BLM5106- Advanced Algorithm Analysis and Design

H. İrem Türkmen

Introduction to algorithms TH Cormen, CE Leiserson, RL Rivest, C Stein
www.otago.ac.nz, Algorithms and Data Structures
Data structures and algorithm analysis in C++ Mark Allen Weiss
https://codecapsule.com/2013/08/11/hopscotch-hashing/

Simple Hash Functions

- The division method

- The multiplication method

- How to hash strings? (Radix Notation)

Division Method

In the **division method** for creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

$h(k) = k \bmod m$.

For example, if the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$. Since it requires only a single division operation, hashing by division is quite fast.

When using the division method, we usually avoid certain values of m. For example, m should not be a power of 2, since if $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k.

Division Method

$k = 18, 22, 26, 30, 34$

And we choose $m = 16$ (which is 2^4), then:

$$h(k) = k \mod 16$$

$18 = 0001\ 0010$
$34 = 0010\ 0010$
$22 = 0001\ 0110$

Results:

$18 \mod 16 = 2, \quad 22 \mod 16 = 6, \quad 26 \mod 16 = 10, \quad 30 \mod 16 = 14, \quad 34 \mod 16 = 2$

Here, we see a collision at **2** for keys 18 and 34.

If we instead use $m = 17$ (a prime), the distribution is better:

$18 \mod 17 = 1, \quad 22 \mod 17 = 5, \quad 26 \mod 17 = 9, \quad 30 \mod 17 = 13, \quad 34 \mod 17 = 0$

Division Method

A prime not too close to an exact power of 2 is often a good choice for m. For example, suppose we wish to allocate a hash table, with collisions resolved by chaining, to hold roughly $n = 2000$ character strings, where a character has 8 bits. We don't mind examining an average of 3 elements in an unsuccessful search, and so we allocate a hash table of size $m = 701$. We could choose $m = 701$ because it is a prime near 2000/3 but not near any power of 2. Treating each key k as an integer, our hash function would be

$$h(k) = k \bmod 701 \ .$$

Multiplication Method

$$h(k) = \lfloor m(kA \quad \mathrm{mod}\ 1) \rfloor$$

where:

- k is the key,

- A is a constant in the range $0 < A < 1$ (often chosen as an irrational number),

- m is the table size,

- $kA \quad \mathrm{mod}\ 1$ extracts the **fractional part** of kA,

- Multiplying by m scales this fraction to the table size,

- Taking the **floor** ($\lfloor \cdot \rfloor$) gives an integer index in the range $[0, m - 1]$.

Multiplication Method

Although this method works with any value of the constant A, it works better with some values than with others. The optimal choice depends on the characteristics of the data being hashed. Knuth suggests that

$$A \approx (\sqrt{5} - 1)/2 = 0.6180339887\ldots$$

is likely to work reasonably well.

Why It Works Well

- Unlike the **division method**, it works well for any m (not just primes).

- It spreads keys more evenly by using the fractional part of multiplication.

- It avoids patterns caused by structured input data.

Multiplication Method

- $k = 1234$ (a key),

- $m = 10$ (table size),

- $A = 0.6180339887$.

1. Compute kA:

$$1234 \times 0.6180339887 = 763.272$$

2. Take the fractional part:

$$0.272$$

3. Multiply by m:

$$0.272 \times 10 = 2.72$$

4. Take the **floor**:

$$h(1234) = \lfloor 2.72 \rfloor = 2$$

Multiplication vs Division

Method	Pros	Cons
Division Method	Simple, efficient if m is prime	Avoids powers of two but can cause patterns
Multiplication Method	Works with any m, spreads values well	Slightly more complex (requires multiplication & flooring)

Open Addressing

- In **open addressing**, all elements occupy the hash table itself. That is, each table entry contains either an element of the dynamic set or NIL. When searching for an element, we systematically examine table slots until either we find the desired element or we have ascertained that the element is not in the table.

- The advantage of open addressing is that it avoids pointers altogether. Instead of following pointers, we *compute* the sequence of slots to be examined. The extra memory freed by not storing pointers provides the hash table with a larger number of slots for the same amount of memory, potentially yielding fewer collisions and faster retrieval.

- To perform insertion using open addressing, we successively examine, or **probe**, the hash table until we find an empty slot in which to put the key. Instead of being fixed in the order 0,1,..,m (which requires O(n) search time), the sequence of positions probed *depends upon the key being inserted*.

Open Addressing

With open addressing, we require that for every key k, the **_probe sequence_**

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$$

HASH-INSERT(T,k)

```
1   i = 0
2   repeat
3         j = h(k,i)
4         if T[j] == NIL
5               T[j] = k
6               return j
7         else i = i + 1
8   until i == m
9   error "hash table overflow"
```

HASH-SEARCH(T,k)

```
1   i = 0
2   repeat
3         j = h(k,i)
4         if T[j] == k
5               return j
6         i = i + 1
7   until T[j] == NIL or i == m
8   return NIL
```

Open Addressing

- What about deleting?
- When we delete a key from slot i , we cannot simply mark that slot as empty by storing NIL in it. Why?
- We can solve this problem by marking the slot, storing in it the special value DELETED instead of NIL.
- In that case we have to modify Hash_Insert to treat such a slot as if it were empty so that we can insert a new key there.
- However, search times no longer depend on the load factor
- For this reason chaining is more commonly selected as a collision resolution technique when keys must be deleted.

Techniques for Open Addressing

- Linear Probing

- Quadratic Probing

- Double Hashing

- See samples..

Analysis of open address hashing

Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

- With a probability n/m first probe hits an occupied slot, and a second probe is necessary

- With a probability (n-1)/(m-1), second probe hits an occupied slot, and a third probe is necessary

- With a probaility (n-2)/(m-2), third probe hits an occupied slot, and a fourth probe is necessary ….

Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

$$X \qquad \text{number of probes made in an unsuccessful search}$$

$$A_i, \text{ for } i = 1, 2, \ldots, \qquad i\text{th probe occurs and it is to an occupied slot}$$

$$\Pr\{X \geq i\} = \Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}$$

$$= \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots$$

$$\Pr\{X \geq i\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2}$$

$$E[X] = \sum_{i=1}^{\infty} \Pr\{X \geq i\}$$

$$\leq \sum_{i=1}^{\infty} \alpha^{i-1}$$

$$= \sum_{i=0}^{\infty} \alpha^i$$

$$= \frac{1}{1-\alpha}.$$

observe that $\dfrac{n-i}{m-i} < \dfrac{n}{m}$ since m>n

$$\leq \left(\frac{n}{m}\right)^{i-1}$$
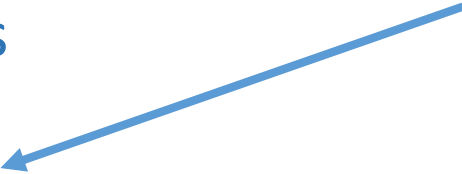
$$= \alpha^{i-1}.$$

Analysis of open address hashing

- ***Theorem***

Given an open-address hash table with load factor $\propto =n/m < 1$, the expected number of probes in an unsuccessful search is at most

$1=1/(1- \propto)$ assuming uniform hashing.

- n/m=0.1 means %10 full
- 1/0.9 corresponds 1.11 probes
- n/m = 0.5 means %50 full
- 1/0.5 corresponds 2 probes
- n/m = 0.9 means %90 full
- 1/0.1 corresponds 10 probes

\# probes increase in a high rate in a %50 full table

Rehashing

- If the table gets too full, the running time for the operations will start taking too long, and insertions might fail for open addressing hashing with quadratic resolution

- A solution, then, is to build another table that is about twice as big (with an associated new hash function) and scan down the entire original hash table, computing the new hash value for each (nondeleted) element and inserting it in the new table.

Rehashing

- $h(x) = x \bmod 7$

Hash table with linear probing with input 13, 15, 6, 24

Hash table with linear probing after 23 is inserted

- The resulting table will be over 70 percent full. Because the table is so full, a new table is created.

Rehashing

- The size of this table is 17, because this is the first prime that is twice as large as the old table size which is 7.

- 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.

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Hash table after rehashing

Pros and Cons?

- 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.
- If this data structure is part of the program, the effect is not noticeable.
- On the other hand, if the hashing is performed as part of an interactive system, then the unfortunate user whose insertion caused a rehash could see a slowdown.

Universal Hashing

- If your hash function is known by an adversarial, it is possible to create a sequence of inputs to hit its worst case.

