

Liste

Il concetto di lista

- E' molto comune dover rappresentare **sequenze di elementi** tutti dello stesso tipo e fare operazioni su di esse.
- **Lista**: **sequenza** (*multi-insieme finito e ordinato*) di elementi dello **stesso tipo**
- **Multi-insieme**: insieme in cui un medesimo elemento può comparire più volte
- Notazione: **$L = [e_1, e_2, \dots, e_N]$**
- Esempi:
 - ['a', 'b', 'c'] denota la lista dei caratteri 'a', 'b', 'c'
 - [5, 8, 5, 21, 8] denota una lista di 5 interi

ADT lista

- Come ogni ***tipo di dato astratto***, la lista è definita in termini di:
 - ***dominio*** dei suoi elementi (dominio-base)
 - Insieme di funzioni (operazioni di ***costruzione*** e ***selezione***) sul tipo lista
 - Insieme di predicati sul tipo lista
- Una lista semplice è un tipo di dato astratto tale che:
 - Il dominio D può essere qualunque
 - Insieme di funzioni = { cons, head, tail, emptyList }
 - Insieme di predicati = { isEmpty }

ADT lista: operazioni primitive

Operazione	Descrizione
$\text{cons: } D \times \text{list} \rightarrow \text{list}$	Costruisce una nuova lista, aggiungendo l'elemento fornito in testa alla lista data
$\text{head: list} \rightarrow D$	Restituisce il primo elemento della lista data
$\text{tail: list} \rightarrow \text{list}$	Restituisce la coda della lista data
$\text{emptyList: } \rightarrow \text{list}$	Restituisce la lista vuota
$\text{isEmpty: list} \rightarrow \text{boolean}$	Restituisce vero se la lista data è vuota, falso altrimenti

- Ogni altra operazione sulle liste potrà essere implementata a partire da queste operazioni primitive elencate

Operazioni primitive: esempi

- `cons: D x list -> list` (*costruttore*)
- `cons(6, [7,11,21,3,6]) -> [6,7,11,21,3,6]`

- `head: list -> D` (*selettore “testa”*)
- `head([6,7,11,21,3,6]) -> 6`

- `tail: list -> list` (*selettore “coda”*)
- `tail([6,7,11,21,3,6]) -> [7,11,21,3,6]`

- `emptyList: -> list` (*costante “lista vuota”*)
- `emptyList() → []`

- `isEmpty: list -> boolean` (*test di “lista vuota”*)
- `isEmpty([]) -> true`

ADT lista

- Pochi linguaggi forniscono il tipo **lista** fra predefiniti (LISP, Prolog); per gli altri, ***ADT lista si costruisce a partire da altre strutture dati*** (in C tipicamente vettori o puntatori)
- Concettualmente, le operazioni precedenti costituiscono un ***insieme minimo completo*** per operare sulle liste
- Tutte le altre operazioni, *quali ad esempio inserimento (ordinato) di elementi, concatenamento di liste, stampa degli elementi di una lista, rovesciamento di una lista*, si possono ***definire in termini delle primitive precedenti***

Rappresentazione di liste in C

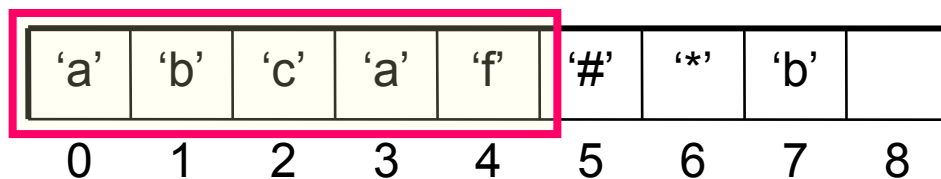
- Rappresentazione sequenziale:
 - Gli elementi della lista sono memorizzati uno dopo l'altro
 - Implementazione tramite vettori (statici o dinamici)

- Rappresentazione collegata
 - A ogni elemento si associa l'informazione (***indice, riferimento***) che permette di individuare la posizione dell'elemento successivo
 - Implementazione tramite vettori (poco usata) e strutture e puntatori (la studieremo in dettaglio)

Rappresentazione sequenziale

(statica)

- Si utilizza un **vettore** per memorizzare gli elementi della lista uno dopo l'altro
- **primo** memorizza l'indice del vettore in cui è inserito primo elemento
- **lunghezza** indica da quanti elementi è composta la lista
- Esempio: ['a', 'b', 'c', 'a', 'f']



