tables with quadratic probing

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 2.18 Closed hash table with double hashing, after each insertion

The first collision occurs when 49 is inserted. h2(49) = 7 - 0 = 7, so 49 is inserted in position 6. h2(58) = 7 - 2 = 5, so 58 is inserted at location 3. Finally, 69 collides and is inserted at a distance h2(69) = 7 - 6 = 1 away. If we tried to insert 60 in position 0, we would have a collision. Since h2(60) = 7 - 4 = 3, we would then try positions 3, 6, 9, and then 2 until an empty spot is found. It is generally possible to find some bad case, but there are not too many here.

As we have said before, the size of our sample hash table is not prime. We have done this for convenience in computing the hash function, but it is worth seeing why it is important to make sure the table size is prime when double hashing is used. If we attempt to insert 23 into the table, it would collide with 58. Since h2(23) = 7 -

2 = 5, and the table size is 10, we essentially have only one alternate location, and it is already taken. Thus, if the table size is not prime, it is possible to run out of alternate locations prematurely. However, if double hashing is correctly implemented, simulations imply that the expected number of probes is almost the same as for a random collision resolution strategy. This makes double hashing theoretically interesting. Quadratic probing, however, does not require the use of a second hash function and is thus likely to be simpler and faster in practice.

2.5. REHASHING

If the table gets too full, the running time for the operations will start taking too long and inserts might fail for closed hashing with quadratic resolution. This can happen if there are too many deletions intermixed with insertions. A solution for this is to build another table that is about twice as big and scan down the entire original hash table, computing the new hash value for each (non-deleted) element and inserting it in the new table.

For example, suppose the elements 13, 15, 24, and 6 are inserted into a closed hash table of size 7. The hash function is h(x) = x mod 7. Suppose linear probing is used to resolve collisions. The resulting hash table appears in Figure 2.19.

If 23 is inserted into the table, the resulting table in Figure 2.20 will be over 70 percent full. Because the table is so full, a new table is created. The size of this table is 17, because this is the first prime which is twice as large as the old table size. The new hash function is then h(x) = x mod 17. The old table is scanned, and elements 6, 15, 23, 24, and 13 are inserted into the new table. The resulting table appears in Figure 2.21. This entire operation is called rehashing. This is obviously a very expensive operation -- the running time is O(n), since there are n elements to rehash and the table size is roughly 2n, but it is actually not all that bad, because it happens very infrequently. In particular, there must have been n/2 inserts prior to the last rehash, so it essentially adds a constant cost to each insertion.

0	6
1	15
2	
3	24
4	
5	
6	13

Figure 2.19 Closed hash table with linear probing with input 13,15, 6, 24

0	6
1	15
2	23
3	24
4	
5	
6	13

Figure 2.20 Closed hash table with linear probing after 23 is inserted

0	
1	
2	
3	
4	
5	

6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Figure 2.21 Closed hash table after rehashing

Rehashing can be implemented in several ways with Quadratic probing. One alternative is to rehash as soon as the table is half full. The other extreme is to rehash only when an insertion fails. A third, middle of the road, strategy is to rehash when the table reaches a certain load factor. Rehashing frees the programmer from worrying about the table size and is important because hash tables cannot be made arbitrarily large in complex programs.

```
HASH_TABLE

rehash( HASH_TABLE H )

{

unsigned int i, old_size;

cell *old_cells;

/*1*/       old_cells = H->the_cells;

/*2*/       old_size = H->table_size;

/* Get a new, empty table */
```

```
/*3*/       H = initialize_table( 2*old_size );

/* Scan through old table, reinserting into new */

/*4*/       for( i=0; i<old_size; i++ )

/*5*/           if( old_cells[i].info == legitimate )

/*6*/               insert( old_cells[i].element, H );

/*7*/       free( old_cells );

/*8*/       return H;

}
```

Figure 2.22 Shows that rehashing is simple to implement.

2.6 BUCKET HASHING

Closed hashing stores all records directly in the hash table. Each record *R* with key value k_R has a **home position** that is **h**(k_R), the slot computed by the hash function. If *R* is to be inserted and another record already occupies *R*'s home position, then *R* will be stored at some other slot in the table. It is the business of the collision resolution policy to determine which slot that will be. Naturally, the same policy must be followed during search as during insertion, so that any record not found in its home position can be recovered by repeating the collision resolution process.

