Hash Tables

Gianpiero Cabodi e Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino

Hash Tables

- ADT with
  - Memory usage equal to O(|K|)
  - Average access time equal to O(1)
- The hash function transform the search key into a table index
- The hash table cannot be perfect, a collision may always happen
- Used to insert, search, delete, not to order or select a key

Hash Function

- The hash table
  - Has size M
  - Stores |K| elements
  - |K|<<|U|
- The hash table has addresses in the range [0 … M-1]

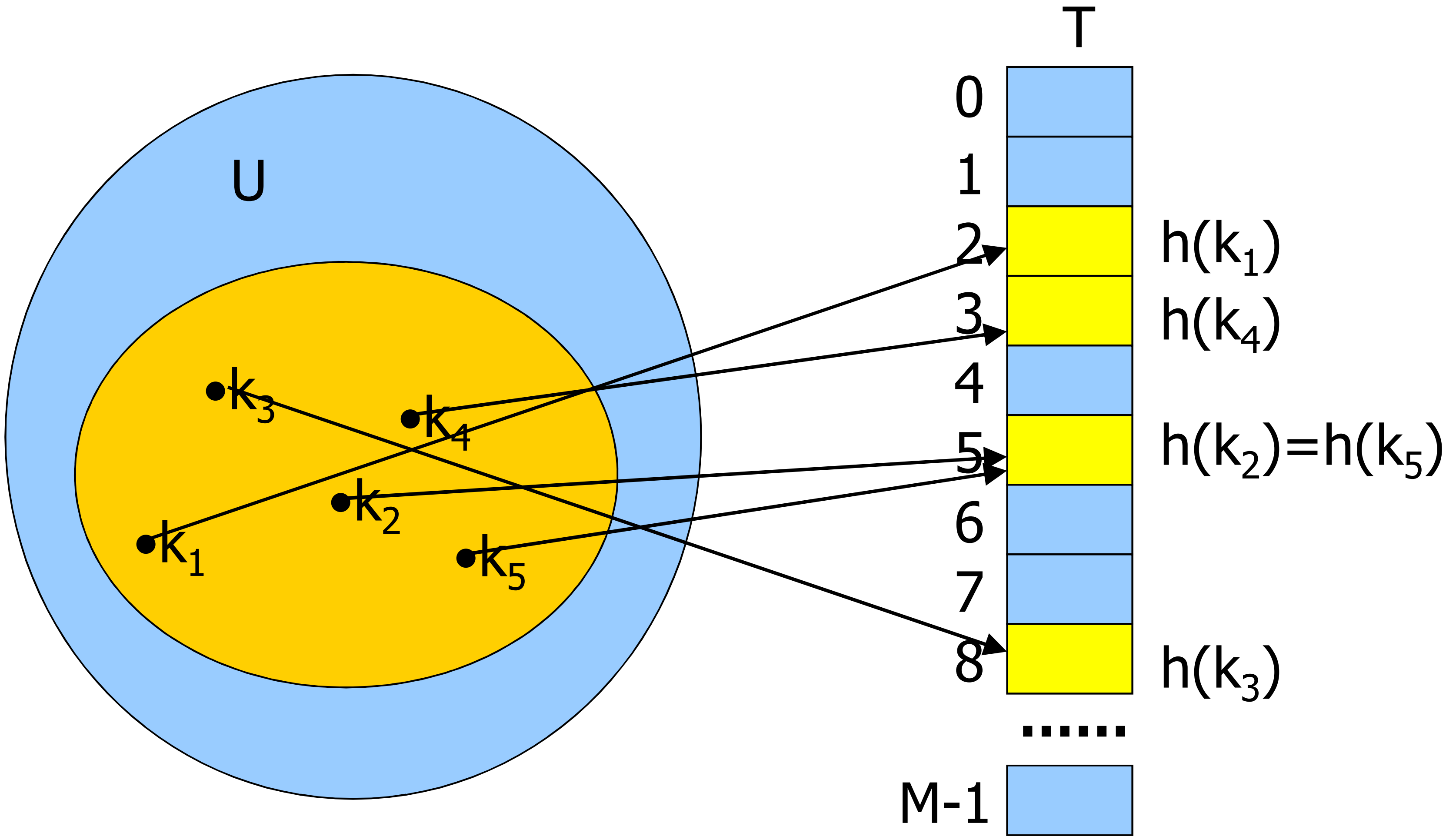
Hash Function

- The hash function h creates a correspondence between a key k and a table address $h(k)$

  $$h: U \rightarrow \{ 0, 1, ..., M-1 \}$$

- Each element is stored at the address h(k) given its key k
  - Pay attention to collision handling !

