	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	0001
12		0100
13	F	0000
14	G	0000
...		

→

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9		**0011**
10	E	0000
11	H	0001
12	**B**	0100
13	F	0000
14	G	0000
...		

→

	Item	Hop
...		
6	C	**1001**
7	A	1100
8	D	0010
9	**I**	0011
10	E	0000
11	H	0001
12	B	0100
13	F	0000
14	G	0000
...		

A: 7
B: 9
C: 6
D: 7
E: 8
F: 12
G: 11
H: 9
I: 6

Figure 5.49 Hopscotch hashing table. Insertion of *I* continues: Next, *B* is evicted, and finally, we have a spot that is close enough to the hash value and can insert *I*.

5.8 Universal Hashing

Although hash tables are very efficient and have constant average cost per operation, assuming appropriate load factors, their analysis and performance depend on the hash function having two fundamental properties:

1. The hash function must be computable in constant time (i.e., independent of the number of items in the hash table).
2. The hash function must distribute its items uniformly among the array slots.

In particular, if the hash function is poor, then all bets are off, and the cost per operation can be linear. In this section, we discuss **universal hash functions**, which allow us to choose the hash function randomly in such a way that condition 2 above is satisfied. As in Section 5.7, we use M to represent *TableSize*. Although a strong motivation for the use of universal hash functions is to provide theoretical justification for the assumptions used in the classic hash table analyses, these functions can also be used in applications that require a high level of robustness, in which worst-case (or even substantially degraded) performance, perhaps based on inputs generated by a saboteur or hacker, simply cannot be tolerated.

As in Section 5.7, we use M to represent *TableSize*.

Definition 5.1

A family H of hash functions is *universal*, if for any $x \neq y$, the number of hash functions h in H for which $h(x) = h(y)$ is at most $|H|/M$.

Notice that this definition holds for each pair of items, rather than being averaged over all pairs of items. The definition above means that if we choose a hash function randomly from a universal family H, then the probability of a collision between any two distinct items

is at most $1/M$, and when adding into a table with N items, the probability of a collision at the initial point is at most N/M, or the load factor.

The use of a universal hash function for separate chaining or hopscotch hashing would be sufficient to meet the assumptions used in the analysis of those data structures. However, it is not sufficient for cuckoo hashing, which requires a stronger notion of independence. In cuckoo hashing, we first see if there is a vacant location; if there is not, and we do an eviction, a different item is now involved in looking for a vacant location. This repeats until we find the vacant location, or decide to rehash [generally within $O(\log N)$ steps]. In order for the analysis to work, each step must have a collision probability of N/M independently, with a different item x being subject to the hash function. We can formalize this independence requirement in the following definition.

Definition 5.2

A family H of hash functions is *k-universal*, if for any $x_1 \neq y_1$, $x_2 \neq y_2, \ldots, x_k \neq y_k$, the number of hash functions h in H for which $h(x_1) = h(y_1)$, $h(x_2) = h(y_2), \ldots$, and $h(x_k) = h(y_k)$ is at most $|H|/M^k$.

With this definition, we see that the analysis of cuckoo hashing requires an $O(\log N)$-universal hash function (after that many evictions, we give up and rehash). In this section we look only at universal hash functions.

To design a simple universal hash function, we will assume first that we are mapping very large integers into smaller integers ranging from 0 to $M - 1$. Let p be a prime larger than the largest input key.

Our universal family H will consist of the following set of functions, where a and b are chosen randomly:

$$H = \{H_{a,b}(x) = ((ax + b) \mod p) \mod M, \text{ where } 1 \leq a \leq p - 1, 0 \leq b \leq p - 1\}$$

For example, in this family, three of the possible random choices of (a, b) yield three different hash functions:

$$H_{3,7}(x) = ((3x + 7) \mod p) \mod M$$
$$H_{4,1}(x) = ((4x + 1) \mod p) \mod M$$
$$H_{8,0}(x) = ((8x) \mod p) \mod M$$

Observe that there are $p(p - 1)$ possible hash functions that can be chosen.

Theorem 5.4

The hash family $H = \{H_{a,b}(x) = ((ax + b) \mod p) \mod M$, where $1 \leq a \leq p - 1$, $0 \leq b \leq p - 1\}$ is universal.

Proof

Let x and y be distinct values, with $x > y$, such that $H_{a,b}(x) = H_{a,b}(y)$.

Clearly if $(ax + b) \mod p$ is equal to $(ay + b) \mod p$, then we will have a collision. However, this cannot happen: Subtracting equations yields $a(x - y) \equiv 0 \pmod{p}$, which would mean that p divides a or p divides $x - y$, since p is prime. But neither can happen, since both a and $x - y$ are between 1 and $p - 1$.

