Politecnico
di Milano

Strutture dati dinamiche

Strutture dati statiche e dinamiche

- In C le dimensioni di ogni dato sono note prima dell'esecuzione
- Sarebbe utile avere contenitori la cui dimensione varia a seconda delle necessità
  - La soluzione in C:
    - Ogni variabile ha sempre una dimensione prefissata…
    - Ma … durante l'esecuzione di un programma è possibile creare dinamicamente nuove variabili

I meccanismi di allocazione e cancellazione dinamica della memoria

- Vengono offerti da due funzioni di libreria
  - malloc e free

```
#include <stdlib.h> /* necessario per usare malloc e free */
#include <stdio.h>
typedef int *tipoPtr;
void main()
{
   tipoPtr p;
   p = malloc(sizeof(int)); /*crea una variab. anonima int */
   *p = 24;
   printf("%d", *p);
   free(p); /* libera lo spazio di memoria a cui p punta */
}
```

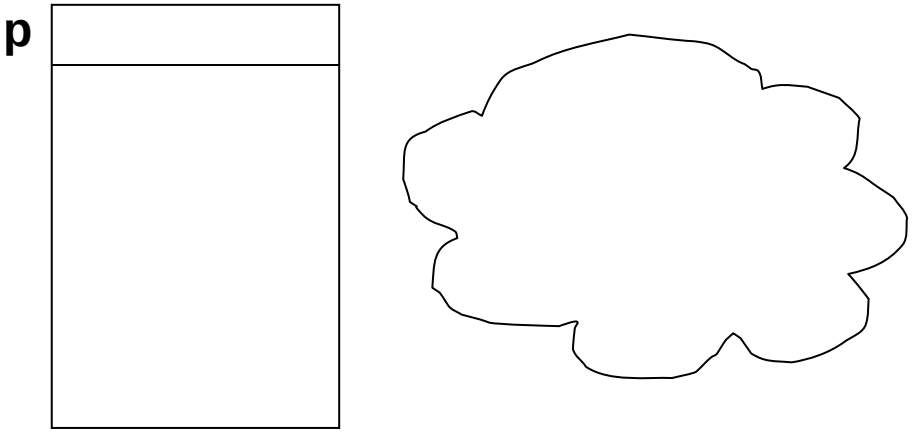
Restituisce la dimensione in byte di una variabile o di elementi di un certo tipo

Modello di esecuzione (1)

- Ad ogni programma in esecuzione vengono associate due aree di memoria con caratteristiche diverse
- Lo stack
  - Contiene le aree di memoria associate alle variabili “normali” organizzate nei rispettivi record di attivazione
  - Viene gestito con politica LIFO
- Lo *heap* (mucchio)
  - Contiene le aree di memoria associate alle variabili allocate dinamicamente

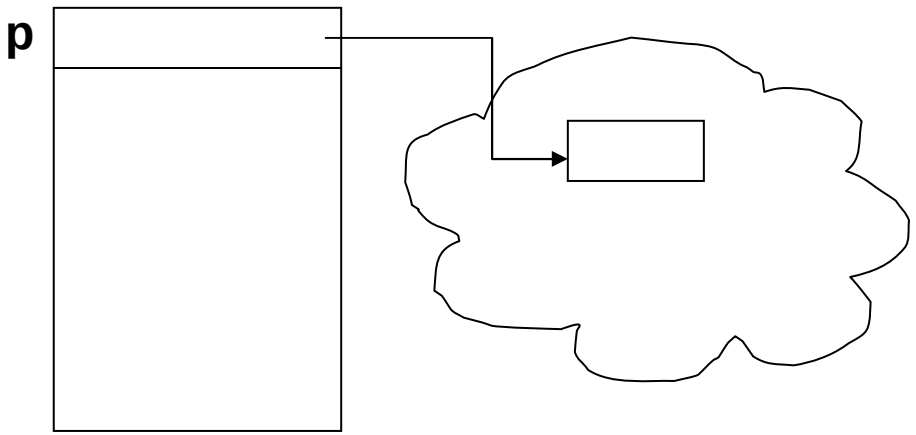
Modello di esecuzione (2)

```
#include <stdlib.h>
#include <stdio.h>
typedef int *tipoPtr;
void main()
{
   tipoPtr p;
   p = malloc(sizeof(int));
   *p = 24;
   printf("%d", *p);
   free(p);
}
```



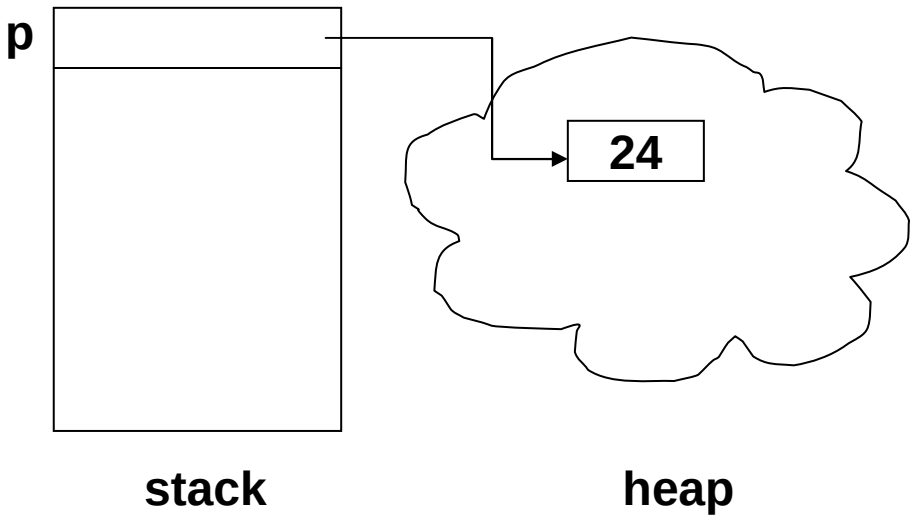
Modello di esecuzione (2)

```
#include <stdlib.h>
#include <stdio.h>
typedef int *tipoPtr;
void main()
{
   tipoPtr p;
   p = malloc(sizeof(int));
   *p = 24;
   printf("%d", *p);
   free(p);
}
```