- Universal hashing attempts to solve this problem by choosing a hash function at random from a family of hash functions.

- Universal Hashing can be used in applications that require a high level of robustness, in which worst-case performance, perhaps based on inputs generated by a saboteur or hacker, simply cannot be tolerated.

Universal Hashing

- Let H be a finite collection of hash functions that map a given universe U of keys into the range {0,1,...m-1}. Such a collection is said to be *universal* if for each pair of distinct keys k,l ϵ U , the number of hash functions h ϵ H for which h(k)=h(l) is at most |H|/m

- In other words, with a hash function randomly chosen from H, the chance of a collision between distinct keys k and l is no more than the chance 1/m of a collision if k and l were randomly and independently chosen from the set {0,1,...m-1}.

Designing an universal class of hash functions

- Modulo Primes Method
- Matrix Method

Designing an universal class of hash functions
Using Primes

- We begin by choosing a prime number p large enough so that every possible key k is in the range of 0 to p -1

- Because we assume that the size of the universe of keys is greater than the number of slots in the hash table, we have p >m.

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

Designing an universal class of hash functions

Using Primes

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

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

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

$$H_{3,7}(x) = ((3x + 7) \bmod p) \bmod M$$

$$H_{4,1}(x) = ((4x + 1) \bmod p) \bmod M$$

$$H_{8,0}(x) = ((8x) \bmod p) \bmod M$$

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

Designing an universal class of hash functions
Using Primes

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

- p = 17, M= 6
- $h_{5,7}$(3)　　　(3*5+7) mod 17 = 5　　　　　　　　　　　　5 mod 6=5
- $h_{3,4}$ (28)　　(3*28+4) mod 17 = (84+4) mod 17=3　　3 mod 6=3
- $h_{10,2}$(3)　　　(10*3+2) mod 17 = 32 mod 17 = 15　　15 mod 6=3

Designing an universal class of hash functions
Matrix Method

- We assume that the keys are u-bits long.

- The size of the key universe is $|U| = 2^u$

- Suppose we want a table of size 2^b, that is, $m = 2^b$.

- We choose a random b × u binary matrix h, and define the hash function as h(x) = hx (mod 2)

- That is to get the hash value, we multiply the matrix h with the u-bit long vector x, taking the sum modulo 2.

- Observe that the product is a b-bit vector, which becomes the hash location for x in the table.

Designing an universal class of hash functions
Matrix Method

What we will do is pick h to be a random b-by-u 0/1 matrix, and define $h(x) = hx$, where we do addition mod 2. These matrices are short and fat. For instance:

$$
\begin{array}{ccc}
h & x & h(x)
\end{array}
$$

$$
\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}
\begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}
=
\begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}
$$

Designing a universal class of hash functions
Matrix Method

- Each b × u matrix is a hash function. How many hash functions are there in this family?

- For instance, our universe could be the IP address space of 32 bits, and the table could have size 256.

- In that case, the matrix h is a 8 × 32 matrix of 0s and 1s, where each bit is equally likely to be 0 or 1.

- Observe that this method produces a family of hash functions.

- Each random matrix h corresponds to a distinct hash function of this family. By choosing h randomly, we are selecting a member of this family uniformly at random.

$$h \quad\quad x \quad\quad h(x)$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$

b*u matrix
u:key size

Designing a universal class of hash functions
## Matrix Method

Claim. Suppose $x \neq y$ are two keys. Then,

$$Prob[h(x) = h(y)] = 1/m = 1/2^b$$

• **Proof?**

Universal hashing with a big table

- Universal hashing, on average, will produce a collision between two random keys $1/m$ of the time, for table size m.

- What's the probability of at least one collision between n keys for a random universal hash function?

- Lets see

Probability of a collision

$y = 1-((m-1)/m)^{\wedge}(n*(n-1)/2)$, and $n = 1000$
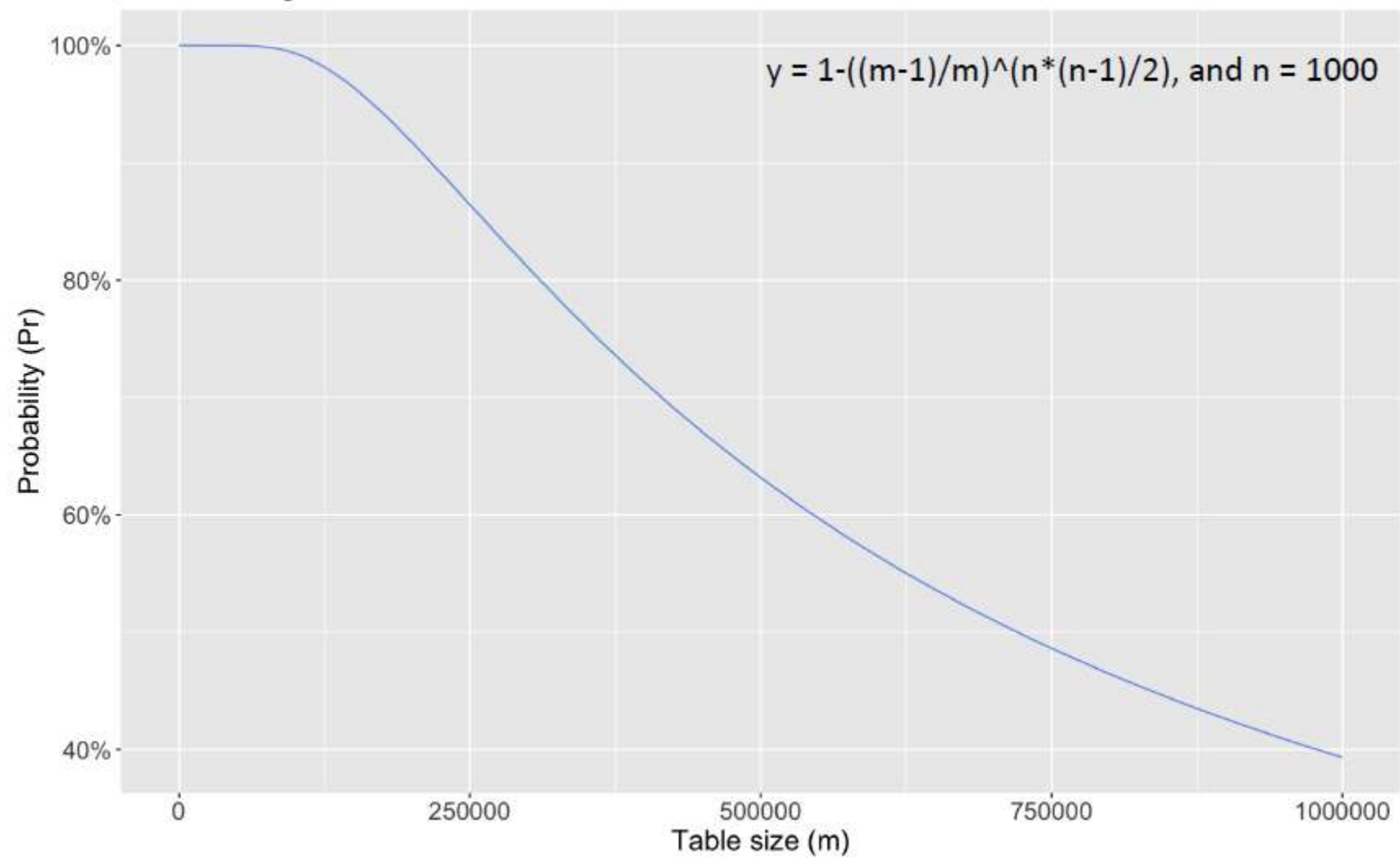
Universal hashing with a big table

- With n = 1000, m = 100000, then Pr((#collisions ≥ 1) = .9932

- If we choose 100 random functions, what's the probability that *all* of them have a collision?

   $.9932^{100} = .507$

1-0.507 : probabilty of at least one of these hash functions does not have collision

That's for a load factor of 1% - not a great way of choosing a perfect hashing function.

