Hashing

Hash Tables

and

Hash Functions

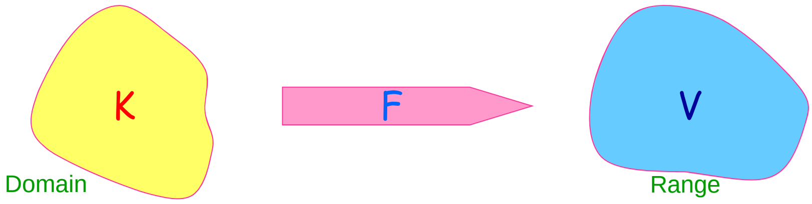
Introduction

The problem of retrieving an entry given its key, so far, took at least O(log n) by using some of the search methods discussed before.

Could there be a better way that will do the same thing with less time?

Hash Tables: Abstract Definition

- A Hash Table with addresses from the set K, of keys, associated with entry values from the set V, is a transformation function, F from K into V.



- Because a hash table is a function, we can write it as:

F : K → V

Hash Table Operations

1. Table Access (Retrieve): Evaluate the function at any key in set K.
2. Table Assignment (Update): Modify the function by changing its value at a specified key in K to a new value.
3. Insertion (Expand the Table): Add a new key , k, to the key set K, and define a corresponding value of the function at k.
4. Delete (Shrink the Table): Delete a key, k, from the key set K, and restrict the function to the resulting smaller domain.

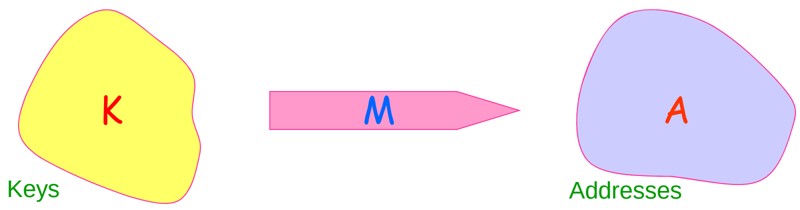
Hash Table Implementation

- To implement a hash table with a general set of keys K, and entry values V of any object type,

  Find an appropriate mapping, M from the set of keys K into memory addresses A, where entry values V of the hash table can be stored.

$M : K \rightarrow A$ ← Stores entry (k, v)

Hash Table Implementation (cont.)



$M : K \rightarrow A$

Hash Table Implementation: Arrays and Hash Function

A Hash Table is implemented with two major components:

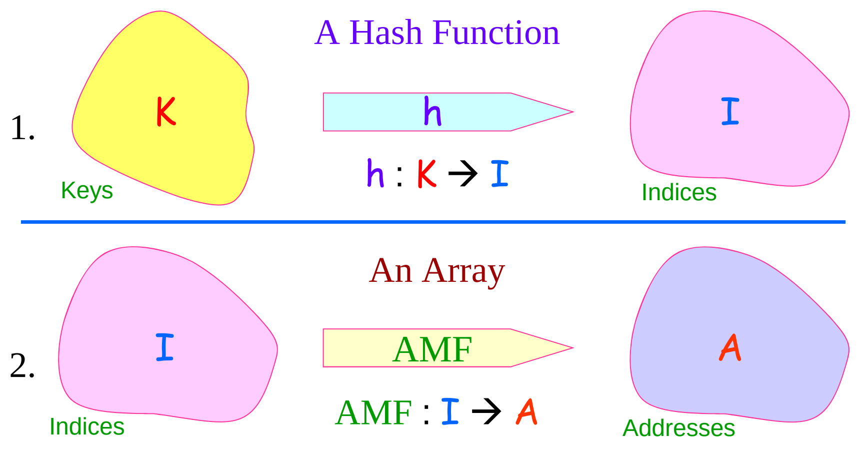
- An array A of size N that implements part of the table function. This array is used to map a set of integer array indices I, in the range $[0, N-1]$, into memory addresses A

$$\text{AMF} : I \rightarrow A$$

- A Hash Function h, is used to map from the set of keys K, into the array indices I.

$$h : K \rightarrow I$$

The Two Parts of a Hash Table Implementation



Thank Queue!

Example

The following hash table implementation has an array defined as:

E[ ] A = (E[ ]) new object[7];

where an entry E, has a key field and a value field: (k, v) pair ;

and a hash function that maps each key to an integer in the range [0, 6]:

h(key) = key % 7.