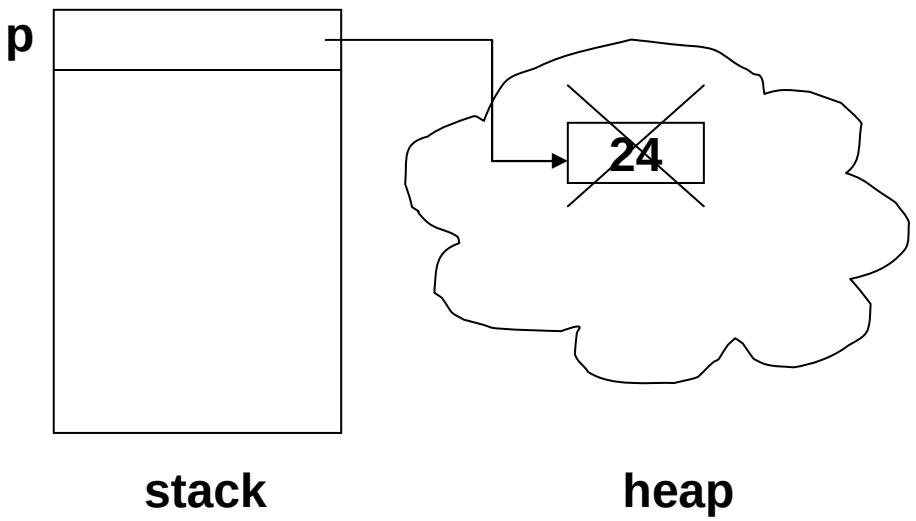
Modello di esecuzione (2)

```
#include <stdlib.h>
#include <stdio.h>
typedef int *tipoPtr;
void main()
{
   tipoPtr p;
   p = malloc(sizeof(int));
   *p = 24;
   printf("%d", *p);
   free(p);
}
```



Modello di esecuzione (2)

```
#include <stdlib.h>
#include <stdio.h>
typedef int *tipoPtr;
void main()
{
   tipoPtr p;
   p = malloc(sizeof(int));
   *p = 24;
   printf("%d", *p);
   free(p);
}
```



Aspetti critici

- Dangling pointer

```
p = malloc(sizeof(int));
free(p);
*p = 12;
```

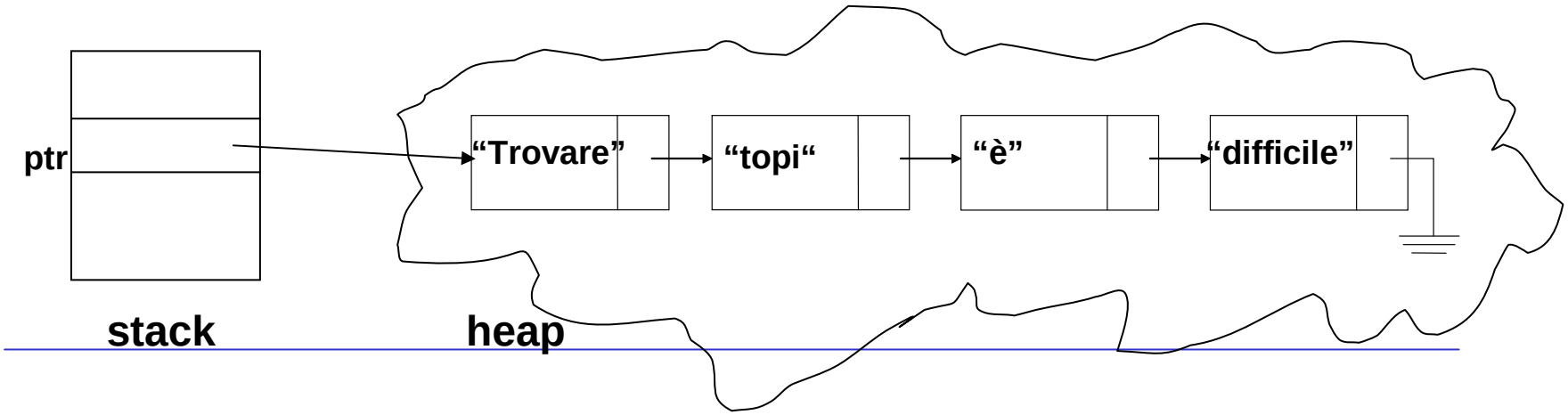
  - Il valore 12 viene assegnato ad un'area di memoria non più associata alla variabile anonima creata con la malloc!
- Garbage
  - La memoria allocata dinamicamente rimane nello heap durante tutta l'esecuzione del programma a meno che non si deallochi esplicitamente
  - Esempio di programma che spreca memoria

```
p = malloc(sizeof(int));
*p = 24;
p = malloc(sizeof(int));
*p = 10;
```

  - Vengono create due aree di memoria dello heap, ma solo la seconda sarà raggiungibile tramite p. La seconda costituisce spazzatura (garbage)

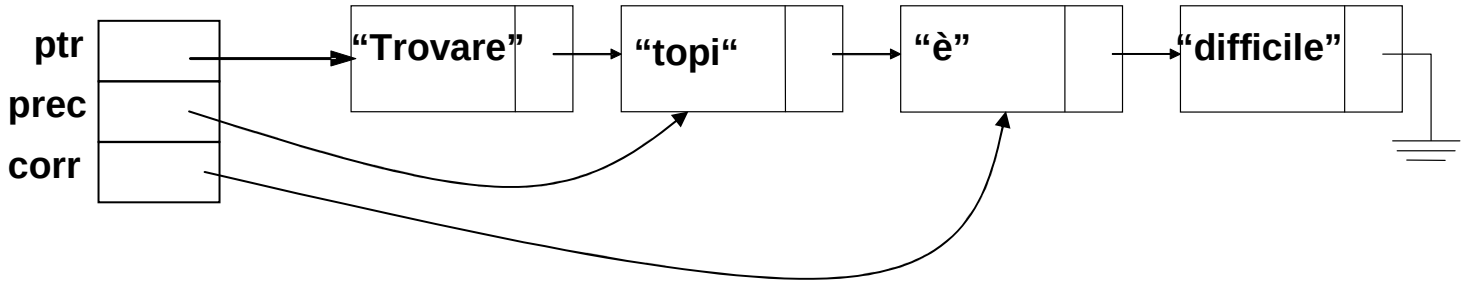
Liste a puntatori (1)

- Sono costruite collegando elementi costituiti da due parti:
  - Il contenuto informativo
    - un intero, i dati di uno studente, un numero complesso, una stringa, ….
  - Un puntatore all'elemento seguente
- Ciascun elemento viene creato con l'allocazione dinamica della memoria



