

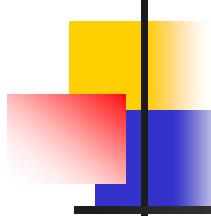
Puntatori e strutture dati dinamiche: allocazione della memoria e modularità in linguaggio C

Capitolo 6: I Tipi di Dato Astratto

G. Cabodi, P. Camurati, P. Pasini, D.
Patti, D. Vendraminetto

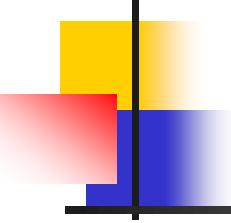


2016



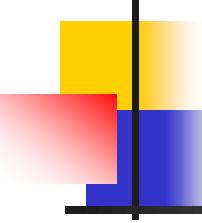
Progettare un programma

- 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 fase di stesura del programma e fissa nel tempo)
- dinamiche (dimensione decisa in fase di utilizzo e variabile nel tempo)



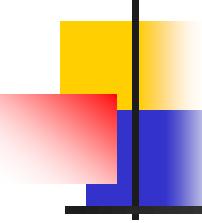
Tipi di dato

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



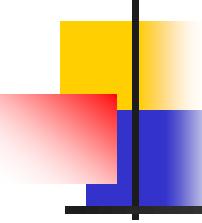
Tipi di dato standard

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)



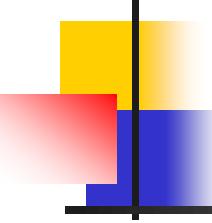
Tipi di dato creati dall'utente

Valori:

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

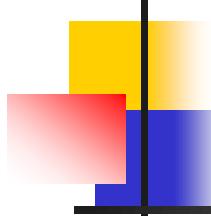
Operazioni

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



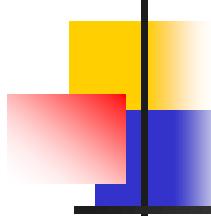
ADT: Tipi di Dato Astratto

- 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**



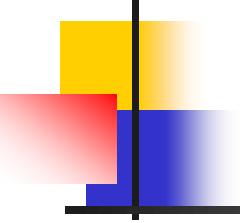
Creazione di ADT

- 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.



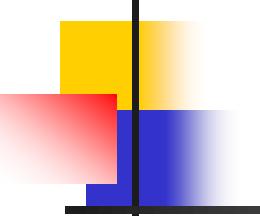
Quasi ADT

- **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`.



complex.h

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

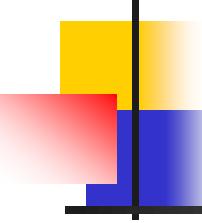
controllo di inclusione
multipla d'ora in poi
omesso

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

complex.c

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

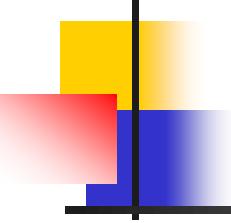
```
Complex prod(Complex c1, Complex c2) {  
    Complex c;  
    c.Re = c1.Re * c2.Re - c1.Im * c2.Im;  
    c.Im = c1.Re * c2.Im + c2.Re * c1.Im;  
  
    return c;  
}
```



ADT di I classe

Per impedire al client di vedere i dettagli della struct:

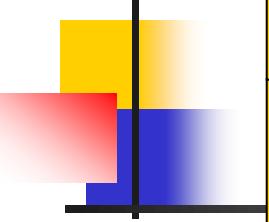
- ❑ il tipo di dato viene dichiarato nel file .h di interfaccia come *struttura incompleta*, o come puntatore a struct incompleta, non viene quindi definita la struct wrapper
- ❑ la struct viene invece completamente definita nel file .c
- ❑ il client utilizza unicamente puntatori alla struttura incompleta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



□ Esempio:

- ADT per numeri complessi : in **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
- in **complex.c**:
 - definizione completa del tipo **struct complex_s**

Il client che include **complex.h** NON vede i dettagli della **struct**.



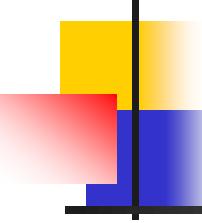
complex.h

```
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;  
}
```

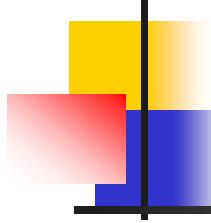
```
void distruggi(Complex c) {  
    free(c);  
}  
  
Complex prod(Complex c1, Complex c2) {  
    Complex c = crea();  
    c->Re = c1->Re * c2->Re - c1->Im * c2->Im;  
    c->Im = c1->Re * c2->Im + c2->Re * c1->Im;  
  
    return c;  
}
```



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.

