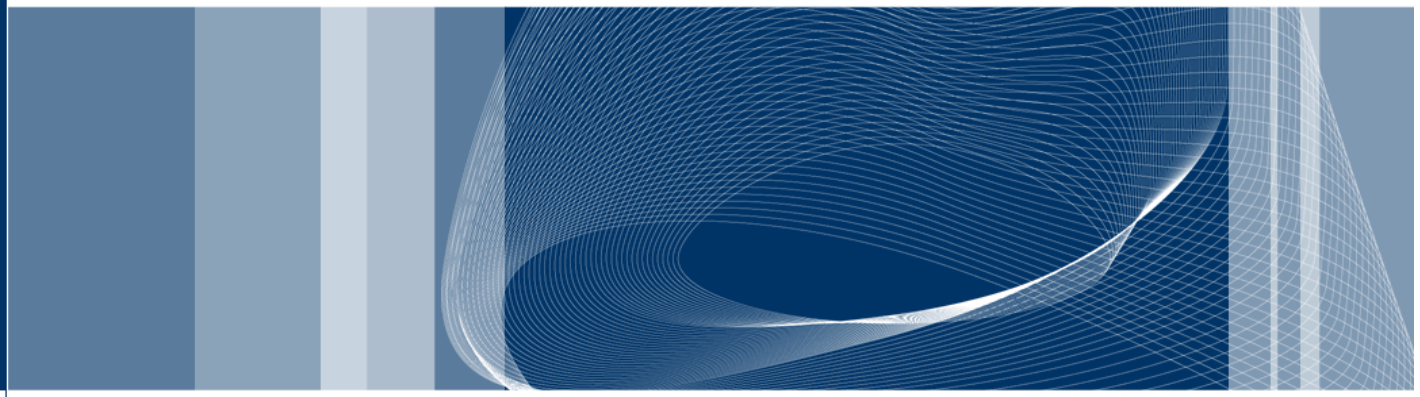
POLITECNICO DI MILANO



Strutture dati dinamiche

Fondamenti di Informatica

Daniele Loiacono

Che problema vogliamo risolvere?

Lavorare con una sequenza di elementi

- Leggiamo una sequenza di interi terminata da uno zero:

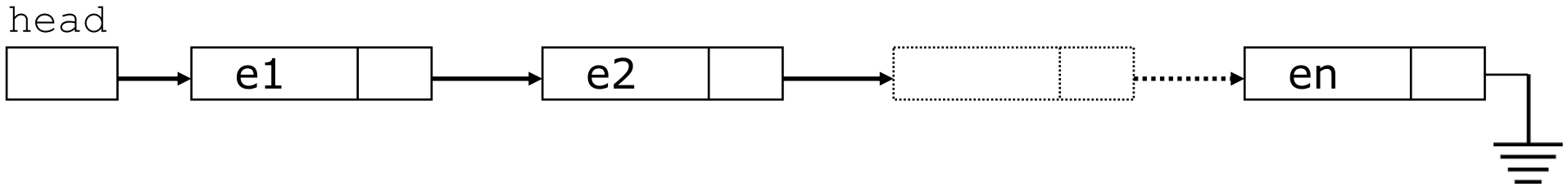
```
int v[N],i;
scanf("%d",&v[0]);
for (i=1; i<N && v[i-1]!=0; i++)
    scanf("%d",&v[i]);
```

- Come scelgo N?
  - N troppo grande, spreco molta memoria
  - N troppo piccolo, non riesco a memorizzare tutta la sequenza

Soluzione? Usiamo una lista

Lista

- Una **lista** è una struttura dati che consente di rappresentare una sequenza di elementi:
  - di lunghezza variabile (non definita a priori)
  - che permette di inserire/rimuovere elementi dinamicamente
- La lista viene implementata attraverso una sequenza di *nodi* ciascuno dei quali contiene:
  - un elemento della sequenza che si vuole rappresentare
  - un puntatore al nodo succesivo
- L'accesso alla lista avviene tramite un puntatore `head` che punta al primo nodo della lista
- Il puntatore dell'ultimo nodo conterrà il valore a `NULL` per indicare che non ci sono elementi successivi (nel caso la lista sia vuota sarà `head` a contenere il valore `NULL`)



