



# Capitolo 6: I Tipi di Dato Astratto

---

PUNTATORI E STRUTTURE DATI DINAMICHE:  
ALLOCAZIONE DELLA MEMORIA E  
MODULARITÀ IN LINGUAGGIO C



# ADT per composti

---

TIPO ITEM, COMPOSTO PER VALORE O RIFERIMENTO

# Progettare la struttura dati di un programma

Scegliere una adeguata struttura dati per:

- **codificare** le informazioni del problema proposto (in input, risultati intermedi e finali)
- consentire la manipolazione delle informazioni (le **operazioni**)

Le informazioni in forma utilizzabile da un sistema di elaborazione si dicono **dati** e sono memorizzati in **strutture dati**:

- interne (memoria centrale)
- esterne (memoria di massa)
- statiche (dimensione decisa in stesura del programma e fissa nel tempo)
- dinamiche (dimensione decisa in fase di utilizzo e variabile nel tempo)

# Tipi di dato: cosa sono

Definiscono organizzazione e manipolazioni di dati in termini di:

- insieme di valori (es. numeri interi)
- collezione di operazioni sui valori (algoritmi), realizzati da funzioni

## Classificazione

- base (standard): forniti dal linguaggio
- **definiti dall'utente**, mediante definizioni di tipo e/o funzioni
- **tipi di dati astratti**: netta separazione tra definizione e implementazione

# Tipi di dato standard

Tipi scalari (numeri e caratteri):

- `int` (`signed`, `unsigned`, `long`, `short`)
- `float` (`double`, `long double`)
- `char`

Tipi strutturati (composti/aggregati)

- `array` (vettori/matrici: campi omogenei)
- `struct` (campi eterogenei)

# Tipi di dato creati dall'utente

Valori:

- **typedef** permette di introdurre un nuovo nome per un tipo (da ricondurre a un tipo base, scalare o composto/aggregato)

Operazioni

- una **funzione** permette di definire una nuova operazione su argomenti e/o dato ritornato.

# ADT: Tipi di Dato Astratto

- **Scopo**
  - livello di astrazione sui dati tale da mascherare completamente l'implementazione rispetto all'utilizzo
- **definizione**
  - tipo di dato (valori + operazioni) accessibile **SOLO** attraverso un'**interfaccia**.
  - utilizzatore = **client**
  - specifica del tipo di dato = **implementazione**

# Creazione di ADT

- ❑ Il C non ha un meccanismo semplice ed automatico di creazione di ADT
- ❑ L'ADT è realizzato come modulo con una coppia di file interfaccia/implementazione
- ❑ Enfasi su come nascondere i dettagli dell'implementazione al client.

# Quasi ADT

- **Interfaccia**
  - definizione del nuovo tipo con `typedef`
  - raramente si appoggia su tipi base, in generale è un tipo composto, aggregato o contenitore (`struct` wrapper)
  - Prototipi delle funzioni
- **Implementazione**
  - Il client include il file header, quindi vede i dettagli interni del dato e/o del wrapper

# Esempio: ADT per numeri complessi

- nuovo tipo **Complex**
  - struct con campi per parte reale e coefficiente dell'immaginario
  - funzione di prodotto tra 2 numeri complessi.

Il client che include **complex.h** vede i dettagli della **struct** (ma non li usa)

## complex.h

```
typedef struct {  
    float Re; float Im;  
} Complex;
```

```
Complex prod(Complex c1, Complex c2);
```

controllo di inclusione  
multipla d'ora in poi  
omesso

## main.c

```
#include "complex.h"
```

```
int main (void) {  
    Complex a, b, c;  
    ...  
    c = prod(a,b);  
    ...  
}
```

## complex.c

```
#include "complex.h"  
  
Complex prod(Complex c1, Complex c2) {  
    Complex c;  
    c.Re = c1.Re * c2.Re - c1.Im * c2.Im;  
    c.Im = c1.Re * c2.Im + c2.Re * c1.Im;  
  
    return c;  
}
```

# ADT di I classe

Per impedire al client di vedere i dettagli della struct:

- il tipo di dato viene dichiarato nel file `.h` di interfaccia come *struttura incompleta*, o come puntatore a `struct` incompleta, non viene quindi definita la `struct` composta, aggregato o wrapper
- la `struct` viene invece completamente definita nel file `.c`
- il client utilizza unicamente puntatori alla struttura incompleta
- Il puntatore è opaco e si dice **handle**.

# ADT per numeri complessi

- Interfaccia ad ADT per numeri complessi : `complex.h`:
  - nuovo tipo `Complex` come **puntatore** a `struct complex_s`
  - prototipi della funzione di prodotto tra 2 numeri complessi e delle funzioni di creazione e distruzione
- Implementazione: `complex.c`:
  - definizione completa del tipo `struct complex_s`
  - della funzione di prodotto tra 2 numeri complessi e delle funzioni di creazione e distruzione
- Il client che include `complex.h` NON vede i dettagli della `struct`.

## complex.h

```
typedef struct complex_s *Complex;  
  
Complex crea(void);  
void distruggi(Complex c);  
Complex prod(Complex c1, Complex c2);
```

## complex.c

```
#include "complex.h"  
  
struct complex_s { float Re; float Im; };  
  
Complex crea(void) {  
    Complex c = malloc(sizeof *c);  
    return c; }  
void distruggi(Complex c) {  
    free(c); }  
Complex prod(Complex c1, Complex c2) {  
    Complex c = crea();  
    c->Re = c1->Re * c2->Re - c1->Im * c2->Im;  
    c->Im = c1->Re * c2->Im + c2->Re * c1->Im;  
    return c;  
}
```

## main.c

```
#include "complex.h"  
  
int main (void) {  
    Complex a, b, c;  
    a = crea();  
    b = crea();  
    ...  
    c = prod(a,b);  
    ...  
    distruggi(a);  
    distruggi(b);  
    distruggi(c);  
}
```

# Quasi ADT o ADT di I classe?

Per i casi semplici di dati composti o aggregati, che non prevedano  
allocazione dinamica, il quasi ADT:

- è un ragionevole compromesso
  - non nasconde completamente i dettagli interni
  - **non richiede allocazione dinamica**
- costituisce una soluzione più semplice e pratica.

# L'ADT Item

Tipo di dato generico per dato unico o record che include un campo chiave (composto o aggregato: dipende dai casi).

Esempi:

- numero
- stringa
- dati su una persona
- numero complesso
- punto di piano o spazio
- ...

# Vantaggi dell'ADT

- enfasi sull'algoritmo e non sui dati
- prelude al polimorfismo
- Tuttavia NON si tratta di un tipo generico, ma di una soluzione che concentra eventuali modifiche all'interno dell'ADT

Sono accettabili soluzioni

- quasi ADT
  - con tipo visibile al client (che può tuttavia ignorare la visibilità), non necessariamente dinamico
- ADT di 1 classe
  - dato nascosto, ma allocazione dinamica e puntatori

# Tipo 1

- Semplice scalare, chiave coincidente

```
typedef int Item;  
typedef int Key;
```

- Nessun problema di proprietà, in quanto non c'è allocazione dinamica

## Tipo 2

- Vettore dinamico di caratteri chiave coincidente

```
typedef char *Item;  
typedef char *Key;
```

- Item e chiave sono puntatori a carattere
- La chiave punta al dato

## Tipo 3: composto per valore

- **struct** con vettore di caratteri sovradimensionato staticamente e intero (composizione per valore). La chiave è il vettore di caratteri.

```
typedef struct item {  
    char name[MAXC];  
    int num;  
} Item;  
typedef char *key;
```

