STRUTTURE DATI DINAMICHE

Esempi già visti di uso dei puntatori:
int a; int *p = &a, *q;
..
q=p; p=NULL; if (p==q) o (p==NULL) ...
(*p) =3;

- dinamiche:
  - a struttura nota a compile-time (tipo);
  - creazione e deallocazione gestite dal programmatore;
  - referenza solo tramite indirizzo (puntatore) perché non hanno nome.

Creazione e distruzione variabili dinamiche

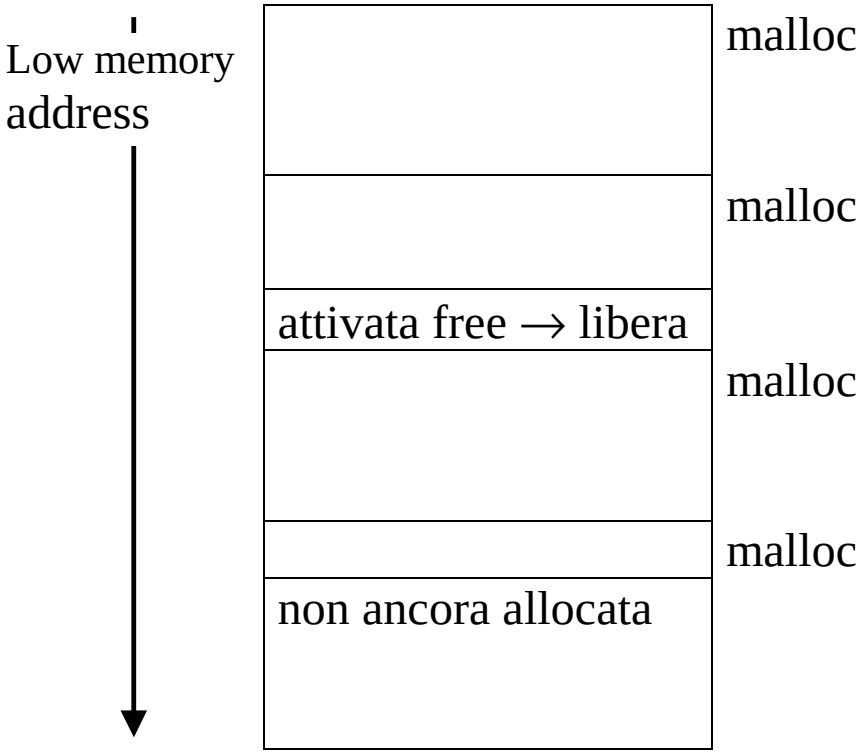
- import dal modulo di libreria #include <stdlib.h>

- void malloc(int num);

- alloca num caratteri e ritorna il puntatore (NULL ≡ problemi);
- il risultato void e il casting:
  es. p = (rec *) malloc(sizeof(rec));
- l'allocazione avviene nell'area di Heap - heap overflow

void free(void *pointer);
  es. free(p);
- dealloca spazio occupato dalla variabile
- pointer non più utile ma con indirizzo

Allocazione e deallocazione nell'area di HEAP



Attenzione

- allocazione a blocchi di dimensione variabile

⇓

Frammentazione interna se allocata dimensione superiore alla necessità

- non esiste deframmentazione

⇓

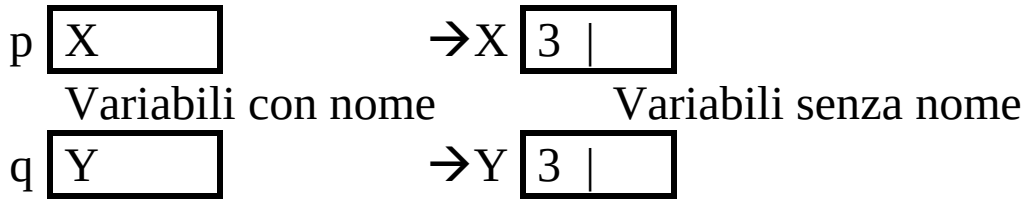
frammentazione esterna: spazio libero grande, ma blocchi liberi troppo piccoli

Allocazione vettore di dimensione dinamica

```
int main()
{int quanti, i, *p;
  scanf("%d",&quanti);
  p=malloc(quanti*sizeof(int));          if (p==NULL) error
….
  for (i=0; i<quanti;i++) p[i]=i; oppure *(p+i)=i;

  for (i=0; i<quanti;i++) printf(" %d", p[i]); oppure  *(p+i));
```

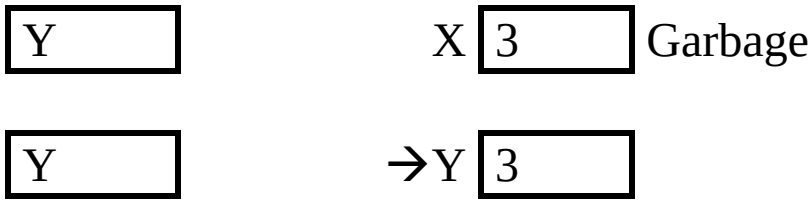
Gestione memoria e problemi

```
   p = (rec *) malloc(sizeof(rec));          if (p==NULL) error
   q = (rec *) malloc(sizeof(rec));          if (q==NULL) error
   (*p).a =3; p->a=3;
   (*q).a =3; q->a=3;
```



a) produzione diretta di garbage: variabile dinamica irraggiungibile.