Insert the entries of the following keys into the hash table:
374, 972, 911, 740.

The idea is to store the entry (k, v) in the array at: A[h(k)].

Example (continued)

h(374) = 374 % 7 = 3,

The Array:

Index:	0	1	2	3	4	5	6
entry key:							

Example (continued)

$h(374) = 374 \% 7 = 3$,

The Array:

Index:	0	1	2	3	4	5	6
entry key:				374			

Example (continued)

$h(374) = 374 \% 7 = 3$,

$h(972) = 972 \% 7 = 6$,

The Array:

Index:	0	1	2	3	4	5	6
entry key:				374			

Example (continued)

$h(374) = 374 \% 7 = 3$,

$h(972) = 972 \% 7 = 6$,

The Array:

Index:	0	1	2	3	4	5	6
entry key:				374			972

Example (continued)

$h(374) = 374 \% 7 = 3$,

$h(972) = 972 \% 7 = 6$,

$h(911) = 911 \% 7 = 1$,

$h(740) = 740 \% 7 = 5$,

The Array:

Index:	0	1	2	3	4	5	6
entry key:		911		374		740	972

Back to Our Question

Could there be a way to retrieve an entry given its key with less time than O(log n)?

The answer is yes, use hash tables:

1. Plug the given key into the hash function to get the corresponding index of the array,
2. Go to the array at that index to get the entry from memory.

Performance: should be O(1) !!

But is this True?

- There are two components:
  - Evaluating the hash function.
  - Accessing the array.
- Almost all computer programming languages provide an efficient built-in array data type, which gives access to memory in O(1) time.
- What about the hash function?

The Hash Function

There are two problems associated with hash functions:

1. A hash function is normally a many-to-one transformation because the set of possible key values are normally much larger than the set of array indices.

   Example:

   Consider the Student ID Number in the university,
   it consists of 7 digits. There are 10^7 possible values.
   But we may be interested in a class of students in one
   course of, say 200 students. The set of indices then
   could be 250 which equals N, the size of the array.

2. The choice of a hash function is critical, since it could require long time to compute or cause other problems.

Collision

- Since a hash function h is a many-to-one mapping, then it is possible that more than one key value could be mapped to the same index value. When this happens, it is called a Collision.

  Example:

  In the previous example, insert a new key value = 227.

  $h(227) = 227 \% 7 = 3$.

  Collision: Place already occupied.

  The Array:

Index:	0	1	2	3	4	5	6
entry key:		911		374		740	972

- To solve this problem, some collision-resolution method should be employed in order to find an alternate index value.

The Perfect Hash Function

- The Perfect Hash Function is a function h that gives a one-to-one transformation.

- Two conditions are needed to find such a function:
  - The set of key values must be limited, and
  - All the key values must be known in advance.

- Example:

  The reserved words in a programming language compiler.

The Performance of Hashing

- The performance of hashing, then, depends on three factors:
  1. The choice of the hash function h,
  2. The choice of the collision-resolution method, and
  3. The load factor of the hash table.

- Definition:

  Let n be the number of entries in a hash table, and N be the size of the array. Then, the load factor λ of the hash table is given by the ratio:

$$\boxed{\lambda = \frac{n}{N} \quad , \quad n < N}$$

How to Choose a Hash Function

- A good hash function must have the following properties:
  1. It distributes the keys evenly over the range of index values.
  2. The distribution is random.
  3. It is easy and quick to compute.

To reduce the possibility of collisions

O(1)

Hash Function Composition

- The key k, of an entry can be of **any** object type, so the evaluation of the hash function h(k), is specified as the composition of two functions:
  1. Hash code function: **Maps** the key k, to an integer.

     h_1: keys $\rightarrow$ integers

  2. Compression function: **Restricts** the hash code to an integer within the range $[0, N-1]$ of the array indexes.

     h_2: integers $\rightarrow [0, N-1]$

- The hash code function is applied first on the key, and the compression function is applied next on the result, i.e.,

  $h(k) = h_2(h_1(k))$

Hash Codes in Java

- The goal of the hash code function is to “disperse” the keys in an apparently random way.
- So, the set of hash codes for the keys should have the following properties:
  1. Avoid collisions as much as possible by being very well spread apart.
  2. Be consistent, i.e. for a key k the hash code should be the same as the hash code for any key that is equal to k.
- In Java:
  The Object class has a default hashCode() method for mapping each object instance to a 32-bit integer of type int representing that object.
- You should override the default hashCode() method when certain objects are used for keys.