- Essendo la stringa statica, è interna alla struct
- La chiave punta al (a parte del) dato

## Tipo 4: composto per riferimento

- **struct** con vettore di caratteri allocato dinamicamente e intero.  
La stringa dinamica è proprietà dell'ADT (composizione per riferimento). La chiave è il vettore di caratteri.

```
typedef struct item {  
    char *name;  
    int num;  
} Item;  
typedef char *key;
```

- Essendo la stringa dinamica, è esterna alla **struct**
- La chiave punta al (a parte del) dato

# Scelte

- quando (la chiave) è un puntatore, la chiave punta al dato o a parte del dato (non si genera un duplicato).
  - Non è l'unica scelta possibile: ci sono programmi e/o funzioni in cui la chiave è esterna al dato (un'aggiunta)
- funzioni di interfaccia indipendenti dallo specifico `Item` per quanto possibile

# Versione quasi ADT: definizione di Item e Key

**item.h**

```
1  typedef int Item;
2  typedef int Key;
...

```

**item.h**

```
3  typedef struct item {
    char name[MAXC];
    int num;
} Item;
typedef char *Key;
...

```

**item.h**

```
2  typedef char *Item;
3  typedef char *Key;
...

```

**item.h**

```
4  typedef struct item {
    char *name;
    int num;
} Item;
typedef char *Key;
...

```

# Funzioni di interfaccia indipendenti da Item

**item.h**

```
/* definizione di Item e Key */

int KEYcompare(Key k1, Key k2);
Key KEYscan();

Item ITEMscan();
void ITEMshow(Item val);
int ITEMless(Item val1, Item val2);
int ITEMgreater(Item val1, Item val2);
int ITEMcheckvoid(Item val);
Item ITEMsetvoid();
```

# Funzione di interfaccia dipendente da Item

**item.h**

```
/* caso 1 e 2 */
```

```
Key KEYget(Item val);
```

struct passata per valore

# Funzione di interfaccia dipendente da Item:

**item.h**

```
/* caso 3 e 4 */
```

```
Key KEYget(Item *pval);
```

la chiave è un riferimento a un campo di Item.  
Se la struct fosse passata per valore, il  
puntatore sarebbe un riferimento alla copia locale  
della stringa, deallocato all'uscita della funzione

## Implementazione: (1) Semplice scalare, chiave coincidente

**item.c**

```
Key KEYget(Item val) {
    return (val);
}
int KEYcompare (Key k1, Key k2);
    return (k1-k2);
}
Item ITEMscan() {
    Item val;
    scanf("%d", &val);
    return val;
}
void ITEMshow(Item v) {
    printf("%d", val);
}
int ITEMless(Item val1, Item val2) {
    return (KEYget(val1)<KEYget(val2));
}
```

**direttive #include**  
d'ora in poi omesse

## Implementazione:

### (2) Vettore dinamico di caratteri chiave coincidente

**item.c**

```
static char buf[MAXC];  
  
Key KEYget(Item val) {  
    return (val);  
}  
int KEYcompare (Key k1, Key k2) {  
    return (strcmp(k1,k2));  
}  
Item ITEMscan() {  
    scanf("%s",buf);  
    return strdup(buf);  
}  
void ITEMshow(Item val) {  
    printf("%s", val);  
}  
int ITEMless(Item val1, Item val2) {  
    return (strcmp(KEYget(val1),KEYget(val2))<0);  
}
```

vettore statico  
sovradimensionato  
per acquisire le stringhe

## Implementazione:

### (3) composizione per valore, chiave campo di struct

#### item.c

```
Key KEYget(Item *pval) {  
    return (pval->name);  
}  
int KEYcompare (key k1, key k2) {  
    return (strcmp(k1,k2));  
}  
Item ITEMscan() {  
    Item val;  
    scanf("%s %d", val.name, &(val.num));  
    return val;  
}  
void ITEMshow(Item val) {  
    printf("%s %d", val.name, val.num);  
}  
int ITEMless(Item val1, Item val2) {  
    return (strcmp(KEYget(&val1),KEYget(&val2))<0);  
}
```

struct con

- vettore di caratteri
- intero

La chiave è il vettore di caratteri

## Implementazione:

### (4) composizione per riferimento, chiave campo di struct

#### item.c

```
static char buf[MAXC];

Key KEYget(Item *pval) {
    return (pval->name);}
int KEYcompare (Key k1, Key k2) {
    return (strcmp(k1,k2));}
Item ITEMscan() {
    Item val;
    scanf("%s %d", buf, &(val.num));
    val.name = strdup(buf);
    return val;
}
void ITEMshow(Item val) {
    printf("%s %d", val.name, val.num);}
int ITEMless(Item val1, Item val2) {
    return (strcmp(KEYget(&val1),KEYget(&val2))<0);}
}
```

struct con

- vettore di caratteri dinamico
- intero

La chiave è il vettore di caratteri

# De-allocare o no?

Caso critico: il client legge 2 dati di tipo `Item`, li elabora e poi li distrugge:

```
a = ITEMscan(); b = ITEMscan();
// elabora a e b
ITEMfree(a); ITEMfree(b);
```

La de-allocazione ha senso se c'è stata allocazione, quindi solo nei casi 2 e 4, non nei casi 1 e 3. Due casi: il client

- rinuncia a de-allocare (oppure chiama funzioni di de-allocazione fittizie, che non fanno nulla)
- tratta diversamente i casi con allocazione e senza, diventando dipendente da `Item`.

# Conclusione

- accettabili le soluzioni 1 e 3
- accettabile la soluzione 2 anche se disuniforme, purché il client sia responsabile
- sconsigliata la soluzione 4:
  - coi quasi-ADT meglio non usare troppo allocazione dinamica
  - (Oppure) Se si vuole allocazione dinamica, meglio non usare i quasi-ADT

# Versione ADT di I classe

- L'ADT di prima classe ha senso per dati «complessi», quindi per le tipologie 3 e 4 di Item basati su struct
- Nelle tipologie 1 e 2 la chiave è talmente semplice che una soluzione a quasi ADT è accettabile

**item.h**

```
typedef struct item *Item;  
typedef char *Key;  
...
```

Item è un puntatore  
a struct incompleta e  
quindi invisibile

Il client conosce  
il tipo Key

# Funzioni di interfaccia indipendenti da Item

## item.h

```
Key KEYget(Item val);
Key KEYscan();
int KEYcompare(Key k1, Key k2);

Item ITEMnew();
void ITEMfree(Item val);
Item ITEMscan();
void ITEMshow(Item val);
int ITEMless(Item val1, Item val2);
int ITEMgreater(Item val1, Item val2);
int ITEMcheckvoid(Item val);
Item ITEMsetvoid();
```

# Implementazione

item.c

```
3 struct item {  
    char name[MAXC];  
    int num;  
};
```

item.c

```
4 static char buf[MAXC];  
struct item {  
    char *name;  
    int num;  
};
```

# Funzioni comuni alle tipologie 3 e 4

**item.c**

```
Key KEYget(Item val) {
    return (val->name);
}
int KEYcompare (Key k1, Key k2) {
    return (strcmp(k1, k2));
}
void ITEMshow(Item val) {
    printf("%s %d", val->name, val->num);
}
int ITEMless(Item val1, Item val2) {
    return (strcmp(KEYget(val1),KEYget(val2))<0);
}
```

# ITEMnew e ITEMfree (tipologia 3)

allocazione/deallocazione della sola struct

## Item.c

```
Item ITEMnew(void) {
    Item val=(Item)malloc(sizeof(struct item));
    if (val==NULL)
        return ITEMsetvoid();
    val->name[0] = '\0';
    val->num = 0;
    return val;
}

void ITEMfree(Item val) {
    free(val);
}
```

