

Operazioni sulle liste

- ▶ Definiamo una serie di procedure e funzioni per **operare** sulle liste.
- ▶ Usiamo liste di interi per semplicità, ma tutte le operazioni sono realizzabili in modo del tutto analogo su liste di altro tipo (salvo rare eccezioni)
- ▶ Facciamo riferimento alle dichiarazioni dei tipi `ElementoLista` e `ListaDiElementi` viste in precedenza

Inizializzazione

- ▶ Definiamo una procedura che inizializza una lista assegnando il valore `NULL` alla variabile **testa della lista**.
- ▶ Tale variabile deve essere modificata e quindi passata per **indirizzo**.
- ▶ Ciò provoca, nell'intestazione della procedura, la presenza di un puntatore a puntatore.

```
void Inizializza(ListaDiElementi *lista)
{
    *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```

PILA

Lista1	?
--------	---

```
void Inizializza(ListaDiElementi *lista)
{
    *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```

PILA

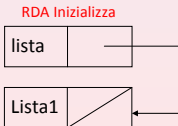


```
void Inizializza(ListaDiElementi *lista)
{
    *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```

PILA

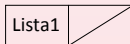


```
void Inizializza(ListaDiElementi *lista)
{
    *lista=NULL;
}
```

- ▶ Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```

PILA



Cosa succedrebbe se passassimo il parametro **per valore**?

```
void Inizializza(ListaDiElementi lista)
{
    lista=NULL;
}

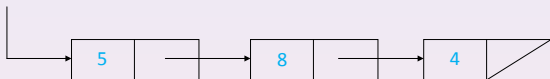
main() {
    ListaDiElementi Lista1;
    Inizializza(Lista1);
    ...
}
```

PILA

Lista1	?
--------	---

Stampa degli elementi di una lista

- ▶ Data la lista



vogliamo che venga stampato:

5 -> 8 -> 4 -> //

Versione iterativa:

Stampa Lista

```
void StampaLista(ListaDiElementi lis)
{
    while (lis != NULL)
    {
        printf("%d -->", lis->info);
        lis = lis->next;
    }
    printf("//");
}
```

- ▶ **N.B.:** `lis = lis->next` fa puntare `lis` all'elemento successivo della lista
- ▶ **Attenzione:** Possiamo usare `lis` per scorrere la lista perché, avendo utilizzato il passaggio per **valore**, le modifiche a `lis` non si ripercuotono sul parametro attuale.


```
void StampaLista(ListaDiElementi lis)
{
    while (lis != NULL)
        {
            printf("%d -->", lis->info);
            lis = lis->next;
        }
    printf("//");
}

main()
{
    ListaDiElementi Lista1;
    ...
    /* costruzione lista 5 --> 8 --> 4 */
    ...
    StampaLista(Lista1);
    ...
}
```

Cosa sarebbe successo passando il parametro per **indirizzo**?

```
void StampaLista(ListaDiElementi *lis)
{
    while (*lis != NULL)
        {
            printf("%d -->", *lis->info);
            *lis = *lis->next;
        }
    printf("//");
}

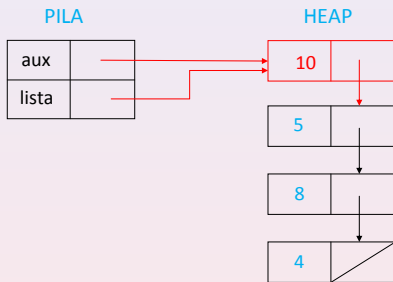
main()
{
    ListaDiElementi Lista1;
    ...
    /* costruzione lista 5 --> 8 --> 4 */
    ...
    StampaLista(&Lista1);
    ...
}
```

Versione ricorsiva

```
void StampaListaRic(ListaDiElementi lis)
{
    if (lis != NULL)
    {
        printf("%d ", lis->info);
        StampaListaRic(lis->next);
    }
    else
        printf("//");
}

main()
{
    ListaDiElementi Lista1;
    ...
    /* costruzione lista 5 --> 8 --> 4 */
    ...
    StampaListaRic(Lista1);
    ...
}
```

Inserimento di un nuovo elemento in testa



1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegnamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista viene fatto puntare alla nuova struttura
⇒ la lista da modificare deve essere passata per `indirizzo`

```
void InserisciTestaLista(ListaDiElementi *lista, int elem)
{
    ListaDiElementi aux;

