Le tabelle di hash

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

Tabelle di hash

Finora gli algoritmi di ricerca si erano basati sul confronto.

Eccezione: tabelle ad accesso diretto dove la chiave $k \in U = \{0, 1, \ldots, card(U)-1\}$ funge da indice di un array st[0, 1, …, card(U)-1]:

Limiti delle tabelle ad accesso diretto:

- |U| grande (vettore st non allocabile)
- |K| << |U| (spreco di memoria).

Tabella di hash:

ADT con occupazione di spazio O(|K|) e tempo medio di accesso O(1).

La funzione di hash trasforma la chiave di ricerca in un indice della tabella.

La funzione di hash non può essere perfetta

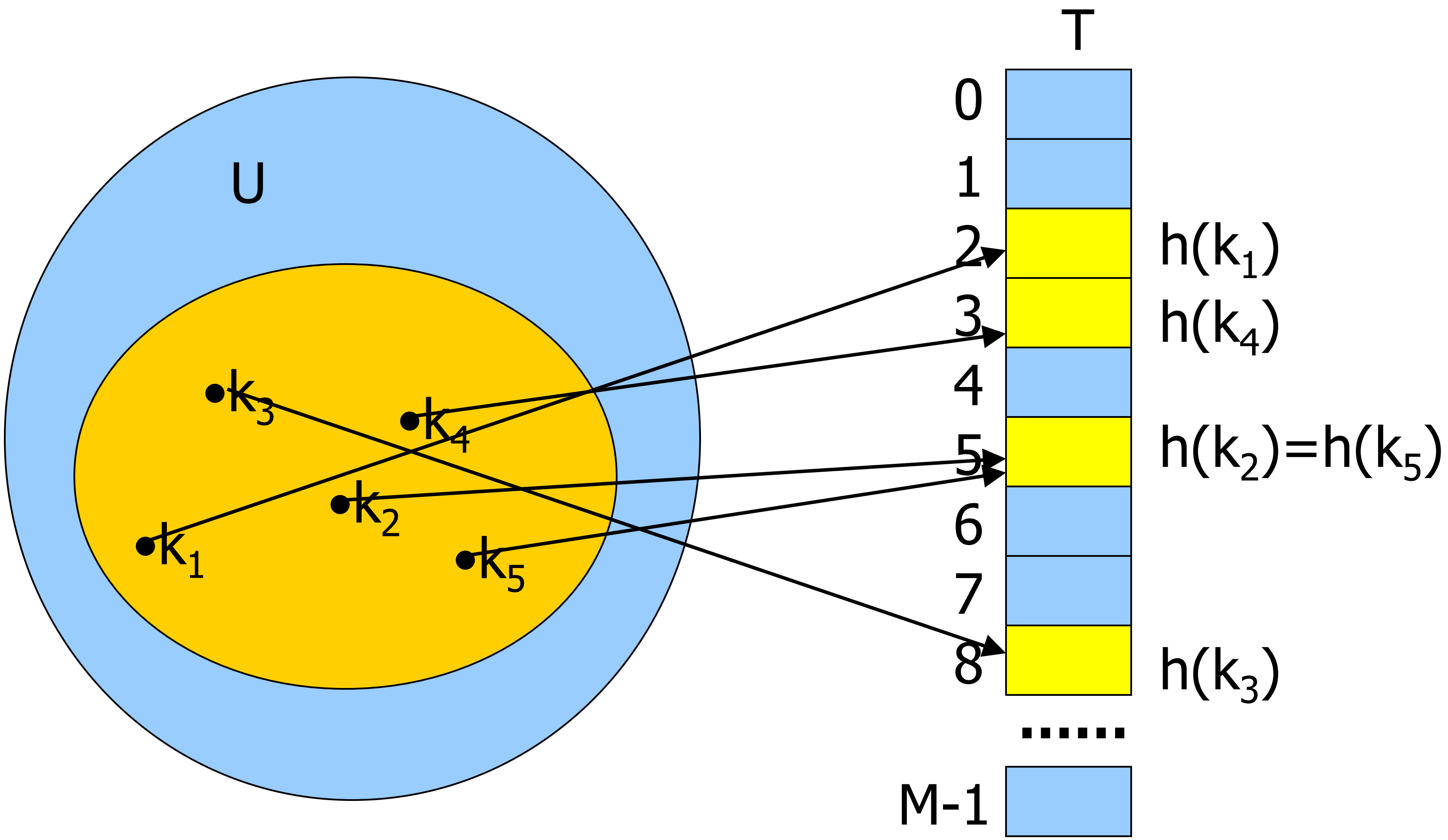
⇒ collisione.

Funzione di hash

- La tabella di hash ha dimensione M e contiene |K| elementi (|K|<<|U|)
- La tabella di hash ha indirizzi nell'intervallo [0 … M-1]
- La funzione di hash h: mette in corrispondenza una chiave k in un indirizzo della tabella h(k)

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

- L'elemento x viene memorizzato all'indirizzo h(k) dato dalla sua chiave k (attenzione alla gestione delle collisioni!).


T
U
0
1
2
3
4
5
6
7
8
M-1
k_1
k_2
k_3
k_4
k_5
$h(k_1)$
$h(k_4)$
$h(k_2)=h(k_5)$
$h(k_3)$

Progetto della funzione di hash

Funzione ideale: hashing uniforme semplice:

se le chiavi k sono equiprobabili, allora i valori di h(k) devono essere equiprobabili.

In pratica:

- le chiavi k non sono equiprobabili, anzi sono correlate:
  - usare tutti i bit della chiave
  - “amplificare” le differenze.

Tipologie di funzioni di hash

Metodo moltiplicativo:

chiavi: numeri in virgola mobile in un intervallo prefissato ($s \leq k \leq t$):

$$h(k) = \lceil (k - s) / (t - s) * M \rceil$$

```
#define hash(k, M)(((k-s)/(t-s))*M)
```

Esempio:

$M = 97$, $s = 0$, $t = 1$

$k = 0.513870656$

$h(k) = \lceil (0.513870656 - 0) / (1 - 0) * 97 \rceil = 50$

Metodo modulare:

chiavi: numeri interi di w bit:

M numero primo

$$h(k) = k \bmod M$$

```
#define hash(k, M)(k %M)
```

Esempio:

$M = 19$

$k = 31$

$h(k) = 31 \bmod 19 = 12$

M numero primo evita:

- di usare solo gli ultimi n bit di k se $M = 2^n$
- di usare solo le ultime n cifre decimali di k se $M = 10^n$.

