and b as this would take $O(n + m)$ time.

10. [*T. Gonzalez*] Let $s = \{s_1, s_2, \cdots, s_n\}$ and $t = \{t_1, t_2, \cdots, t_r\}$ be two sets. Assume $1 \le s_i \le m$, $1 \le i \le n$, and $1 \le t_i \le m$, $1 \le i \le r$. Using the idea of Exercise 9, write a function to determine if $s \subseteq t$. Your function should work in $O(r + n)$ time. Since $s \equiv t$ iff $s \subseteq t$ and $t \subseteq s$, one can determine in linear time whether two sets are the same. How much space is needed by your function?

11. [*T. Gonzalez*] Using the idea of Exercise 9, write an $O(n + m)$ time function to carry out the task of *Verify2* (Program 7.3). How much space does your function need?

12. Using the notation of Section 8.2.4, show that when linear probing is used

$$S_n = \frac{1}{n}\sum_{i=0}^{n-1} U_i$$

Using this equation and the approximate equality

$$U_n \approx \frac{1}{2}\left[1 + \frac{1}{(1-\alpha)^2}\right] \quad \text{where } \alpha = \frac{n}{b}$$

show that

$$S_n \approx \frac{1}{2}\left[1 + \frac{1}{(1-\alpha)}\right]$$

8.3 DYNAMIC HASHING

8.3.1 Motivation for Dynamic Hashing

To ensure good performance, it is necessary to increase the size of a hash table whenever its loading density exceeds a prespecified threshold. So, for example, if we currently have b buckets in our hash table and are using the division hash function with divisor $D = b$, then, when an insert causes the loading density to exceed the prespecified threshold, we use array doubling to increase the number of buckets to $2b + 1$. At the same time, the hash function divisor changes to $2b + 1$. This change in divisor requires us to rebuild the hash table by collecting all dictionary pairs in the original smaller size table and reinserting these into the new larger table. We cannot simply copy dictionary entries from the smaller table into corresponding buckets of the bigger table as the home bucket for each entry has potentially changed. For very large dictionaries that must be accessible on a 24/7 basis, the required rebuild means that dictionary operations must be suspended for unacceptably long periods while the rebuild is in progress. Dynamic

hashing, which also is known as extendible hashing, aims to reduce the rebuild time by ensuring that each rebuild changes the home bucket for the entries in only 1 bucket. In other words, although table doubling increases the total time for a sequence of n dictionary operations by only $O(n)$, the time required to complete an insert that triggers the doubling is excessive in the context of a large dictionary that is required to respond quickly on a per operation basis. The objective of dynamic hashing is to provide acceptable hash table performance on a per operation basis.

We consider two forms of dynamic hashing—one uses a directory and the other does not—in this section. For both forms, we use a hash function h that maps keys into non-negative integers. The range of h is assumed to be sufficiently large and we use $h(k,p)$ to denote the integer formed by the p least significant bits of $h(k)$.

For the examples of this section, we use a hash function $h(k)$ that transforms keys into 6-bit non-negative integers. Our example keys will be two characters each and h transforms letters such as A, B and C into the bit sequence 100, 101, and 110, respectively. Digits 0 through 7 are transformed into their 3-bit representation. Figure 8.7 shows 8 possible 2 character keys together with the binary representation of $h(k)$ for each. For our example hash function, $h(A0,1) = 0$, $h(A1,3) = 1$, $h(B1,4) = 1001 = 9$, and $h(C1,6) = 110\,001 = 49$.

k	$h(k)$
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C1	110 001
C2	110 010
C3	110 011
C5	110 101

Figure 8.7: An example hash function

8.3.2 Dynamic Hashing Using Directories

We employ a directory, d, of pointers to buckets. The size of the directory depends on the number of bits of $h(k)$ used to index into the directory. When indexing is done using, say, $h(k, 2)$, the directory size is $2^2 = 4$; when $h(k, 5)$ is used, the directory size is 32. The number of bits of $h(k)$ used to index the directory is called the *directory depth*. The

size of the directory is 2^t, where t is the directory depth and the number of buckets is at most equal to the directory size. Figure 8.8 (a) shows a dynamic hash table that contains the keys A0, B0, A1, B1, C2, and C3. This hash table uses a directory whose depth is 2 and uses buckets that have 2 slots each. In Figure 8.8, the directory is shaded while the buckets are not. In practice, the bucket size is often chosen to match some physical characteristic of the storage media. For example, when the dictionary pairs reside on disk, a bucket may correspond to a disk track or sector.