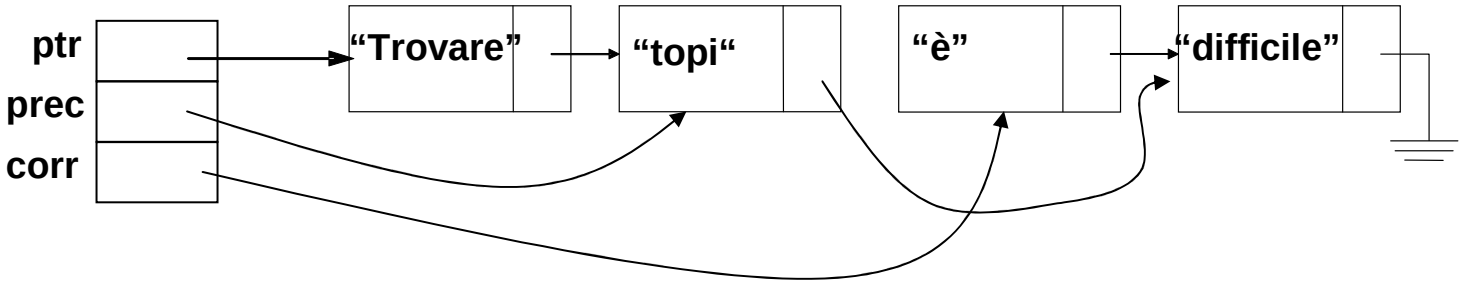
Liste a puntatori (2)

- Vantaggi
  - E` possibile aggiungere o eliminare elementi al bisogno ed in modo semplice
  - Cancellazione



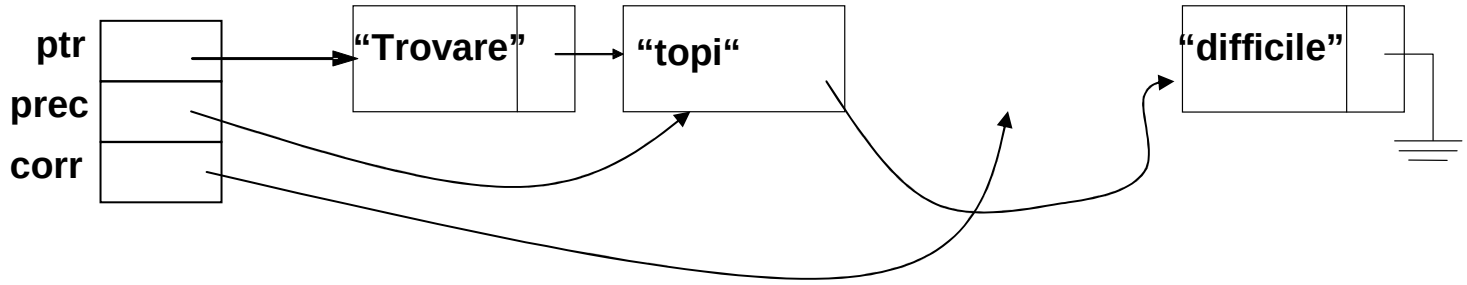
Liste a puntatori (2)

- Vantaggi
  - E` possibile aggiungere o eliminare elementi al bisogno ed in modo semplice
  - Cancellazione



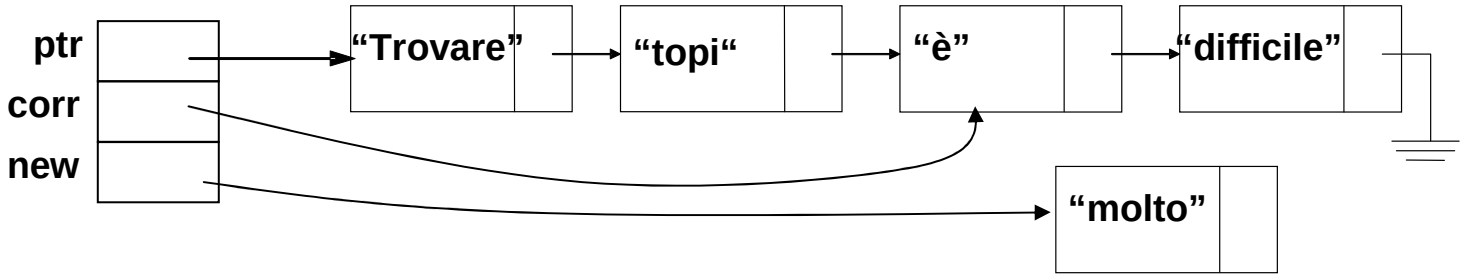
Liste a puntatori (2)

- Vantaggi
  - E` possibile aggiungere o eliminare elementi al bisogno ed in modo semplice
  - Cancellazione



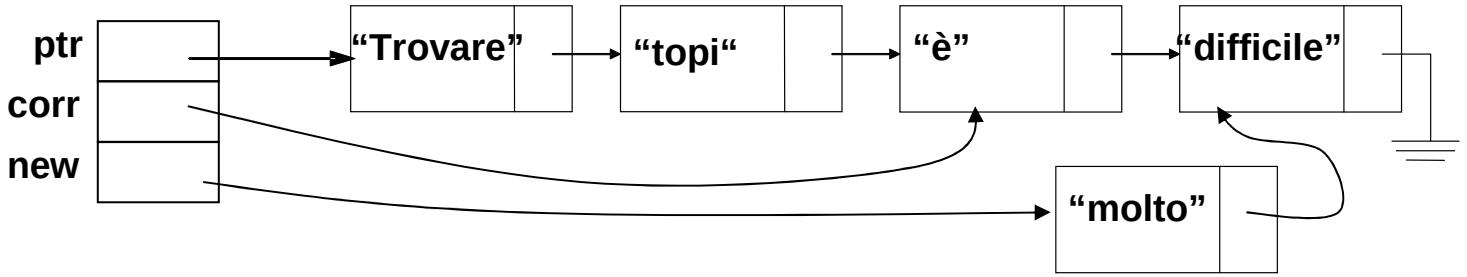
Liste a puntatori (3)

- Vantaggi
  - E` possibile aggiungere o eliminare elementi al bisogno ed in modo semplice
  - Inserimento



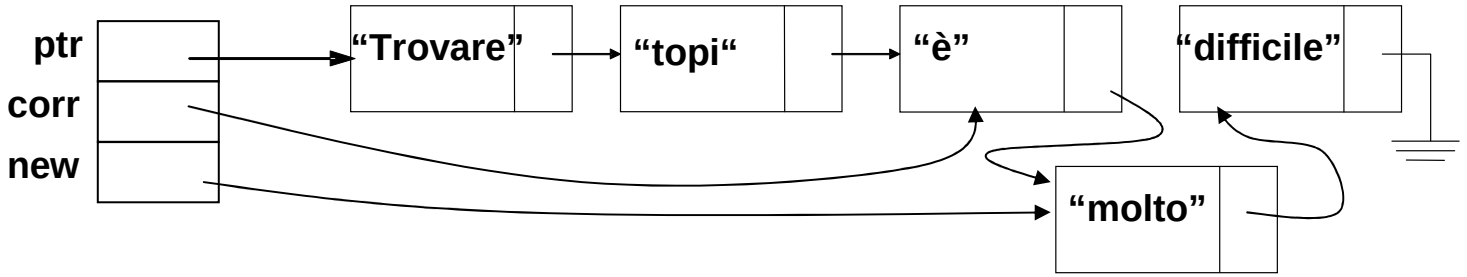
Liste a puntatori (3)

