POLITECNICO
DI TORINO
Dipartimento
di Automatica e Informatica

# Le tabelle di hash

Gianpiero Cabodi e Paolo Camurati

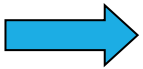
# Tabelle di hash

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: è un 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.
Usate per inserzione, ricerca, cancellazione, non per ordinamento e selezione.

# ADT di I classe ST

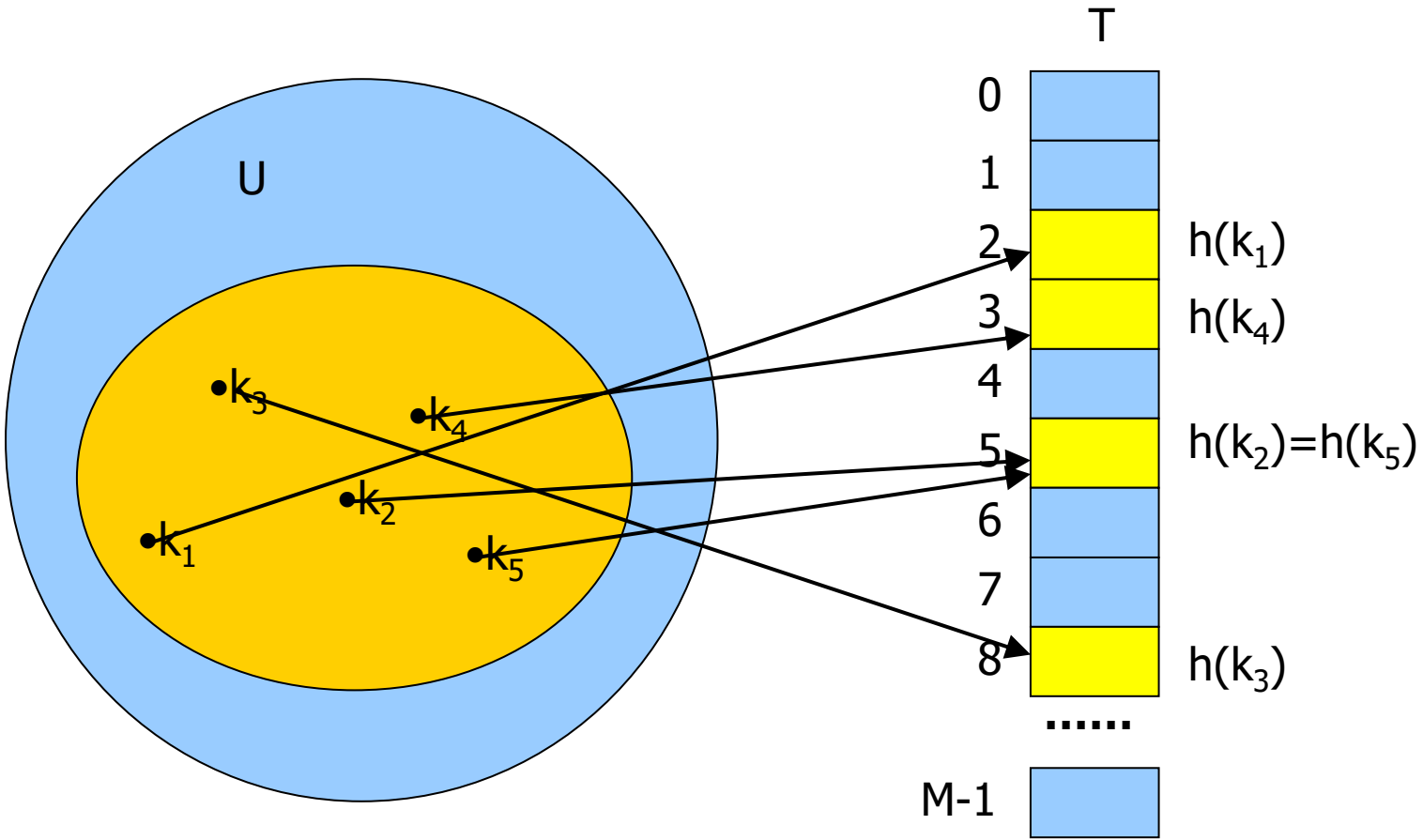
**ST.h**

```
typedef struct symboltable *ST;

ST      STinit(int maxN, float r) ;
void    STinsert(ST st, Item val);
Item    STsearch(ST st, Key k) ;
void    STdelete(ST st, Key k) ;
void    STdisplay(ST st) ;
void    STfree(ST st);
int     STcount(ST st);
int     STempty(ST st);
```

# 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 con un indirizzo della tabella h(k)

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

- chiavi k equiprobabili => valori di h(k) devono essere equiprobabili.

In pratica

- le chiavi k non sono equiprobabili
- chiavi diverse $k_i$, $k_j$ sono correlate.

Per rendere i **valori di h(k) equiprobabili** occorre:

- rendere $h(k_i)$ scorrelato da $h(k_j)$
  - "amplificare" le differenze
  - scorrelare h(k) da k
- distribuire gli h(k) in modo uniforme:
  - usare tutti i bit della chiave
  - moltiplicare per un numero primo.

# Tipologie di funzioni di hash

Metodo moltiplicativo:

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

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

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

Esempio:

M = 97, s = 0.0, t = 1.0

k = 0.513870656

h(k) = (0.513870656 -0) /(1 - 0) * 97 = 49

## Metodo modulare:

chiavi: numeri interi; M numero primo

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

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

Esempio:

M = 19

k = 31

h(k) = 31 % 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 $0<A<1$

    $A = \phi = (\sqrt{5} - 1) / 2 = 0.6180339887$
  - $h(k) = \lfloor k \cdot A \rfloor \% M$

## Metodo modulare

- chiavi: stringhe alfanumeriche corte come interi derivati dalla valutazione di polinomi in una data base
  - M numero primo
  - $h(k) = k \% M$

# Esempio

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

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

k = 1816567

k = 1816567  M = 19

h(k) = 1816567 % 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

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

# 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:

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

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

È 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;
}
```

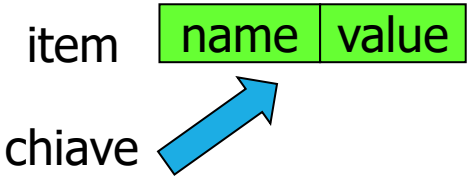
Funzione di hash per chiavi stringa con hash universale:

```
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;
}
```

# Item

- Quasi ADT Item
- Dati:
  - Nome (stringa), valore (intero)
  - Chiave = nome
  - Tipologia 3

  Negli esempi stringhe da 1 carattere e non visualizzato l'intero



# 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 $\Rightarrow$ 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 / r così che la lunghezza media delle liste sia r.

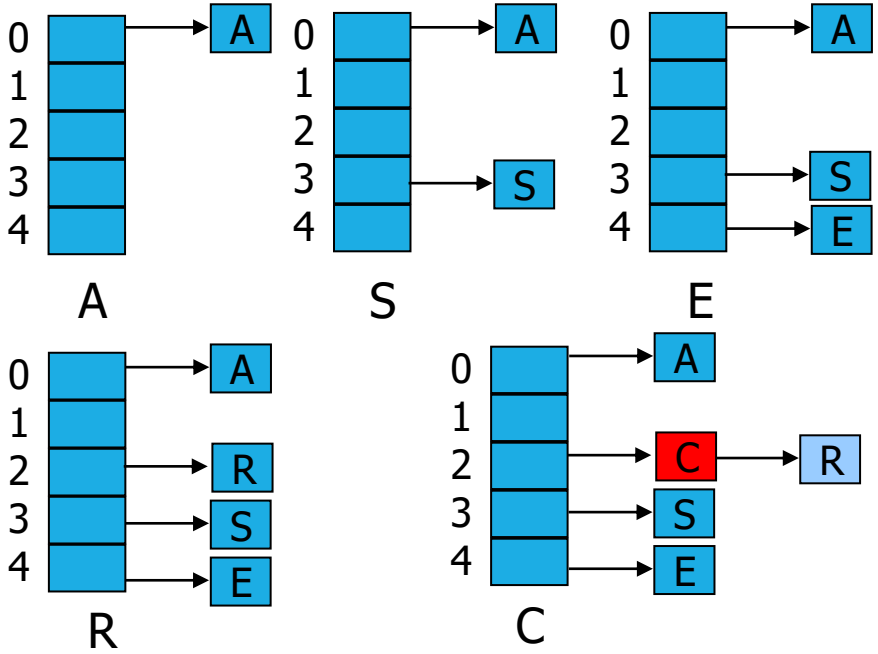
# Esempio

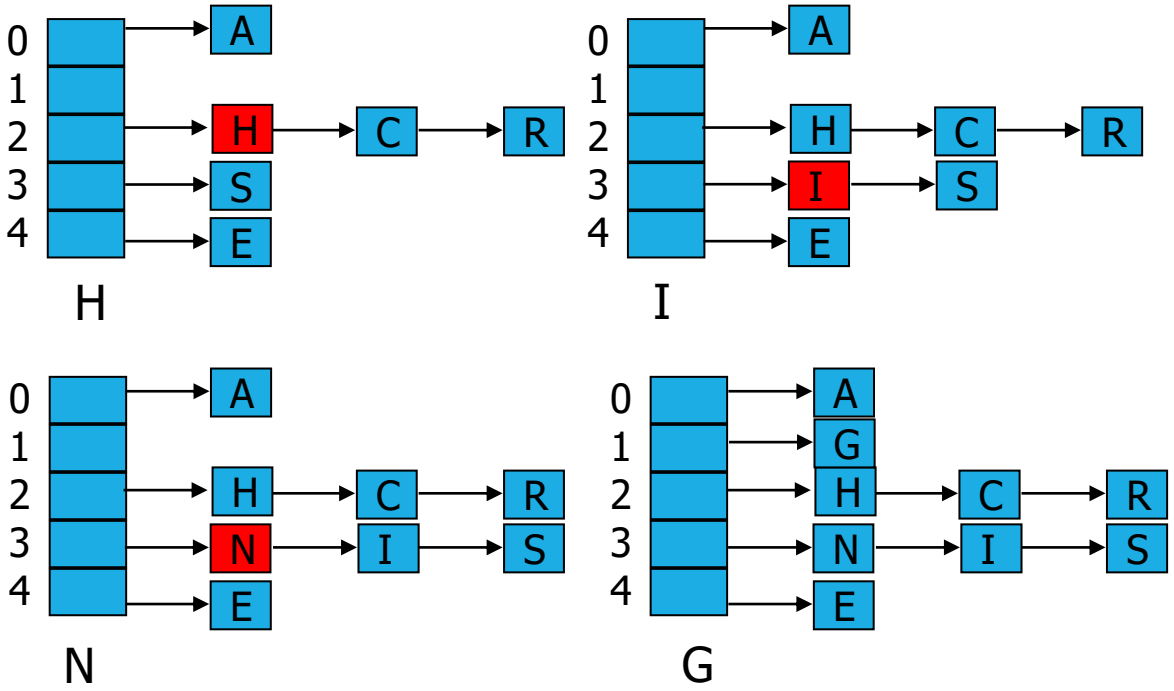
```
int hash (Key k, int M) {
  int h = 0, base = 127;
  for (; *k != '\0'; k++)
    h = (base * h + *k) % M;
  return h;
}
```

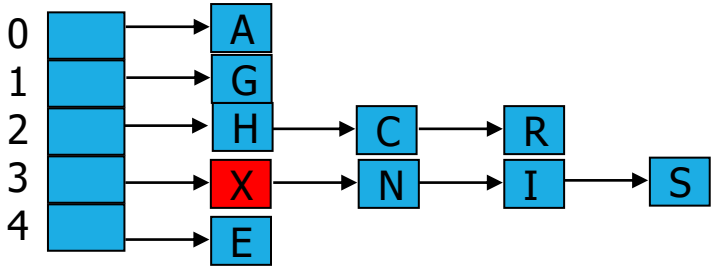
A S E R C H I N G X M P L

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

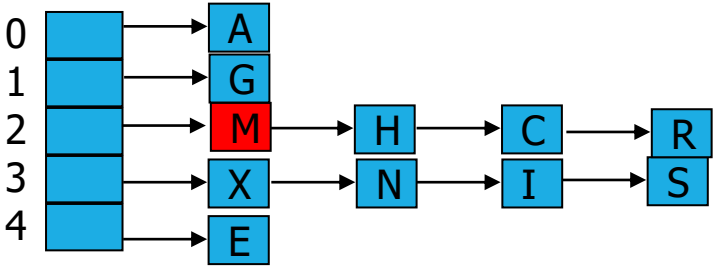
M = 5



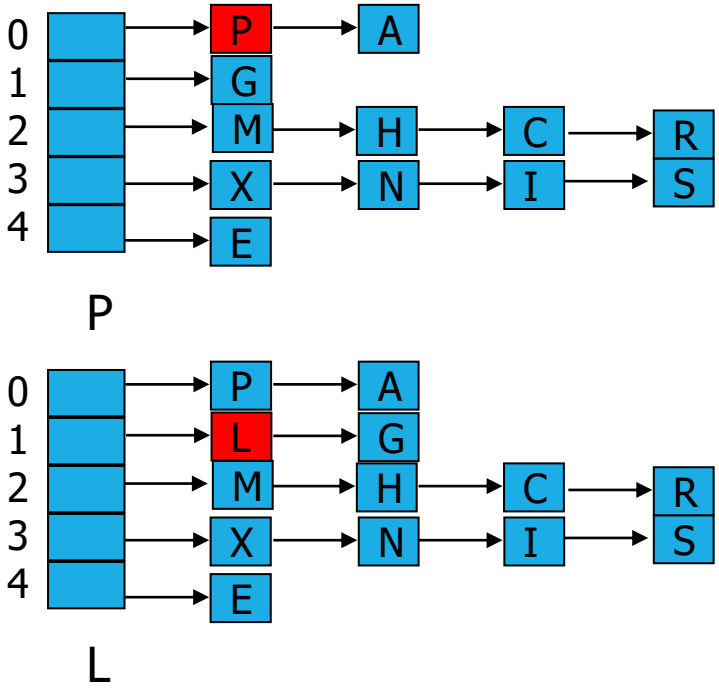

0
1
2
3
4
A
H
C
R
S
E
H
0
1
2
3
4
A
H
C
R
I
S
E
I
0
1
2
3
4
A
H
C
R
N
I
S
E
N
0
1
2
3
4
A
G
H
C
R
N
I
S
E
G




X




M


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

# Linear chaining

**ST.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Item.h"
#include "ST.h"

typedef struct STnode* link

struct STnode { Item item; link next; } ;

struct symbtab { link *heads; int N; int M; link z; };

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

vettore di liste con nodo sentinella in coda

nodo sentinella

dimensione tabella

numero di chiavi

