Hashing

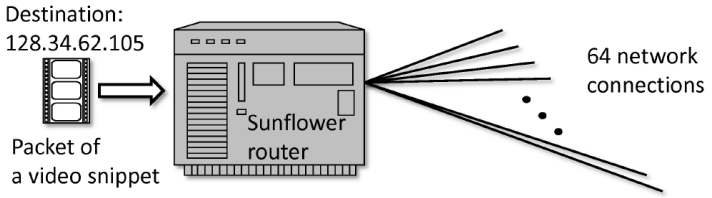
Hash Tables

What is Hashing & Why?

- Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE.
- A hash table is an effective data structure for implementing dictionaries. Although searching for an element in a hash table can take as long as searching for an element in a linked list O(n) time in the worst case (almost pathological) – in practice, hashing performs extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is O(1).
- ***A hash table generalizes the simpler notion of an ordinary array***. Directly addressing into an ordinary array makes effective use of our ability to examine an arbitrary position in an array in O(1) time.
- ***A hash function is any function that can be used to map data of arbitrary size to data of fixed size***. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.
- When the number of keys actually stored is small relative to the total number of possible keys, hash tables become an effective alternative to directly addressing an array, since a hash table typically uses an array of size proportional to the number of keys actually stored. ***Instead of using the key as an array index directly, the array index is computed from the key.***

Application: Network Routers

- Network routers process multiple streams of information packets at high speed.
- To process a packet, (k,x), where k is a key for the destination and x is the data payload, a router must very quickly determine which of its network connections to send this packet.
- Such a system needs to support key-based lookups, i.e., get(k) operations, as well as put(k,c) updates to add a new connection, c, for a destination key, k.
- Ideally, we would like to achieve O(1) time performance for both get and put operations.



Maps & Operations

- **Maps:**
  - A map models a searchable collection of key-value entries
  - The main operations of a map are for searching, inserting, and deleting items
  - Multiple entries with the same key are not allowed
  - Other Applications:
    - address book
    - student-record database
- **Operations:**
  - ***get(k)***: if the map M has an entry with key k, return its associated value; else, return null
  - ***put(k, v):*** insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
  - ***remove(k***): if the map M has an entry with key k, rer associated value; else, return null
  - ***size(), isEmpty()***

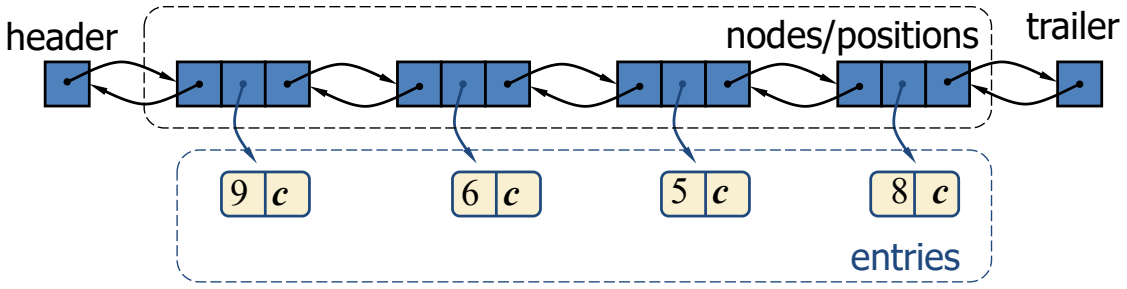


Example

Operation	*Output*	*Map*
isEmpty()	**true**	Ø
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*),(7,*B*)
put(2,*C*)	**null**	(5,*A*),(7,*B*),(2,*C*)
put(8,*D*)	**null**	(5,*A*),(7,*B*),(2,*C*),(8,*D*)
put(2,*E*)	*C*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(7)	*B*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(4)	**null**	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(2)	*E*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
size()	4	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
remove(5)	*A*	(7,*B*),(2,*E*),(8,*D*)
remove(2)	*E*	(7,*B*),(8,*D*)
get(2)	**null**	(7,*B*),(8,*D*)
isEmpty()	**false**	(7,*B*),(8,*D*)

