http://algs4.cs.princeton.edu

3.4 HASH TABLES → يعلمني وين اخزنه

- *hash functions*
- *separate chaining*
- *linear probing*
- *context*

7, 15, 6, 3

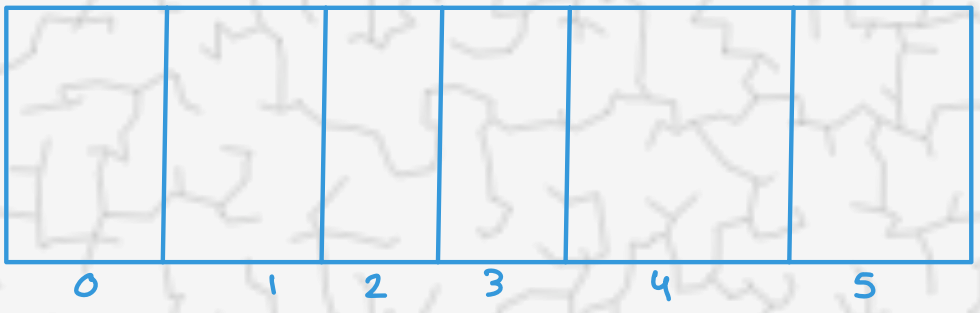
انا بعطيه الـ (element) وبستعمل المعادله علشان يعلمني وين اخزنه

$f(x) = x \,\%\, 3 \bmod 6$

لي ممكن تختلف

Uniform → اذا كل رقم بعطيني (index) مختلف لان مافيه تصادم

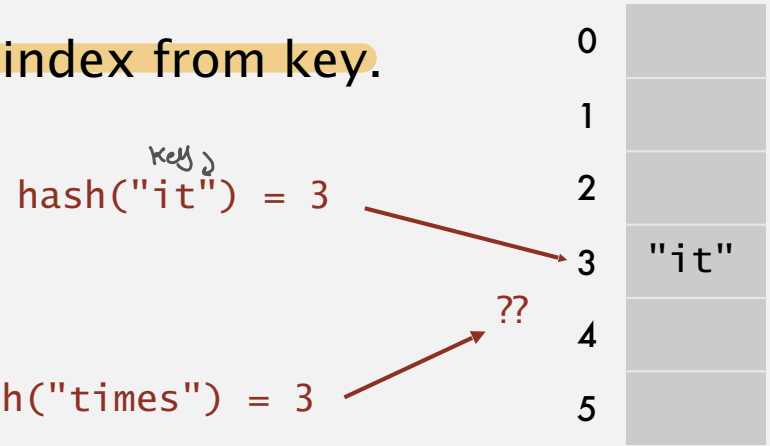
Collugin ← not uniform → بيكون اوردي مخزنه بجيبي الـ index ويطلب مني اخزنها ،



Hashing: basic plan

Save items in a key-indexed table (index is a function of the key).

Hash function. Method for computing array index from key.



Issues.

- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.
- Collision resolution: Algorithm and data structure to handle two keys that hash to the same array index.

Classic space-time tradeoff.

- No space limitation: trivial hash function with key as index.
- No time limitation: trivial collision resolution with sequential search.
- Space and time limitations: hashing (the real world).

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.
- Applications:
  - address book.
  - student-record database.

http://algs4.cs.princeton.edu

3.4 Hash Tables

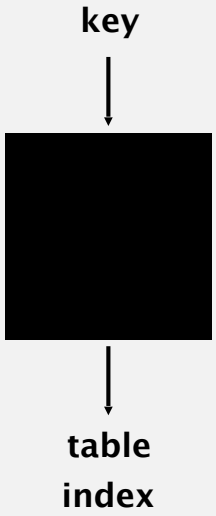
- *hash functions*
- *separate chaining*
- *linear probing*
- *context*

Computing the hash function

Idealistic goal. Scramble the keys uniformly to produce a table index.

- Efficiently computable.
- Each table index equally likely for each key.

thoroughly researched problem,
still problematic in practical applications



Ex 1. Phone numbers.

- Bad: first three digits.
- Better: last three digits.

Ex 2. Social Security numbers.

- Bad: first three digits.
- Better: last three digits.

573 = California, 574 = Alaska
(assigned in chronological order within geographic region)

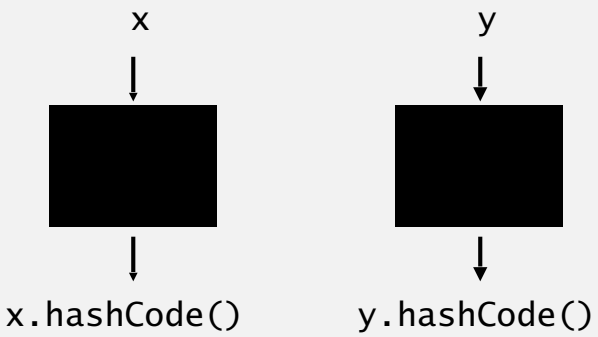
Practical challenge. Need different approach for each key type.

Java's hash code conventions

All Java classes inherit a method `hashCode()`, which returns a 32-bit `int`.

Requirement. If `x.equals(y)`, then `(x.hashCode() == y.hashCode())`.

Highly desirable. If `!x.equals(y)`, then `(x.hashCode() != y.hashCode())`.



Default implementation. Memory address of `x`.

Legal (but poor) implementation. Always return `17`.

Customized implementations. `Integer`, `Double`, `String`, `File`, `URL`, `Date`, ...

User-defined types. Users are on their own.

Map ADT

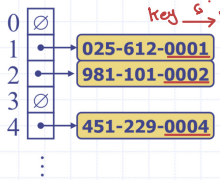
- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterator of the values in M

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*)

More General Kinds of Keys

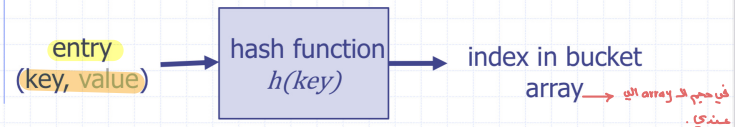
- But what should we do if our keys are not integers in the range from 0 to N – 1?
  - Use a **hash function** to map general keys to corresponding indices in a table.
  - For instance, the last four digits of a Social Security number.



Hash Functions and Hash Tables

- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
- Example:
  $$h(x) = x \bmod N$$
  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x
- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N

Hash Functions and Hash Tables



في حجم الـ array اللي
عندي.

When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

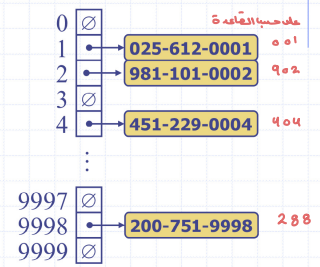
انفذ عندي الـfunction على عندي ك
الـkey بحول الـ h(k) بيدخل بـ index معين تبعه أ.

Example

- We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function $h(x) =$ last four digits of x

0000 1235

دايماً عدد الـ digit متساوي وتزيد من اليسار بالصفر.



على حسب القاعدة
001
002
004
288

Implementing hash code: integers, booleans, and doubles

Java library implementations

```
public final class Integer
{
   private final int value;
   ...

   public int hashCode()
   {  return value;  }
}
```

```
public final class Boolean
{
   private final boolean value;
   ...

   public int hashCode()
   {
      if (value) return 1231;
      else       return 1237;
   }
}
```

```
public final class Double
{
   private final double value;
   ...

   public int hashCode()
   {
      long bits = doubleToLongBits(value);
      return (int) (bits ^ (bits >>> 32));
   }
}
```

convert to IEEE 64-bit representation;
xor most significant 32-bits
with least significant 32-bits

Warning: -0.0 and +0.0 have different hash codes

Implementing hash code: strings

Java library implementation

```
public final class String
{
   private final char[] s;
   ...

   public int hashCode()
   {
      int hash = 0;
      for (int i = 0; i < length(); i++)
         hash = s[i] + (31 * hash);
      return hash;
   }
}
```

i^th character of s

char	Unicode
...	...
'a'	97
'b'	98
'c'	99
...	...

- Horner's method to hash string of length L: L multiplies/adds.
- Equivalent to $h = s[0] \cdot 31^{L-1} + \ldots + s[L-3] \cdot 31^2 + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0$.