Example Hash Code Functions

1. Memory address:
   - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects).
   - Good in general, except for numeric and string keys
2. Integer cast:
   - We reinterpret the bits of the key as an integer.
   - Suitable for keys of bit length less than or equal to number of bits of integer type (e.g., byte, short, int, char and float in Java)
   - A float type variable x can be converted to integer by calling:

     Float.floatToIntBits(x)

Example Hash Code Functions

3. Component sum:
   - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) then sum the components (ignoring overflows).
   - Suitable for numeric keys of bit length greater than or equal to number of bits of integer type (e.g., long and double in Java)
   - A double type variable y can be converted to long by calling:

     Double.doubleToLongBits(y)
   - Example: Function to convert a long integer

```
static int hashCode(long i) {
    return (int)((i>>32)+(int) i);
}
```

Example Hash Code Functions

4. Polynomial accumulation:
   - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)

     $a_0\ a_1\ \ldots\ a_{n-1}$
   - Then we evaluate the polynomial:

     $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots\ \ldots + a_{n-1} z^{n-1}$$

     at a fixed value z, ignoring overflows.
   - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)
   - Polynomial $p(z)$ can be evaluated in $O(n)$ time

Example Compression Functions

1. Digit Selection (Truncation):

   The hash code is considered as a string of digits, and a number of these digits are selected so that an index within the range of the table's index set is formed.

   Example:

   Given a hash code of 9 digits = 123456789,
   and a table size = 1000.

   Then an index value in the required range has 3 digits:
   from 000 to 999.

   We can select any three digits from the hash code, like: 159

Example Compression Functions

2. Division:

   The integer hash code c of the key is divided by the table size N, and the remainder of this division is taken as the index value,

   $$h_2(c) = c \% N.$$

   For even distribution, N should be chosen as a prime number.

   Example:

   Given a hash code of 9 digits = 123456789,
   and a table size = 1003.

   Then, $h_2(123456789) = 123456789 \% 1003 = 528$.

Example Compression Functions

3. Multiplication:
   a. **Mid-Square:** The hash code is multiplied by itself, and selection is made from the middle digits of the result to get an index value.

Example:

Given a hash code of 9 digits = 123456789,
and a table size = 1000.

Then, hash code squared = 15241578750190521
Selecting the middle three digits gives index = 875

Example Compression Functions

3. Multiplication:
   b. **Fraction:** The hash code c is multiplied by a constant fraction, and selection is made from the first few digits of the fractional part of the result to get an index value.

Example:

Given a hash code of 9 digits = 123456789,
and a table size = 1000.

Then, let the fraction, f = 0.531013731
Then, (c * f) = (123456789 * 0.531013731)
= 65557250.144169759
Select first 3 digits of fraction part to get index = 144

Example Compression Functions

4. Folding:

This is a class of methods that involves partitioning the key into several parts and then combining the portions to form a smaller result, using the ADD or XOR operations. Other methods like truncation or division may be used to obtain an index value in the proper range.

Example:

Given a key of 9 digits = 123456789,
and a table size = 1003.

Then, partitioning to 3 digits results in: 123 456 789.
Then, add partitions to get: 1368.
Then, use division to get index = (1368 % 1003) = 365.

Example Compression Functions

5. Multiply, Add and Divide (MAD):

The integer hash code y is plugged into the following MAD function which eliminates repeated patterns in y :

$$h_2(y) = |ay + b| \% N$$

In Java: int h = Math.abs(a*y + b) % N;

Where: a and b are non-negative integer constants such that:

$$(a \% N) \neq 0$$

Otherwise, every integer would map to the same value b.

b can be zero, a and b are best if they were prime numbers.

Example: $h_2(123456789) = |13 * 123456789 + 17| \% 1003 = 863$

Hashing

Collision-resolution Methods

Open Address Methods

- Keys which hash to an occupied table location, are rehashed or placed into some other un-occupied (open) location.
- There are several methods to determine which open location to choose. These methods include:
  1. Linear Probing.
  2. Quadratic Probing.
  3. Double Hashing.
- We are going to look into each one of theses methods next.

Open Address Methods
1. Linear Probing

Let h(key) = h_0 = Home Address.

If there is a collision at h_0 then calculate another address using:

$$h_i = (h_0 + i) \% N, \qquad i = 1, 2, \ldots, N\text{-}1$$

until either the target key is located (retrieving) or an empty position is found (inserting).

