DET102 Data Structures and Algorithms

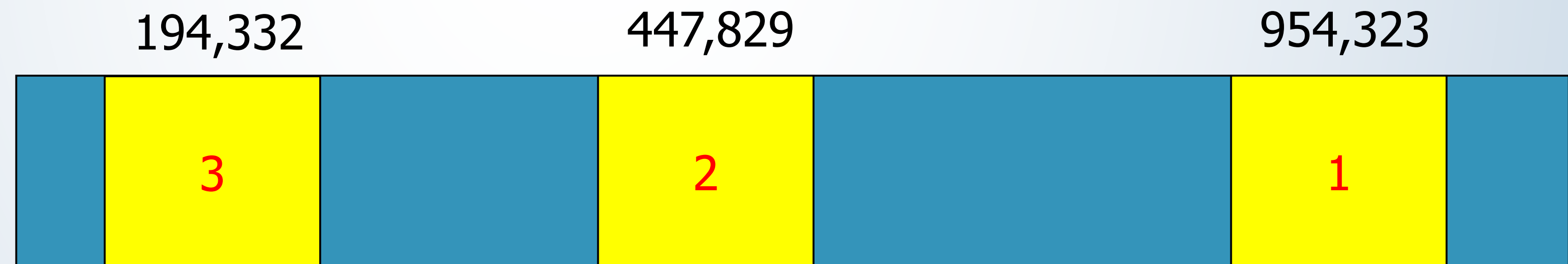
Lecture 11: Hash

Sparse data

- There are many players in a complex game. Each player has an identification number (key). The range of the keys can be 1~1,000,000. No two players have the same key
- Suppose now we have 3 players
  - 954,323
  - 447,829
  - 194,332
- They are far away from each other.
- The player information is called key-based data

Sparse data

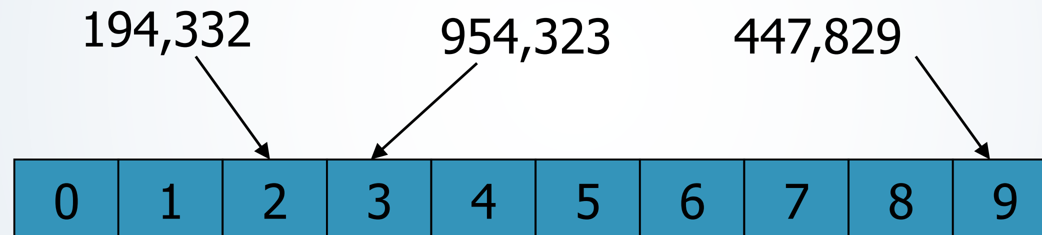
- How to store those data in the computer so that we can easily get the player's information by their keys?
  - Array:
    - A lot of memory space wasted

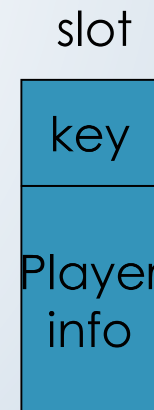


  - Linked List:
    - Hard to search if we have 10,000 players
  - Hash Table
    - Best solution in this case!

Basic Hash Table

- Advantages:
  - Quickly store sparse key-based data in a reasonable amount of space
  - Quickly determine if a certain key is within the table





194,332%10=2 or 194,332≡2 (mod 10)

447,879%10=9 or 447,879≡9 (mod 10)

954,323%10=3 or 954,323≡3 (mod 10)

To get the information, we use: player=table[key%10];

Hash Table

- Hash table is one of the most practical data structures.
- The location of each inserted item is decided by a hash function.

```
insert(data)
    A[h(data.key)]=data

search(data)
    return h(data.key)
```

data is the input
data.key is an integer

h is the hash function, which calculates an index based on data.key

h(data.key) is called hash value.

Hash function

- The goal of a ***hash function***, h, is to map each key k to an integer in the range $[0, N-1]$, where N is the capacity of the bucket array for a hash table.
- Equipped with such a hash function, h, the main idea of this approach is to use the hash function value, $h(k)$, as an index into our bucket array, A, instead of the key k (which may not be appropriate for direct use as an index). That is, we store the item (k,v) in the bucket $A[h(k)]$.