```
ST STinit(int maxN, float r) {
  int i;
  ST st;

  st = malloc(sizeof(*st));
  st->N = 0;
  st->M = STsizeSet(maxN, r);
  st->heads = malloc(st->M*sizeof(link));
  st->z = NEW(ITEMsetNull(), NULL);

  for (i=0; i < st->M; i++)
    st->heads[i] = st->z;

  return st;
}
```

maxN ≤ 53

```
static int STsizeSet(int maxN, float r) {
  int primes[16]={2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53};
  int i = 0;
  int size;
  size = maxN /r;
  if (size < primes[15]) {
    for (i = 0; i<16; i++)
      if (size <= primes[i])
        return primes[i];
  }
  else
    printf("Too many entries!\n");
  return -1;
}
```

```
void STfree(ST st) {
  int i;
  link t,u;
  for(i=0; i<st->M; i++)
    for (t = st->heads[i]; t != st->z; t = u){
      u = t->next;
      free(t);
  }
  free(st->z);
  free(st->heads);
  free(st);
}

int STcount(ST st) {
  return st->N;
}

int STempty(ST st) {
  if (STcount(st) == 0)
    return 1;
  return 0;
}
```

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

static int hashU(Key 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;
}

void STinsert (ST st, Item val) {
  int i;
  i = hash(KEYget(&val), st->M);
  st->heads[i] = NEW(val, st->heads[i]);
}
```

```
Item STsearch(ST st, Key k) {
  return searchR(st->heads[hash(k, st->M)], k, st->z);
}
Item searchR(link t, Key k, link z) {
  if (t == z) return ITEMsetNull();
  if ((KEYcmp(KEYget(&t->item), k))==0) return t->item;
  return searchR(t->next, k, z);
}

void STdelete(ST st, Key k) {
  int i = hash(k, st->M);
  st->heads[i] = deleteR(st->heads[i], k);
}

link deleteR(link x, Key k) {
  if ( x == NULL ) return NULL;
  if ((KEYcmp(KEYget(&x->item), k))==0) {
    link t = x->next; free(x); return t;
  }
  x->next = deleteR(x->next, k);
  return x;
}
```

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

void visitR(link h, link z) {
  if (h == z) return;
  ITEMstore(h->item);
  visitR(h->next, z);
}

void STdisplay(ST st)  {
  int i;
  for (i=0; i < st->M; i++) {
    printf("st->heads[%d] = ", i);
    visitR(st->heads[i], st->z);
    printf("\n");
  }
}
```

# 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 $\alpha=N/M$ (>, = 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 può contenere un solo elemento
- Tutti gli elementi sono memorizzati in tabella

$N \leq M$

$\alpha \leq 1$

- Collisione: ricerca di cella non ancora occupata mediante probing:
  - generazione di una permutazione delle celle = ordine di ricerca della cella libera.
  - Concettualmente:

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

chiave tentativo (0…M-1)

# Open addressing

**ST.c**

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

ST STinit(int maxN, float alpha) {
  int i;
  ST st = malloc(sizeof(*st));
  st->N = 0;
  st->M = STsizeSet(maxN, alpha);
  if (st->M == -1)
    st = NULL;
  else {
    st->a = malloc(st->M * sizeof(Item) );
    for (i = 0; i < st->M; i++)
      st->a[i] = ITEMsetNull();
  }
  return st;
}
```

```
static int STsizeSet(int maxN, float alpha) {
  int primes[16]={2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53};
  int i = 0;
  if (maxN < primes[15]*alpha) {
    for (i = 0; i<16; i++)
      if (maxN <= primes[i]*alpha)
        return primes[i];
  }
  else
    printf("Too many entries!\n");
  return -1;
}
```

