

Le tabelle di hash



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

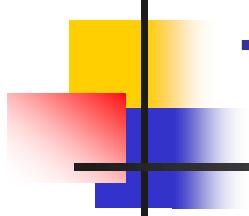


Tabelle di hash

Finora gli algoritmi di ricerca si erano basati sul confronto.

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

Limiti delle tabelle ad accesso diretto:

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

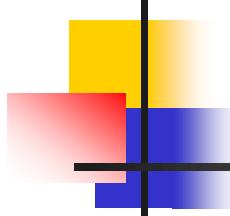


Tabella di hash:

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



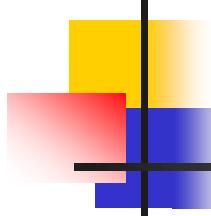
collisione.

Usate per inserzione, ricerca, cancellazione, non per ordinamento e selezione.

ADT di I categoria ST

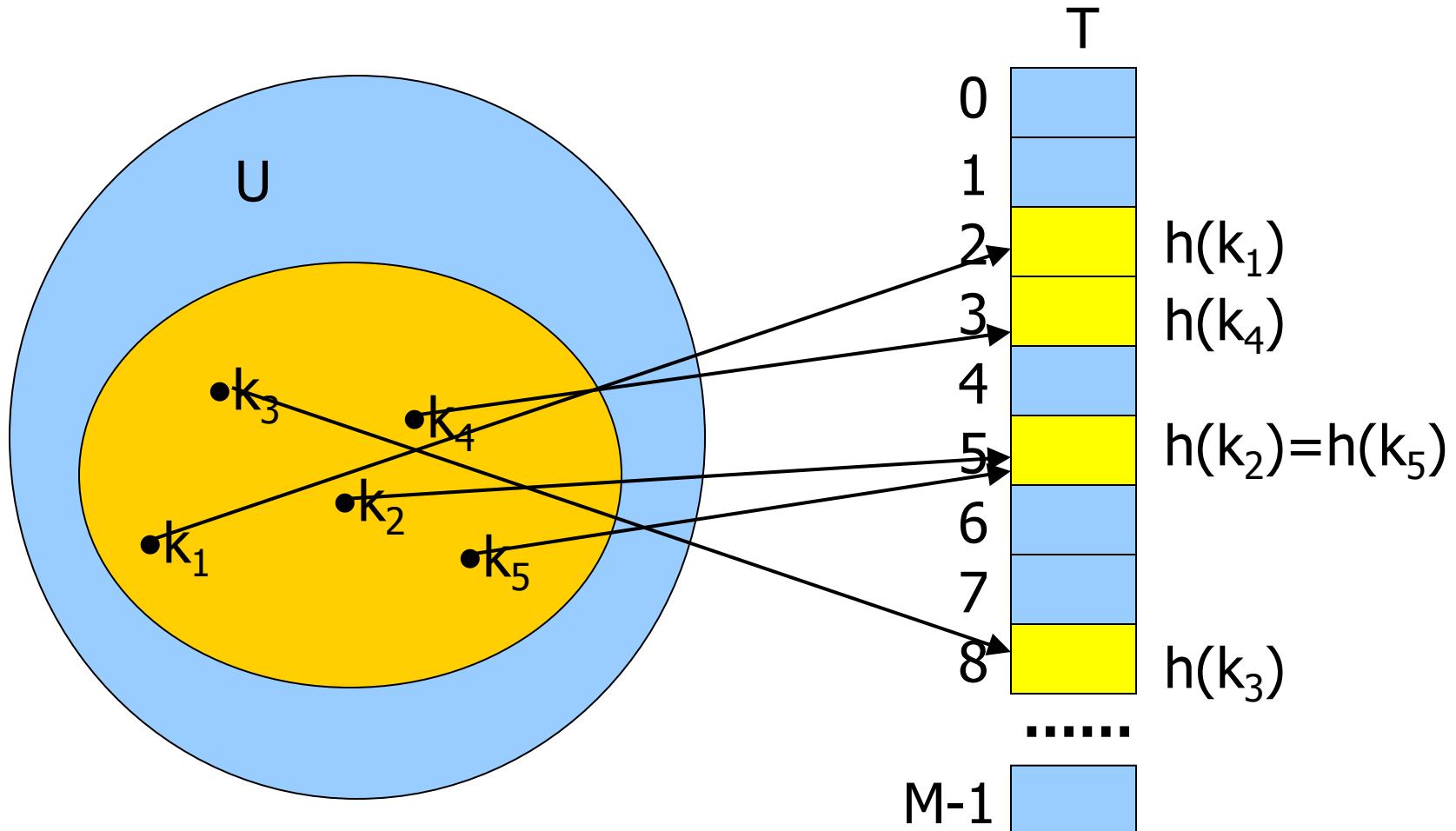
ST.h

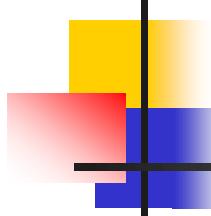
```
typedef struct symboltable *ST;  
  
ST      STinit(int) ;  
void   STinsert(ST, Item) ;  
Item   STsearch(ST, Key) ;  
void   STdelete(ST, Key) ;  
void   STdisplay(ST) ;
```