0 primo

5 lunghezza

- Le componenti del vettore con indice pari o successivo a (primo + lunghezza) **non sono significative**

Rappresentazione sequenziale

(statica)

Vantaggi

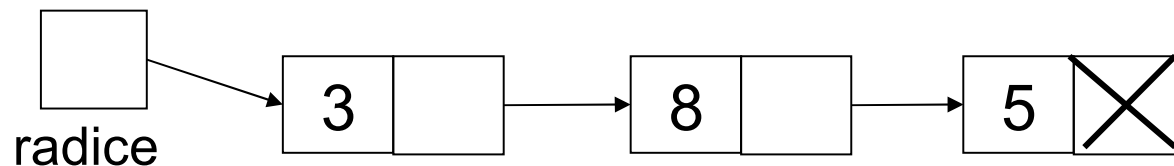
- Accesso agli elementi è diretto (tramite indice) ed efficiente
- L'ordine degli elementi è quello in memoria → non servono strutture dati particolari
- È semplice manipolare l'intera struttura

Svantaggi

- Dimensioni fisse del vettore → dimensione massima della lista
- Operazioni di inserimento e cancellazione di un elemento molto costose (bisogna spostare gli elementi che vengono dopo)

Rappresentazione collegata

- A ogni elemento si associa l'informazione (***indice***, ***riferimento***) che permette di individuare la posizione dell'elemento successivo
 - Sequenzialità elementi della lista non è più rappresentata mediante l'adiacenza delle locazioni di memorizzazione
- NOTAZIONE GRAFICA
 - elementi della lista come ***nodi***
 - riferimenti (indici) come ***archi***
- Esempio: [3, 8, 5]



Rappresentazione collegata

IMPLEMENTAZIONE MEDIANTE VETTORI

- Ogni elemento del vettore deve mantenere:
 - valore dell'elemento della lista (***dato***)
 - riferimento (***indice***) al prossimo elemento (***next***)

0	'a'	7
1		
2	'c'	9
3	'#'	
4	'f'	-1
5		
6	'*'	
7	'b'	2
8		
9	'a'	4

Esempio: ['a', 'b', 'c', 'a', 'f']

0 primo

5 lunghezza

Rappresentazione collegata

IMPLEMENTAZIONE MEDIANTE VETTORI

Vantaggi

- Cancellazione ed inserimento di elementi più efficienti: non richiedono più lo spostamento fisico di elementi, ma solo aggiornamento di riferimenti

Svantaggi

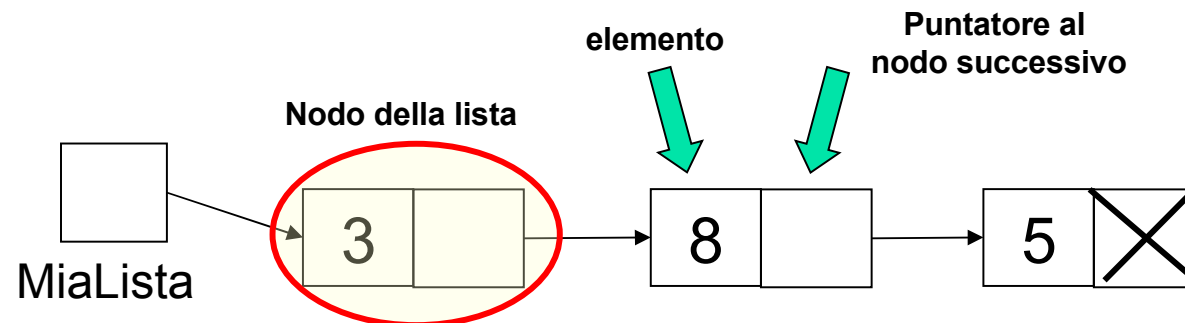
- Maggiore spazio occupato
- Dimensioni fisse del vettore → dimensione massima della lista
- Gestione della **lista libera** → cioè degli elementi del vettore non ancora occupati da dati

Rappresentazione collegata

IMPLEMENTAZIONE MEDIANTE PUNTATORI

- Problema della dimensione massima del vettore: adottare un approccio basato su **allocazione dinamica** memoria
- Ciascun nodo della lista è una struttura di due campi:
 - **valore** dell'elemento
 - un **puntatore** al nodo successivo della lista (NULL se ultimo elemento)
- L'intera sequenza è **rappresentata** da un **puntatore al suo primo elemento** (vogliamo un'unica variabile per accedere a tutti gli elementi della sequenza)
NOTA: tale puntatore **non è** la sequenza, ma la rappresenta soltanto