Hash Function



Designing a hash function

- Ideal Function
  - Simple uniform hashing
- If the k keys are equiprobable, then the h(k) values must be equiprobable
- Practically, the k keys are not equiprobable, as they have a correlation
  - Keys k_i and k_j are not uncorrelated

Designing a hash function

- To make the h(k) values equiprobable it is necessary to
  - Make $h(k_i)$ uncorrelated from $h(k_j)$
  - “Amplify” differences
  - Uncorrelate h(k) from k
- Distribute h(k) in a uniform way
  - Use all key bits
  - Multiply for a prime number

The Multiplication Method

- If keys are floating point numbers in a pre-defined range ($s \le k \le t$)
  - $h(k) = \lfloor (k - s) / (t - s) \cdot M \rfloor$

floor

```
int hash(float k, int M) {
  return ((k-s)/(t-s))*M
}
```

- Example
  - $M = 97$, $s = 0$, $t = 1$
  - $k = 0.513870656$
  - $h(k) = \lfloor (0.513870656 - 0) / (1 - 0) \cdot 97 \rfloor = 49$

The Module Method

- If keys are integer numbers of w bits and M is a prime number
  - h(k) = k % M

M prime number allows
using only the last n bits of k, if $M = 2^n$
using only the last n decimal digits of k, if $M = 10^n$

- Example
  - M = 19
  - k = 31
  - h(k) = 31 % 19 = 12

```
int hash(int k, int M){
  return (k%M);
}
```

The Multiplication-Module Method

- If keys are integer numbers
- Given a constant value $0<A<1$
  - $A = \phi = (\sqrt{5} - 1) / 2 = 0.6180339887$
- Then the hash function can be computed as
  - $h(k) = \lfloor k \cdot A \rfloor \% M$

The Modular Method

- If keys are short alphanumeric strings it is possible to convert them into integers
  - Each interger is obtained from a polinomial evaluation in a given base of the original string
  - h(k) = k % M
- Example
  - K = "now"
  - $h(k) = (\text{'n'} \cdot 128^2 + \text{'o'} \cdot 128 + \text{'w'}) \% 19$

    $= (110 \cdot 128^2 + 111 \cdot 128 + 119) \% 19$

    $= 1816567 \% 19$

    $= 15$

The Modular Method

- If keys are long alphanumeric strings k cannot be represented on a reasonable number of bits
- It is possible to use the Horner's method to rule-out M multiples after each step, instead of doing that after the application of the modular technique
  - $K = p_7x^7+p_6x^6 + p_5x^5+p_4x^4+p_3x^3+p_2x^2+p_1x+p_0$
    $= ((((((p_7\cdot x+p_6)\cdot x+p_5)\cdot x+p_4)\cdot x+p_3)\cdot x+p_2)\cdot x+p_1)\cdot x+p_0$
    $= ((((((p_7\%M)\cdot x+p_6)\%M)\cdot x+p_5)\cdot x)\%M \ldots$

The Modular Method

```
int hash (char *v, int M){
  int h = 0, base = 128;
  for (; *v != '\0'; v++)
    h = (base * h + *v) % M;
  return h;
}
```

- Example
  - K = ”averylongkey”
  - with a 128 base (ASCII)
  - k = $97*128^{11}+118*128^{10}+101*128^{9}+114*128^{8}+121*128^{7}+108*128^{6}+111*128^{5}+110*128^{4}+103*128^{3}+107*128^{2}+101*128^{1}+121*128^{0}$ =

    $(((((((((((97\cdot128+118)\cdot128+101)\cdot128+114)\cdot128+121)\cdot128+108)\cdot128+111)\cdot128+110)\cdot128+103)\cdot128+107)\cdot128+101)\cdot128+121$ =