A Simple List-Based Map

- We can implement a map using an unsorted list
  - We store the items of the map in a list S (based on a doubly-linked list), in arbitrary order



Performance of a List-Based Map

- Performance:
  - put takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - get and remove take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The unsorted list implementation is effective only for maps of small size or for maps in which puts are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

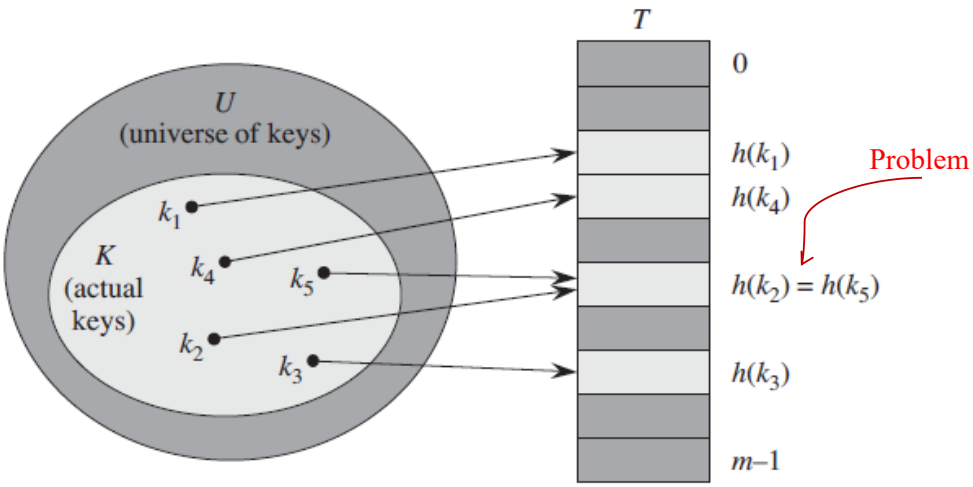
Hash Tables

- Assume that, universe of the keys is very large, say the set of all possible integers (−MAX, …, MAX). We can have a huge array, initialized with some kind of a sentinel (to indicate empty) and then we can use any key as the array index – insert, delete, find is constant time – sometimes known as Direct Addressing. Disadvantages: (a) the set K of keys actually stored may be so small relative to U that most of the space allocated for T would be wasted; (b) storage inefficiency; (c) problematic for storing duplicate keys.
- When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table requires much less storage than a direct address table. With direct addressing, an element with key k is stored in slot k. With hashing, this element is stored in slot h(k); that is, we use a **hash function** h to compute the slot from the key k. Here, h maps the universe U of keys into the slots of a hash table T[0 … m-1]:

  $h: U \rightarrow (0, 1, 2, \ldots, m - 1)$

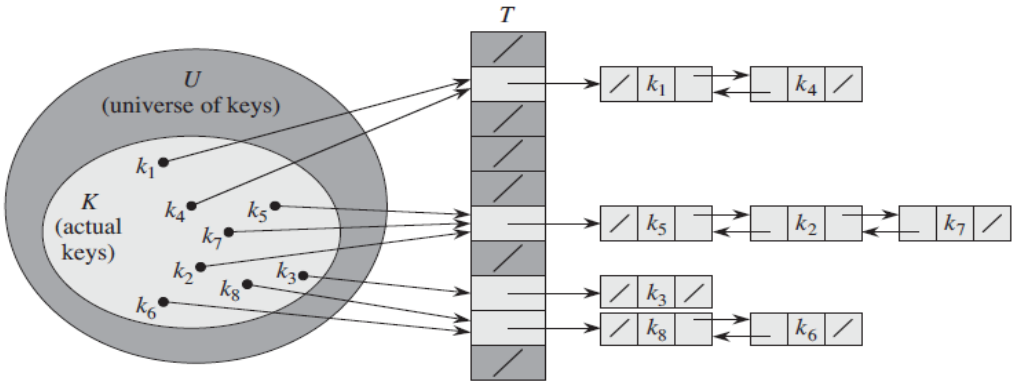
  where the size m of the hash table is typically much less than |U|. We say that an element with **key k hashes to slot h(k);** we also say that **h(k) is the hash value of key k**.
- The hash function reduces the range of array indices and hence the size of the array. Instead of a size of |U|, the array can have size m.