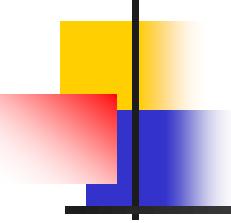


L'ADT Item

Tipo di dato generico per dato unico o record che include un campo chiave.

Esempi:

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

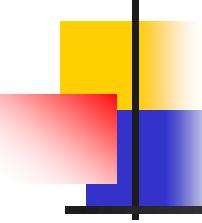


Vantaggi:

- enfasi sull'algoritmo e non sui dati
- prelude al polimorfismo

Sono accettabili soluzioni

- quasi ADT (con tipo visibile al client)
- ADT di 1 classe.

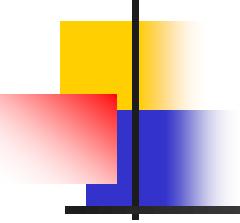


Tipologie esemplificative

1. Semplice scalare, chiave coincidente

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

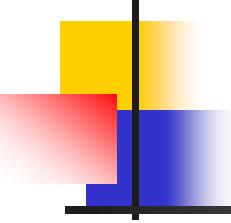
- ❑ Nessun problema di proprietà, in quanto non c'è allocazione dinamica



2. Vettore dinamico di caratteri chiave coincidente

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

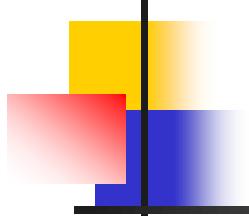
- Item e chiave sono puntatori a carattere
- Decidere se sono 2 stringhe indipendenti con lo stesso contenuto o se la chiave punta al dato



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

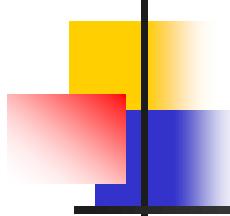
```
typedef struct item {  
    char name[MAXC];  
    int num;  
} Item;  
typedef char *Key;
```

- Essendo la stringa statica, è interna alla struct
- Decidere se sono 2 stringhe indipendenti con lo stesso contenuto o se la chiave punta al dato

- 
4. 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.

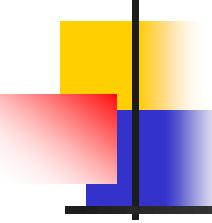
```
typedef struct item {  
    char *name;  
    int num;  
} Item;  
typedef char *key;
```

- Essendo la stringa dinamica, è esterna alla struct
- Decidere se sono 2 stringhe indipendenti con lo stesso contenuto o se la chiave punta al dato



Scelte:

- quando è un puntatore, la chiave punta al dato (non si genera un duplicato)
- funzioni di interfaccia indipendenti dallo specifico Item per quanto possibile



Versione quasi ADT

Definizione di Item e Key:

1 item.h

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

2 item.h

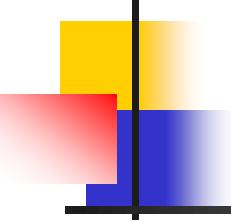
```
typedef char *Item;  
typedef char *Key;  
...
```

3 item.h

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

4 item.h

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



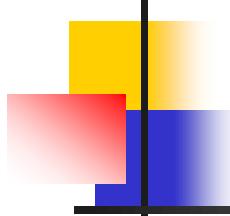
Funzioni di interfaccia indipendenti da Item:

item.h

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

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

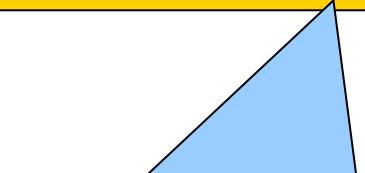
Item ITEMscan();
void ITEMshow(Item val);
int ITEMless(Item val1, Item val2);
int ITEMgreater(Item val1, Item val2);
int ITEMcheckvoid(Item val);
Item ITEMsetvoid();
```



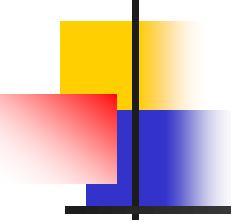
Funzione di interfaccia dipendente da Item:

item.h

