Liste concatenate

Fondamenti di Informatica, AA 2022/23

Luca Cassano

luca.cassano@polimi.it

Liste concatenate

- C’è spesso la necessità di gestire una sequenza di lunghezza indefinita di valori

- Come dimensioniamo l’array?
  - Se viene sovrastimata la dimensione si spreca spazio
  - Se viene sottostimata non c’è spazio per tutti i valori

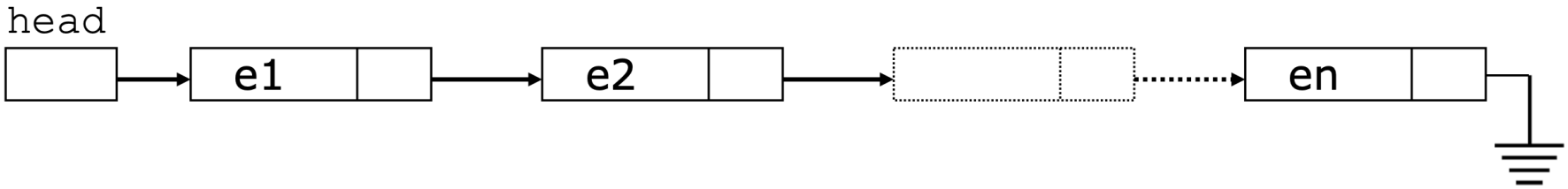
- Allocare dinamicamente l’array risolve solo parzialmente il problema poiché se termina lo spazio bisogna riallocarlo

Liste concatenate

- 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

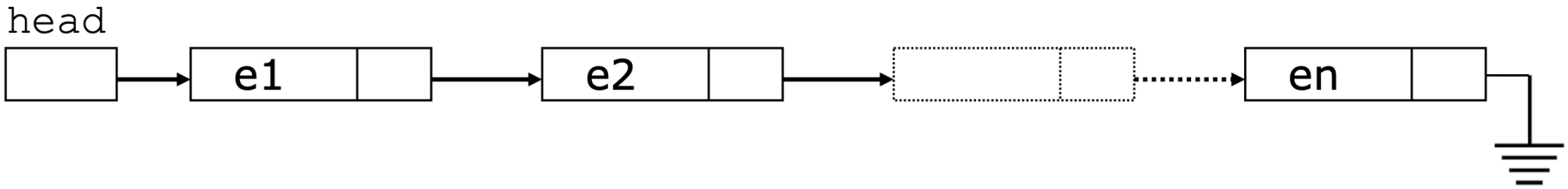
Liste concatenate

- 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 successivo



Liste concatenate

- L'accesso alla lista avviene tramite un puntatore `head` che punta al primo nodo della lista
- Il puntatore dell'ultimo nodo conterrà `NULL` per indicare che non ci sono elementi successivi
  - Nel caso la lista sia vuota sarà `head` a contenere `NULL`



Liste concatenate

- Il nodo della lista è definito con la seguente `struct` :

```
typedef struct nodo_{
   int num;
   struct nodo_ *next;
} nodo_t;
```

Valore contenuto nel nodo (un intero nell'esempio)

Puntatore al nodo successivo

Liste concatenate

- Il nodo della lista è definito con la seguente `struct`:

```
typedef struct nodo_{
   int num;
   struct nodo_ *next;
} nodo_t;
```

Valore contenuto nel nodo (un intero nell'esempio)

Puntatore al nodo successivo

- Il puntatore alla testa della lista è definito nel `main`
  - All'inizio la lista è vuota quindi inizializiamo il puntatore a `NULL`

```
int main(){
   nodo_t *head = NULL;
   ...
```

Liste concatenate – operazioni sulle liste

- Per manipolare le liste si possono definire i seguenti sottoprogrammi (comuni per qualsiasi tipo di lista)
  - Visualizzazione del contenuto di una lista
  - Calcolo della lunghezza di una lista
  - Ricerca di un elemento nella lista
  - Controllo esistenza di un elemento
  - Inserimento di un elemento in testa alla lista
  - Inserimento di un elemento nel mezzo della lista
  - Inserimento di un elemento in coda ad una lista
  - Rimozione dell'elemento in testa ad una lista
  - Rimozione di un elemento dal mezzo della lista
  - Rimozione dell'elemento in coda ad una lista
  - ...