Probability of a collision

$y = 1-((m-1)/m)^{(n*(n-1)/2)}$, and $n = 1000$

Probability (Pr) vs Table size (m)
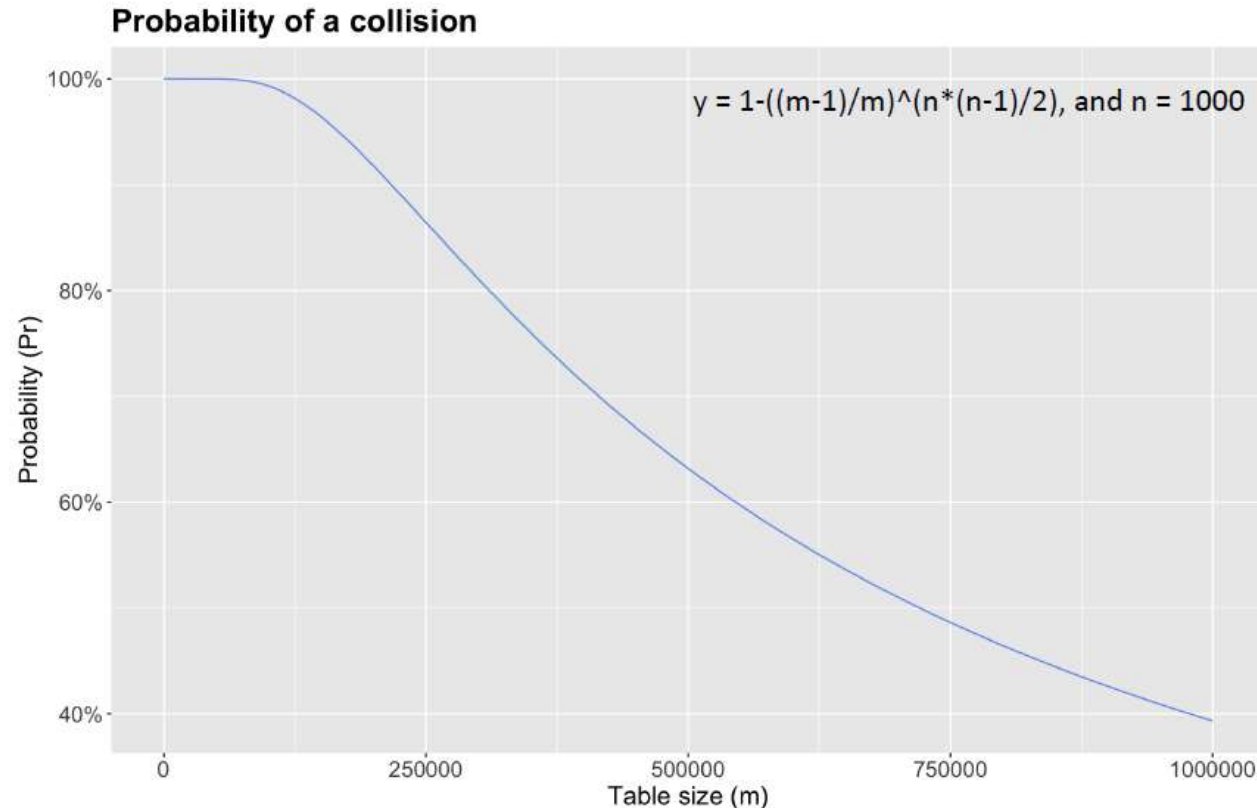
Hash Tables with Worst-Case $O(1)$ Access

- The hash tables that we have examined so far all have the property that with reasonable load factors, and appropriate hash functions, we can expect $O(1)$ cost on average for insertions, removes, and searching.

- But what is the expected worst case for a search assuming a reasonably well-behaved hash function?

- We would like to obtain $O(1)$ worst-case cost.

- In some applications, such as hardware implementations of lookup tables for routers and memory caches, it is especially important that the search have a definite (i.e., constant) amount of completion time.

Perfect Hashing

- Although hashing is often a good choice for its excellent average-case performance, hashing can also provide excellent *worst-case* performance when the set of keys is **static**: once the keys are stored in the table, the set of keys never changes.

- Assume that N is known in advance, so no rehashing is needed. If we are allowed to rearrange items as they are inserted, then $O(1)$ worst-case cost is achievable for searches.

- Some applications naturally have static sets of keys: consider the set of reserved words in a programming language, or the set of file names on a CD-ROM.

Perfect Hashing

- To create a perfect hashing scheme, we use two levels of hashing, with universal hashing at each level.

- Instead of making a linked list of the keys hashing to slot j, however, we use a small **secondary hash table** S_j with an associated hash function h_j.

- By trying several hash functions from a universal class, we can easily achieve the goal of having no collisions in the secondary tables.

Perfect Hashing

- In order to guarantee that there are no collisions at the secondary level, however, we will need to let the size m_j of hash table S_j be the square of the number n_j of keys hashing to slot j .

- Although the quadratic dependence of m_j on n_j may seem likely to cause the overall storage requirement to be excessive, by choosing the first-level hash function well, we can limit the expected total amount of space used to $O(n)$

Perfect Hashing

- The first level is essentially the same as for hashing with chaining: we hash the n keys into m slots using a hash function h carefully selected from a family of universal hash functions.

- If the primary hash function h is good enough:

1. For each slot i, get the secondary hash function h_i by setting $p_i = p$ (p doesn't change, setting $m_i = n^2$), and choosing a_i and b_i randomly.

2. Check to make sure that the resulting h_i doesn't cause any collisions within the secondary table.

Hash of hashes

$K = \{10, 22, 37, 40, 52, 60, 70, 72, 75\}$

Outer hash $h(k) = ((ak + b) \% p) \% m$

$a = 3$, $b = 42$, $p = 101$, and $m = 9$
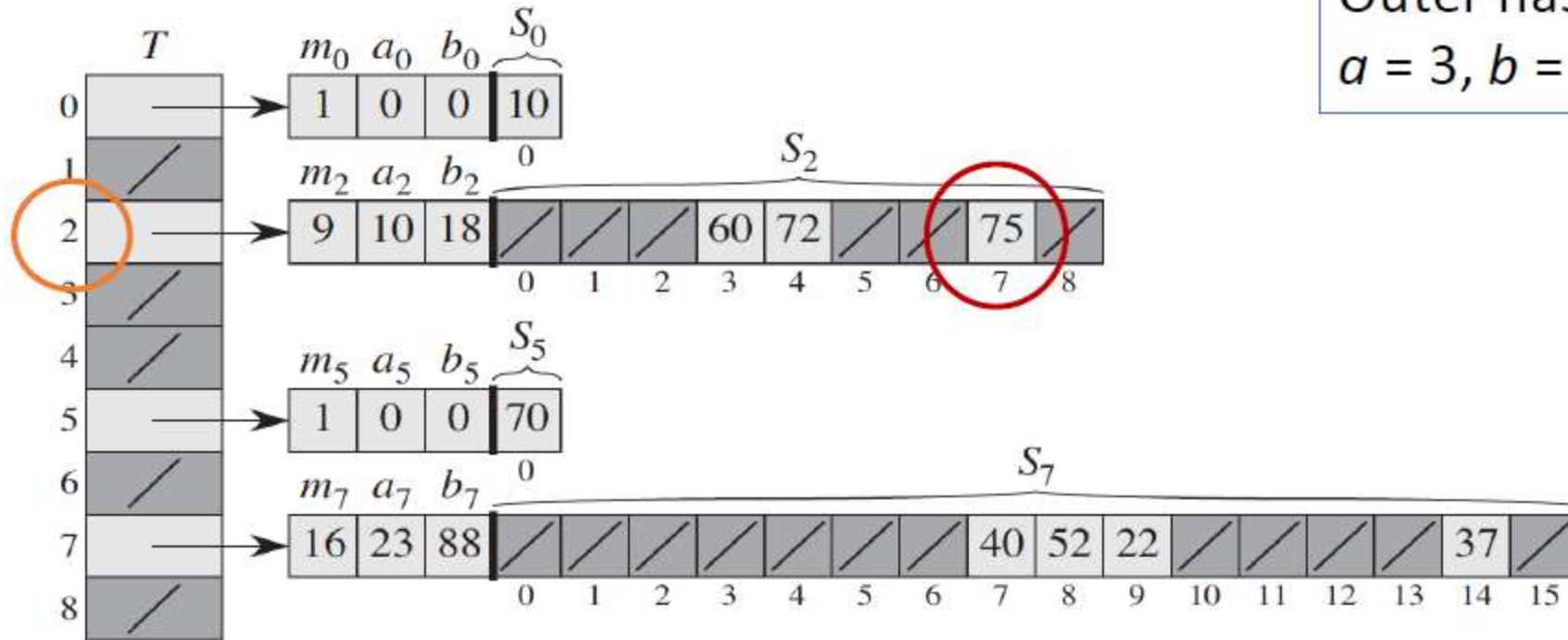
Example: $h(75) = 2$, and so key 75 hashes to slot 2 of table T. A secondary hash table S_j stores all keys hashing to slot j. The size of hash table S_j is $m_j = n_j^2$, and the associated hash function is $h_j(k) = \left((a_j k + b_j) \% p \right) \% m_j$

Since $h_2(75) = 7$, key 75 is stored in slot 7 of secondary hash table S_2.

