

Introduzione

Liste

DD p. 449-474

KP p.397-424

- Abbiamo fin ora studiato strutture dati con dimensione fissa:
 - vettori e matrici
 - strutture
- Introduciamo ora le strutture dati dinamiche (liste concatenate):
 - strutture dati la cui dimensione cresce o decresce durante l'esecuzione del programma, secondo le necessità

Strutture ricorsive (1)

- Una struttura ricorsiva contiene un membro di tipo puntatore, che fa riferimento ad una struttura dello stesso tipo di quella in cui è contenuto.

Esempio:

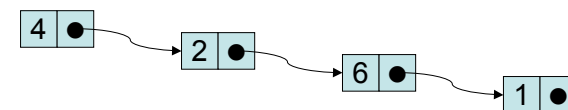
```
struct nodo  
{  
    int dato;  
    struct nodo* succ;  
};
```

succ punta ad una struttura dati dello stesso tipo di quella in cui ci troviamo

Strutture ricorsive (2)

- Le strutture ricorsive possono essere usate per formare utili organizzazioni di dati: le liste.

Esempio:



Allocazione dinamica della memoria

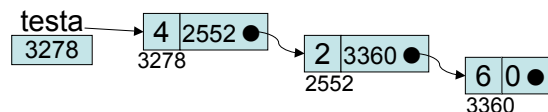
- L'allocazione dinamica della memoria è la capacità di un programma di ottenere, durante la sua esecuzione, un maggior spazio di memoria per immagazzinare nuovi dati e di poterlo rilasciare quando non è più necessario.
- Essenziali le funzioni `malloc`, `free` e `sizeof`.

Un esempio

```
struct nodoLista
```

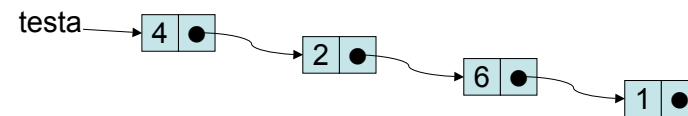
```
{  
    int dato;  
    struct nodoLista* succ;  
};  
struct nodoLista* testa;
```

- `testa` punta all'inizio della lista
- `testa->dato` vale 4
- `testa->succ` punta al secondo elemento della lista
- `testa->succ->dato` vale 2 e così via...



Liste

- Una lista concatenata è una collezione di strutture ricorsive (nodi) connesse da puntatori
- Si accede alla lista tramite un puntatore al suo primo elemento (testa della lista)
- Si accede agli elementi intermedi per mezzo dei puntatori di concatenazione
- Il puntatore dell'ultimo elemento della lista deve essere posto a NULL.



Operazioni su liste

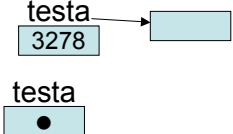
- Dettaglieremo le seguenti operazioni su liste:
 - verifica se la lista è vuota
 - stampa
 - inserimento in testa
 - cancellazione di un elemento
 - somma degli elementi
 - verifica se la lista è ordinata
 - inserimento in una lista ordinata

Verifica lista vuota

```

struct nodoLista
{
    int dato;
    struct nodoLista* succ;
};
struct nodoLista testa;

int ListaVuota(struct nodoLista* testa)
{
    return testa==NULL;
}
    
```



Stampa lista (1)

```

void StampaLista (nl* corr)
{
    if (ListaVuota(corr))
        printf("lista vuota");
    else
    {
        while(corr!=NULL)
        {
            printf("%d->", corr->dato);
            corr=corr->succ;
        }
    }
}
    
```

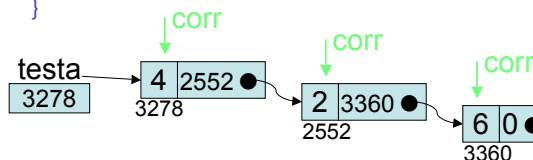
```

struct nodoLista
{
    int dato;
    struct nodoLista* succ;
};
struct nodoLista* testa;
typedef struct nodoLista nl;
    
```