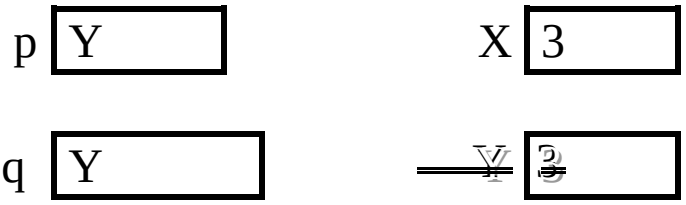
```
p=q;
```



- non esiste garbage collector

b) generazione dangling reference: puntatore con indirizzo non valido

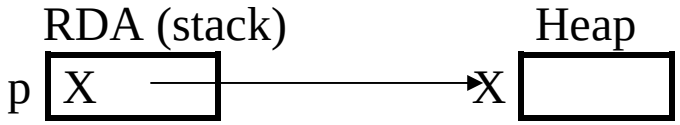
free(q);



p->a = 2;
q->a = 2; errore in esecuzione

c) produzione indiretta di garbage:

```
void P()
{ int *p;   p = (int *) malloc(sizeof(int)); }

void main() {P();}
```



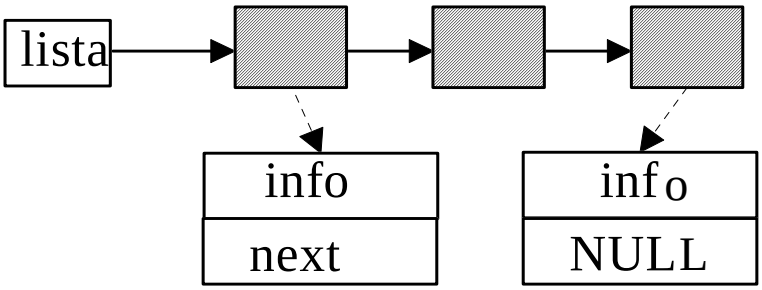
d) produzione indiretta di dangling reference:

```
void P()
{ int n; p=&n;}

void main() {int *p; P();}



}
```

Strutture dati concatenate – tipo ricorsivo

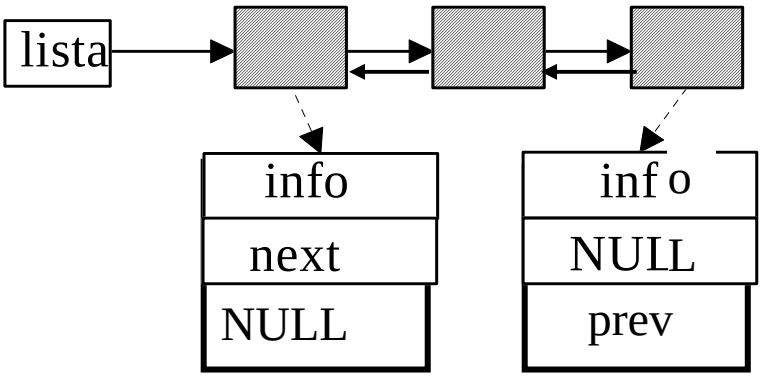
- insieme di elementi di tipo omogeneo
- collegamento tramite puntatori
- almeno un “handle” per accedere alla struttura

Lista monodirezionale

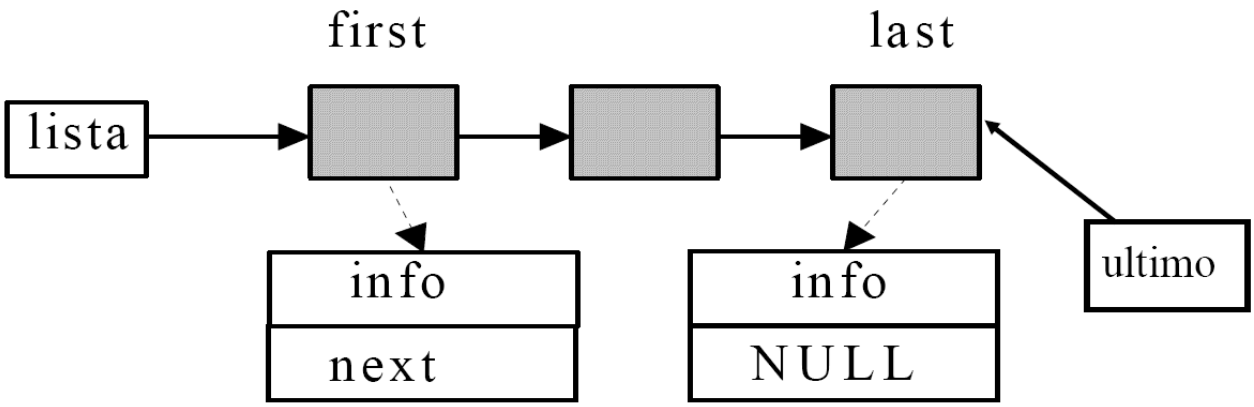


definizioni
struct el {int info; struct el *next;};
struct el *lista=NULL;