Ex.

```
String s = "call";
int code = s.hashCode();
```

3045982 = 99·31^3 + 97·31^2 + 108·31^1 + 108·31^0
= 108 + 31· (108 + 31 · (97 + 31 · (99)))
(Horner's method)

Implementing hash code: strings

Performance optimization.

- Cache the hash value in an instance variable.
- Return cached value.

```
public final class String
{
   private int hash = 0;
   private final char[] s;
   ...

   public int hashCode()
   {
      int h = hash;
      if (h != 0) return h;
      for (int i = 0; i < length(); i++)
         h = s[i] + (31 * h);
      hash = h;
      return h;
   }
}
```

cache of hash code

return cached value

store cache of hash code

Q. What if `hashCode()` of string is 0?

Implementing hash code: user-defined types

```
public final class Transaction implements Comparable<Transaction>
{
   private final String  who;
   private final Date    when;
   private final double  amount;

   public Transaction(String who, Date when, double amount)
   {  /* as before */  }

   ...

   public boolean equals(Object y)
   {  /* as before */  }

   public int hashCode()
   {
      int hash = 17;
      hash = 31*hash + who.hashCode();
      hash = 31*hash + when.hashCode();
      hash = 31*hash + ((Double) amount).hashCode();
      return hash;
   }
}
```

nonzero constant

typically a small prime

for reference types, use hashCode()

prime number نستخدم

for primitive types, use hashCode() of wrapper type

Hash code design

"Standard" recipe for user-defined types.

- Combine each significant field using the $31x + y$ rule.
- If field is a primitive type, use wrapper type `hashCode()`.
- If field is `null`, return 0. or reference type (نطلع لـ hash code) or array (نعمل لكل فيه بال array اللي عندي)
- If field is a reference type, use `hashCode()`. ← applies rule recursively
- If field is an array, apply to each entry. ← or use `Arrays.deepHashCode()`

In practice. Recipe works reasonably well; used in Java libraries.

In theory. Keys are bitstring; "universal" hash functions exist.

Basic rule. Need to use the whole key to compute hash code; consult an expert for state-of-the-art hash codes.

Modular hashing

Hash code. An `int` between -2^{31} and $2^{31} - 1$.

Hash function. An `int` between `0` and `M - 1` (for use as array index).

typically a prime or power of 2

```
private int hash(Key key)
{  return key.hashCode() % M;  }
```

bug

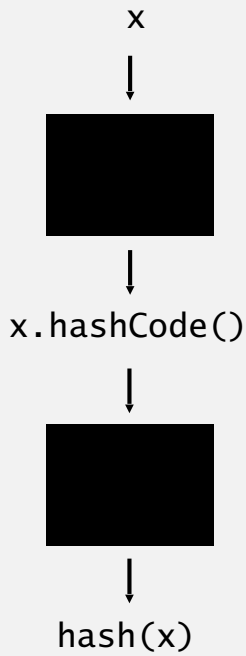
```
private int hash(Key key)
{  return Math.abs(key.hashCode()) % M;  }
```

1-in-a-billion bug

hashCode() of "polygenelubricants" is -2^{31}

```
private int hash(Key key)
{  return (key.hashCode() & 0x7fffffff) % M;  }
```

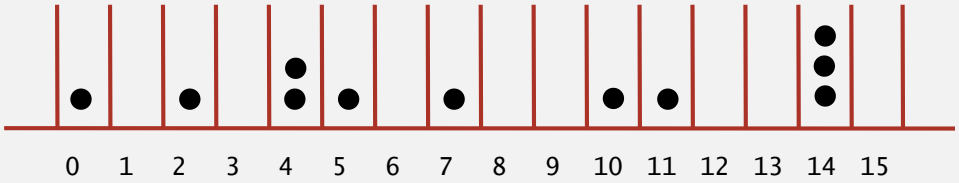
correct



Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to an integer between 0 and $M - 1$.

Bins and balls. Throw balls uniformly at random into M bins.



Birthday problem. Expect two balls in the same bin after $\sim \sqrt{\pi M / 2}$ tosses.

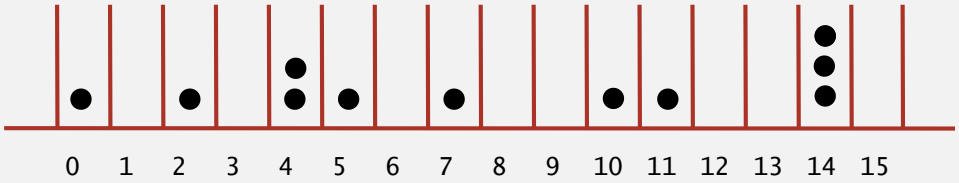
Coupon collector. Expect every bin has ≥ 1 ball after $\sim M \ln M$ tosses.

Load balancing. After M tosses, expect most loaded bin has $\Theta (\log M / \log \log M)$ balls.

Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to an integer between 0 and $M - 1$.

Bins and balls. Throw balls uniformly at random into M bins.



Java's `String` data uniformly distribute the keys of Tale of Two Cities

http://algs4.cs.princeton.edu

3.4 Hash Tables

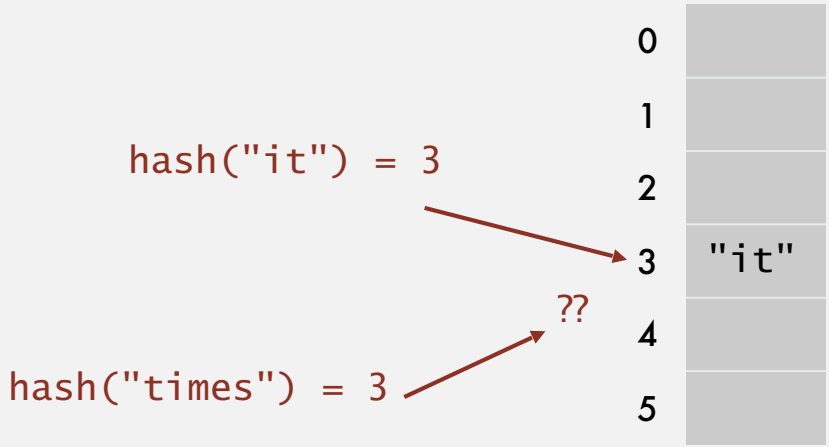
- *hash functions*
- *separate chaining*
- *linear probing*
- *context*

سلسلة ح

Collisions

Collision. Two distinct keys hashing to same index.

- Birthday problem $\Rightarrow$ can't avoid collisions unless you have a ridiculous (quadratic) amount of memory.
- Coupon collector + load balancing $\Rightarrow$ collisions are evenly distributed.



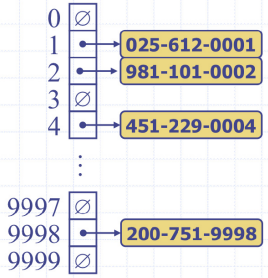
Challenge. Deal with collisions efficiently.

Example – Cont.

- What if we have a new entry to add to the bucket array:

  (037-879-0002)

- *h*(037-879-0002) = 0002 = 2
- A *collision* has just occurred.



* نضيف الرقم بـ (Index) (٢) بلاقيها فل محجوزه ،
بنسمية (collision) .
تصادم

* كيف نوجد الـ mod ؟

١- نروح لـ mod ونختار (٩) بعدين نطبق القاعده عندي : A - A ÷ B X B

٢- او نقسم A على B ونأخذ الرقم الصحيح بدون فواصل بعدين B X الناتج من A- الي بالقصاده من القسمه

Ex: 263 mod 19

263 ÷ 19 = 18 → 263 - 18 X 19 = 11

What is Hash function for following values to store them in array

Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou	Max	Ada	Ted
0	1	2	3	4	5	6	7	8	9	10

Index number = sum ASCII codes Mod size of array

(باقي القسمه ٩)

حجم الـ array Prime : ١١ يكون عدد التكرارات قليل .

								جموع كلهم	
Mia	M	77	i	105	a	97	279 →		4
Tim	T	84	i	105	m	109	298		1
Bea	B	66	e	101	a	97	264		0
Zoe	Z	90	o	111	e	101	302		5
Jan	J	74	a	97	n	110	281		6
Ada	A	65	d	100	a	97	262		9
Leo	L	76	e	101	o	111	288		2
Sam	S	83	a	97	m	109	289		3
Lou	L	76	o	111	u	117	304		7
Max	M	77	a	97	x	120	294		8
Ted	T	84	e	101	d	100	285		10

→ ما فيها تكرار

الي يجي بالاول يحجز والي يجي بعده هو الي عنده المشكله لازم يحل مشكلته

وطالما اذا بنختار حجم الـ array ناخذ ارقام Prime .