Stampa lista (2)

```

void StampaLista (nl* corr)
{
    if (ListaVuota(corr))
        printf("lista vuota");
    else
    {
        while(corr!=NULL)
        {
            printf("%d->", corr->dato);
            corr=corr->succ;
        }
    }
}
    
```



all'inizio, la funzione
viene invocata come:
`StampaLista(testa);`

`corr` vale 3278 e punta
alla testa della lista

viene stampato 4-> e
spostato `corr` su 2552

viene stampato 2-> e
spostato `corr` su 3360

viene stampato 6-> e
spostato `corr` su 0

Inserimento in testa (1)

```

nl* InserisciInTesta(nl* testa, int val)
{
    nl* nuovo;
    nuovo=(nl*)malloc(sizeof(nl));
    if (nuovo!=NULL)
        nuovo->dato=val;
        nuovo->succ=testa;
        return nuovo;
}
    
```

```

struct nodoLista
{
    int dato;
    struct nodoLista* succ;
};
struct nodoLista* testa;
typedef struct nodoLista nl;
    
```

oppure

```

void InserisciInTesta(nl** ptesta,
int val)
{
    nl* nuovo;
    nuovo=(nl*)malloc(sizeof(nl));
    if (nuovo!=NULL)
        nuovo->dato=val;
        nuovo->succ=*ptesta;
        *ptesta=nuovo;
}
    
```

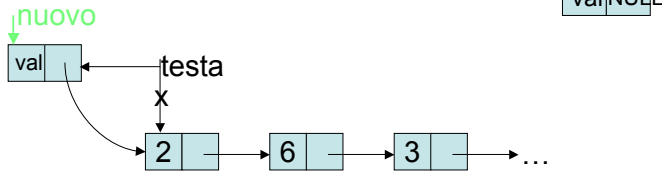
Inserimento in testa (2)

```
nl* InserisciInTesta(nl* testa, int val)
{
    nl* nuovo;
    nuovo=(nl*)malloc(sizeof(nl));
    if (nuovo!=NULL)
        nuovo->dato=val;
        nuovo->succ=testa;
        return nuovo;
}
```

all'inizio, la funzione viene invocata come:
 testa=InserisciInTesta(testa);

Due casi:
 1. lista vuota
 2. lista non vuota

testa=NULL



Cancellazione

```
void Cancella(nl** ptesta, int val)
{
    nl* prec, corr, app;
    if (val==( *ptesta)->dato)
    {
        app=*ptesta;
        *ptesta=( *ptesta)->succ;
        free(app);
    }
    else
    {
        prec=*ptesta;
        corr=( *ptesta)->succ;
        while (corr!=NULL
                && corr->dato!=val)
        {
            prec=corr;
            corr=corr->succ;
        }
        if (corr!=NULL)
        {
            app=corr;
            prec->succ=corr->succ;
            free(app);
        }
    }
}
```

```
struct nodoLista
{
    int dato;
    struct nodoLista* succ;
};
struct nodoLista* testa;
typedef struct nodoLista nl;
N.B. verifica lista vuota
fuori della funzione!
```

cosa succede in mem: alla lavagna

Somma elementi di una lista

```
int SommaLista (nl* corr)
{
    int somma=0;
    if (ListaVuota(corr))
        return 0;
    else
    {
        while(corr!=NULL)
        {
            somma+= corr->dato;
            corr=corr->succ;
        }
        return somma;
    }
}
```

```
struct nodoLista
{
    int dato;
    struct nodoLista* succ;
};
struct nodoLista* testa;
typedef struct nodoLista nl;
```

cosa succede in mem: alla lavagna

Esercizi (liste)

- Progettare ed implementare una funzione C che risolva i seguenti problemi, se la testa della lista è sempre data come parametro:
 - verificare se una lista è ordinata o no: se sì restituire 1, se no restituire 0
 - inserire un valore (dato come parametro) in una lista ordinata e restituire la lista con l'elemento in più ancora ordinata
 - calcolare il numero di occorrenze in una lista di un elemento fissato (dato come parametro)
 - inserire un valore (dato come parametro) come ultimo elemento di una lista (*inserimento in coda*)

Funzioni ricorsive per liste (1)