Lista bidirezionale
definizioni
struct el {int info; struct el *prev; struct el *next;};
struct el *lista=NULL;

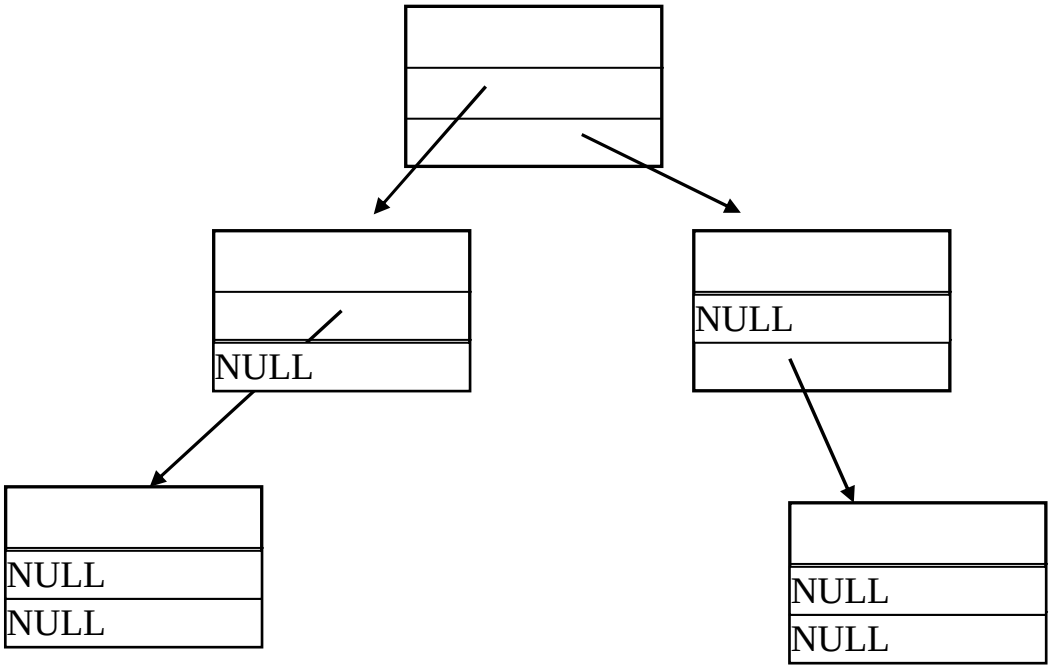


Lista monodirezionale con doppio handle

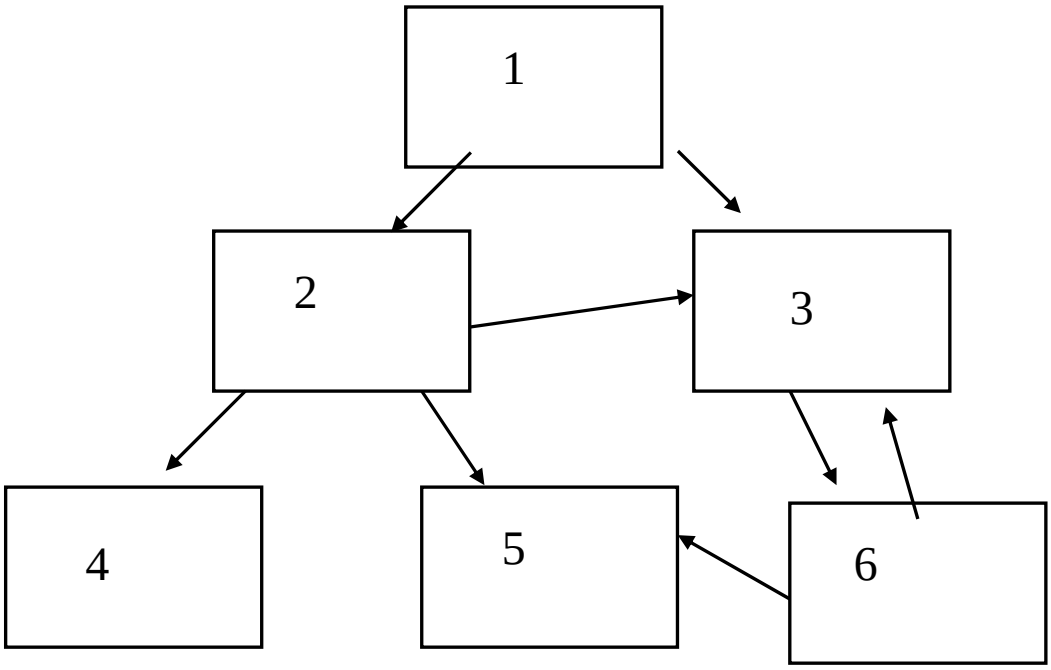


Albero binario

```
struct el  {int info;
            struct el *sinistro;
            struct el *destro;};
struct el *lista=NULL;
```