Link:- https://www.youtube.com/watch?v=KyUTuwz_b7Q

طريقة ملتان نحل التصادمات : بقولها دوري على اقرب مكان فاضي وادخلي فيه .

If new value is getting the address which occupied then new value will be inserted in next available position

Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4

مكرر

فاضي

Solving collusion by adding new value to next available position is called **open addressing** because every location is open to any item. Open addressing can use variety of open addressing techniques but this particular is called linear probing because if calculated address is occupied then linear search is used find next available slot

linear search

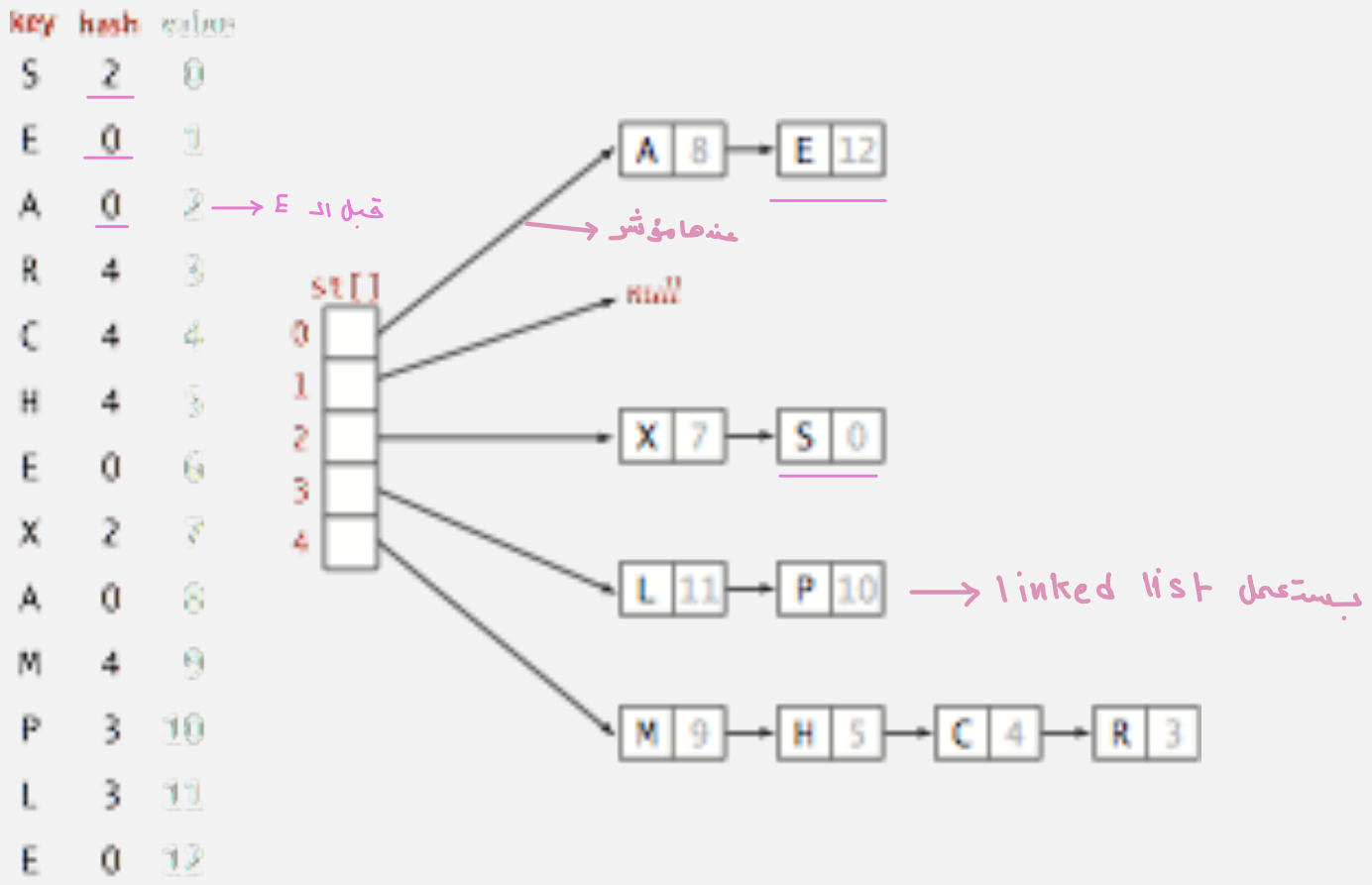
Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4
Len	L	76	e	101	n	110	287	1
Moe	M	77	o	111	e	101	289	3
Lou	L	76	o	111	u	117	304	7
Rae	R	82	a	97	e	101	280	5
Max	M	77	a	97	x	120	294	8
Tod	T	84	o	111	d	100	295	9

محجوز فاداحت للثاني

Separate-chaining symbol table

Use an array of $M < N$ linked lists. [H. P. Luhn, IBM 1953]

- Hash: map key to integer i between 0 and $M - 1$.
- Insert: put at front of i^{th} chain (if not already there).
- Search: need to search only i^{th} chain.



تسلسل تحت بعضات بـ linked list

Other way to deal wit collision is **chaining**, here array has pointer to first node

شركت بها linked بـ

Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4
Len	L	76	e	101	n	110	287	1
Moe	M	77	o	111	e	101	289	3
Lou	L	76	o	111	u	117	304	7
Rae	R	82	a	97	e	101	280	5
Max	M	77	a	97	x	120	294	8
Tod	T	84	o	111	d	100	295	9

★ Hash function : O(1)

address = key mod n

alphanumeric → ASCII code بياخدون الـ
يجمعونه او يضربونه

Folding method: divides key into parts then adds the parts together.
لو نبدا من اليمين لليسار
نجمعه ونجيب باقي القسمه من الـ array

Separate-chaining symbol table: Java implementation

```
public class SeparateChainingHashST<Key, Value>
{
   private int M = 97;                // number of chains
   private Node[] st = new Node[M];  // array of chains

   private static class Node
   {
      private Object key;
      private Object val;
      private Node next;
      ...
   }

   private int hash(Key key)
   {  return (key.hashCode() & 0x7fffffff) % M;  }

   public Value get(Key key) {
      int i = hash(key);
      for (Node x = st[i]; x != null; x = x.next)
         if (key.equals(x.key)) return (Value) x.val;
      return null;
   }

}
```

array doubling and halving code omitted

no generic array creation (declare key and value of type Object)

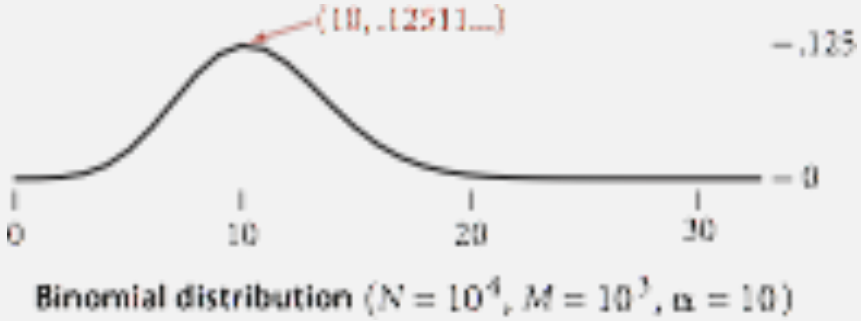
Separate-chaining symbol table: Java implementation

```
public class SeparateChainingHashST<Key, Value>
{
   private int M = 97;                // number of chains
   private Node[] st = new Node[M];  // array of chains

   private static class Node
   {
      private Object key;
      private Object val;
      private Node next;
      ...
   }

   private int hash(Key key)
   {  return (key.hashCode() & 0x7fffffff) % M;  }

   public void put(Key key, Value val) {
      int i = hash(key);
      for (Node x = st[i]; x != null; x = x.next)
         if (key.equals(x.key)) { x.val = val; return; }
      st[i] = new Node(key, val, st[i]);
   }

}
```

Analysis of separate chaining

Proposition. Under uniform hashing assumption, prob. that the number of keys in a list is within a constant factor of N / M is extremely close to 1.

Pf sketch. Distribution of list size obeys a binomial distribution.



Binomial distribution ($N = 10^4$, $M = 10^3$, $\alpha = 10$)

equals() and hashCode()

Consequence. Number of probes for search/insert is proportional to N / M.