Perfect hashing example

K = {8, 22, 36, 75, 61, 13, 84, 58}

$$h(k) = ((ak + b)\%p)\% m$$
$$h_j(k) = \left((a_jk + b_j)\%p\right)\%m$$

0	
1	
2	
3	
4	
5	
6	
7	

Step 1: Randomly generate values for a and b, select $p > K$

Step 2: For each key k, work out home cell $h(k)$. Keep a count.

Step 3: Check if $\sum_{j=0}^{m-1} n_j^2 < 2n$

Step 4: Populate sub-tables, calculating new hash functions

Perfect hashing example

K = {8, 22, 36, 75, 61, 13, 84, 58}

p = 87, a = 64, b = 5

$$h(k) = ((ak + b)\%p)\% m$$
$$h_j(k) = ((a_jk + b_j)\%p)\%m$$

0	
1	
2	
3	
4	
5	
6	
7	

<u>Step 1</u>: Randomly generate values for a and b, select $p > K$

<u>Step 2</u>: For each key k, work out home cell $h(k)$. Keep a count.

<u>Step 3</u>: Check if $\sum_{j=0}^{m-1} n_j^2 < 2n$

<u>Step 4</u>: Populate sub-tables, calculating new hash functions

Perfect hashing example

$$h(k) = ((ak + b)\%p)\% m$$
$$h_j(k) = ((a_j k + b_j)\%p)\% m$$

K = {8, 22, 36, 75, 61, 13, 84, 58}

p = 87, a = 64, b = 5

0	
1	
2	
3	
4	
5	
6	
7	

Step 1: Randomly generate values for *a* and *b*, select *p* > *K*

Step 2: For each key *k*, work out home cell *h*(*k*). Keep a count.

Step 3: Check if $\sum_{j=0}^{m-1} n_j^2 < 2n$

Step 4: Populate sub-tables, calculating new hash functions

Perfect hashing example

K = {8, 22, 36, 75, 61, 13, 84, 58}

p = 87, a = 64, b = 5

$$h(k) = ((ak + b)\%p)\% \, m$$
$$h_j(k) = ((a_jk + b_j)\%p)\%m$$

Step 1: Randomly generate values for a and b, select p > K

Step 2: For each key k, work out home cell h(k). Keep a count.

Step 3: Check if $\sum_{j=0}^{m-1} n_j^2 < 2n$

Step 4: Populate sub-tables, calculating new hash functions

0	/	0
1	/	61
2	/	8, 84
3	/	0
4	/	75
5	/	22
6	/	13
7	/	36, 58

Perfect hashing example

K = {8, 22, 36, 75, 61, 13, 84, 58}

p = 87, a = 64, b = 5

$$h(k) = ((ak + b)\%p)\% m$$
$$h_j(k) = \left((a_jk + b_j)\%p\right)\%m$$

			Count
0	/	0	0
1	/	61	1
2	/	8, 84	2
3	/	0	0
4	/	75	1
5	/	22	1
6	/	13	1
7	/	36, 58	2

Step 1: Randomly generate values for *a* and *b*, select *p > K*

Step 2: For each key *k*, work out home cell *h(k)*. Keep a count.

Step 3: Check if $\sum_{j=0}^{m-1} n_j^2 < 2n$

Step 4: Populate sub-tables, calculating new hash functions

Perfect hashing example

K = {8, 22, 36, 75, 61, 13, 84, 58}

p = 87, a = 64, b = 5

$$h(k) = ((ak + b)\%p)\% m$$
$$h_j(k) = ((a_j k + b_j)\%p)\%m$$

			Count
0	/	0	0
1	/	61	1
2	/	8, 84	2
3	/	0	0
4	/	75	1
5	/	22	1
6	/	13	1
7	/	36, 58	2

Step 1: Randomly generate values for *a* and *b*, select *p* > *K*

Step 2: For each key *k*, work out home cell *h(k)*. Keep a count.

Step 3: Check if $\sum_{j=0}^{m-1} n_j^2 < 2n$

Step 4: Populate sub-tables, calculating new hash functions

$$\sum_{j=0}^{m-1} n_j^2 = 0 + 1 + 4 + 0 + 1 + 1 + 1 + 4 = 12$$

$$12 < 2 * n = 16 \checkmark$$

Perfect hashing example

$K = \{8, 22, 36, 75, 61, 13, 84, 58\}$

$p = 87$, $a = 64$, $b = 5$

$$h(k) = \big((ak + b)\%p\big)\% m$$
$$h_j(k) = \big((a_jk + b_j)\%p\big)\%m$$

Step 4: Populate sub-tables, calculating new hash functions

Perfect hashing example

$$h(k) = \big((ak + b)\%p\big)\% \, m$$
$$h_j(k) = \big((a_j k + b_j)\%p\big)\%m$$

K = {8, 22, 36, 75, 61, 13, 84, 58}

p = 87, a = 64, b = 5

Step 4: Populate sub-tables, calculating new hash functions

Perfect hashing example

$$h(k) = ((ak + b)\%p)\% m$$
$$h_j(k) = ((a_jk + b_j)\%p)\%m$$

K = {8, 22, 36, 75, 61, 13, 84, 58}

p = 87, a = 64, b = 5

	m_j	a_j	b_j				
0	/						
1 →	1	0	0	61			
2 →	4	82	53	84	8	/	/
3	/						
4 →	1	0	0	75			
5 →	1	0	0	22			
6 →	1	0	0	13			
7 →	4	10	54	/	58	36	/

No S_j keys hash to the same sub-slots. **All done!**

Cuckoo Hashing

- Cuckoo hashing is similar to double hashing and perfect hashing. Cuckoo hashing is inspired by the Cuckoo bird, which lays its eggs in other birds' nests, bumping out the eggs that are originally there.

- Cuckoo hashing solves the dynamic dictionary problem, achieving O(1) worst-case time for queries and deletes, and O(1) expected time for inserts.

Cuckoo Hashing

- As usual, f and g map to a table T with m rows. But now, we will state that f and g hash to two separate hash tables.

- So T1[f(x)] and T2[g(x)] refer to hash entries in two adjacent hash tables.

- The cuckoo part of Cuckoo hashing thus refers to bumping out a keys of one table in the event of collision, and hashing them into the other table, repeatedly until the collision is resolved.

Cuckoo Hashing

- We implement the functions as follows:
- **Query(x) – Check T1[f(x)] and T2[g(x)] for x.**
- **Delete(x) – Query x and delete if found.**
- **Insert(x) –**
  - If T1[f(x)] is empty, we put x in T1[f(x)] and are done.
  - Otherwise say y is originally in T1[f(x)]. We put x in T1[f(x)] as before, and bump y to whichever of T1[f(y)] and T2[g(y)] it didn't just get bumped from.
  - If that new location is empty, we are done. Otherwise, we place y there anyway and repeat the process, moving the newly bumped element z to whichever of T1[f(z)] and T2[g(z)] doesn't now contain y.
  - We continue in this manner until we're either done or reach a hard cap of bumping 6 log n elements. Once we've bumped 6 log n elements we pick a new pair of hash functions f and g and rehash every element in the table.

Cuckoo Hashing

- Note that at all times we maintain the invariant that each element x is either at T1[f(x)] or T2[g(x)], which makes it easy to show correctness.

- It is clear that query and delete are O(1) operations.

- The time analysis of insertion is harder. The reason Insert(x) is not horribly slow is that the number of items that get bumped is generally very small, and we rehash the entire table very rarely when m is large enough. We take m = 4n.

Cuckoo Hashing

Table 1	
0	B
1	C
2	
3	E
4	

Table 2	
0	D
1	
2	A
3	
4	F

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4

Potential cuckoo hash table.

Cuckoo Hashing

- How to build hash tables?
- Lets see

Hopscotch hashing

- **Hopscotch hashing** tries to improve on the classic linear probing algorithm.

- In linear probing, cells are tried in sequential order, starting from the hash location. Because of clustering, this sequence can be long on average as the table gets loaded, and thus many improvements such as quadratic probing, double hashing, and so forth, have been proposed to reduce the number of collisions.