    ((((((((((97%M)·128+118)%M)·128+114)%M)·128+121)% …

The Modular Method

- Notice that even for ASCII strings, 128 is not used as a base
- Instead it is used
  - A prime number (for example 127)
  - A random number different for each digit of the key (universal hashing)
- The target being to obtain a uniform distribution (collision probability for 2 different keys equal to 1/M)

The Modular Method

Hash Function for string keys with a prime base:

```
int hash (char *v, int M) {
  int h = 0, base = 127;
  for (; *v != '\0'; v++)
    h = (base * h + *v) % M;
  return h;
}
```

Hash function for string keys with universal hashing:

```
int hashU( char *v, int M) {
  int h, a = 31415, b = 27183;
  for ( h = 0; *v != '\0'; v++, a = a*b % (M-1))
    h = (a*h + *v) % M;
  return h;
}
```

Collisions

- A collision happens when
  - $h(k_i)=h(k_j)$ with $k_i \neq k_j$
- Collisions are inevitable, then it is necessary to
  - Minimize their number (good hash function)
  - Dealing with them
- Collisions can be dealt with
  - Linear chaining
  - Open addressing

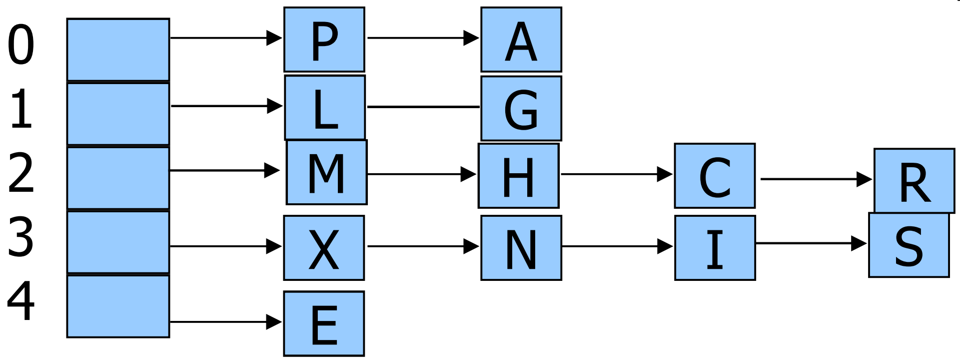
Linear Chaining

- More elements can be stored in the same table location T, i.e., each element points to a linked list
- Operations
  - Insert on the list head
  - List search
  - Delete from the list
- Table size M
  - The smallest prime M $\geq$ max. number of keys / 5 (or 10) such that the average list length would be 5 (or 10)

Example: Linear Chaining

A S E R C H I N G X M P L
h(k) = 0 3 4 2 2 2 3 3 1 3 2 0 1

```
M = 5;
int hash (Key k, int M) {
  int h = 0, base = 127;
  for (; *k != '\0'; k++)
    h = (base * h + *k) % M;
  return h;
}
```



Complexity

- With non-ordered lists
  - N =|K| = number of stored elements
  - M = size of the hash table
- Simple Uniform Hashing
  - h(k) has the same probability to generate M output values
- Definition
  - Load factor $\alpha = N/M$ (>, = o < 1)

Complexity

- Insert
  - $T(n) = O(1)$
- Search
  - Worst case $T(n) = \Theta(N)$
  - Average case $T(n) = O(1+\alpha)$
- Delete
  - As the search

Open addressing

$N \leq M$

$\alpha \leq 1$

- Each cell table T can store a single element
- All elements are stored in T
- Once there is a collision it is necessary to look-for an empty cell with probing
  - Generate a cell permutation, i.e, an order to search for an empty cell
  - The same order has to be used to insert and to search a key

Probing Functions

- There are several ways to perform probing
  - Linear probing
  - Quadratic probing
  - Double hashing
- A problem with open addressing is **clustering**, that is, the presence of clusters of contiguous full cells.