```
/* caso 1 e 2 */  
Key KEYget(Item val);
```



struct passata per valore

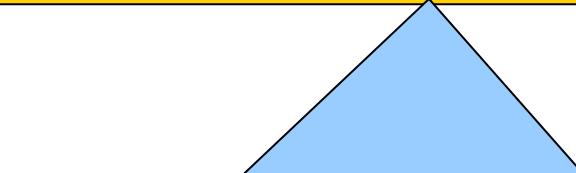


Funzione di interfaccia dipendente da Item:

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

direttive #include
d'ora in poi omesse

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;  
}
```

item.c

```
void ITEMshow(Item v) {
    printf("%d", val);
}

int ITEMless(Item val1, Item val2) {
    return (KEYget(val1)<KEYget(val2));
}
```

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);  
}
```

vettore statico
sovradimensionato
per acquisire le stringhe

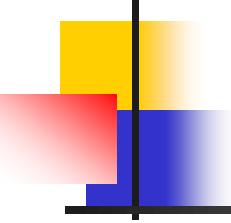
item.c

```
void ITEMshow(Item val) {  
    printf("%s", val);  
}  
  
int ITEMless(Item val1, Item val2) {  
    return (strcmp(KEYget(val1), KEYget(val2))<0);  
}
```

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

item.c

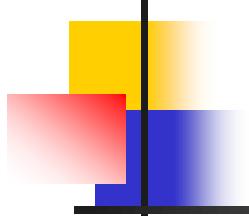
```
Key KEYget(Item *pval) {
    return (pval->name);
}
int KEYcompare (Key k1, Key k2) {
    return (strcmp(k1,k2));
}
Item ITEMscan() {
    Item val;
    scanf("%s %d", val.name, &(val.num));
    return val;
}
```



item.c

```
void ITEMshow(Item val) {
    printf("%s %d", val.name, val.num);
}

int ITEMless(Item val1, Item val2) {
    return (strcmp(KEYget(&val1), KEYget(&val2))<0);
}
```

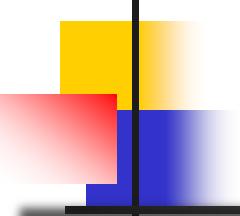
- 
4. struct con vettore di caratteri allocato dinamicamente e intero. La stringa dinamica è proprietà dell'ADT (composizione per riferimento). La chiave è il vettore di caratteri

item.c

```
static char buf[MAXC];

Key KEYget(Item *pval) {
    return (pval->name);
}

int KEYcompare (Key k1, Key k2) {
    return (strcmp(k1, k2));
```



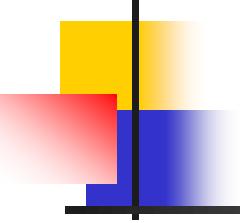
ATTENZIONE: stringa
allocata dinamicamente
in un Item allocato
staticamente

item.c

```
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);
}
```

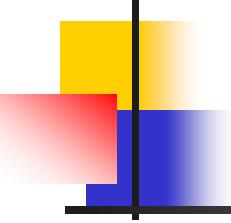


Caso critico: il client legge 2 dati di tipo `Item`, li elabora e poi li distrugge:

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

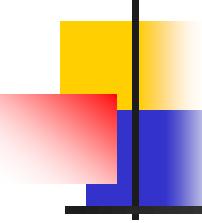
La deallocazione ha senso se c'è stata allocazione, quindi solo nei casi 2 e 4, non nei casi 1 e 3. Due casi: il client

- rinuncia a deallocare
- 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.



Versione ADT di I classe

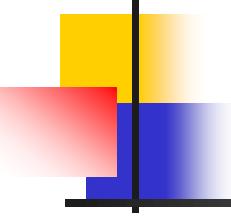
- ❑ L'ADT di prima classe ha senso per dati «complessi», quindi per le tipologie 3 e 4 di Item basati su struct
- ❑ La chiave è talmente semplice che una soluzione a quasi ADT è accettabile

item.h

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

Item è un puntatore
a struct incompleta e
quindi invisibile

Il client conosce
il tipo Key



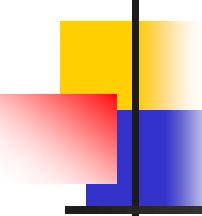
Funzioni di interfaccia indipendenti da Item:

item.h

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

Item ITEMnew();
void ITEMfree(Item val);
Item ITEMscan();
void ITEMnew(Item val);
int ITEMget(Item val1, Item val2);
int ITEMputer(Item val1, Item val2);
int ITEMdel(void(Item val));
Item ITEMalloc();
```

Free esplicita: manca l'allocatore che è implicito nella ITEMscan()