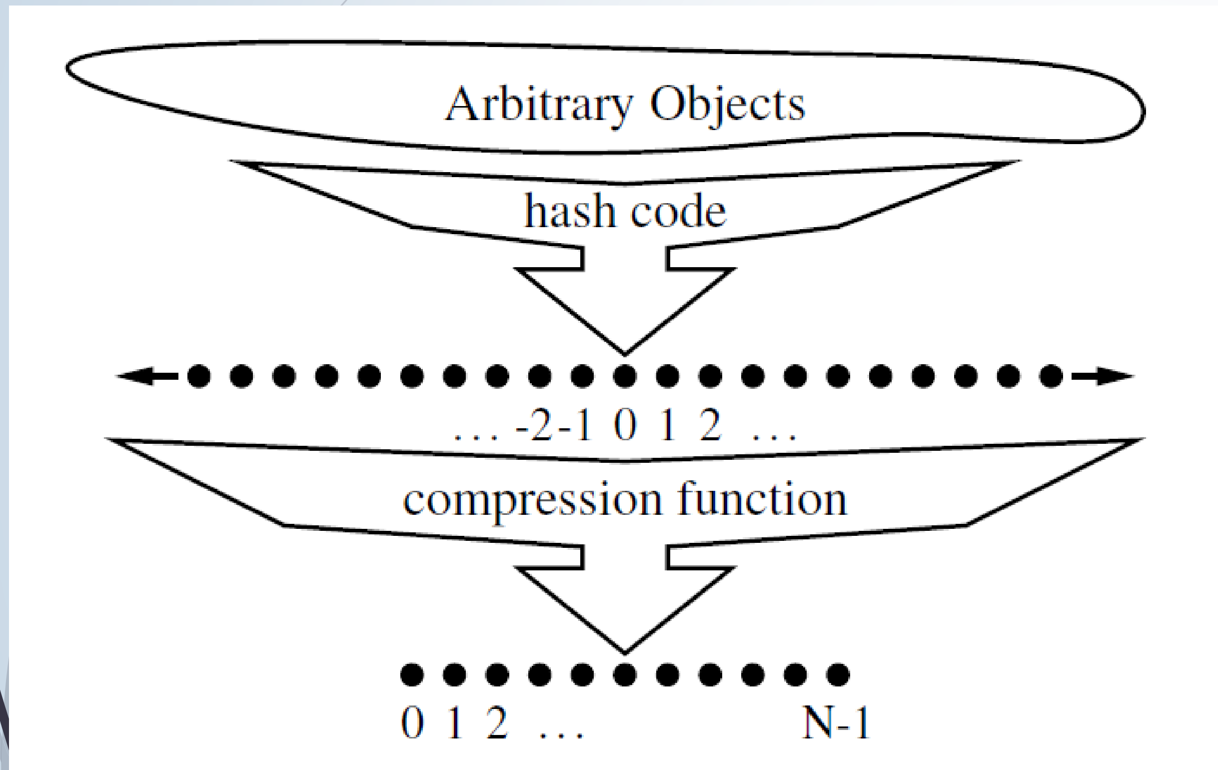
Hash function

- h(k) is called a hash function, which returns an index of A.
- Assume that A's size is m, then h(k) is from 0 to m-1.
- Hash function examples
  - h(k)=k mod m

Question 1: How to get k ?

Question 2: How about two keys with the same hash value ?

