Problema:

Data una sequenza di elementi in ordine qualsiasi, ordinarla.

- Questo è un problema fondamentale, che si presenta in moltissimi contesti, ed in diverse forme.
- Nel nostro caso formuliamo il problema in termini di ordinamento di vettori:

  Dato un vettore A di n elementi, ordinarlo in modo crescente
- Per semplicità faremo sempre riferimento a vettori di interi.

$$\begin{array}{c} 5\ 2\ 4\ 6\ 1\ 3 \\ \Longrightarrow \\ 1\ 2\ 3\ 4\ 5\ 6 \end{array}$$

Ordinamento per inserzione (insertion sort)

- **Esempio:** Ordinamento di una mano di ramino
  - si inizia con la mano sinistra vuota e le carte coperte sul tavolo
  - si prende dalla tavola una carta alla volta e la si inserisce nella corretta posizione nella mano sinistra
  - ...
  - si termina quando si sono finite tutte le carte sul tavolo.
- Stesso procedimento per ordinare un vettore:
  - inizialmente il vettore rappresenta il mazzo sul tavolo
  - si usa un ciclo per analizzare uno alla volta gli elementi del vettore
  - Alla generica iterazione la situazione è la seguente

mano sinistra	carte ancora da scoprire
	↑ nuova carta
  - per inserire la nuova carta al posto giusto nella mano sinistra dobbiamo
    - scorrere gli elementi che lo precedono per decidere la posizione che gli compete
    - spostare di un posto verso destra gli elementi maggiori per fargli spazio.

Esempio

In verde le carte ancora da esaminare, in rosso quelle già esaminate (mano sinistra). La nuova carta da esaminare è sottolineata.

5	2	4	6	1	3
5	2	4	6	1	3
2	5	4	6	1	3
2	4	5	6	1	3
2	4	5	6	1	3
1	2	4	5	6	3
1	2	3	4	5	6

- Una volta individuata la posizione k in cui inserire la nuova carta dobbiamo farle spazio, ovvero spostare verso destra di una posizione tutte le carte rosse da k in poi.

  Esempio:

1	2	4	5	6	3
1	2	4	5	6	3
1	2	4	5		6
1	2	4		5	6
1	2		4	5	6

- A questo punto possiamo piazzare la carta in modo ordinato

1	2	3	4	5	6

- Definiamo allora una procedura che sposta tutti gli elementi di un vettore verso destra di una posizione tra due indici dati `from` e `to`

```
void shiftR(int v[], int from, int to)
{
int i;
for (i=to-1; i>=from; i--)
  v[i+1] = v[i];
}
```

- L'elemento in posizione `to` viene perso
- Bisogna procedere da destra verso sinistra (perché?)
- se `to` è minore o uguale a `from` non succede nulla

Possiamo allora definire la procedura di ordinamento per inserzione come segue

```
void sort(int v[], int dim)
{
int h, curr, j=1;
while (j<dim)
  { h=0;
    curr=v[j];
          /* curr e' l'elemento da piazzare */

    while((v[h]<curr) && (h<j))
       h++;
          /* curr va inserito in posizione h */

    shiftR(v,h,j);
     v[h]=curr;
    j++;
   }
}
```

Possiamo anche rendere la procedura più efficiente, nel modo seguente

```
void sort (int v[], int dim) {
  int i, j, prossimo;
  for (i = 1; i < dim; i++) {
     prossimo = v[i];
     j = i;
     while ((j > 0) && (v[j-1] > prossimo)) {
        v[j] = v [j-1];
        j--;
     }
     v[j] = prossimo;
  }
}
```


Ordinamento per selezione del minimo (selection sort)

- **Esempio:** Ordinamento di un mazzo di carte
  - si seleziona la carta più piccola e si mette da parte
  - delle rimanenti si seleziona la più piccola e si mette da parte
  - ...
  - si termina quando rimane una sola carta
- Ordinamento di un vettore:
  - per selezionare l'elemento più piccolo tra quelli rimanenti si utilizza un ciclo
  - mettere da parte significa scambiare con l'elemento che si trova nella posizione che compete a quello selezionato

- in verde la parte che rimane da analizzare
  in blu l'elemento minimo selezionato
  in marrone lo scambio effettuato
  in rosso la parte ordinata

```
5  2  4  6  1  3
   5  2  4  6  1  3
   1  2  4  6  5  3
1  2  4  6  5  3
   1  2  4  6  5  3
   1  2  4  6  5  3
1  2  4  6  5  3
   1  2  4  6  5  3
   1  2  3  6  5  4
1  2  3  6  5  4
   1  2  3  6  5  4
   1  2  3  4  5  6
1  2  3  4  5  6
   1  2  3  4  5  6
   1  2  3  4  5  6
1  2  3  4  5  6

1  2  3  4  5  6
```