Implementazione lista

```
typedef int data;

struct nodo
{
  data el;
  struct nodo *next;
};

typedef struct nodo *lista;

...

lista l = NULL;
```

Operazioni sulle liste

- Alcuni esempi di operazioni su una lista
  - calcola lunghezza di una lista
  - ricerca di un elemento
  - inserimento di un elemento in testa, in coda
  - rimuovi un elemento in testa o in coda
  - rimuovi un elemento specifico

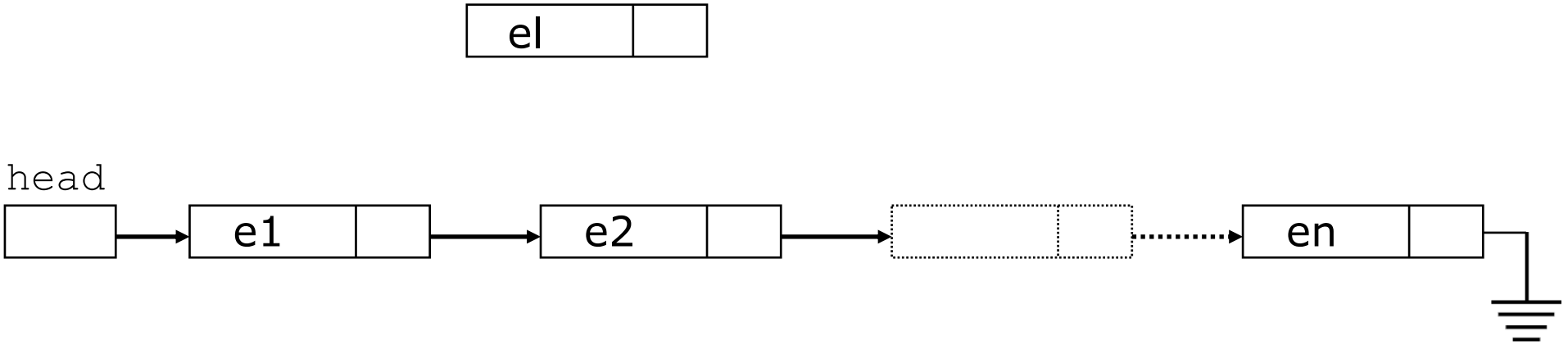
Calcola lunghezza di una lista

```
int lunghezza(lista l)
{
  if (l==NULL)
    return 0;
  else
    return 1 + lunghezza (l->next);
}
```

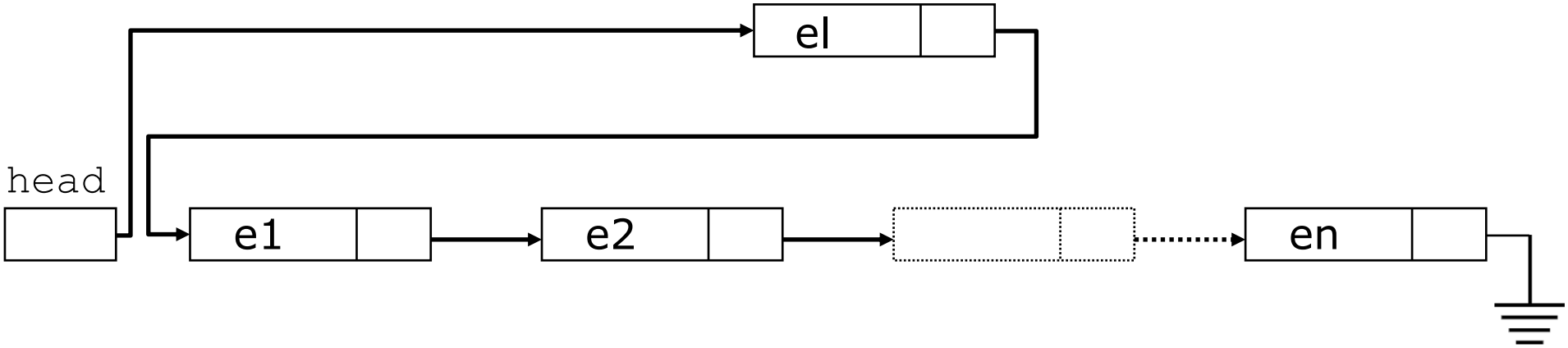
Ricerca di un elemento

```
lista ricerca(lista l, data el)
{
  if (l==NULL || l->el == el)
    return l;
  else
    return ricerca(l->next,el);
}
```