Linear Probing Example

The following hash table implementation has an array defined as:

E[ ] A = (E[ ]) new object[7];

where an entry E, has a key field and a value field: (k, v) pair ;

and a hash function that maps each key to an integer in the range [0, 6]:

h(key) = key % 7.

Insert the entries of the following keys into the hash table:
374, 972, 911, 740, 227, 934, 362.

Use linear probing to resolve collisions.

Linear Probing Example (continued)

h(374) = 374 % 7 = 3.
h(972) = 972 % 7 = 6.
h(911) = 911 % 7 = 1.
h(740) = 740 % 7 = 5.

The array:

Index:	0	1	2	3	4	5	6
entry key:		911		374		740	972

Linear Probing Example (continued)

h(374) = 374 % 7 = 3.
h(972) = 972 % 7 = 6.
h(911) = 911 % 7 = 1.
h(740) = 740 % 7 = 5.
h(227) = 227 % 7 = 3; h_1 = 4.

Clustering

The array:

Index:	0	1	2	3	4	5	6
entry key:		911		374	227	740	972

Linear Probing Example (continued)

h(374) = 374 % 7 = 3.
h(972) = 972 % 7 = 6.
h(911) = 911 % 7 = 1.
h(740) = 740 % 7 = 5.
h(227) = 227 % 7 = 3; h_1 = 4.
h(934) = 934 % 7 = 3; h_1 = 4; h_2 = 5; h_3 = 6; h_4 = 0.

Primary Clustering

The array:

Index:	0	1	2	3	4	5	6
entry key:	934	911		374	227	740	972

Linear Probing Example (continued)

h(374) = 374 % 7 = 3.
h(972) = 972 % 7 = 6.
h(911) = 911 % 7 = 1.
h(740) = 740 % 7 = 5.
h(227) = 227 % 7 = 3; h_1 = 4.
h(934) = 934 % 7 = 3; h_1 = 4; h_2 = 5; h_3 = 6; h_4 = 0.
h(362) = 362 % 7 = 5; h_1 = 6; h_2 = 0; h_3 = 1; h_4 = 2.

Secondary Clustering

The array:

Index:	0	1	2	3	4	5	6
entry key:	934	911	362	374	227	740	972

Clustering

- **Clusters**: are long sequences of entries that are in the table with small gaps between them.
- Clustering has a bad effect on the efficiency of table operations.

Linear Probing Disadvantages

- As the table becomes half full there is a tendency towards clustering.
- Primary clustering happens when a key that hashes to some index position will follow the same rehashing pattern as all the other keys that hashed to the same index position before it.
- secondary clustering happens when rehash patterns that start from two or more index positions merge together.

Open Address Methods
2. Quadratic Probing

Let h(key) = h_0 = Home Address.

If there is a collision at h_0 then calculate another address using:

$$h_i = (h_0 \pm i^2) \% N, \qquad i = 1, 2, \ldots, (N-1)/2$$

until either the target key is located (retrieving) or an empty position is found (inserting) or the table is completely searched.

Note:
All table positions are visited without repetition if the following is true:

N = 4*k + 3 = prime number

Quadratic Probing Example

The following hash table implementation has an array defined as:

E[ ] A = (E[ ]) new object[7];

where an entry E, has a key field and a value field: (k, v) pair ;

and a hash function that maps each key to an integer in the range [0, 6]:

h(key) = key % 7.

Insert the entries of the following keys into the hash table:
374, 972, 911, 740, 227, 934, 362.

Use quadratic probing to resolve collisions.

Quadratic Probing Example (continued)

h(374) = 374 % 7 = 3.
h(972) = 972 % 7 = 6.
h(911) = 911 % 7 = 1.
h(740) = 740 % 7 = 5.

The array:

Index:	0	1	2	3	4	5	6
entry key:		911		374		740	972

Quadratic Probing Example (continued)

h(374) = 374 % 7 = 3.
h(972) = 972 % 7 = 6.
h(911) = 911 % 7 = 1.
h(740) = 740 % 7 = 5.
h(227) = 227 % 7 = 3;

The array:

Index:	0	1	2	3	4	5	6
entry key:		911		374		740	972

Quadratic Probing Example (continued)

h(374) = 374 % 7 = 3.
h(972) = 972 % 7 = 6.
h(911) = 911 % 7 = 1.
h(740) = 740 % 7 = 5.
h(227) = 227 % 7 = 3; h_{1+} = 4.