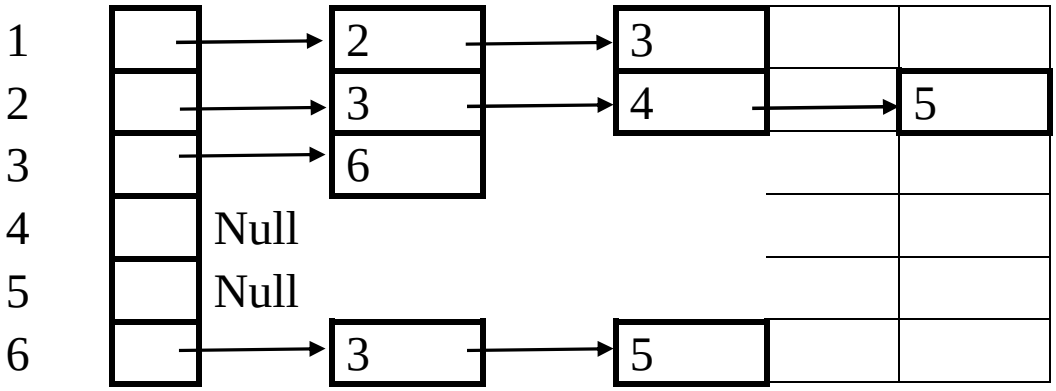


Grafo



Puntatori negli elementi – quanti?

Liste dei successori

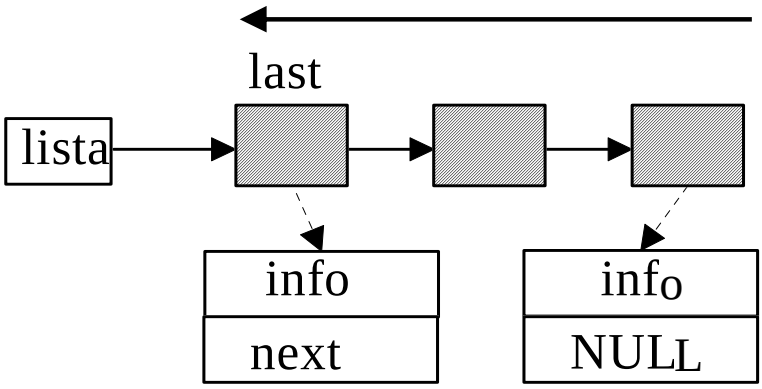


Quale scegliere in base all'applicazione?

- Gestione di una sequenza di valori (vettore)
- Gestione coda
- Gestione pila
- Indice di accesso ai file

……..

La gestione di una pila/stack con una lista monodirezionale




```
struct el   {int info;  struct el *next;} ;
struct el *lista=NULL;    //lista

struct el *elemento;        //singolo elemento
```

Creazione di un elemento nuovo
e caricamento contenuto

-

```
struct el *creael ()
 {  struct el *temp;    temp= malloc(sizeof(struct el));
    if (temp != NULL)
      {printf("\nintroduci campo info");
       scanf("%d", &(temp->info));  temp->next=NULL;
       }
    return(temp);
  }
void main(){lista=creael();}
```

Variante con dati ricevuti come parametri:

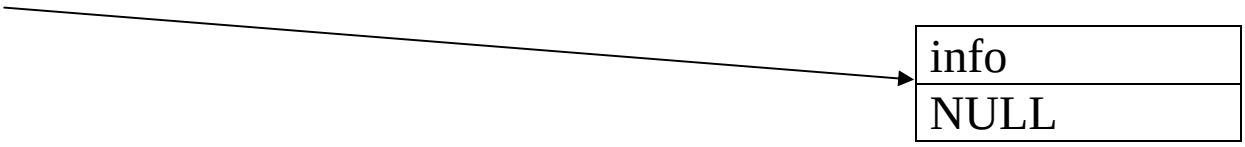
struct elemento *creael (int info){.....}