- Esempio: [3, 8, 5]



Rappresentazione collegata

IMPLEMENTAZIONE MEDIANTE PUNTATORI

Vantaggi

- Non si ha più il problema della dimensione massima: ogni qualvolta devo aggiungere un nodo posso allocare dinamicamente la memoria necessaria
- Cancellazione ed inserimento di elementi efficienti: non richiedono più lo spostamento fisico di elementi, ma solo aggiornamento di riferimenti (puntatori)

Svantaggi

- Maggiore spazio occupato per ogni nodo: ho comunque solo un puntatore per ogni elemento. Più la struttura è complessa e più tale overhead diventa trascurabile.
- Accesso non diretto agli elementi (in molte applicazioni può non essere un problema)

Liste: Rappresentazione collegata

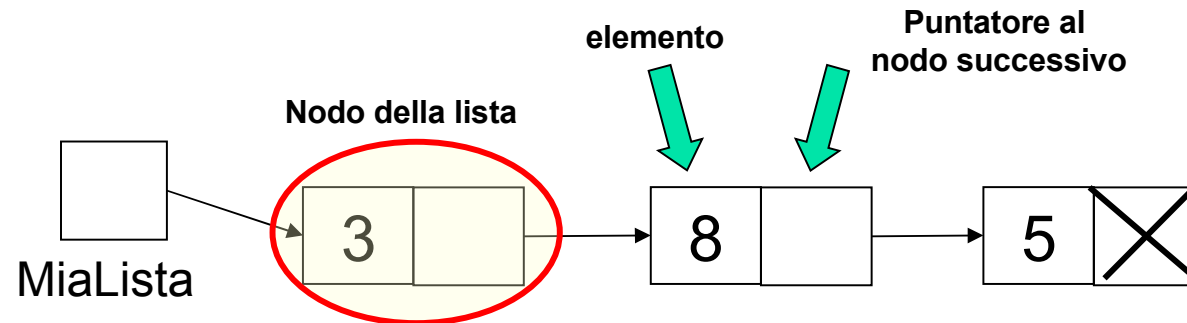
Implementazione mediante puntatori e strutture

Rappresentazione collegata

IMPLEMENTAZIONE MEDIANTE PUNTATORI

- Problema della dimensione massima del vettore: adottare un approccio basato su **allocazione dinamica** memoria
- Ciascun nodo della lista è una struttura di due campi:
 - **valore** dell'elemento
 - un **puntatore** al nodo successivo della lista (NULL se ultimo elemento)
- L'intera sequenza è **rappresentata** da un **puntatore al suo primo elemento** (vogliamo un'unica variabile per accedere a tutti gli elementi della sequenza)
NOTA: tale puntatore **non è** la sequenza, ma la rappresenta soltanto

- Esempio: [3, 8, 5]



Implementazione liste

Dichiarazioni di tipo

```
struct nodo {
```

```
    [contenuto]
```

E' il contenuto dell'elemento della lista. Può essere un tipo semplice (int, char, float) oppure un tipo strutturato definito dall'utente

```
    struct nodo *prossimo;
```

Puntatore al nodo successivo della lista

```
};
```

```
typedef struct nodo Tnodo;
```

Tnodo è il tipo dei nodi costituenti la lista.

Inizializzazione di una lista

Una lista e' associata ad un puntatore iniziale che si riferisce al primo elemento della suddetta. Si inizializza una lista vuota associandola ad un puntatore di testa (head) nullo.

```
Tnodo * head = NULL;
```