Linear Probing

- Given a key k
  - $h'(k) = (h(k) + i)\%M$
  - i is the attempt counter (initially 0)
- Set i=0
  - Compute h(k), then h'(k)
  - If free, insert the key
  - Otherwise increase i and repeat until an empty cell is found

Quadratic Probing

- Given a key k
  - $h`(k) = (h(k) + c_1 i + c_2 i^2)\%M$
  - i is the attempt counter (initially 0)
- Set i=0
  - Compute h(k), then h'(k)
  - If free, insert the key
  - Otherwise increase i and repeat until an empty cell is found

Quadratic probing

- Constants c_1 and c_2 must be selected carefully
- They must guarantee that h'(k) assumes distinct values for $1 \leq i \leq (M-1)/2$
  - If M = 2K, select c1 = c2 = ½ to generate all indexes between 0 and M-1
  - If M is prime and $\alpha < ½$ the following values
    - $c_1 = c_2 = ½$
    - $c_1 = c_2 = 1$
    - $c_1 = 0, c_2 = 1$

Double Hashing

- Given a key k
  - $h`(k) = (h_1(k) + i \cdot h_2(k))\%M$
  - i is the attempt counter (initially 0)
- Set i=0
  - Compute $h_1(k)$, then $h'(k)$
  - If free, insert the key
  - Otherwise increase i, compute $h_2(k)$, and repeat until an empty cell is found

Double Hashing

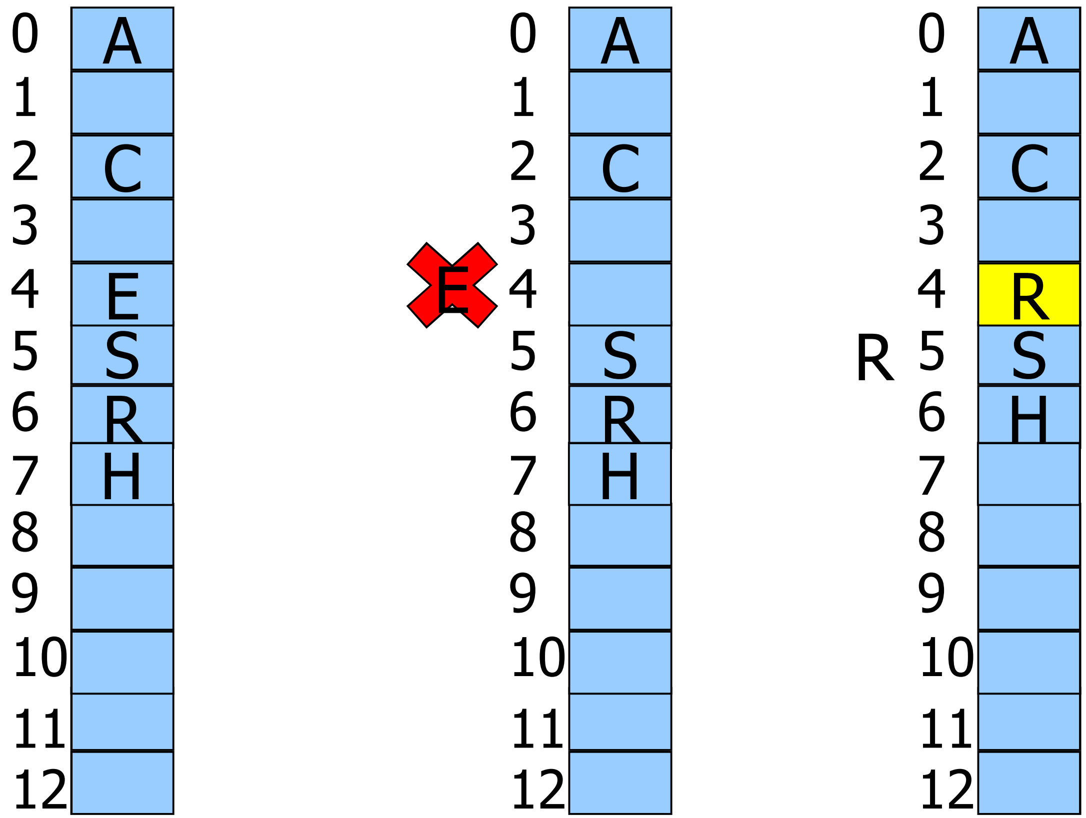
- It must be true that the new value of h'(k) differ from the previous one otherwise we enter an infinite loop
- To avoid this
  - h_2 should never return 0
  - h_2%M should never return 0
- Example
  - $h_1(k) = k \% M$ and M prime
  - $h_2(k) = 1 + k\%97$
  - $h_2(k)$ never returns 0 and h_2%M never returns 0 if M > 97.