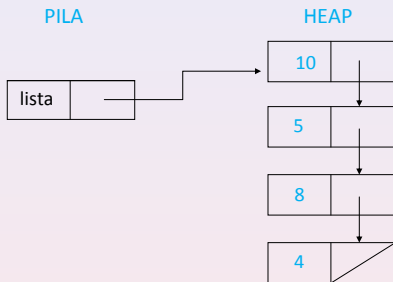
    aux = malloc(sizeof(ElementoLista));
    aux->info = elem;
    aux->next = *lista;
    *lista = aux;
}
```

- ▶ il primo parametro è la lista da modificare (passata per indirizzo)
- ▶ il secondo parametro è l'elemento da inserire (passato per indirizzo)
 - ▶ Attenzione: nel caso di liste di tipo `TipoElemLista` la procedura può essere generalizzata se su tale tipo è definito l'assegnamento

Esercizio

Impostare una chiamata alla procedura e tracciare l'evoluzione di pila e heap

Cancellazione del primo elemento



- ▶ se la lista è vuota non facciamo nulla
- ▶ altrimenti eliminiamo il primo elemento
 - ⇒ la lista deve essere passata per indirizzo
- ▶ **cancellare** significa anche **deallocare** la memoria occupata dall'elemento
 - ⇒ dobbiamo invocare **free** passando il puntatore all'elemento da cancellare
 - ⇒ è necessario un puntatore ausiliario

```
void CancellaPrimo(ListaDiElementi *lista)
{
    ListaDiElementi aux;
    if (*lista != NULL)
        {
            aux = *lista;
            *lista = (*lista)->next;
            free(aux);
        }
}
```

Cancellazione di tutta una lista

Una prima soluzione

```
void CancellaLista(ListaDiElementi *lista)
{
    ListaDiElementi aux;

    while (*lista != NULL) {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
    }
}
```

- Osserviamo che il corpo del ciclo corrisponde alle azioni della procedura `CancellaPrimo`. Possiamo allora scrivere:

Possiamo allora scrivere:

```
void CancellaLista(ListaDiElementi *lista)
{
    while (*lista != NULL)
        CancellaPrimo(lista);
}
```

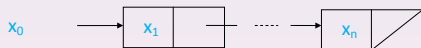
- Si noti il parametro attuale della chiamata a `CancellaPrimo`, che è `lista` (di tipo `ListaDiElementi *`) e non `&lista`


```
void CancellaPrimo(ListaDiElementi *lista)
{
    ListaDiElementi aux;
    if (*lista != NULL)
        {
            aux = *lista;
            *lista = (*lista)->next;
            free(aux);
        }
}
```

```
void CancellaLista(ListaDiElementi *lista)
{
    while (*lista != NULL)
        CancellaPrimo(lista);
}
```

```
main()
{
    ListaDiElementi lista;
    ...
    CancellaLista(&lista);
    ...
}
```

Visione ricorsiva delle liste



- ▶ Una lista di elementi è una struttura dati ricorsiva per sua natura
 1. data una lista L di elementi x_1, \dots, x_n
 2. dato un ulteriore elemento x_0
 3. anche la **concatenazione** di x_0 e L è una lista
- ▶ Si noti che in 1. L può anche essere la lista vuota

Visione ricorsiva delle liste



- ▶ Una lista di elementi è una struttura dati ricorsiva per sua natura
 1. data una lista L di elementi x_1, \dots, x_n
 2. dato un ulteriore elemento x_0
 3. anche la **concatenazione** di x_0 e L è una lista
- ▶ Si noti che in 1. L può anche essere la lista vuota

Cancellazione lista: versione ricorsiva

- ▶ Sfruttiamo la visione ricorsiva della struttura dati lista per realizzare la cancellazione in modo **ricorsivo**
 1. la cancellazione della lista vuota non richiede alcuna azione
 2. la cancellazione della lista ottenuta come concatenazione dell'elemento x e della lista L richiede l'eliminazione di x e la cancellazione di L
- ▶ la traduzione in **C** è immediata

```
void CancellaListaRic(ListaDiElementi *lista)
{
    ListaDiElementi aux;
    if (*lista != NULL)
    {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```

Appartenenza di un elemento ad una lista

- ▶ Ricordiamo la ricerca sequenziale sui vettori
- ▶ sostituiamo l'indice i con un puntatore p
- ▶ scorriamo la lista attraverso p
- ▶ l'elemento corrente è quello **puntato** da p
- ▶ Incapsuliamo questo codice in una funzione a valori booleani

Appartenenza di un elemento ad una lista

```
boolean Appartiene(TipoElementoLista elem, ListaDiElementi lista)
{
    boolean trovato = false;

    while (lista != NULL && !trovato)
        if (lista->info==elem)
            trovato = true;
        else
            lista = lista->next;
    return trovato;
}
```