The array:

Index:	0	1	2	3	4	5	6
entry key:		911		374		740	972

Quadratic Probing Example (continued)

h(374) = 374 % 7 = 3.
h(972) = 972 % 7 = 6.
h(911) = 911 % 7 = 1.
h(740) = 740 % 7 = 5.
h(227) = 227 % 7 = 3; h_{1+} = 4.

The array:

Index:	0	1	2	3	4	5	6
entry key:		911		374	227	740	972

Quadratic Probing Example (continued)

$h(374) = 374 \% 7 = 3.$
$h(972) = 972 \% 7 = 6.$
$h(911) = 911 \% 7 = 1.$
$h(740) = 740 \% 7 = 5.$
$h(227) = 227 \% 7 = 3; \quad h_{1+} = 4.$
$h(934) = 934 \% 7 = 3;$

The array:

Index:	0	1	2	3	4	5	6
entry key:		911		374	227	740	972

Quadratic Probing Example (continued)

$h(374) = 374 \% 7 = 3.$
$h(972) = 972 \% 7 = 6.$
$h(911) = 911 \% 7 = 1.$
$h(740) = 740 \% 7 = 5.$
$h(227) = 227 \% 7 = 3; \quad h_{1+} = 4.$
$h(934) = 934 \% 7 = 3; \quad h_{1+} = 4;$

The array:

Index:	0	1	2	3	4	5	6
entry key:		911		374	227	740	972

Quadratic Probing Example (continued)

$h(374) = 374 \% 7 = 3.$
$h(972) = 972 \% 7 = 6.$
$h(911) = 911 \% 7 = 1.$
$h(740) = 740 \% 7 = 5.$
$h(227) = 227 \% 7 = 3; \quad h_{1+} = 4.$
$h(934) = 934 \% 7 = 3; \quad h_{1+} = 4; h_{1-} = 2.$

The array:

Index:	0	1	2	3	4	5	6
entry key:		911		374	227	740	972

Quadratic Probing Example (continued)

$h(374) = 374 \% 7 = 3.$
$h(972) = 972 \% 7 = 6.$
$h(911) = 911 \% 7 = 1.$
$h(740) = 740 \% 7 = 5.$
$h(227) = 227 \% 7 = 3; \quad h_{1+} = 4.$
$h(934) = 934 \% 7 = 3; \quad h_{1+} = 4; h_{1-} = 2.$

The array:

Index:	0	1	2	3	4	5	6
entry key:		911	934	374	227	740	972

Quadratic Probing Example (continued)

$h(374) = 374 \% 7 = 3.$
$h(972) = 972 \% 7 = 6.$
$h(911) = 911 \% 7 = 1.$
$h(740) = 740 \% 7 = 5.$
$h(227) = 227 \% 7 = 3; \quad h_{1+} = 4.$
$h(934) = 934 \% 7 = 3; \quad h_{1+} = 4; h_{1-} = 2.$
$h(362) = 362 \% 7 = 5;$

The array:

Index:	0	1	2	3	4	5	6
entry key:		911	934	374	227	740	972

Quadratic Probing Example (continued)

$h(374) = 374 \% 7 = 3.$
$h(972) = 972 \% 7 = 6.$
$h(911) = 911 \% 7 = 1.$
$h(740) = 740 \% 7 = 5.$
$h(227) = 227 \% 7 = 3; \quad h_{1+} = 4.$
$h(934) = 934 \% 7 = 3; \quad h_{1+} = 4; h_{1-} = 2.$
$h(362) = 362 \% 7 = 5; \quad h_{1+} = 6;$

The array:

Index:	0	1	2	3	4	5	6
entry key:		911	934	374	227	740	972

Quadratic Probing Example (continued)

$h(374) = 374 \% 7 = 3.$
$h(972) = 972 \% 7 = 6.$
$h(911) = 911 \% 7 = 1.$
$h(740) = 740 \% 7 = 5.$
$h(227) = 227 \% 7 = 3; \quad h_{1+} = 4.$
$h(934) = 934 \% 7 = 3; \quad h_{1+} = 4; h_{1-} = 2.$
$h(362) = 362 \% 7 = 5; \quad h_{1+} = 6; h_{1-} = 4;$

The array:

Index:	0	1	2	3	4	5	6
entry key:		911	934	374	227	740	972

Quadratic Probing Example (continued)

$h(374) = 374 \% 7 = 3.$
$h(972) = 972 \% 7 = 6.$
$h(911) = 911 \% 7 = 1.$
$h(740) = 740 \% 7 = 5.$
$h(227) = 227 \% 7 = 3; \quad h_{1+} = 4.$
$h(934) = 934 \% 7 = 3; \quad h_{1+} = 4; h_{1-} = 2.$
$h(362) = 362 \% 7 = 5; \quad h_{1+} = 6; h_{1-} = 4; h_{2+} = 2;$

The array:

Index:	0	1	2	3	4	5	6
entry key:		911	934	374	227	740	972

Quadratic Probing Example (continued)

$h(374) = 374 \% 7 = 3.$
$h(972) = 972 \% 7 = 6.$
$h(911) = 911 \% 7 = 1.$
$h(740) = 740 \% 7 = 5.$
$h(227) = 227 \% 7 = 3;\quad h_{1+} = 4.$
$h(934) = 934 \% 7 = 3;\quad h_{1+} = 4;\ h_{1-} = 2.$
$h(362) = 362 \% 7 = 5;\quad h_{1+} = 6;\ h_{1-} = 4;\ h_{2+} = 2;\ h_{2-} = 1;$

The array:

Index:	0	1	2	3	4	5	6
entry key:		911	934	374	227	740	972

Quadratic Probing Example (continued)

$h(374) = 374 \% 7 = 3.$
$h(972) = 972 \% 7 = 6.$
$h(911) = 911 \% 7 = 1.$
$h(740) = 740 \% 7 = 5.$
$h(227) = 227 \% 7 = 3;\quad h_{1+} = 4.$
$h(934) = 934 \% 7 = 3;\quad h_{1+} = 4;\ h_{1-} = 2.$
$h(362) = 362 \% 7 = 5;\quad h_{1+} = 6;\ h_{1-} = 4;\ h_{2+} = 2;\ h_{2-} = 1;$
$h_{3+} = 0.$

The array:

Index:	0	1	2	3	4	5	6
entry key:		911	934	374	227	740	972

Quadratic Probing Example (continued)

$h(374) = 374 \% 7 = 3$.
$h(972) = 972 \% 7 = 6$.
$h(911) = 911 \% 7 = 1$.
$h(740) = 740 \% 7 = 5$.
$h(227) = 227 \% 7 = 3$; $h_{1+} = 4$.
$h(934) = 934 \% 7 = 3$; $h_{1+} = 4$; $h_{1-} = 2$.
$h(362) = 362 \% 7 = 5$; $h_{1+} = 6$; $h_{1-} = 4$; $h_{2+} = 2$; $h_{2-} = 1$; $h_{3+} = 0$.

The array:

Index:	0	1	2	3	4	5	6
entry key:	362	911	934	374	227	740	972

Quadratic Probing Advantages & Disadvantages

- No secondary clustering.
- Primary clustering still a problem.
- Not all the table entries are searched:
  - If table size is a prime number, then at least half the table is searched.
  - If table size is 4*k+3, and a prime number, then all the table is searched.

Open Address Methods
3. Double Hashing

- The ideal rehashing method is to use a random rehashing function, where the distance between any two probes is random.
- However, it must also be repeatable, which makes it inefficient.
- The double hashing method attempts to approximate a random hashing function, that is both repeatable, and efficient.
- The idea is to rehash using a variable distance that depends on the key being hashed.

Open Address Methods
3. Double Hashing (cont.)

Let h(key) = h_0 = Home Address.

Define any randomizing function c(key), where $c < N$, and N is relatively prime with c. For example:

$$c(\text{key}) = 1 + [\ \text{key} \ \%\ (N\text{-}2)\ ]$$

If there is a collision at h_0 then calculate another address using:

$$h_i = (h_0 + i * c)\ \%\ N, \qquad i = 1, 2, \ldots, N\text{-}1$$

until either the target key is located (retrieving) or an empty position is found (inserting) or the table is completely searched.

Double Hashing Example

The following hash table implementation has an array defined as:

E[ ] A = (E[ ]) new object[7];

where an entry E, has a key field and a value field: (k, v) pair ;

and a hash function that maps each key to an integer in the range [0, 6]:

h(key) = key % 7.

Insert the entries of the following keys into the hash table:
374, 972, 911, 740, 227, 934, 362.

