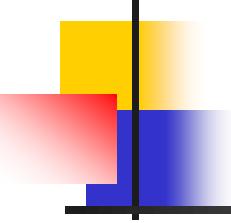


# L'ADT tabella di simboli

Definizione: ADT che supporta operazioni di:

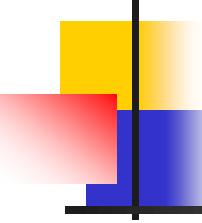
- **insert**: inserisci un elemento (STinsert)
- **search**: ricerca dato con certa chiave (STsearch)
- **delete**: cancella il dato con una certa chiave (STdelete).

Talora la tabella di simboli è detta **dizionario**.



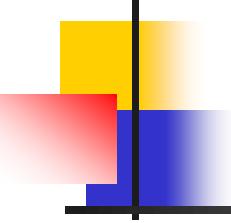
## Altre operazioni:

- inizializzare la tabella
- distruggere la tabella
- contare il numero di dati
- visualizzare della tabella
- se sulla chiave è definita una relazione d'ordine:
  - ordinare la tabella
  - selezionare la chiave di rango  $r$  ( $r$ -esima più piccola chiave)



# Applicazioni delle tabelle di simboli

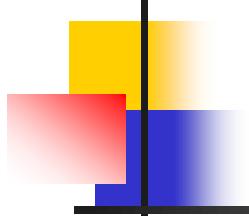
<b>Applicazione</b>	<b>scopo: trovare</b>	<b>chiave</b>	<b>valore ritornato</b>
dizionario	la definizione	parola	definizione
indice libro	pagine rilevanti	termine	lista pagine
DNS	indirizzo IP dato URL	URL	IP address
DNS inverso	URL dato indirizzo IP	IP address	URL
file system	file su disco	nome file	localizzazione disco
web search	pagine web	parola chiave	lista di pagine



# ADT I classe tabella di simboli

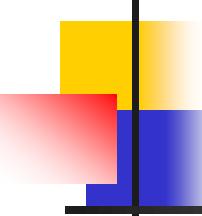
## **ST.h**

```
typedef struct symboltable *ST;  
  
ST STinit(int maxN);  
void STfree(ST st);  
int STcount(ST st);  
void STinsert(ST st, Item val);  
Item STsearch(ST st, Key k);  
void STdelete(ST st, Key k);  
Item STselect(ST st, int r);  
void STdisplay(ST st);
```



## Possibili versioni dell'ADT tabella di simboli:

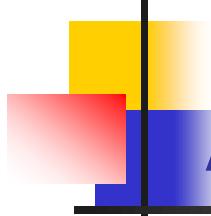
- tavole ad accesso diretto
  - strutture lineari (vettore/lista ordinato/non ordinato)
  - strutture ad albero (alberi binari di ricerca (BST) e loro varianti)
  - tavole di hash.
- casi trattati



# Complessità

Complessità:	caso peggiore		
	inserim.	ricerca	selezione
tab. acc. dir.	1	1	maxN
array non ord.	1	n	
array ord. ric. lin.	n	n	1
array ord. e ric. bin.	n	logn	1
lista non ord.	1	n	
lista ord.	n	n	n
BST	n	n	n
RB-tree	logn	logn	nlogn
hashing	1	n	

		caso	medio
	inserim.	hit	miss
tab. acc. dir.	1	1	1
array non ord.	1	$n/2$	$n$
array ord. e ric. lin.	$n/2$	$n/2$	$n/2$
array ord. e ric. bin.	$n/2$	$\log n$	$\log n$
lista non ord.	1	$n/2$	$n$
lista ord.	$n/2$	$n/2$	$n/2$
BST	$\log n$	$\log n$	$\log n$
RB-tree	$\log n$	$\log n$	$\log n$
hashing	1	1	1