Probing and Delete

- With probing delete a key is a complex operation which stops collision chains
- L'open addressing is used only when it is not necessary to delete keys
- Solution
  - Substitute the deleted key with a sentinel key that is considered as a full element during search operations and an empty element during insertion operations
  - Re-insert cluster keys within the deleted key

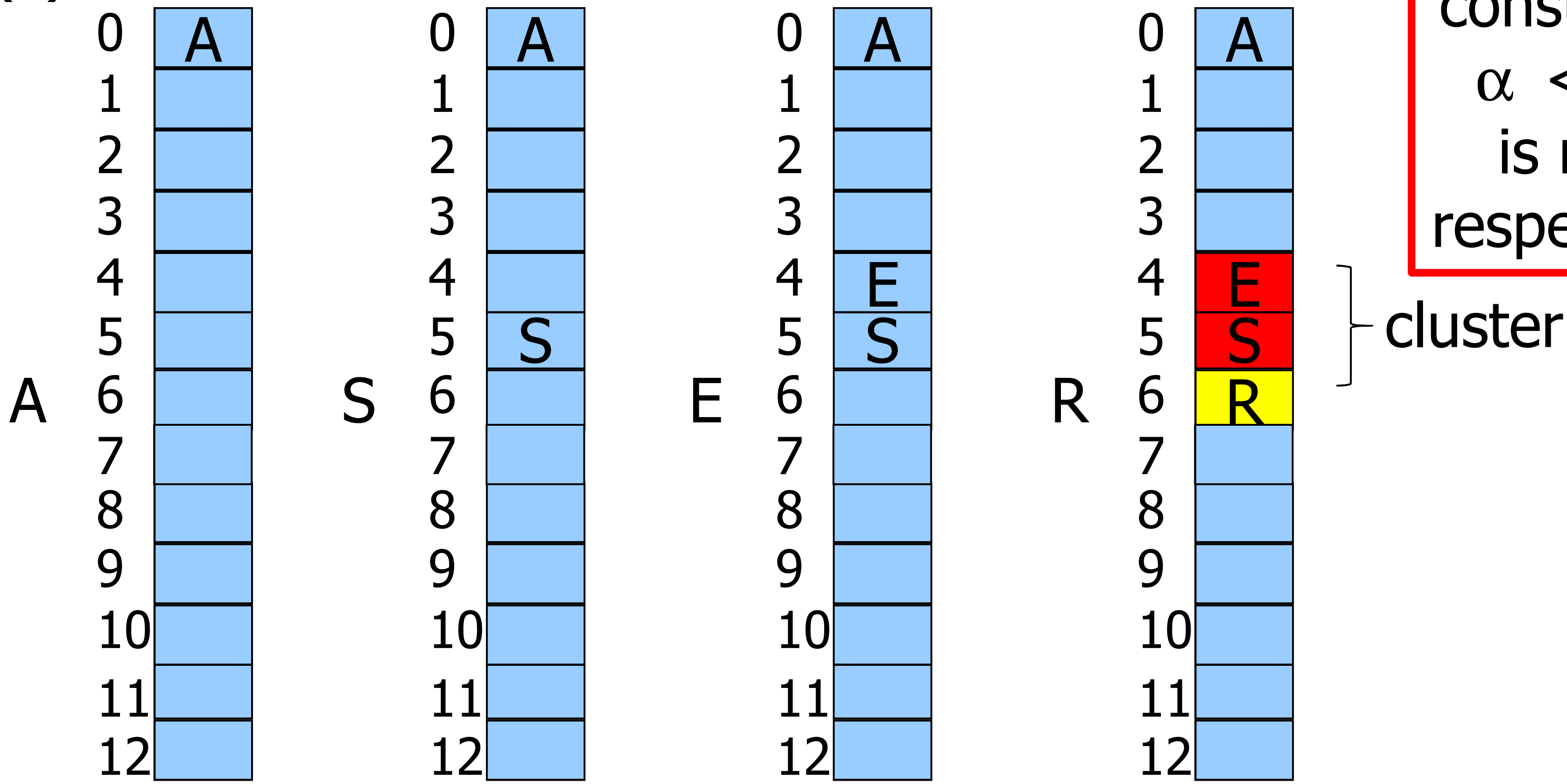
Example: Delete with Probing

Delete E remembering that there was a collision between E and R



Example: Linear Probing

	A	S	E	R	C	H	I	N	G	X	M	P
h(k) =	0	5	4	4	2	7	8	0	6	10	12	2

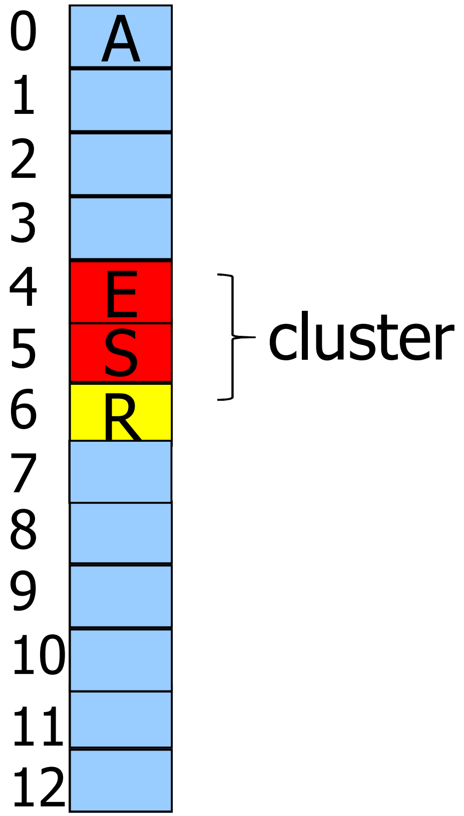


The constraint $\alpha < \frac{1}{2}$ is not respected

Example: Linear Probing

A S E R C H I N G X M P