Implementazione

item.c

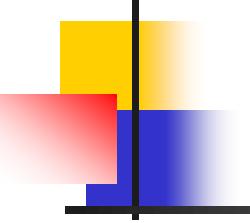
3

```
static char buf[MAXC];
typedef struct item {
    char name[MAXC];
    int num;
} Item;
```

item.c

4

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



Funzioni comuni alle tipologie 3 e 4C

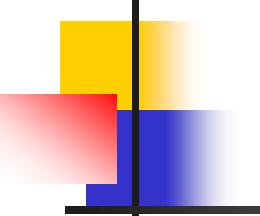
item.c

```
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);
}
```



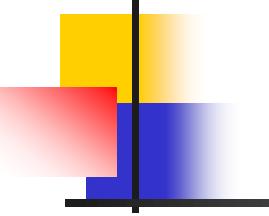
allocazione/deallocazione della sola struct

ITEMnew e ITEMfree

Item.c

```
3
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);
}
```



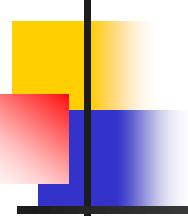
allocazione della struct che include stringa vuota

ITEMnew e ITEMfree

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);
}
```

deallocazione stringa
se non vuota

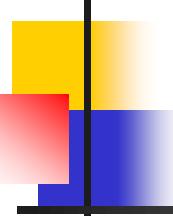


Versione 1 di ITEMscan per tipologia 4

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

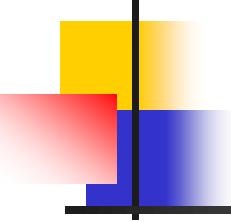


Versione 2 di ITEMscan per tipologia 4

Item.c

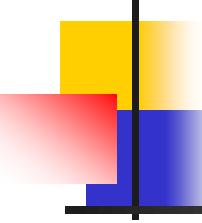
```
4 void ITEMscan(Item val) {  
    scanf("%s %d", buf, &val->num);  
    val->name = strdup(buf);  
}
```

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 di dati

- Suggerita la soluzione ad 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

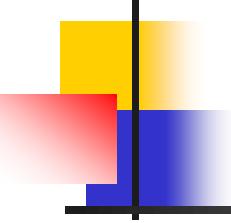
ADT di 1 classe lista (non ordinata)

list.h

```
typedef struct list *LIST;  
  
void listInsHead (LIST l, Item val);  
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);  
}  
//implementazione delle altre funzioni
```

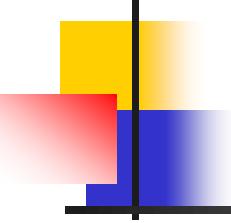


L' ADT di I classe:

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.

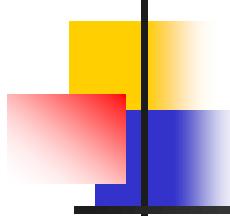
I quasi ADT visti sinora violavano la prima regola.



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`).

Quasi meglio chiamarlo **non ADT**, manteniamo il nome storico di quasi ADT.

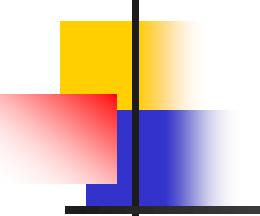


Lista non ordinata come quasi ADT in violazione
della regola 2:

list.h

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

manca typedef
e non c'è il parametro LIST 1



definizione di nodo e di
puntatore a nodo

list.c

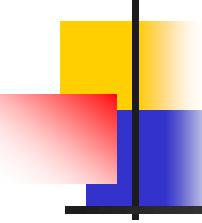
```
typedef struct node *link;
struct node { Item val; link next; } ;
```

```
static link head=NULL;
static int N=0;
```

variabili globali per
puntatore alla
testa e cardinalità

```
void LISTinsHead (Item val) {
    head = newNode(val,head);
}
```

```
//implementazione delle altre funzioni
```

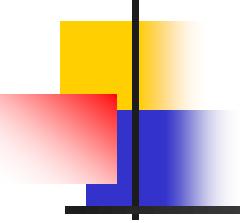


ADT di classe Set

Set.h

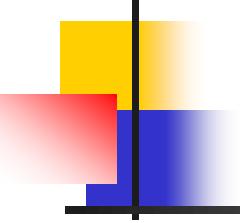
```
typedef struct set *SET;

SET SETinit(int maxN);
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:

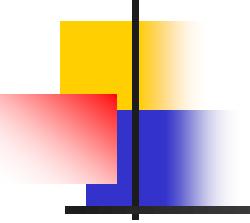
- vettore
 - non ordinato
 - ordinato
- lista
 - non ordinata
 - Ordinata



Vantaggi/svantaggi:

- la dimensione della lista virtualmente può crescere all'infinito
- 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 $\Rightarrow O(N^2)$



dimensione massima

Implementazione con vettore dimensionato:

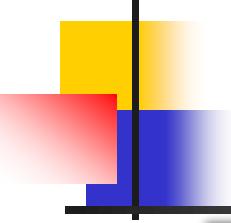
Set.c

```
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;
}
```



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;
}
```

strategia simile alla
Merge del MergeSort

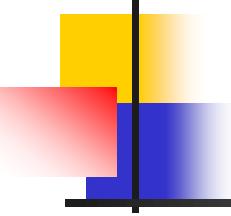
Set.c

```
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; k < size1+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;
}
```

Set.c

```
SET SETintersection(SET s1, SET s2) {
    int i=0, j=0, k=0; int 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;
}
```



dimensione massima per uniformità ma non usata

Implementazione con lista non ordinata:

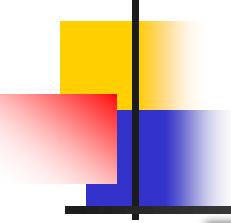
Set.c

```
typedef struct SETnode { link; } *link;

struct set { link head; int N; };

SET SETinit(int maxN) {
    SET s = malloc(sizeof *s);
    s->head = NULL;
    s->N = 0;
    return s;
}
```

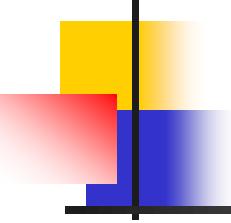
wrapper



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

inserimento in testa
a lista non ordinata

```
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;}
    x2 = s2->head;
    while (x2 != NULL) {
        x1 = s1->head;
        founds2 = 0;
```

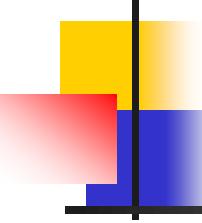
Set.c

```
    while (x1 != NULL) {
        if (ITEMeq(x1->val, x2->val))
            founds2 = 1;
        x1 = x1->next;
    }
    if (founds2 == 0) {
        SETfill(s, x2->val); counts2++; }
    x2 = x2->next;
}
s->N = s1->N + counts2;
return s;
}
```

Set.c

```
SET SETintersection(SET s1, SET s2) {
    link x1, x2; int counts=0;
    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++;
            }
            x2 = x2->next;
        }
        x1 = x1->next;
    }
    s->N = counts;
    return s;
}
```

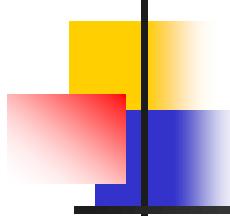
inserimento in testa
a lista non ordinata



Le code generalizzate

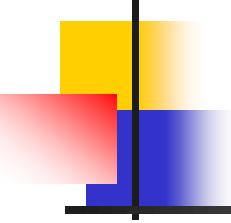
Code generalizzate: collezioni di oggetti (dati) di tipo **Item** con operazioni principali:

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



Altre operazioni:

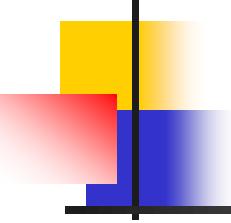
- inizializzare la coda generalizzata
- conteggio oggetti (o verifica collezione vuota)
- distruzione della coda generalizzata
- copia della coda generalizzata



Criteri per operazione di **Delete**:

- **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:

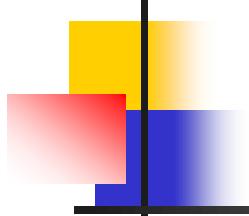
- l'inserzione garantisce che, estraendo dalla testa, si ottenga il dato a priorità massima (o minima)

coda a priorità

caso:

- estraendo si ottiene un dato a caso
- **coda casuale**

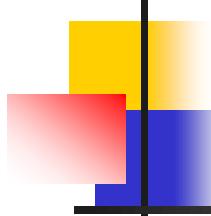
- **contenuto:**
 - l'estrazione ritorna un contenuto secondo determinati criteri
 - **tabella di simboli**



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.

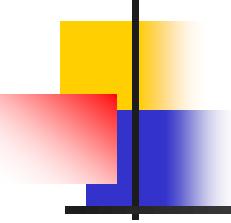


L'ADT pila (stack)

Definizione: ADT che supporta operazioni di

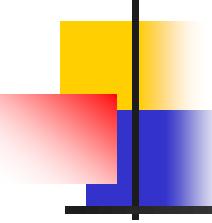
- **STACKpush:** inserimento in cima
- **STACKpop:** preleva (e cancella) dalla cima l'oggetto inserito più di recente

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



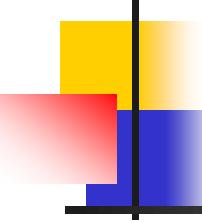
Vettore vs. lista: vantaggi/svantaggi

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 $T(n) = O(1)$



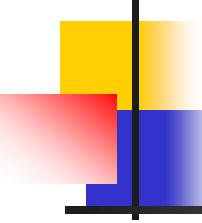
Quasi ADT vs. ADT I classe

Quasi ADT

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

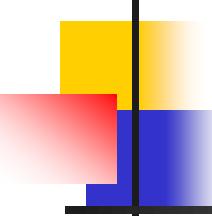
ADT di I classe

- ❑ una `struct` puntata (da handle), contenente, come campi, la 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



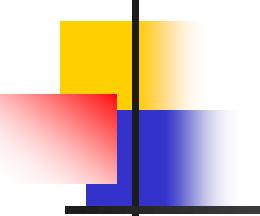
quasi ADT

stack.h

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

stack.c