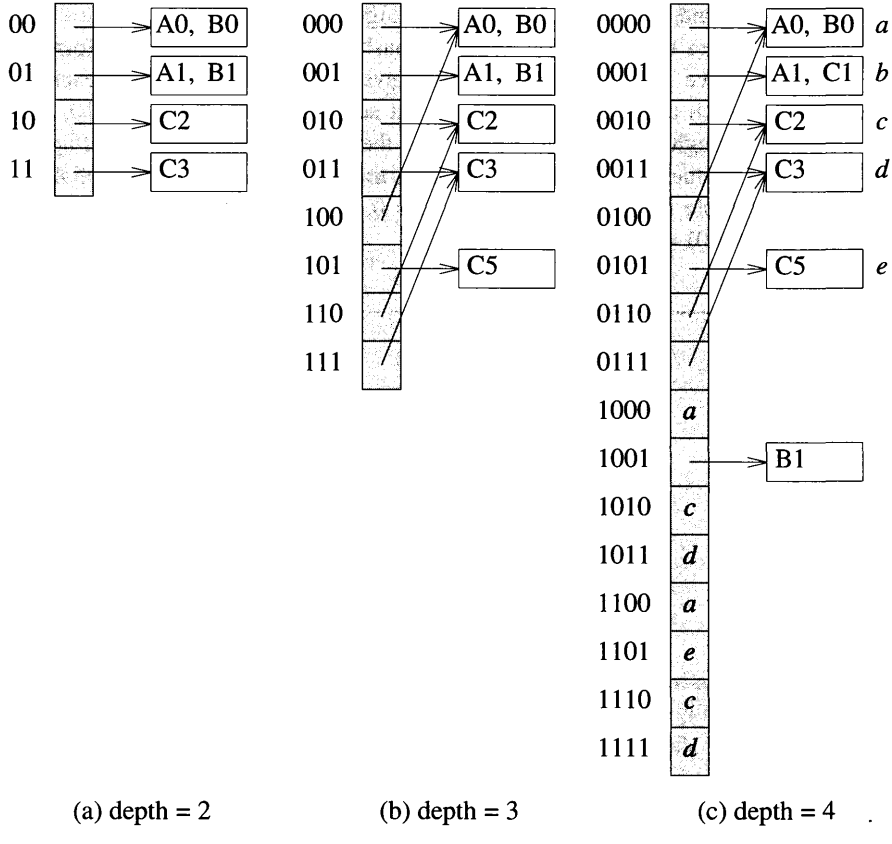


Figure 8.8: Dynamic hash tables with directories

To search for a key k, we merely examine the bucket pointed to by $d[h(k,t)]$, where t is the directory depth.

Suppose we insert C5 into the hash table of Figure 8.8 (a). Since, $h(\text{C5},2) = 01$,

we follow the pointer, $d[01]$, in position 01 of the directory. This gets us to the bucket with A1 and B1. This bucket is full and we get a bucket overflow. To resolve the overflow, we determine the least u such that $h(k,u)$ is not the same for all keys in the overflowed bucket. In case the least u is greater than the directory depth, we increase the directory depth to this least u value. This requires us to increase the directory size but not the number of buckets. When the directory size doubles, the pointers in the original directory are duplicated so that the pointers in each half of the directory are the same. A quadrupling of the directory size may be handled as two doublings and so on. For our example, the least u for which $h(k,u)$ is not the same for A1, B1, and C5 is 3. So, the directory is expanded to have depth 3 and size 8. Following the expansion, $d[i] = d[i + 4], 0 \le i < 4$.

Following the resizing (if any) of the directory, we split the overflowed bucket using $h(k,u)$. In our case, the overflowed bucket is split using $h(k, 3)$. For A1 and B1, $h(k, 3) = 001$ and for C5, $h(k, 3) = 101$. So, we create a new bucket with C5 and place a pointer to this bucket in $d[101]$. Figure 8.8 (b) shows the result. Notice that each dictionary entry is in the bucket pointed at by the directory position $h(k, 3)$, although, in some cases the dictionary entry is also pointed at by other buckets. For example, bucket 100 also points to A0 and B0, even though $h(A\,0,3) = h(B\,0,3) \neq 000$.

Suppose that instead of C5, we were to insert C1. The pointer in position $h(C1,2) = 01$ of the directory of Figure 8.8 (a) gets us to the same bucket as when we were inserting C5. This bucket overflows. The least u for which $h(k,u)$ isn't the same for A1, B1 and C1 is 4. So, the new directory depth is 4 and its new size is 16. The directory size is quadrupled and the pointers $d[0:3]$ are replicated 3 times to fill the new directory. When the overflowed bucket is split, A1 and C1 are placed into a bucket that is pointed at by $d[0001]$ and B1 into a bucket pointed at by $d[1001]$.