# 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(KEYget(&item), st->M);
  while (full(st, i))
    i = (i+1)%st->M;
  st->a[i] = item;
  st->N++;
}
int full(ST st, int i) {
  if (ITEMcheckNull(st->a[i])) return 0;
  return 1;
}
```

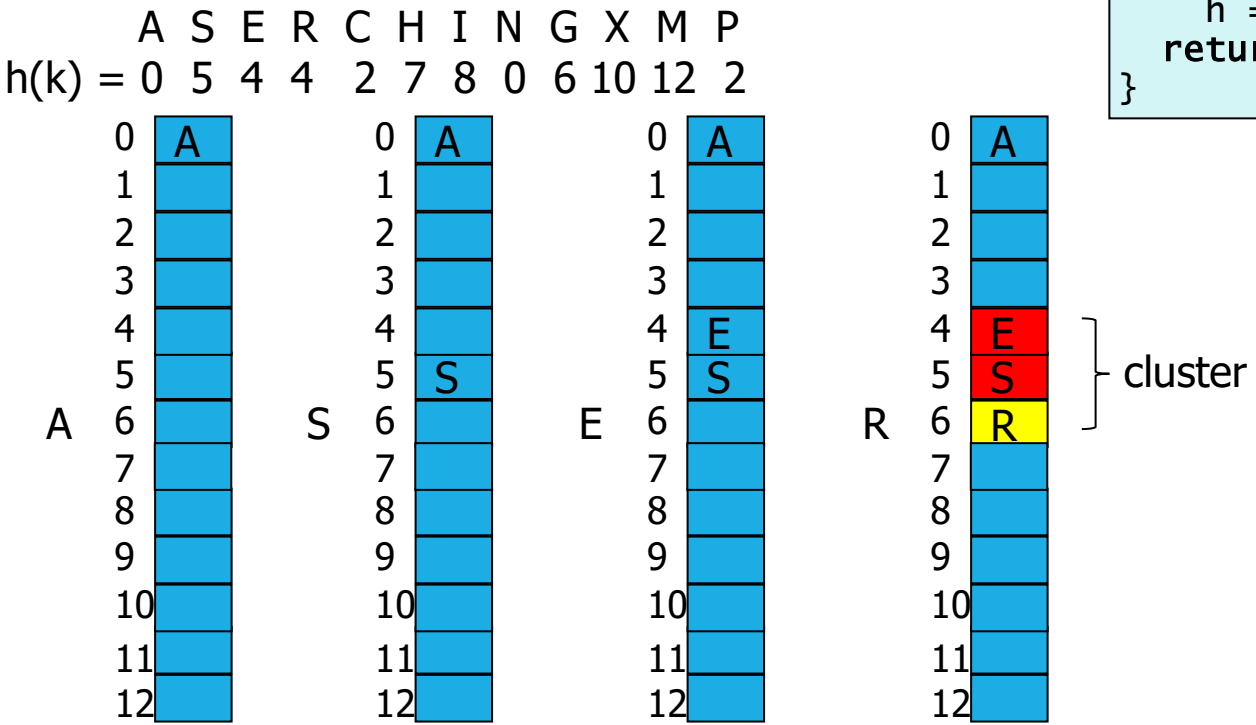
## Search:

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

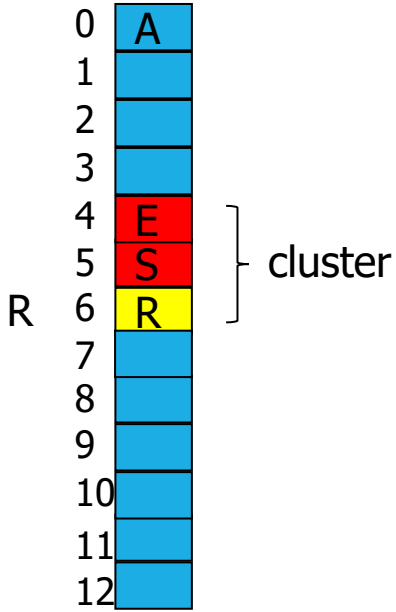
```
Item STsearch(ST st, Key k) {
  int i = hash(k, st->M);
  while (full(st, i))
    if (KEYcmp(k, KEYget(&st->a[i]))==0)
      return st->a[i];
    else
      i = (i+1)%st->M;
  return ITEMsetNull();
}
```

# Esempio

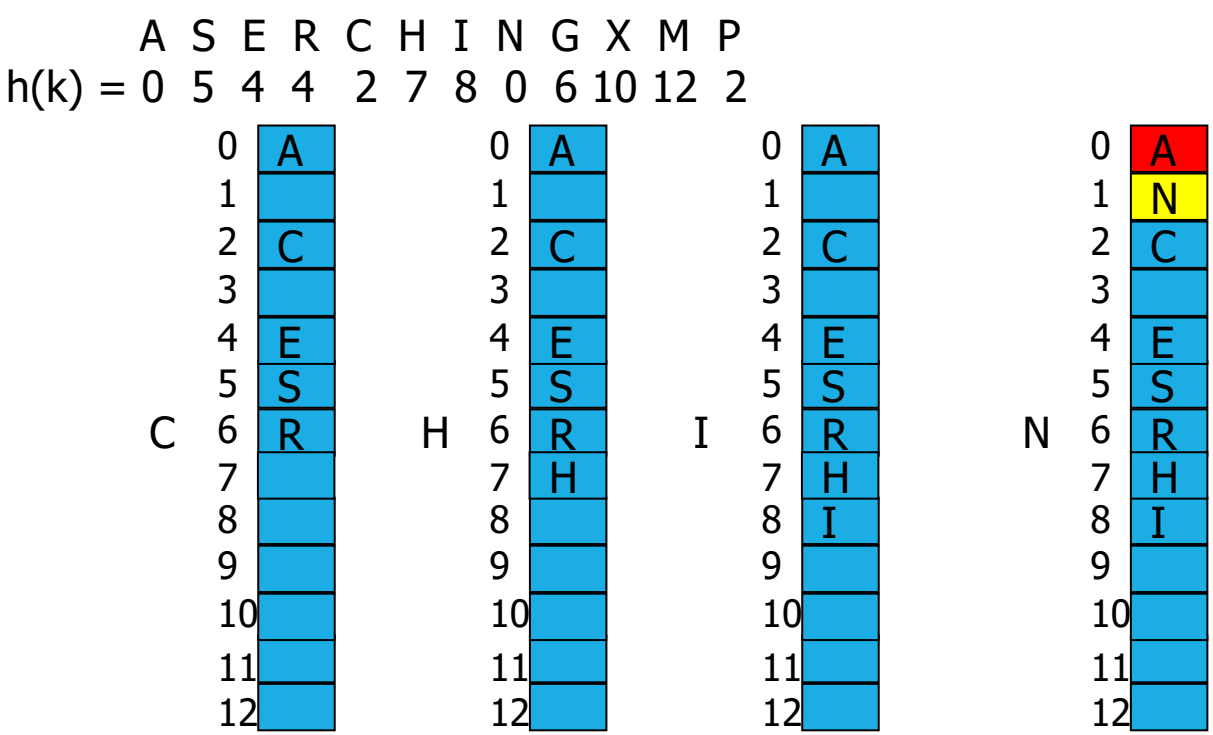
```
static int hash (Key k,int M){
  int h = 0, base = 127;
  for (; *k != '\0'; k++)
    h = (base * h + *k) % M;
  return h;
}
```



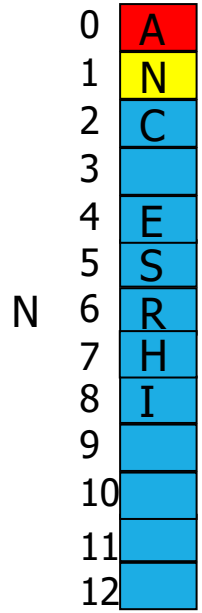
| | A | S | E | R | C | H | I | N | G | X | M | P |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| h(k) = | 0 | 5 | 4 | 4 | 2 | 7 | 8 | 0 | 6 | 10 | 12 | 2 |




i = h('R') = 82 % 13 = 4 collisione
i = (4+1) % 13 = 5 collisione
i = (5+1) % 13 = 6


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

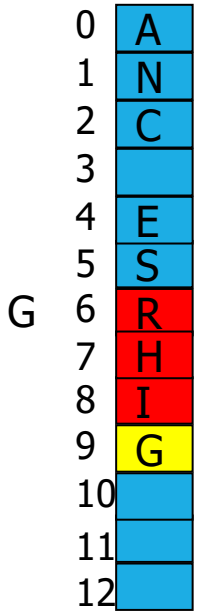