Liste concatenate – visualizzazione del contenuto

- Bisogna scorrere tutta la lista per visualizzare ciascun elemento
  - Realizzazione iterativa:

```
void visualizza(nodo_t* l){
  while(l != NULL){
    printf("%d", l->num);
    l = l->next;
  }
}
```

- Possiamo utilizzare direttamente `l` per scorrere la lista poiché non ci serve mantenere nel sottoprogramma il puntatore alla testa

Liste concatenate – visualizzazione del contenuto

- Realizzazione ricorsiva:

```
void visualizza(nodo_t* l){
  if(l == NULL)
    return;
  printf("%d", l->num);
  visualizza(l->next);
}
```

Caso base: la lista è vuota. Quindi non c'è niente da fare

Passo ricorsivo: stampo l'elemento corrente e chiamo il sottoprogramma sull'elemento successivo della lista

Liste concatenate – calcolo della lunghezza

- Sottoprogramma molto simile alla visualizzazione del contenuto della lista
  - Realizzazione iterativa:

```
int lunghezza(nodo_t* l){
   int len = 0;
   while(l != NULL) {
        len++;
        l = l->next;
  }
  return len;
}
```

Liste concatenate – calcolo della lunghezza

- Realizzazione ricorsiva:

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

- Caso base: se la lista è vuota la sua lunghezza è 0
- Passo iterativo: la lunghezza della lista è pari a 1 più la lunghezza della lista che parte dall'elemento successivo

Liste concatenate – cerca un elemento

- Sottoprogramma molto simile alla visualizzazione del contenuto della lista
- Il sottoprogramma restituisce il puntatore al nodo, se l'elemento è trovato, altrimenti `NULL`
  - Realizzazione iterativa:

```
nodo_t* cerca(nodo_t* l, int num){
  while(l != NULL && l->num != num){
    l = l->next;
  }
  return l;
}
```

Liste concatenate – cerca un elemento

- Realizzazione ricorsiva:

```
nodo_t* cerca(nodo_t* l, int num){
   if(l == NULL || l->num == num)
        return l;
   return cerca(l->next, num);
}
```

Se la lista è vuota, `num` non è stato trovato; altrimenti se il nodo corrente contiene un valore uguale a `num`, è stato trovato.

Ricordarsi di non invertire le due condizioni in or altrimenti si può verificare un crash. Perché?

- Nel chiamante il risultato va assegnato ad un puntatore (con entrambe le soluzioni)

```
nodo_t *head, *v;
...
v = cerca(head, 5);
```

Liste concatenate – controllo esistenza di un elemento

- Il sottoprogramma restituisce 1 se l'elemento è trovato altrimenti 0
  - Realizzazione iterativa:

```
int esiste(nodo_t* l, int num){
   while(l != NULL && l->num!=num)
      l = l->next;
   return l!=NULL;
}
```

Si itera finché la lista non è finita e l'elemento non è stato trovato

Viene restituito 1 se `l` non è `NULL`. Infatti in tal caso si è usciti dal ciclo perché `num` è stato trovato. Inoltre è necessario confrontare esplicitamente `l` con `NULL` per ottenere come responso 1 o 0

Liste concatenate – controllo esistenza di un elemento

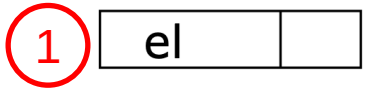
- Realizzazione ricorsiva:

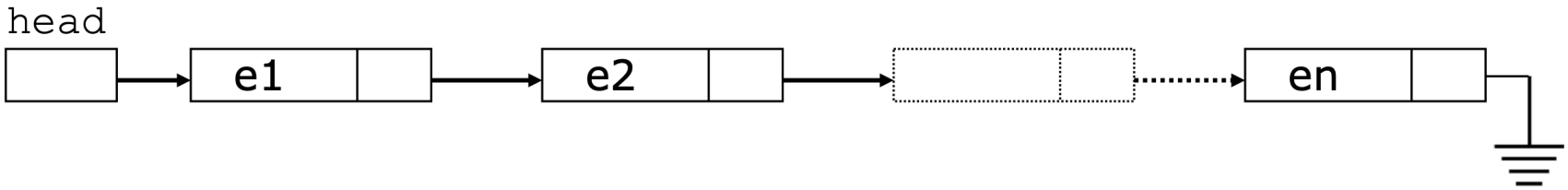
```
int esiste(nodo_t* l, int num){
  if(l==NULL || l->num==num)
    return l!=NULL;
  return cerca(l->next, num);
}
```

- Caso base:
  - Se la lista è vuota l'elemento non è esiste e quindi restituisco 0
  - Se il primo elemento è quello cercato restituisco 1
- Passo iterativo: restituisco il risultato della ricerca sulla lista che parte dall'elemento successivo

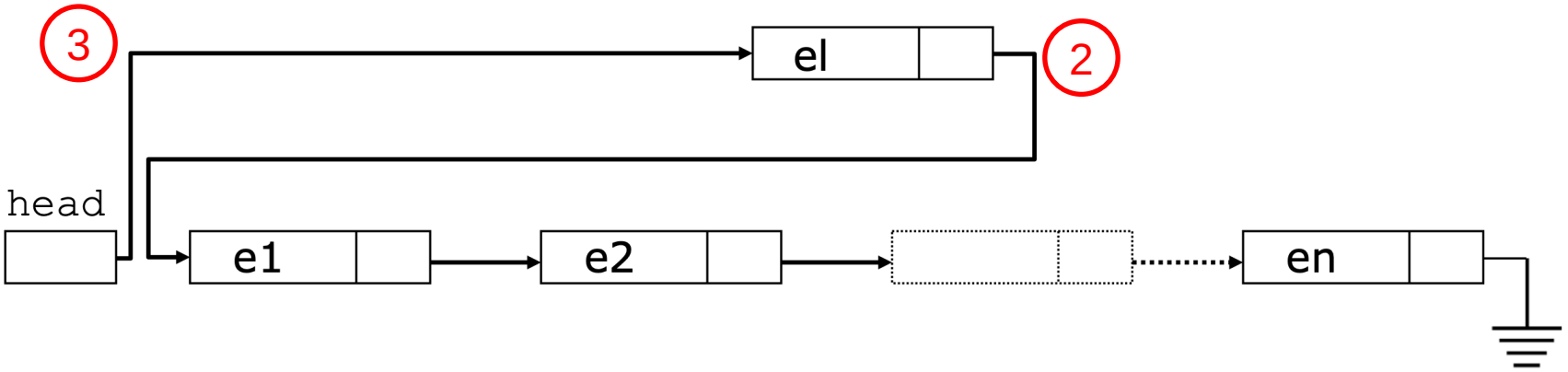
Liste concatenate – inserimento in testa

1. Si alloca il nuovo nodo e si salva dentro il valore





2. Si assegna al puntatore `next` del nuovo nodo la testa della lista (contenuto di `head`)
3. Si assegna il nuovo nodo al puntatore `head`



Liste concatenate – inserimento in testa

```
nodo_t* inserisciInTesta(nodo_t* l, int num){
  nodo_t *tmp;
  tmp = malloc(sizeof(nodo_t));
  if(tmp != NULL){
    tmp->num = num;
  tmp->next = l;
  l = tmp;
  } else
      printf("Mem. esaurita\n");
  return l;
}
```

1

2

3

La testa della lista è stata modificata quindi bisogna restituire l'indirizzo alla nuova testa. In alternativa si può passare la testa con un doppio puntatore