- Vantaggi
  - E` possibile aggiungere o eliminare elementi al bisogno ed in modo semplice
  - Inserimento



Liste a puntatori (3)

- Vantaggi
  - E` possibile aggiungere o eliminare elementi al bisogno ed in modo semplice
  - Inserimento



Tipi necessari per costruire la lista


```
typedef struct {
   int a;
   int b;
} TipoDato; /* TipoDato può avere qualsiasi struttura */

typedef struct El {
   TipoDato info; /*si assume che tipoDato sia stato definito
                    precedentemente */
   struct El *next; /* puntatore all'elemento seguente */
} ElemLista;
typedef ElemLista *Lista;

typedef enum {false, true} boolean; /* usato dopo */
```

Operazioni fondamentali per la gestione delle liste


```
/* creazione di una lista vuota */
Lista crea();

/* inserimento di un elemento nella lista */
void inserisciInTesta(Lista *l, TipoDato dato);
void inserisciInCoda(Lista *l, TipoDato dato);
void inserisciInOrdine(Lista *l, TipoDato dato);

/* controllo lista vuota */
boolean listaVuota(Lista l);

/* controllo esistenza di un elemento nella lista */
boolean esiste(Lista l, TipoDato dato);
```

Operazioni fondamentali per la gestione delle liste


```
/* cancellazione di un elemento dalla lista */
void cancella(Lista *l, TipoDato dato);

/* restituisce la sottolista costituita da tutti gli
   elementi di quella di partenza ad esclusione del
   primo */
Lista coda(Lista l);

/* stampa sullo schermo il contenuto della lista */
void stampaLista(Lista l);
```

Operazioni di servizio relative a TipoDato

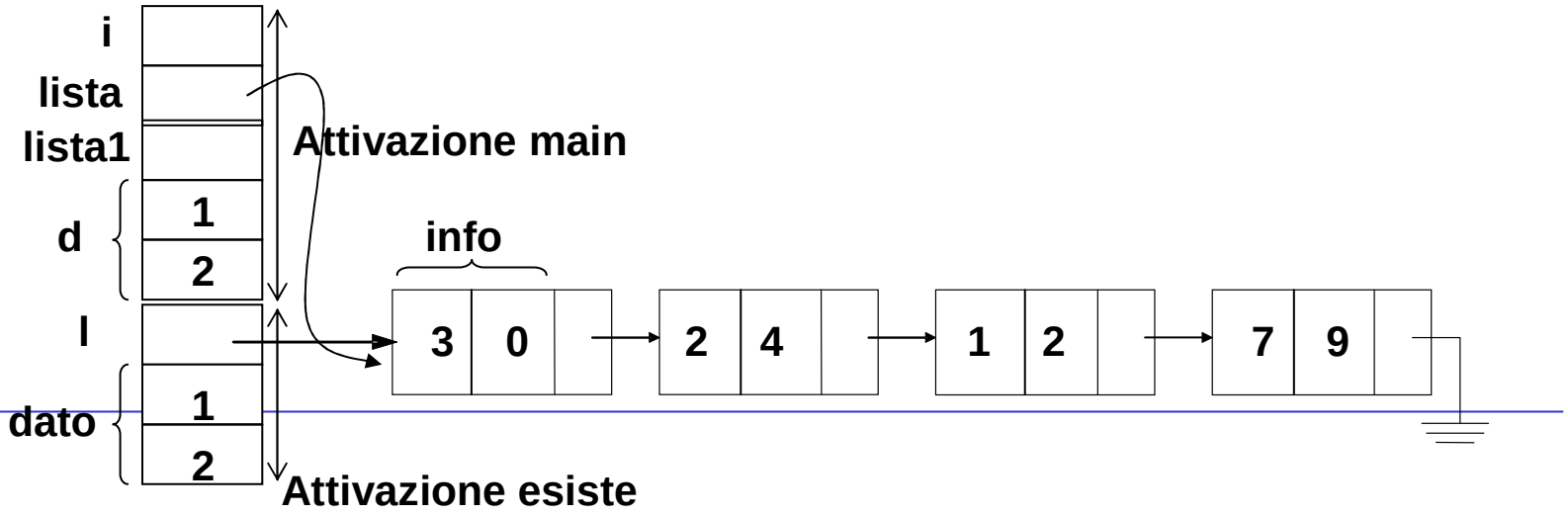

```
TipoDato acquisisci()
{
   TipoDato x;
   scanf("%d%d", &(x.a), &(x.b));
   return x;
}

boolean uguale(TipoDato d1, TipoDato d2)
{
   if((d1.a == d2.a)&&(d1.b == d2.b)) return true;
   else return false;
}

void stampa(TipoDato d)
{  printf("%d, %d\n", d.a, d.b); }
```

Un main di esempio


```
void main()
{
  int i;
  Lista lista, lista1;
  TipoDato d;

  lista = crea();
  for(i=0;i<5;i++)
  {
    d = acquisisci();
    inserisciInTesta(&lista, d);
     stampaLista(lista);
  }
  d = acquisisci();
  if(esiste(lista, d) == vero)
  cancella(&lista, d);
  stampaLista(lista);
  lista1 = coda(lista);
  stampaLista(lista1);
}
```

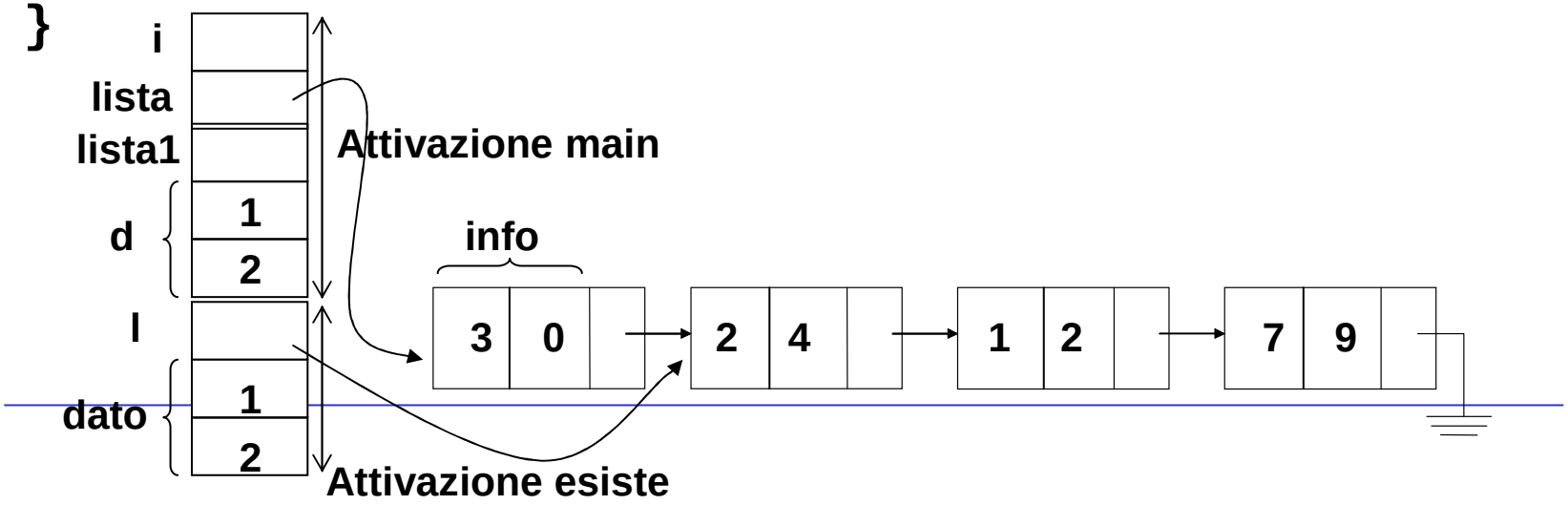
Implementazione di crea ed esiste

```
Lista crea() {  return NULL; }

boolean esiste(Lista l, TipoDato dato)
{
  while((l!=NULL) && uguale(l->info,dato)==false)
    l = l->next;
  if(l!=NULL) return true;
  else return false;
}
```