Funzione di hash

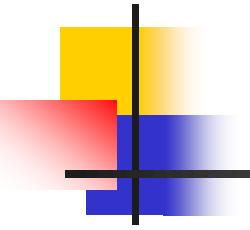
- La tabella di hash ha dimensione M e contiene $|K|$ elementi ($|K| << |U|$)
- La tabella di hash ha indirizzi nell'intervallo $[0 \dots M-1]$
- La funzione di **hash** h mette in corrispondenza una chiave k in un indirizzo della tabella $h(k)$
$$h: U \rightarrow \{ 0, 1, \dots, M-1 \}$$
- L'elemento x viene memorizzato all'indirizzo $h(k)$ dato dalla sua chiave k (attenzione alla gestione delle collisioni!).





Progetto della funzione di hash

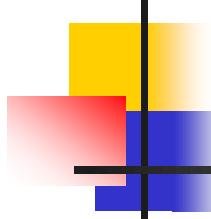
Funzione ideale: hashing uniforme semplice:
se le chiavi k sono equiprobabili, allora i
valori di $h(k)$ devono essere equiprobabili.



In pratica, le chiavi k non sono equiprobabili, anzi sono correlate e 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 ($s \leq k < t$):

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

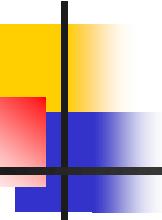
```
int hash(float k, int M, float s, float t) {  
    return ((k-s)/(t-s))*M;  
}
```

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 di w bit; M numero primo

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

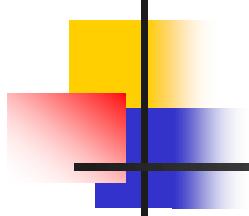
```
int hash(int k, int M){  
    return (k%M);  
}
```

Esempio:

$$M = 19$$

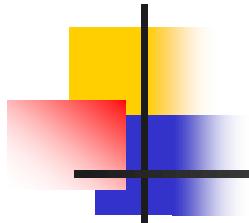
$$k = 31$$

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



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 $M = 10^n$.



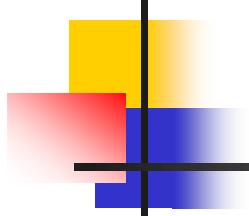
Metodo moltiplicativo-modulare

chiavi: numeri interi:

- data costante $0 < A < 1$

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

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

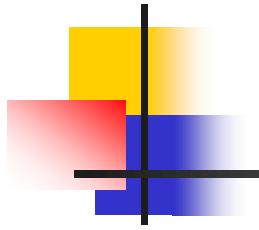


Metodo modulare:

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

M numero primo

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

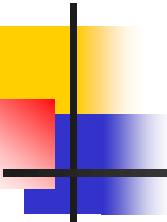


Esempio:

$$\begin{aligned}\text{stringa now} &= 'n'*128^2 + 'o'*128 + 'w' \\ &= 110*128^2 + 111*128 + 119 \\ k &= 1816567\end{aligned}$$

$$k = 1816567 \quad M = 19$$

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



Metodo modulare:

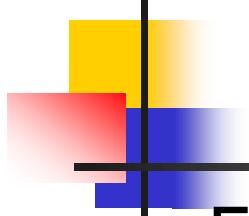
chiavi: stringhe alfanumeriche lunghe come interi derivati dalla valutazione di polinomi in una database con il metodo di Horner: ad esempio

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

Come prima:

M numero primo

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



Esempio: stringa averylongkey con base 128 (ASCII)

$k =$

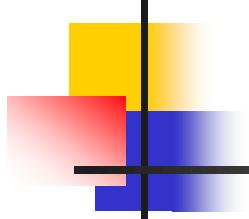
$$97*128^{11}+118*128^{10}+101*128^9+114*128^8+121*128^7+108*128^6+111*128^5+110*128^4+103*128^3+107*128^2+101*128^1+121*128^0$$

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

Con il metodo di Horner:

$k =$

$$\begin{aligned} & (((((((((97*128+118)*128+101)*128+114)*128+121)*128+108)* \\ & 128+111)*128+110)*128+103)*128+107)*128+101)*128+121 \end{aligned}$$



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)

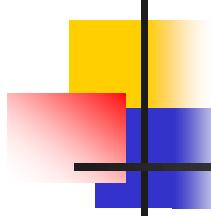
con lo scopo di ottenere una distribuzione abbastanza uniforme (probabilità di collisione tra 2 chiavi diverse prossima a $1/M$).