h(k) = 0 5 4 4 2 7 8 0 6 10 12 2



i = h('R') = 82 % 13 = 4 collision

i = (4+1) % 13 = 5 collision

i = (5+1) % 13 = 6

Example: Linear Probing

	A	S	E	R	C	H	I	N	G	X	M	P
h(k) =	0	5	4	4	2	7	8	0	6	10	12	2

	C	H	I	N
0	A	A	A	A
1				N
2	C	C	C	C
3				
4	E	E	E	E
5	S	S	S	S
6	R	R	R	R
7		H	H	H
8			I	I
9				
10				
11				
12				

Example: Linear Probing

	A	S	E	R	C	H	I	N	G	X	M	P
h(k) =	0	5	4	4	2	7	8	0	6	10	12	2

	0	A
	1	N
	2	C
	3	
	4	E
	5	S
N	6	R
	7	H
	8	I
	9	
	10	
	11	
	12	

i = h('N') = 78 % 13 = 0 collision

i = (0+1) % 13 = 1

Example: Linear Probing

	A	S	E	R	C	H	I	N	G	X	M	P
h(k) =	0	5	4	4	2	7	8	0	6	10	12	2

	0	A
	1	N
	2	C
	3	
	4	E
	5	S
G	6	R
	7	H
	8	I
	9	G
	10	
	11	
	12	

i = h('G') = 71 % 13 = 6 collision

i = (6+1) % 13 = 7 collision

i = (7+1) % 13 = 8 collision