Situazione dopo lista=…

area globale stack heap

temp

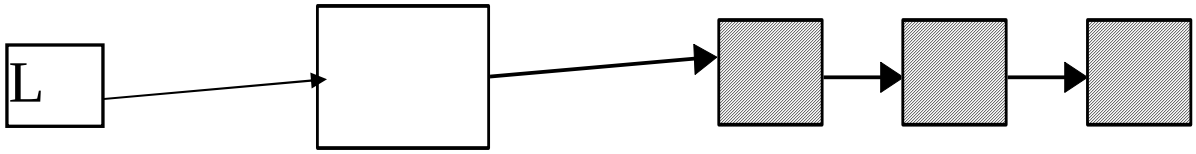
lista



Inserimento elemento creato nella lista

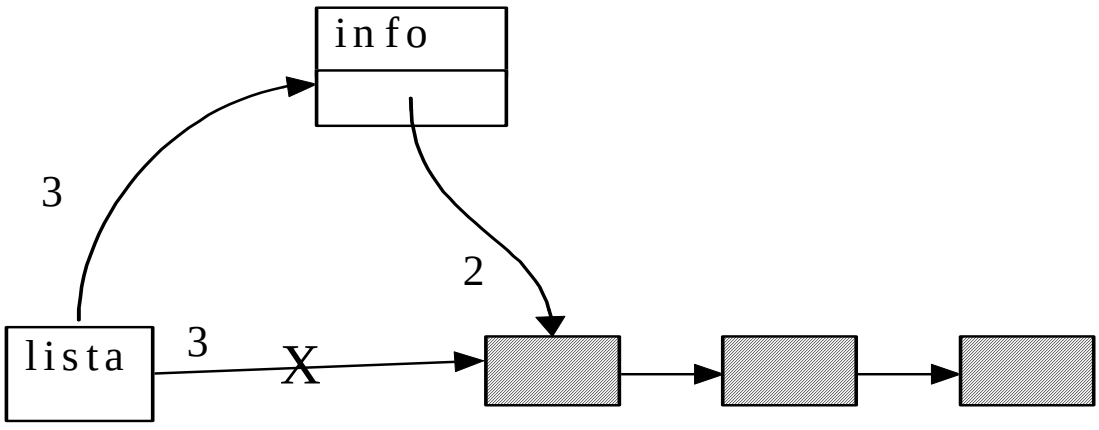
Push (lista ↓↑ elemento ↓) non consideriamo elemento=NULL

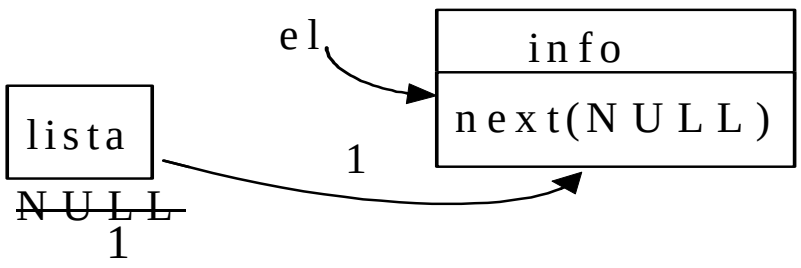
1) struct el *push (struct el *L, struct el *e)
2) void push (struct el **L, struct el *e)




```
1) struct el *push (struct el *L, struct el *e)
        {if (L ==NULL)  return(e); /*1*/
         else
            { e->next = L; /*2*/        L=e; /*3*/}
         return(L);      /*4*/
        }
```

Invocazione:
1. elemento= creael();
2. if (elemento!=NULL) lista= push(lista, elemento);





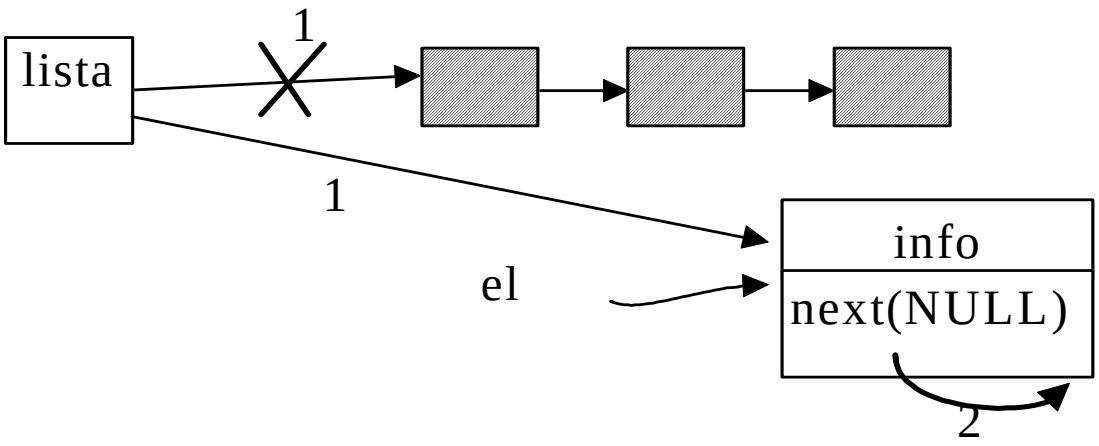
Attenzione

Passaggio parametric

void push (struct el *L, struct el *e)