3

# ITEMscan (tipologia 3)

**Item.c**

```
Item ITEMscan() {
    Item val = ITEMnew();
    if (val != NULL) {
        scanf("%s %d", val->name, &val->num);
    }
    return val;
}
```

3

dato creato (e valori assegnati)  
internamente alla ITEMscan

# ITEMnew e ITEMfree (tipologia 4)

**Item.c**

```
4   Item ITEMnew(void) {
        Item val=(Item)malloc(sizeof(struct item));
        if (val==NULL)
            return ITEMsetvoid();
        val->name = NULL;
        val->num = 0;
        return val;
    }
    void ITEMfree(Item val) {
        if (val->name!=NULL) free(val->name);
        free(val);
    }
```

allocazione della struct che include stringa vuota

deallocazione stringa se non vuota

# ITEMscan (tipologia 4): versione 1

**Item.c**

```
4   Item ITEMscan() {
    Item val = ITEMnew();
    if (val != NULL) {
        scanf("%s %d", buf, &val->num);
        val->name = strdup(buf);
    }
    return val;
}
```

dato creato (e valori assegnati)  
internamente alla ITEMscan

# ITEMscan (tipologia 4): versione 2

**Item.c**

```
void ITEMscan(Item val) {  
    4   scanf("%s %d", buf, &val->num);  
    val->name = strdup(buf);  
}
```

Item ricevuto per riferimento  
e valori assegnati internamente  
alla ITEMscan

# Conclusione

- ADT di I classe con ITEMnew e ITEMscan garantisce al client completa responsabilità su allocazione/deallocazione
- Consigliabile per tipologia 4 (campo stringa dinamica).

# ADT per collezioni

---

CONTENITORI DI DATI: LISTE, INSIEMI, CODE GENERALIZZATE

# ADT per collezioni di dati

- Suggerita la soluzione ADT di I classe
- Operazioni principali
  - insert: inserisci nuovo oggetto nella collezione
  - delete: cancella un oggetto della collezione
- Altre operazioni
  - inizializzare struttura dati
  - conteggio elementi (o verifica collezione vuota)
  - distruzione struttura dati
  - copia struttura dati

# ADT di classe lista (non ordinata)

**list.h**

```
typedef struct list *LIST;  
  
void listInsHead (LIST l, Item val);  
Item listSearch(LIST l, Key k);  
void listDelKey(LIST l, Key k);
```

**list.c**

```
typedef struct node *link;  
struct node { Item val; link next; };  
struct list { link head; int N; };  
  
void LISTinsHead (LIST l, Item val) {  
    l->head = newNode(val,l->head);  
    l->N++;  
}  
//implementazione delle altre funzioni
```

# Vantaggi dell'ADT di 1 classe

L' ADT di 1 classe:

1. nasconde al client i dettagli
2. permette al client di istanziare più variabili del tipo dell'ADT

Un quasi ADT viola una delle 2 regole precedenti o entrambe.

I quasi ADT visti sinora (per composti/aggregati) violavano la prima regola.

# Quasi-ADT (non ADT) per collezioni

Per gli ADT collezioni di dati può bastare avere a disposizione un solo contenitore, facendone una variabile globale dell'implementazione.

Scompare il tipo di dato per la collezione (non c'è un'istruzione `typedef`).

Sarebbe meglio chiamarlo **non ADT**, ma si mantiene il nome storico di «quasi ADT».

Lista non ordinata come quasi ADT in violazione della regola 2

**list.h**

```
void listInsHead (Item val);  
Item listSearch(Key k);  
void listDelKey(Key k);
```

definizione di nodo e di  
puntatore a nodo

**list.c**

manca typedef  
e non c'è il parametro LIST 1

```
typedef struct node *link;  
struct node { Item val; link next; } ;  
  
static link head=NULL;  
static int N=0;  
  
void LISTinsHead (Item val) {  
    head = newNode(val,head);  
    N++;  
}  
//implementazione delle altre funzioni
```

variabili globali per  
puntatore alla  
testa e cardinalità

# ADT di 1 classe Set (insieme)

**Set.h**

```
typedef struct set *SET;

SET SETinit(int maxN);
void SETfree(SET s);
void SETfill(SET s, Item val);
int SETsearch(SET s, Key k);
SET SETunion(SET s1, SET s2);
SET SETintersection(SET s1, SET s2);
int SETsize(SET s);
int SETempty(SET s);
void SETdisplay(SET s);
```

# Implementazioni possibili

- vettore
  - non ordinato
  - ordinato
- lista
  - non ordinata
  - Ordinata
- Perche NON si adotta una soluzione stile union-find (vettore con accesso diretto e corrispondenza dato-indice)?
  - Nella union-find un elemento appartiene a UN SOLO INSIEME
  - Il caso generale è diverso. Un elemento può appartenere a più insiemi, che supportano operazioni di unione, intersezione, differenza, ecc.

# Vantaggi/svantaggi

- ❑ la dimensione della lista virtualmente può crescere all'infinito, mentre il vettore va dimensionato
- ❑ complessità della funzione **SETsearch** di appartenenza:
  - vettore ordinato  $\Rightarrow$  ricerca dicotomica  $\Rightarrow O(\log N)$
  - vettore non ordinato  $\Rightarrow$  ricerca lineare  $\Rightarrow O(N)$
  - lista (ordinata/non ordinata)  $\Rightarrow$  ricerca lineare  $\Rightarrow O(N)$
- ❑ complessità delle funzioni **SETunion** e **SETintersection**:
  - vettore/lista ordinato  $\Rightarrow O(N)$
  - vettore/lista non ordinato  $\Rightarrow O(N^2)$

# Implementazione con vettore ordinato

**Set.c**

```
struct set { Item *v; int N; };

SET SETinit(int maxN) {
    SET s = malloc(sizeof *s);
    s->v = malloc(maxN*sizeof(Item));
    s->N=0;
    return s;
}
void SETfree(SET s) {
    free(s->v);
    free(s);
}
```

dimensione massima

wrapper

ricerca dicotomica

**Set.c**

```
int SETsearch(SET s, Key k) {
    int l = 0, m, r = s->N -1;
    while (l <= r) {
        m = l + (r-l)/2;
        if (KEYeq(key(s->v[m]), k))
            return 1;
        if (KEYless(key(s->v[m]), k))
            l = m+1;
        else
            r = m-1;
    }
    return 0;
}
```

## Set.c

```
SET SETunion(SET s1, SET s2) {  
    int i=0, j=0, k=0, size1=SETsize(s1);  
    int size2=SETsize(s2);  
    SET s;  
    s = SETinit(size1+size2);  
    for(k = 0; (i < size1) || (j < size2); k++)  
        if (i >= size1) s->v[k] = s2->v[j++];  
        else if (j >= size2) s->v[k] = s1->v[i++];  
        else if (ITEMless(s1->v[i], s2->v[j]))  
            s->v[k] = s1->v[i++];  
        else if (ITEMless(s2->v[j], s1->v[i]))  
            s->v[k] = s2->v[j++];  
        else { s->v[k] = s1->v[i++]; j++; }  
    s->N = k;  
    return s;  
}
```

strategia simile alla  
Merge del MergeSort

## Set.c

```
SET SETintersection(SET s1, SET s2) {
    int i=0, j=0, k=0, size1=SETsize(s1);
    int size2=SETsize(s2), minsize;
    SET s;
    minsize = min(size1, size2);
    s = SETinit(minsize);
    while ((i < size1) && (j < size2)) {
        if (ITEMeq(s1->v[i], s2->v[j])) {
            s->v[k++] = s1->v[i++]; j++;
        }
        else if (ITEMless(s1->v[i], s2->v[j])) i++;
        else j++;
    }
    s->N = k;
    return s;
}
```

```
int min (int x, int y) {
    if (x <= y)
        return x;
    return y;
}
```