```
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();
```

stack.c

```
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]; }
```

ADT I classe

le variabili globali del quasi ADT sono diventate campi della struct

stack.h

```
typedef struct stack *STACK;

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

stack.c

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

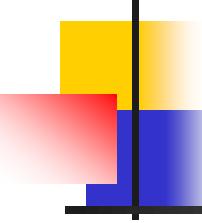
stack.c

```
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;
}

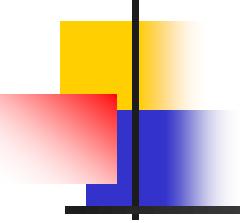
Item STACKpop(STACK s) {
    return sp->s[--(sp->N)];
}
```



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.h

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

stack.c

variabili globali:
1 solo stack

```
typedef struct STACKnode* link;

struct STACKnode { Item val; link next; };

static link head;
static int N;

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

stack.c

```
void STACKinit(int maxN) { head = NULL; }

int STACKempty() {return head == NULL; }

void STACKpush(Item val) {
    head = NEW(val, head);
}

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

ADT I classe

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);
```

stack.c

```
typedef struct STACKnode* link;
```

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

```
struct stack { link head; int N; };
```

stack.c

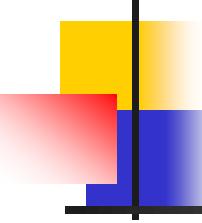
```
static link NEW (Item val, link next) {
    link x = (link) malloc(sizeof *x);
    x->val = val;
    x->next = next;
    return x;
}

STACK STACKinit(int maxN) {
    STACK s = malloc(sizeof *s) ;
    s->head = NULL;
    s->N = 0;
    return s;
}

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

stack.c

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

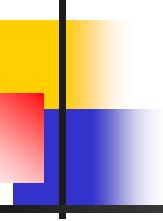


L'ADT coda (queue)

Definizione: ADT che supporta operazioni di:

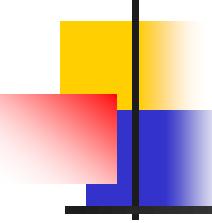
- **enqueue/put**: inserisci un elemento (QUEUEput)
- **dequeue/get**: preleva (e cancella) l'elemento che è stato inserito meno recentemente (QUEUEget)

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



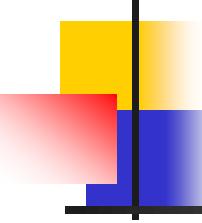
Vettore vs. lista: vantaggi/svantaggi

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 $T(n) = O(1)$



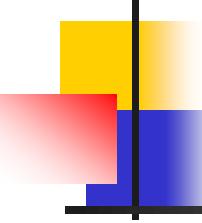
Quasi ADT vs. ADT I classe

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, la variabili globali del quasi ADT.



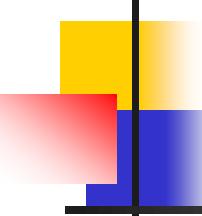
Accesso a testa e coda

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.