i = (8+1) % 13 = 9

Example: Linear Probing

A S E R C H I N G X M P

h(k) = 0 5 4 4 2 7 8 0 6 10 12 2

		M	P
0	A	A	A
1	N	N	N
2	C	C	C
3			P
4	E	E	E
5	S	S	S
6	R	R	R
7	H	H	H
8	I	I	I
9	G	G	G
10	X	X	X
11			
12		M	M

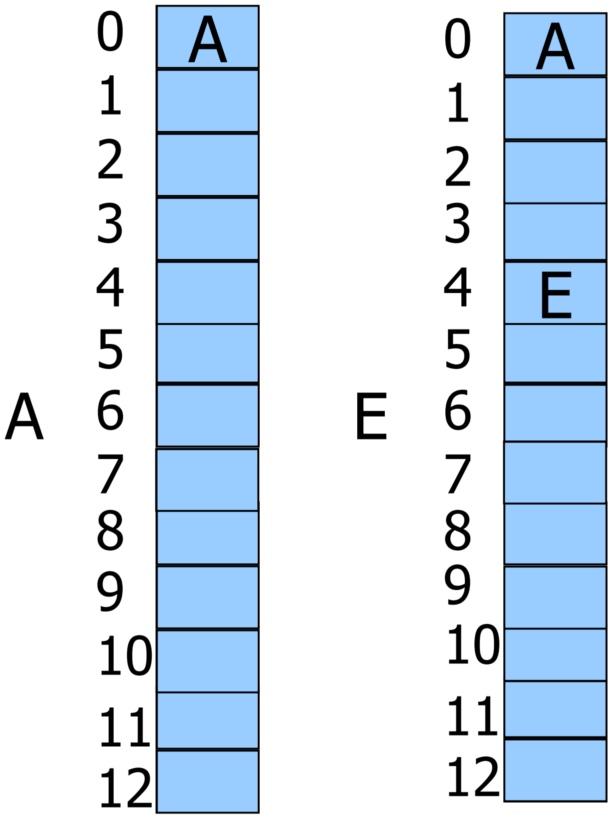
collision

$i = h('P') = 80 \% 13 = 2$

$i = (2+1) \% 13 = 3$

Example: Quadratic Probing

	A	E	R	C	N	P
h(k) =	0	4	4	2	0	2



Quadratic probing function

$c_1=1$ $c_2=1$

$i + i^2$

```
M = 13;
int hash (Key k, int M) {
  int h = 0, base = 127;
  for (; *k != '\0'; k++)
    h = (base * h + *k) % M;
  return h;
}
```

$\boxed{\alpha = 6/13 < ½}$

Example: Quadratic Probing

	A	E	R	C	N	P
h(k) =	0	4	4	2	0	2

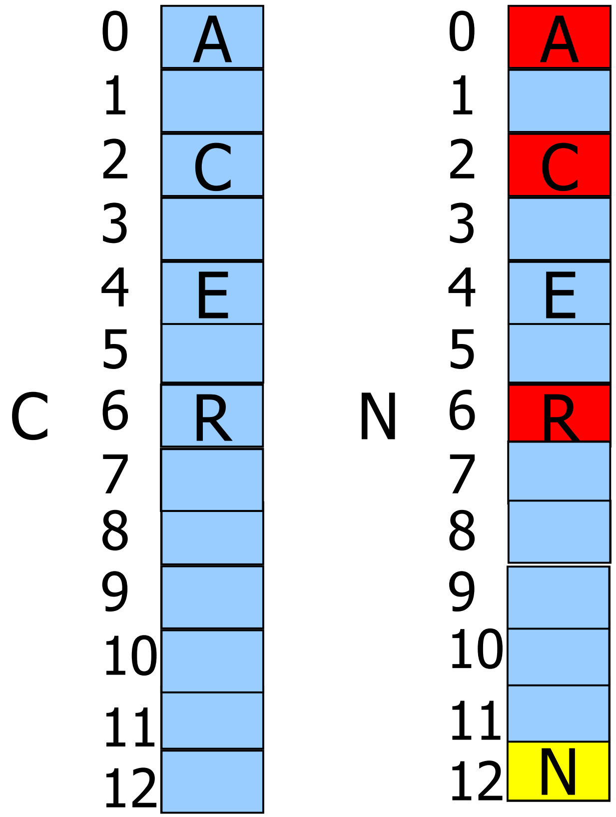
	0	A
	1	
	2	
	3	
	4	E
	5	
R	6	R
	7	
	8	
	9	
	10	
	11	
	12	