| | A | S | E | R | C | H | I | N | G | X | M | P |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| h(k) = | 0 | 5 | 4 | 4 | 2 | 7 | 8 | 0 | 6 | 10 | 12 | 2 |



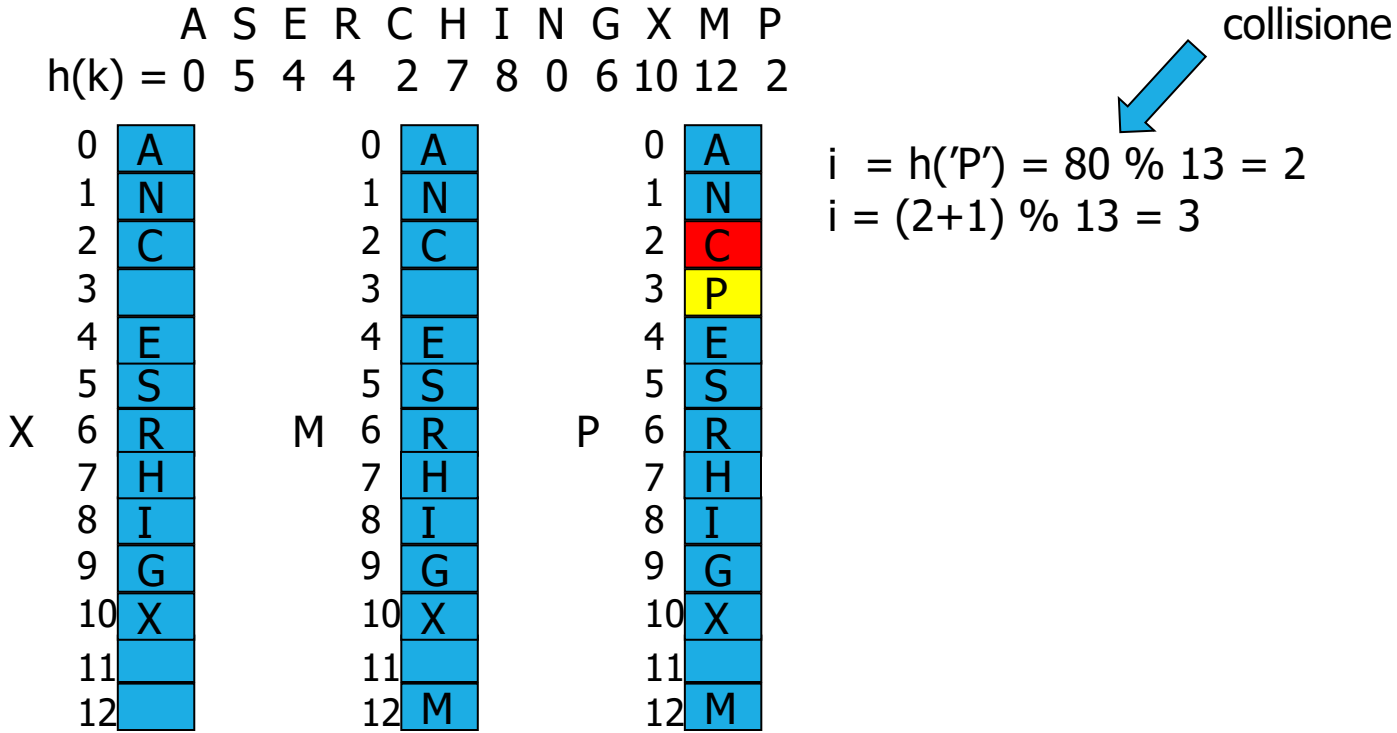

i = h('N') = 78 % 13 = 0 collisione

i = (0+1) % 13 = 1

| | 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('G') = 71 % 13 = 6 collisione
i = (6+1) % 13 = 7 collisione
i = (7+1) % 13 = 8 collisione
i = (8+1) % 13 = 9


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
collisione
i = h('P') = 80 % 13 = 2
i = (2+1) % 13 = 3
X
M
P
0 A
1 N
2 C
3
4 E
5 S
6 R
7 H
8 I
9 G
10 X
11
12
0 A
1 N
2 C
3
4 E
5 S
6 R
7 H
8 I
9 G
10 X
11
12 M
0 A
1 N
2 C
3 P
4 E
5 S
6 R
7 H
8 I
9 G
10 X
11
12 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:

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

# Soluzione 1

Nell'ADT si introduce un vettore `status` di interi: 0 se la cella è vuota, 1 se è occupata, -1 se cancellata.

La funzione `CheckFull` controlla se la cella `i` è piena (`status`=1). La funzione `CheckDeleted` controlla se la cella è cancellata (`status`=-1).

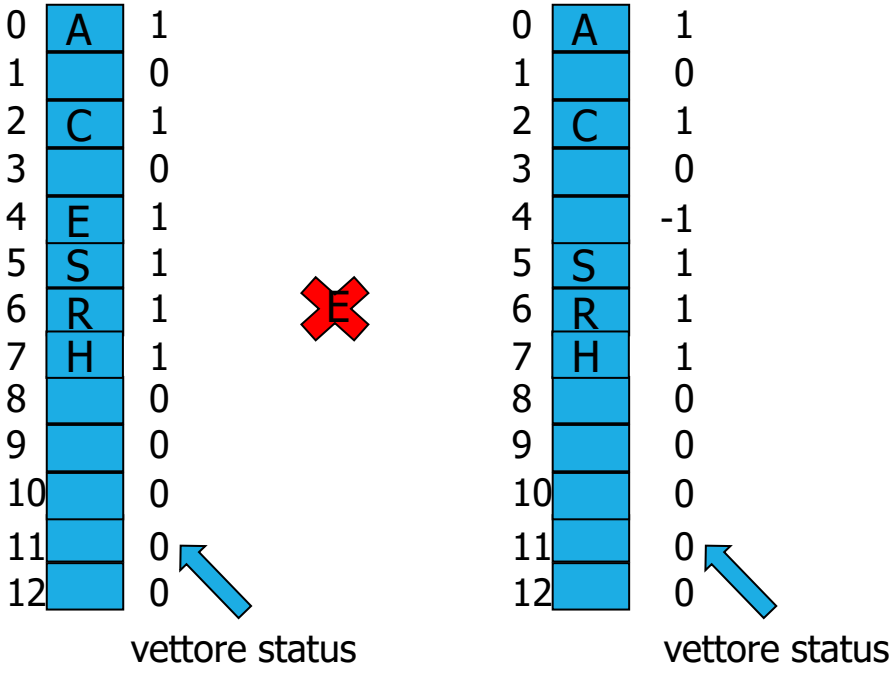
```
struct symboltable { Item *a; int *status; int N; int M;};
static int CheckFull(ST st, int i);
static int CheckDeleted(ST st, int i);
```