Rispettare l'ordine delle operazioni

```
  struct el *push (struct el *e, struct el *L)
 {if (L==NULL) return(e);
   L = e;                /* 1 */
   e->next = L;          /* 2 */  }
   return L;
}
```



Estrazione elemento

Pop (lista ↓ ↑ elemento ↑) not∃ ⇒ return NULL

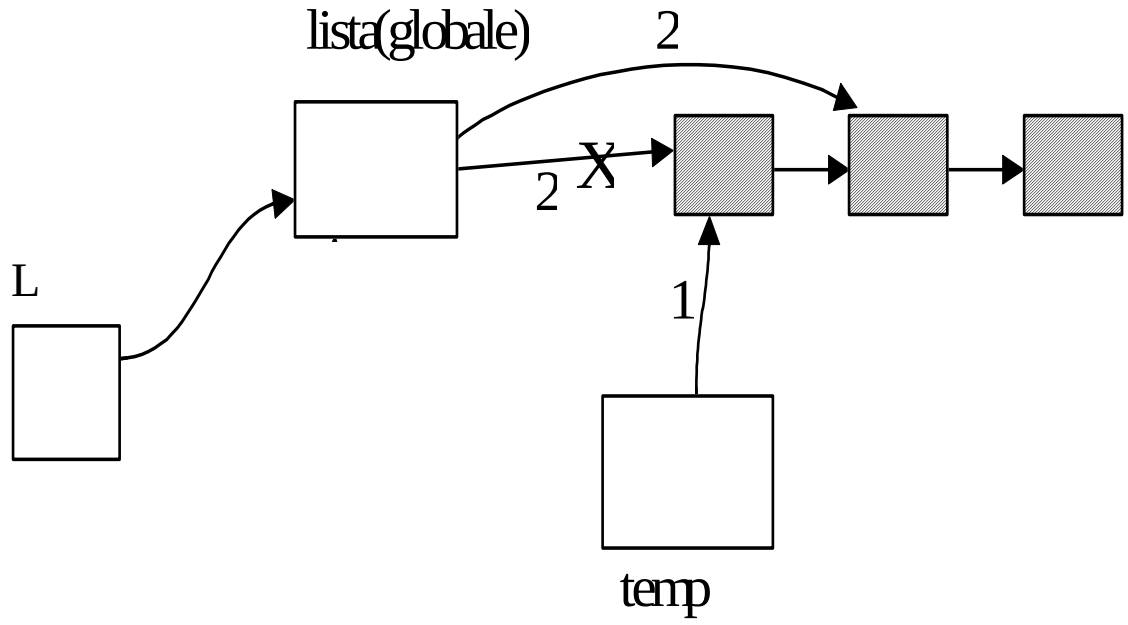
1. void pop (struct el *L, struct el *e) errata

2. struct el *pop (struct el **e, struct el *L)
3. struct el *pop (struct el **L)

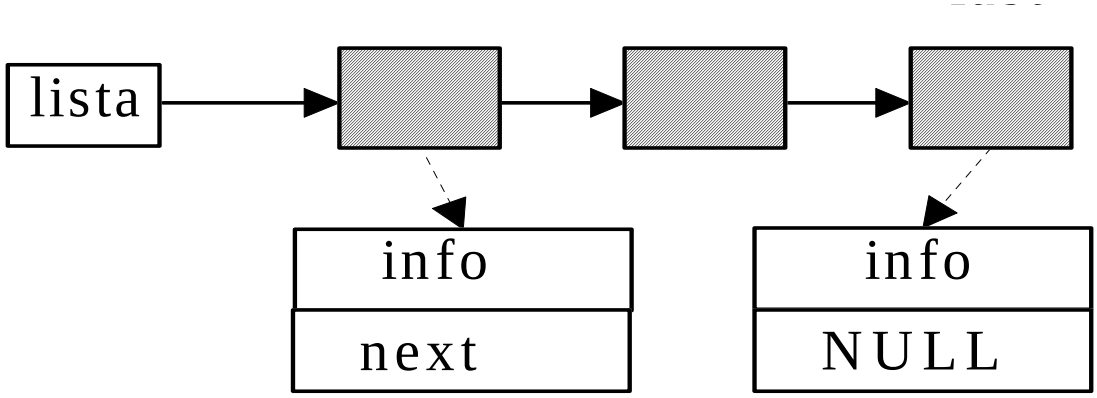
3. struct el *pop (struct el **L)

```
{struct el *temp;
  if (*L ==NULL)  return(NULL);
  else
      { temp = *L;                    /*non usare L x scansione   1*/
        *L = (*L)->next;  /*2*/
       }
  return(temp);
 }
```

Invocazione
elemento= pop(&lista); if (elemento!=NULL)…..



La gestione di una sequenza con una lista monodirezionale



Criterio di inserimento dipende dall'applicazione:

- Libero: inserisco in testa
- Vincolato: ordinamento, …

Scansione totale della sequenza (stampa) – vale anche per pila

```
void visualizza(struct el *L)
  { while (L !=NULL)                    //L per la scansione?
       { printf("\ninfo: %d", L ->info);
         L = L ->next;
        }
   }
```

Invocazione: visualizza (lista);

Ricordare: la lista può essere vuota?