Metodo moltiplicativo-modulare

chiavi: numeri interi:

- data costante $0<A<1$

$A = \phi = (\sqrt{5} - 1) / 2 = 0.6180339887$

- $h(k) = \lfloor k \cdot A \rfloor \bmod M$

Metodo modulare:

chiavi: stringhe alfanumeriche corte come interi derivati dalla valutazione di polinomi in una data base

M numero primo

$$h(k) = k \bmod M$$

Esempio:

stringa now = ‘n’*128^2 + ‘o’*128 + ‘w’

= $110*128^2 + 111*128 + 119$

$k = 1816567$

$k = 1816567 \quad M = 19$

$h(k) = 1816567 \bmod 19 = 15$

Metodo modulare:

chiavi: stringhe alfanumeriche lunghe come interi derivati dalla valutazione di polinomi in una data base con il metodo di Horner: ad esempio

$$P_7(x) = p_7x^7+p_6x^6 + p_5x^5+p_4x^4+p_3x^3+p_2x^2+p_1x+p_0$$
$$= ((((((p_7x+p_6)x + p_5)x+p_4)x+p_3)x+p_2)x+p_1)x+p_0$$

Come prima:

M numero primo

$$h(k) = k \bmod M$$

Esempio: stringa averylongkey con base 128 (ASCII)

$k = 97*128^{11}+118*128^{10}+101*128^{9}+114*128^{8}+121*128^{7}+108*128^{6}+111*128^{5}+110*128^{4}+103*128^{3}+107*128^{2}+101*128^{1}+121*128^{0}$

Ovviamente k non è rappresentabile su un numero ragionevole di bit.

Con il metodo di Horner:

$k = ((((((((((97*128+118)*128+101)*128+114)*128+121)*128+108)*128+111)*128+110)*128+103)*128+107)*128+101)*128+121$

Ma anche con il metodo di Horner k non è rappresentabile su un numero ragionevole di bit.

E' possibile però ad ogni passo eliminare i multipli di M, anziché farlo dopo in fase di applicazione del metodo modulare, ottenendo la seguente funzione di hash per stringhe con base 128 per l'ASCII:

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

In realtà anche per stringhe ASCII non si usa 128 come base, bensì:

- un numero primo (ad esempio 127)
- numero pseudocasuale diverso per ogni cifra della chiave (hash universale)

con lo scopo di ottenere una distribuzione abbastanza uniforme (probabilità di collisione tra 2 chiavi diverse prossima a 1/M).

Funzione di hash per chiavi stringa con base prima:

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

Funzione di hash per chiavi stringa con hash universale:

```
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;
}
```

Collisioni

Definizione:

collisione: $h(k_i)=h(k_j)$ per $k_i \neq k_j$

Le collisioni sono inevitabili, occorre:

- minimizzarne il numero (buona funzione di hash):
- gestirle:
  - linear chaining
  - open addressing.

Linear Chaining

Più elementi possono risiedere nella stessa locazione della tabella T $\Rightarrow$ lista concatenata.

Operazioni:

- inserimento in testa alla lista
- ricerca nella lista
- cancellazione dalla lista.

Determinazione della dimensione M della tabella:

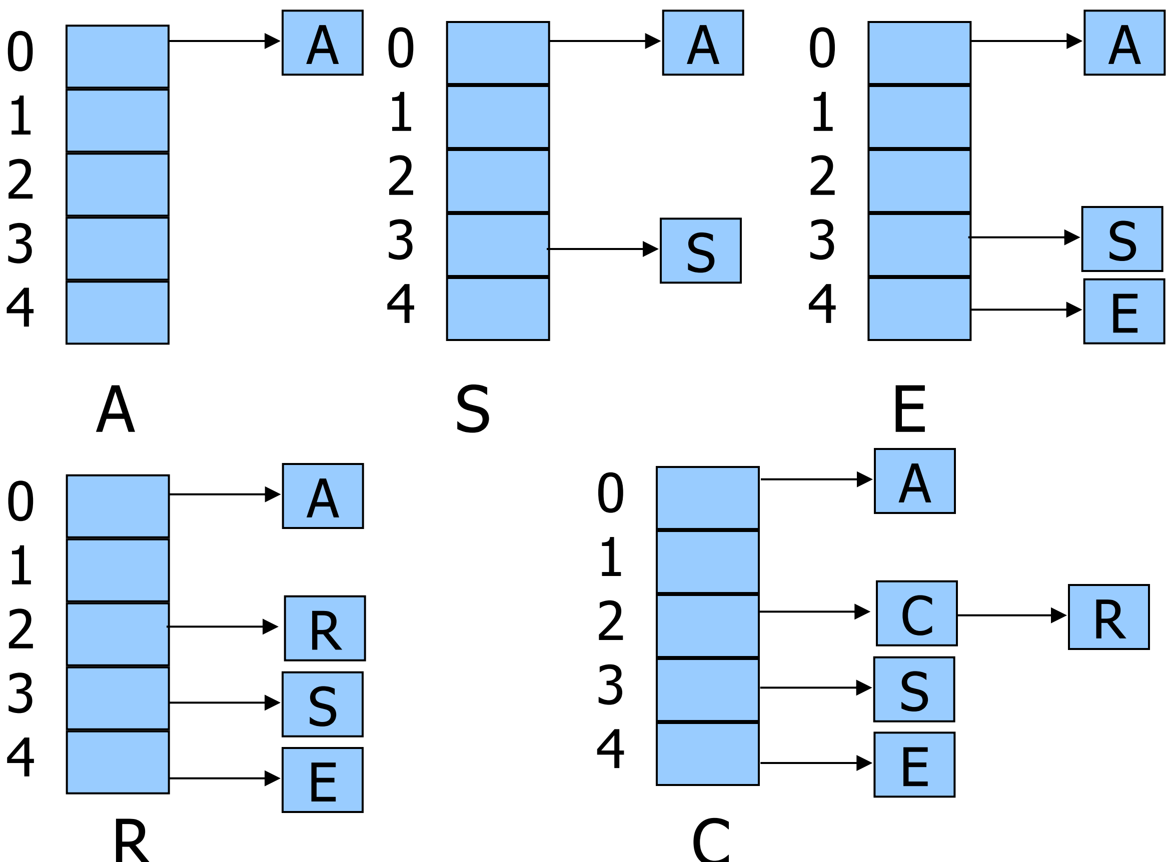
- il più piccolo primo M $\geq$ numero di chiavi max / 5 (o 10) così che la lunghezza media delle liste sia 5 (o 10)