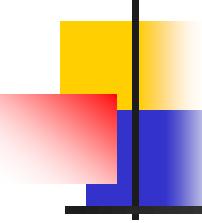


# ADT I classe Tabella ad accesso diretto

---

- insieme universo  $U$  con  $M = \text{card}(U) = \text{maxN}$  elementi
- corrispondenza biunivoca tra ciascuna delle chiavi  $k \in U$  e gli interi tra 0 e  $M-1$  (funzione `int getindex(Key k)`). L'intero funge da **indice** in un vettore

- vettore  $st \rightarrow a [maxN]$ :
  - se la chiave  $k$  è nella tabella, essa è in posizione  $st \rightarrow a [\text{getindex}(k)]$ , altrimenti  $st \rightarrow a [\text{getindex}(k)]$  contiene l'elemento vuoto
- si memorizza un insieme di  $N$  chiavi ( $N \leq M$ ) e  $\text{card}(K) = st \rightarrow \text{size}$ .



## Esempi di getindex

- se le chiavi sono le lettere maiuscole dell'alfabeto inglese A..Z ( $M = 26$ )

```
int getindex(key k) {  
    int i;  
    i = k - 'A';  
    return i;  
}
```

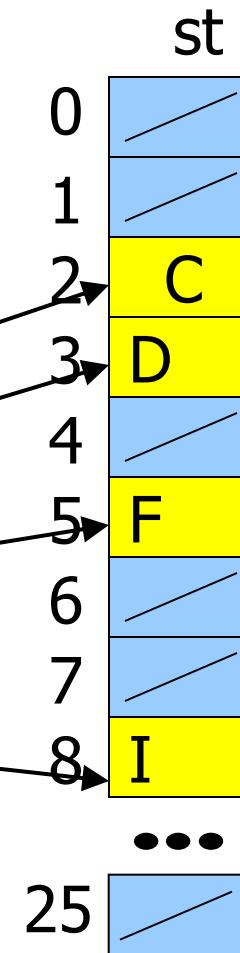
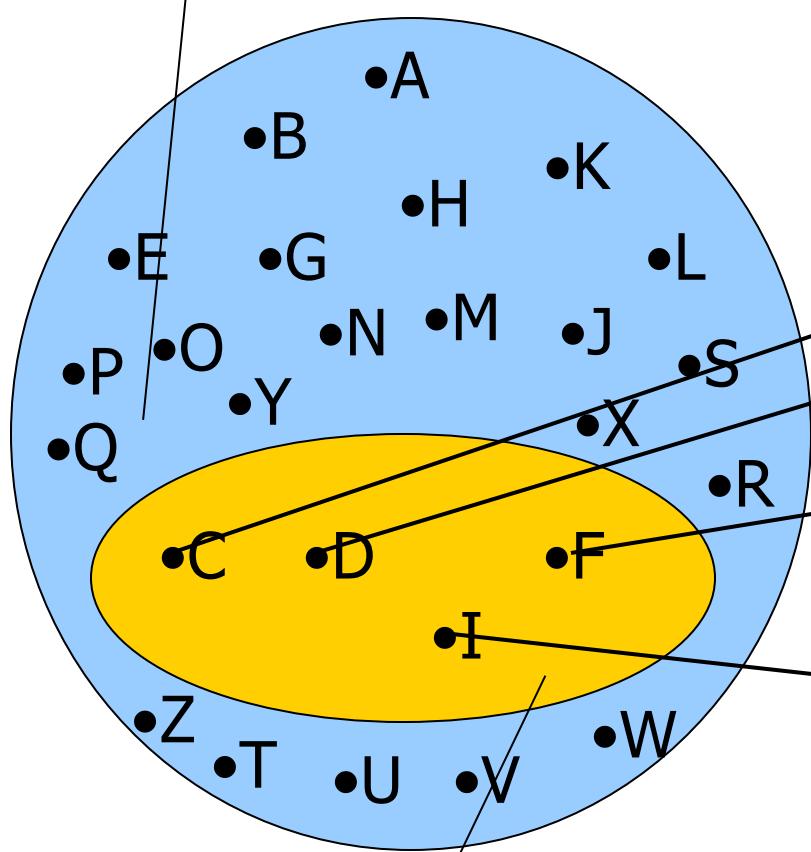
- se le chiavi sono interi tra 0 e  $M-1$

```
int getindex(key k) {  
    int i;  
    i = (int) k;  
    return i;  
}
```