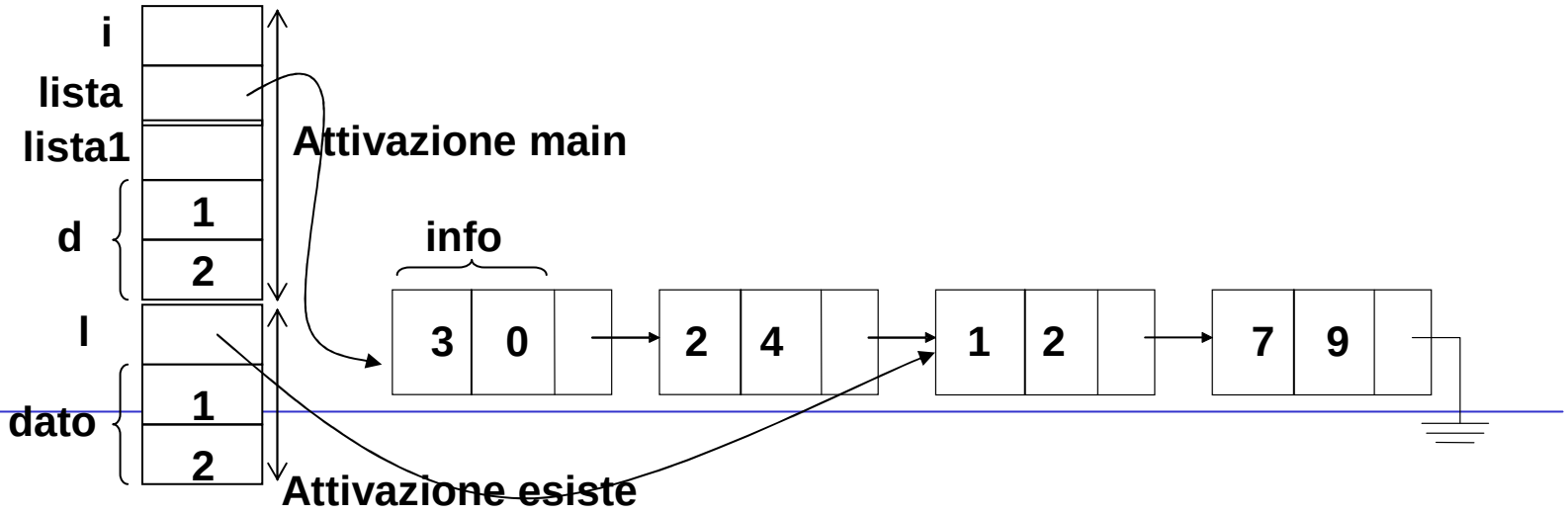
Implementazione di crea ed esiste

```
Lista crea() {  return NULL; }

boolean esiste(Lista l, TipoDato dato)
{
  while((l!=NULL) && uguale(l->info,dato)==false)
    l = l->next;
  if(l!=NULL) return true;
  else return false;
}
```



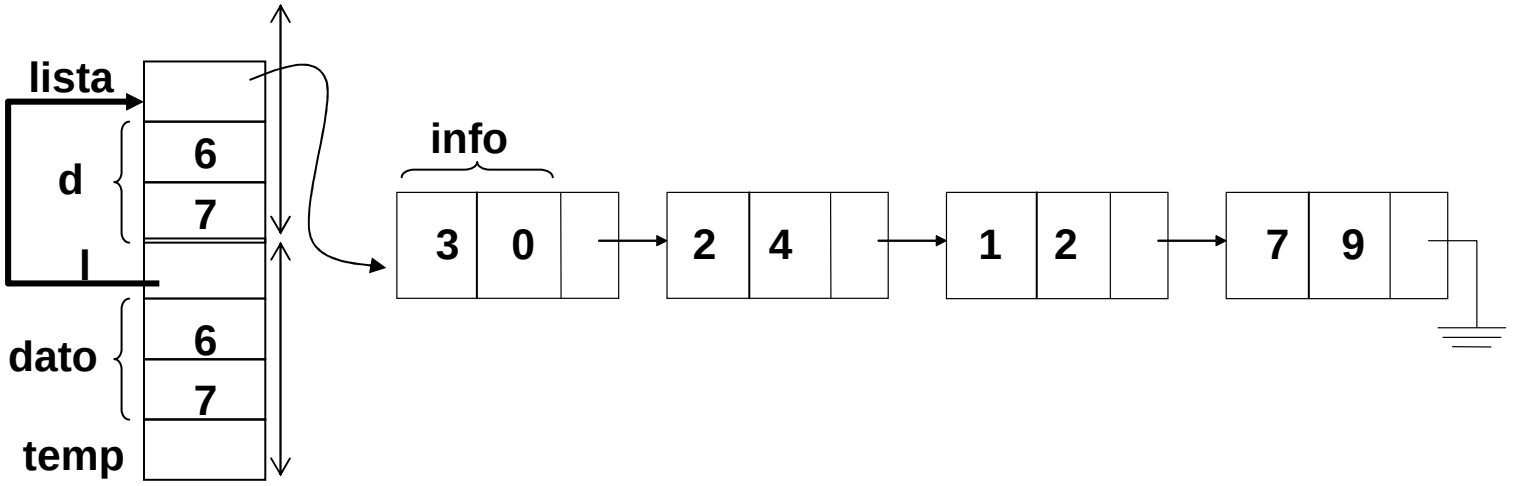
Implementazione di crea ed esiste

```
Lista crea() {  return NULL; }

boolean esiste(Lista l, TipoDato dato)
{
  while((l!=NULL) && uguale(l->info,dato)==false)
    l = l->next;
  if(l!=NULL) return true;
  else return false;
}
```