- M too large $\Rightarrow$ too many empty chains.
- M too small $\Rightarrow$ chains too long.
- Typical choice: $M \sim N / 4$ $\Rightarrow$ constant-time ops.

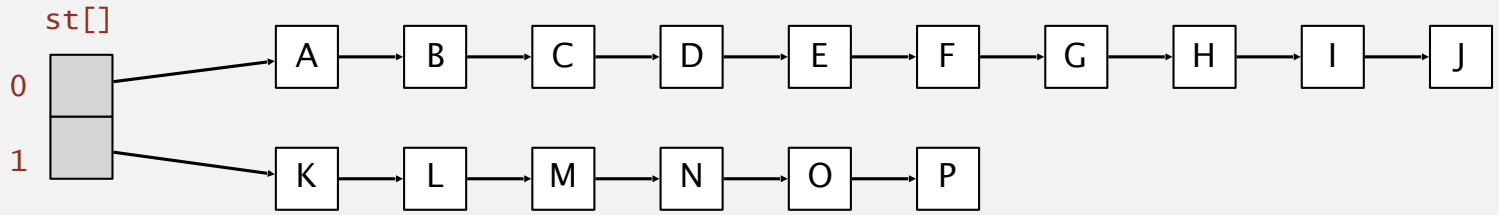
M times faster than sequential search

Resizing in a separate-chaining hash table

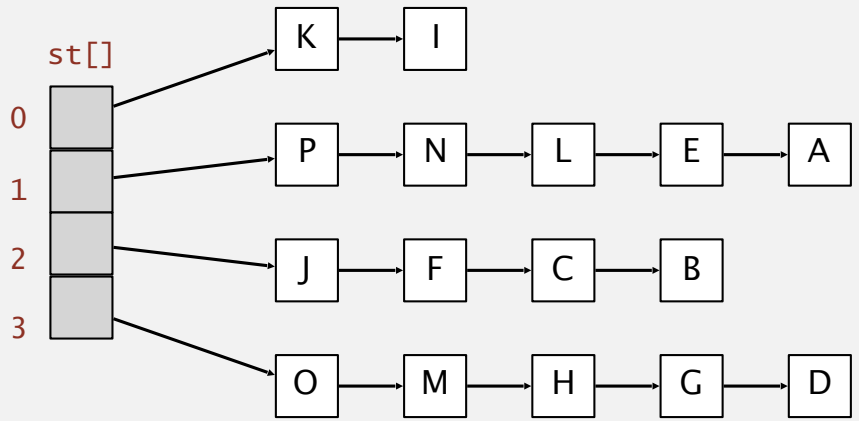
Goal. Average length of list N / M = constant.

- Double size of array M when $N / M \geq 8$.
- Halve size of array M when $N / M \leq 2$.
- Need to rehash all keys when resizing. ← x.hashCode() does not change but hash(x) can change

before resizing



after resizing



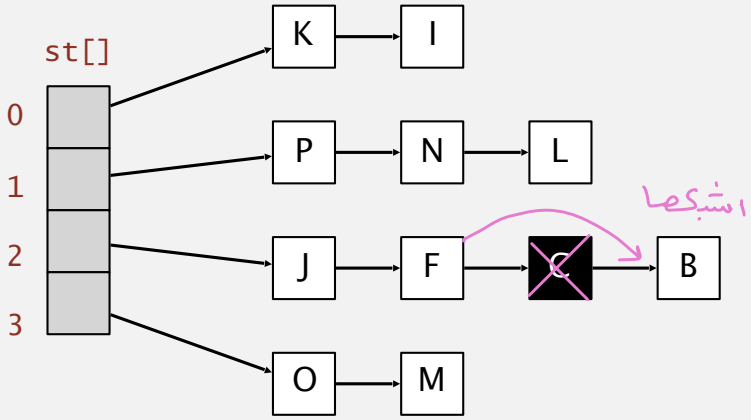
Deletion in a separate-chaining hash table

Q. How to delete a key (and its associated value)?

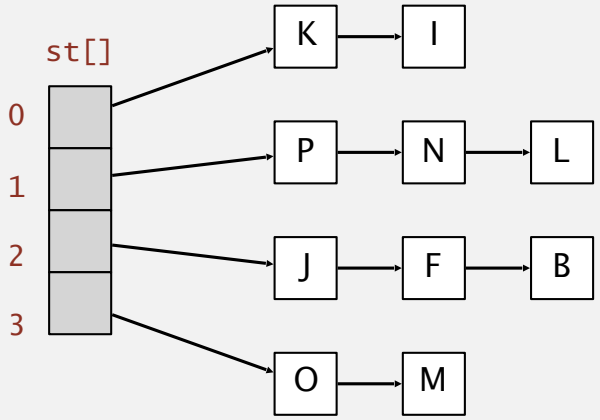
A. Easy: need only consider chain containing key.

١- نـدود عليه
2- بحذفه واشبك
الي بعده.

before deleting C



after deleting C



Q1. Draw the 11-entry hash table (length) 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.

↳ collisions نحل فيها الـ
نستخدم الـ linked list

h(12) = (3x12+5) mod 11 = 8

h(44) = (3x44+5) mod 11 = 5

h(13) = (3x13+5) mod 11 = 0

h(88) = (3x88+5) mod 11 = 5

h(23) = (3x23+5) mod 11 = 8

h(94) = (3x94+5) mod 11 = 1

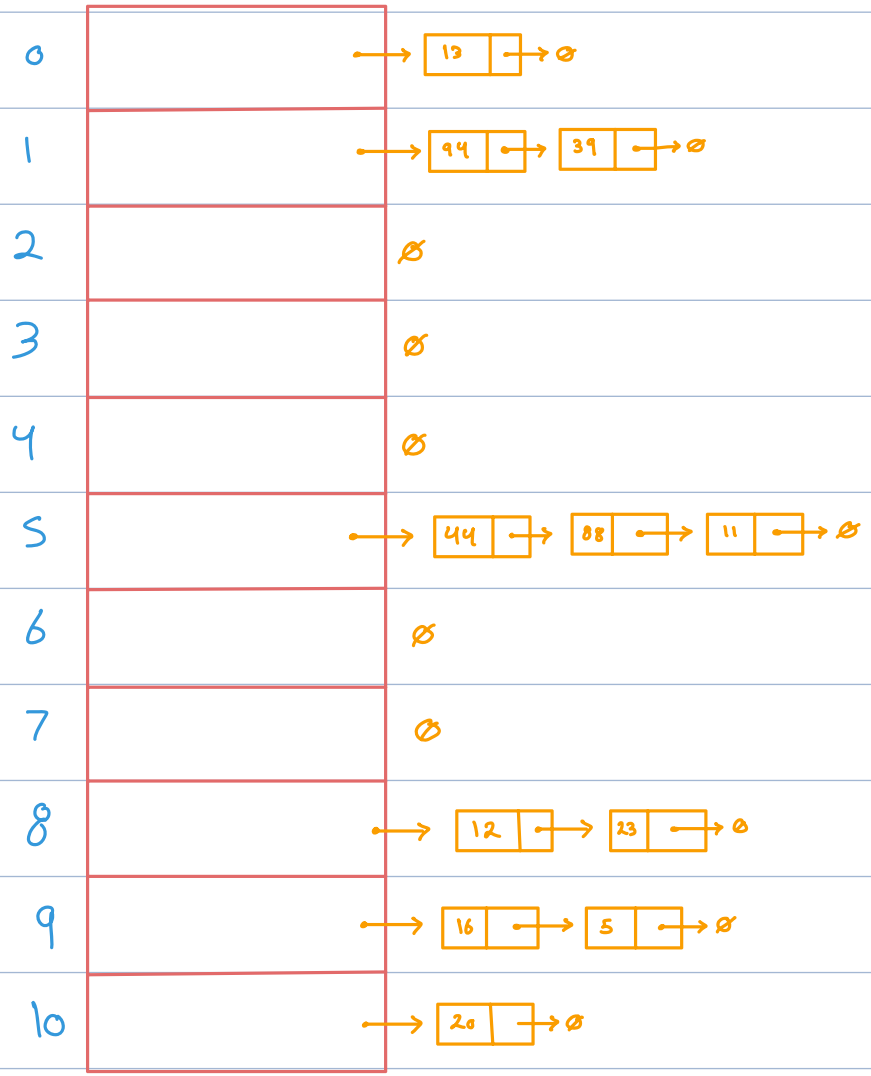
h(11) = (3x11+5) mod 11 = 5

h(39) = (3x39+5) mod 11 = 1

h(20) = (3x20+5) mod 11 = 10

h(16) = (3x16+5) mod 11 = 9

h(5) = (3x5+5) mod 11 = 9



Symbol table implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔	`compareTo()`
BST ما فيها ترتيب او نوعا ما مرتبة	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔	`compareTo()`
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✔	`compareTo()`
separate chaining	N	N	N ↓	3-5 *	3-5 *	3-5 *		`equals()` `hashCode()`

عدد الـ element ↓
لأن لازم ادخل element.