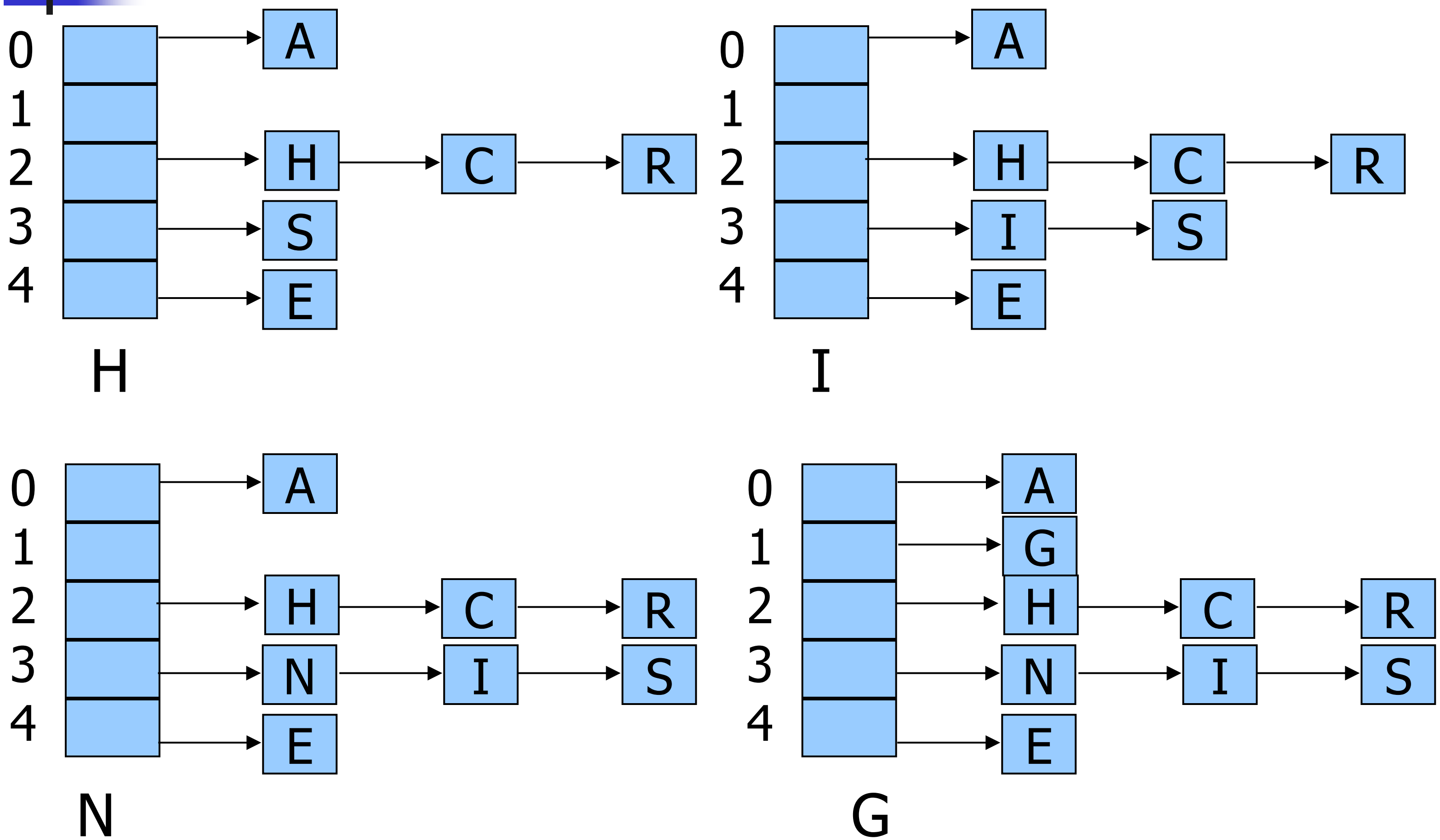
Esempio

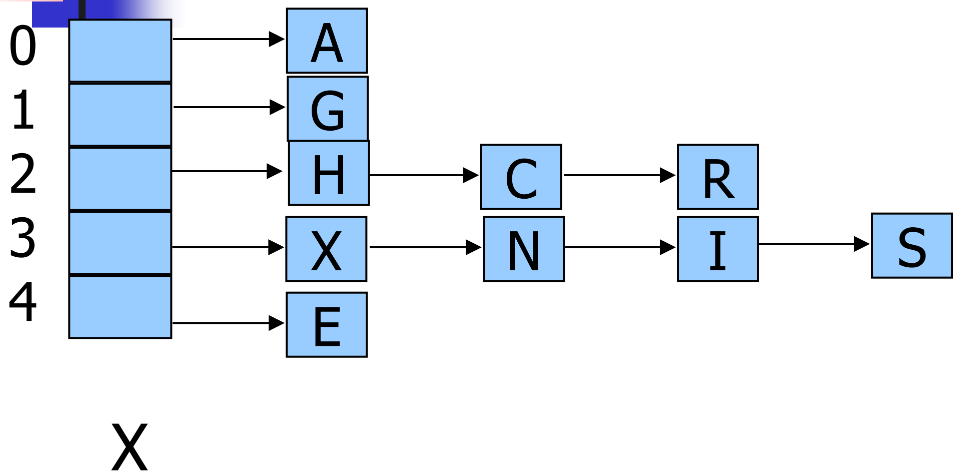
```
M = 5;
int hash(Key v, int M) {
  int h = 0, base = 127;
  for ( ; *v != '\0'; v++)
    h = (base * h + *v) % M;
  return h;
}
```