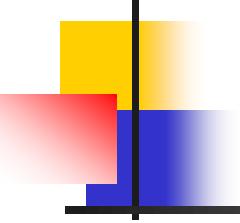
`head` e `tail` sono:

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



Implementazione con vettore

- 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 può avvenire:
 - da posizione fissa ($head = 0$), ma comporta scalare a sinistra tutti gli elementi restanti con costo $O(n)$
 - 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**). Il vettore ha $N=\maxN+1$ celle di cui al massimo \maxN usate. $head$ e $tail$ sono incrementati MODULO N .

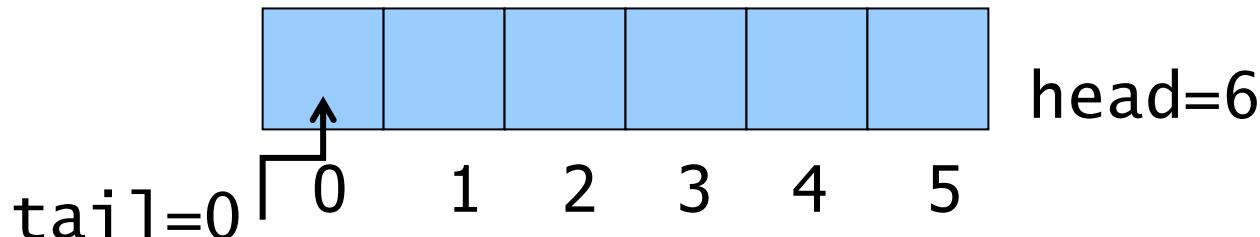


La dimensione del vettore ($N=\maxN+1$) è il limite massimo: si ammettono al massimo \maxN elementi (quindi tail NON può raggiungere head).

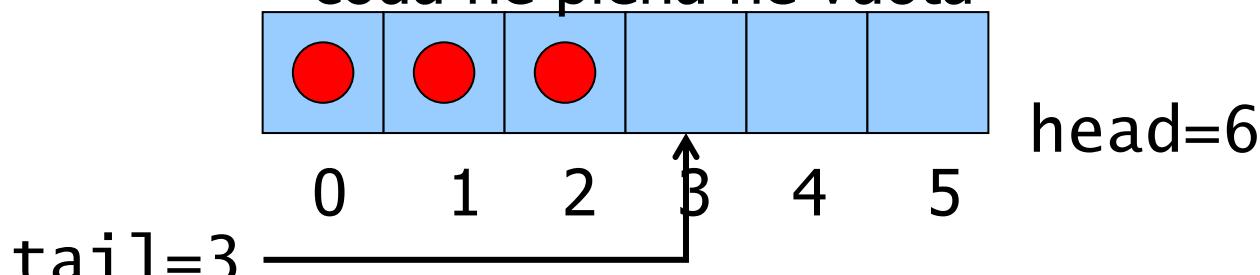
- inizializzazione della coda (QUEUEinit): vettore dinamico di dimensione ricevuta (come parametro \maxN) dal programma client
- inizialmente $\text{head}=\text{N}$, $\text{tail}=0$
- successivamente $\text{head}\%N == \text{tail}$ indica coda vuota (tail non può raggiungere head per coda piena, è proibito!)

maxN = 5
N = maxN+1 = 6

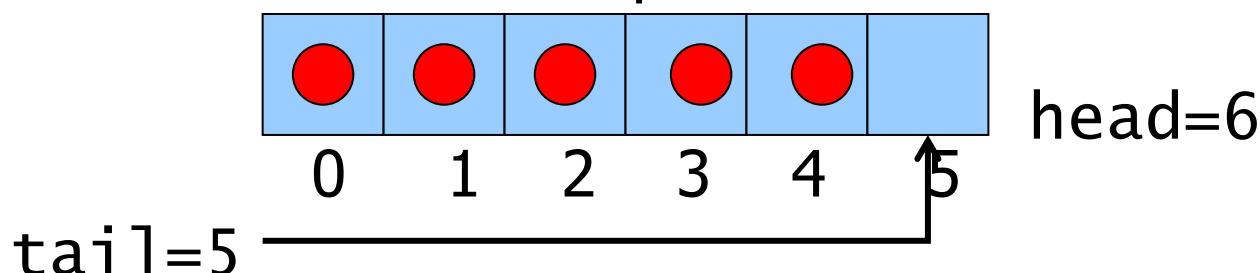
coda vuota



coda né piena né vuota



coda piena



variabili globali:
1 sola coda

quasi ADT

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;  
}
```

queue.c

```
void QUEUEput(Item val) {  
    q[tail++] = val;  
    tail = tail%N;  
}  
  
Item QUEUEget() {  
    head = head%N;  
    return q[head++];  
}
```

ADT I classi

le variabili globali del quasi ADT sono diventate campi della struct

queue.h