- se le chiavi sono stringhe di lunghezza k fissa e corta, si trasformano in intero valutandole come polinomio di grado k in base 26 ( $M=26^k$ )

```
int getindex(key k) {  
    int i = 0, b = 26;  
  
    for ( ; *k != '\0'; k++)  
        i = (b * i + (*k - ((int) 'A')));  
    return i;  
}
```

**U (universo delle chiavi)**  
 $M = \text{card}(U) = \text{maxN} = 26$



**N (chiavi usate)**

## ST.c

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

## ST.c

```
void STfree(ST st) {
    int i;
    for (i=0; i<st->M; i++)
        ITEMfree(st->a[i]);
    if (st->a != NULL)
        free(st->a);
    free(st);
}

void STinsert(ST st, Item val) {
    int index = getIndex(KEYget(val));
    st->a[index] = val;
    st->N++;
}
```

## **ST.c**

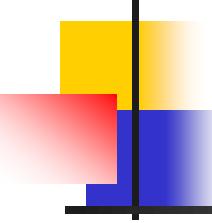
```
Item STsearch(ST st, Key k) {
    int index = getindex(k);
    if (ITEMcheckvoid(st->a[index]))
        return ITEMsetvoid();
    return st->a[index];
}

void STdelete(ST st, Key k) {
    st->a[getindex(k)] = ITEMsetvoid();
    st->N--;
}
```

## ST.c

```
Item STselect(ST st, int r) {
    int i;
    for (i = 0; i < st->M; i++)
        if ((ITEMcheckvoid(st->a[i]))==0) &&
            (r-- == 0))
            return st->a[i];
    return ITEMsetvoid();
}

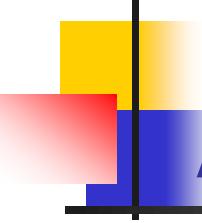
void STdisplay(ST st){
    int i;
    for (i = 0; i < st->M; i++)
        if (ITEMcheckvoid(st->a[i]))==0)
            ITEMshow(st->a[i]);
}
```



## Vantaggi/svantaggi

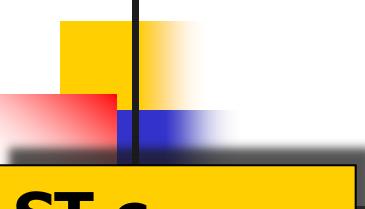
---

- Complessità delle operazioni di inserimento, ricerca e cancellazione:  $T(n) = \Theta(1)$
- Complessità delle operazioni di inizializzazione e selezione:  $T(n) = \Theta(M)$
- Occupazione di memoria  $S(n) = \Theta(\text{card}(U)) = \Theta(M)$ 
  - applicabile per M piccolo
  - spreco di memoria per  $N \ll M$
- Molto usate in pratica per trasformare chiavi in interi e viceversa a costo unitario.



# ADT I classe tabella di simboli (vettore)

- Vettore non ordinato:
  - inserzione in fondo per avere complessità  $O(1)$
  - realloc per ridimensionare la tabella se piena in inserzione
  - ricerca lineare preliminare alla cancellazione
  - la selezione non ha senso (non è ordinato)
  
- Vettore ordinato:
  - inserzione con scansione con complessità  $O(N)$
  - ricerca dicotomica preliminare alla cancellazione.