Hash function

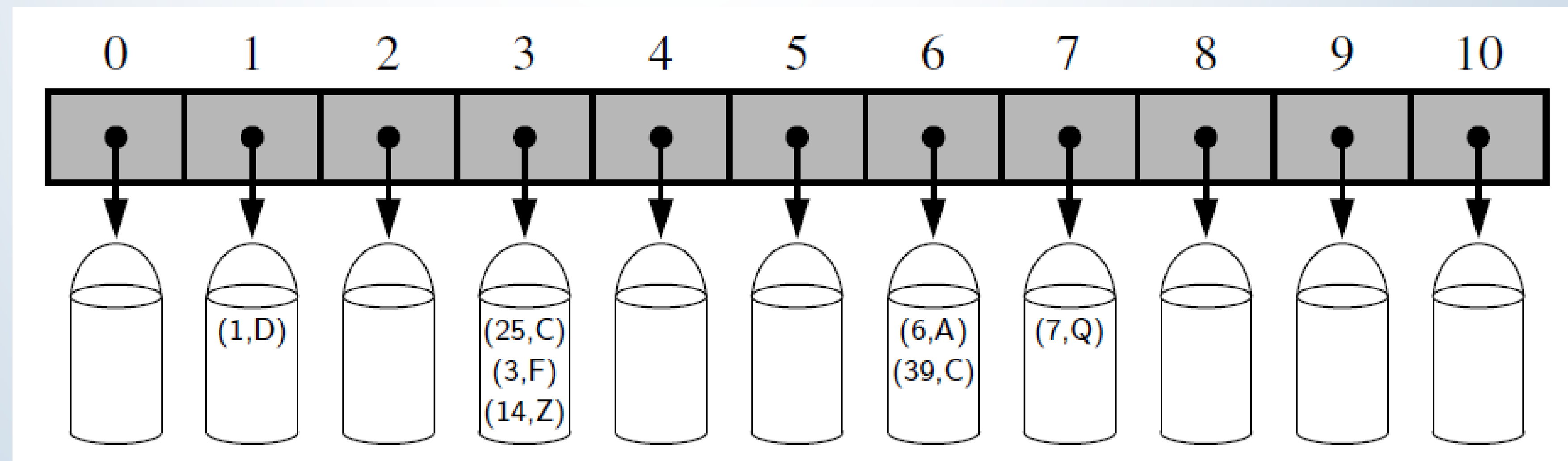


hash code : map a key k to an integer

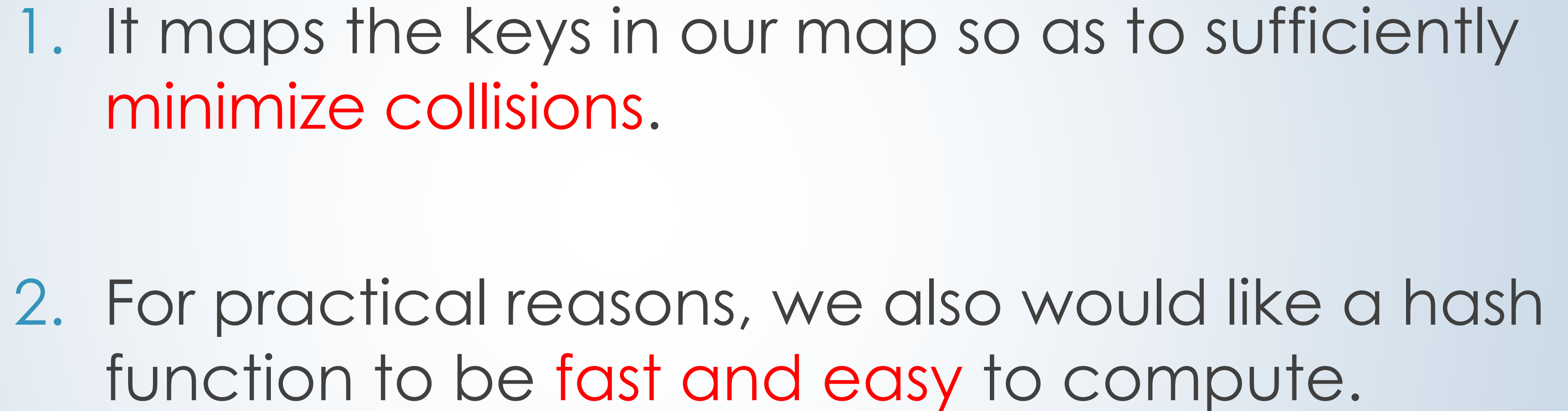
Compression function : maps the hash code to an integer within a range of indices, $[0, N-1]$, for a bucket array.