# Implementazione con lista non ordinata

**Set.c**

```
typedef struct SETnode *link;

struct set { link head; int N; };
struct setNode { Item val; link next; };

SET SETinit(int maxN) {
    SET s = malloc(sizeof *s);
    s->head = NULL;
    s->N = 0;
    return s;
}
void SETfree(SET s) {
    link x, t;
    for (x=s->head; x!=NULL; x=t) {
        t = x->next; free(x);
    }
    free(s);
}
```

wrapper

dimensione massima per uniformità ma non usata

ricerca lineare

**Set.c**

```
int SETsearch(SET s, Key k) {
    link x;
    x = s->head;
    while (x != NULL) {
        if (KEYeq(key(x->val), k))
            return 1;
        x = x->next;
    }
    return 0;
}
```

## Set.c

```
SET SETunion(SET s1, SET s2) {
    link x1, x2; int founds2, counts2=0;
    SET s = SETinit(s1->N + s2->N);
    x1 = s1->head;
    while (x1 != NULL) {
        SETfill(s, x1->val); x1 = x1->next;}
    for (x2 = s2->head; x2 != NULL; x2 = x2->next) {
        x1 = s1->head;
        founds2 = 0;
        while (x1 != NULL) {
            if (ITEMeq(x1->val, x2->val)) founds2 = 1;
            x1 = x1->next;
        }
        if (founds2 == 0) {
            SETfill(s, x2->val); counts2++; }
    }
    s->N = s1->N + counts2;
    return s;
}
```

inserimento in testa  
a lista non ordinata

inserimento in testa  
a lista non ordinata

## Set.c

```
SET SETintersection(SET s1, SET s2) {
    link x1, x2; int counts=0; SET s;
    s = SETinit(s1->N + s2->N);
    x1 = s1->head;
    while (x1 != NULL) {
        x2 = s2->head;
        while (x2 != NULL) {
            if (ITEMeq(x1->val, x2->val)) {
                SETfill(s, x1->val); counts++; break;}
            x2 = x2->next;
        }
        x1 = x1->next;
    }
    s->N = counts;
    return s;
}
```

inserimento in testa  
a lista non ordinata

# Code Generalizzate

---

CODA E STACK, CODA PRIORITARIA, TABELLE DI SIMBOLI

# Le code generalizzate

**Code generalizzate**: collezioni di oggetti (dati) di tipo **Item** con operazioni principali:

- **Insert**: inserisci un nuovo oggetto nella collezione
- **Search**: ricerca se un oggetto è nella collezione
- **Delete**: cancella un oggetto della collezione

Altre operazioni:

- inizializzare la coda generalizzata
- conteggio oggetti (o verifica collezione vuota)
- distruzione della coda generalizzata
- copia della coda generalizzata

# Criteri per operazione di Delete (extract)

- **cronologico:**
  - estrazione dell'elemento inserito più recentemente
    - politica LIFO: Last-In First-Out
    - **stack o pila**
    - inserzione (push) ed estrazione (pop) dalla testa
  - estrazione dell'elemento inserito meno recentemente
    - politica FIFO: First-In First-Out
    - **queue o coda**
    - inserzione (enqueue o put) in coda (tail) ed estrazione (dequeue o get) dalla testa (head)
- **priorità:**
  - l'inserzione garantisce che, estraendo dalla testa, si ottenga il dato a priorità massima (o minima)
  - **coda a priorità**

# Criteri per operazione di Delete (extract)

- **caso:**
  - estraendo si ottiene un dato a caso
  - **coda casuale**
- **contenuto:**
  - l'estrazione ritorna un contenuto secondo determinati criteri
  - **tabella di simboli**

# Pieno/vuoto

Controlli pieno/vuoto per evitare di inserire in coda piena o estrarre da coda vuota.

Scelta tra 2 strategie:

1. il client tiene conto del numero di dati nella coda oppure l'ADT fornisce funzioni di interfaccia per il controllo pieno/vuoto
2. l'ADT controlla la correttezza delle operazioni, indicando il successo/fallimento.

Nel seguito si adotta la prima strategia (la più semplice per l'implementazione).

# L'ADT pila (stack)

Definizione: ADT che supporta operazioni di

- STACKpush: inserimento in cima
- STACKpop: preleva (e cancella) dalla cima l'oggetto inserito più di recente

Terminologia: la strategia di gestione dei dati è detta LIFO (Last In First Out)

# Possibili versioni dell'ADT stack

- con vettore
  - quasi ADT
  - ADT di I classe
- con lista
  - quasi ADT
  - ADT di I classe

# Vettore vs. lista: vantaggi/svantaggi

Spazio:

- vettore: spazio allocato sempre pari al massimo previsto, vantaggioso per stack quasi pieni
- lista: spazio utilizzato proporzionale al numero di elementi correnti, vantaggioso per stack che cambiano rapidamente dimensione

Tempo:

- push e pop  $\text{T}(n) = \mathbf{O}(1)$

# Quasi ADT vs. ADT I classe

## Quasi ADT

- implementazione mediante variabili **globali** (dichiarate fuori da funzioni) e **invisibili** da altri file sorgenti (**static**)

## ADT di I classe

- una **struct** puntata (da handle), contenente, come campi, le variabili globali del quasi ADT.

# Implementazione con vettore

- inizializzazione dello stack (**STACKinit**): array dinamico la cui dimensione viene ricevuta (come parametro `maxN`) dal programma client
- NON viene controllato il rispetto dei casi limite (pop da stack vuoto o push in stack pieno)
- suggerimento: implementare i controlli

# Quasi ADT

stack.c

stack.h

```
void STACKinit(int maxN);
int STACKempty();
void STACKpush(Item val);
Item STACKpop();
```

```
static Item *s;
static int N;

void STACKinit(int maxN) {
    s = malloc(maxN*sizeof(Item));
    N=0;
}

int STACKempty() {
    return N == 0;
}
void STACKpush(Item val) {
    s[N++] = val;
}
Item STACKpop() {
    return s[--N];
}
```

# quasi ADT

stack.c

variabili globali:  
1 solo stack

static Item \*s;  
static int N;

```
void STACKinit(int maxN) {  
    s = malloc(maxN*sizeof(Item));  
    N=0;  
}  
  
int STACKempty() {  
    return N == 0;  
}  
void STACKpush(Item val) {  
    s[N++] = val;  
}  
Item STACKpop() {  
    return s[--N];  
}
```

stack.h

```
void STACKinit(int maxN);  
int STACKempty();  
void STACKpush(Item val);  
Item STACKpop();
```

# ADT I classe

stack.c

le variabili globali del quasi ADT sono diventate campi della struct

stack.h

```
typedef struct stack *STACK;

STACK STACKinit(int maxN);
int STACKempty(STACK s);
void STACKpush(STACK s,
               Item val);
Item STACKpop (STACK s);
```

```
struct stack { Item *s; int N; };

STACK STACKinit(int maxN) {
    STACK sp = malloc(sizeof *sp) ;
    sp->s = malloc(maxN*sizeof(Item));
    sp->N=0;
    return sp;
}
int STACKempty(STACK sp) {
    return sp->N == 0;
}
void STACKpush(STACK s, Item val) {
    sp->s[sp->N++] = val;
}
Item STACKpop(STACK s) {
    return sp->s[--(sp->N)];
}
```

# Implementazione con lista

Stack di elementi in lista concatenata:

- coda della lista: primo elemento inserito
- testa della lista: ultimo elemento inserito
- push: inserzione in testa
- pop: estrazione dalla testa

La dimensione dello stack è (virtualmente) illimitata.