- Le liste sono strutture ricorsive.
- Per esse è naturale implementare funzioni ricorsive

```
StampaListaRic(testa);
```

```
void StampaListaRic(nl* corr)
{
    if (ListaVuota(corr))
        printf("NULL");
    else
    {
        printf("%d->", corr->dato);
        StampaListaRic(corr->succ);
    }
}
```

3->1->6->9->NULL

Funzioni ricorsive per liste (2)

Esempio: somma degli elementi di una lista

```
x=SommaListaRic(testa);
```

```
int SommaListaRic(nl* corr)
{
    if (!ListaVuota(corr))
        return(corr->dato+
            SommaListaRic(corr->succ));
    else return 0;
}
```

0+9+6+1+3

Funzioni ricorsive per liste (3)

Esempio: trasformazione da vettore a lista

```
nl *x;
x=DaVettALista(vett);
```

```
nl* DaVettALista(int v[])
{
    nl* testa;
    if (v[0]==0) return NULL;
    else
    {
        testa=(nl*)malloc(sizeof(nl));
        testa->dato=v[0];
        testa->succ=DaVettALista(v+1);
        return testa;
    }
}
```

testa → 3 → 1 → 6 → 9 → NULL

Funzioni ricorsive per liste (4)

Esempio: contare il numero di elementi di cui è costituita una lista

```
int ContaRic(nl* testa)
{
    if (ListaVuota(testa))
        return 0;
    else
        return(1 + ContaRic(testa->succ));
}
```

```
int ContaIter(nl* testa)
{
    int cont=0;
    for(; testa!=NULL; testa=testa->succ)
        ++cont;
    return cont;
}
```

N.B. testa è passato per valore, quindi la funzione non lo modifica

cosa succede in mem: alla lavagna

Esercizi (ricorsione su liste)

- Progettare ed implementare una funzione iterativa ed una ricorsiva C che risolva i seguenti problemi, se la testa della lista è sempre data come parametro:
 - restituire la lista invertita (*inversione*) modificando la lista originaria, senza cioè costruirne una nuova
 - restituire la lista in cui siano stati eliminati gli elementi di valore pari
 - restituire la lista in cui siano stati eliminati gli elementi di posizione pari
 - restituire la lista in cui ogni elemento sia stato sostituito dalla somma degli elementi che lo precedono

Liste vs Vettori (2)

- Il contenuto di una lista concatenata potrebbe essere immagazzinato in un vettore, e viceversa.
- Le liste hanno alcuni vantaggi:
 - quando non è possibile determinare a priori la dimensione, la si può allocare dinamicamente, mentre la dimensione di un vettore è fissa
 - le liste non possono mai riempirsi, come i vettori, causando overflow
 - certi inserimenti e cancellazioni sono meno costosi, mentre i vettori presuppongono una operazione di scorrimento (verso destra per l'inserimento e verso sinistra per la cancellazione)
- ... e svantaggi:
 - gli elementi dei vettori sono ad accesso diretto mentre quelli delle liste ad accesso sequenziale
 - le liste occupano più spazio, dovuto ai puntatori (salvo sovradimensionamento del vettore)

Liste vs Vettori (1)

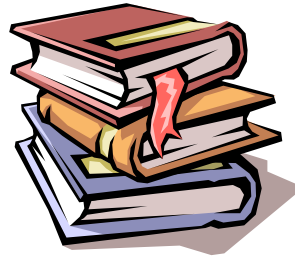
- Inserimento in testa:
 - L: operazione costante
 - V: operazione lineare
- Inserimento in coda:
 - L: operazione lineare (ricerca coda, se non c'è un puntatore alla coda)+ operazione costante
 - V: operazione costante
- Ricerca i-esimo elemento:
 - L: operazione lineare
 - V: operazione costante
- Inserimento in posizione i:
 - L: operazione lineare (ricerca)+operazione costante
 - V: operazione costante (ricerca)+operazione lineare
- Cancellazione dell'elemento i:
 - L: operazione lineare (ricerca)+ operazione costante
 - V: operazione costante (ricerca)+ operazione lineare

Pile e Code

- Pile e code sono strutture dati particolari, su cui è possibile fare solo un certo tipo di operazioni.
- Esse possono essere implementate tanto con i vettori che con le liste.
- Dunque il punto non è programmare le funzioni che le fanno funzionare, ma capire cosa ci si può fare...