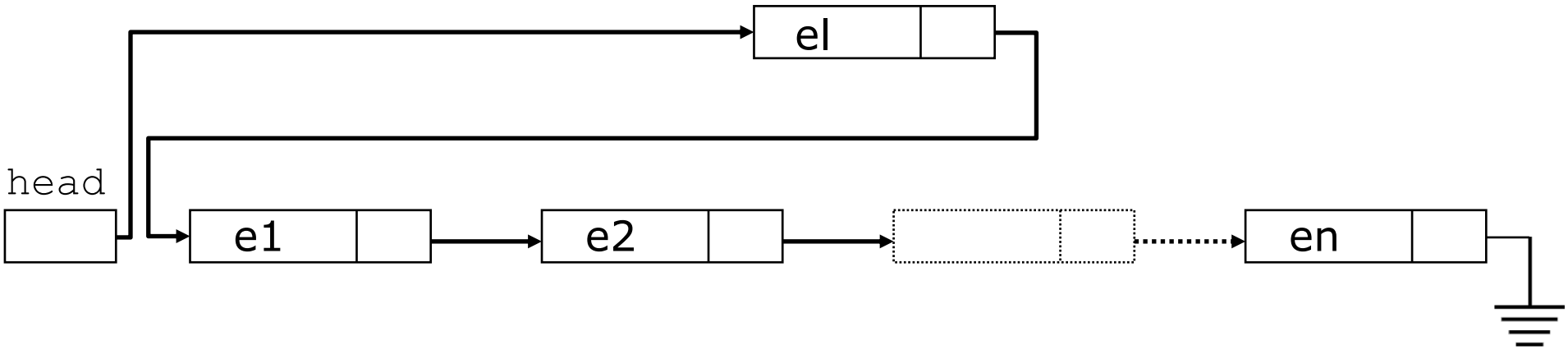
Inserisci in testa



Inserisci in testa

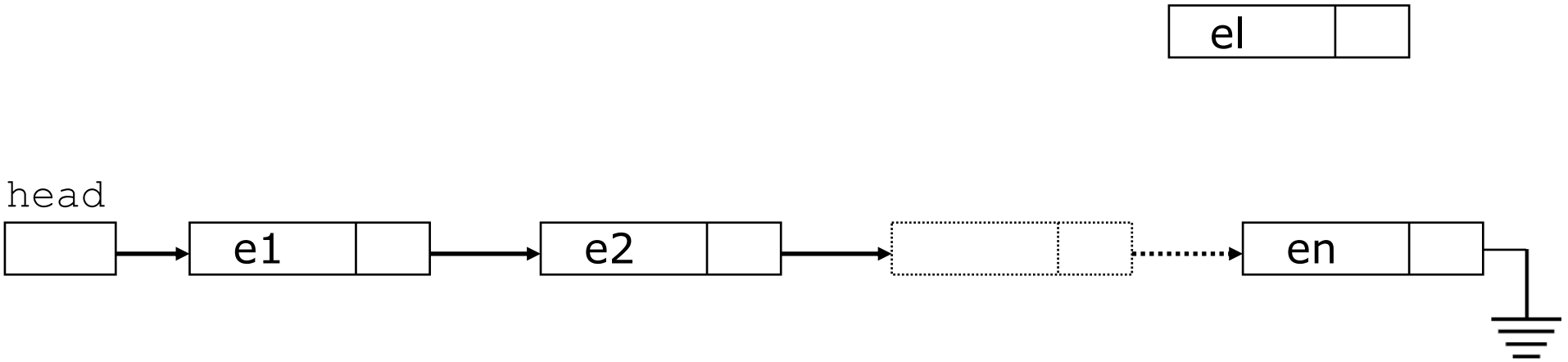


Inserisci in testa

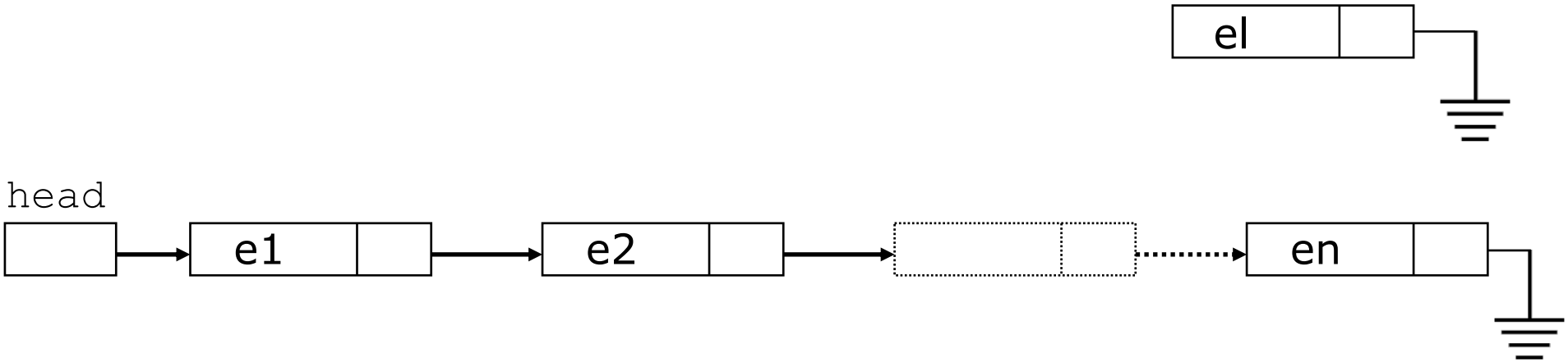



```
lista inserisci_testa (lista l, data el)
{
  struct nodo *temp = malloc(sizeof(struct nodo));
  temp->el = el;
  temp->next = l;
  return temp;
}
```