- inizializzazione dello stack come lista vuota (`maxN` non viene utilizzato)
- funzione `NEW` per creare (dinamicamente) un nuovo elemento
- NON viene controllato il rispetto del caso limite (pop da stack vuoto)

# quasi ADT

stack.c

variabili globali:  
1 solo stack

```
typedef struct STACKnode* link;
struct STACKnode {Item val; link next;};
static link head;
```

```
static link NEW (Item val, link next){
    link x = (link) malloc(sizeof *x);
    x->val = val; x->next = next;
    return x;
}
void STACKinit(int maxN) { head = NULL; }
int STACKempty() {return head == NULL; }
void STACKpush(Item val) {
    head = NEW(val, head);
}
```

stack.h

```
void STACKinit(int maxN);
int STACKempty();
void STACKpush(Item val);
Item STACKpop();
```

# quasi ADT

stack.c

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

static link head;

...

Item STACKpop() {
    Item tmp;
    tmp = head->val;
    link t = head->next;
    free(head);
    head = t;
    return tmp;
}
```

stack.h

```
void STACKinit(int maxN);
int STACKempty();
void STACKpush(Item val);
Item STACKpop();
```

# ADT I classe

stack.c

le variabili globali del quasi ADT sono diventate campi della struct

stack.h

```
typedef struct stack *STACK;

STACK STACKinit(int maxN);
int STACKempty(STACK s);
void STACKpush(STACK s,
               Item val);
Item STACKpop (STACK s);
```

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

struct stack { link head; };

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

STACK STACKinit(int maxN) {
    STACK s = malloc(sizeof *s) ;
    s->head = NULL;
    return s;
}

int STACKempty(STACK s) {
    return s->head == NULL; }
```

# ADT I classe

stack.c

## stack.h

```
typedef struct stack *STACK;  
  
STACK STACKinit(int maxN);  
int STACKempty(STACK s);  
void STACKpush(STACK s,  
              Item val);  
Item STACKpop (STACK s);
```

```
typedef struct STACKnode* link;  
struct STACKnode {Item val; link next;};  
  
struct stack { link head; };  
  
...  
  
void STACKpush(STACK s, Item val) {  
    s->head = NEW(val, s->head); }  
  
Item STACKpop (STACK s) {  
    Item tmp;  
    tmp = s->head->val;  
    link t = s->head->next;  
    free(s->head);  
    s->head = t;  
    return tmp;  
}
```

# L'ADT coda (queue)

Definizione: ADT che supporta operazioni di:

- **enqueue/put**: inserisci un elemento (QUEUEput)
- **dequeue/get**: preleva (e cancella) l'elemento che è stato inserito meno recentemente (QUEUEget)

Terminologia: la strategia di gestione dei dati è detta FIFO (First In First Out).

# Possibili versioni dell'ADT queue

- con vettore
  - quasi ADT
  - ADT di I classe
- con lista
  - quasi ADT
  - ADT di I classe

## Vettore vs. lista: vantaggi/svantaggi

Spazio:

- vettore: spazio allocato sempre pari al massimo previsto, vantaggioso per code quasi piene
- lista: spazio utilizzato proporzionale al numero di elementi correnti, vantaggioso per code che cambiano rapidamente dimensione

Tempo:

- put e get **T(n) = O(1)**

# Quasi ADT vs. ADT I classe

## Quasi ADT

- implementazione mediante variabili **globali** (dichiarate fuori da funzioni) e **invisibili** da altri file sorgenti (**static**)

## ADT di I classe

- una **struct** puntata (da handle), contenente, come campi, le variabili globali del quasi ADT.

## Accesso a testa e coda

Servono 2 variabili `head` e `tail`:

- `head` permette di accedere all'elemento in testa, cioè il prossimo da estrarre
- `tail` permette di accedere:
  - implementazione a vettore: alla locazione che segue l'ultimo elemento in coda, cioè alla posizione della prossima inserzione
  - implementazione a lista: alla posizione dell'ultimo elemento in coda.

`head` e `tail` sono:

- indici nell'implementazione a vettore
- puntatori nell'implementazione a lista.

## Implementazione con vettore ( $O(n)$ )

- `put` assegna alla prima cella libera, se esiste, in fondo al vettore con complessità  $O(1)$ . L'indice `tail` contiene il numero di elementi nella coda
- `get` da posizione fissa (`head = 0`), ma comporta scalare a sinistra tutti gli elementi restanti con costo  $O(n)$

## Implementazione con vettore ( $O(1)$ ): buffer circolare

- put assegna alla prima cella libera, se esiste, in posizione indicata da indice tail ( $O(1)$ )
- get da posizione variabile (**head** assume valori tra 0 e  $N-1$ ). Le celle del vettore occupate da elementi si spostano per via di put e get (**buffer circolare**).
  - head e tail sono incrementati MODULO N ( $N-2, N-1, 0, 1, \dots$ )
  - Non è più garantito  $\text{head} \leq \text{tail}$ :
    - coda vuota: head raggiunge tail
    - coda piena: tail raggiunge head

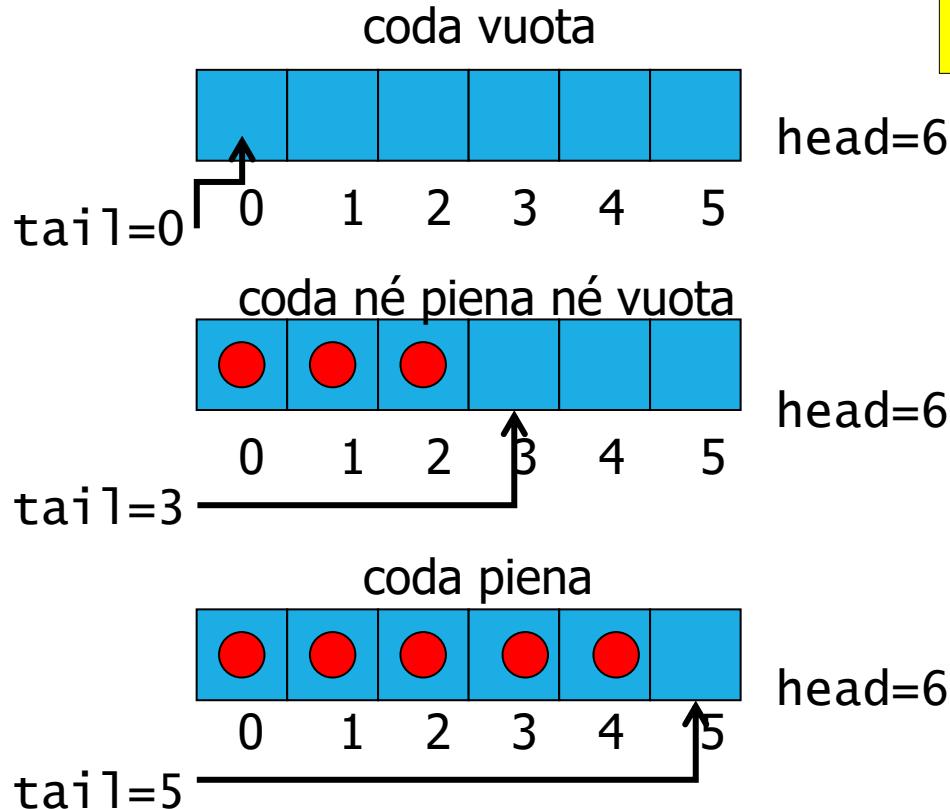
# Strategie per «riconoscere» coda piena/vuota

- A. Usare un contatore di dati (terza variabile oltre e `head` e `tail`)
- B. Evitare lo riempimento: se si conosce il numero massimo «possibile» di dati in coda (`maxN`), allocare  $N=maxN+1$  celle, di cui al massimo `maxN` usate.
  - o `tail` NON può raggiungere `head`
  - o La condizione `head==tail` può solo essere raggiunta per coda vuota.