* under uniform hashing assumption

http://algs4.cs.princeton.edu

3.4 Hash Tables

- *hash functions*
- *separate chaining*
- *linear probing* ⟵ طريقه تخزين
- *context*

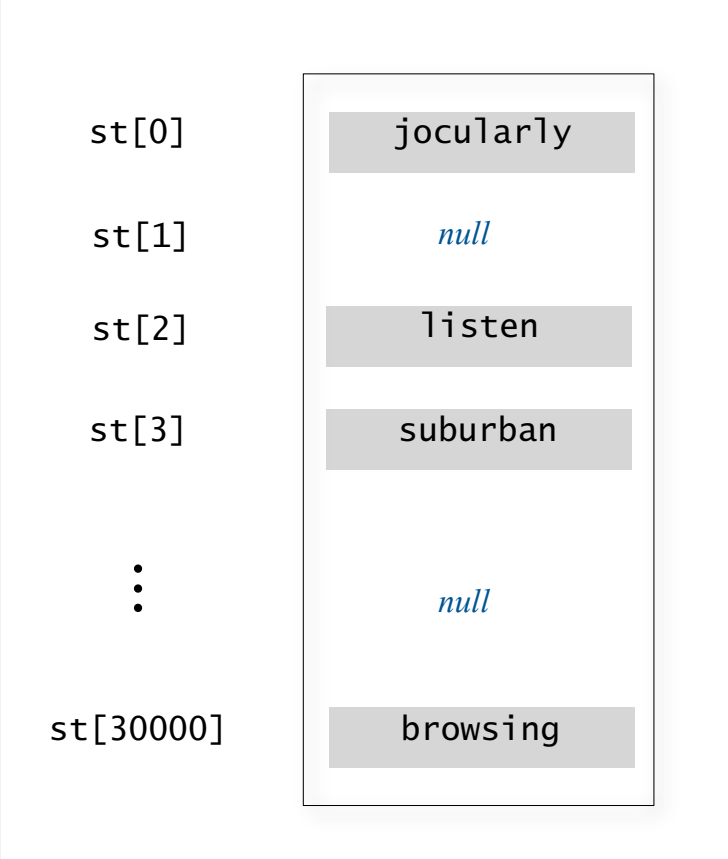
+ اذا عندي تصادم بتاكد
بروح للي بعدها اذا فاضية
بروح لـ element واذا موفاضيه اكمل
واذا وصلت نهايه الاري ما اوقف اعتبر
حذي (b %٢٢٠) توصلني للبداية .

Collision resolution: open addressing

Open addressing. [Amdahl-Boehme-Rocherster-Samuel, IBM 1953]

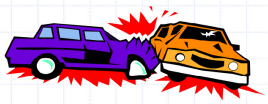
When a new key collides, find next empty slot, and put it there.

فكرتها : بروح للخانه اللي فيها مليانه
بروح اللي بعدها اشوف عليها
اللي ما احصل اول واحده فاضيه

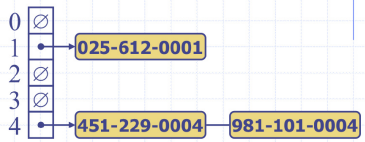


linear probing (M = 30001, N = 15000)

Collision Handling



- (1) Separate Chaining: let each cell in the table point to a linked list of entries that map there



  - Separate chaining is simple, but requires additional memory outside the table

: Colliding حلول الـ

★ Linear Probing: بضيف جاول مكان فاضي
ويرجع من البداية .

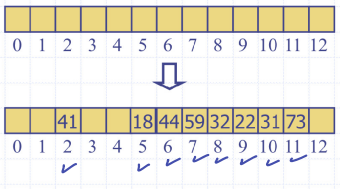
★ عدد الـ Probe : عدد المحاولات اللي حاول فيهم يلاقي مكان فاضي

Linear Probing

- Example:
  - $h(x) = x \bmod 13$ ← جمعها
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order. ← بالترتيب

13
26
39
52
65
78
...

1- 18 mod 13 = 18 - 1x13 = 5 ✓
2- 41 mod 13 = 41 - 3x13 = 2 ✓
3- 22 mod 13 = 22 - 1x13 = 9 ✓
4- 44 mod 13 = 44 - 3x13 = 5 ✓
5- 59 mod 13 = 59 - 4x13 = 7 ✓
6- 32 mod 13 = 32 - 2x13 = 6 ✓
7- 31 mod 13 = 31 - 2x13 = 5 ✓
8- 73 mod 13 = 73 - 5x13 = 8 ✓



wcs
collision: O(n) ← بيفك كل الـ array ويدور على اول مكان فاضي

bcs: O(1) ← يروح على حاول

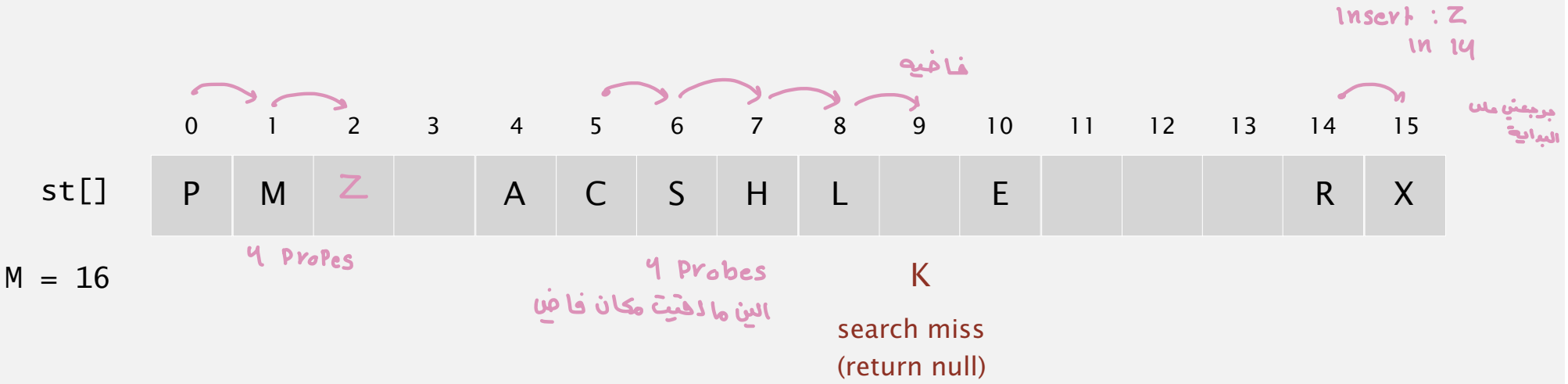
Linear-probing hash table demo

Hash. Map key to integer `i` between `0` and `M-1`.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

search K

hash(K) = 5



Q2. Do the same as above using Linear Probing.

h(12) = (3x12+5) mod 11 = 8

h(44) = (3x44+5) mod 11 = 5

h(13) = (3x13+5) mod 11 = 0

h(88) = (3x88+5) mod 11 = 5 collision

h(23) = (3x23+5) mod 11 = 8

h(94) = (3x94+5) mod 11 = 1

h(11) = (3x11+5) mod 11 = 5

h(39) = (3x39+5) mod 11 = 1

h(20) = (3x20+5) mod 11 = 10

h(16) = (3x16+5) mod 11 = 9

h(5) = (3x5+5) mod 11 = 9

# of collisions? 6

# of Probes ? 16

كم مره نزلت ادور مكان فاضي

علشان اعرف كم تصادم عندي

نزلت مرتين ادور مكان فاضي

Linear-probing hash table summary

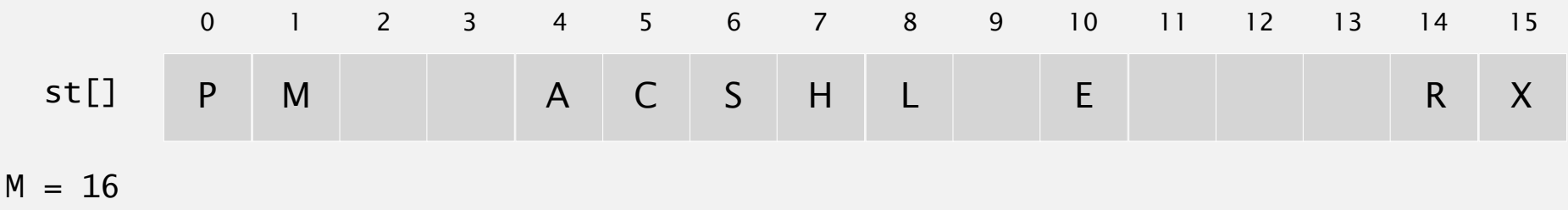
Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