Collision

- If there are two or more keys with the same hash value, then two different items will be mapped to the same bucket in *T*. In this case, we say that a ***collision*** has occurred.



Good Hash function


1. It maps the keys in our map so as to sufficiently minimize collisions.
2. For practical reasons, we also would like a hash function to be fast and easy to compute.

Hash function example

also called Compression Functions

1. *The division method*

$$h(i) = i \bmod N$$

- i is the hash code
- N is the size of the bucket array (a fixed positive integer, prime is better)

Example: hash codes {200,205,210, 215,…,600} into a bucket array of size N.

if N=100

if N=101

Hash function example

2. The MAD method (Multiply-Add-and-Divide)

$$h(i)= [(ai+b) \bmod p] \bmod N$$

- i is the hash code
- p is a prime number *larger than* N
- N is the size of the bucket array (a fixed positive integer)
- a and b are integers chosen at random from the interval $[0, p-1]$.

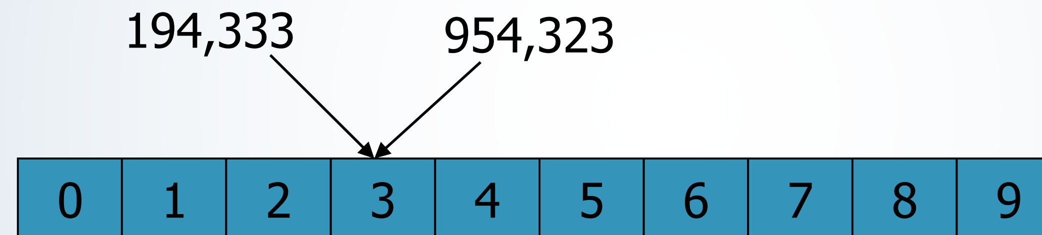
Other example: $h(k) = (k^2 + k + 41) \% N$

Combination of Hash Functions

- Collision is easy to happen if we use % function
- Combination:
  - Apply hash function h_1 on key to obtain mid_key
  - Apply hash function h_2 on mid_key to obtain Slot_id
- Example:
  - We apply %101 on 12320324111220 and get 79
  - We apply %10 on the result 79 obtained by %101
    - 79 % 10 =9

Collisions

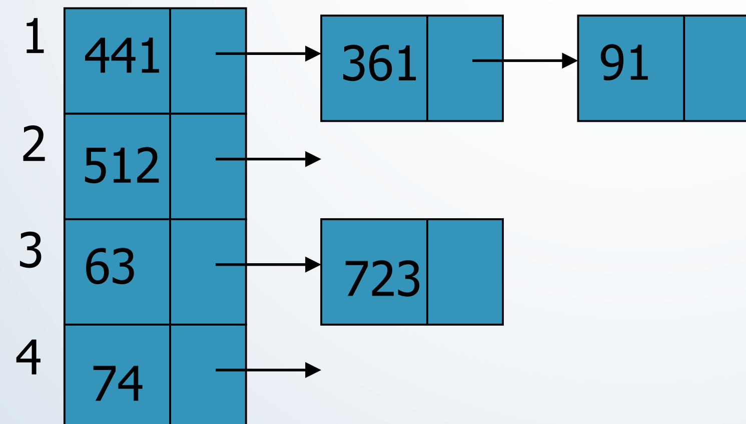
- Two players mapped to the same cell



- Method to deal with collisions
  - Change the table
  - collision-handling schemes
    - separate chaining
    - open addressing

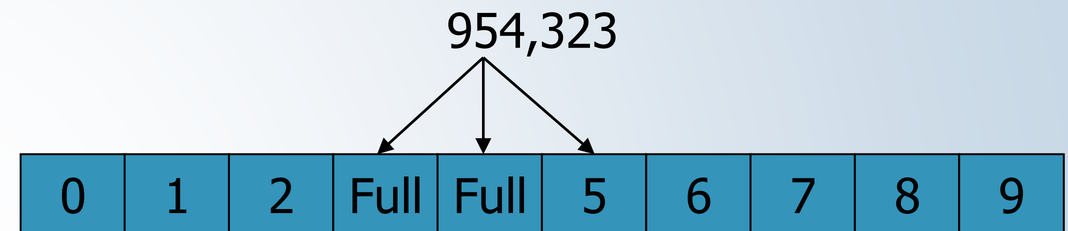
Collision Resolution - Separate Chaining