When the current directory depth is greater than or equal to u, some of the other pointers to the split bucket also must be updated to point to the new bucket. Specifically, the pointers in positions that agree with the last u bits of the new bucket need to be updated. The following example illustrates this. Consider inserting A4 ($h(A4) = 100$ 100) into Figure 8.8 (b). Bucket $d[100]$ overflows. The least u is 3, which equals the directory depth. So, the size of the directory is not changed. Using $h(k, 3)$, A0 and B0 hash to 000 while A4 hashes to 100. So, we create a new bucket for A4 and set $d[100]$ to point to this new bucket.

As a final insert example, consider inserting C1 into Figure 8.8 (b). $h(C1,3) = 001$. This time, bucket $d[001]$ overflows. The minumum u is 4 and so it is necessary to double the directory size and increase the directory depth to 4. When the directory is doubled, we replicate the pointers in the first half into the second half. Next we split the overflowed bucket using $h(k, 4)$. Since $h(k, 4) = 0001$ for A1 and C1 and 1001 for B1, we create a new bucket with B1 and put C1 into the slot previously occupied by B1. A pointer to the new bucket is placed in $d[1001]$. Figure 8.8 (c) shows the resulting configuration. For clarity, several of the bucket pointers have been replaced by lowercase letters indicating the bucket pointed to.

Deletion from a dynamic hash table with a directory is similar to insertion. Although dynamic hashing employs array doubling, the time for this array doubling is considerably less than that for the array doubling used in static hashing. This is so because, in dynamic hashing, we need to rehash only the entries in the bucket that overflows rather than all entries in the table. Further, savings result when the directory resides in memory while the buckets are on disk. A search requires only 1 disk access; an insert makes 1 read and 2 write accesses to the disk, the array doubling requires no disk access.

8.3.3 Directoryless Dynamic Hashing

As the name suggests, in this method, which also is known as linear dynamic hashing, we dispense with the directory, d, of bucket pointers used in the method of Section 8.3.2. Instead, an array, ht, of buckets is used. We assume that this array is as large as possible and so there is no possibility of increasing its size dynamically. To avoid initializing such a large array, we use two variables q and r, $0 \le q < 2^r$, to keep track of the *active buckets*. At any time, only buckets 0 through $2^r + q - 1$ are active. Each active bucket is the start of a chain of buckets. The remaining buckets on a chain are called *overflow buckets*. Informally, r is the number of bits of $h(k)$ used to index into the hash table and q is the bucket that will split next. More accurately, buckets 0 through $q - 1$ as well as buckets 2^r through $2^r + q - 1$ are indexed using $h(k, r + 1)$ while the remaining active buckets are indexed using $h(k, r)$. Each dictionary pair is either in an active or an overflow bucket.

Figure 8.9 (a) shows a directoryless hash table ht with $r = 2$ and $q = 0$. The hash function is that of Figure 8.7, $h(\text{B4}) = 101\ 100$, and $h(\text{B5}) = 101\ 101$. The number of active buckets is 4 (indexed 00, 01, 10, and 11). The index of an active bucket identifies its chain. Each active bucket has 2 slots and bucket 00 contains B4 and A0. There are 4 bucket chains, each chain begins at one of the 4 active buckets and comprises only that active bucket (i.e., there are no overflow buckets). In Figure 8.9 (a), all keys have been mapped into chains using $h(k, 2)$. In Figure 8.9 (b), $r = 2$ and $q = 1$; $h(k, 3)$ has been used for chains 000 and 100 while $h(k, 2)$ has been used for chains 001, 010, and 011. Chain 001 has an overflow bucket; the capacity of an overflow bucket may or may not be the same as that of an active bucket.

To search for k, we first compute $h(k,r)$. If $h(k,r) < q$, then k, if present, is in a chain indexed using $h(k, r + 1)$. Otherwise, the chain to examine is given by $h(k, r)$. Program 8.5 gives the algorithm to search a directoryless dynamic hash table.