Implementazione

```
int minPos(int v[], int from, int to);
/* calcola la posizione del minimo elemento di
   v nella porzione [from,to]                */

void swap(int *p, int *q);
/* scambia le variabili puntate da p e q */

/** PROCEDURA DI ORDINAMENTO PER SELEZIONE **/

void sort(int v[], int dim)
{
  int i, min;
  for(i=0; i<dim-1; i++)
    {
      min = minPos(v, i, dim-1);
      swap(v+i, v+min);
    }
}
```

Scrivere per **esercizio** le procedure `swap` e `minpos`

```
int minPos(int v[], int from, int to) {
/* calcola la posizione del minimo elemento di
   v nella porzione [from,to]                */

    int i, pos;
    pos = from;
    for (i=from+1; i<=to; i++)
        if (v[i] < v[pos])
            pos = i;
    return pos;
}

void swap(int *p, int *q) {
/* scambia le variabili puntate da p e q */
    int temp = *p;
    *p = *q;
    *q = temp;
}
```


Ordinamento a bolle (bubble sort)

- Si fanno salire gli elementi più piccoli ("più leggeri") verso l'inizio del vettore ("verso l'alto"), scambiandoli con quelli adiacenti.
- L'ordinamento è suddiviso in n-1 fasi:
  - fase 0: 0^o elemento (il più piccolo) in posizione 0
  - fase 1: 1^o elemento in posizione 1
  - ...
  - fase n-2: $(n-2)^o$ elemento in posizione n-2, e quindi $(n-1)^o$ elemento in posizione n-1
- Nella fase i: cominciamo a confrontare dal basso e portiamo l'elemento più piccolo (più leggero) in posizione i

5 2 4 6 1 3
5 2 4 6 1 3
5 2 4 1 6 3
5 2 1 4 6 3
5 1 2 4 6 3
1 5 2 4 6 3
1 5 2 4 6 3
1 5 2 4 3 6
1 5 2 3 4 6
1 5 2 3 4 6
1 2 5 3 4 6
1 2 5 3 4 6

1 2 3 5 4 6

1 2 3 4 5 6

1 2 3 4 5 6


```
/** PROCEDURA BUBBLE SORT **/


void sort(int v[], int dim)
{
  int temp,i,j;
  for (i = 0; i < dim-1; i++)      /* fase i-esima */

    for (j = dim-1; j > i; j--)    /* bolla piu' leggera in posizione i */
      if (v[j] < v[j-1])
        swap(v+j, v+j-1);
}
```

Ordinamenti ricorsivi

Selection Sort ricorsivo

- Il metodo del selection sort può essere facilmente realizzato in modo ricorsivo
- si definisce una procedura che ordina (ricorsivamente) la porzione di array individuata da due indici `from` e `to`
- il minimo elemento della porzione viene messo in posizione `from` per poi ordinare ricorsivamente la porzione tra `from+1` e `to`
- Il caso base corrisponde all'ordinamento di una porzione fatta da un solo elemento (è già ordinata)

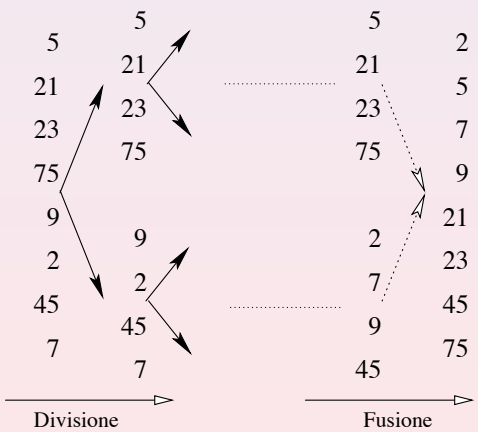
```
void SelectionSort(int v[], int from, int to){
  if (from < to) {
    int min = minPos(v,from,to);
    swap(v+from, v+min);
    SelectionSort(v, from+1, to);
  } }

void sort(int v[], int dim) {
    SelectionSort(v,0,dim-1);
}
```


Merge sort

Si divide il vettore da ordinare in due parti:

- si ordina ricorsivamente la prima parte
- si ordina ricorsivamente la seconda parte
- si combinano (operazione di fusione, merge) le due parti ordinate

Esempio:



Esprimiamo il procedimento in uno pseudo-linguaggio

```
ordina per fusione gli elementi di A da from a to

IF from < to (c'è più di un elemento tra from e to)

THEN
     mid =  (from + to) /2
     ordina per fusione gli elementi di A da from a mid
     ordina per fusione gli elementi di A da mid +1 a to
     fondi
        gli elementi di A da from a mid con
        gli elementi di A da mid+1 a to
        restituendo il risultato nel sottovettore
         di A da from a to
```


Implementiamo l'algoritmo in C, definendo una procedura ricorsiva

```
void mergeRicorsivo(int A[], int from, int to)
```

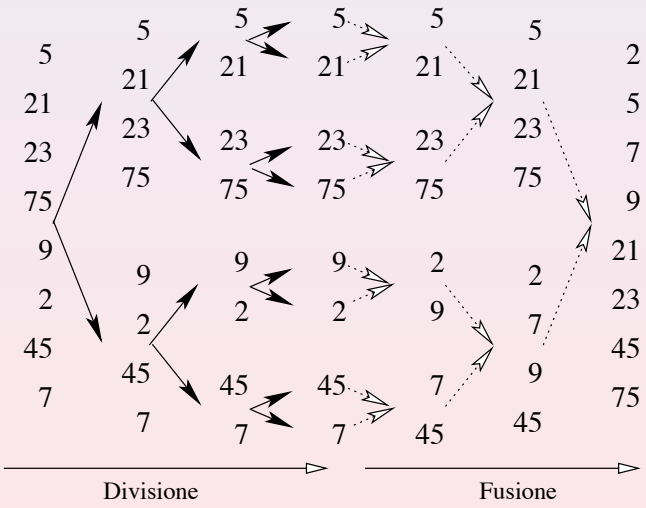
che ordina la porzione dell'array A individuata dagli indici from e to.

```
void mergeRicorsivo(int A[], int from, int to)

{
  int mid;

  if (from < to) {              /* l'intervallo da mid a to, estremi
                                   inclusi, comprende almeno due elementi */
    mid = (from + to) / 2;

    mergeRicorsivo(A, from, mid);
    mergeRicorsivo(A, mid+1, to);

    merge(A, from, mid, to);    /* fonde le due porzioni ordinate [from, mid],
                                   [mid+1, to] nel sottovettore [from, to]  */
  }
}
```

La procedura `mergeSort` che ordina un array di interi è semplicemente

```
void sort(int v[], int dim)
{
mergeRicorsivo(v, 0, dim-1);
}
```

Esempio:



- Vediamo l'operazione di *fusione*, definendo la procedura

  `void merge(int A[], int from, int mid, int to)`

  che fonde le due porzioni dell'array `A` con indici compresi tra `from` e `mid` e tra `mid+1` e `to`.
- La procedura utilizza un array di supporto `B`: per semplicità, supponiamo di avere una costante `LUNG` che definisce la lunghezza degli array che stiamo trattando.

```
void merge(int A[], int from, int mid, int to)
 {
  int B[LUNG];                              /* vettore di appoggio */
  int primo, secondo, appoggio, da_copiare;

  primo = from;
  secondo = mid + 1;
  appoggio = from;

  while (primo <= mid && secondo <= to) { /* copia in modo ordinato      */
    if (A[primo] <= A[secondo]) {         /* gli elementi della prima e  */
      B[appoggio] = A[primo];             /* della seconda porzione in B */
      primo++;                            /* fino ad esaurire una delle due */
    }
    else {
      B[appoggio] = A[secondo];
      secondo++;
    }
    appoggio++;
  }
```



```
 if (secondo > to)              /* e' finita prima la seconda porzione */
 /* copia da A in B tutti gli elementi della
    prima porzione fino a mid */

    for (da_copiare = primo; da_copiare <= mid; da_copiare++) {
       B[appoggio] = A[da_copiare];
       appoggio++;
    }
  else                           /* e' finita prima la prima porzione */

    for (da_copiare = secondo; da_copiare <= to; da_copiare++) {
    /* copia da A in B tutti gli elementi della
    /* seconda porzione fino a to */

       B[appoggio] = A[da_copiare];
       appoggio++;
    }

/* ricopia tutti gli elementi da from a to da B ad A */
  for (da_copiare = from; da_copiare <= to; da_copiare++)
    A[da_copiare] = B[da_copiare];
}
```

Liste

- È molto comune dover rappresentare sequenze di elementi tutti dello stesso tipo e fare operazioni su di esse.