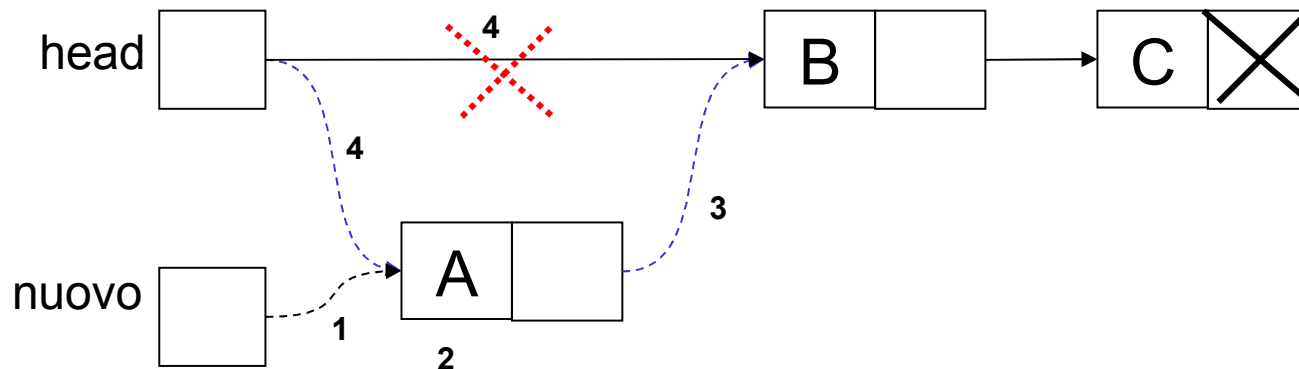
Tutte le funzioni per l'accesso e/o modifica della lista restituiscono un puntatore head che fa riferimento alla lista aggiornata.

```
Tnodo * <nome_funzione>(..., Tnodo * lista){  
    /* ...codice per la modifica della lista... */  
    return lista;  
}
```

Inserimento di un nuovo elemento in testa

`add_to_head` (*cons*)

- allochiamo una nuova struttura per l'elemento (usando *malloc*)
- assegnamo il valore da inserire nella struttura
- concateniamo la nuova struttura con la vecchia lista
- il puntatore iniziale della lista viene fatto puntare alla nuova struttura



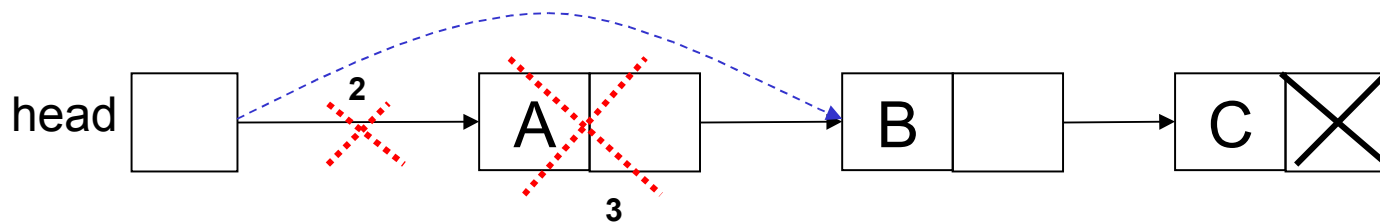
Inserimento di un nuovo elemento in testa

Inserimento dell'elemento u in testa alla lista $lista$

```
Tnodo * add_to_head(Tnodo u, Tnodo * lista)
{
    Tnodo * nuovo;
    nuovo = (Tnodo*)malloc(sizeof(Tnodo));
    *nuovo = u;
    nuovo->prossimo = lista;
    return nuovo;
}
```

Cancellazione del primo elemento

- “Cancellare” significare deallocare la memoria occupata dall’elemento.
 - se la lista è vuota non facciamo nulla
 - altrimenti deallochiamo la struttura del primo elemento (con *free*)
 - la lista deve essere passata per indirizzo



Cancellazione del primo elemento

Cancella il primo elemento dalla lista *lista*

```
Tnodo * remove_head(Tnodo * lista)
{
    Tnodo * tmp;
    if (lista==NULL) return NULL;

    tmp = lista->prossimo;
    free(lista);
    return tmp;
}
```

Cancellazione di tutta la lista

Cancella tutti gli elementi della lista liberando la memoria occupata

```
Tnodo * free_list(Tnodo * lista)
{
    Tnodo * successivo;
    while (lista!=NULL)
    {
        successivo = lista->prossimo;
        free(lista);
        lista = successivo;
    }
    return NULL;
}
```

Ricerca di un elemento in una lista

Scandiamo la lista continuando fino a che:

- l'elemento viene trovato, in base ad un criterio di uguaglianza rappresentato da una certa funzione (es. In questo caso assumiamo che “compara” e' uguale a 0 se gli elementi sono uguali)
- non siamo giunti alla fine della lista (lista == NULL)

```
Tnodo * search_in_list(Tnodo u, Tnodo * lista)
{
    while (lista != NULL)
    {
        if (compara(u, *lista) == 0)
            return lista;
        lista = lista->prossimo;
    }
    return NULL;
}
```