- However, on some modern architectures, the locality produced by probing adjacent cells is a more significant factor than the extra probes, and linear probing can still be practical or even a best choice.

Hopscotch hashing

- The idea of hopscotch hashing is to bound the maximal length of the probe sequence by a predetermined constant that is optimized to the underlying computer's architecture.

- Doing so would give constant-time lookups in the worst case, and like cuckoo hashing, the lookup could be parallelized to simultaneously check the bounded set of possible locations.

- If an insertion would place a new item too far from its hash location, then we efficiently go backward toward the hash location, evicting potential items.

- If we are careful, the evictions can be done quickly and guarantee that those evicted are not placed too far from their hash locations.

Hopscotch hashing

- The algorithm is deterministic in that given a hash function, either the items can be evicted or they can't. The latter case implies that the table is likely too crowded, and a rehash is in order; but this would happen only at extremely high load factors, exceeding 0.9.

- For a table with a load factor of 1/2, the failure probability is almost zero

- Let *MAX_DIST* be the chosen bound on the maximum probe sequence. This means that item *x* must be found somewhere in the *MAX_DIST* positions listed in *hash*(*x*), *hash*(*x*) + 1, . . . , *hash*(*x*) + (*MAX_DIST* − 1).

- In order to efficiently process evictions, we maintain information that tells for each position *x*, whether the item in the alternate position is occupied by an element that hashes to position *x*.

Hopscotch hashing

Insertion of entry x in a hash table using hopscotch hashing with neighborhoods of size H = 4

Occupied bucket. Z is the entry and 6 is the base bucket for that entry, i.e. the initial bucket to which the entry was hashed: mod(hash(z), size) = 6. Z is in the neighborhood of bucket 6.

Empty bucket

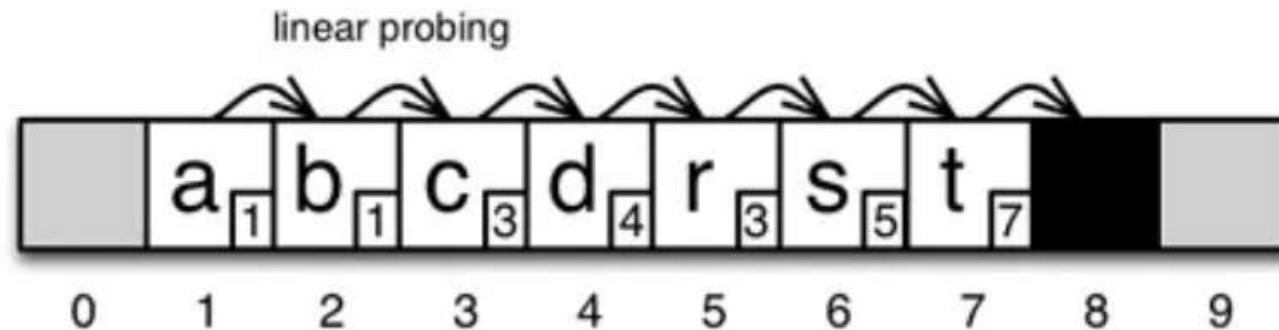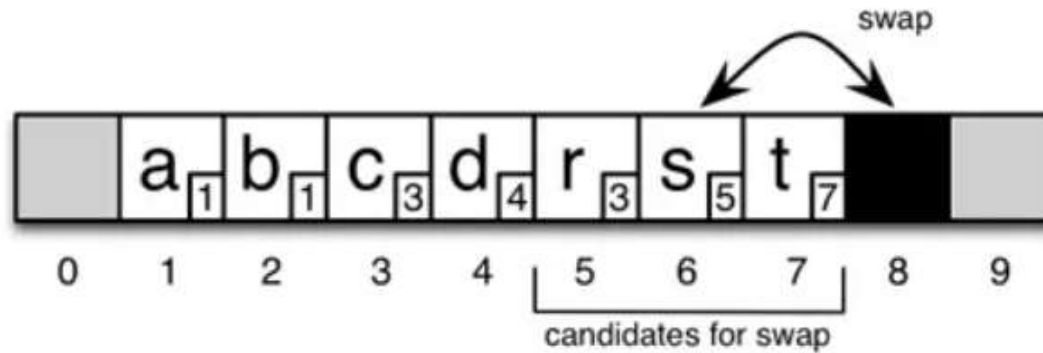Empty bucket found by the linear probing

## Step 1: Linear probing from the initial bucket to find an empty bucket

linear probing

- The initial bucket is bucket 1, because mod(hash(x), size) = 1
- Starting from bucket 1, linear probing finds an empty bucket in position 8, but it is beyond the neighborhood of initial bucket in position 1
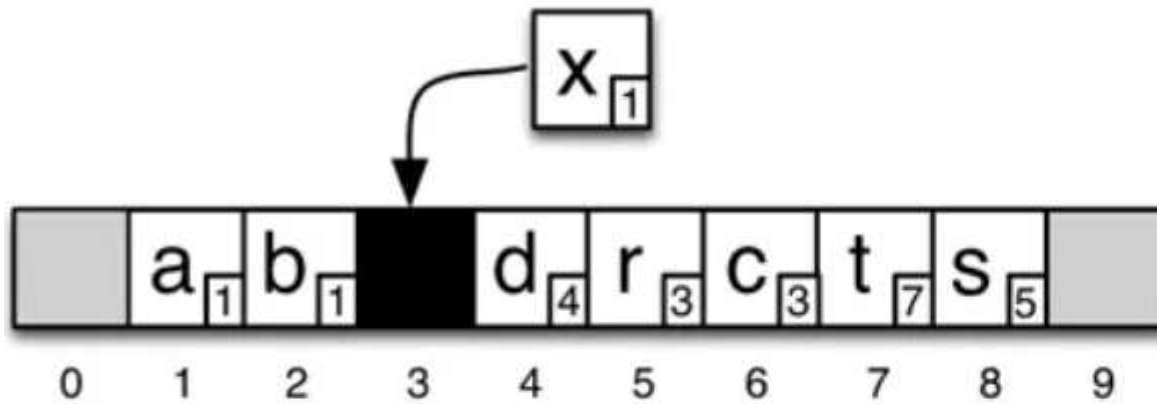
Step 2(a): The empty bucket is moved

- The empty bucket needs to be moved
- Swap candidates are buckets 5, 6, and 7, because 8 - (H-1) = 5
- In bucket 5 there is r, but it cannot be swapped because its base bucket, which is bucket 3, is too far from bucket 8.
- Then bucket 6 is tried, and the base bucket of s being 5, it can be swapped with 8.

Step 2(b): The empty bucket is moved again
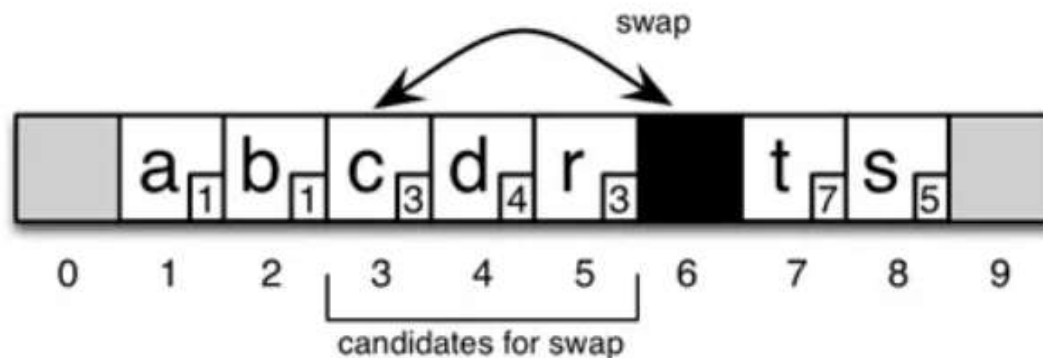
- The empty bucket is now in position 6
- It is still too far from bucket 1 and has to be moved again.
- Swap candidates are now bucket 3, 4 and 5 because 6 - (H-1) = 3
- Bucket 3 can be swapped because its base bucket is 3.
- The bucket will be moved again until it reaches a stable position, or until it cannot be moved, in which case the table needs to be resized.