Use double hashing to resolve collisions, with the randomizing function:

c(key) = 1 + [key % 5].

Double Hashing Example (continued)

h(374) = 3.
h(972) = 6.
h(911) = 1.
h(740) = 5.

The array:

Index:	0	1	2	3	4	5	6
entry key:		911		374		740	972

Double Hashing Example (continued)

h(374) = 3.
h(972) = 6.
h(911) = 1.
h(740) = 5.
h(227) = 3;

The array:

Index:	0	1	2	3	4	5	6
entry key:		911		374		740	972

Double Hashing Example (continued)

h(374) = 3.
h(972) = 6.
h(911) = 1.
h(740) = 5.
h(227) = 3; c(227) = 3; h_1 = (3 + 1*3) % 7 = 6;

The array:

Index:	0	1	2	3	4	5	6
entry key:		911		374		740	972

Double Hashing Example (continued)

$h(374) = 3.$
$h(972) = 6.$
$h(911) = 1.$
$h(740) = 5.$
$h(227) = 3;\ c(227) = 3;\ h_1 = 6;\ h_2 = (3 + 2*3)\ \%\ 7 = 2.$

The array:

Index:	0	1	2	3	4	5	6
entry key:		911		374		740	972

Double Hashing Example (continued)

$h(374) = 3.$
$h(972) = 6.$
$h(911) = 1.$
$h(740) = 5.$
$h(227) = 3;\ c(227) = 3;\ h_1 = 6;\ h_2 = 2.$

The array:

Index:	0	1	2	3	4	5	6
entry key:		911	227	374		740	972

Double Hashing Example (continued)

h(374) = 3.
h(972) = 6.
h(911) = 1.
h(740) = 5.
h(227) = 3; c(227) = 3; h_1 = 6; h_2 = 2.
h(934) = 3;

The array:

Index:	0	1	2	3	4	5	6
entry key:		911	227	374		740	972

Double Hashing Example (continued)

h(374) = 3.
h(972) = 6.
h(911) = 1.
h(740) = 5.
h(227) = 3; c(227) = 3; h_1 = 6; h_2 = 2.
h(934) = 3; c(934) = 5; h_1 = (3 + 1*5) % 7 = 1;

The array:

Index:	0	1	2	3	4	5	6
entry key:		911	227	374		740	972

Double Hashing Example (continued)

$h(374) = 3.$
$h(972) = 6.$
$h(911) = 1.$
$h(740) = 5.$
$h(227) = 3;\ c(227) = 3;\ h_1 = 6;\ h_2 = 2.$
$h(934) = 3;\ c(934) = 5;\ h_1 = 1;\ h_2 = (3 + 2*5) \% 7 = 6;$

The array:

Index:	0	1	2	3	4	5	6
entry key:		911	227	374		740	972

Double Hashing Example (continued)

$h(374) = 3.$
$h(972) = 6.$
$h(911) = 1.$
$h(740) = 5.$
$h(227) = 3;\ c(227) = 3;\ h_1 = 6;\ h_2 = 2.$
$h(934) = 3;\ c(934) = 5;\ h_1 = 1;\ h_2 = 6;\ h_3 = (3 + 3*5) \% 7 = 4.$

The array:

Index:	0	1	2	3	4	5	6
entry key:		911	227	374		740	972

Double Hashing Example (continued)

h(374) = 3.
h(972) = 6.
h(911) = 1.
h(740) = 5.
h(227) = 3; c(227) = 3; h_1 = 6; h_2 = 2.
h(934) = 3; c(934) = 5; h_1 = 1; h_2 = 6; h_3 = 4.

The array:

Index:	0	1	2	3	4	5	6
entry key:		911	227	374	934	740	972

Double Hashing Example (continued)

h(374) = 3.
h(972) = 6.
h(911) = 1.
h(740) = 5.
h(227) = 3; c(227) = 3; h_1 = 6; h_2 = 2.
h(934) = 3; c(934) = 5; h_1 = 1; h_2 = 6; h_3 = 4.
h(362) = 5; c(362) = 3; h_1 = 1; h_2 = 4; h_3 = 0.

The array:

Index:	0	1	2	3	4	5	6
entry key:	362	911	227	374	934	740	972

Double Hashing Advantages & Disadvantages

- No primary or secondary clustering.
- Quick and efficient. Also simple to apply.
- Performs very close to the ideal case (random rehashing), for both successful and un-successful accesses.
- Table size N, must be a prime number.
- If (N-2) is used in the randomizing function c, it must also be a prime number, so that all the table is searched without repetition.