Note. Array size `M` must be greater than number of key-value pairs `N`.

number of element



Linear-probing symbol table: Java implementation

```
public class LinearProbingHashST<Key, Value>
{
   private int M = 30001;
   private Value[] vals = (Value[]) new Object[M];
   private Key[]   keys = (Key[])   new Object[M];

   private int hash(Key key)             {  /* as before */   }

   private void put(Key key, Value val) {  /* next slide */  }

   public Value get(Key key)
   {
      for (int i = hash(key); keys[i] != null; i = (i+1) % M)
         if (key.equals(keys[i]))
             return vals[i];
      return null;
   }



}
```

array doubling and halving code omitted

Linear-probing symbol table: Java implementation

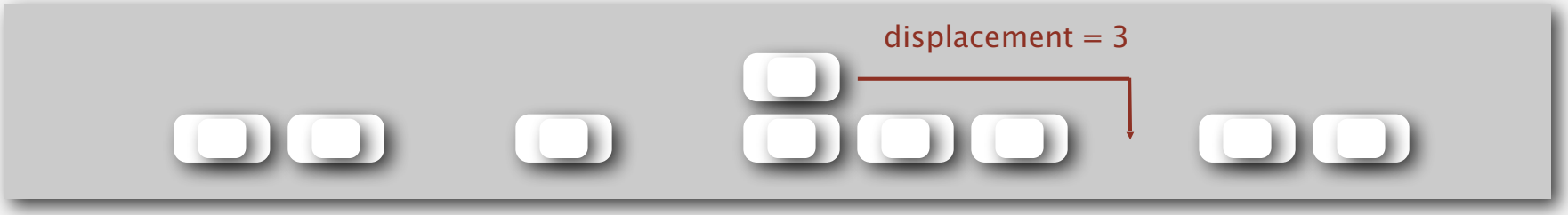
```
public class LinearProbingHashST<Key, Value>
{
   private int M = 30001;
   private Value[] vals = (Value[]) new Object[M];
   private Key[]   keys = (Key[])   new Object[M];

   private int hash(Key key)   { /* as before      */  }

   private Value get(Key key)  { /* previous slide */  }

   public void put(Key key, Value val)
   {
      int i;
      for (i = hash(key); keys[i] != null; i = (i+1) % M)
         if (keys[i].equals(key))
             break;
      keys[i] = key;
      vals[i] = val;
   }

}
```

Knuth's parking problem

Model. Cars arrive at one-way street with M parking spaces.

Each desires a random space i : if space i is taken, try $i+1, i+2,$ etc.

Q. What is mean displacement of a car?



Half-full. With $M/2$ cars, mean displacement is $\sim 3/2$.

Full. With M cars, mean displacement is $\sim \sqrt{\pi M/8}$.

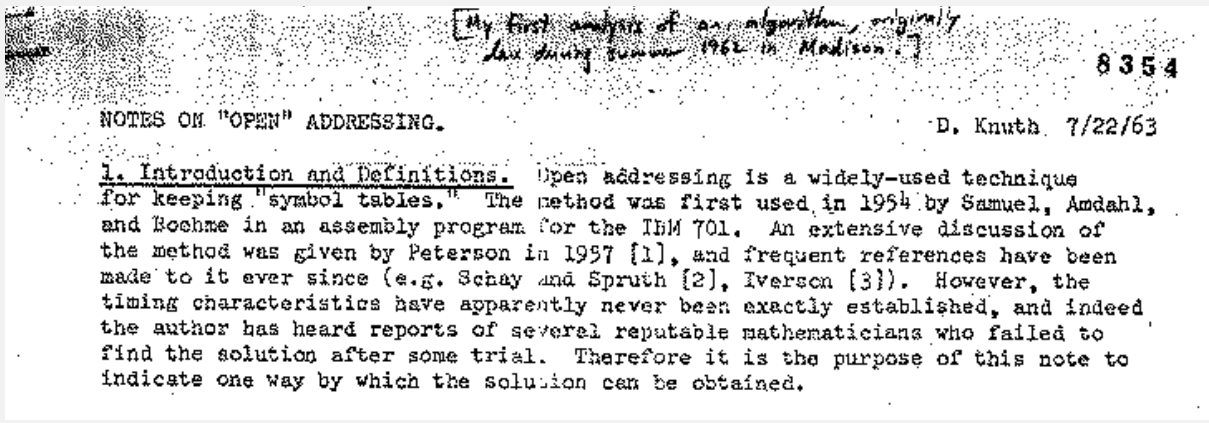
Analysis of linear probing

Proposition. Under uniform hashing assumption, the average # of probes in a linear probing hash table of size M that contains $N = \alpha M$ keys is:

$$\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \qquad \sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

search hit **search miss / insert**

Pf.



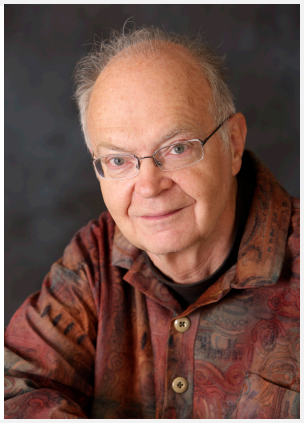
[My first analysis of an algorithm, originally done during summer 1962 in Madison.]

8354

NOTES ON "OPEN" ADDRESSING.

D. Knuth 7/22/63

1. Introduction and Definitions. Open addressing is a widely-used technique for keeping "symbol tables." The method was first used in 1954 by Samuel, Amdahl, and Boehme in an assembly program for the IBM 701. An extensive discussion of the method was given by Peterson in 1957 [1], and frequent references have been made to it ever since (e.g. Schay and Spruth [2], Iverson [3]). However, the timing characteristics have apparently never been exactly established, and indeed the author has heard reports of several reputable mathematicians who failed to find the solution after some trial. Therefore it is the purpose of this note to indicate one way by which the solution can be obtained.



Parameters.

- M too large $\Rightarrow$ too many empty array entries.
- M too small $\Rightarrow$ search time blows up.
- Typical choice: $\alpha = N / M \sim ½$. ← # probes for search hit is about 3/2; # probes for search miss is about 5/2

Resizing in a linear-probing hash table

Goal. Average length of list $N / M \leq ½$.

- Double size of array M when $N / M \geq ½$.
- Halve size of array M when $N / M \leq ⅛$.
- Need to rehash all keys when resizing.

before resizing

	0	1	2	3	4	5	6	7
keys[]		E	S			R	A	
vals[]		1	0			3	2	

after resizing

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]					A		S				E				R	
vals[]					2		0				1				3	

Deletion in a linear-probing hash table

Q. How to delete a key (and its associated value)?

A. Requires some care: can't just delete array entries.

1- Search for the key.

2- اذا كان لـ (null) ما ← اول مكان فاضي بيوقف
بيوصل للحرف اللي بدورة
بيوقف عند الخانه الفاضيه.

before deleting S

delete(S)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C	S	H	L		E				R	X
vals[]	10	9			8	4	0	5	11		12				3	7

doesn't work, e.g., if hash(H) = 4

after deleting S ?