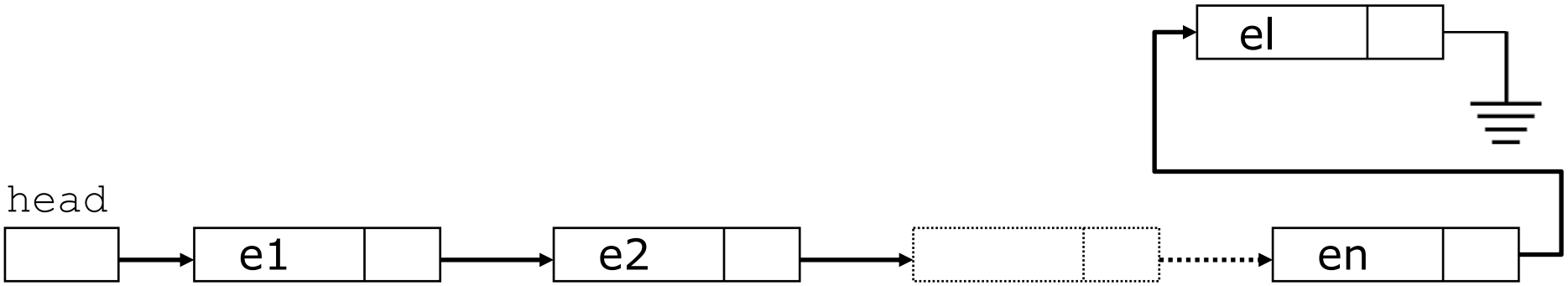
Inserisci in coda



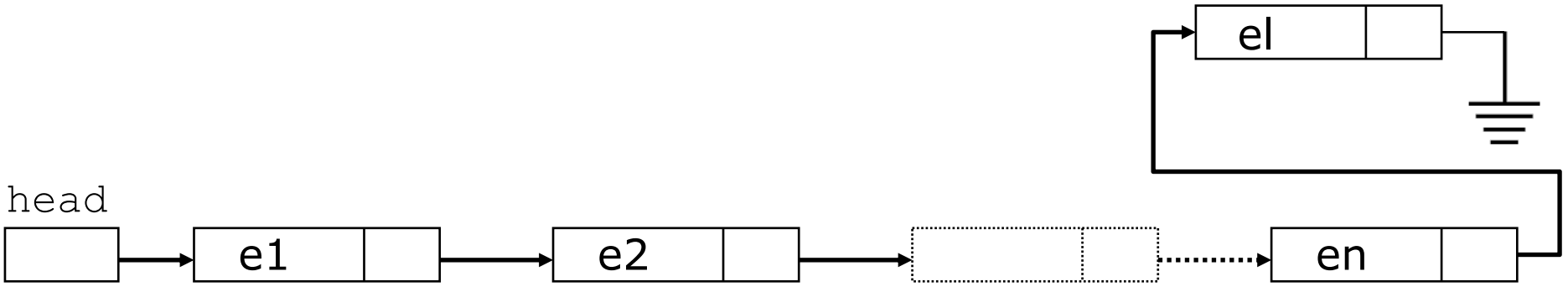
Inserisci in coda



Inserisci in coda

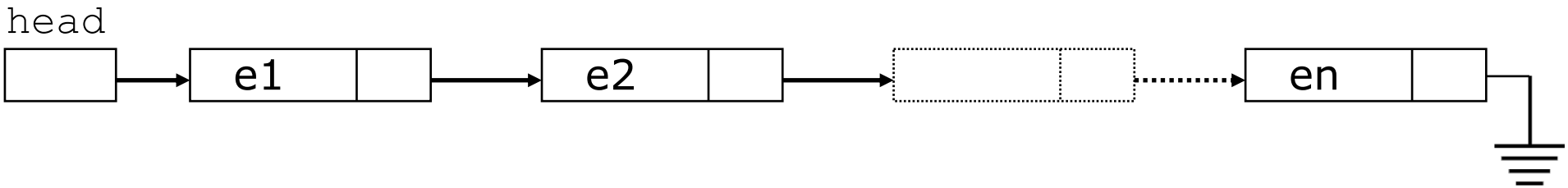


Inserisci in coda

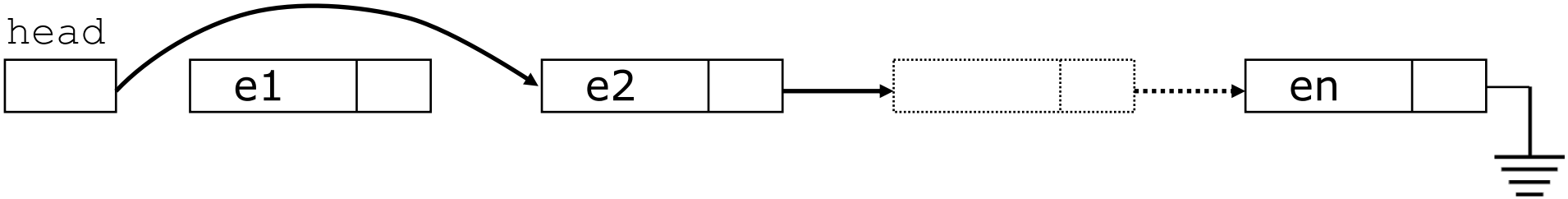



```
lista inserisci_coda (lista l, data el)
{
  if (l == NULL)
    return inserisci_testa(l,el);
  else
  {
    l->next = inserisci_coda(l->next,el);
    return l;
  }
}
```