- ▶ Non c'è bisogno di un puntatore ausiliario per scorrere la lista
⇒ il passaggio per **valore** consente di scorrere utilizzando il parametro formale!
- ▶ Abbiamo assunto che sul tipo `TipoElementoLista` sia definito l'operatore di uguaglianza `==`

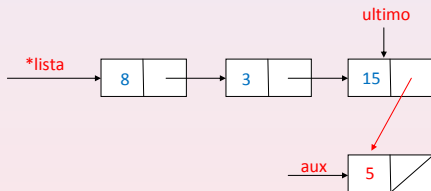
Versione ricorsiva

```
boolean Appartiene(TipoElementoLista elem, ListaDiElementi lis)
{
    if (lis == NULL)
        return false;
    else if (lis->info==elem)
        return true;
    else
        return (Appartiene(elem, lis->next));
}
```

- ▶ Un elemento `elem`
 - ▶ non appartiene alla lista vuota
 - ▶ appartiene alla lista con testa `x` se `elem` coincide con `x`
 - ▶ appartiene alla lista con testa `x` diversa da `elem` e resto `L` se e solo se appartiene a `L`

Inserimento di un elemento in coda

- ▶ Se la lista è vuota coincide con l'inserimento in testa
⇒ è necessario il passaggio per indirizzo!
- ▶ Se la lista non è vuota, bisogna scandirla fino in fondo
⇒ dobbiamo usare un puntatore ausiliario per la scansione
- ▶ La scansione deve terminare in corrispondenza dell'ultimo elemento al quale va collegato quello nuovo



Codice della versione iterativa

```
void InserzioneInCoda(ListaDiElementi *lista, TipoElementoLista elem)
{
    ListaDiElementi ultimo;    /* puntatore usato per la scansione */
    ListaDiElementi aux;

                                /* creazione del nuovo elemento */
    aux = malloc(sizeof(ElementoLista));
    aux->info = elem;
    aux->next = NULL;

    if (*lista == NULL)
        *lista = aux;
    else {
        ultimo = *lista;
        while (ultimo->next != NULL)
            ultimo = ultimo->next;
                                /* concatenazione del nuovo elemento in coda alla lista */
        ultimo->next = aux;
    }
}
```

Inserimento ricorsivo di un elemento in coda

Caratterizzazione **induttiva** dell'inserimento in coda

Sia **nuovaLista** la lista ottenuta inserendo **elem** in coda a **lista**.

- ▶ se **lista** è vuota, allora **nuovaLista** è costituita dal solo **elem** (**caso base**)
- ▶ altrimenti **nuovaLista** è ottenuta da **lista** facendo l'inserimento di **elem** in coda al resto di **lista** (**caso ricorsivo**)

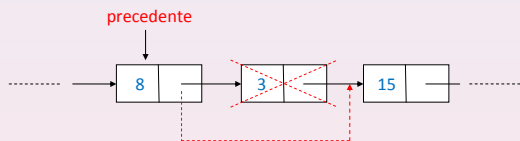
```
void InserzioneInCoda(ListaDiElementi *lista, TipoElementoLista elem)
{if (*lista == NULL)
  {
    *lista = malloc(sizeof(ElementoLista));
    (*lista)->info = elem;
    (*lista)->next = NULL;
  }
else
  InserisciCodaLista(      ??      , elem);}
```

Cancellazione della prima occorrenza di un elemento

- ▶ si scandisce la lista alla ricerca dell'elemento
- ▶ se l'elemento non compare non si fa nulla
- ▶ altrimenti, a seconda di dove si trova l'elemento, si distinguono tre casi
 1. l'elemento è il primo della lista: si aggiorna il puntatore iniziale in modo che punti all'elemento successivo
⇒ passaggio per indirizzo!!
 2. l'elemento non è né il primo né l'ultimo: si aggiorna il campo `next` dell'elemento che precede quello da cancellare in modo che punti all'elemento che segue
 3. l'elemento è l'ultimo: come (2), solo che il campo `next` dell'elemento precedente viene posto a `NULL`
- ▶ in tutti e tre i casi bisogna liberare la memoria occupata dall'elemento da cancellare

Osservazioni:

- ▶ per poter aggiornare il campo **next** dell'elemento precedente, bisogna **fermare la scansione sull'elemento precedente** (e non su quello da cancellare)