  Esempi: sequenza di interi (23 46 5 28 3)
  sequenza di caratteri (`'x' 'r' 'f'`)
  sequenza di persone con nome e data di nascita
- Finora abbiamo usato gli array per realizzare tali strutture, nonostante ciò porti talvolta a un impiego inefficiente della memoria.
- Vediamo adesso un modo basato sull'allocazione dinamica di variabili, che ci permette di realizzare liste di elementi in maniera che la memoria fisica utilizzata corrisponda meglio a quella astratta, cioè al numero di elementi della sequenza che vogliamo rappresentare.

Diversi modi di rappresentare sequenze di elementi

1. Rappresentazione sequenziale: tramite array
   - Vantaggi:
     - l'accesso agli elementi è diretto (tramite indice) ed efficiente
     - l'ordine degli elementi è quello in memoria $\Longrightarrow$ non servono strutture dati addizionali
     - è semplice manipolare l'intera struttura (copia, ordinamento, ...)
   - Svantaggi:
     - dobbiamo avere un'idea precisa della dimensione della sequenza
     - inserire o eliminare elementi è complicato ed inefficiente (comporta un numero di spostamenti che nel caso peggiore può essere dell'ordine del numero degli elementi della struttura)

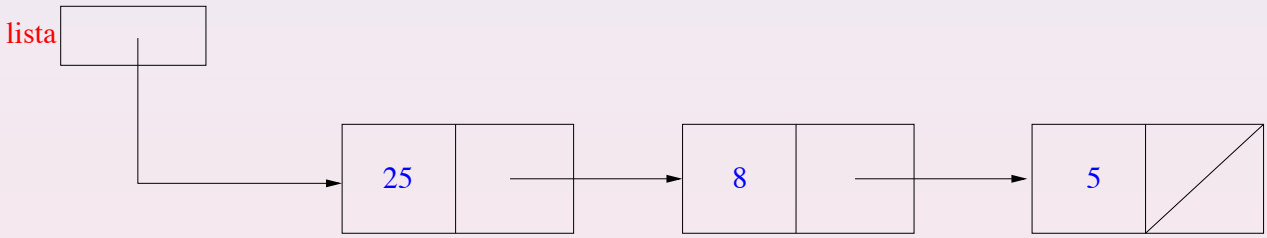
2. Rappresentazione collegata

- Una lista concatenata è una sequenza lineare di nodi, ciascuno dei quali memorizza un valore e contiene un riferimento (puntatore) al nodo successivo nella sequenza.
- Per aggiungere e cancellare nodi in qualunque posizione semplicemente aggiustando il sistema di puntatori senza operare sui nodi non interessati dalla aggiunta o dalla cancellazione.
- L'accesso agli elementi è di tipo sequenziale: per accedere al generico nodo, si deve scandire la lista, dato che l'accesso ad un elemento è possibile attraverso il puntatore contenuto nell'elemento precedente.

2. Rappresentazione collegata (continua)

- La sequenza di elementi viene rappresentata da una struttura di dati collegata, realizzata tramite strutture e puntatori.
- Ogni elemento è rappresentato con una struttura C:
  - un campo (o più campi se necessario) per l'elemento (ad es. `int`)
  - un campo puntatore alla struttura che rappresenta l'elemento successivo (ovviamente, tale struttura ha tipo indentico a quello della struttura corrente)
- L'ultimo elemento non ha un elemento successivo
  - il campo puntatore ha valore `NULL` che assume quindi il significato di "fine lista".
- L'inizio della lista è individuato da una variabile del tipo dei puntatori ai vari elementi.
  - Sarà nostra abitudine attribuire a questa variabile il nome stesso della lista, identificando il concetto di "inizio lista" (o "testa della lista") con la lista stessa.
- l'accesso a una lista avviene attraverso il puntatore al primo elemento.

Graficamente



- La variabile lista, di tipo puntatore, è utilizzata per accedere alla sequenza.

Esempio: Sequenze di interi.

```
struct EL {
  int info;
  struct EL *next;
};
typedef struct EL ElementoLista;
typedef ElementoLista *ListaDiElementi;
```

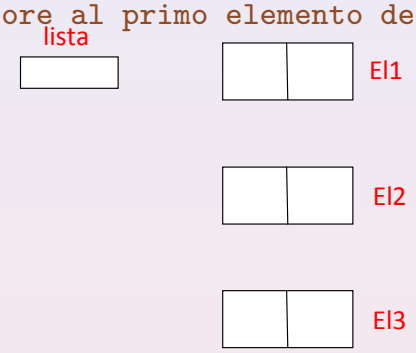
1. La prima dichiarazione `struct EL` definisce un primo campo, `info`, di tipo `int` e permette di dichiarare il campo `next` come puntatore al tipo strutturato che si sta definendo;
2. la seconda dichiarazione utilizza `typedef` per ridenominare il tipo `struct EL` come `ElementoLista`;
3. la terza dichiarazione definisce il tipo `ListaDiElementi` come puntatore al tipo `ElementoLista`.