Step 3: The new entry is inserted

- The empty bucket is now in position 3
- It is now in the neighborhood of the initial bucket, bucket 1, and thus does not need to be moved.
- The new entry can be inserted in bucket 3

Step 4: Final state after displacements and insertion
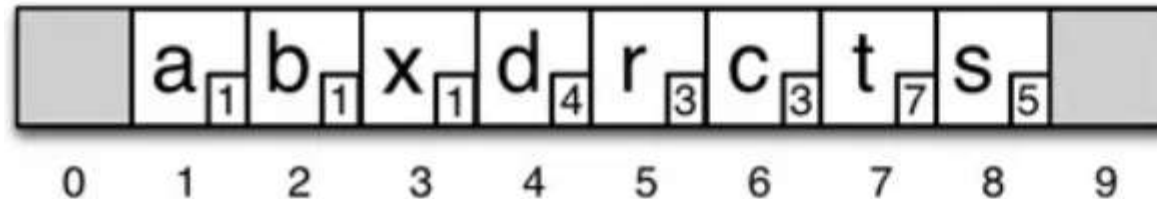
- The new entry x was inserted in the empty bucket, in bucket 3

Hopscotch hashing - Search

- The first step to retrieve an entry is to determine its initial bucket. Then, all what has to be done is to start from the position of this initial bucket and to scan through the next (H-1) buckets, and for each bucket, to compare the key with the key of the entry being searched.

- If the key does not match any of the keys for the entries in the neighborhood of the initial bucket, then the entry is simply not in the table.

- O(1)

Hopscotch hashing - Delete

- Replace the entry to delete by an empty entry, which can be null.
- This is a major improvement compared to basic open addressing that uses only probing.
- In that that case, entries would be removed by *lazy deletion*, which require the introduction of special values to mark deleted buckets. O(1)
- That are unused but yet must be scanned through during look-ups, a phenomenon called *contamination*.

Bit array representation

- Neighborhoods can be stored using bit arrays.
- Each bit in that bit array indicates if the current bucket or one of its following (H-1) buckets are holding an entry of the current bucket.

Step 1: Linear probing from the initial bucket to find an empty bucket
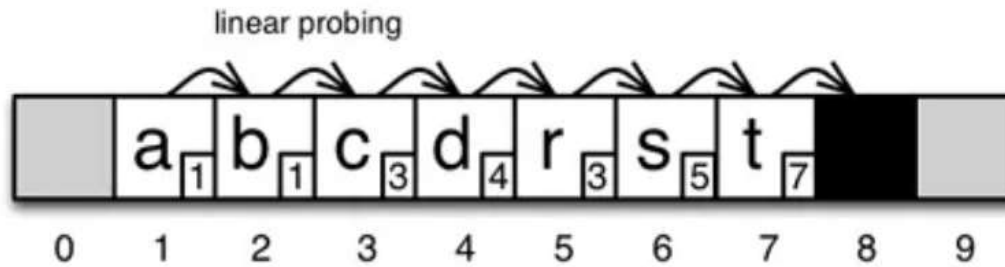
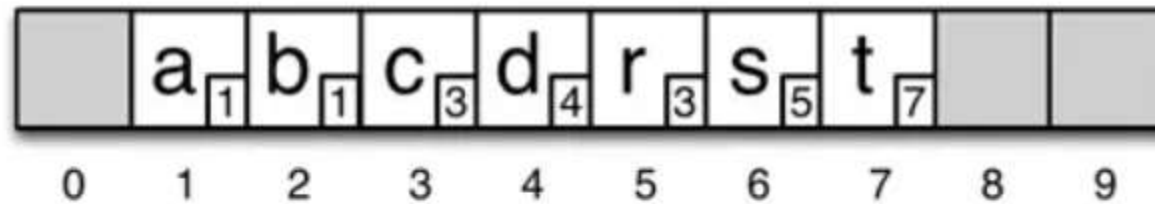- The initial bucket is bucket 1, because mod(hash(x), size) = 1
- Starting from bucket 1, linear probing finds an empty bucket in position 8, but it is beyond the neighborhood of initial bucket in position 1

- The bitmap for bucket 5 is therefore 0100

Bit array representation

(H = 4)

Bucket array

	a	b	c	d	r	s	t		
	a_1	b_1	c_3	d_4	r_3	s_5	t_7		

0   1   2   3   4   5   6   7   8   9

Bit array

0: 0000
1: 1100
2: 0000
3: 1010
4: 1000
5: 0100
6: 0000
7: 1000
8: 0000
9: 0000

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

→

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

→

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

A: 7

B: 9

C: 6

D: 7

E: 8

F: 12

G: 11

H: 9

Figure 5.47 Hopscotch hashing table. Attempting to insert H. Linear probing suggests location 13, but that is too far, so we evict G from position 11 to find a closer position.

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

→

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

→

	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
...		

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

Figure 5.48 Hopscotch hashing table. Attempting to insert I. Linear probing suggests location 14, but that is too far; consulting Hop[11], we see that G can move down, leaving position 13 open. Consulting Hop[10] gives no suggestions. Hop[11] does not help either (why?), so Hop[12] suggests moving F.

	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
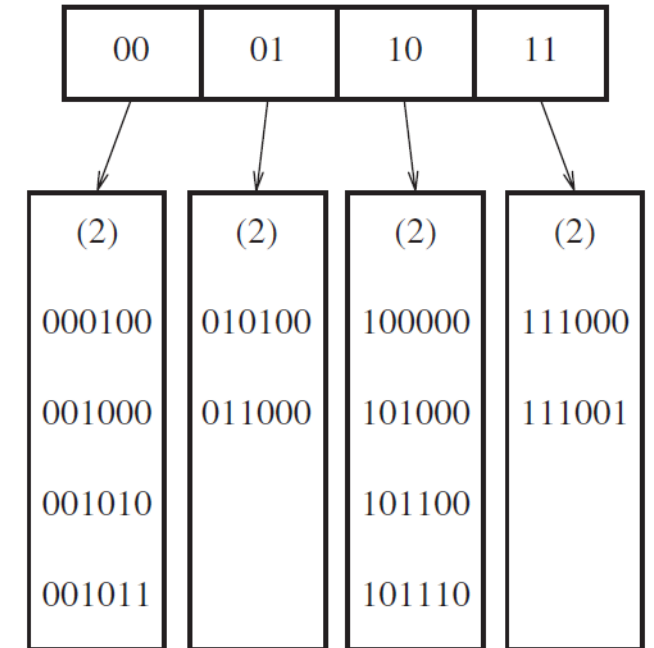E: 8
F: 12
G: 11
H: 9
I: 6

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.

Extendible Hashing

- What if the amount of data is too large to fit in main memory?

- The main consideration then is the number of disk accesses required to retrieve data.

- It is a flexible method in which the hash function also experiences dynamic changes.

- 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.