## Implementazione (versione B)

- Dimensione del vettore  $N=maxN+1$ . Si ammettono al massimo `maxN` elementi
- inizializzazione della coda (QUEUEinit): vettore dinamico di dimensione ricevuta (come parametro `maxN`) dal programma client
  - o inizialmente `head=N`, `tail=0`
  - o successivamente `head%N == tail` indica coda vuota (`tail` non può raggiungere `head` per coda piena, è proibito!)

maxN = 5  
N = maxN+1 = 6



quasi ADT

variabili globali:  
1 sola coda

## queue.h

```
void QUEUEinit(int maxN);  
int QUEUEempty();  
void QUEUEput(Item val);  
Item QUEUEget();
```

## queue.c

```
static Item *q;  
static int N, head, tail;  
  
void QUEUEinit(int maxN) {  
    q = malloc((maxN+1)*sizeof(Item));  
    N = maxN+1;  
    head = N; tail = 0;  
}  
int QUEUEempty() {  
    return head%N == tail;  
}  
void QUEUEput(Item val) {  
    q[tail++] = val;  
    tail = tail%N;  
}  
Item QUEUEget() {  
    head = head%N;  
    return q[head++];  
}
```

## ADT I classe

le variabili globali del quasi ADT sono diventate campi della struct

### queue.h

```
typedef struct queue *QUEUE;

QUEUE QUEUEinit(int maxN);
int QUEUEempty(QUEUE q);
void QUEUEput(QUEUE q,
              Item val);
Item QUEUEget (QUEUE q);
```

### queue.c

```
struct queue {
    Item *q;
    int N, head, tail;
};

QUEUE QUEUEinit(int maxN) {
    QUEUE q = malloc(sizeof *q) ;
    q->q = malloc(maxN*sizeof(Item));
    q->N=maxN+1;
    q->head = N;
    q->tail = 0;
    return q;
}

int QUEUEempty(QUEUE q) {
    return (q->head)%(q->N) == q->tail;
}
...
```

## ADT I classe

### queue.h

```
typedef struct queue *QUEUE;

QUEUE QUEUEinit(int maxN);
int QUEUEempty(QUEUE q);
void QUEUEput(QUEUE q,
              Item val);
Item QUEUEget (QUEUE q);
```

### queue.c

```
struct queue {
    Item *q;
    int N, head, tail;
};

...

void QUEUEput(QUEUE q, Item val) {
    q->q[tail++] = val;
    q->tail = q->tail%N;
}

Item QUEUEget(QUEUE q) {
    q->head = q->head%N;
    return q->q[q->head++];
}
```

# Implementazione con lista

Coda di elementi in lista concatenata:

- testa della lista (head): primo elemento inserito
- coda della lista (tail): ultimo elemento inserito
- put: inserzione in coda
- get: estrazione dalla testa

La dimensione della coda è (virtualmente) illimitata:

- inizializzazione della coda come lista vuota (maxN non viene utilizzato, è mantenuto per uniformità con la versione basata su vettore)
- funzione NEW per creare (dinamicamente) un nuovo elemento
- put e get sono funzioni standard di inserzione in fondo ad una lista ed estrazione dalla testa della lista.

quasi ADT

## queue.h

```
void QUEUEinit(int maxN);  
int QUEUEempty();  
void QUEUEput(Item val);  
Item QUEUEget();
```

variabili globali:  
1 sola coda

## queue.c

```
typedef struct QUEUEnode *link;  
struct QUEUEnode{Item val; link next;};  
  
static link head, tail;  
  
link NEW (Item val, link next) {  
    link x = malloc(sizeof *x);  
    x->val = val;  
    x->next = next;  
    return x;  
}  
void QUEUEinit(int maxN) {  
    head = tail = NULL;  
}  
int QUEUEempty() {  
    return head == NULL;  
}  
...
```

quasi ADT

## queue.h

```
void QUEUEinit(int maxN);
int QUEUEempty();
void QUEUEput(Item val);
Item QUEUEget();
```

## queue.c

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

static link head, tail;
...
void QUEUEput(Item val) {
    if (head == NULL) {
        head = (tail = NEW(val, head));
        return;
    }
    tail->next = NEW(val, tail->next);
    tail = tail->next;
}
Item QUEUEget() {
    Item tmp = head->val;
    link t = head->next;
    free(head); head = t;
    return tmp;
}
```

## ADT I classe

le variabili globali del quasi ADT sono diventate campi della struct

### queue.h

```
typedef struct queue *QUEUE;  
  
QUEUE QUEUEinit(int maxN);  
int QUEUEempty(QUEUE q);  
void QUEUEput(QUEUE q,  
              Item val);  
Item QUEUEget (QUEUE q);
```

### queue.c

```
typedef struct QUEUEnode *link;  
struct QUEUEnode{ Item val; link next; };  
  
struct queue { link head; link tail; };  
  
link NEW(Item val, link next) {  
    link x = malloc(sizeof *x) ;  
    x->val = val; x->next = next;  
    return x;  
}  
QUEUE QUEUEinit(int maxN) {  
    QUEUE q = malloc(sizeof *q) ;  
    q->head = NULL;  
    return q;  
}  
int QUEUEempty(QUEUE q) {  
    return q->head == NULL;  
}
```

## ADT I classe

### queue.h

```
typedef struct queue *QUEUE;  
  
QUEUE QUEUEinit(int maxN);  
int QUEUEempty(QUEUE q);  
void QUEUEput(QUEUE q,  
              Item val);  
Item QUEUEget (QUEUE q);
```

### queue.c

```
...  
  
void QUEUEput (QUEUE q, Item val) {  
    if (q->head == NULL){  
        q->tail = NEW(val, q->head) ;  
        q->head = q->tail;  
        return;  
    }  
    q->tail->next = NEW(val,q->tail->next);  
    q->tail = q->tail->next;  
}  
Item QUEUEget(QUEUE q) {  
    Item tmp = q->head->tmp;  
    link t = q->head->next;  
    free(q->head); q->head = t;  
    return tmp;  
}
```

# L'ADT coda a priorità

Definizione: ADT che supporta operazioni di:

- **insert**: inserisci un elemento (PQinsert)
- **extract**: preleva (e cancella) l'elemento a priorità massima (o minima) (PQextractmax o PQextractmin).

Terminologia: la strategia di gestione dei dati è detta priority-first.

Altre operazioni:

- inizializzare la coda a priorità
- verifica se vuota
- visualizzazione senza estrazione di elemento a massima/minima priorità
- cambio della priorità di un elemento

# Possibili versioni dell'ADT coda a priorità

- con vettore/lista
  - ordinato/non ordinato
  - quasi ADT/ADT di I classe
- con heap.

Trattato più avanti

# Complessità

- implementazione con vettore/lista NON ordinato:
  - inserzione in testa alla lista o in coda al vettore ->  $O(1)$
  - estrazione/visualizzazione del massimo/minimo con scansione ->  $O(N)$
  - cambio di priorità: richiede ricerca dell'elemento con scansione ->  $O(N)$
- implementazione con vettore/lista ordinato:
  - inserzione ordinata nella lista o nel vettore mediante scansione ->  $O(N)$
  - estrazione/visualizzazione del massimo/minimo se memorizzato in testa alla lista o in coda al vettore con accesso diretto ->  $O(1)$
  - cambio di priorità:
    - lista: richiede ricerca dell'elemento mediante scansione ( $O(N)$ ), eliminazione ( $O(1)$ ), reinserimento ( $O(N)$ ), globalmente complessità  $O(N)$
    - vettore: richiede ricerca dell'elemento mediante ricerca dicotomica ( $O(\log N)$ ), eliminazione ( $O(N)$ ), reinserimento ( $O(N)$ ), globalmente complessità  $O(N)$

# Complessità

- implementazione con heap:
  - inserzione/estrazione del massimo/minimo con complessità  $O(\log N)$
  - visualizzazione del massimo/minimo con complessità  $O(1)$
  - cambio di priorità: richiede ricerca dell'elemento (con tabella di hash complessità media  $O(1)$ ), globalmente complessità  $O(\log N)$ .

# ADT I classe coda a priorità

## Implementazione con liste ordinate

**PQ.h**