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;  
}
```



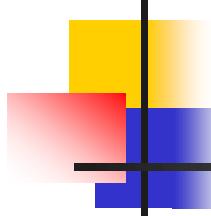
Collisioni

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:

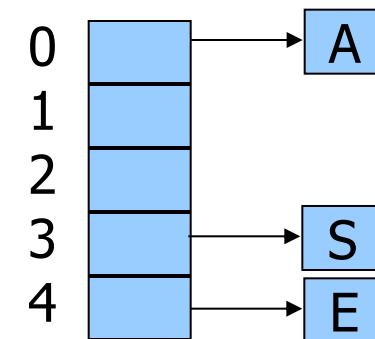
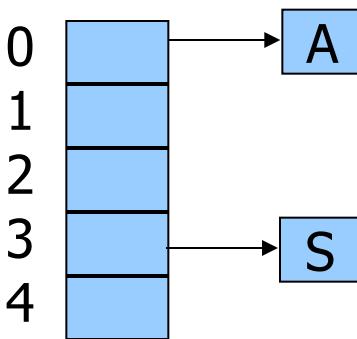
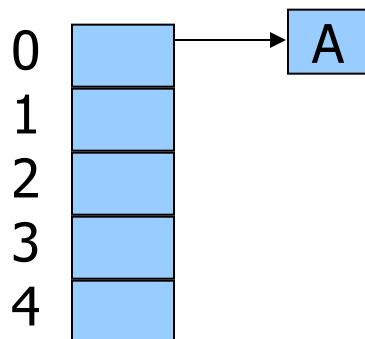
- il più piccolo primo $M \geq$ numero di chiavi max / 5
(o 10) così che la lunghezza media delle liste sia 5
(o 10)

Esempio

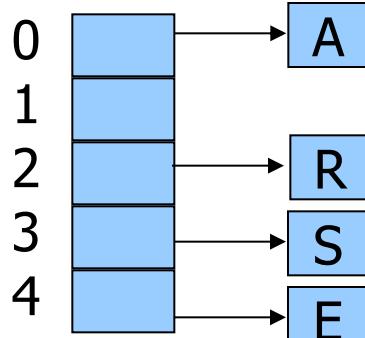
```
M = 5;
int hash (Key k, int M) {
    int h = 0, base = 127;
    for (; *k != '\0'; k++)