start = h('R') = 82 % 13 = 4 collision

index = $(4+1+1^2)$ % 13 = 6

Example: Quadratic Probing

	A	E	R	C	N	P
h(k) =	0	4	4	2	0	2



start = h('N') = 78 % 13 = 0 collision

index = $(0+1+1^2)$ % 13 = 2 collision

index = $(0+2+2^2)$ % 13 = 6 collision

index = $(0+3+3^2)$ % 13 = 12

Example: Quadratic Probing

	A	E	R	C	N	P
h(k) =	0	4	4	2	0	2

	0	A
	1	
	2	C
	3	
	4	E
	5	
P	6	R
	7	
	8	P
	9	
	10	
	11	
	12	N

start = h('P') = 80 % 13 = 2 collision

index = $(2+1+1^2)$ % 13 = 4 collision

index = $(2+2+2^2)$ % 13 = 8

Example: Double Hashing

	A	S	E	R	C	H	I	N	G	X	M	P
$h_1(k)=$	0	5	4	4	2	7	8	0	6	10	12	2

	A	S	E
0	A	A	A
1			
2			
3			
4			E
5		S	S
6			
7			
8			
9			
10			
11			
12			

The constraint
$\alpha < ½$
is not respected

```
M = 13;
int hash (Key k, int M) {
  int h = 0, base = 127;
  for (; *k != '\0'; k++)
    h = (base * h + *k) % M;
  return h;
}
```

Example: Double Hashing

	A	S	E	R	C	H	I	N	G	X	M	P
$h_1(k)=$	0	5	4	4	2	7	8	0	6	10	12	2

	0	A
	1	
	2	
	3	
	4	E
	5	S
R	6	
	7	
	8	
	9	R
	10	
	11	
	12	

i = h('R') = 82 % 13 = 4 collision

j = (82 % 97 +1) % 13) = 5

i = (4 + 5) % 13 = 9

Example: Double Hashing

	A	S	E	R	C	H	I	N	G	X	M	P
$h_1(k)=$	0	5	4	4	2	7	8	0	6	10	12	2

	C	H	I
0	A	A	A
1			
2	C	C	C
3			
4	E	E	E
5	S	S	S
6			
7		H	H
8			I
9	R	R	R
10			
11			
12			

Example: Double Hashing

	A	S	E	R	C	H	I	N	G	X	M	P
$h_1(k)=$	0	5	4	4	2	7	8	0	6	10	12	2

	0	A
	1	N
	2	C
	3	
	4	E
	5	S
N	6	
	7	H
	8	I
	9	R
	10	
	11	
	12	

i = h('N') = 78 % 13 = 0 collision

j = (78 % 97 +1) % 13) = 1

i = (0 + 1) % 13 = 1

Example: Double Hashing

	A	S	E	R	C	H	I	N	G	X	M	P
$h_1(k)=$	0	5	4	4	2	7	8	0	6	10	12	2

	0	A		0	A		0	A
	1	N		1	N		1	N
	2	C		2	C		2	C
	3			3			3	
	4	E		4	E		4	E
	5	S		5	S		5	S
G	6	G	X	6	G	M	6	G
	7	H		7	H		7	H
	8	I		8	I		8	I
	9	R		9	R		9	R
	10			10	X		10	X
	11			11			11	
	12			12			12	M

Example: Double Hashing

	A	S	E	R	C	H	I	N	G	X	M	P
$h_1(k)=$	0	5	4	4	2	7	8	0	6	10	12	2

	0	A
	1	N
	2	C
	3	
	4	E
	5	S
P	6	G
	7	H
	8	I
	9	R
	10	X
	11	P
	12	M

i = h('P') = 80 % 13 = 2 collision

j = (80 % 97 +1) % 13) = 3

i = (2 + 3) % 13 = 5 collision

i = (5 + 3) % 13 = 8 collision

i = (8 + 3) % 13 = 11

Tree vs Hash Table Comparison

- Hash Table
  - Easier to implement
  - Unique solution with keys without an ordering relation
  - Faster for simple keys
- Trees (BST and variants)
  - Better average performances (balanced trees)
  - Allow operations on keys with an ordering relation