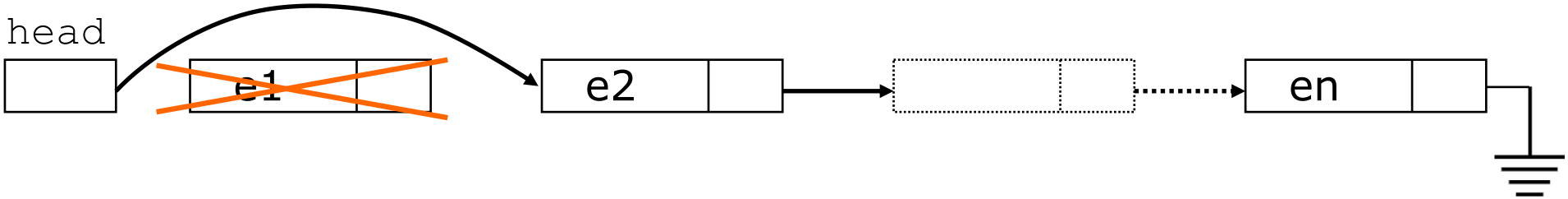
Rimuovi in testa



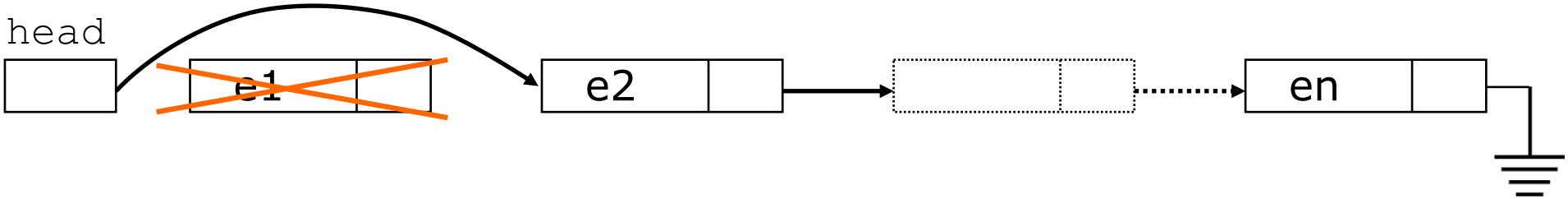
Rimuovi in testa



Rimuovi in testa

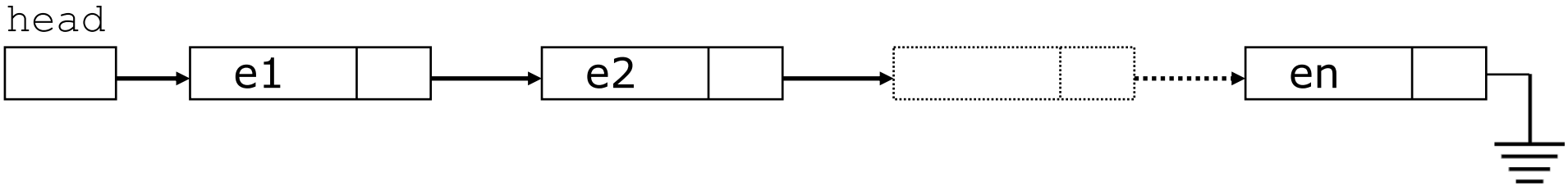


Rimuovi in testa

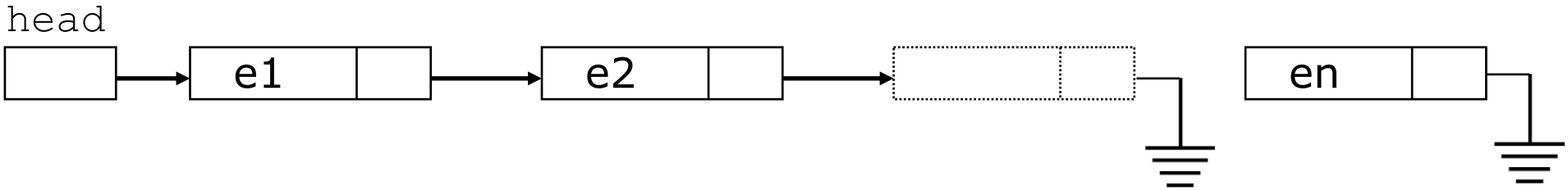



```
lista rimuovi_testa (lista l)
{
  if (l!=NULL)
  {
    lista temp = l;
    l = l->next;
    free(temp);
  }
  return l;
}
```