        h = (base * h + *k) % M;
    return h;
}
```

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

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

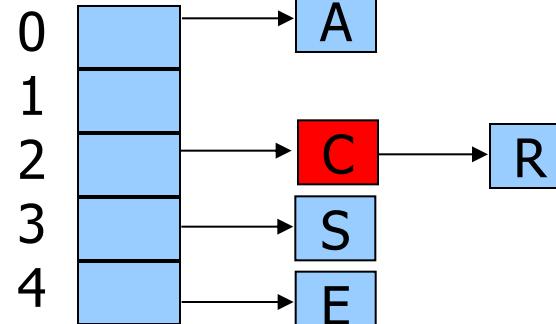


A

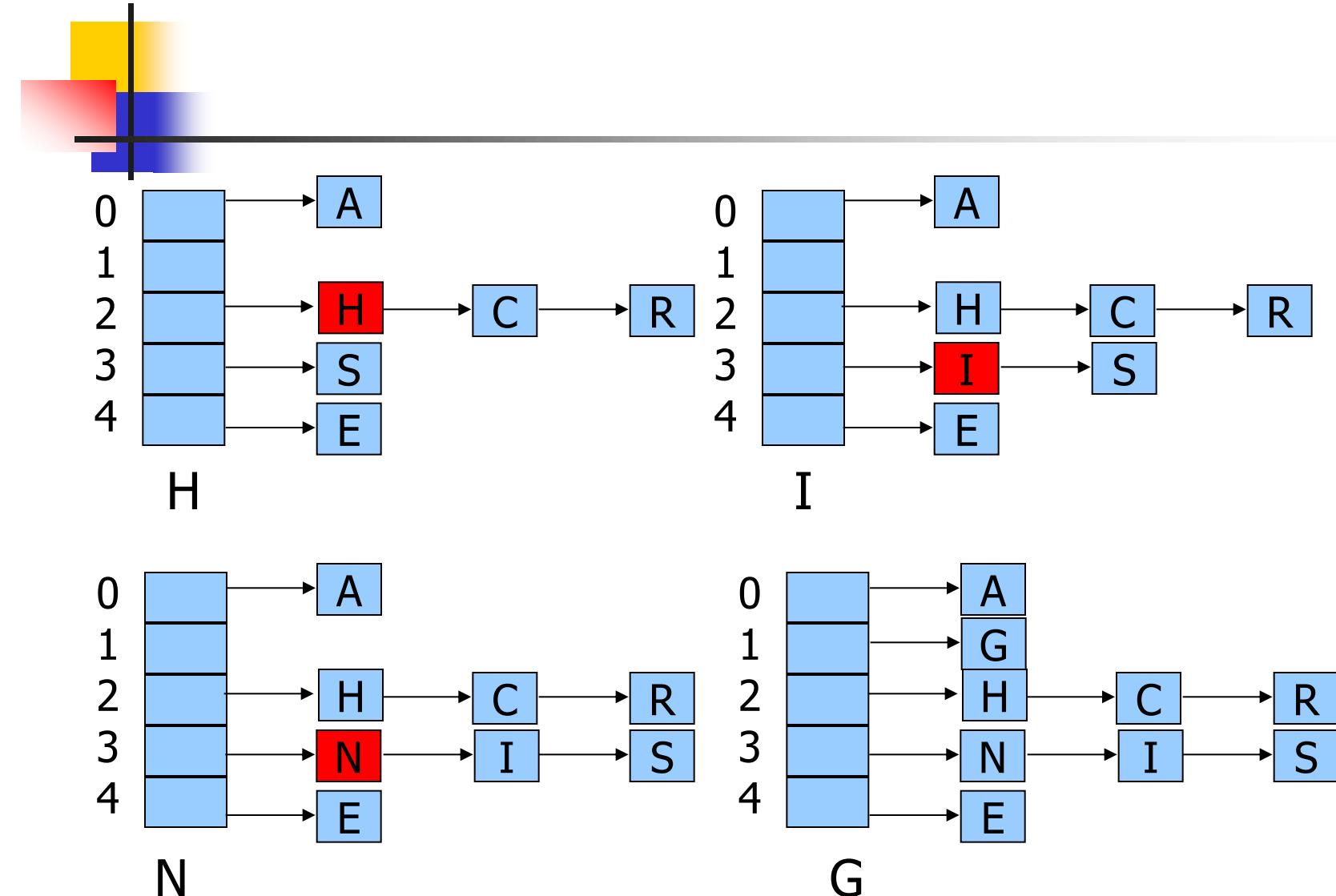


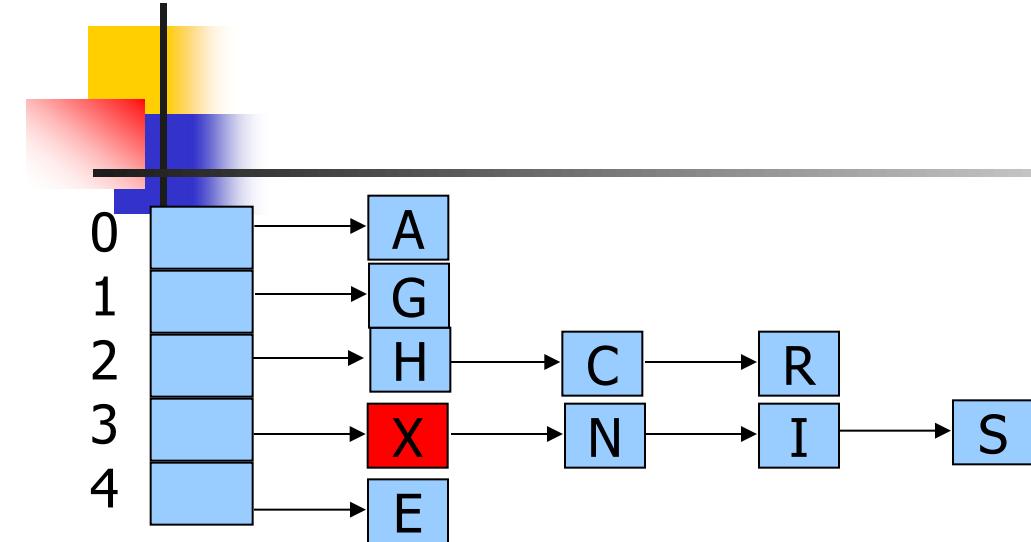
R

S

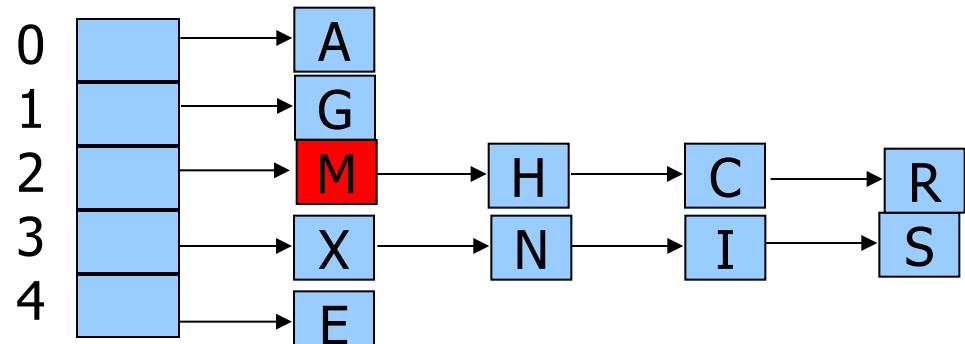


C

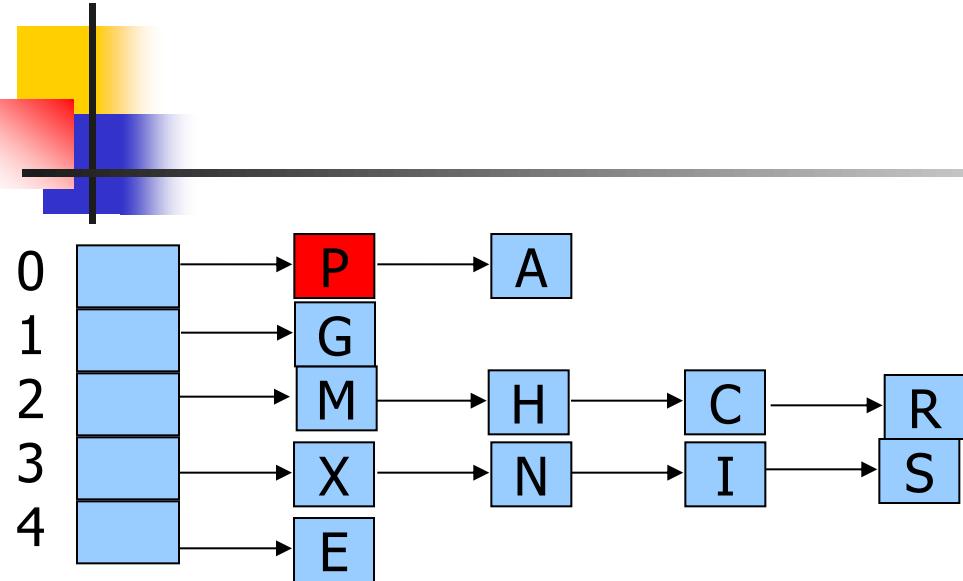




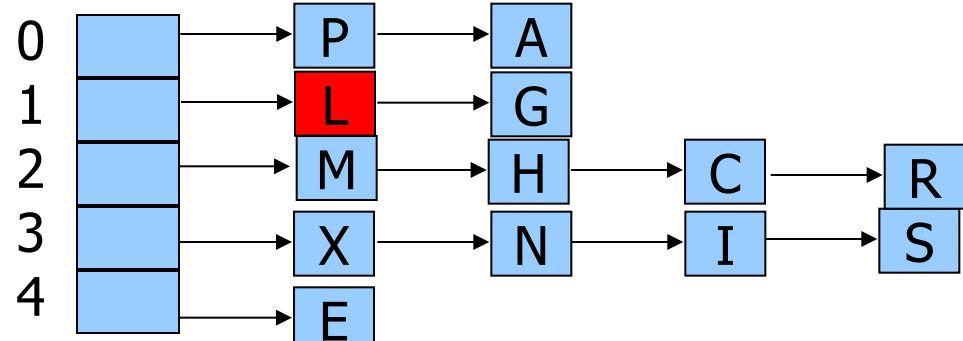
X



M



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

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

nodo sentinella