Liste concatenate – inserimento in testa

- Nel chiamante il sottoprogramma va invocato con `head` come parametro ed il risultato restituito va assegnato a `head`

```
...
nodo_t *head;
int n;
...
head = inserisciInTesta(head, n);
```

Liste concatenate – inserimento in testa

- Realizzazione mediante doppio puntatore:

```
void inserisciInTesta(nodo_t** l, int num){
  nodo_t *tmp;
  tmp = malloc(sizeof(nodo_t));
  if(tmp != NULL){
    tmp->num = num;
  tmp->next = *l;
  *l = tmp;
  } else
      printf("Mem. esaurita\n");
}
```

1

2

3

La testa della lista è stata modificata. Perciò utilizzando un doppio puntatore possiamo «propagare» la modifica al chiamante

Liste concatenate – inserimento in testa

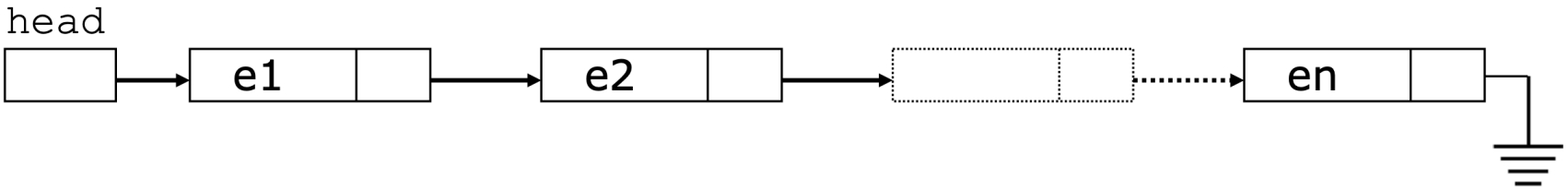
- Nel chiamante il sottoprogramma va invocata passano l'indirizzo del puntatore `head` come parametro

```
...
nodo_t *head;
int n;
...
inserisciInTesta(&head, n);
```

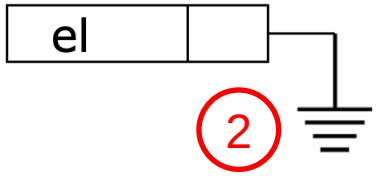
- Tutti i prossimi sottoprogrammi possono essere realizzate in entrambi i modi mostrati:
  - Passando il puntatore alla testa per valore e quindi dovendolo restituire, o
  - Passando il puntatore alla testa per indirizzo (con un doppio puntatore), quindi permettendo la modifica del puntatore nel record di attivazione del chiamante