```
typedef struct queue *QUEUE;

QUEUE QUEUEinit(Item val, 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.c

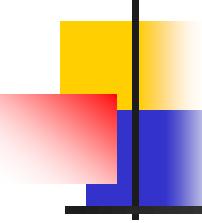
```
QUEUE QUEUEinit(int maxN) {
    QUEUE q = malloc(sizeof *q) ;
    q->q = malloc(maxN*sizeof(Item));
    q->N=maxN+1;
    q->head = N;
    q->tail = 0;
    return q;
}

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

queue.c

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

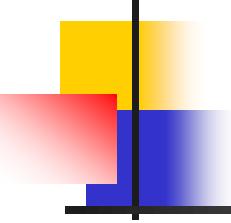
Item QUEUEget(QUEUE q) {
    q->head = q->head%N;
    return q->q[q->head++];
}
```



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.c

variabili globali:
1 sola coda

queue.h

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

```
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;
}
```

queue.c

```
void QUEUEinit(int maxN) {  
    head = tail = NULL;  
}  
  
int QUEUEempty() {  
    return head == NULL;  
}
```

queue.c

```
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; };
```

queue.c

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

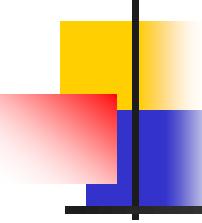
```
QUEUE QUEUEinit(int maxN) {
    QUEUE q = malloc(sizeof *q) ;
    q->head = NULL;
    return q;
```

```
int QUEUEempty(QUEUE q) {
    return q->head == NULL;
}
```

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;
}
```

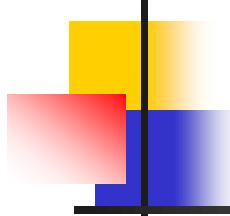


L'ADT coda a priorità

Definizione: ADT che supporta operazioni di:

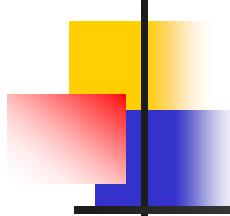
- **insert**: inserisci un elemento (PQinsert)
- **extract**: preleva (e cancella) l'elemento a priorità massima (o minima) (PQextractmax o PQextractmin).

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



Altre operazioni:

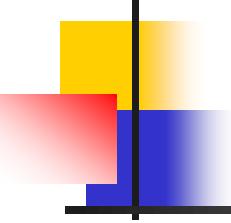
- inizializzare la coda a priorità
- verifica se vuota
- visualizzazione senza estrazione di elemento a massima/minima priorità
- cambio della priorità di un elemento



Possibili versioni dell'ADT coda a priorità:

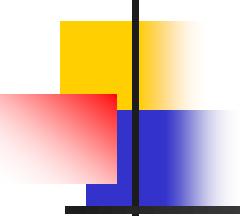
- con vettore/lista
 - ordinato/non ordinato
 - quasi ADT/ADT di I classe
- con heap.

non trattato

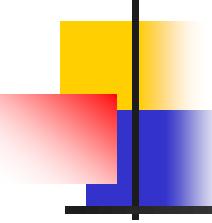


Complessità:

- implementazione con vettore/lista NON ordinato:
 - inserzione in testa alla lista o in coda al vettore, complessità $O(1)$
 - estrazione/visualizzazione del massimo/minimo mediante scansione, complessità $O(N)$
 - cambio di priorità: richiede ricerca dell'elemento mediante scansione, complessità $O(N)$

- 
- implementazione con vettore/lista ordinato:
 - inserzione ordinata nella lista o nel vettore mediante scansione, complessità $O(N)$
 - estrazione/visualizzazione del massimo/minimo se memorizzato in testa alla lista o in coda al vettore con accesso diretto, complessità $O(1)$
 - 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 ($O(N \log N)$), eliminazione ($O(N)$), reinserimento ($O(N)$), globalmente complessità $O(N)$

- implementazione con heap:
 - inserzione/estrazione del massimo/minimo con complessità $O(N \log N)$
 - 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.c

```
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;
}
```

PQ.c

```
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;
}
```

inserimento in testa

cerca posizione

PQ.c

```
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;
}
```

PQ.c

```
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 &&
         KEYgeq(KEYget(x->val), KEYget(val));
         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;
}
```