Inserimento in coda

Algoritmo:

alloca una struttura per il nuovo elemento

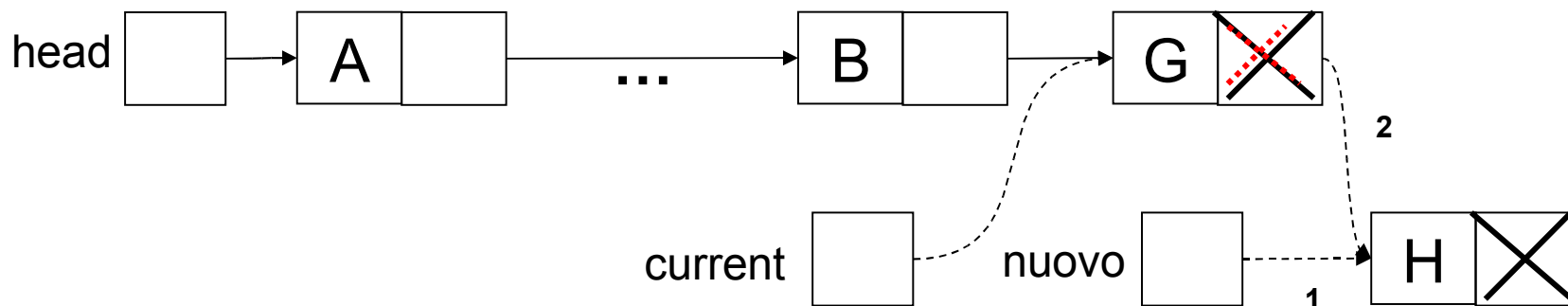
if la *lista* è vuota

 restituisce la *lista* costituita dal solo elemento

else

 scandisci la *lista* fermandoti quando il puntatore corrente punta all'ultimo elem.

concatena il nuovo elemento con l'ultimo



Inserimento in coda

```
Tnodo * add_to_tail(Tnodo u, Tnodo * lista)
{
    Tnodo * current;
    Tnodo * nuovo;
    nuovo = (Tnodo*)malloc(sizeof(Tnodo));
    *nuovo = u;
    nuovo->prossimo = NULL;

    if (lista==NULL) return nuovo;

    current = lista;

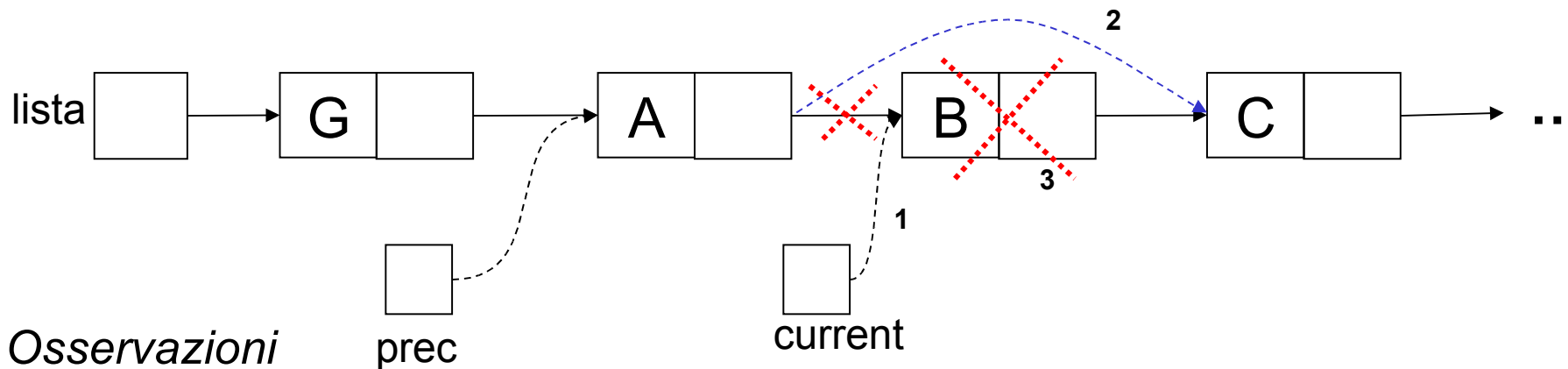
    while(current->prossimo!=NULL)
        current = current->prossimo;

    current->prossimo = nuovo;
    nuovo->prossimo = NULL;
    return lista;
}
```

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
 - 2. l'elemento non è ne il primo ne 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

Cancellazione della prima occorrenza di un elemento



Osservazioni

- per poter aggiornare il campo *prossimo* dell'elemento precedente è necessario utilizzare un puntatore apposito oltre a quello per la scansione *current*.
- dopo aver trovato e cancellato l'elemento, la scansione ha termine poiché si restituisce il controllo al chiamante mediante *return*.