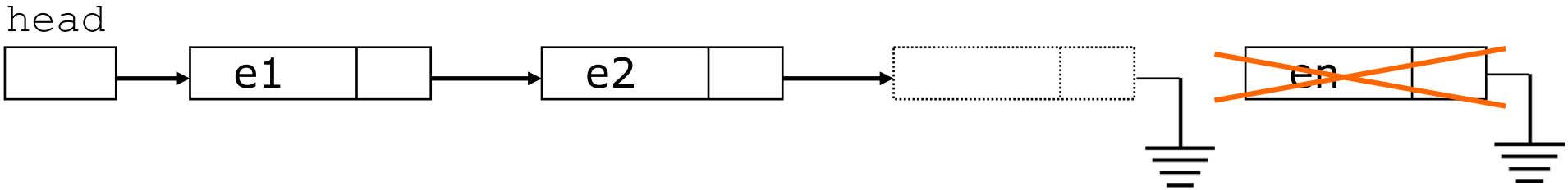
Rimuovi in coda



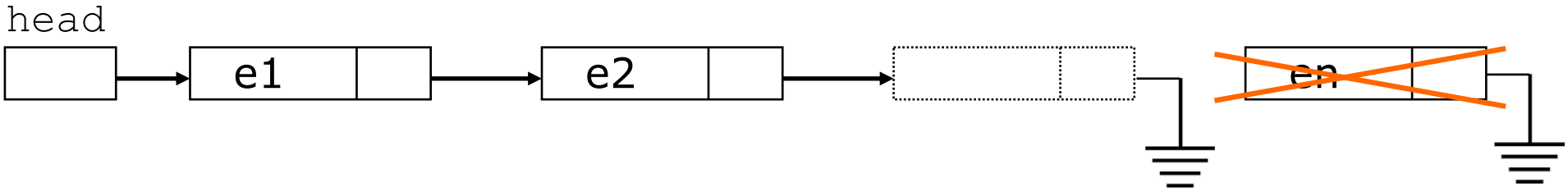
Rimuovi in coda



Rimuovi in coda

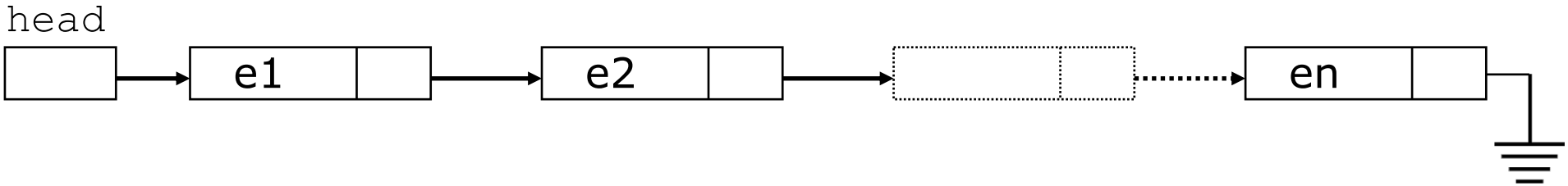


Rimuovi in coda

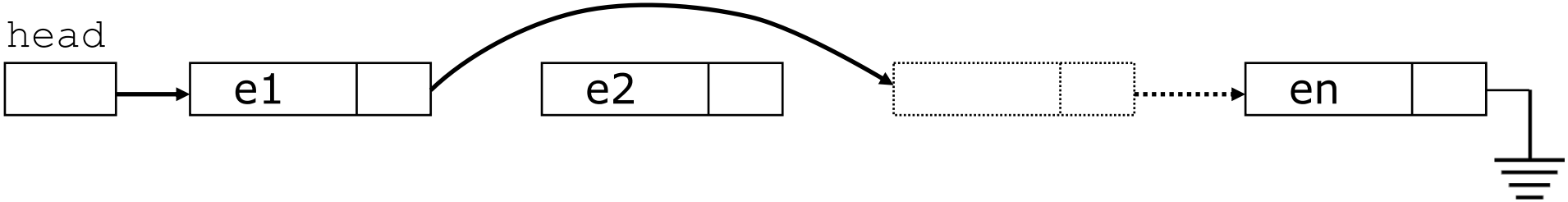



```
lista rimuovi_coda (lista l)
{
  if (l!=NULL)
  {
    if (l->next == NULL)
     return rimuovi_testa(l);
    else
    {
      l->next = rimuovi_coda(l->next);
      return l;
    }
  }
  else return NULL;
}
```