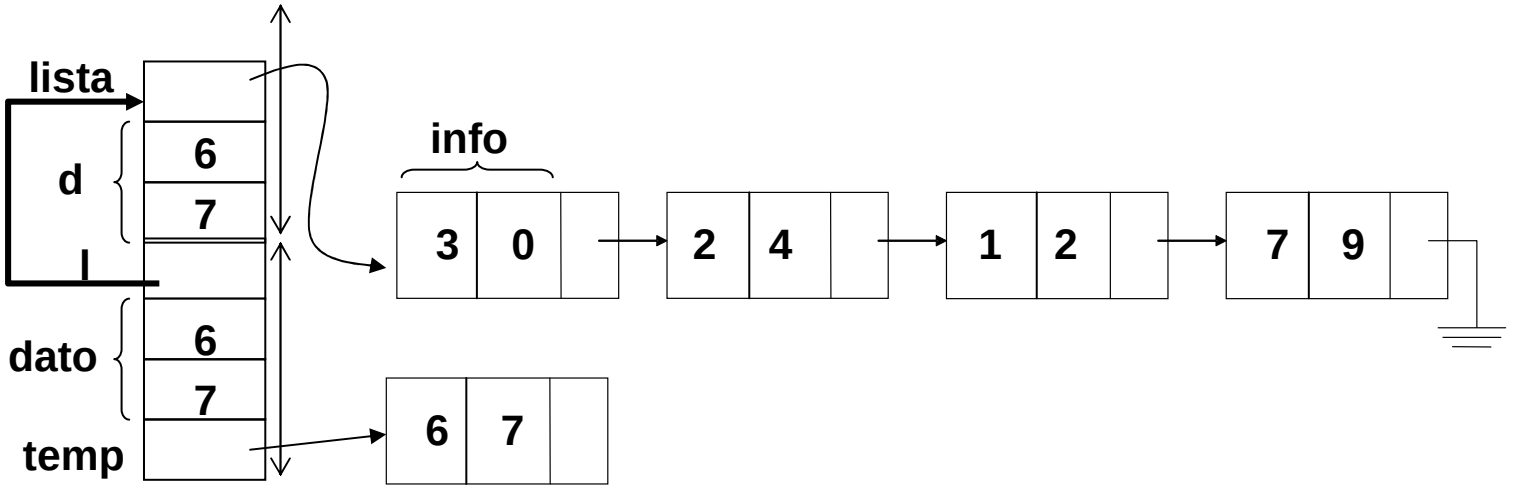
Implementazione di inserisciInTesta

```
void inserisciInTesta(Lista *l, TipoDato dato)
{
  Lista temp;
  temp = malloc (sizeof(ElemLista));
  temp->info = dato;
  temp->next = *l;
  *l = temp;
}
```



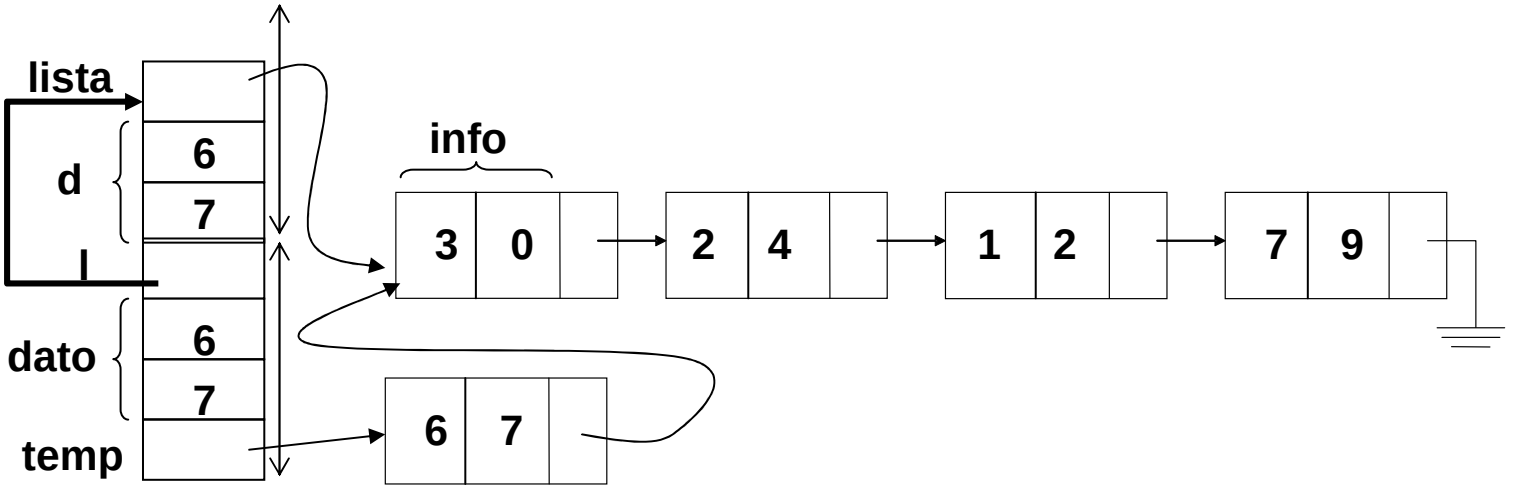
Implementazione di inserisciInTesta

```
void inserisciInTesta(Lista *l, TipoDato dato)
{
  Lista temp;
  temp = malloc (sizeof(ElemLista));
  temp->info = dato;
  temp->next = *l;
  *l = temp;
}
```