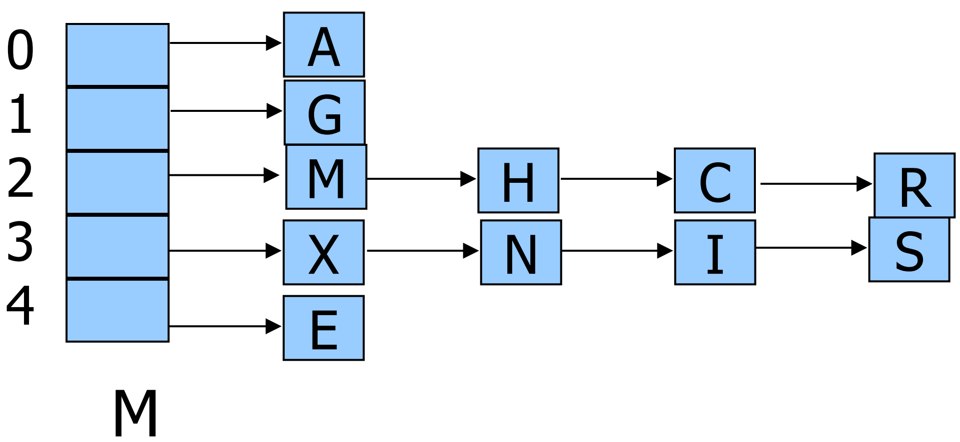
A S E R C H I N G X M P L

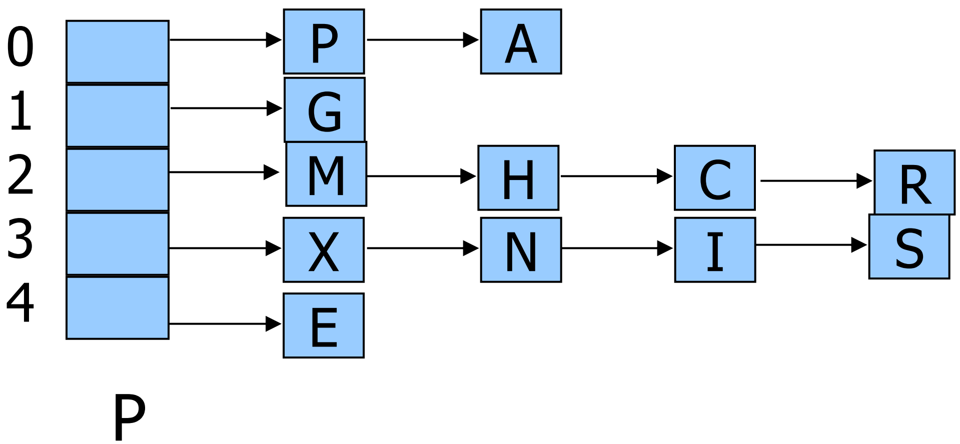
h(k) = 0 3 4 2 2 2 3 3 1 3 2 0 1

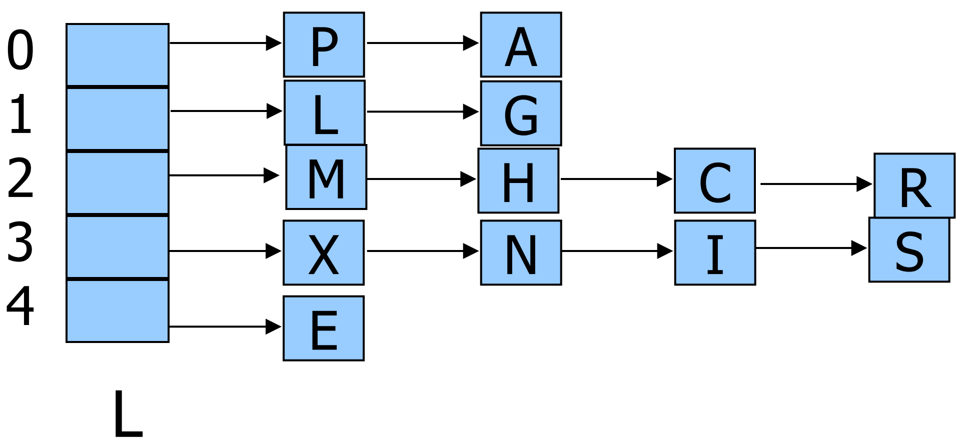



0
1
2
3
4
A
H
C
R
S
E
H
0
1
2
3
4
A
H
C
R
I
S
E
I
0
1
2
3
4
A
H
C
R
N
I
S
E
N
0
1
2
3
4
A
G
H
C
R
N
I
S
E
G


0
1
2
3
4
A
G
H
C
R
X
N
I
S
E
X


0
1
2
3
4
A
G
M
H
C
R
X
N
I
S
E
M


0
1
2
3
4
P
A
G
M
H
C
R
X
N
I
S
E
P


0
1
2
3
4
P
A
L
G
M
H
C
R
X
N
I
S
E
L

Linear chaining

```
typedef struct STnode* link;

struct STnode { Item item; link next; } ;

struct symboltable { link *heads; int M; link z; };

Item NULLitem = EMPTYitem;

link NEW( Item item, link next)
{
  link x = malloc(sizeof *x);
  x->item = item;
  x->next = next;
  return x;
}
```

```
ST STinit(int maxN) {
  int i;
  ST st = malloc(sizeof *st) ;
  st->M = maxN/5;
  st->heads = malloc(st->M*sizeof(link));
  st->z = NEW(NULLitem, NULL);
  for (i=0; i < st->M; i++)
    st->heads[i] = st->z;
  return st;
}
Item searchR(link t, Key v, link z) {
  if (t == z) return NULLitem;
  if (eq(key(t->item), v)) return t->item;
  return searchR(t->next, v, z);
}
Item STsearch(ST st, Key v) {
  return searchR(st->heads[hash(v, st->M)], v, st->z);
}
```

```
void STinsert (ST st, Item item) {
  int i = hash(key(item), st->M);
  st->heads[i] = NEW(item, st->heads[i]);
}
link deleteR(link x, Item v) {
  if ( x == NULL ) return NULL;
  if (eq(key(x->item), key(v))) {
    link t = x->next; free(x); return t;
  }
  x->next = deleteR(x->next, v);
  return x;
}

void STdelete(ST st, Item item) {
    int i = hash(key(item), st->M);
    st->heads[i] = deleteR(st->heads[i], item);
}
```

Complessità

Ipotesi:

Liste non ordinate:

- N =|K| = numero di elementi memorizzati
- M = dimensione della tabella di hash

Hashing semplice uniforme:

h(k) ha egual probabilità di generare gli M valori di uscita.