Extendible Hashing

- Keys are placed into buckets which are independent parts of a file in disk

- Keys having a hashing adress with same prefix share the same bucket

- The directories store addresses of the buckets in pointers. An id is assigned to each directory which may change each time when Directory Expansion takes place.
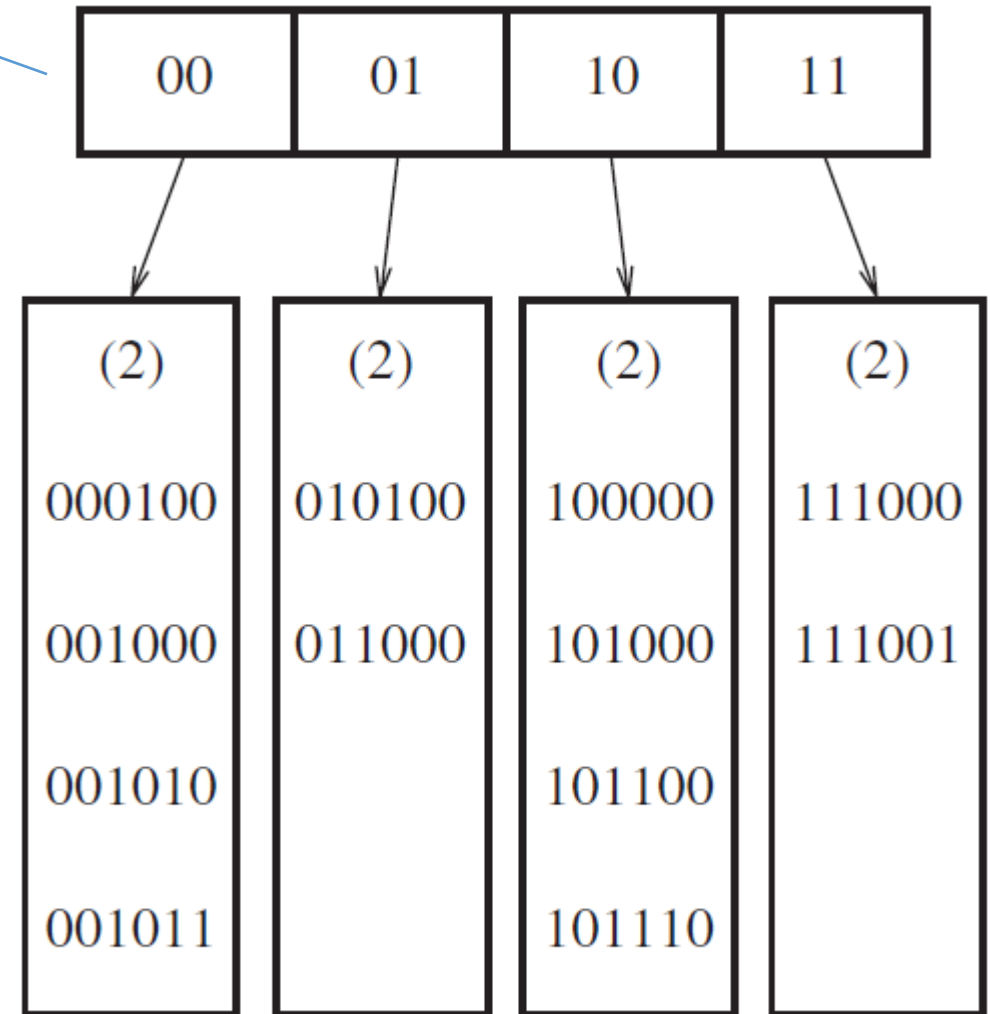
00	01	10	11

(2)	(2)	(2)	(2)
000100	010100	100000	111000
001000	011000	101000	111001
001010		101100	
001011		101110	

Extendible hashing: original data

Extendible Hashing

- Let us suppose that our data consists of several 6-bit integers.

- 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 $M = 4$.

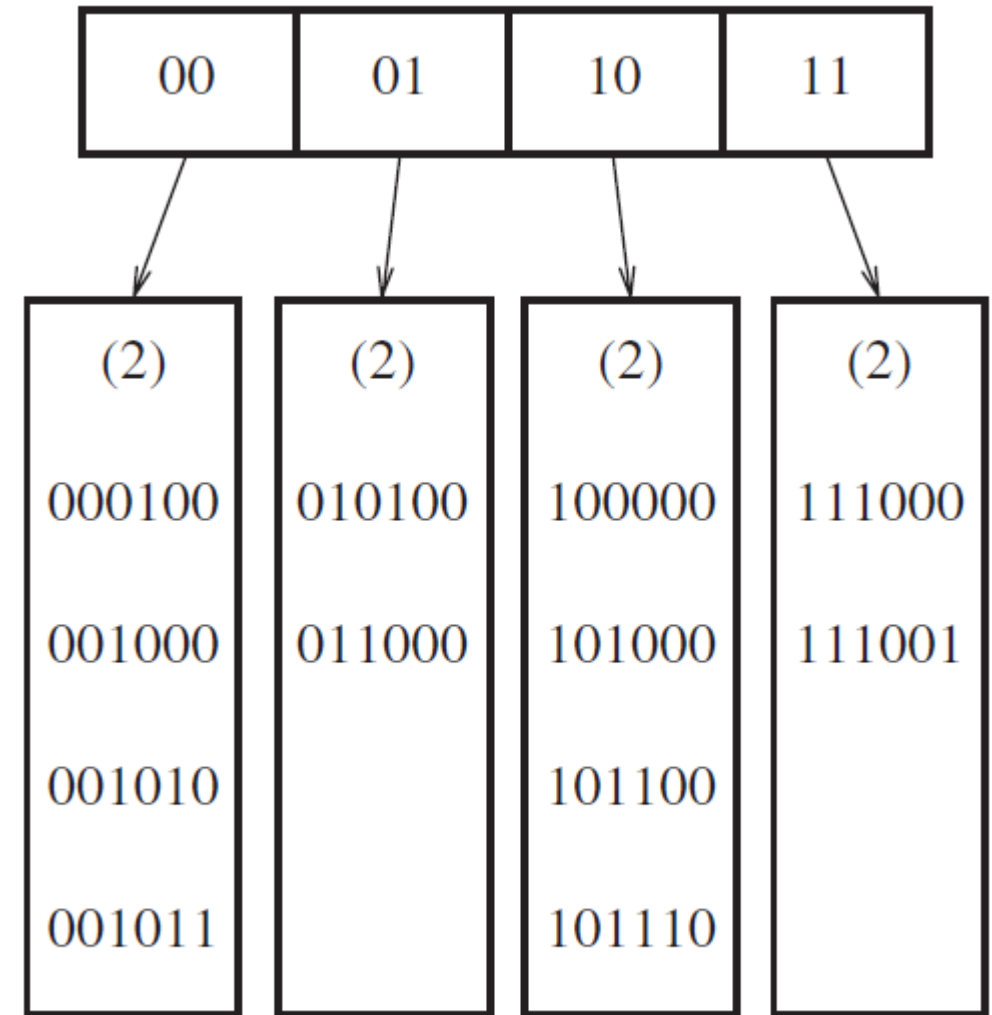**Bucket**

| 00 | 01 | 10 | 11 |

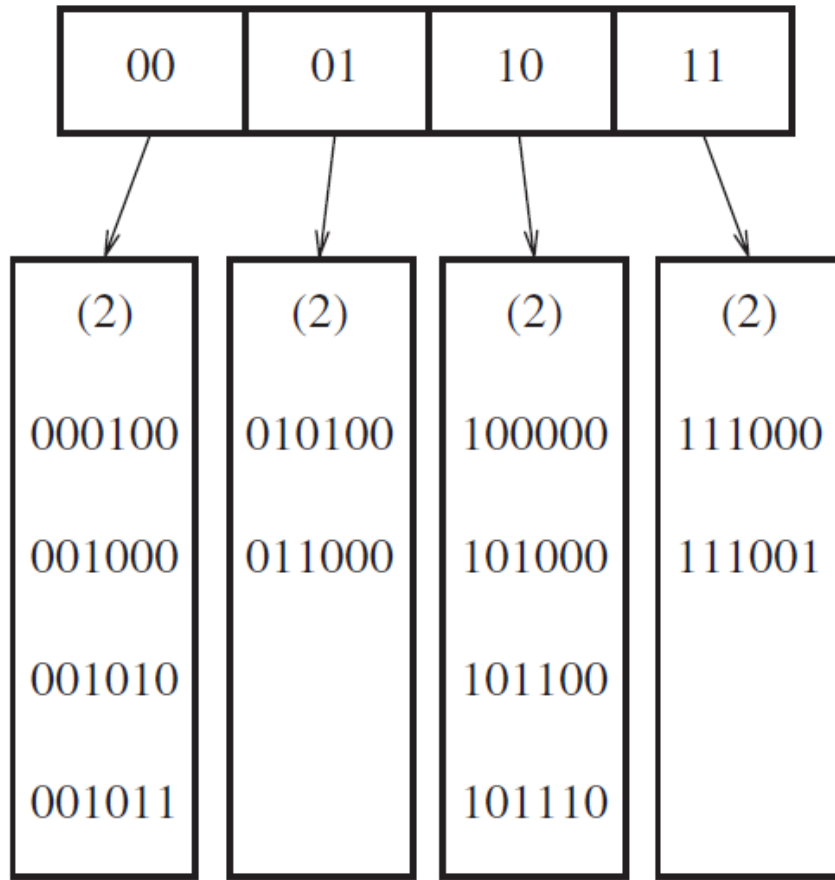| (2)	(2)	(2)	(2)
000100	010100	100000	111000
001000	011000	101000	111001
001010		101100	
001011		101110	

Extendible hashing: original data

Extendible Hashing

- The root of the "tree" contains four pointers determined by the leading two bits of the data. Each leaf has up to *M* = 4 elements. It happens that in each leaf the first two bits are identical