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C		H	L		E				R	X
vals[]	10	9			8	4		5	11		12				3	7

ST implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔	`compareTo()`
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔	`compareTo()`
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✔	`compareTo()`
separate chaining	N	N	N	3-5 *	3-5 *	3-5 *		`equals()` `hashCode()`
linear probing	N	N	N	3-5 *	3-5 *	3-5 *		`equals()` `hashCode()`

Assume the array is full

* under uniform hashing assumption

http://algs4.cs.princeton.edu

3.4 Hash Tables

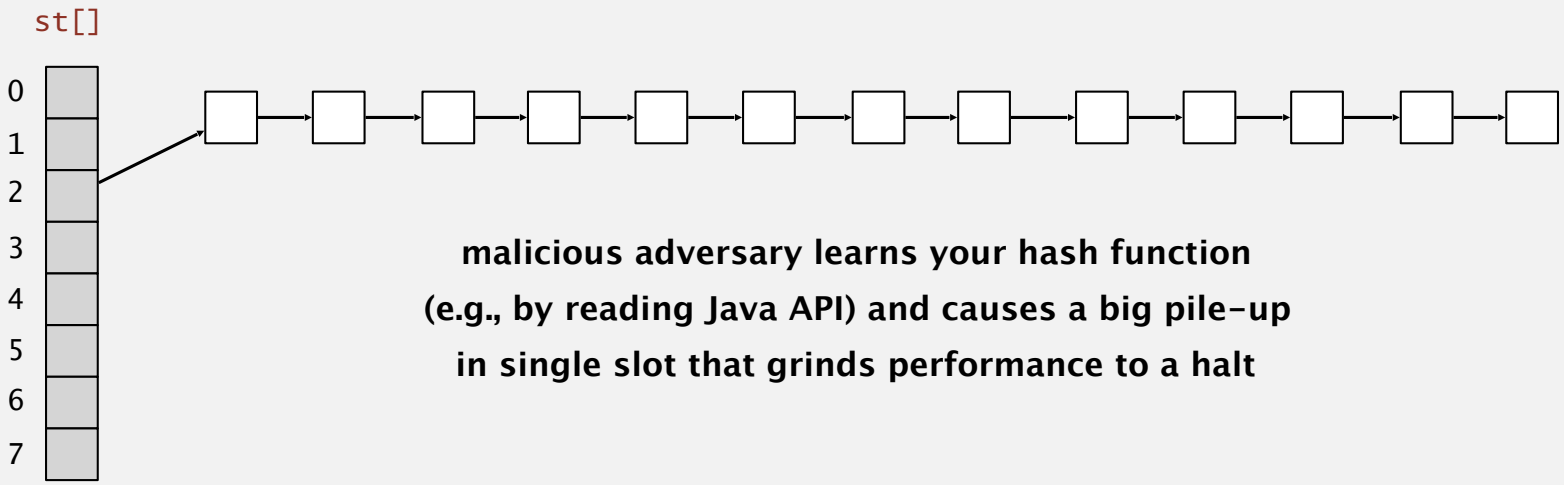
- *hash functions*
- *separate chaining*
- *linear probing*
- *context*

War story: algorithmic complexity attacks

Q. Is the uniform hashing assumption important in practice?

A. Obvious situations: aircraft control, nuclear reactor, pacemaker.

A. Surprising situations: denial-of-service attacks.



Real-world exploits. [Crosby-Wallach 2003]

- Bro server: send carefully chosen packets to DOS the server, using less bandwidth than a dial-up modem.
- Perl 5.8.0: insert carefully chosen strings into associative array.
- Linux 2.4.20 kernel: save files with carefully chosen names.

Algorithmic complexity attack on Java

Goal. Find family of strings with the same hash code.

Solution. The base-31 hash code is part of Java's string API.

key	hashCode()
"Aa"	2112
"BB"	2112

key	hashCode()
"AaAaAaAa"	-540425984
"AaAaAaBB"	-540425984
"AaAaBBAa"	-540425984
"AaAaBBBB"	-540425984
"AaBBAaAa"	-540425984
"AaBBAaBB"	-540425984
"AaBBBBAa"	-540425984
"AaBBBBBB"	-540425984

key	hashCode()
"BBAaAaAa"	-540425984
"BBAaAaBB"	-540425984
"BBAaBBAa"	-540425984
"BBAaBBBB"	-540425984
"BBBBAaAa"	-540425984
"BBBBAaBB"	-540425984
"BBBBBBAa"	-540425984
"BBBBBBBB"	-540425984

2^N strings of length 2N that hash to same value!

Diversion: one-way hash functions

One-way hash function. "Hard" to find a key that will hash to a desired value (or two keys that hash to same value).

Ex. MD4, MD5, SHA-0, SHA-1, SHA-2, WHIRLPOOL, RIPEMD-160, ...

known to be insecure

```
String password = args[0];
MessageDigest sha1 = MessageDigest.getInstance("SHA1");
byte[] bytes = sha1.digest(password);

/* prints bytes as hex string */
```

Applications. Digital fingerprint, message digest, storing passwords.

Caveat. Too expensive for use in ST implementations.

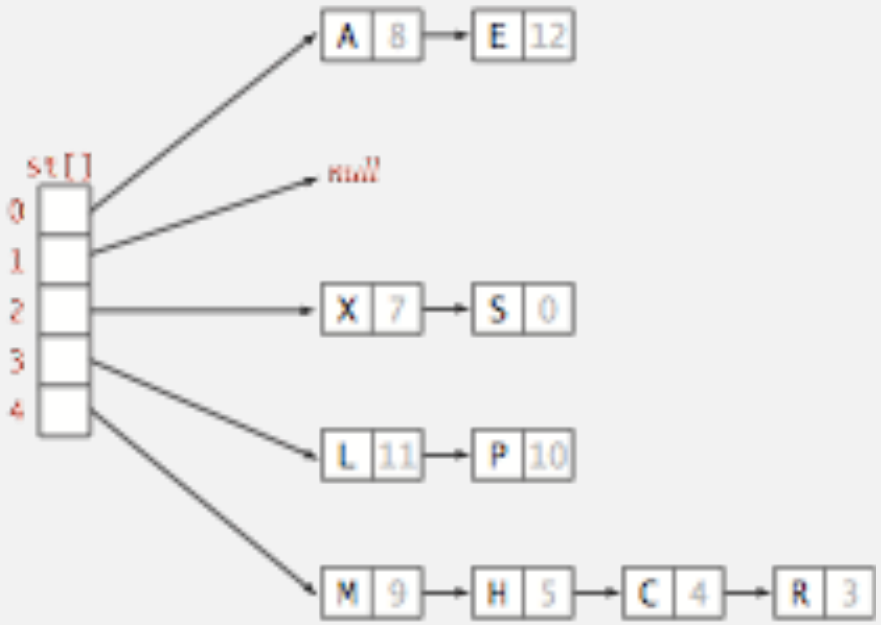
Separate chaining vs. linear probing

Separate chaining.

- Performance degrades gracefully.
- Clustering less sensitive to poorly-designed hash function.

Linear probing.

- Less wasted space.
- Better cache performance.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C	S	H	L		E				R	X
vals[]	10	9			8	4	0	5	11		12				3	7

Hashing: variations on the theme

Many improved versions have been studied.

Two-probe hashing. [separate-chaining variant]

- Hash to two positions, insert key in shorter of the two chains.
- Reduces expected length of the longest chain to $\log \log N$.

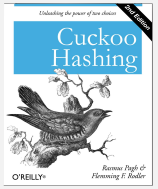
two function : primry → اساسية → index اضائع الـ
seconed : ثانوية → يعطيني كم مره
انتقل

Double hashing. [linear-probing variant]

- Use linear probing, but skip a variable amount, not just 1 each time.
- Effectively eliminates clustering.
- Can allow table to become nearly full.
- More difficult to implement delete.

Cuckoo hashing. [linear-probing variant]

- Hash key to two positions; insert key into either position; if occupied, reinsert displaced key into its alternative position (and recur).
- Constant worst-case time for search.