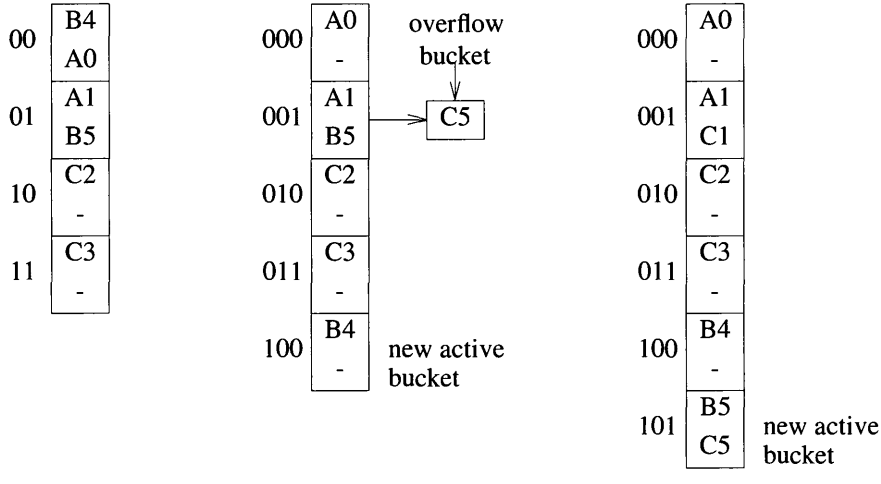
To insert C5 into the table of Figure 8.9 (a), we use the search algorithm of Program 8.5 to determine whether or not C5 is in the table already. Chain 01 is examined and we verify that C5 is not present. Since the active bucket for the searched chain is full, we get an overflow. An overflow is handled by activating bucket $2^r + q$; reallocating the entries in the chain q between q and the newly activated bucket (or chain) $2^r + q$,

if $(h(k,r) < q)$ search the chain that begins at bucket $h(k,r+1)$;
else search the chain that begins at bucket $h(k,r)$;

Program 8.5: Searching a directoryless hash table

and incrementing q by 1. In case q now becomes 2^r, we increment r by 1 and reset q to 0. The reallocation is done using $h(k, r + 1)$. Finally, the new pair is inserted into the chain where it would be searched for by Program 8.5 using the new r and q values.

For our example, bucket 4 = 100 is activated and the entries in chain 00 ($q = 0$) are rehashed using $r + 1 = 3$ bits. B4 hashes to the new bucket 100 and A0 to bucket 000. Following this, $q = 1$ and $r = 2$. A search for C5 would examine chain 1 and so C5 is added to this chain using an overflow bucket (see Figure 8.9 (b)). Notice that at this time, the keys in buckets 001, 010 and 011 are hashed using $h(k, 2)$ while those in buckets 000 and 100 are hashed using $h(k, 3)$.



(a) $r = 2$, $q = 0$ (b) Insert C5, $r = 2$, $q = 1$ (c) Insert C1, $r = 2$, $q = 2$

Figure 8.9: Inserting into a directoryless dynamic hash table

Let us now insert C1 into the table of Figure 8.9 (b). Since, $h(C1,2) = 01 = q$, chain 01 = 1 is examined by our search algorithm (Program 8.5). The search verifies that C1 is not in the dictionary. Since the active bucket 01 is full, we get an overflow. We activate bucket $2^r + q = 5 = 101$ and rehash the keys A1, B5, and C5 that are in chain q. The rehashing is done using 3 bits. A1 is hashed into bucket 001 while B5 and C5 hash into bucket 101. q is incremented by 1 and the new key C1 is inserted into bucket 001. Figure 8.9 (c) shows the result.

EXERCISES

1. Write an algorithm to insert a dictionary pair into a dynamic hash table that uses a directory.
2. Write an algorithm to delete a dictionary pair from a dynamic hash table that uses a directory.
3. Write an algorithm to insert a dictionary pair into a directoryless dynamic hash table.
4. Write an algorithm to delete a dictionary pair from a directoryless dynamic hash table.

8.4 BLOOM FILTERS

8.4.1 An Application—Differential Files

Consider an application where we are maintaining an indexed file. For simplicity, assume that there is only one index and hence just a single key. Further assume that this is a dense index (i.e., one that has an entry for each record in the file) and that updates to the file (inserts, deletes, and changes to an existing record) are permitted. It is necessary to keep a backup copy of the index and file so that we can recover from accidental loss or failure of the working copy. This loss or failure may occur for a variety of reasons, which include corruption of the working copy due to a malfunction of the hardware or software. We shall refer to the working copies of the index and file as the *master index* and *master file*, respectively.

Since updates to the file and index are permitted, the backup copies of these generally differ from the working copies at the time of failure. So, it is possible to recover from the failure only if, in addition to the backup copies, we have a log of all updates made since the backup copies were created. We shall call this log the *transaction log*. To recover from the failure, it is necessary to process the backup copies and the transaction log to reproduce an index and file that correspond to the working copies at the time of failure. The time needed to recover is therefore a function of the sizes of the backup index and file and the size of the transaction log. The recovery time can be reduced by making more frequent backups. This results in a smaller transaction log. Making sufficiently frequent backups of the master index and file is not practical when these are