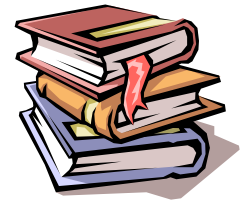
Pile (1)

- Una pila (o stack) è una struttura dati che memorizza delle informazioni.
- Si possono aggiungere elementi solo in testa alla pila.
- Si possono eliminare elementi solo dalla testa della pila.
- La pila è anche detta struttura LIFO (last in first out)



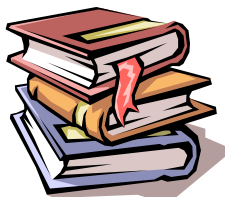
Pile (2)

- Le principali funzioni usate per gestire una pila sono push e pop, più la funzione che verifica se la pila è vuota.
- La loro implementazione dipende dal fatto che usiamo un vettore o una lista per gestire la pila.



Pile (3)

- **Implementazione con le liste**
 - Inserimento (push):
 - Crea un nuovo nodo con malloc
 - Inserisci il nuovo nodo in testa
 - Cancellazione (pop):
 - Salva il valore del primo nodo
 - Cancella il nodo di testa
 - Rilascia la memoria
 - Verifica di pila vuota:
 - Guarda se la testa della pila punta a NULL



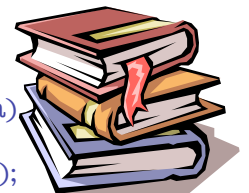
Pile (4)

```
int pop(nl** ptesta)
{
    nl* paux; int x;
    if (!pilavuota(*ptesta))
    {
        x=*ptesta->dato;
        paux=*ptesta;
        *ptesta=*ptesta->succ;
        free(paux);
        return x;
    }
    else printf("underflow");
}

void push(nl** ptesta, int x)
{
    nl* paux;
    paux=malloc(sizeof(nl));
    paux->dato=x;
    paux->succ=*ptesta;
    *ptesta=paux;
}

int pilavuota(nl* testa)
{
    return (testa==NULL);
}
```

cosa succede in mem: alla lavagna

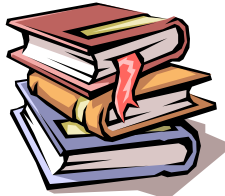


Pile (5)

- **Implementazione con i vettori**

N.B. inserire e cancellare in testa è oneroso, ma posso invertire la struttura e usare la coda!

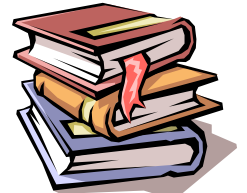
- Inserimento (push):
 - Se c'è spazio sufficiente nel vettore inserisci il nuovo elemento in coda
 - Aggiorna la lunghezza del vettore
- Cancellazione (pop):
 - Salva il valore dell'ultimo elemento
 - Aggiorna la lunghezza del vettore
- Verifica di pila vuota:
 - Guarda se la lunghezza è 0



Pile (6)

```
int pop(int pila[], int* pn)    void push(int pila[], int* pn, int x)
{
    int x;
    if (!pilavuota(*pn))
    {
        x=pila[(*pn)-1];
        (*pn)--1;
        return x;
    }
    else
        printf("underflow");
}

int pilavuota(int n)
{
    return (n==0);
}
```



cosa succede in mem: alla lavagna

Pile (7)

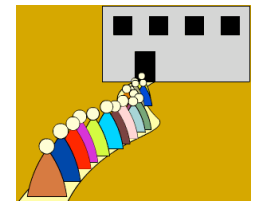
- Le pile hanno molte interessanti applicazioni:

- Ogni volta che si richiama una funzione, questa deve sapere come restituire il controllo alla funzione chiamante, e l'indirizzo di ritorno è memorizzato in una pila (es. supporto alle chiamate di funzioni ricorsive - per es. prendere in input una linea di testo e, usando una pila, stamparla in senso inverso)
- Nei compilatori, per il processo di valutazione delle espressioni.