```

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(ITEMsetvoid(), NULL);
    for (i=0; i < st->M; i++) st->heads[i] = st->z;
    return st;
}
int hash(Key v, int M) {
    int h = 0, base = 127;
    for ( ; *v != '\0'; v++) h = (base * h + *v) % M;
    return h;
}
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 item) {
    int i = hash(KEYget(item), st->M);
    st->heads[i] = NEW(item, st->heads[i]);
}

Item searchR(link t, Key k, link z) {
    if (t == z) return ITEMsetvoid();
    if ((KEYcompare(KEYget(t->item), k))==0) return t->item;
    return searchR(t->next, k, z);
}

Item STsearch(ST st, Key k) {
    return searchR(st->heads[hash(k, st->M)], k, st->z);
}

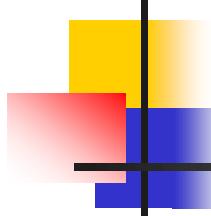
link deleteR(link x, Key k) {
    if (x == NULL) return NULL;
    if ((KEYcompare(KEYget(x->item), k))==0) {
        link t = x->next; free(x); return t;
    }
    x->next = deleteR(x->next, k);
    return x;
}

```

```
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;
    ITEMshow(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

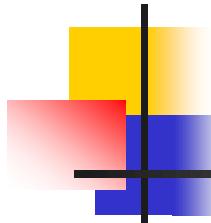
Hashing semplice uniforme:

$h(k)$ 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 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:

$$\left. \begin{array}{l} N \leq M \\ \alpha \leq 1 \end{array} \right\}$$

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

chiave tentativo (0...M-1)

Open addressing

Determinazione della dimensione M della tabella:

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

ST.h

```
typedef struct symboltable *ST;

ST      STinit(int) ;
void   STinsert(ST, Item) ;
int    STcount(ST st) ;
int    STempty(ST st) ;
Item   STsearch(ST, Key) ;
void   STdelete(ST, Key) ;
void   STdisplay(ST) ;
```

ST.c

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

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

ST STinit(int maxN) {
    int i;
    ST 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] = ITEMsetvoid();
    return st;
}
```

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

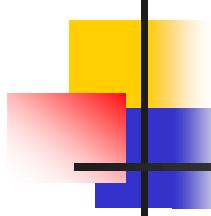
int hashU(Key k, int M) {
    int h, a = 31415, b = 27183;
    for ( h = 0; *k != '\0'; k++, a = a*b % (M-1))
        h = (a*h + *k) % M;
    return h;
}

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

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

```
int full(ST st, int i) {
    if (ITEMcheckvoid(st->a[i]))
        return 0;
    else
        return 1;
}

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



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++;  
}
```

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 (KEYcompare(k, KEYget(st->a[i]))==0)  
            return st->a[i];  
        else  
            i = (i+1)%st->M;  
    return ITEMsetvoid();  
}
```