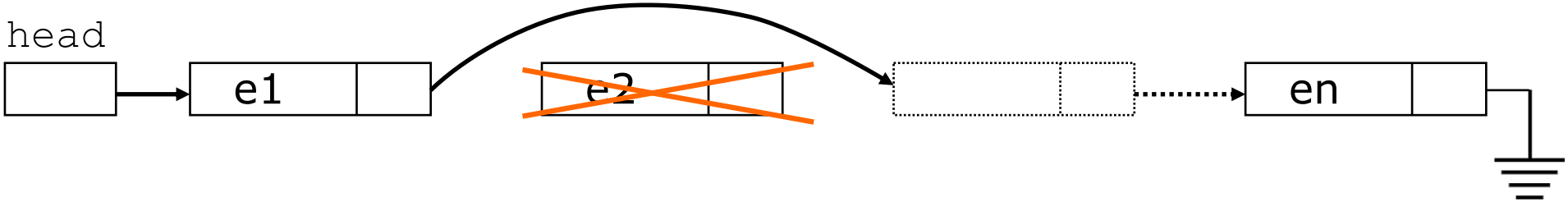
Rimuovi un elemento



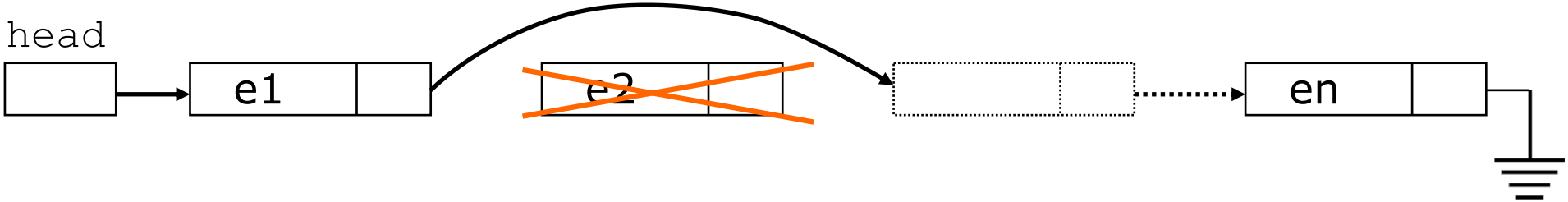
Rimuovi un elemento



Rimuovi un elemento



Rimuovi un elemento




```
lista rimuovi(lista l, data el)
{
  if (l==NULL)
   return l;
  if (l->el == el)
   return rimuovi_testa(l);
  else
  {
    l->next = rimuovi(l->next, el);
    return l;
  }
}
```

Riassumendo…

- Strutture dati dinamiche realizzate mediante puntatori
- Liste: primo e fondamentale (ma non unico) esempio di struttura dinamica
- Una prima valutazione dell'efficienza della struttura dinamica lista rispetto all'array:
  - Si evita lo spreco di memoria/rischio di overflow (obiettivo iniziale)
  - A prezzo di un (lieve) aumento di occupazione di memoria dovuto ai puntatori
  - Da un punto di vista del tempo necessario all'esecuzione degli algoritmi: pro e contro (inserire in testa meglio, inserire in coda peggio, …)

Organizziamo meglio il codice…

Programmazione modulare…

- Un sistema software è costituito da un insieme di moduli e da relazioni tra questi
- Ogni modulo è costituito da una **interfaccia** e da una **implementazione**
  - L'interfaccia è l'insieme di tutti e soli i suoi elementi che devono essere conosciuti da chi **usa** il modulo per farne un uso appropriato
  - L'**implementazione** è l'insieme dei meccanismi che permettono di **realizzare** le funzionalità del modulo

Linee guida

- Information hiding
  - Meno informazioni sono note all'utilizzatore di un modulo, meno vincoli sussitono per l'implementazione.
  - Tuttavia l'interfaccia deve contenere tutte le informazioni necessarie per utilizzare il modulo, per evitare che l'utente sia costretto a esaminare l'implementazione.
- Alta coesione
  - È bene che variabili, procedure, e altri elementi spesso utilizzati congiuntamente siano raggruppati nello stesso modulo dando ad ogni modulo un alto livello di coesione interna
- Basso accopiamento
  - Elementi che raramente interagiscono tra loro possono essere allocati in moduli diversi, ottenendo così moduli con un basso livello di accoppiamento

Modularizzazione in C

- In C si utilizzano gli header file (file .h) per definire le interfacce, principalmente per:
  - dichiarazioni di tipo e variabili
  - dichiarazione di funzione
- Le implementazione vengono invece incluse in corrispondenti file sorgenti (file .c)
- Per utilizzare un modulo, occorre
  - includere il suo header file
    `#include "nome_modulo.h"`
  - compilare l'implementazione del modulo insieme al sorgente che lo utilizza

Esempio: lista.h

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef int data;
struct nodo
{
  data el;
  struct nodo *next;
};
typedef struct nodo *lista;

int lunghezza (lista l);
lista ricerca (lista l, data el);
†lista inserisci_testa (lista l, data el);
lista inserisci_coda (lista l, data el);
lista rimuovi_testa (lista l);
lista rimuovi_coda (lista l);
lista rimuovi(lista l, data el);
void stampa (lista l);
```