Cancellazione della prima occorrenza di un elemento

```
Tnode * remove_item(Tnode u, Tnode * lista)  
{  
    Tnode * prec, *current;  
    if (lista==NULL) return NULL;  
    if (compara(u,*lista)==0) return remove_head(lista);  
    prec = lista;  
    current = lista->prossimo;  
    while(current!=NULL) {  
        if (compara(*current,u)==0) {  
            prec->prossimo = current->prossimo;  
            free(current);  
            return lista;  
        }  
        prec = current;  
        current = current->prossimo;  
    }  
    return lista; /* non trovato! */  
}
```

Cancellazione di tutte le occorrenze di un elemento

- Analoga alla cancellazione della prima occorrenza di un elemento
- però, dopo aver trovato e cancellato l'elemento, bisogna continuare la scansione
- ci si ferma solo quando si è arrivati alla fine della lista

Osservazioni:

- Il codice relativo e' una semplice variante della cancellazione di un elemento.
- ...E' lasciata allo studente per esercitazione.

Visita di una lista

- L'operazione di visita permette di *visitare* una sola volta ogni elemento della lista
- Per ogni elemento visitato, tipicamente, all'interno di una applicazione effettuerò la stampa dei dati del nodo o altre elaborazioni dipendenti dal contesto
- Nell'esempio sotto, viene richiamata l'ipotetica funzione `Stampa_elemento()`

```
void scan_list(Tnodo * lista)
{
    while (lista!=NULL)
    {
        Stampa_elemento(lista);
        lista = lista->prossimo;
    }
}
```

Sommario funzioni

- **add/remove_head** Aggiunge/Rimuove il primo elemento dalla lista
- **free_list** Cancella tutti gli elementi della lista
- **seach_in_list** Ricerca un dato elemento nella lista
- **add/remove_to_tail** Aggiunge/Rimuove un elemento in coda alla lista
- **remove_item** Cancella la prima occorrenza di un elemento
- **remove_items** Cancella tutte le occorrenze di un elemento
- **scan_list** Visita gli elementi di una lista

Liste ordinate

- E' necessario che sia definita una **relazione d'ordine** sul **dominio-base** degli elementi della lista
- NOTA: criterio di ordinamento dipende da **dominio base** e dalla specifica **necessità applicativa**
- Ad esempio:
 - interi ordinati in senso crescente, decrescente, ...
 - stringhe ordinate in ordine alfabetico, in base alla loro lunghezza,...
 - persone ordinate in base all'ordinamento alfabetico del loro cognome, all'età, al codice fiscale, ...
- NB: come negli esempi precedenti, assumeremo una generica funzione `compara(...)` che resituisce valori < 0 , $= 0$, > 0 a seconda il primo argomento preceda, sia uguale o sia successivo a quello del secondo argomento.

Liste ordinate: inserimento

- Data una lista (ad es. di interi) già ordinata, si vuole inserire un nuovo elemento, mantenendo l'ordinamento
- Si distinguono 5 possibilità nell'inserimento del nuovo elemento:
 1. lista vuota
 2. prima della testa
 3. secondo e ultimo
 4. nel mezzo della lista
 5. in coda alla lista

Liste ordinate: inserimento (1/2)

```
Tnodo * sorted_insert(Tnodo u, Tnodo * lista)
{
    Tnodo * nuovo, * current, * prec;

    if (lista == NULL || compara(u, *lista) < 0) /*CASI 1,2*/
        return add_to_head(u, lista);

    if (lista->prossimo == NULL) /*CASO 3*/
        return add_to_tail(u, lista);

    prec = lista;
    current = lista->prossimo;

    while (current != NULL)
    { /*CASO 4*/
        if (compara(u, *current) < 0)
        {
            nuovo = (Tnodo*) malloc(sizeof(Tnodo));
```

Liste ordinate: inserimento (2/2)

```
        *nuovo = u;
        prec->prossimo = nuovo;
        nuovo->prossimo = current;
        return lista;
    }
    prec = current;
    current = current->prossimo;
}

/*CASO 5*/
nuovo = (Tnodo*)malloc(sizeof(Tnodo));
*nuovo = u;
prec->prossimo = nuovo;
nuovo->prossimo = NULL;
return lista;
}
```