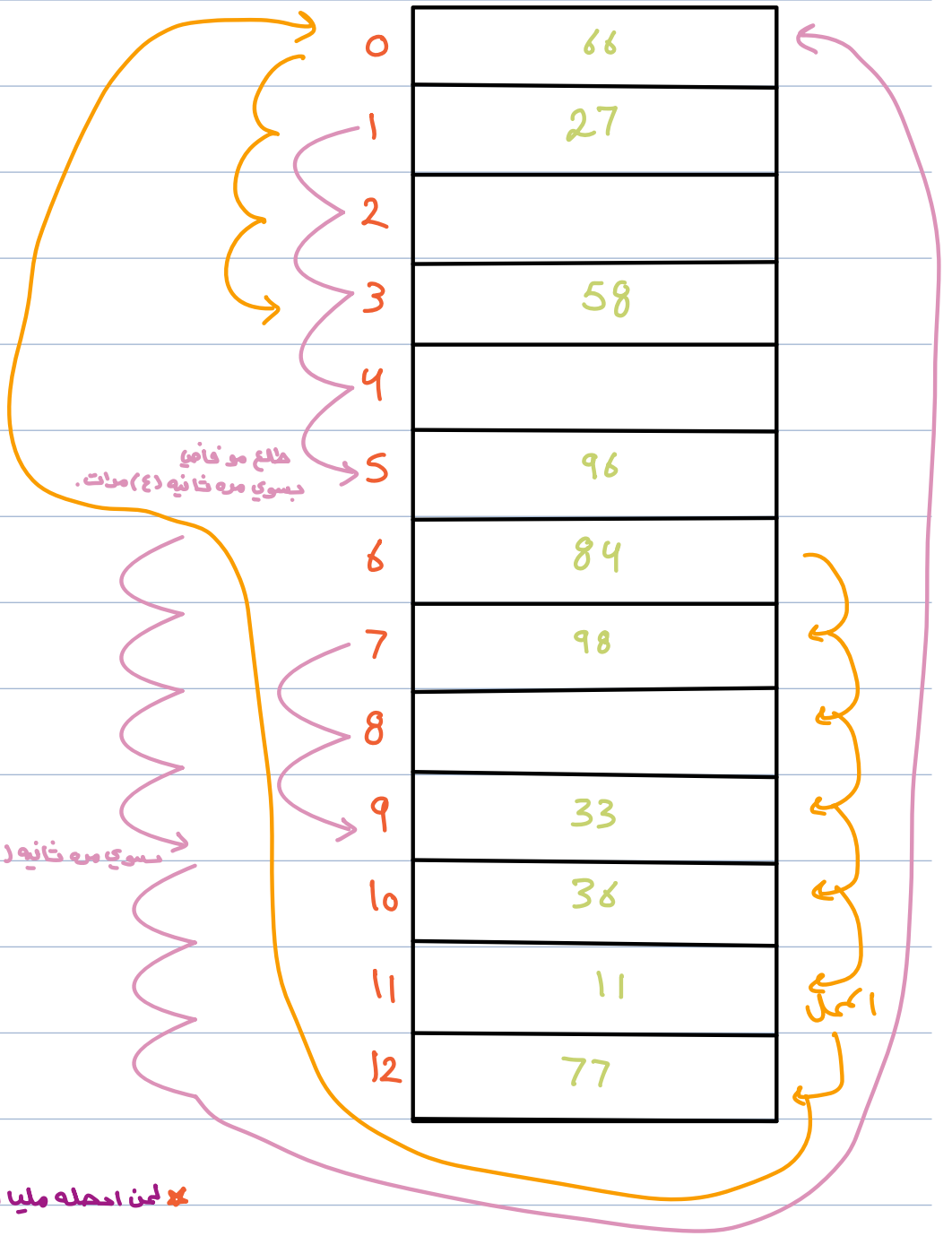
Use double hashing to enter the following keys in an array of size 13.

98, 33, 36, 96, 77, 11, 27, 84, 66, 58

Use primary hash function h(k) = k % 13.

Secondary hash function s(k) = 7 - k%7. → حال لو فيه تصادم اروح اللي بعدها .

	h(k) = k%13	S(k) = 7-k%7 → مانتتبعها اين مانحتاج لها
98	7	7
33	7	2
36	10	6
96	5	2
77	12	7
11	11	3
27	1	1
84	6	7
66	1	4
58	6	5



✖ لمن احطه مليان بمشي بنفس الزياده

مره ثانيه اين ما احصل خانه فاضيه.

بطريقه دكتور نجلاء.

Use double hashing to enter the following keys in an array of size 13.

98, 33, 36, 96, 77, 11, 27, 84, 66, 58

Use primary hash function h(k) = k % 13. → بستخدمها اول ما عندي تعارض

Secondary hash function s(k) = 7 - k%7. → لو عندي تعارض

	h(k) = k%13	S(k) = 7-k%7 → مانتتبعها الين مانحتاج لها
98	7	7
33	7	2
36	10	6
96	5	2
77	12	7
11	11	3
27	1	1
84	6	7
66	1 تعارض X	4 ← اخذ هنا
58	6 تعارض	5 تعارض → بناخذ اللي تحت اكتر

+ بناخذ الـ فهم prob اقل

0	
1	27
2	33
3	
4	66
5	96
6	84
7	98
8	58
9	
10	36
11	11
12	77

← لان بنزل مرتين عكس الـ (5) بنزل ثلاث مرات.

Double hashing.

Suppose you are asked to insert the following keys in a hash table of size 13 (خانه)

19, 41, 22, 5, 13, 44, 26, 57

Given hash functions:

H1(x) = x mode (الاول د) and H2(x) = 7- x mode 7 (الثاني د)

- Use linear Probing to solve collision using H1(x)
- Use Chaining to solve collision using H1(x)
- Use Double hash where h(x) = H1(x) + jH2(x)
- Count the number of collisions and number of probes for each method
- Compare the methods. Which one has better performance cording the # of collisions and probes.

Soultion of use linear Probing to solve collision using H1(x):

Counts of Probes

1- 19 = 19 ÷ 13 = 6 — 0

2- 41 = 41 ÷ 13 = 2 — 0

3- 22 = 22 ÷ 13 = 9 — 0

4- 5 = 5 ÷ 13 = 5 — 0

5- 7 = 7 ÷ 13 = 7 — 0

6- 13 = 13 ÷ 13 = 0 — 0

7- 44 = 44 ÷ 13 = 5 → هنا لان اوديكي محجوزه بتروح لاي بعدها اين ما تلاقي مكان فاضي وتدخله — 3

8- 26 = 26 ÷ 13 = 0 — 1

9- 57 = 57 ÷ 13 = 5 — 5

0	13
1	26
2	41
3	
4	
5	5
6	19
7	7
8	44
9	22
10	57
11	
12	

عدد التصادمات

Counts of collisions: 3 item

Counts of Probes = 9 Probes

Soultion of use linear Probing to Solve collision using H1(x):

	Counts of Probes	
1- 19 = 19 ÷ 13 = 6	0	
2- 41 = 41 ÷ 13 = 2	0	7- 44 = 44 ÷ 13 = 5 → هنا لان اوديدي محجوزه بتروح لاي بعدها اللي ما تلاقي مكان فاضي وتحطه — 3
3- 22 = 22 ÷ 13 = 9	0	8- 26 = 26 ÷ 13 = 0 — 1
4- 5 = 5 ÷ 13 = 5	0	9- 57 = 57 ÷ 13 = 5 — 5
5- 7 = 7 ÷ 13 = 7	0	
6- 13 = 13 ÷ 13 = 0	0	

عدد التحامات

Counts of collisions: 3 item

No of Probes = 0

0	13 → 26
1	26
2	41
3	
4	
5	5 → 44 → 57
6	19
7	7
8	44
9	22
10	57
11	
12	

Soultion of use linear Probing to Solve collision using $h(x) = H1(x) + jH2(x)$

H1(x)	H2(x) = 7 - x mod 7	H(x) = h1(x) + j h2(x)	
13	1- 19 = 19 ÷ 7 = 2	6 + (0) x 2 = 6	0
26	2- 41 = 41 ÷ 7 = 1	2 + (0) x 1 = 2	0
41	3- 22 = 22 ÷ 7 = 6	9 + (0) x 6 = 9	0
	4- 5 = 5 ÷ 7 = 2	5 + (0) x 2 = 5	0
	5- 7 = 7 ÷ 7 = 7	7 + (0) x 7 = 7	0
5	6- 13 = 13 ÷ 7 = 1	0 + (0) x 1 = 0	0
19	7- 44 = 44 ÷ 7 = 5	5 + (0) x 5 = 5 5 + (1) x 5 = 10	1
7	8- 26 = 26 ÷ 7 = 2	0 + (0) x 2 = 0 → لا مجوزة 0 + (2) x 2 = 4	2
44	9- 57 = 57 ÷ 7 = 6	5 + (0) x 6 = 5 5 + (1) x 6 = 11	1
22			
57			

0	13
1	
2	41
3	
4	26
5	5
6	19
7	7
8	
9	22
10	44

عدد التصادمات

Counts of collisions: 3 Item

number of Probes = 4 Probes

11	57
12	