Definizione

fattore di carico α=N/M (>, = o < 1)

- Inserimento: $T(n) = O(1)$
- Ricerca:
  - caso peggiore $T(n) = \Theta(N)$
  - caso medio $T(n) = O(1+\alpha)$
- Cancellazione:
  - $T(n) = O(1)$ se disponibile il puntatore ad x e la lista è doppiamente linkata
  - come la ricerca se disponibile il valore di x, oppure il valore della chiave k, oppure la lista è semplicemente linkata

Open addressing

- Ogni cella della tabella T può contenere un solo elemento.
- Tutti gli elementi sono memorizzati in T.
- Collisione: ricerca di cella non ancora occupata mediante probing:
  - generazione di una permutazione delle celle = ordine di ricerca della cella libera. Concettualmente:

$N \leq M$
$\alpha \leq 1$

$$h(k, t) : U \times \{ 0,1,\ldots,M-1 \} \rightarrow \{ 0,1,\ldots,M-1 \}$$

chiave tentativo (0…M-1)

Open addressing

Determinazione della dimensione M della tabella:

- il più piccolo primo M ≥ doppio del massimo numero di chiavi presenti (input dell'utente)

```
#define full(A) (neq(key(st->a[A]), key(NULLitem)))
#define null(A) (eq(key(st->a[A]), key(NULLitem)))

struct symboltable { Item *a; int N; int M;};

ST STinit(int maxN) {
  ST st; int i;
  st = malloc(sizeof(*st));
  st->N = 0;   st->M = maxN;
  st->a = malloc(st->M * sizeof(Item) );
  for (i = 0; i < st->M; i++)
    st->a[i] = NULLitem;
  return st;
}
```

Funzioni di probing

- Linear probing
- Quadratic probing
- Double hashing

Un problema dell'open addressing è il **clustering**, cioè il raggruppamento di posizioni occupate contigue.

Linear probing

Insert:

- calcola i = h(k)
- se libero, inserisci chiave, altrimenti incrementa i di 1 modulo M
- Ripeti fino a cella vuota.

```
void STinsert(St st, Item item) {
  int i = hash(key(item), st->M);
  while (full(i))
    i = (i+1)%st->M;
  st->a[i] = item;
  st->N++;
}
```

Search:

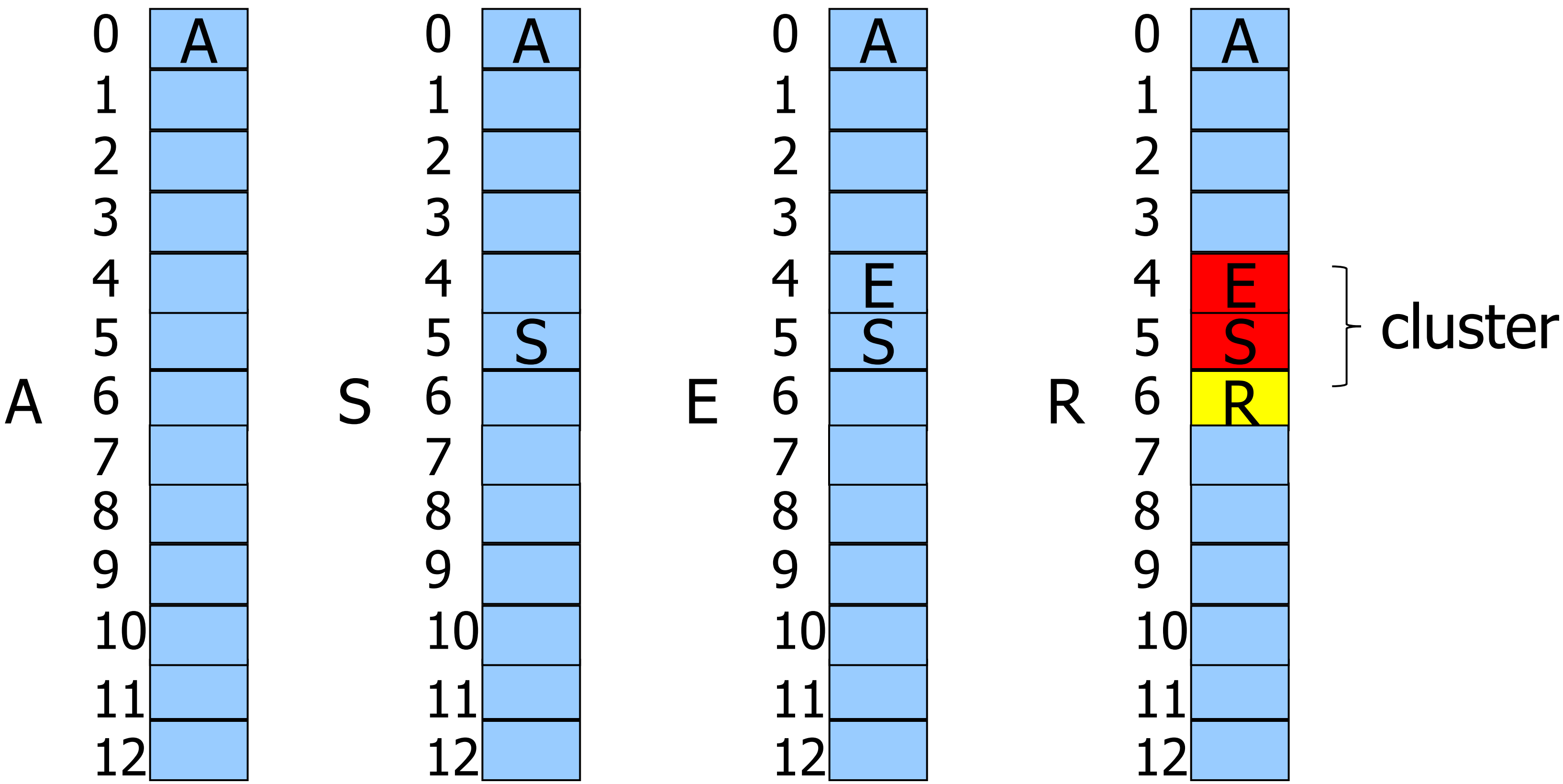
- calcola i = h(k)
- se trovata chiave, termina con successo
- incrementa i di 1 modulo M
- ripeti fino a cella vuota (insuccesso).

```
Item STsearch(ST st, Key v) {
  int i = hash(v, st->M);
  while (full(i))
    if (eq(v, key(st->a[i])))
      return st->a[i];
    else
      i = (i+1)%st->M;
  return NULLitem;
}
```