Esempio

```
M = 13;  
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
 $h(k) = 0 \ 5 \ 4 \ 4 \ 2 \ 7 \ 8 \ 0 \ 6 \ 10 \ 12 \ 2$

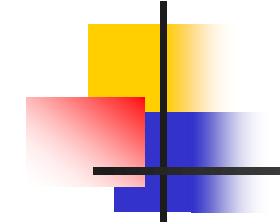
0	A	0
1		1
2		2
3		3
4		4
5		5
A	6	S
6		6
7		7
8		8
9		9
10		10
11		11
12		12

A vertical scale labeled **E** on the left side. The scale has numerical markings from 0 to 12, with horizontal grid lines extending across the page at each integer value. The labels are positioned to the left of the scale line.

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

A vertical scale with numerical markings from 0 to 12. The scale is color-coded: blue for values 0 through 3, red for 4 and 5, yellow for 6, blue for 7 through 10, and light blue for 11 and 12. The letter 'R' is positioned to the left of the scale at the 6 mark. The letters 'E' and 'S' are positioned above the 4 and 5 marks respectively, covering the red section of the scale.

NB: non si rispetta il vincolo $\alpha < \frac{1}{2}$

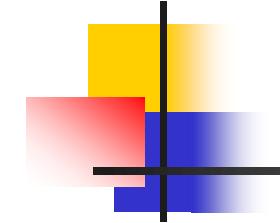


A S E R C H I N G X M P
 $h(k) = 0 \ 5 \ 4 \ 4 \ 2 \ 7 \ 8 \ 0 \ 6 \ 10 \ 12 \ 2$

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

} cluster

$$\begin{aligned} i &= h('R') = 82 \% 13 = 4 \text{ collisione} \\ i &= (4+1) \% 13 = 5 \text{ collisione} \\ i &= (5+1) \% 13 = 6 \end{aligned}$$



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

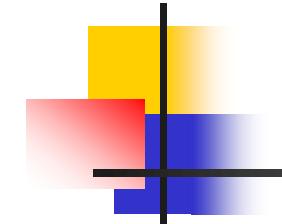
$$h(k) = 0 \ 5 \ 4 \ 4 \ 2 \ 7 \ 8 \ 0 \ 6 \ 10 \ 12 \ 2$$

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

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

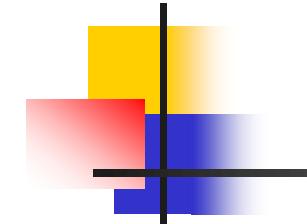
	A											
0												
1												
2	C											
3												
4	E											
5	S											
6	R											
7	H											
8	I											
9												
10												
11												
12												

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



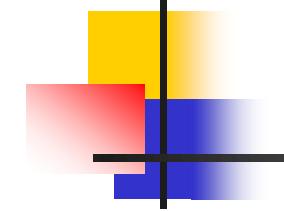
	A	S	E	R	C	H	I	N	G	X	M	P
$h(k) =$	0	5	4	4	2	7	8	0	6	10	12	2
N	0	A										
	1	N										
	2	C										
	3											
	4	E										
	5	S										
	6	R										
	7	H										
	8	I										
	9											
	10											
	11											
	12											

$$i = h('N') = 78 \% 13 = 0 \quad \text{collisione}$$
$$i = (0+1) \% 13 = 1$$



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

	0	A	
G	1	N	$i = h('G') = 71 \% 13 = 6$ collisione
	2	C	$i = (6+1) \% 13 = 7$ collisione
	3		$i = (7+1) \% 13 = 8$ collisione
	4	E	
	5	S	$i = (8+1) \% 13 = 9$
	6	R	
	7	H	
	8	I	
	9	G	
	10		
	11		
	12		



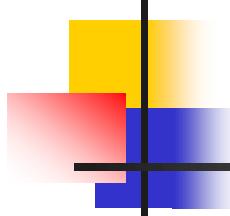
A S E R C H I N G X M P
 $h(k) = 0 \ 5 \ 4 \ 4 \ 2 \ 7 \ 8 \ 0 \ 6 \ 10 \ 12 \ 2$

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

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

collisione

$$i = h('P') = 80 \% 13 = 2$$
$$i = (2+1) \% 13 = 3$$



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

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

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

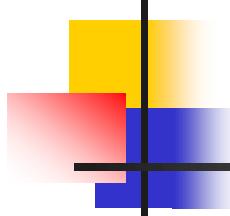


0	A
1	
2	C
3	
4	
5	S
6	R
7	H
8	
9	
10	
11	
12	



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

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



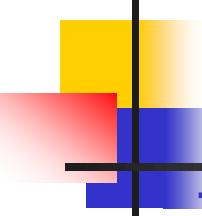
Complessità con l'ipotesi di:

- hashing semplice uniforme
- probing uniforme.

Tentativi in media di “probing” per la ricerca:

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

	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)
- $\text{index} = (\text{h}'(\text{k}) + c_1 i + c_2 i^2) \% M$
- se libero, inserisci chiave, altrimenti incrementa i e ripeti fino a cella vuota.

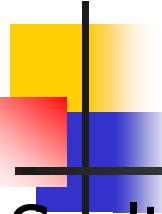
```
#define c1 1
#define c2 1
void STinsert(ST st, Item item) {
    int i = 0, start = hash(KEYget(item), st->M), index = start;
    while (full(st, index)) {
        i++;
        index = (start + c1*i + c2*i*i)%st->M;
    }
    st->a[index] = item;
    st->N++;
}
```

Search:

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

Delete:

```
void STdelete(ST st, Key k) {
    int i=0, j, start = hash(k, st->M), index = start;
    Item tmp;
    while (full(st, index))
        if (KEYcompare(k, KEYget(st->a[index]))==0) break;
    else {
        i++;
        index = (start + c1*i + c2*i*i)%st->M;
    }
    if (ITEMcheckvoid(st->a[index])) return;
    st->a[index] = ITEMsetvoid();
    st->N--;
    i = 1;
    for (j=(index+c1*i+c2*i*i)%st->M;
         full(st,j);
         j=(index+c1*i+ c2*i*i)%st->M,st->N--){
        tmp = st->a[j]; st->a[j] = ITEMsetvoid();
        STinsert(st, tmp); i++;
    }
}
```



Scelta di c_1 e c_2 :

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

garantiscono che, con inizialmente $start = h(k)$ e poi $index = (start + c_1i + c_2i^2) \text{ modulo } M$ 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

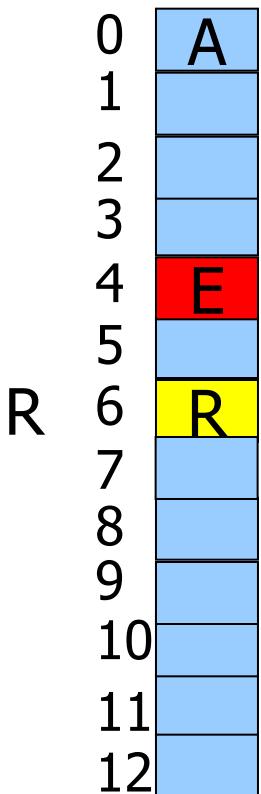
A	0	A
	1	
	2	
	3	
	4	
	5	
	6	
	7	
	8	
	9	
	10	
	11	
	12	

```
M = 13;  
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
 $c_1 = 1$ $c_2 = 1$
 $i + i^2$

$$\alpha = 6/13 < 1/2$$

$$\begin{array}{ccccccc} & A & E & R & C & N & P \\ h(k) = & 0 & 4 & 4 & 2 & 0 & 2 \end{array}$$



start = $h('R') = 82 \% 13 = 4$ collisione
index = $(4+1+1^2) \% 13 = 6$

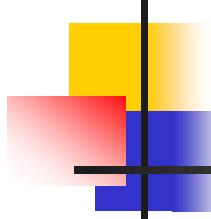
$$h(k) = \begin{matrix} A & E & R & C & N & P \\ 0 & 4 & 4 & 2 & 0 & 2 \end{matrix}$$

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

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

$$\begin{array}{ccccccc} & A & E & R & C & N & P \\ h(k) = & 0 & 4 & 4 & 2 & 0 & 2 \end{array}$$

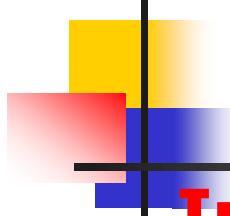
P	0	A	
	1	C	start = $h('P') = 80 \% 13 = 2$ collisione
	2	E	index = $(2+1+1^2) \% 13 = 4$ collisione
	3		index = $(2+2+2^2) \% 13 = 8$
	4		
	5		
	6	R	
	7		
	8	P	
	9		
	10		
	11		
	12	N	



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 $M = 2 * \text{max}$, $\alpha < 1$



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 :