Hashing



- There is a problem when two keys may hash to the same slot. We call this situation a **collision**. Fortunately, we have effective techniques for resolving the conflict created by collisions.
- One idea is to make h appear to be "random," thus at least minimizing collisions. The very term "to hash," evoking images of random mixing and chopping, captures the spirit of this approach. Of course, a hash function h must be deterministic in that a given input k should always produce the same output h(k). But, avoiding collisions altogether is therefore impossible. we still need a method for resolving the collisions that do occur. One simple method is called **Chaining**.

Hashing with Chaining

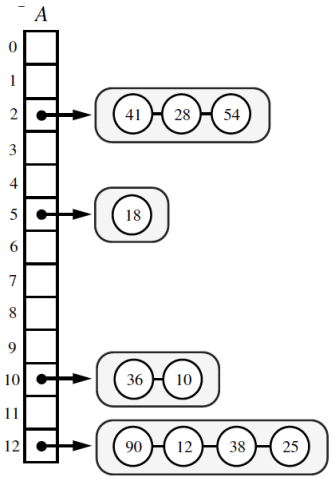


- In chaining, we place all the elements that hash to the same slot into the same linked list. Slot j contains a pointer to the head of the list of all stored elements that hash to j ; if there are no such elements, slot j contains NIL.
- Insertion, deletion, search are extremely easy. Interesting question is how long does it take to search for a key? Worst case answer is the length of the longest list – obviously.
- How probable is the worst case? The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average
- Given a hash table T with m slots that stores n elements, we define the ***load factor*** **α** for T as n/m, that is, the average number of elements stored in a chain. Assuming **simple uniform hashing** (a key is equally likely to be hashed into any slot) For j = 0, 1, …, m-1, if n_j T[j] denotes the length of T[j], then $n = n_0 + n_1 + \ldots + n_{m-1}$ or expected value of n_j is = **$\alpha = n/m$**.

10

Performance of Separate Chaining

- Let us assume that our hash function, h, maps keys to independent uniform random values in the range [0,N−1].
- Thus, if we let X be a random variable representing the number of items that map to a bucket, i, in the array A, then the expected value of X, E(X) = n/N, where n is the number of items in the map, since each of the N locations in A is equally likely for each item to be placed.
- This parameter, n/N, which is the ratio of the number of items in a hash table, n, and the capacity of the table, N, is called the **load factor** of the hash table.
- If it is O(1), then the above analysis says that the expected time for hash table operations is O(1) when collisions are handled with separate chaining.



Hash Functions

- A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to. We typically have no way to check this condition, since we rarely know the probability distribution from which the keys are drawn. Moreover, the keys might not be drawn independently.
- For example, if we know that the keys are random real numbers k independently and uniformly distributed in the range $0 \leq k < 1$, then the hash function $h(k) = \lfloor km \rfloor$ satisfies the condition of simple uniform hashing.
- A good approach derives the hash value in a way that we expect to be independent of any patterns that might exist in the data.
- For example, the "division method" computes the hash value as the remainder when the key is divided by a specified prime number. This method frequently gives good results, assuming that we choose a prime number that is unrelated to any patterns in the distribution of keys.
- We will consider two simple hash functions here. The best way to design a good hash function is "uniform hashing"; we may not discuss those in this course.
- Most hash functions assume that the universe of keys is the set $N = \{0, 1, 2, \ldots\}$ of natural numbers. Thus, if the keys are not natural numbers, we need a way to interpret them as natural numbers.