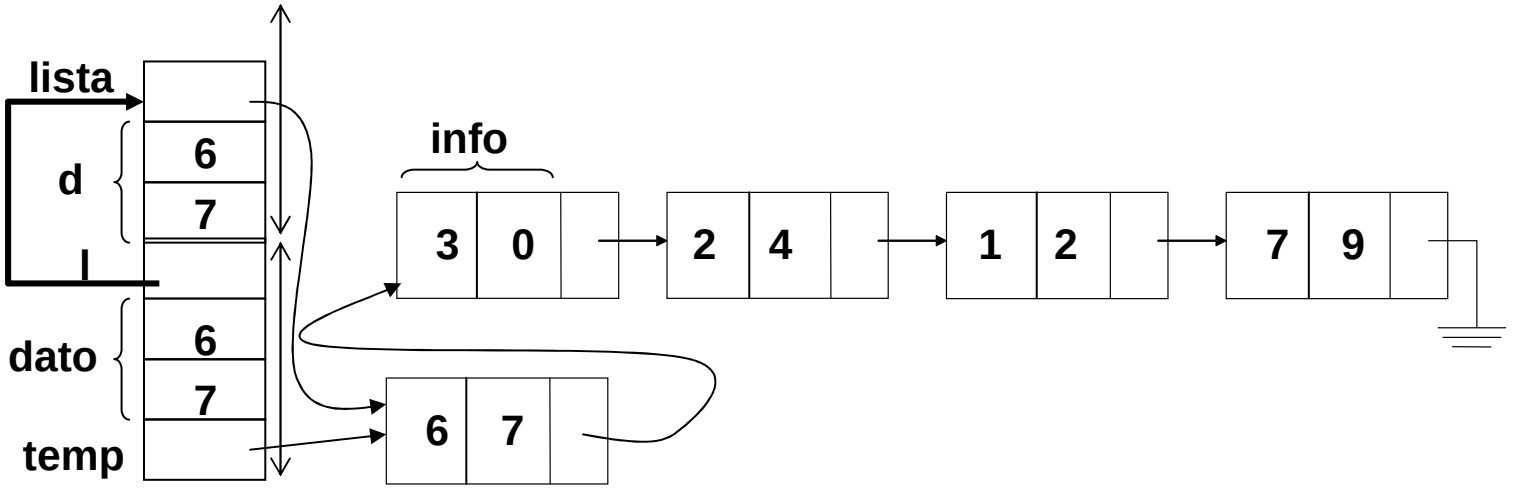
Implementazione di inserisciInTesta

```
void inserisciInTesta(Lista *l, TipoDato dato)
{
  Lista temp;
  temp = malloc (sizeof(ElemLista));
  temp->info = dato;
  temp->next = *l;
  *l = temp;
}
```



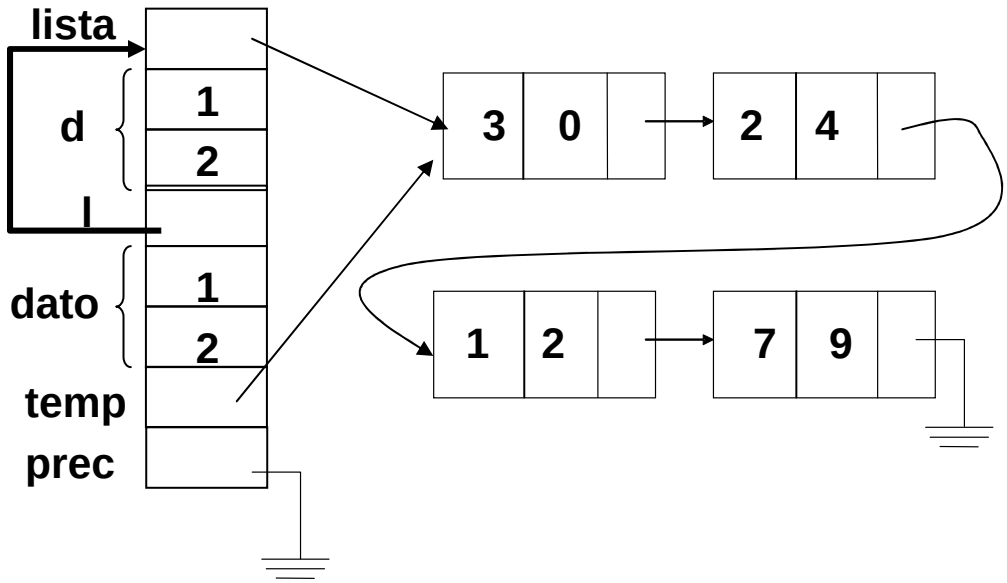
Implementazione di inserisciInTesta

```
void inserisciInTesta(Lista *l, TipoDato dato)
{
  Lista temp;
  temp = malloc (sizeof(ElemLista));
  temp->info = dato;
  temp->next = *l;
  *l = temp;
}
```



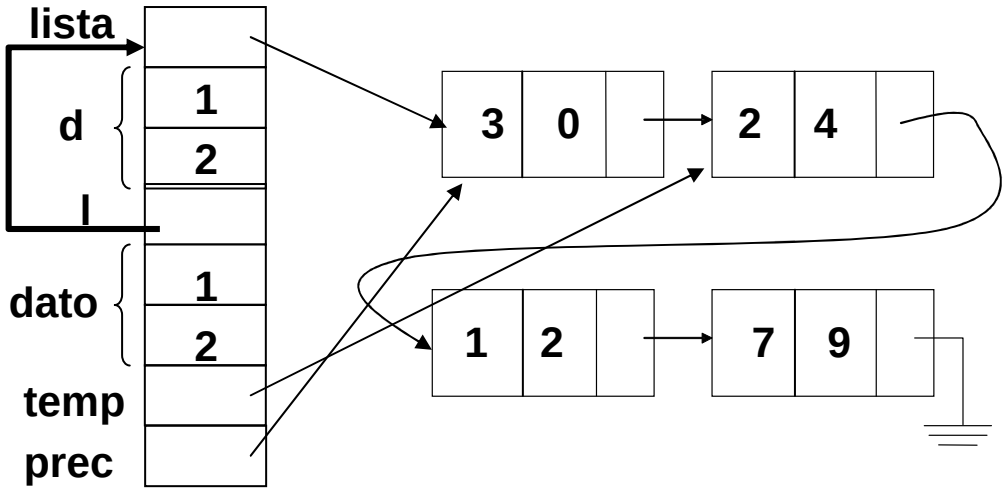
Implementazione di cancella

```
void cancella(Lista *l, TipoDato dato)
{
  Lista temp, prec;
  temp = *l;
  prec = NULL;
  while(temp!=NULL&&
        uguale(temp->info, dato)==false) {
    prec = temp;
    temp = temp->next;
  }
  if (temp!=NULL && prec!=NULL)
  {
    prec->next = temp->next;
    free(temp);
  }
  else if(prec == NULL)
  {
    *l=temp->next;
    free(temp);
  }}
```