## ST.c

```
struct symboltable {Item *a;int maxN;int size;};

ST STinit(int maxN) {
    ST st;
    st = malloc(sizeof(*st));
    st->a = malloc(maxN * sizeof(Item) );
    st->maxN = maxN;
    st->size = 0;
    return st;
}

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

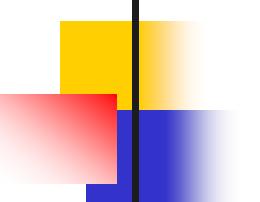
## - ST.c

```
void STfree(ST st) {
    int i;
    for (i=0; i<st->size; i++)
        ITEMfree(st->a[i]);
    if (st->a != NULL)
        free(st->a);
    free(st);
}

void STdisplay(ST st){
    int i;
    for (i = 0; i < st->size; i++)
        ITEMshow(st->a[i]);
}
```

## **ST.c**

```
void STdelete(ST st, Key k) {  
    int i, j=0;  
    while (KEYcompare(KEYget(st->a[j]), k) !=0)  
        j++;  
    for (i = j; i < st->size-1; i++)  
        st->a[i] = st->a[i+1];  
    st->size--;  
}
```



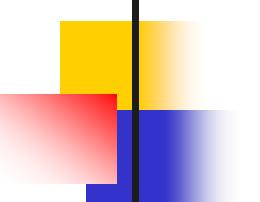
## Inserzione e ricerca in vettore non ordinato

**ST.c**

```
void STinsert(ST st, Item val) {
    int i = st->size;
    if (st->size >= st->maxN) {
        st->a=realloc(st->a, (2*st->maxN)*sizeof(Item));
        if (st->a == NULL)
            return -1;
        st->maxN = 2*st->maxN;
    }
    st->a[i] = val;
    st->size++;
}
return;
```

## **ST.c**

```
Item STsearch(ST st, Key k) {
    int i;
    if (st->size == 0)
        return ITEMsetvoid();
    for (i = 0; i < st->size; i++)
        if (KEYcompare(k, KEYget(st->a[i]))==0)
            return st->a[i];
    return ITEMsetvoid();
}
```



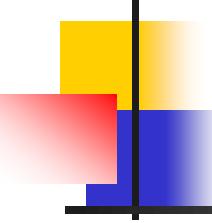
## Inserzione e ricerca in vettore ordinato

**ST.c**

```
void STinsert(ST st, Item val) {
    int i = st->size++;
    if (st->size > st->maxN) {
        st->a=realloc(st->a, (2*st->maxN)*sizeof(Item));
        if (st->a == NULL) return -1;
        st->maxN = 2*st->maxN;
    }
    while( (i>0) && ITEMless(val, st->a[i-1]) ) {
        st->a[i] = st->a[i-1];
        i--;
    }
    st->a[i] = val;
    return;
}
```

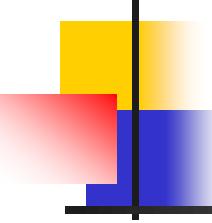
## ST.c