So let $r = (ax + b) \mod p$ and let $s = (ay + b) \mod p$, and by the above argument, $r \neq s$. Thus there are p possible values for r, and for each r, there are $p - 1$ possible

values for s, for a total of $p(p-1)$ possible (r,s) pairs. Notice that the number of (a,b) pairs and the number of (r,s) pairs is identical; thus each (r,s) pair will correspond to exactly one (a,b) pair if we can solve for (a,b) in terms of r and s. But that is easy: As before, subtracting equations yields $a(x-y) \equiv (r-s) \pmod{p}$, which means that by multiplying both sides by the unique multiplicative inverse of $(x-y)$ (which must exist, since $x-y$ is not zero and p is prime), we obtain a, in terms of r and s. Then b follows.

Finally, this means that the probability that x and y collide is equal to the probability that $r \equiv s \pmod{M}$, and the above analysis allows us to assume that r and s are chosen randomly, rather than a and b. Immediate intuition would place this probability at $1/M$, but that would only be true if p were an exact multiple of M, and all possible (r,s) pairs were equally likely. Since p is prime, and $r \neq s$, that is not exactly true, so a more careful analysis is needed.

For a given r, the number of values of s that can collide mod M is at most $\lceil p/M \rceil - 1$ (the -1 is because $r \neq s$). It is easy to see that this is at most $(p-1)/M$. Thus the probability that r and s will generate a collision is at most $1/M$ (we divide by $p-1$, because, as mentioned earlier in the proof, there are only $p-1$ choices for s given r). This implies that the hash family is universal.

Implementation of this hash function would seem to require two mod operations: one mod p and the second mod M. Figure 5.50 shows a simple implementation in C++, assuming that M is significantly less than $2^{31}-1$. Because the computations must now be exactly as specified, and thus overflow is no longer acceptable, we promote to `long long` computations, which are at least 64 bits.

However, we are allowed to choose any prime p, as long as it is larger than M. Hence, it makes sense to choose a prime that is most favorable for computations. One such prime is $p = 2^{31}-1$. Prime numbers of this form are known as Mersenne primes; other Mersenne primes include 2^5-1, $2^{61}-1$ and $2^{89}-1$. Just as a multiplication by a Mersenne prime such as 31 can be implemented by a bit shift and a subtract, a mod operation involving a Mersenne prime can also be implemented by a bit shift and an addition:

Suppose $r \equiv y \pmod{p}$. If we divide y by $(p+1)$, then $y = q'(p+1) + r'$, where q' and r' are the quotient and remainder, respectively. Thus, $r \equiv q'(p+1) + r' \pmod{p}$. And since $(p+1) \equiv 1 \pmod{p}$, we obtain $r \equiv q' + r' \pmod{p}$.

Figure 5.51 implements this idea, which is known as the **Carter-Wegman trick**. On line 8, the bit shift computes the quotient and the bitwise-and computes the remainder when dividing by $(p+1)$; these bitwise operations work because $(p+1)$ is an exact power

```
1  int universalHash( int x, int A, int B, int P, int M )
2  {
3      return static_cast<int>( ( ( static_cast<long long>( A ) * x ) + B ) % P ) % M;
4  }
```

Figure 5.50 Simple implementation of universal hashing

```
1  const int DIGS = 31;
2  const int mersennep = (1<<DIGS) - 1;
3
4  int universalHash( int x, int A, int B, int M )
5  {
6      long long hashVal = static_cast<long long>( A ) * x + B;
7
8      hashVal = ( ( hashVal >> DIGS ) + ( hashVal & mersennep ) );
9      if( hashVal >= mersennep )
10         hashVal -= mersennep;
11
12     return static_cast<int>( hashVal ) % M;
13 }
```

Figure 5.51 Simple implementation of universal hashing

of two. Since the remainder could be almost as large as p, the resulting sum might be larger than p, so we scale it back down at lines 9 and 10.

Universal hash functions exist for strings also. First, choose any prime p, larger than M (and larger than the largest character code). Then use our standard string hashing function, choosing the multiplier randomly between 1 and $p - 1$ and returning an intermediate hash value between 0 and $p - 1$, inclusive. Finally, apply a universal hash function to generate the final hash value between 0 and $M - 1$.

5.9 Extendible Hashing

Our last topic in this chapter deals with the case where the amount of data is too large to fit in main memory. As we saw in Chapter 4, the main consideration then is the number of disk accesses required to retrieve data.

As before, we assume that at any point we have N records to store; the value of N changes over time. Furthermore, at most M records fit in one disk block. We will use $M = 4$ in this section.

If either probing hashing or separate chaining hashing is used, the major problem is that collisions could cause several blocks to be examined during a search, even for a well-distributed hash table. Furthermore, when the table gets too full, an extremely expensive rehashing step must be performed, which requires $O(N)$ disk accesses.

A clever alternative, known as **extendible hashing**, allows a search to be performed in two disk accesses. Insertions also require few disk accesses.

We recall from Chapter 4 that a B-tree has depth $O(\log_{M/2} N)$. As M increases, the depth of a B-tree decreases. We could in theory choose M to be so large that the depth of the B-tree would be 1. Then any search after the first would take one disk access, since, presumably, the root node could be stored in main memory. The problem with this strategy is that the branching factor is so high that it would take considerable processing to determine which leaf the data was in. If the time to perform this step could be reduced, then we would have a practical scheme. This is exactly the strategy used by extendible hashing.