In alternativa?

```
void visualizza(struct el *L)
{ if (L!=NULL) {printf("\ninfo:%d",L->info);
                        visualizza(L->next);
                    }
}
```

Attenzione se lista può essere vuota

```
void visualizza (struct el *L)
  {do
      {printf("%d ", L->info); L=L->next;}
    while (L ->next != NULL);
  }
```

Ricerca nella sequenza
(es. ricerca studente con matricola = XX)

Ricerca (lista ↓ valore ↓ el ↑)

```
   struct el *ricerca (struct el *L, int valore)
      {int trovato=0;
        if (L==NULL) return NULL;            <- lista vuota
(else?)while ((L !=NULL)&& !trovato)       //L per la scansione?
       { if (L->info==valore) trovato=1;
         else L = L ->next;
        }
       if trovato return L;
       else return NULL;                          <- non trovato in lista
  }                                                  con elementi
```

Invocazione elemento = ricerca(lista, XX);

Forzatura ritorno

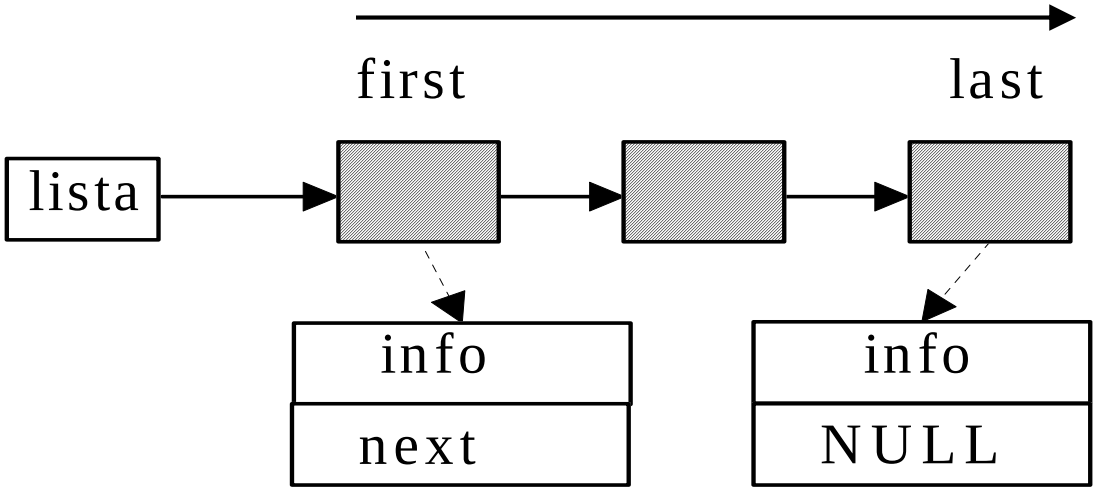
```
struct el *ricerca (struct el *L, int valore)
  {if (L==NULL) return NULL;
    while (L !=NULL)
  { if (L->info==valore) return L
    L = L ->next;
   }
}
```

Attenzione

```
struct el  *Ricerca (struct el *L, int valore)
{ while (L ->info != valore)L = L–>next;
  return L;
}
```

Se la lista fosse ordinata sulla matricola e volessimo inserire un nuovo studente?

La gestione di una coda con una lista monodirezionale



Inserimento elemento in coda (estrazione come pila)

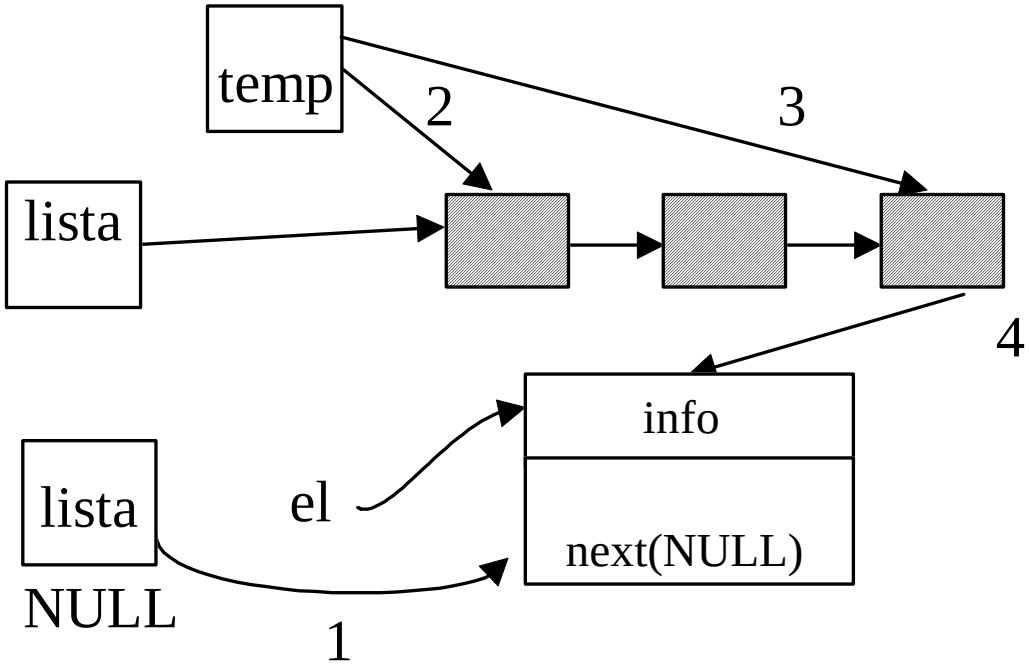
Insert (lista ↓ ↑ el ↓)

```
struct el *insert (struct el *e, struct el *L)
 {struct el *temp;
  if (L == NULL)  return(e);    /*1*/
   else {temp = L;                          /*2*/
          while (temp->next !=NULL) temp=temp->next;  /*3*/
          temp->next=e;                    /*4*/
         }
   return(L)
 }
```