```
Item searchR(ST st, int l, int r, Key k) {
    int m = (l + r)/2;
    if (l > r) return ITEMsetvoid();
    if (KEYcompare(k, KEYget(st->a[m]))==0)
        return st->a[m];
    if (l == r) return ITEMsetvoid();
    if (KEYcompare(k, KEYget(st->a[m]))==-1)
        return searchR(st, l, m-1, k);
    else
        return searchR(st, m+1, r, k);
}
Item STsearch(ST st, Key k) {
    return searchR(st, 0, st->size-1, k) ;
}
Item STselect(ST st, int r) {
    return st->a[r];
}
```



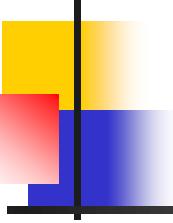
## Vantaggi/svantaggi (vettore non ordinato)

- Complessità dell'operazione di inizializzazione e inserimento:  $T(n) = \Theta(1)$
  - Complessità delle operazioni di ricerca e cancellazione:  $T(n) = O(N)$
  - Occupazione di memoria  $S(n) = \Theta(\text{maxN})$   
⇒ spreco di memoria per  $|K| << \text{maxN}$
  - Molto usate in pratica per trasformare chiavi in interi a costo unitario. L'operazione inversa ha costo lineare.
- dimensione massima presunta



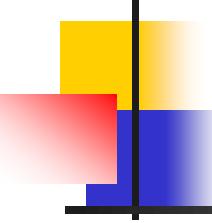
## Vantaggi/svantaggi (vettore ordinato)

- ogni inserzione ordinata ha costo lineare  $T(n) = O(N)$ , costo quadratico complessivo per n inserzioni  $T(n) = O(N^2)$
- ricerca dicotomica con costo logaritmico  
 $T(n) = O(\log N)$
- ricerca lineare con interruzione non appena possibile  $T(n) = O(N)$
- cancellazione con costo lineare  $T(n) = O(N)$
- selezione banale: rango e indice coincidono.



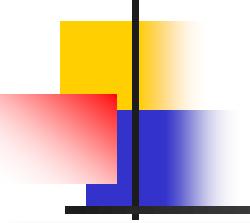
## Implementazione in funzione del contesto:

- **dinamico**: con molte inserzioni/cancellazioni: inserimento ordinato (con spostamento di una posizione degli elementi più grandi) con costo quadratico
- **statico**: con inserzioni solo in fase di lettura da file, nessuna cancellazione e molte ricerche: inserzione in fondo e ordinamento una sola volta con algoritmo  $O(N \log N)$ .



## ADT I classe tabella di simboli (lista)

- ❑ Lista con sentinella in coda
- ❑ Ricerca preliminare alla cancellazione sempre lineare
- ❑ Cancellazione iterativa e ricorsiva
- ❑ Lista non ordinata:
  - inserzione in testa per avere complessità  $O(1)$
  - la selezione non ha senso
- ❑ Lista ordinata:
  - inserzione con scansione con complessità  $O(N)$

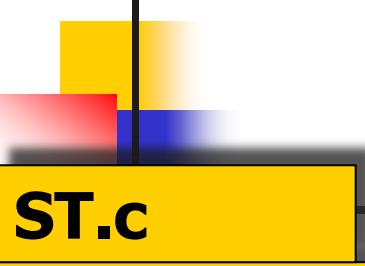


## ST.c

```
typedef struct STnode* link;
struct STnode { Item val; link next; } ;

struct symboltable {link head;int size;link z;};

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



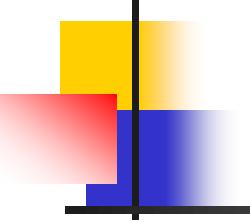
## **ST.c**

```
ST STinit(int maxN) {
    ST st;
    st = malloc(sizeof(*st));
    st->size = 0;
    st->z = NEW(ITEMsetvoid(), NULL);
    st->head = NEW(ITEMsetvoid(), st->z);
    return st;
}

void STfree(ST st) {
    if (st == NULL)
        return;
    free(st->head);
    free(st);
}
```

## ST.c

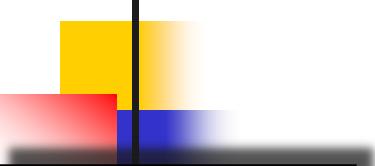
```
int STcount(ST st) {
    return st->size;
}
Item STsearch(ST st, Key k) {
    link x;
    for (x=st->head->next;x!=st->z; x=x->next)
        if (KEYcompare( KEYget(x->val), k) ==0)
            return x->val;
    return ITEMsetvoid();
}
void STdisplay(ST st) {
    link x;
    for (x=st->head->next; x!=st->z; x = x->next)
        ITEMshow(x->val);
}
```



versione iterativa

## ST.c

```
void STdelete(ST st, Key k) {  
    link x;  
    for (x=st->head;x->next!= st->z; x = x->next)  
        if (KEYcompare(KEYget(x->next->val), k) ==0)  
            x->next = x->next->next;  
    return;  
}
```

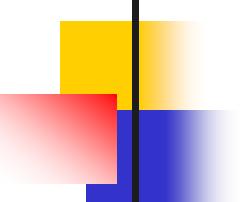


versione ricorsiva

## ST.c

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

void STdelete(ST st, Key k) {
    st->head = deleteR(st->head, k);
    st->size--;
}
```



## Inserzione in lista non ordinata

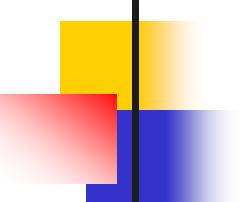
**ST.c**