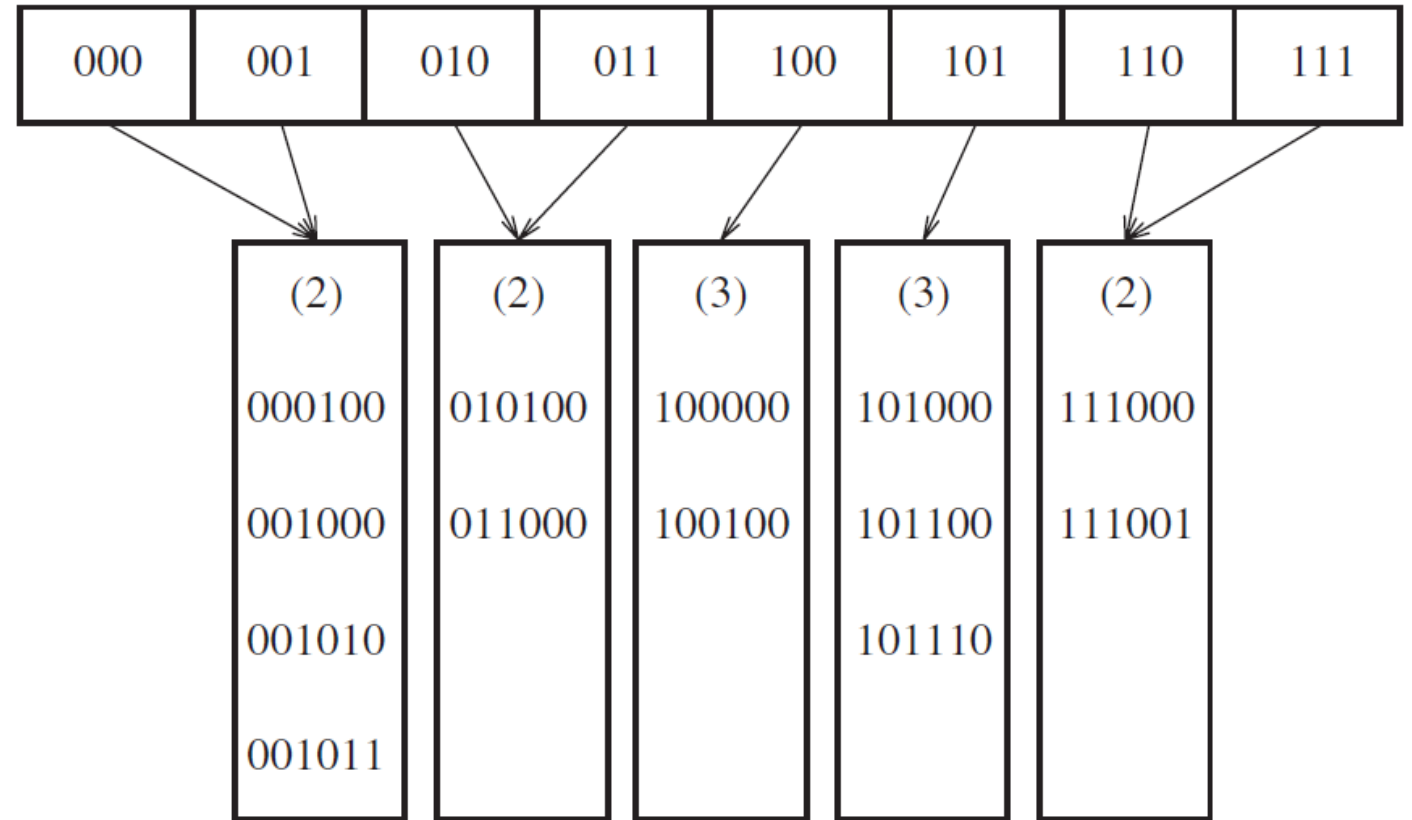- This is indicated by the number in parentheses.

00	01	10	11

(2)	(2)	(2)	(2)
000100	010100	100000	111000
001000	011000	101000	111001
001010		101100	
001011		101110	

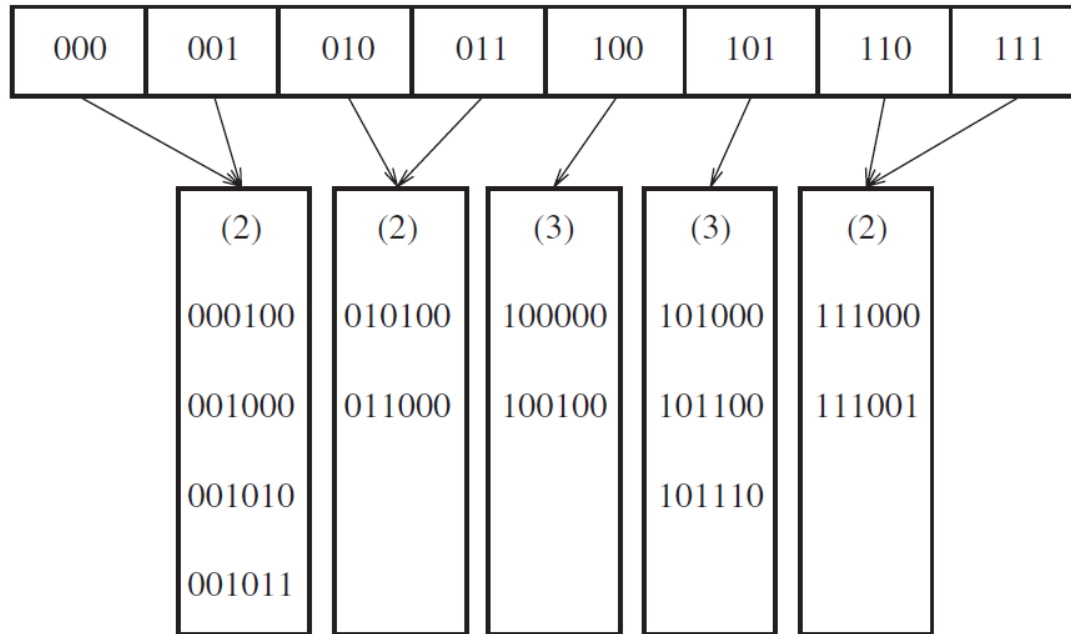Extendible hashing: original data

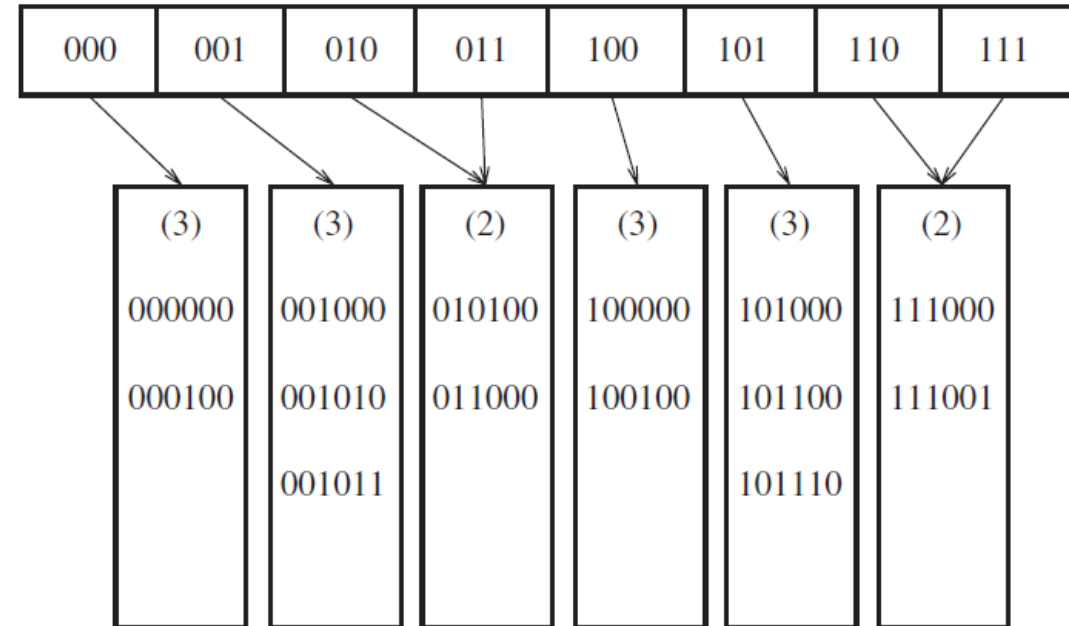# Extendible Hashing

Extendible hashing: original data

Extendible hashing: after insertion of 100100 and directory split

Extendible Hashing

Extendible hashing: after insertion of 100100 and directory split

Extendible hashing: after insertion of 000000 and leaf split