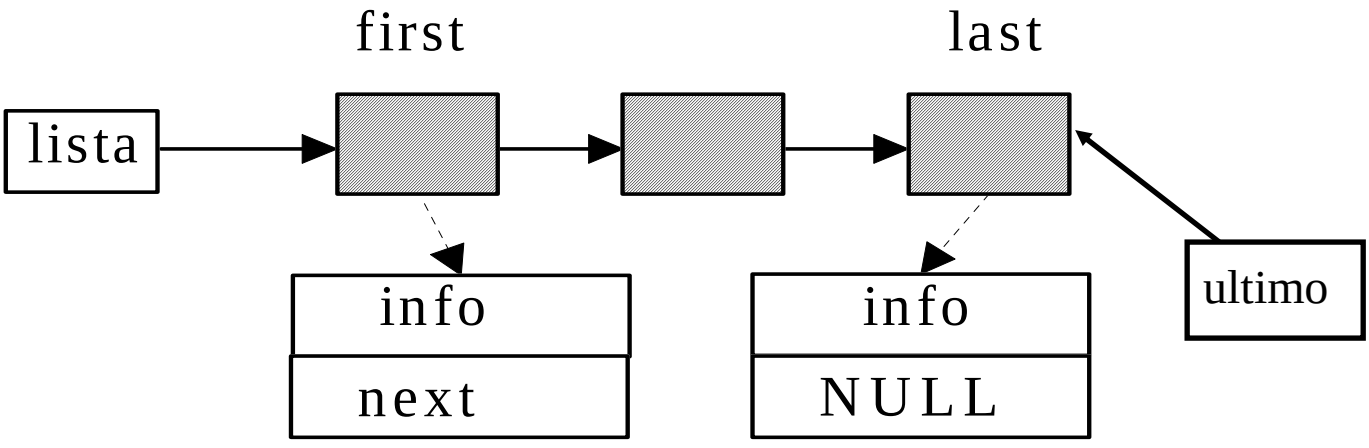
- Perchè lista deve essere restituita?
- Perché si usa temp?
- Lista vuota gestita?

Invocazione lista=insert(elemento,lista)

Situazione



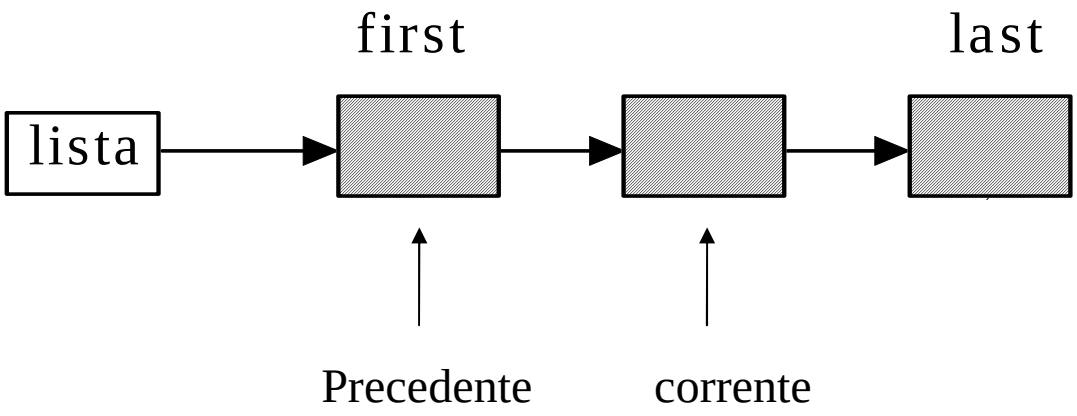
Come migliorare gli inserimenti?



Cancellazione di un elemento (info =xx)?

La base è la scansione di ricerca

- Lista vuota



E in questo caso