- Using linked list to solve Collision
  - Every slot in the hash table is a linked list
  - Collision→Insert into the corresponding list
  - Find data→Search the corresponding list

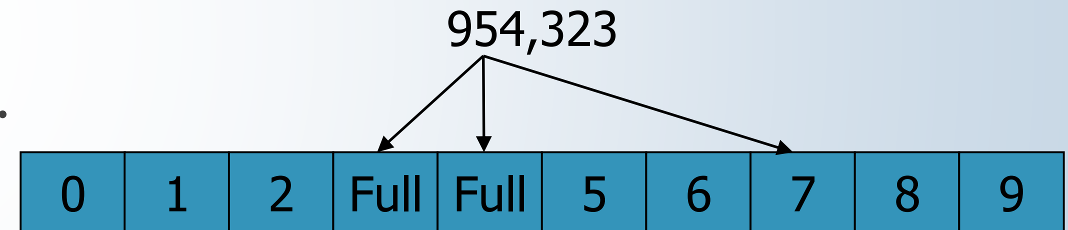


Collision Resolution - Open Addressing

- Linear Probing
  - If collide, try Slot_id+1, Slot_id+2

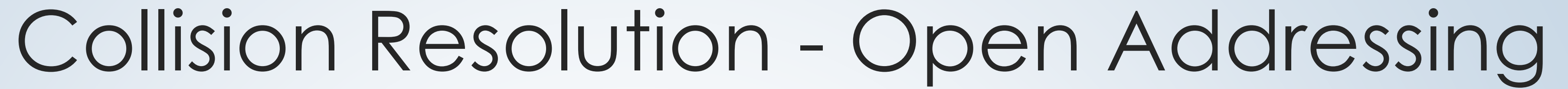


- Quadratic Probing
  - If collide, try Slot_id+1, Slot_id+4,…



- Double Hashing
  - If collide, try Slot_id+$h_2(x)$, Slot_id+$2h_2(x)$,… (prime size important)

Collision Resolution - Open Addressing



- General rule: If collide, try other slots in a certain order
- How to find data?
  - If not found, try the next position according to different probing rule
  - Every key has a preference over all the positions
  - When finding them, just search in the order of their preferences

Collision Resolution

- Example: 11,22,33,44,55,66,77,88,99,21
  - Using linear probing

21	11	22	33	44	55	66	77	88	99

  - Using separate chaining

1	11	→ 21
2	22	
3	33	
4	44	
5	55	

More on Hash Table Size

Table of prime size is important in the following cases:

a) For quadratic probing, we have the following property:

- If quadratic probing is used and the table size is prime, then a new element can always be inserted if the table is at least half empty (Why only prime can do?).

b) For double hashing, we have the following property:

- If double hashing is used and the table size is prime, then a new element can always be inserted if the table is not full (Is this correct?).

$$\lambda = n/N$$

- n is the number of slots occupied.
- N is the number of total slots

With open addressing, as the load factor λ grows beyond 0.5 and starts approaching 1, clusters of entries in the bucket array start to grow as well.

- suggest to maintain $\lambda < 0.5$ for an open addressing scheme with linear probing.
- perhaps only a bit higher for other open addressing schemes (Python's implementation $\lambda < 2/3$)

- Each rehashing will generally scatter the items throughout the new bucket array.
- When rehashing to a new table, it is a good requirement for the new array’s size to be at least double the previous size.
- Indeed, if we always double the size of the table with each rehashing operation, then we can amortize the cost of rehashing all the entries in the table against the time used to insert them in the first place.