Interpreting keys as natural numbers

why 128?

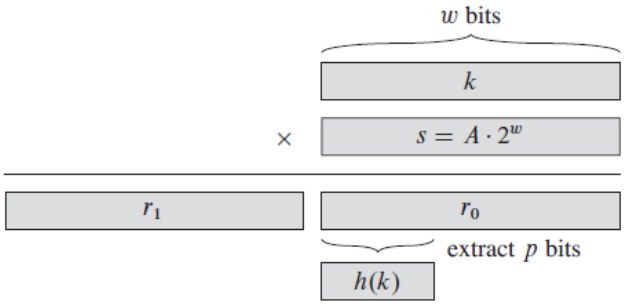
- Most hash functions assume that the universe of keys is the set N = {0, 1, 2, …} of natural numbers. Thus, if the keys are not natural numbers, we need a way to interpret them as natural numbers.
- For example, we can interpret a character string as an integer expressed in suitable radix notation. Thus, we might interpret the character string pt as the pair of decimal integers (112, 116 in the ASCII character set; then, expressed as a radix-128 integer, pt becomes $(112 \times 128) + 116 = 14452$. In the context of a given application, we can usually devise some such method for interpreting each key as a (possibly large) natural number. In our subsequent discussions , we assume that the keys are natural numbers.
- Simple Modulo Arithmetic and Integer GCD and LCM (we have seen in High School and again in 1010; refresh your memory; let us see a few examples; you should be able to write simple programs to compute those functions.

The 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 mod m. For example, if the hash table has size m = 12 and the key is k = 100, then h(k) = 4; k = 543 ⇒ h(k) = 3; k = 148 ⇒ h(k) = 4. Since it requires only a single division operation, hashing by division is quite fast. Did you see the collision?
- Efficiency of the division method depends on appropriate choice of m.
  - Avoid certain values of m. For example, m should not be a power of 2, since if $m = 2^p$, then h(k) is always the p lowest-order bits of k. Such values of m cannot in general distribute the keys appropriately. Unless we know that all low-order p-bit patterns are equally likely, we are better off designing the hash function to depend on all the bits of the key. Can you evaluate if $m = 2^p - 1$ is a good choice? Why or why not?
  - A prime not too close to an exact power of 2 is often a good choice for m. 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 mod 701.

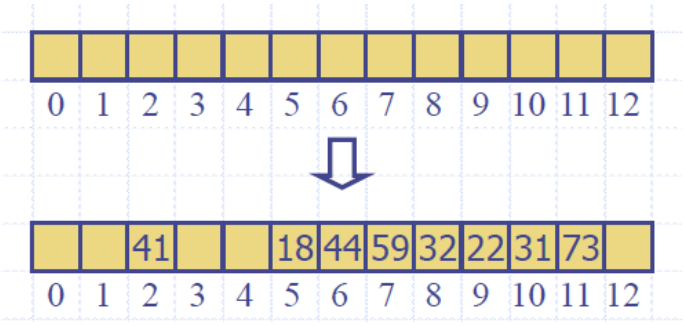
The Multiplication method

- The ***multiplication method*** for creating hash functions operates in two steps. First, we multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA. Then, we multiply this value by m and take the floor of the result. In short, the hash function is $h(k) = \lfloor m\ (kA \bmod 1) \rfloor$, where "kA mod 1" means the fractional part of kA, that is, $kA - \lfloor kA \rfloor$.
- An advantage of the multiplication method is that the value of m is not critical. We typically choose it to be a power of 2 ($m = 2^p$ for some integer p), since we can then easily implement the function on most computers.
- Suppose, the word size of the machine is w bits and that k fits into a single word. We restrict A to be a fraction of the form $s/2^w$, where s is an integer in the range $0 < s < 2^w$.
  - We first multiply k by the w-bit integer $s = A \cdot 2^w$. The result is a 2w-bit value $r_1 2^w + r_0$, where r_1 is the high-order word of the product and r_0 is the low-order word of the product. The desired p-bit hash value consists of the p most significant bits of r_0.
  - While this method works for any value of A, it works better with some values than with others. The optimal choice depends on the characteristics of the data being hashed.
  - $A \approx (\sqrt{5} - 1)/2$ works very well