- A questo punto possiamo definire variabili di tipo lista:

```
ListaDiElementi Lista1, Lista2;
```

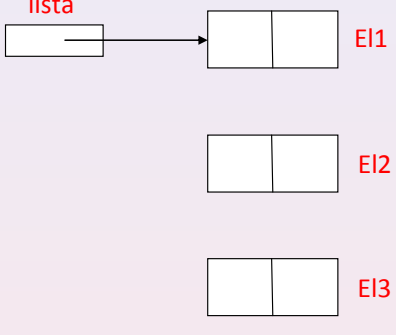
Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



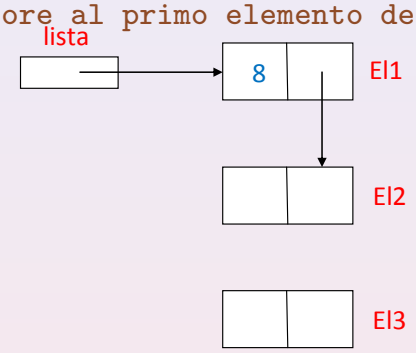
Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



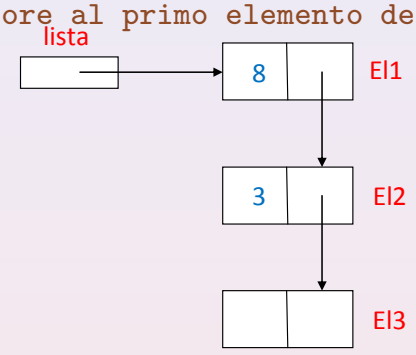
Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



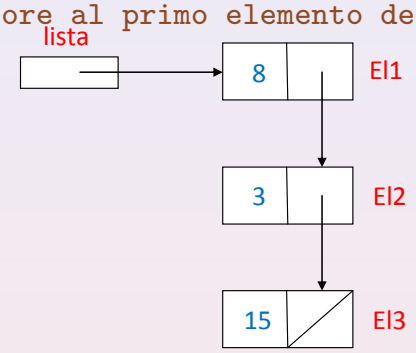
Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



- Nel programma che abbiamo appena visto per creare una lista di tre elementi dobbiamo dichiarare tre variabili di tipo `ElementoLista`.
- Il numero degli elementi della lista deve essere deciso a tempo di compilazione.
- Non è possibile, per esempio, creare una lista con un numero di elementi letti in ingresso. Ma allora qual è il vantaggio rispetto all'array?
- Quello che abbiamo visto non è l'unico modo. . . .

Allocazione Dinamica della memoria

- L'allocazione dinamica della memoria è possibile in C grazie all'utilizzo di alcune funzioni messe a disposizione dalla libreria standard (standard library). Infatti è richiesta l'inclusione del file header <stdib.h>
- Le due funzioni principali sono
  - malloc: consente di allocare dinamicamente memoria per una variabile di un tipo specificato
  - free: consente di rilasciare dinamicamente memoria (precedentemente allocata con malloc)
- I tipi di dato sono ancora statici, ovvero hanno una dimensione fissata a priori. Le variabili di un certo tipo di dato possono invece essere create.

malloc

- La chiamata di funzione

  malloc(sizeof(TipoDato));

  crea in memoria una variabile di tipo TipoDato, e restituisce come risultato l'indirizzo della variabile creata.
- Se p è una variabile di tipo puntatore a TipoDato, l'istruzione

  p=malloc(sizeof(TipoDato));

  assegna l'indirizzo restituito dalla funzione malloc a p che punta quindi alla nuova variabile (p già esiste).
- Una variabile creata dinamicamente è necessariamente anonima: a essa si può fare riferimento solo tramite un puntatore

  a differenza di una variabile dichiarata mediante un proprio identificatore, che può essere riferita sia direttamente sia tramite un puntatore

free

- Se p è l'indirizzo di una variabile allocata dinamicamente, la chiamata

  free(p);

  rilascia lo spazio di memoria puntato da p

  la corrispondente memoria fisica è resa disponibile per qualsiasi altro uso.
- free deve ricevere come parametro attuale un puntatore al quale era stato assegnato come valore l'indirizzo restituito da una funzione di allocazione dinamica di memoria (cioè malloc).