Rehashing

- When half full, rehash all the elements into a double-size table
- In an interactive system, the user who triggers rehashing is unlucky
- In total, only O(n) cost incurred for a hash table of size n
- Example: initial hash table size 2, when the size grows to 32, how many rehashes are done?
  - 2→4 1 number rehashed
  - 4→8 2 numbers rehashed
  - 8→16 4 numbers rehashed
  - 16→32 8 numbers rehashed
  - In total, 15 numbers rehashed, 15<16=32/2

More Questions

- How can rehashing be used?
  - If we allow rehashing, then quadratic probing can always succeed in inserting new items because the table will always be at least half empty.
- How to keep the table size still prime when you do rehashing?

Application —— Dictionary

- How do Word perform spelling check?
- A dictionary (large hash table) is kept
- Hash words into that dictionary
- The way to hash words
  - Establish a map between characters and numbers
  - E.g. A—136, F—356, T—927, E—442, R—091
  - “AFTER” corresponds to the key 136,356,927,442,091
  - Hashing ‘AFTER’ will be equivalent to hashing the key

Double Hashing

- *Double hashing can be done using :*
  $(h_1(key) + i * h_2(key))$ % TABLE_SIZE
  Here $h_1()$ and $h_2()$ are hash functions and TABLE_SIZE is size of hash table.
  (We repeat by increasing i when collision occurs)
- First hash function is typically

  $h_1(key)$ = key % TABLE_SIZE
- A popular second hash function is :

  $h_2(key)$ = PRIME – (key % PRIME)

  where PRIME is a prime smaller than the TABLE_SIZE.

$h_2(key)$ cannot be zero.

Exercise

1. Draw the 11-entry hash table that results from using the hash function, h(i) = (3i+5) mod 11, to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5, assuming collisions are handled by chaining.
2. What is the result of the previous exercise, assuming collisions are handled by linear probing?
3. What is the result of the previous exercise, assuming collisions are handled by quadratic probing?
4. What is the result of the previous exercise, assuming collisions are handled by double hashing and

   $h_2(x) = 5-(x \bmod 5)$.

Answer to Exercises

- $h(i) = (3i+5) \bmod 11$
- $h_2(x) = 5-(x \bmod 5)$.
- 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5

	12	**44**	**13**	**88**	**23**	**94**	**11**	**39**	**20**	**16**	**5**
$h(x)$	8	5	0	5	8	1	5	1	10	9	9
$h_2(x)$	3	1	2	2	2	1	4	1	5	4	5

1.collisions are handled by chaining

- $h(i) = (3i+5) \bmod 11$
- 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5

	12	**44**	**13**	**88**	**23**	**94**	**11**	**39**	**20**	**16**	**5**
h(x)	8	5	0	5	8	1	5	1	10	9	9
$h_2(x)$	3	1	2	2	2	1	4	1	5	4	5

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
13	94				44			12	16	20
	39				88			23	5	
					11					

2.collisions are handled by linear probing

- h(i) = (3i+5) mod 11
- 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5

	12	**44**	**13**	**88**	**23**	**94**	**11**	**39**	**20**	**16**	**5**
h(x)	8	5	0	5	8	1	5	1	10	9	9
$h_2(x)$	3	1	2	2	2	1	4	1	5	4	5

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
13	94	39	16	5	44	88	11	12	23	20

3. collisions are handled by quadratic probing

- $h(i) = (3i+5) \bmod 11$
- 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5

	12	44	13	88	23	94	11	39	20	16	5
h(x)	8	5	0	5	8	1	5	1	10	9	9
$h_2(x)$	3	1	2	2	2	1	4	1	5	4	5

0	1	2	3	4	5	6	7	8	9	10
13	94	39	11		44	88	16	12	23	20

(9+1)%11=10, (9+4)%11=2, (9+9)%11=7, (9+16)%11=3, (9+25)%11=1, (9+36)%11=1, (9+49)%11=3, (9+64)%11=7, (9+81)%11=2, (9+100)%=10

4.collisions are handled by double hashing

- h(i) = (3i+5) mod 11
- $h_2(x)$= 5-(x mod 5).
- 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5

	12	44	13	88	23	94	11	39	20	16	5
h(x)	8	5	0	5	8	1	5	1	10	9	9
$h_2(x)$	3	1	2	2	2	1	4	1	5	4	5

0	1	2	3	4	5	6	7	8	9	10
13	94	39	5	20	44	16	88	12	11	23