```
typedef struct pqueue *PQ;

PQ PQinit(int maxN);
int PQempty(PQ pq);
void PQinsert(PQ pq, Item data);
Item PQextractMax(PQ pq);
Item PQshowMax(PQ pq);
void PQdisplay(PQ pq);
void PQchange(PQ pq, Item data);
```

## PQ.c

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

struct pqueue { link head; };

link NEW(Item val, link next) {
    link x = malloc(sizeof *x);
    x->val = val; x->next = next;
    return x;
}
PQ PQinit(int maxN) {
    PQ pq = malloc(sizeof *pq);
    pq->head = NULL;
    return pq;
}
int PQempty(PQ pq) {
    return pq->head == NULL;
}
Item PQshowMax(PQ pq) {
    return pq->head->val;
}
```

```
void PQdisplay(PQ pq) {
    link x;
    for (x=pq->head; x!=NULL; x=x->next)
        ITEMdisplay(x->val);
    return;
}
void PQinsert (PQ pq, Item val) {
    link x, p;
    Key k = KEYget(val);
    if (pq->head==NULL || KEYless(KEYget(pq->head->val),k)) {
        pq->head = NEW(val, pq->head);
        return;
    }
    for(x=pq->head->next, p=pq->head;
        x!=NULL&&KEYless(k,KEYget(x->val));
        p=x, x=x->next);
    p->next = NEW(val, x);
    return;
}
```

## PQ.c

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

struct pqueue { link head; };

...
Item PQextractMax(PQ pq) {
    Item tmp;
    link t;
    if (PQempty(pq)) {
        printf("PQ empty\n");
        return ITEMsetvoid();
    }
    tmp = pq->head->val;
    t = pq->head->next;
    free(pq->head);
    pq->head = t;
    return tmp;
}
```

```
void PQchange (PQ pq, Item val) {
    link x, p;
    if (PQempty(pq)) {
        printf("PQ empty\n");
        return;
    }
    for(x=pq->head, p=NULL; x!=NULL;
        p=x, x=x->next) {
        if (ITEMeq(x->val, val)){
            if (x==pq->head)
                pq->head = x->next;
            else
                p->next = x->next;
            free(x);
            break;
        }
    }
    PQinsert(pq, val);
    return;
}
```

# Tabelle di Simboli

---

IMPLEMENTAZIONI BASATE SU VETTORI E LISTE

# L'ADT Tabella di Simboli

Definizione: ADT che supporta operazioni di:

- **insert**: inserisci un dato (item) (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:

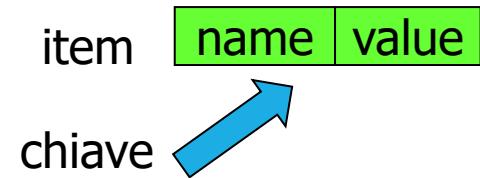
- inizializzare la tabella
- distruggere la tabella
- contare il numero di dati
- visualizzare della tabella
- se sulla chiave è definita una relazione d'ordine:
  - ordinare la tabella
  - selezionare la chiave di rango  $r$  ( $r$ -esima più piccola chiave)

# 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

# Item

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



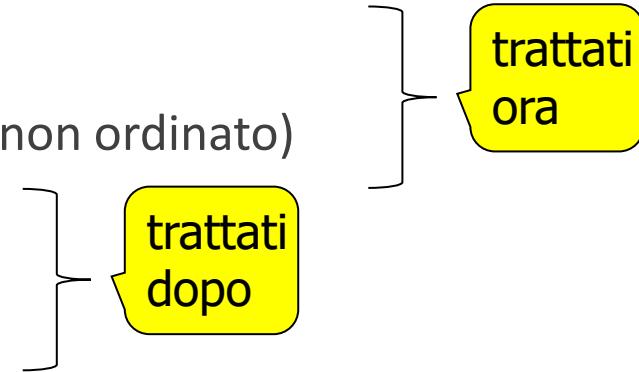
# ADT di I classe Tabella di simboli

**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

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



# Complessità

	caso peggiore		
	inserim.	ricerca	selezione
Tabelle ad accesso diretto	1	1	maxN
Array non ordinato	1	n	
Array ordinato e ricerca lineare	n	n	1
Array ordinato e ricerca binaria	n	logn	1
Lista non ordinata	1	n	
Lista ordinata	n	n	n
BST	n	n	n
RB-tree	logn	logn	logn
Hashing	1	n	

# Complessità

	caso medio		
	inserim.	ricerca	selezione
Tabelle ad accesso diretto	1	1	$\text{maxN}/2$
Array non ordinato	1	$n/2$	
Array ordinato e ricerca lineare	$n/2$	$n/2$	$n/2$
Array ordinato e ricerca binaria	$n/2$	$\log n$	$\log n$
Lista non ordinata	1	$n/2$	
Lista ordinata	$n/2$	$n/2$	$n/2$
BST	$\log n$	$\log n$	$\log n$
RB-tree	$\log n$	$\log n$	$\log n$
Hashing	1	1	

# ADT di 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  $\text{st} \rightarrow a[ ]$  di dimensione  $\text{maxN}$ :
- se la chiave  $k$  è nella tabella, essa è in posizione  $\text{st} \rightarrow a[\text{GETindex}(k)]$ , altrimenti  $\text{st} \rightarrow a[\text{GETindex}(k)]$  contiene l'elemento vuoto
- si memorizza un insieme di  $N$  chiavi ( $N \leq M$ ). La cardinalità è  $N$  ritornata dalla funzione  $\text{st} \rightarrow \text{size}$ .

# 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;  
}
```