```
static int CheckFull(ST st, int i) {
  if (st->status[i] == 1) return 1;
  return 0;
}

static int CheckDeleted(ST st, int i){
  if (st->status[i] == -1) return 1;
  return 0;
}

void STinsert(ST st, Item item) {
  int i = hash(KEYget(&item), st->M);
  while (CheckFull(st, i))
    i = (i+1)%st->M;
  st->a[i] = item;
  st->status[i] = 1;
  st->N++;
}
```

```
Item STsearch(ST st, Key k) {
    int i = hash(k, st->M);
    while (CheckFull(st, i)==1 || CheckDeleted(st, i)==1)
        if (KEYcmp(k, KEYget(&st->a[i]))==0) return st->a[i];
        else i = (i+1)%st->M;
    return ITEMsetNull();
}

void STdelete(ST st, Key k){
  int i = hash(k, st->M);
  while (CheckFull(st, i)==1 || CheckDeleted(st, i)==1)
    if (KEYcmp(k, KEYget(&st->a[i]))==0) break;
    else i = (i+1) % st->M;
  if (ITEMcheckNull(st->a[i])) return;
  st->a[i] = ITEMsetNull();
  st->N--;
  st->status[i]=-1;
}
```

# Esempio

Cancellare E, ricordando che c'era stata collisione tra E e R.

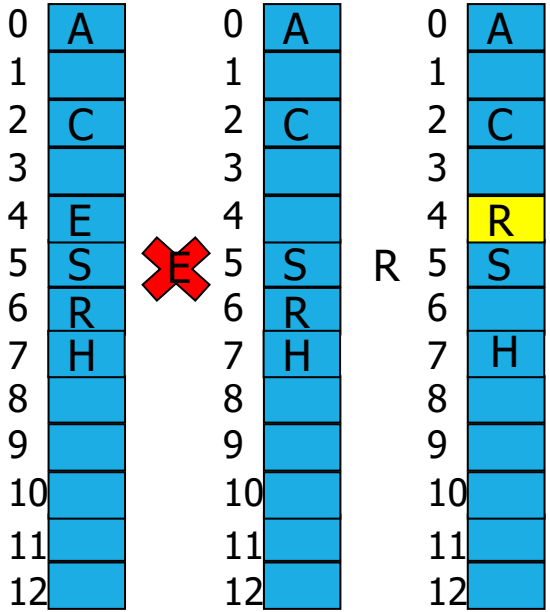


# Soluzione 2

```
void STdelete(ST st, Key k) {
  int j, i = hash(k, st->M);
  Item tmp;
  while (full(st, i))
    if (KEYcmp(k, KEYget(&st->a[i]))==0)
      break;
    else
      i = (i+1) % st->M;
  if (ITEMcheckNull(st->a[i]))
    return;
  st->a[i] = ITEMsetNull();
  st->N--;
  for (j = i+1; full(st, j); j = (j+1)%st->M, st->N--) {
    tmp = st->a[j];
    st->a[j] = ITEMsetNull();
    STinsert(st, tmp);
  }
}
```

# Esempio

Cancellare E, ricordando che c'era stata collisione tra E e R.



# Complessità

Complessità con l'ipotesi di:

- hashing semplice uniforme
- probing uniforme.

Tentativi in media di "probing" per la ricerca:

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

| $\alpha$ | 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:

- i è il contatore dei tentativi (all'inizio 0)
- index = (h'(k) + $c_1$i + $c_2$$i^2$)%M
- se libero, inserisci chiave, altrimenti incrementa i e ripeti fino a cella vuota.

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

# Search

```
Item STsearch(ST st, Key k) {
  int i=0, start = hash(k, st->M), index = start;
  while (full(st, index))
    if (KEYcmp(k, KEYget(&st->a[index]))==0)
      return st->a[index];
    else {
      i++;
      index = (start + c1*i + c2*i*i)%st->M;
    }
  return ITEMsetNull();
}
```

# Delete (soluzione 2)

```
void STdelete(ST st, Key k) {
  int i=0, start = hash(k, st->M), index = start;
  Item tmp;
  while (full(st, index))
    if (KEYcmp(k, KEYget(&st->a[index]))==0) break;
    else { i++;  index = (start + c1*i + c2*i*i)%st->M;
    }
  if (ITEMcheckNull(st->a[index])) return;
  st->a[index] = ITEMsetNull();
  st->N--; i++;
  index = (start + c1*i + c2*i*i)%st->M;
  while(full(st, index)) {
    tmp = st->a[index];
    st->a[index] = ITEMsetNull();
    st->N--; i++;
    STinsert(st, tmp);
    index = (start + c1*i + c2*i*i)%st->M;
  }
}
```

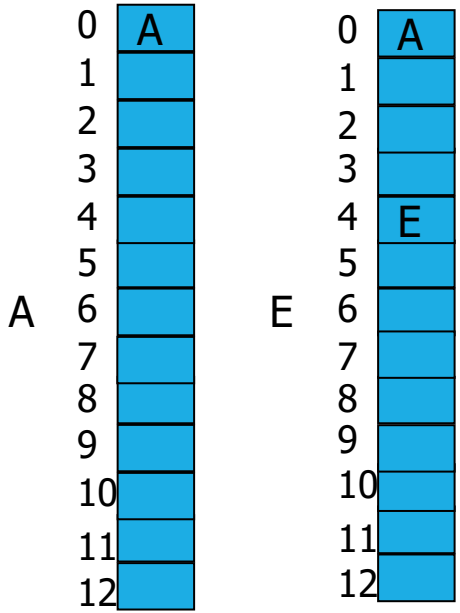
# Scelta di $c_1$ e $c_2$

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

# Esempio

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

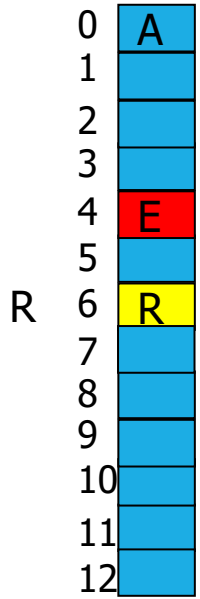