Open Address Methods Advantages & Disadvantages

- Require minimum amount of memory.
- Searching and insertion are easy and efficient.
- Deletion is very difficult and inefficient.
- Table size N, is fixed, no more than N items can be inserted.
- The worst case is very bad, but rare.

External (Separate) Chaining

- Table positions contain pointers to nodes of actual entries that are dynamically created.
- When a collision occurs at some position, a new entry node is created and added to the linked list that starts at that position.

External (Separate) Chaining Example

The following hash table implementation has an array defined as:

E[ ] A = (E[ ]) new object[7];

where an entry E, has a key field and a value field: (k, v) pair ;

and a hash function that maps each key to an integer in the range [0, 6]:

h(key) = key % 7.

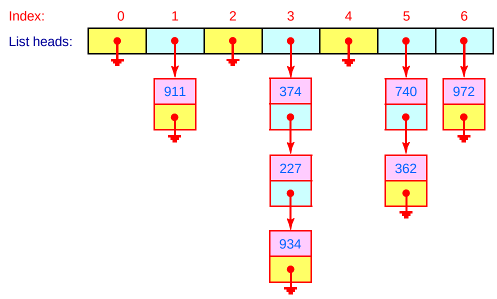
Insert the entries of the following keys into the hash table:
374, 972, 911, 740, 227, 934, 362.

Use external chaining to resolve collisions.

External (Separate) Chaining Example (continued)

h(374) = 374 % 7 = 3.
h(972) = 972 % 7 = 6.
h(911) = 911 % 7 = 1.
h(740) = 740 % 7 = 5.
h(227) = 227 % 7 = 3.
h(934) = 934 % 7 = 3.
h(362) = 362 % 7 = 5.

The Bucket Array:



External (Separate) Chaining Advantages & Disadvantages

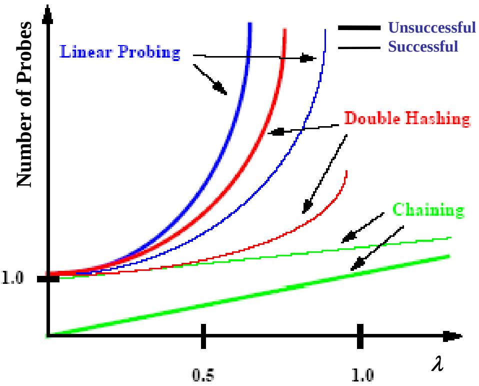
- Deletion is possible with no problems.
- The number of items can be more than N.
- Has better search performance than previous methods.
- Requires more storage for the pointers.

The Performance of Hashing

Method	Expected Number of Un-successful Probes	Expected Number of successful Probes	Required Memory
Linear Probing	$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$	$\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right)$	$N \cdot w$
Double Hashing	$\frac{1}{1-\lambda}$	$\frac{-\ln(1-\lambda)}{\lambda}$	$N \cdot w$
External Chaining	$\lambda + e^{\lambda}$	$1+\frac{\lambda}{2}$	$N + n(w+1)$

The Effect of the Load Factor on Hash Table Performance

- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The expected running time of all operations in a hash table is $O(1)$
- With open addressing λ should be less than 0.5
- With external chaining λ should be between 0.75 and 0.9
- As the load factor gets closer to 1.0, clustering becomes a problem, pushing performance towards $O(n)$



Expected Number of Probes vs. Load Factor λ

Hash Tables Applications

- **Database systems**: Specifically, those that require efficient random access. Generally, database systems try to optimize between two types of access methods: sequential and random. Hash tables are an important part of efficient random access because they provide a way to locate data in a constant amount of time.
- **Symbol tables**: The tables used by compilers to maintain information about symbols from a program. Compilers access information about symbols frequently. Therefore, it is important that symbol tables be implemented very efficiently.
- **Data dictionaries**: Data structures that support adding, deleting, and searching for data. Although the operations of a hash table and a data dictionary are similar, other data structures may be used to implement data dictionaries. Using a hash table is particularly efficient.
- **Network processing algorithms**: Hash tables are fundamental components of several network processing algorithms and applications, including route lookup, packet classification, and network monitoring.
- **Browser Cashes**: Hash tables are used to implement browser cashes.

Empty Slide