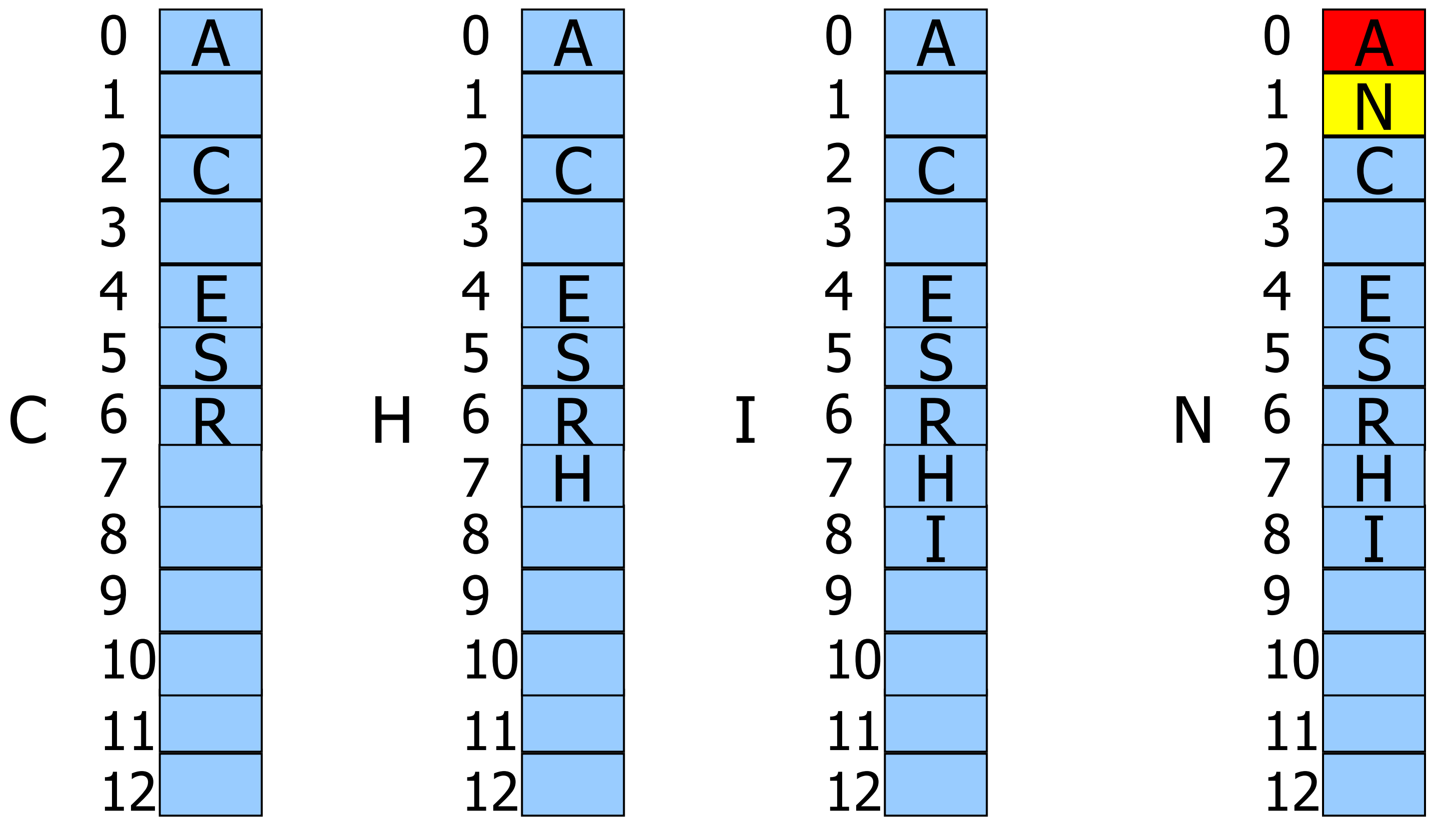
Esempio

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

	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



	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



	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

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

Delete:

operazione complessa che interrompe le catene di collisione.

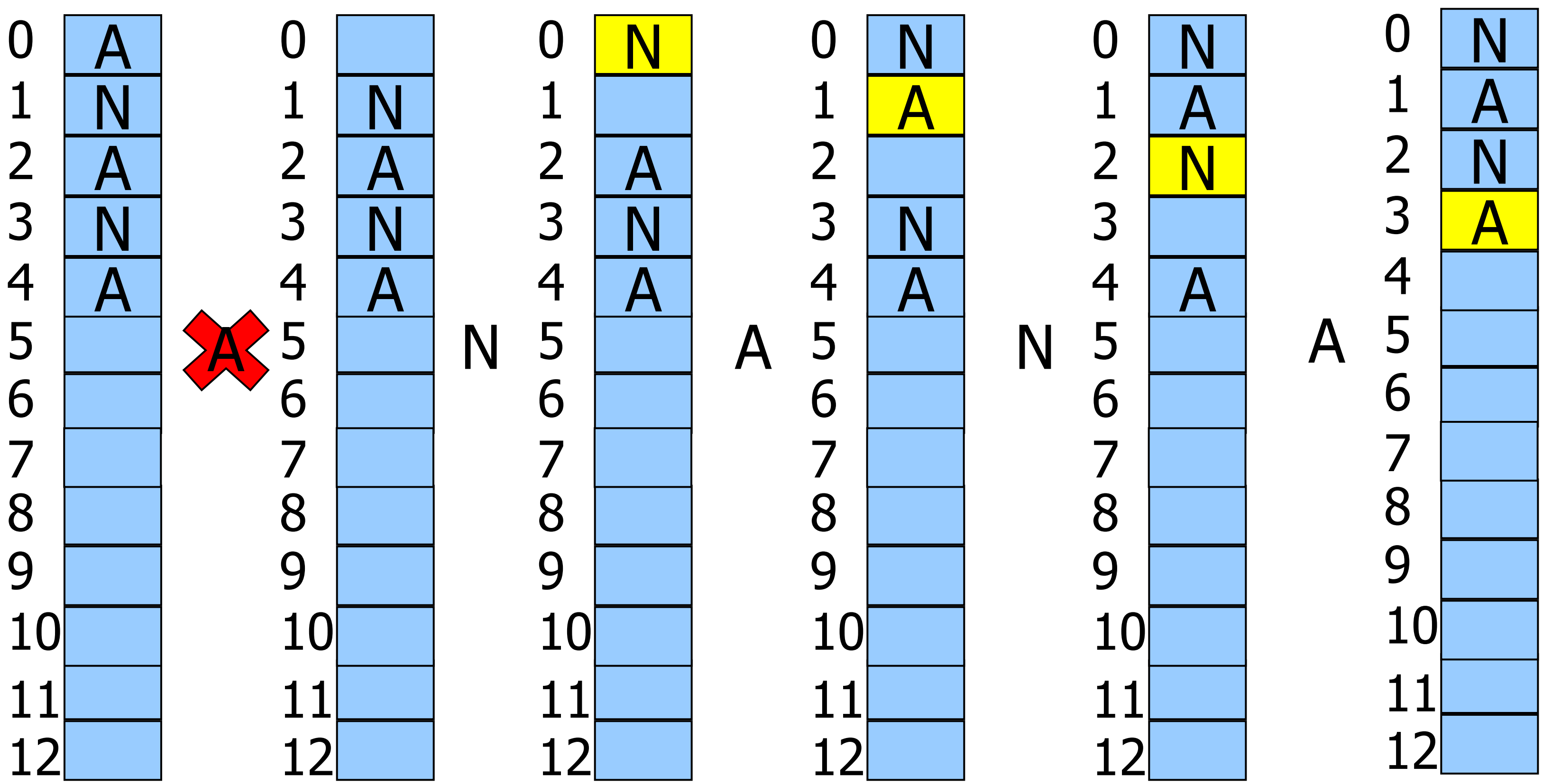
L’open addressing è in pratica utilizzato solo quando non si deve mai cancellare.