```
static int hash (Key k,int M){
  int h = 0, base = 127;
  for (; *k != '\0'; k++)
    h = (base * h + *k) % M;
  return h;
}
```

Funzione di quadratic probing
$c_1=1$ $c_2=1$
$i + i^2$

$\alpha = 6/13 < ½$

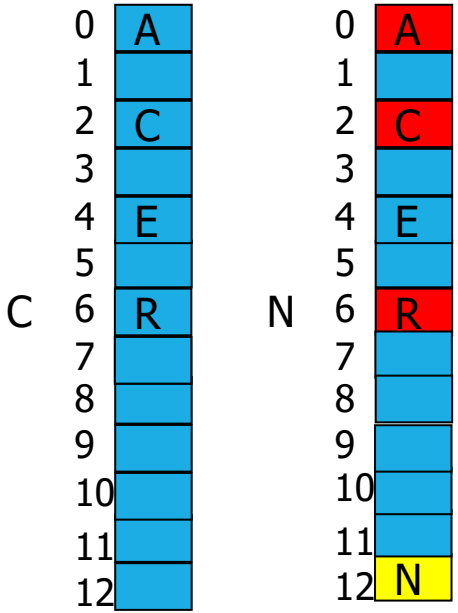
| | A | E | R | C | N | P |
|---|---|---|---|---|---|---|
| h(k) = | 0 | 4 | 4 | 2 | 0 | 2 |




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

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

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

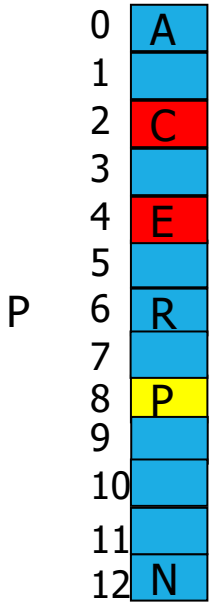






start = h('N') = 78 % 13 = 0 collisione
index = (0+1+$1^2$) % 13 = 2 collisione
index = (0+2+$2^2$) % 13 = 6 collisione
index = (0+3+$3^2$) % 13 = 12

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




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

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

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

# Double hashing

**Insert:**

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

**Importante**: bisogna che il nuovo valore

$$i = (i + j) \% M = (h_1(k) + h_2(k)) \% M$$

sia diverso dal vecchio valore di i, altrimenti si entra in un ciclo infinito. Per evitarlo:

- $h_2$ non deve mai ritornare 0
- $h_2 \% M$ non deve mai ritornare 0

Esempi di $h_1$ e $h_2$:

$h_1(k) = k \% M$ e M primo

$h_2(k) = 1 + k\%97$

$h_2(k)$ non ritorna mai 0 e $h_2\%M$ non ritorna mai 0 se $M > 97$.

```
static int hash1(Key k, int M) {
  int h = 0, base = 127;
  for ( ; *k != '\0'; k++) h = (base * h + *k) % M;
  return h;
}
static int hash2(Key k, int M) {
  int h = 0, base = 127;
  for ( ; *k != '\0'; k++) h = (base * h + *k);
  h = ((h % 97) + 1)%M;
  if (h==0) h=1;
  return h;
}
void STinsert(ST st, Item item) {
  int i = hash1(KEYget(&item), st->M);
  int j = hash2(KEYget(&item), st->M);
  while (full(st, i))
    i = (i+j)%st->M;
  st->a[i] = item;
  st->N++;
}
```

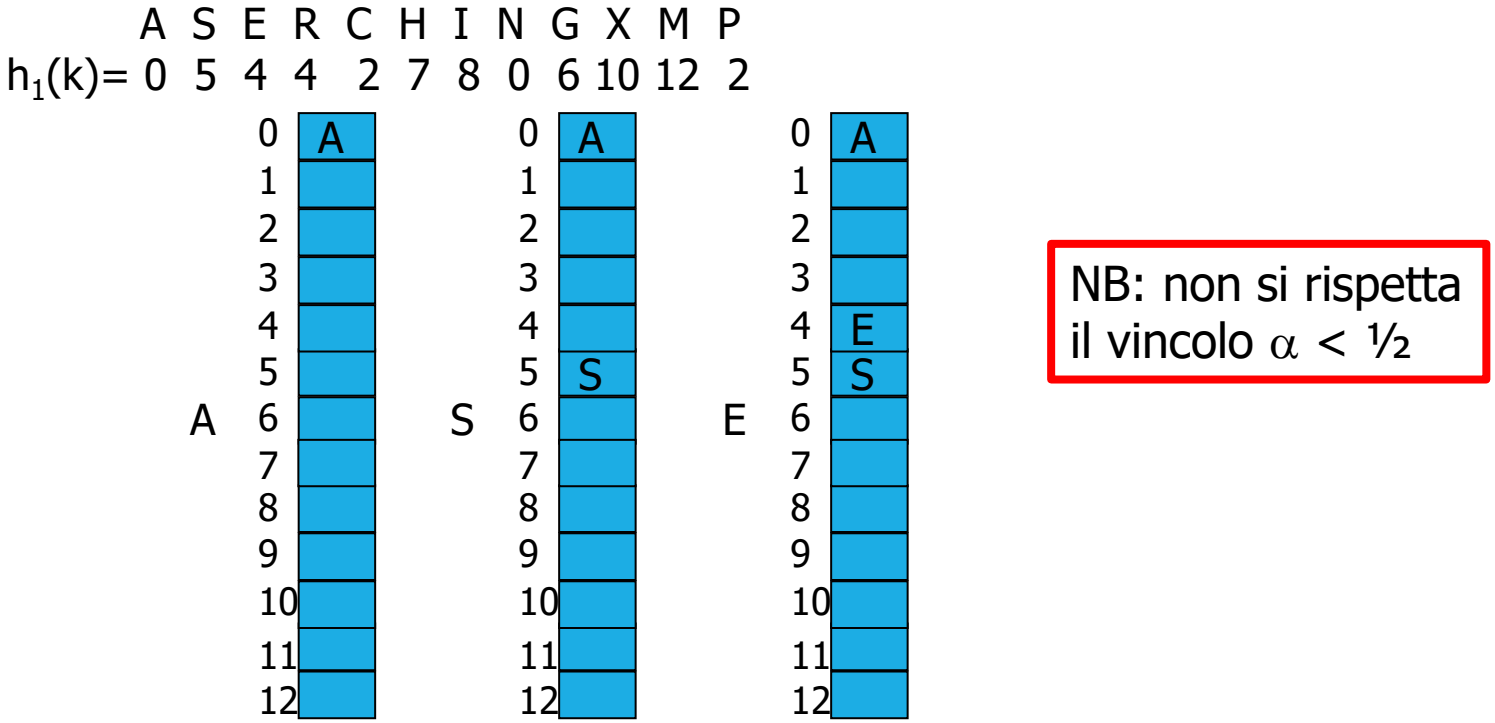
# Search