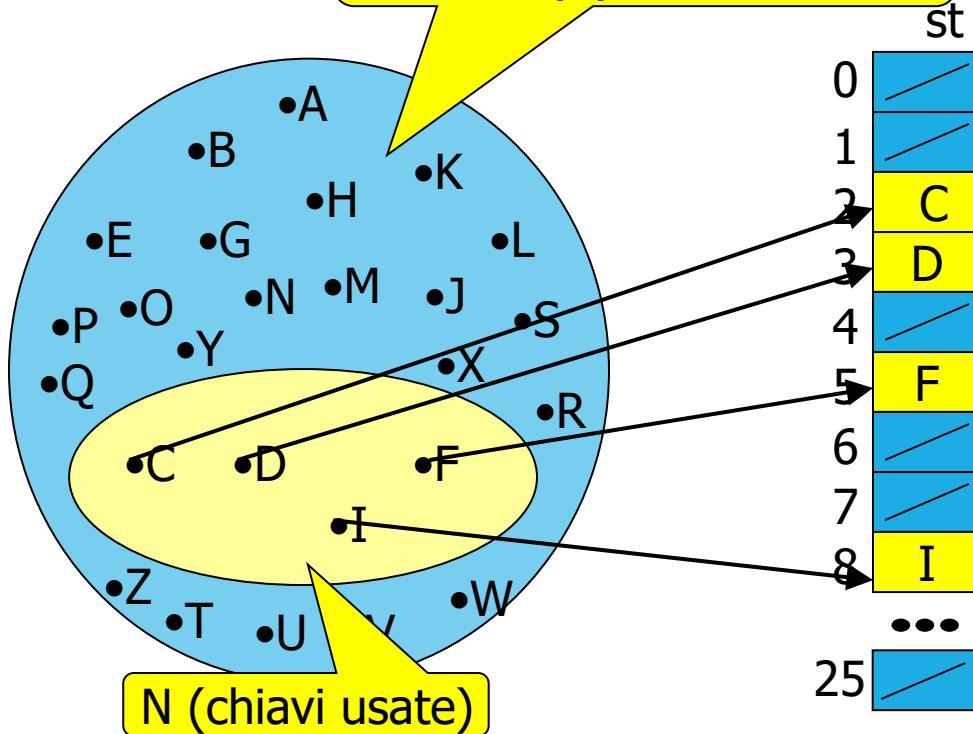
se le chiavi sono interi tra 0 e  $M-1$

```
int GETindex(Key k) {  
    int i;  
    i = (int) k;  
    return i;  
}
```

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

```
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)  
 $M = \text{card}(U) = \text{maxN} = 26$



**ST.c**

```
struct symbtab {Item *a; int N; int M;};

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;
}
```

```
void STfree(ST st) {
    free(st->a);
    free(st);
}

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

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

```
void STdelete(ST st, Key k) {
    st->a[GETindex(k)] = ITEMsetvoid();
    st->N--;
}
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 NULL;
}
void STdisplay(ST st){
    int i;
    for (i = 0; i < st->M; i++)
        if (ITEMcheckvoid(st->a[i])==0)
            ITEMstore(st->a[i]);
}
```

# Vantaggi/svantaggi

- Complessità delle operazioni di inserimento, ricerca e cancellazione:  $T(n) = \Theta(1)$
- Complessità delle operazioni di inizializzazione e selezione:  $T(n) = \Theta(\text{card}(U)) = \Theta(M)$
- 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 di 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 con complessità  $O(N)$
  - la selezione non ha senso (non è ordinato)
- Vettore ordinato:
  - inserzione con scansione con complessità  $O(N)$
  - ricerca dicotomica preliminare alla cancellazione con complessità  $O(\log N)$ .

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

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->maxN = maxN;
    st->size = 0;

    return st;
}

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

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

È responsabilità del client cancellare solo dopo aver accertato che la chiave è presente

# Inserzione e ricerca in vettore non ordinato

```
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;
        st->maxN = 2*st->maxN;
    }
    st->a[i] = val; st->size++;
}
Item STsearch(ST st, Key k) {
    int i;
    if (st->size == 0) return ITEMsetvoid();
    for (i = 0; i < st->size; i++)
        if (KEYcmp(k, KEYget(&st->a[i]))==0) return st->a[i];
    return ITEMsetvoid();
}
```

# Inserzione, selezione e ricerca in vettore ordinato

```
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;
        st->maxN = 2*st->maxN;
    }
    while((i>0)&&KEYcmp(KEYget(&val),KEYget(&st->a[i-1]))==-1){
        st->a[i] = st->a[i-1];
        i--;
    }
    st->a[i] = val;
}
Item STselect(ST st, int r) {
    return st->a[r];
}
```

```
Item STsearch(ST st, Key k) {
    return searchR(st, 0, st->size-1, k) ;
}

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

# Vantaggi/svantaggi (vettore non ordinato)

- Complessità dell'operazione di inizializzazione e inserimento:  $T(n) = \Theta(1)$
- Complessità delle operazioni di ricerca, cancellazione:  $T(n) = O(N)$
- Occupazione di memoria  $S(n) = \Theta(\max N)$   
 $\Rightarrow$  spreco di memoria per  $|K| \ll \max N$

dimensione massima  
presunta

# Vantaggi/svantaggi (vettore ordinato)

- ogni inserzione ordinata ha costo lineare  $T(n) = O(N)$ , costo quadratrico 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 di I classe Tabella di simboli (lista)

- Ricerca preliminare alla cancellazione sempre lineare  $O(N)$
- 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; } ;
typedef struct { link head; int size; } list;
struct symbtab { list tab; };
static link NEW( Item val, link next) {
    link x = malloc(sizeof(*x));
    if (x == NULL) return NULL;
    x->val = val;    x->next = next;
    return x;
}
ST STinit(int maxN) {
    ST st;
    st = malloc(sizeof(*st));
    if(st == NULL) return NULL;
    st->tab.size = 0;  st->tab.head = NULL;
    return st;
}
```

```
void STfree(ST st) {
    link x, t;
    for (x = st->tab.head; x != NULL; x = t) {
        t = x->next;
        free(x);
    }
    free(st);
}

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

void STdisplay(ST st) {
    link x;
    for (x = st->tab.head; x != NULL; x = x->next)
        ITEMstore(x->val);
}
```

```
Item STsearch(ST st, Key k) {
    link x;
    if (st == NULL)
        return ITEMsetvoid();
    if (st->tab.head == NULL)
        return ITEMsetvoid();

    for (x = st->tab.head; x != NULL; x = x->next)
        if (KEYcmp( KEYget(&x->val), k) ==0)
            return x->val;
    return ITEMsetvoid();
}
```

```
void STdelete(ST st, Key k) {  
    link x, p;  
    if (st == NULL) return;  
    if (st->tab.head == NULL) return;  
  
    for (x=st->tab.head, p=NULL; x!=NULL; p=x, x=x->next) {  
        if (KEYcmp(k, KEYget(&x->val)) == 0) {  
            if (x == st->tab.head)  
                st->tab.head = x->next;  
            else  
                p->next = x->next;  
            free(x);  
            break;  
        }  
    }  
    st->tab.size--;  
}
```

È responsabilità del client cancellare solo dopo aver accertato che la chiave è presente

## Inserzione in lista non ordinata

```
void STinsert(ST st, Item val) {  
    if (st == NULL)  
        return;  
    st->tab.head = NEW(val, st->tab.head);  
    st->tab.size++;  
}
```

## Selezione in lista ordinata

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

## Inserzione in lista ordinata

```
void STinsert(ST st, Item val) {
    link x, p;
    if (st == NULL)
        return;

    if ((st->tab.head == NULL) ||
        (KEYcmp(KEYget(&st->tab.head->val), KEYget(&val))==1))
        st->tab.head = NEW(val,st->tab.head);
    else {
        for (x = st->tab.head->next, p = st->tab.head;
             x!=NULL&& (KEYcmp(KEYget(&val), KEYget(&x->val))==1);
             p = x, x = x->next);
        p->next = NEW(val, x);
    }
    st->tab.size++;
}
```

# Vantaggi/svantaggi

- complessità dell'operazione di inizializzazione e inserimento:  $T(n) = \Theta(1)$  (inserimento in lista ordinata  $T(n) = O(n)$ )
- 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. 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.