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

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

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

```
int hash1(Key k, int M) {  
    int h = 0, base = 127;  
    for ( ; *k != '\0'; k++)  
        h = (base * h + *k) % M;  
    return h;  
}  
  
int hash2(Key k, int M) {  
    int h = 0, base = 127;  
    for ( ; *k != '\0'; k++)  
        h = (base * h + *k);  
    h = ((h % 97) + 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++;
}

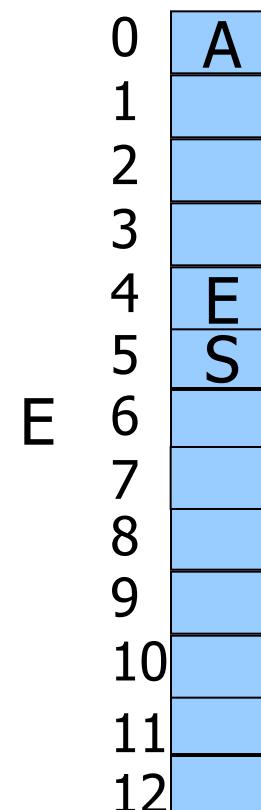
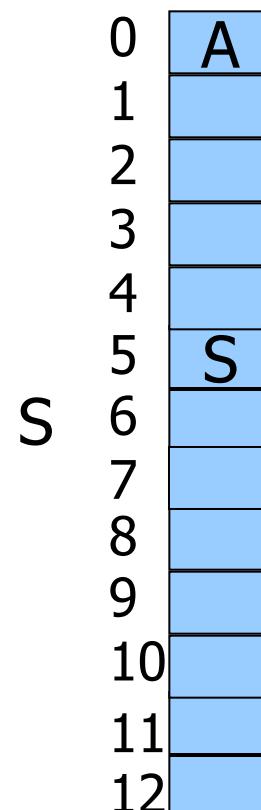
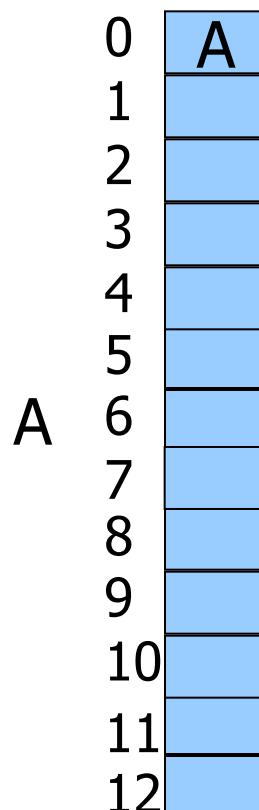
Item STsearch(ST st, Key k) {
    int i = hash1(k, st->M);
    int j = hash2(k, st->M);
    while (full(st, i))
        if (KEYcompare(k, KEYget(st->a[i]))==0)
            return st->a[i];
        else
            i = (i+j)%st->M;
    return ITEMsetvoid();
}

```

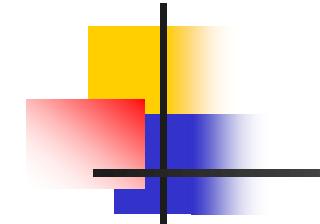
Esempio

```
M = 13;  
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
 $h_1(k) = 0 \ 5 \ 4 \ 4 \ 2 \ 7 \ 8 \ 0 \ 6 \ 10 \ 12 \ 2$



NB: non si rispetta
il vincolo $\alpha < 1/2$

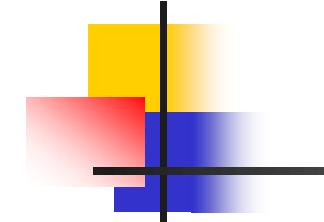


A S E R C H I N G X M P

$h_1(k) = 0 \ 5 \ 4 \ 4 \ 2 \ 7 \ 8 \ 0 \ 6 \ 10 \ 12 \ 2$

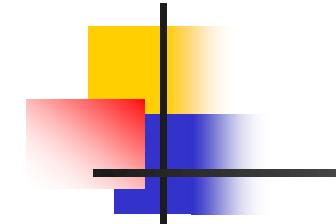
R	0	A
	1	
	2	
	3	
	4	E
	5	S
	6	
	7	
	8	
	9	R
	10	
	11	
	12	

$$\begin{aligned} i &= h('R') = 82 \% 13 = 4 \quad \text{collisione} \\ j &= (82 \% 97 + 1) \% 13 = 5 \\ i &= (4 + 5) \% 13 = 9 \end{aligned}$$



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



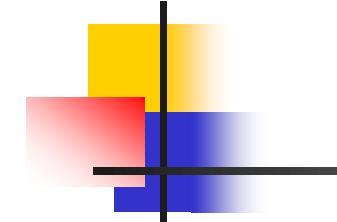
A S E R C H I N G X M P

$h_1(k) = 0 \ 5 \ 4 \ 4 \ 2 \ 7 \ 8 \ 0 \ 6 \ 10 \ 12 \ 2$

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

$$\begin{aligned} i &= h('N') = 78 \% 13 = 0 \quad \text{collisione} \\ j &= (78 \% 97 + 1) \% 13 = 1 \\ i &= (0 + 1) \% 13 = 1 \end{aligned}$$

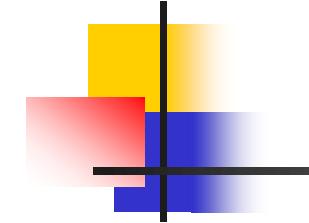
N



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	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
	6	G	6	G	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

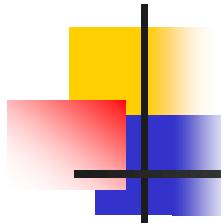


A S E R C H I N G X M P

$h_1(k) = 0 \ 5 \ 4 \ 4 \ 2 \ 7 \ 8 \ 0 \ 6 \ 10 \ 12 \ 2$

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

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 = $(8 + 3) \% 13 = 11$



Complessità del double hashing

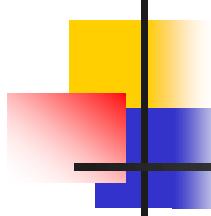
Ipotesi:

- 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



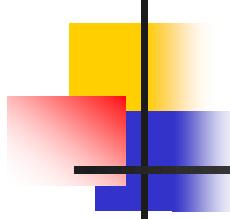
Cfronto tra alberi e tabelle di hash

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