```
Item STsearch(ST st, Key k) {
  int i = hash1(k, st->M);
  int j = hash2(k, st->M);
    while (full(st, i))
      if (KEYcmp(k, KEYget(&st->a[i]))==0)
        return st->a[i];
      else
        i = (i+j)%st->M;
  return ITEMsetNull();
}
```

# Delete (soluzione 2)

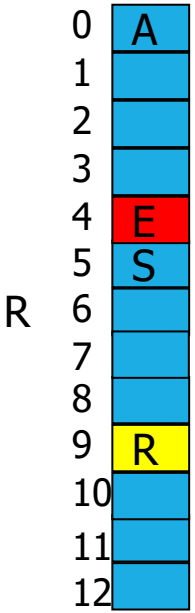
```
void STdelete(ST st, Key k) {
  int i = hash1(k, st->M), j = hash2(k); Item tmp;
  while (full(st, i))
    if (KEYcmp(k, KEYget(&st->a[i]))==0) break;
    else i = (i+j) % st->M;
  if (ITEMcheckNull(st->a[i]))
    return;
  st->a[i] = ITEMsetNull();
  st->N--;
  i = (i+j) % st->M;
  while(full(st, i)) {
    tmp = st->a[i];
    st->a[i] = ITEMsetNull();
    st->N--;
    STinsert(st, tmp);
    i = (i+j) % st->M;
  }
}
```

# Esempio



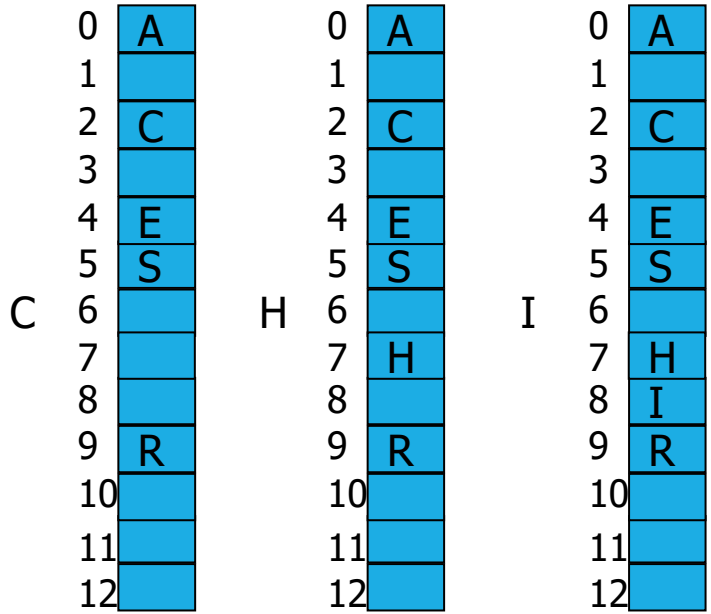

NB: non si rispetta il vincolo $\alpha < ½$

| | 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 |

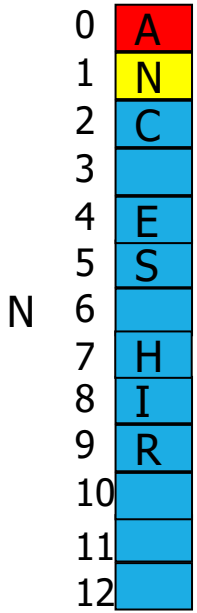



i = h('R') = 82 % 13 = 4 collisione
j = (82 % 97 +1) % 13) = 5
i = (4 + 5) % 13 = 9

| | 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 |

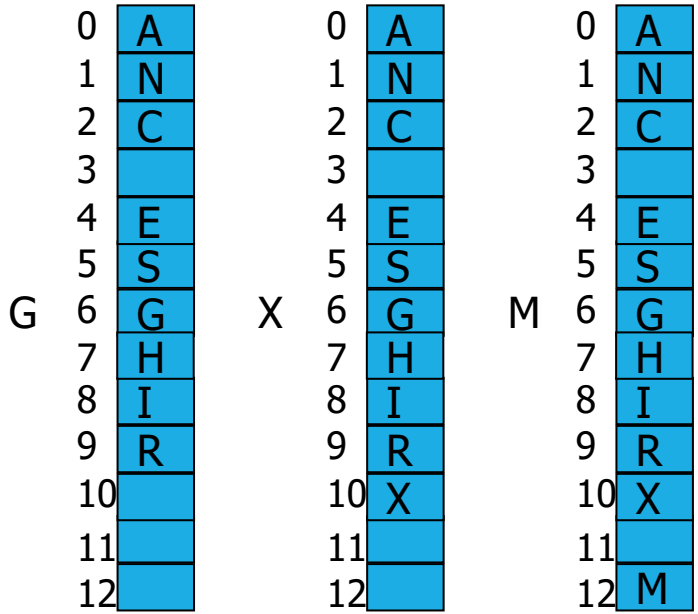


| | A | S | E | R | C | H | I | N | G | X | M | P |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $h_1(k)=$ | 0 | 5 | 4 | 4 | 2 | 7 | 8 | 0 | 6 | 10 | 12 | 2 |

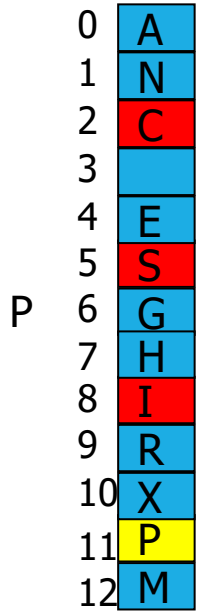



i = h('N') = 78 % 13 = 0 collisione
j = (78 % 97 +1) % 13) = 1
i = (0 + 1) % 13 = 1

| | 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 |

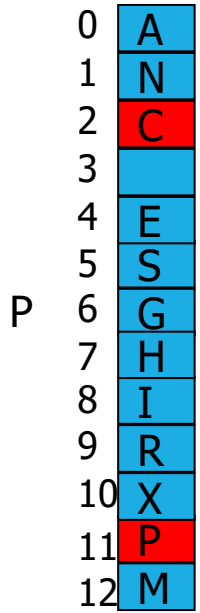


| | A | S | E | R | C | H | I | N | G | X | M | P |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $h_1(k)=$ | 0 | 5 | 4 | 4 | 2 | 7 | 8 | 0 | 6 | 10 | 12 | 2 |

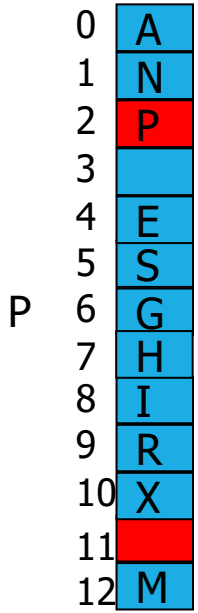



i = h('P') = 80 % 13 = 2 collisione
j = (80 % 97 +1) % 13) = 3
i = (2 + 3) % 13 = 5 collisione
i = (5 + 3) % 13 = 8 collisione
i = (8 + 3) % 13 = 11

| | 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 |




In inserzione c'è stata collisione tra 'P' e 'C'. Non ci sono state altre collisioni.
Se si cancella 'C', 'P' prende il suo posto.



# Complessità del double hashing

Ipotesi:

- hashing semplice uniforme
- probing uniforme.

Tentativi di “probing” per la ricerca:

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

| $\alpha$ | 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 |

# Confronto 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

# Esercizi di teoria

- 6. Tabelle di hash
  - 6.1 Hashing
  - 6.2 Linear chaining
  - 6.3 Open addressing con linear probing
  - 6.3 Open addressing con quadratic probing
  - 6.3 Open addressing con double hashing