- ▶ per fermare la scansione dopo aver trovato e cancellato l'elemento, si utilizza una sentinella booleana

```

void CancellaElementoLista(ListaDiElementi *lista, TipoElementoLista elem)
{
    ListaDiElementi prec;    /* puntatore all'elemento precedente */
    ListaDiElementi corr;    /* puntatore all'elemento corrente */
    boolean trovato;        /* usato per terminare la scansione */

    if (*lista != NULL)
        if ((*lista)->info==elem)
            { /* cancella il primo elemento */
                CancellaPrimo(lista);
            }
        else { /* scansione della lista e cancellazione dell'elemento */
            prec = *lista; corr = prec->next; trovato = false;
            while (corr != NULL && !trovato)
                if (corr->info == elem)
                    { /* cancella l'elemento */
                        trovato = true;          /* provoca l'uscita dal ciclo */
                        prec->next = corr->next;
                        free(corr); }
                    else {
                        prec = prec->next;      /* avanzamento dei due puntatori */
                        corr = corr->next; }
        }
}}

```

Versione ricorsiva:

```
void CancellaElementoLista(ListaDiElementi *lista, TipoElementoLista elem)
{
    if (*lista != NULL)
        if ((*lista)->info== elem)
            { /* cancella il primo elemento */
                CancellaPrimo(lista);
            }
        else /* cancella elem dal resto */
            CancellaElementoLista(&((*lista)->next), elem);
}
```

Cancellazione di tutte le occorrenze di un elemento

Versione iterativa

- ▶ analoga alla cancellazione della prima occorrenza
- ▶ però, dopo aver trovato e cancellato l'elemento, bisogna continuare la scansione
- ▶ ci si ferma solo quando si è arrivati alla fine della lista
 - ⇒ non serve la sentinella booleana per fermare la scansione

Cancellazione di tutte le occorrenze di un elemento

Caratterizzazione induttiva

Sia ris la lista ottenuta cancellando tutte le occorrenze di $elem$ da $lista$.

Allora:

1. se $lista$ è la lista vuota, allora ris è la lista vuota (caso base)
2. altrimenti, se il primo elemento di $lista$ è uguale ad $elem$, allora ris è ottenuta da $lista$ cancellando il primo elemento e tutte le occorrenze di $elem$ dal resto di $lista$ (caso ricorsivo)
3. altrimenti ris è ottenuta da $lista$ cancellando tutte le occorrenze di $elem$ dal resto di $lista$ (caso ricorsivo)

Esercizio

Implementare le due versioni

Versione iterativa

```
void CancellaTuttiLista(ListaDiElementi *lista, TipoElementoLista elem)
{
    ListaDiElementi prec;    /* puntatore all'elemento precedente */
    ListaDiElementi corr;    /* puntatore all'elemento corrente */
    boolean trovato = false;
    while ((*lista != NULL) && !trovato) /* cancella le occorrenze */
        if ((*lista)->info!=elem)      /* di elem in testa      */
            trovato = true;
        else CancellaPrimo(lista);

    if (*lista != NULL)
        {
            prec = *lista; corr = prec->next;
            while (corr != NULL)
                if (corr->info == elem)
                    { /* cancella l'elemento */
                        prec->next = corr->next;
                        free(corr);
                        corr = prec->next;}
                else {
                    prec = prec->next; /* avanzamento dei due puntatori */
                    corr = corr->next; }
        }
}
```

Versione ricorsiva

```
void CancellaTuttiLista(ListaDiElementi *lista, TipoElementoLista elem)
{
    ListaDiElementi aux;

    if (*lista != NULL)
        if ((*lista)->info==elem)
            {
                CancellaPrimo(lista);                /* cancellazione del primo elemento */
                CancellaTuttiLista(lista, elem);    /* cancellazione di elem dal resto della lista */
            }
        else
            CancellaTuttiLista(&((*lista)->next), elem);
}
```

Inserimento di un elemento in una lista **ordinata**

- ▶ Data una lista (ad es. di interi) già ordinata (in ordine crescente), si vuole inserire un nuovo elemento **mantenendo l'ordinamento**.

Versione iterativa: per **esercizio**

Versione ricorsiva

- ▶ Caratterizziamo il problema **induttivamente**
- ▶ Sia **ris** la lista ottenuta inserendo l'elemento **elem** nella lista ordinata **lista**.
 1. se **lista** è la lista vuota, allora **ris** è costituita solo da **elem** (**caso base**)
 2. se il primo elemento di **lista** è maggiore o uguale a **elem**, allora **ris** è ottenuta da **lista** inserendo **elem** in testa (**caso base**)
 3. altrimenti **ris** è ottenuta da **lista** inserendo ordinatamente **elem** nel resto di **lista** (**caso ricorsivo**)

```
void InserzioneOrdinata(ListaDiElementi *lista, int elem)
{
    if (*lista == NULL)
        InserisciTestaLista(lista, elem);
    else
        if ((*lista) -> info >= elem)
            InserisciTestaLista(lista, elem);
        else
            InserzioneOrdinata(&((*lista)->next), elem);
}
```