One implementation for closed hashing groups hash table slots into **buckets**. The *M* slots of the hash table are divided into *B* buckets, with each bucket consisting of *M/B* slots. The hash function assigns each record to the first slot within one of the buckets. If this slot is already occupied, then the bucket slots are searched sequentially until an open slot is found. If a bucket is entirely full, then the record is stored in an **overflow bucket** of infinite capacity at the end of the table. All buckets share the same overflow bucket. A good implementation will use a hash function that distributes the records evenly among the buckets so that as few records as possible go into the overflow bucket.

When searching for a record, the first step is to hash the key to determine which bucket should contain the record. The records in this bucket are then searched. If the desired key value is not found and the bucket still has free slots, then the search is complete. If the bucket is full, then it is possible that the desired record is stored in the overflow bucket. In this case, the overflow bucket must be searched until the record is found or all records in

the overflow bucket have been checked. If many records are in the overflow bucket, this will be an expensive process.

Exercises

1. Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function h(x) = x(mod 10), show the resulting

a. open hash table

b. closed hash table using linear probing

c. closed hash table using quadratic probing

d. closed hash table with second hash function h2(x) = 7 - (x mod 7)

2. Show the result of rehashing the hash tables in above input data.

3. Write a program to compute the number of collisions required in a long random sequence of insertions using linear probing, quadratic probing, and double hashing.

4. A large number of deletions in an open hash table can cause the table to be fairly empty, which wastes space. In this case, we can rehash to a table half as large. Assume that we rehash to a larger table when there are twice as many elements as the table size. How empty should an open table be before we rehash to a smaller table?

5. An alternative collision resolution strategy is to define a sequence, f(i) = ri, where r0 = 0 and r1, r2, . . . , rn is a random permutation of the first n integers (each integer appears exactly once).

a. Prove that under this strategy, if the table is not full, then the collision can always be resolved.

b. Would this strategy be expected to eliminate clustering?

c. If the load factor of the table is ƛ, what is the expected time to perform an insert?

d. If the load factor of the table is ƛ, what is the expected time for a successful search?

e. Give an efficient algorithm (theoretically as well as practically) to generate the random sequence. Explain why the rules for choosing P are important.

6. What are the advantages and disadvantages of the various collision resolution strategies?

7. A spelling checker reads an input file and prints out all words not in some online dictionary. Suppose the dictionary contains 30,000 words and the file is one megabyte, so that the algorithm can make only one pass through the input file. A simple strategy is to read the dictionary into a hash table and look for each input word as it is read. Assuming that an average word is seven characters and that it is possible to store words of length l in l + 1 bytes (so space waste is not much of a consideration), and assuming a closed table, how much space does this require?

8. If memory is limited and the entire dictionary cannot be stored in a hash table, we can still get an efficient algorithm that almost always works. We declare an array H_TABLE of bits (initialized to zeros) from 0 to TABLE_SIZE - 1. As we read in a word, we set H_TABLE[hash(word)] = 1. Which of the following is true?

a. If a word hashes to a location with value 0, the word is not in the dictionary.

b. If a word hashes to a location with value 1, then the word is in the dictionary.

Suppose we choose TABLE_SIZE = 300,007.

c. How much memory does this require?

d. What is the probability of an error in this algorithm?

e. A typical document might have about three actual misspellings per page of 500 words. Is this algorithm usable?

9. *Describe a procedure that avoids initializing a hash table (at the expense of memory).

10. Suppose we want to find the first occurrence of a string p1p2. . . pk in a long input string a1a2 . . . an. We can solve this problem by hashing the pattern string, obtaining a hash value hp, and comparing this value with the hash value formed from a1a2 . . . ak,a2a3 . . . ak+1,a3a4 . . . ak+2, and so on until an-k+1an-k+2 . . . an. If we have a match of hash values, we compare the strings

character by character to verify the match. We return the position (in a) if the strings actually do match, and we continue in the unlikely event that the match is false.

a. Show that if the hash value of aiai+1 . . . ai + k - 1 is known, then the hash value of ai+1ai+2 . . . ai+k can be computed in constant time.

b. Show that the running time is O(k + n) plus the time spent refuting false matches.

c. Show that the expected number of false matches is negligible.

d. Write a program to implement this algorithm.

e. Describe an algorithm that runs in O(k + n) worst case time.

f. Describe an algorithm that runs in O(n/k) average time.

11. Write a note on bucket hashing.