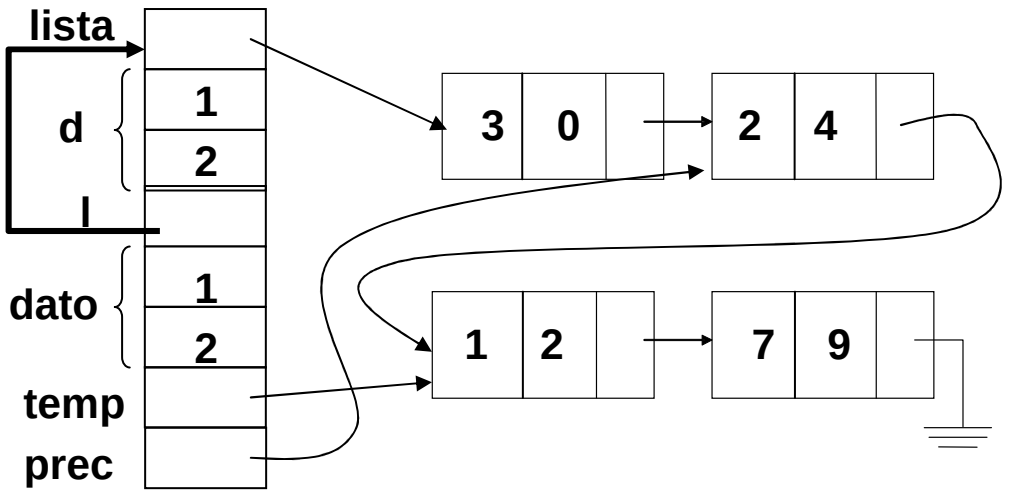
Implementazione di cancella

```
void cancella(Lista *l, TipoDato dato)
{
  Lista temp, prec;
  temp = *l;
  prec = NULL;
  while(temp!=NULL&&
        uguale(temp->info, dato)==false) {
    prec = temp;
    temp = temp->next;
  }
  if (temp!=NULL && prec!=NULL)
  {
    prec->next = temp->next;
    free(temp);
  }
  else if(prec == NULL)
  {
    *l=temp->next;
    free(temp);
  }}
```



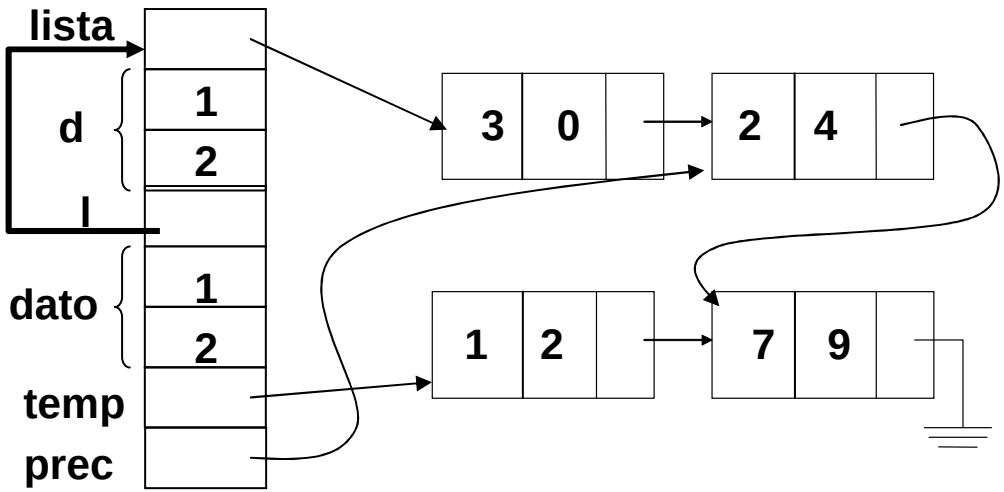
Implementazione di cancella

```
void cancella(Lista *l, TipoDato dato)
{
  Lista temp, prec;
  temp = *l;
  prec = NULL;
  while(temp!=NULL&&
        uguale(temp->info, dato)==false) {
    prec = temp;
    temp = temp->next;
  }
  if (temp!=NULL && prec!=NULL)
  {
    prec->next = temp->next;
    free(temp);
  }
  else if(prec == NULL)
  {
    *l=temp->next;
    free(temp);
  }}
```



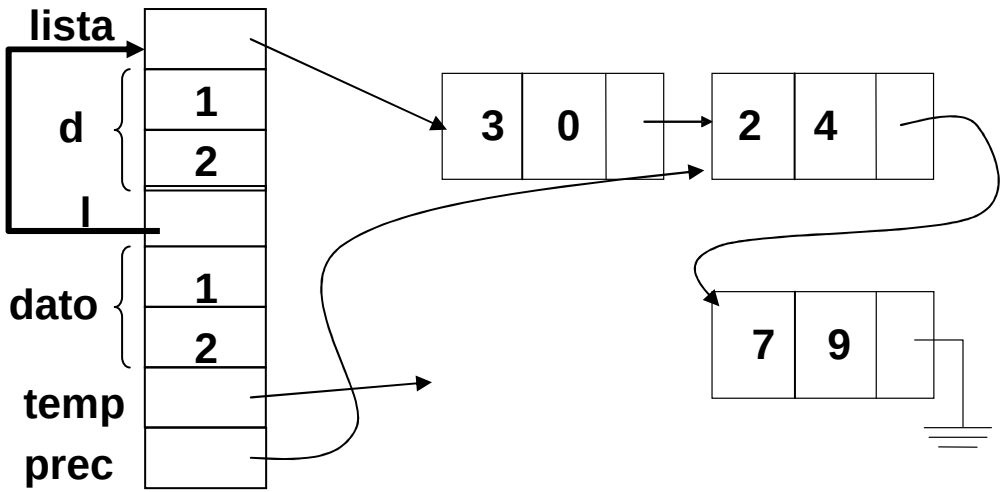
Implementazione di cancella

```
void cancella(Lista *l, TipoDato dato)
{
  Lista temp, prec;
  temp = *l;
  prec = NULL;
  while(temp!=NULL&&
        uguale(temp->info, dato)==false) {
    prec = temp;
    temp = temp->next;
  }
  if (temp!=NULL && prec!=NULL)
  {
    prec->next = temp->next;
    free(temp);
  }
  else if(prec == NULL)
  {
    *l=temp->next;
    free(temp);
  }}
```



Implementazione di cancella

```
void cancella(Lista *l, TipoDato dato)
{
  Lista temp, prec;
  temp = *l;
  prec = NULL;
  while(temp!=NULL&&
        uguale(temp->info, dato)==false) {
    prec = temp;
    temp = temp->next;
  }
  if (temp!=NULL && prec!=NULL)
  {
    prec->next = temp->next;
    free(temp);
  }
  else if(prec == NULL)
  {
    *l=temp->next;
    free(temp);
  }}
```



Cancella ricorsiva (1)

- Algoritmo
  - Se la lista è vuota non è necessario svolgere alcuna operazione
  - Se il primo elemento della lista è quello cercato, lo si elimina (**`*l=temp->next; free(temp);`**)
  - Altrimenti si richiama la cancella sulla sottolista che segue il primo elemento

Cancella ricorsiva (2)

```
void cancRic(Lista *l, TipoDato dato)
{

  Lista temp;

  temp = *l;
  if(temp!=NULL)
      if(uguale(temp->info, dato)==true)
      {
        *l=temp->next;
        free(temp);
      }
      else cancRic(&(temp->next), dato);
}
```