```
void STinsert(ST st, Item data) {  
    st->head = NEW(data, st->head);  
    st->size++;  
}
```

## Selezione in lista ordinata

**ST.c**

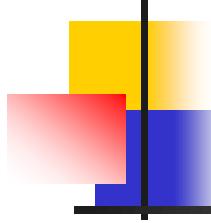
```
Item STselect(ST st, int r) {  
    int i;  
    link x = st->head;  
    for (i = r; i>=0; i--)  
        x = x->next;  
    return x->val;  
}
```



## Inserzione in lista ordinata

**ST.c**

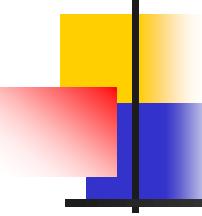
```
void STinsert(ST st, Item val) {
    link x, t;
    t = malloc(sizeof(link));
    t->val = val;
    t->next = NULL;
    for (x=st->head;x->next!=st->z;x = x->next)
        if (ITEMgreater(x->next->val, val))
            break;
    t->next = x->next;
    x->next = t;
    st->size++;
}
```



# Vantaggi/svantaggi

---

- complessità dell'operazione di inizializzazione e inserimento:  $T(n) = \Theta(1)$
- complessità delle operazioni di ricerca e cancellazione:  $T(n) = O(n)$
- complessità dell'operazione di selezione con lista ordinata:  $T(n) = O(n)$
- occupazione di memoria  $S(n) = \Theta(n)$

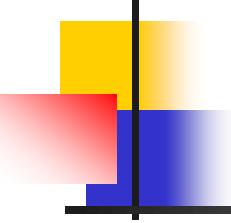


# Gestione dei duplicati nelle tabelle di simboli

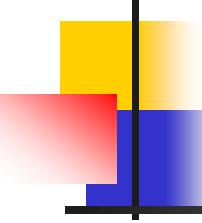
Casistica:

- le chiavi sono per sé distinte (IBAN, matricola, codice fiscale). In inserzione:
  - “**si ignora il nuovo elemento**” : si prosegue come se l’istanza (di inserimento) non sia stata avanzata = si ignora l’inserimento
  - “**si dimentica il vecchio elemento**” : si cancella (o sovrascrive) l’elemento già presente, poi si procede al nuovo inserimento

- le chiavi possono essere rese distinte (per es. si aggiunge il prefisso della nazione al numero telefonico. Si ricade nel caso precedente)
- nel modello chiavi duplicate hanno senso (per es. numero di crediti superati condiviso da più studenti). Ci si riconduce al modello precedente creando una chiave univoca e un riferimento alla lista degli elementi che la condividono

- 
- nel modello chiavi duplicate hanno senso (per es. numero di crediti superati condiviso da più studenti). Ci si riconduce al modello precedente creando una chiave univoca e un riferimento alla lista degli elementi che la condividono.  
L'inserzione è in 2 passi:
    - data la chiave, si identifica la lista ad essa associata
    - si inserisce nella lista

- nel modello hanno senso elementi che condividono la stessa chiave (per es. nome e cognome cui sono associati diversi indirizzi di e-mail). In ricerca è il client che decide cosa viene ritornato:
  - il primo elemento con quella chiave
  - un qualsiasi elemento con quella chiave
  - tutti gli elementi con quella chiave.



# Gestione dei duplicati in pile e code

Bisogna tener presente il criterio temporale:

- un elemento con chiave duplicata potrebbe essere considerato diverso in quanto inserito a un tempo diverso
- prevale la duplicazione sul tempo: bisogna decidere
  - se scartare l'elemento
  - estrarre quello già presente e inserire quello nuovo al tempo corrente
  - modificare l'elemento già presente lasciandone invariata la posizione
  - etc.