Linear Probing

- Open addressing: the colliding item is placed in a different cell of the table
- Linear probing: handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes
- Example: h(x) = x mod 13, Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Search with Linear Probing

- Consider a hash table A that uses linear probing
- **get(k)**
  - We start at cell h(k)
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed
- To handle insertions and deletions, we introduce a special object, called DEFUNCT, which replaces deleted elements.
- **remove(k)**
  - We search for an entry with key k
  - If such an entry, (k, v), is found, we move elements to fill the "hole" created by its removal.
- **put(k, v)**
  - We throw an exception if the table is full
  - We start at cell h(k)
  - We probe consecutive cells until a A cell i is found that is empty.
  - We store (k, v) in cell i

Pseudo-code for get and put

- get(k):

```
i ← h(k)
while A[i] ≠ NULL do
    if A[i].key = k then
        return A[i]
    i ← (i + 1) mod N
return NULL
```

- put(k, v):

```
i ← h(k)
while A[i] ≠ NULL do
    if A[i].key = k then
        A[i] ← (k, v)        // replace the old (k, v')
    i ← (i + 1) mod N
A[i] ← (k, v)
```

Pseudo-code for remove

- remove(k):

```
i ← h(k)
while A[i] ≠ NULL do
    if A[i].key = k then
        temp ← A[i]
        A[i] ← NULL
        Call Shift(i) to restore A to a stable state without k
        return temp
    i ← (i + 1) mod N
return NULL
```

- Shift(i):

```
s ← 1        // the current shift amount
while A[(i + s) mod N] ≠ NULL do
    j ← h(A[(i + s) mod N].key)        // preferred index for this item
    if j ∉ (i, i + s] mod N then
        A[i] ← A[(i + s) mod N]        // fill in the "hole"
        A[(i + s) mod N] ← NULL        // move the "hole"
        i ← (i + s) mod N
        s ← 1
    else
        s ← s + 1
```

Performance of Linear Probing

- In the worst case, searches, insertions and removals on a hash table take O(n) time.
- The worst case occurs when all the keys inserted into the map collide.
- The load factor $\alpha = n/N$ affects the performance of a hash table.
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is $1 / (1 - \alpha)$.
- The expected running time of all the dictionary ADT operations in a hash table is O(1) with constant load < 1. In practice, hashing is very fast provided the load factor is not close to 100%

A More Careful Analysis of Linear Probing[*]

- Recall that, in the linear-probing scheme for handling collisions, whenever an insertion at a cell i would cause a collision, then we instead insert the new item in the first cell of i+1, i+2, and so on, until we find an empty cell.

> Let $X_1, X_2, \ldots, X_n$ be a set of mutually independent indicator random variables, such that each X_i is 1 with some probability $p_i > 0$ and 0 otherwise. Let $X = \sum_{i=1}^{n} X_i$ be the sum of these random variables, and let μ denote the mean of X, that is, $\mu = E(X) = \sum_{i=1}^{n} p_i$.| The following bound, which is due to Chernoff (and which we derive in Section 19.5), establishes that, for $\delta > 0$,
>
> $$\Pr(X > (1+\delta)\mu) < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\mu}.$$

- For this analysis, let us assume that we are storing n items in a hash table of size N = 2n, that is, our hash table has a load factor of 1/2.