Soluzioni:

- sostituire la chiave cancellata con una chiave sentinella che conta come piena in ricerca e vuota in inserzione
- reinserire le chiavi del cluster sottostante la chiave cancellata

Esempio

Delete A, poi reinserire le chiavi del cluster N A N A



```c
void STdelete(ST st, Item item) {
  int j, i = hash(key(item), st->M);
  Item v;
  while (full(i))
    if eq(key(item), key(st->a[i]))
      break;
    else
      i = (i+1) % st->M;
  if (null(i))
    return;
  st->a[i] = NULLitem;
  st->N--;
  for (j = i+1; full(j); j = (j+1)%st->M, st->N--) {
    v = st->a[j];
    st->a[j] = NULLitem;
    STinsert(st, v);
  }
}
```

Complessità con l'ipotesi di:

- hashing semplice uniforme
- probing uniforme.

Tentativi in media di "probing" per la ricerca:

- search miss: $1/2(1 + 1/(1-\alpha))$
- search hit: $1/2(1 + 1/(1-\alpha)^2)$

α	1/2	2/3	3/4	9/10
hit	1.5	2.0	3.0	5.5
miss	2.5	5.0	8.5	55.5

Quadratic probing

Insert:

- calcola index = h(k)
- se libero, inserisci chiave, altrimenti incrementa index di $c_1 i + c_2 i^2$ modulo M
- ripeti fino a cella vuota.

```
#define c1 1
#define c2 1
void STinsert(ST st, Item item) {
  int i = 0;
  int index = hash(key(item), st->M);
  while (full(index)) {
    i++;
    index = (index + c1*i + c2*i*i)%st->M;
  }
  st->a[index] = item;
  st->N++;
}
```

Scelta di c_1 e c_2:

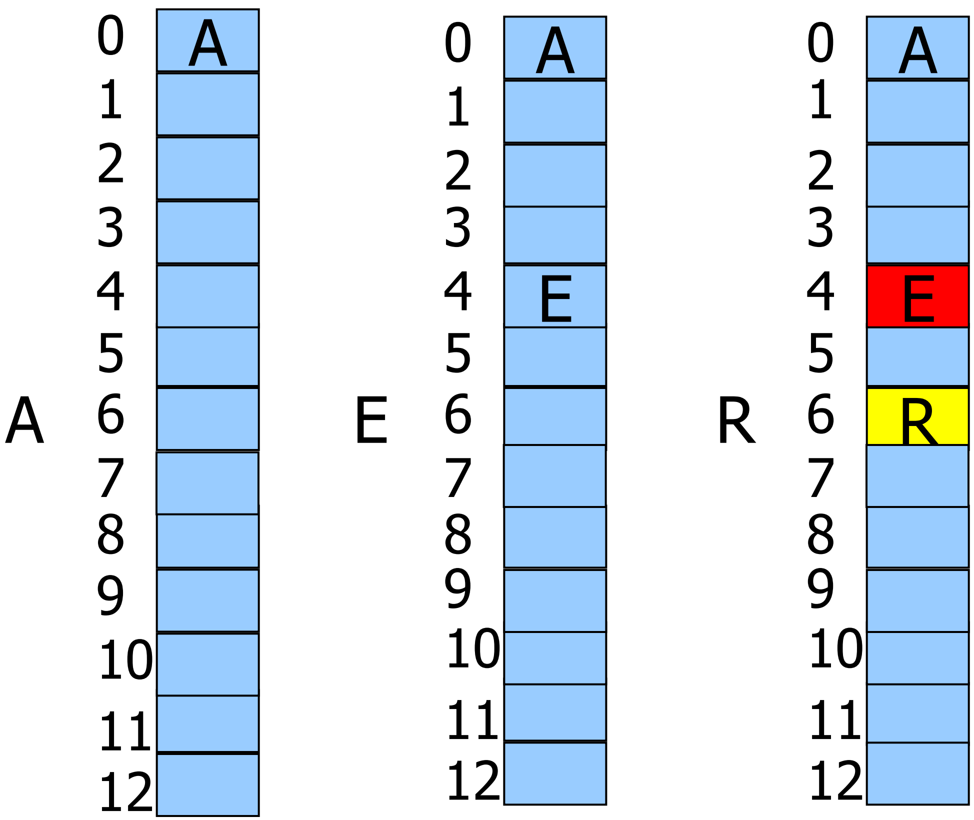
- se $M = 2^K$, scegliere $c_1 = c_2 = ½$ per garantire che siano generati tutti gli indici tra 0 e M-1:
- se M è primo, se $\alpha < ½$ i seguenti valori
  - $c_1 = c_2 = ½$
  - $c_1 = c_2 = 1$
  - $c_1 = 0$, $c_2 = 1$.

garantiscono che, con inizialmente index = h(k) e poi index = (index + $c_1 i + c_2 i^2$) modulo M si abbiano valori distinti per $1 \leq i \leq (M-1)/2$.