Heap

- Poiché le variabili dinamiche possono essere create e distrutte in un qualsiasi punto del programma esse non possono essere allocate sullo stack.
- Vengono allocate in un'altra zona di memoria chiamata heap (mucchio). La loro gestione risulta molto più inefficiente.

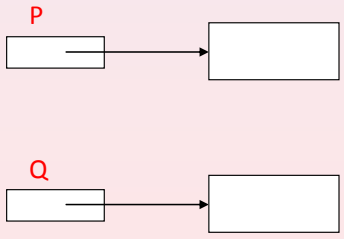
Produzione di garbage (spazzatura)

- Si verifica quando la memoria allocata dinamicamente risulta logicamente inaccessibile, e quindi sprecata, perché non esiste alcun riferimento ad essa.

  Esempio:

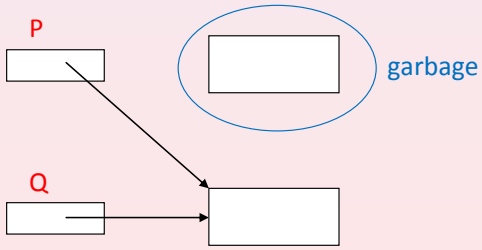
```
P=malloc(sizeof(TipoDato));
...
P=Q;
```

- In questo modo la cella puntata da P subito dopo l'assegnamento P=Q perde ogni possibilità di accesso (da cui il termine spazzatura).



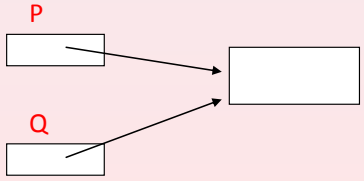
Produzione di garbage (spazzatura)

- Si verifica quando la memoria allocata dinamicamente risulta logicamente inaccessibile, e quindi sprecata, perché non esiste alcun riferimento ad essa.
  Esempio:
  ```
  P=malloc(sizeof(TipoDato));
  ...
  P=Q;
  ```
- In questo modo la cella puntata da `P` subito dopo l'assegnamento `P=Q` perde ogni possibilità di accesso (da cui il termine spazzatura).



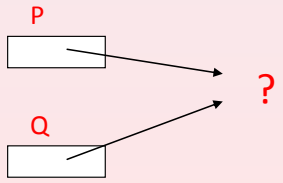
Riferimenti fluttuanti (dangling references)

- Simmetrico al problema precedente: consiste nel creare riferimenti fasulli a zone di memoria logicamente inesistenti.
  Esempio:
  ```
  P=Q;
  free(Q);
  ```
- L'operazione `free(Q)` provoca il rilascio della memoria allocata per la variabile cui `Q` punta
- `P` punta a una zona di memoria non più significativa (può essere riusata in futuro).
- `*P` comporterebbe l'accesso all'indirizzo fisico puntato da `P` e l'interpretazione del suo contenuto come un valore del tipo di `*P` con risultati imprevedibili.



Riferimenti fluttuanti (dangling references)

- Simmetrico al problema precedente: consiste nel creare riferimenti fasulli a zone di memoria logicamente inesistenti.
  Esempio:
  ```
  P=Q;
  free(Q);
  ```
- L'operazione `free(Q)` provoca il rilascio della memoria allocata per la variabile cui `Q` punta
- `P` punta a una zona di memoria non più significativa (può essere riusata in futuro).
- `*P` comporterebbe l'accesso all'indirizzo fisico puntato da `P` e l'interpretazione del suo contenuto come un valore del tipo di `*P` con risultati imprevedibili.

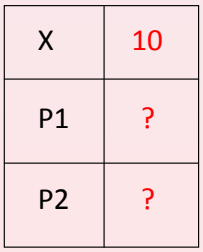


- Produzione di garbage e riferimenti fluttuanti hanno svantaggi simmetrici:
  - la prima comporta spreco di memoria
  - la seconda comporta risultati imprevedibili e scorretti.
- La seconda è più pericolosa della prima e in alcuni linguaggi non è prevista l'istruzione `free`.
- Viene lasciato al supporto del linguaggio l'onere di effettuare garbage collection ("raccolta rifiuti").

Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

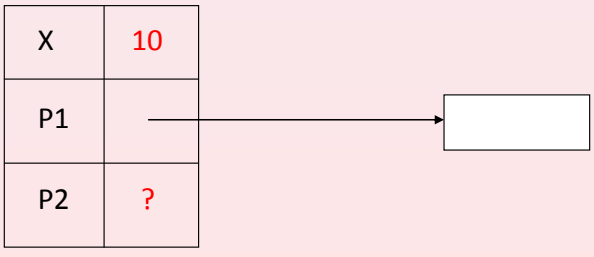
PILA HEAP



Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

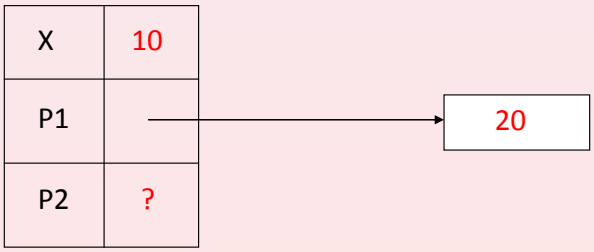
PILA HEAP



Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

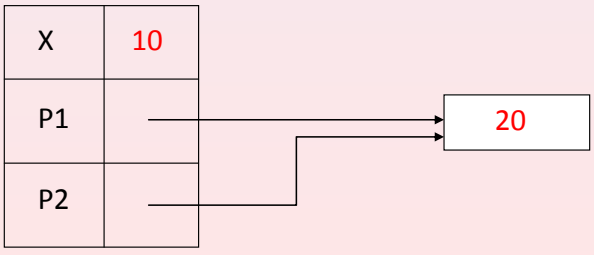
PILA HEAP



Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

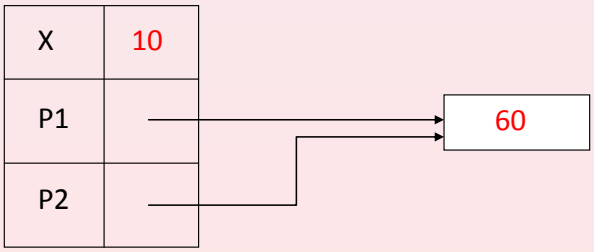
PILA HEAP



Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

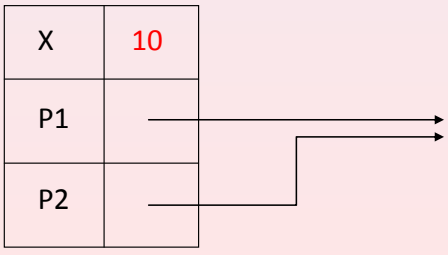
PILA HEAP



Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

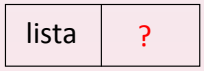
PILA HEAP



Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));       /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```

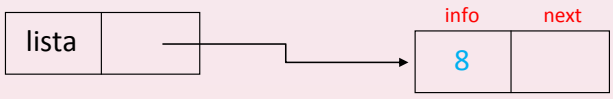
PILA HEAP



Creazione di una lista di tre interi fissati: (8, 3, 15)

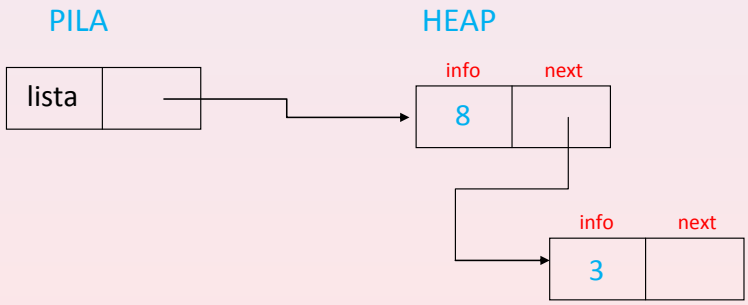
```
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));       /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```

PILA HEAP



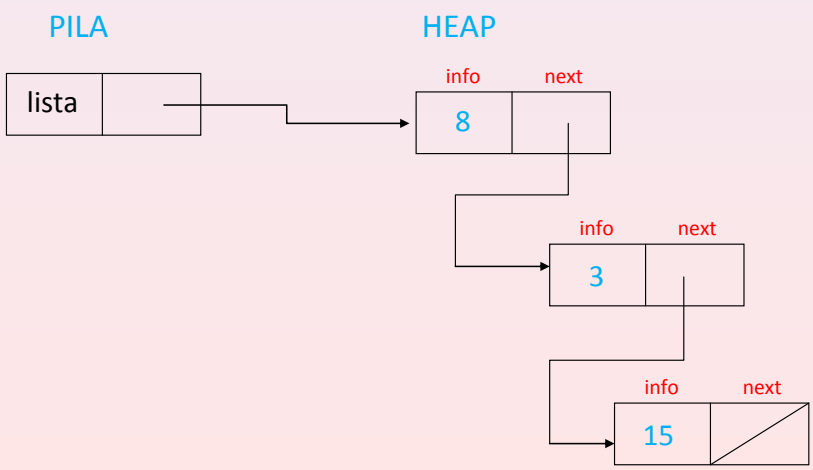
Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));       /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```



Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));       /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```

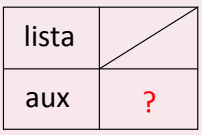


Osservazioni:

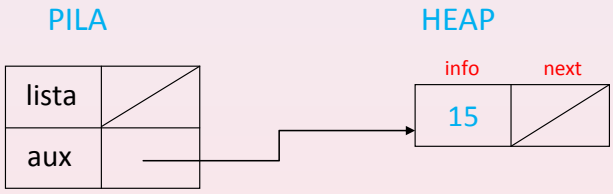
- `lista` è di tipo `ListaDiElementi`, quindi è un puntatore e non una struttura
- la zona di memoria per ogni elemento della lista (non per ogni variabile di tipo `ListaDiElementi`) deve essere allocata esplicitamente con `malloc`
- Esiste un modo più semplice di creare la lista di 3 elementi?
- Creiamo la lista a partire dal fondo!


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;    aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;     aux->next = lista;
lista = aux;
```