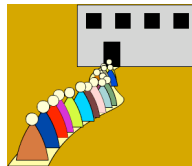
Code (1)

- Una coda (o queue) è una struttura dati che memorizza delle informazioni.
- Si possono aggiungere elementi solo in coda alla coda.
- Si possono eliminare elementi solo dalla testa della coda.
- La coda è anche detta struttura FIFO (first in first out)



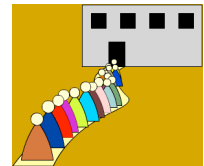
Code (2)

- Le principali funzioni usate per gestire una coda sono enqueue e dequeue, più la funzione che verifica se la coda è vuota.
- La loro implementazione dipende dal fatto che usiamo un vettore o una lista per gestire la pila.



Code (3)

- Implementazione con le liste**
 - Inserimento (enqueue):
 - Crea un nuovo nodo con malloc
 - Inserisci il nuovo nodo in coda
 - Cancellazione (dequeue):
 - Salva il valore del primo nodo
 - Cancella il nodo di testa
 - Rilascia la memoria
 - Verifica di coda vuota:
 - Guarda se la testa della pila punta a NULL



Code (4)

```
void enqueue(nl** ptesta,
nl** pcoda, int x)
{
    nl* paux;
    paux=malloc(sizeof(nl));
    paux->dato=x;
    paux->succ=NULL;
    if (codavuota(*ptesta))
        *ptesta=*pcoda=paux;
    else
    {
        *pcoda->succ=paux;
        *pcoda=paux;
    }
}

int dequeue(nl** ptesta)
{
    nl* paux; int x;
    if (!codavuota(*ptesta))
    {
        x=*ptesta->dato;
        paux=*ptesta;
        *ptesta=*ptesta->succ;
        free(paux);
        return x;
    }
    else
        printf("underflow");
}

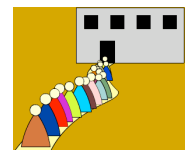
int codavuota(nl* testa)
{
    return (testa==NULL);
}
```

cosa succede in mem: alla lavagna



Code (5)

- Implementazione con i vettori**
N.B. Voglio evitare gli shift quindi inverte la struttura: inserisco in coda ed estraggo in testa
 - Inserimento (enqueue):
 - Se c'è spazio sufficiente nel vettore inserisci il nuovo elemento in coda
 - Aggiorna la lunghezza del vettore
 - Aggiorna l'indice di coda
 - Cancellazione (dequeue):
 - Salva il valore dell'elemento in testa
 - Aggiorna la lunghezza del vettore
 - Aggiorna l'indice di testa
 - Verifica di coda vuota:
 - Guarda se la lunghezza è 0



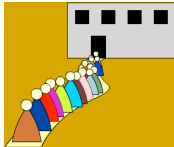
Code (6)

```
int dequeue(int coda[],
int*ptesta, int* pn)
{
    int x;
    if(!codavuota(*pn))
    {
        x=coda[*ptesta];
        (*pn)--=1;
        (*ptesta)+=1;
        return x;
    }
    else
        printf("underflow");
}

void enqueue(int coda[], int *pcoda,
int* pn, int x)
{
    if (*pn<N)
    {
        coda[*pcoda]=x;
        (*pn) +=1; (*pcoda)+=1;
    }
    else printf("overflow");
}

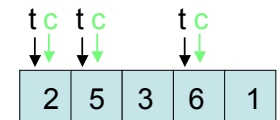
int codavuota(int n)
{
    return (n==0);
}
```

Attenzione!



cosa succede in mem: alla lavagna

Code (7)



n=0, t=0, c=0;

Enqueue 2: n=1, t=0, c=1;

Enqueue 5, 3: n=3, t=0, c=3;

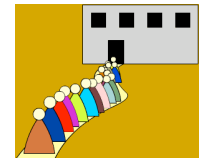
Dequeue 2: n=2, t=1, c=3;

Enqueue 6, 1: n=4, t=1, c=5;

Dequeue 5, 3: n=2, t=3, c=5;

Enqueue 4: n<N ma con c siamo fuori dal vettore!!

CODA CIRCOLARE



Code (8)

- Anche le code hanno molte interessanti applicazioni:
 - Le richieste dei vari utenti su una macchina monoprocesso sono immesse in una coda in attesa di essere servite
 - I pacchetti di dati che viaggiano in una rete, quando essa è già satura, attendono in una coda
 - ...