A More Careful Analysis of Linear Probing, 2

Let X denote a random variable equal to the number of probes that we would perform in doing a search or update operation in our hash table for some key, k. Furthermore, let X_i be a 0/1 indicator random variable that is 1 if and only if $i = h(k)$, and let Y_i be a random variable that is equal to the length of a run of contiguous nonempty cells that begins at position i, wrapping around the end of the table if necessary. By the way that linear probing works, and because we assume that our hash function $h(k)$ is random,

$$X = \sum_{i=0}^{N-1} X_i(Y_i + 1),$$

which implies that

$$\begin{aligned} E(X) &= \sum_{i=0}^{N-1} \frac{1}{2n} E(Y_i + 1) \\ &= 1 + (1/2n)E\left(\sum_{i=0}^{N-1} Y_i\right). \end{aligned}$$

- Thus, if we can bound the expected value of the sum of Y_i's, then we can bound the expected time for a search or update operation in a linear-probing hashing scheme.

A More Careful Analysis of Linear Probing, 3

Consider, then, a maximal contiguous sequence, S, of k nonempty table cells, that is, a contiguous group of occupied cells that has empty cells next to its opposite ends. Any search or update operation that lands in S will, in the worst case, march all the way to the end of S. That is, if a search lands in the first cell of S, it would make k wasted probes, if it lands in the second cell of S, it would make $k-1$ wasted probes, and so on. So the total cost of all the searches that land in S can be at most k^2. Thus, if we let $Z_{i,k}$ be a 0/1 indicator random variable for the existence of a maximal sequence of nonempty cells of length k, then

$$\sum_{i=0}^{N-1} Y_i \leq \sum_{i=0}^{N-1} \sum_{k=1}^{2n} k^2 Z_{i,k}.$$

Put another way, it is as if we are "charging" each maximal sequence of nonempty cells for all the searches that land in that sequence.

A More Careful Analysis of Linear Probing, 4

So, to bound the expected value of the sum of the Y_i's, we need to bound the probability that $Z_{i,k}$ is 1, which is something we can do using the Chernoff bound given above. Let Z_k denote the number of items that are mapped to a given sequence of k cells in our table. Then,

$$\Pr(Z_{i,k} = 1) \leq \Pr(Z_k \geq k).$$

Because the load factor of our table is $1/2$, $E(Z_k) = k/2$. Thus, by the above Chernoff bound,

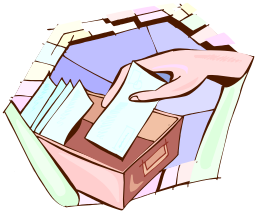
$$\begin{aligned} \Pr(Z_k \geq k) &= \Pr(Z_k \geq 2(k/2)) \\ &\leq (e/4)^{k/2} \\ &< 2^{-k/4}. \end{aligned}$$

Therefore, putting all the above pieces together,

$$\begin{aligned} E(X) &= 1 + (1/2n)E\left(\sum_{i=0}^{N-1} Y_i\right) \\ &\leq 1 + (1/2n)\sum_{i=0}^{N-1}\sum_{k=1}^{2n} k^2\, 2^{-k/4} \\ &\leq 1 + \sum_{k=1}^{\infty} k^2\, 2^{-k/4} \\ &= O(1). \end{aligned}$$

That is, the expected running time for doing a search or update operation with linear probing is $O(1)$, so long as the load factor in our hash table is at most $1/2$.

Double Hashing



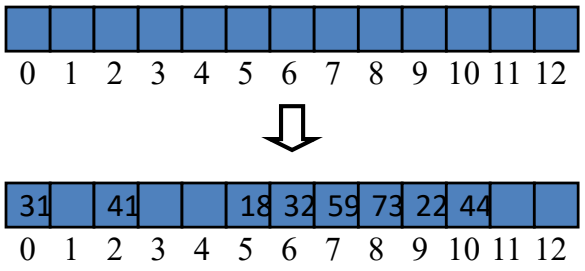
- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
  $$(i + jd(k)) \bmod N$$
  for $j = 0, 1, \ldots, N-1$
- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells

- Common choice of compression function for the secondary hash function:
  $$d_2(k) = q - k \bmod q$$
  where
  - $q < N$
  - q is a prime
- The possible values for $d_2(k)$ are
  $$1, 2, \ldots, q$$

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



26