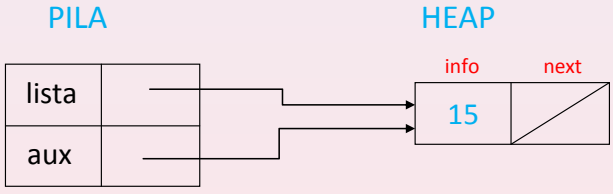
PILA HEAP



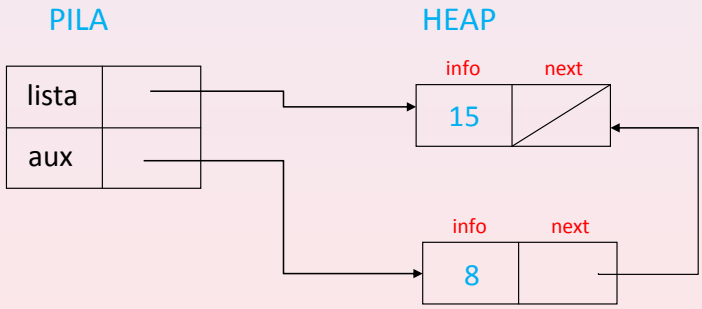
```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```



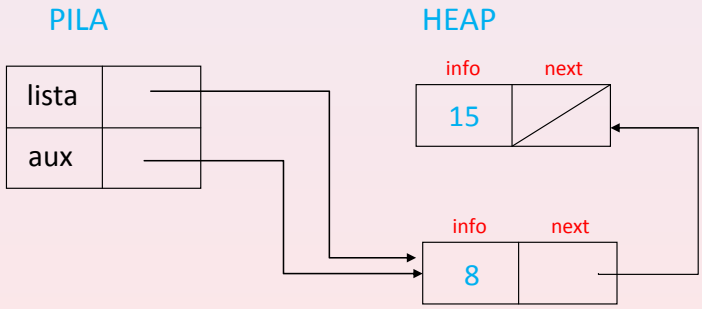

```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

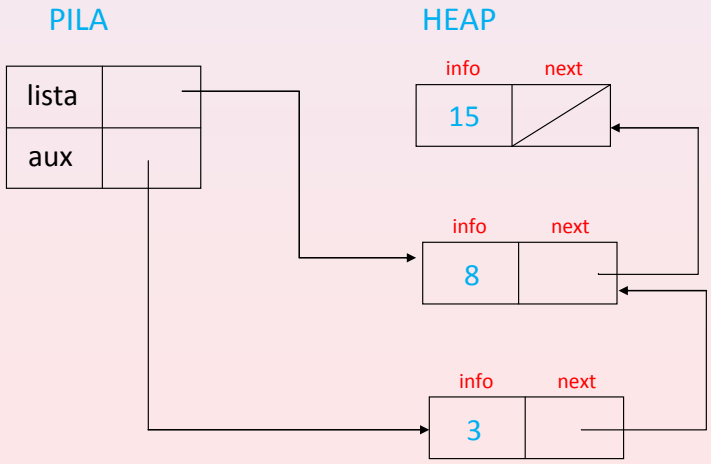
```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```




```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```



```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```




```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```