Esempio

$\alpha = 6/13 < ½$

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

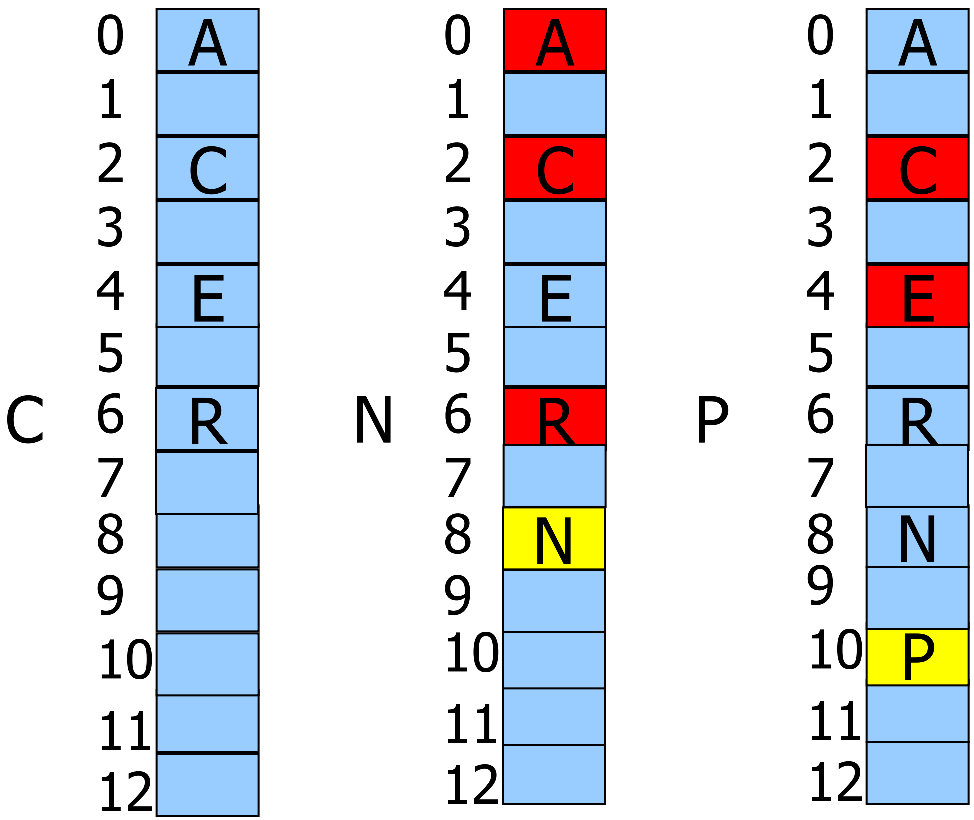


Funzione di quadratic probing

$i + i^2$

$(4 + 1 + 1^2) \bmod 13 = 6$

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



Funzione di quadratic probing

$i + i^2$

$(0 + 1 + 1^2) \bmod 13 = 2$

$(2 + 2 + 2^2) \bmod 13 = 8$

$(2 + 1 + 1^2) \bmod 13 = 4$

$(4 + 2 + 2^2) \bmod 13 = 10$

Double hashing

Insert:

- calcola $i = h_1(k)$
- se posizione libera, inserisci chiave, altrimenti calcola $j = h_2(k)$ e prova in
  $i = (i + j) \bmod M$
- ripeti fino a cella vuota. Ricordare che, se $M = 2*max$, $\alpha < 1$

Esempi di h_1 e h_2

$$h_1(k) = k \bmod M \text{ e } M \text{ primo}$$

$$h_2(k) = (1 + (k \bmod 97)) \bmod M$$

```c
int hash1(Key v, int M) {
  int h = 0, base = 127;
  for ( ; *v != '\0'; v++)
    h = (base * h + *v) % M;
  return h;
}

int hash2(Key v, int M) {
  int h = 0, base = 127;
  for ( ; *v != '\0'; v++)
    h = (base * h + *v);
  h = ((h % 97) + 1) % M;
  return h;
}
```

```c
void STinsert(St st, Item item) {
  int i = hash1(key(item), st->M);
  int j = hash2(key(item), st->M);
  while (full(i))
    i = (i+j)%st->M;
  st->a[i] = item;
  st->N++;
}
Item STsearch(ST st, Key v) {
  int i = hash1(v, st->M);
  int j = hash2(v, st->M);
  while (full(i))
    if (eq(v, key(st->a[i])))
      return st->a[i];
    else
      i = (i+j)%st->M;
  return NULLitem;
}
```

Esempio

`M = 13;`

	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			

	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

R

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

Double hashing

$$(4 + (17 \bmod 97 + 1) \bmod 13) \bmod 13 = 9$$

	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			

	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	

Double hashing

$(0 + (13 \bmod 97 + 1) \bmod 13) \bmod 13 = 1$

	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

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

	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

Double hashing

$$(2 + (15 \bmod 97 + 1) \bmod 13) \bmod 13 = 5$$

$$(5 + (15 \bmod 97 + 1) \bmod 13) \bmod 13 = 8$$

$$(8 + (15 \bmod 97 + 1) \bmod 13) \bmod 13 = 11$$

Complessità del double hashing

Ipotesi:

- hashing semplice uniforme
- probing uniforme.

Tentativi di "probing" per la ricerca:

- search miss: $1/(1-\alpha)$
- search hit: $1/\alpha \ln (1/(1- \alpha))$

α	1/2	2/3	3/4	9/10
hit	1.4	1.6	1.8	2.6
miss	1.5	2.0	3.0	5.5

Cofronto tra alberi e tabelle di hash

Tabelle di hash:

- più facili da realizzare
- unica soluzione per chiavi senza relazione d'ordine
- più veloci per chiavi semplici

Alberi (BST e loro varianti):

- meglio garantite le prestazioni (per alberi bilanciati)
- permettono operazioni su insiemi con relazione d'ordine.

Riferimenti

- Tabelle di hash
  - Cormen 12.1, 12.2, 12.3, 12.4
  - Sedgewick 14.1, 14.2, 14.3, 14.4