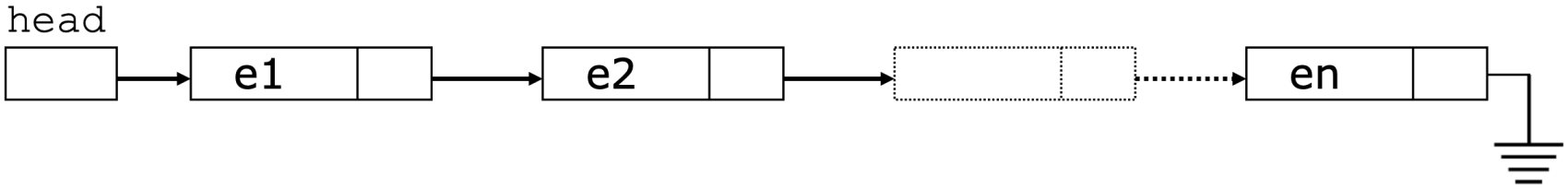
Liste concatenate – inserimento in coda

1. Si alloca il nuovo nodo e si salva dentro il valore



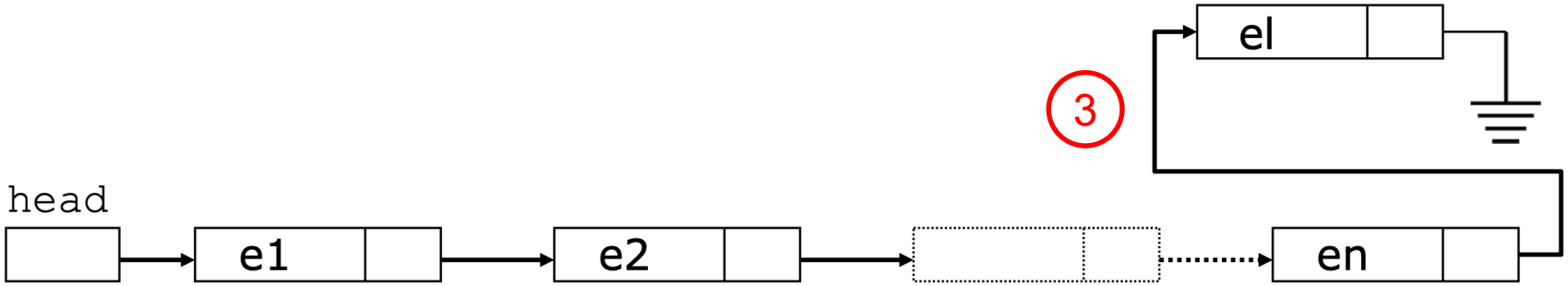
2. Si assegna al puntatore `next` il valore `NULL`





Liste concatenate – inserimento in coda

3. Si scorre tutta la lista per arrivare all'ultimo elemento e si assegna al suo `next` il nuovo nodo



- NOTA: se la lista è vuota l'inserimento in coda diventa un inserimento in testa
  - In tal caso va modificato il puntatore `head`

Liste concatenate – inserimento in coda

- Realizzazione iterativa:

```
nodo_t* inserisciInCoda(nodo_t * l, int num){
  nodo_t *tmp, *prec;
  tmp = malloc(sizeof(nodo_t));
1 if(tmp != NULL){
    tmp->num = num;
2   tmp->next = NULL;
    if(l == NULL)
      l = tmp;
    else{
3     for(prec=l; prec->next!=NULL; prec=prec->next);
      prec->next = tmp;
    }
  } else
      printf("Mem. esaurita!\n");
  return l;
}
```

Se la lista vuota si esegue l'inserimento in testa

Liste concatenate – inserimento in coda

- Realizzazione ricorsiva:

```
nodo_t* inserisciInCoda(nodo_t * l, int num){
  if (l == NULL)
    return inserisciInTesta(l, num);
  l->next = inserisciInCoda(l->next, num);
  return l;
}
```

- Caso base: se la lista è vuota si effettua un inserimento in testa e si restituisce la testa
- Passo iterativo: si effettua l'inserimento in coda sulla lista che parte dall'elemento successivo e la si concatena al nodo corrente (cioè la testa della lista della chiamata ricorsiva corrente)

Liste concatenate – inserimento dopo un dato elemento

1. Si alloca il nuovo nodo, si salva dentro il valore, e si fa puntare da tmp
2. Si scorre la lista con un puntatore ausiliario prec fino a trovare l'elemento cercato o fino alla fine della lista
3. Si assegna prec->next a tmp >next e si assegna tmp a prec->next
4. Se non si è trovato l'elemento, diventa un inserimento in coda
5. Se la lista è vuota diventa un inserimento in testa

Liste concatenate – inserimento dopo un dato elemento

- Realizzazione iterativa:

```
nodo_t* inserisciDopoElemento(nodo_t * l, int cerc, int num){
  nodo_t *tmp, *prec;
  tmp = malloc(sizeof(nodo_t));
  if(tmp != NULL){
    tmp->num = num;
    tmp->next = NULL;
    if(l == NULL)
      l = tmp;
    else{
      for(prec=l; prec->next!=NULL && prec->num != cerc;
prec=prec->next);
      tmp->next = prec->next;
      prec->next = tmp;
    }
  } else
      printf("Mem. esaurita!\n");
  return l;
}
```

Se la lista è vuota

Scorro fino all'ultimo elemento
o fino all'elemento cercato

Liste concatenate – inserimento dopo un dato elemento

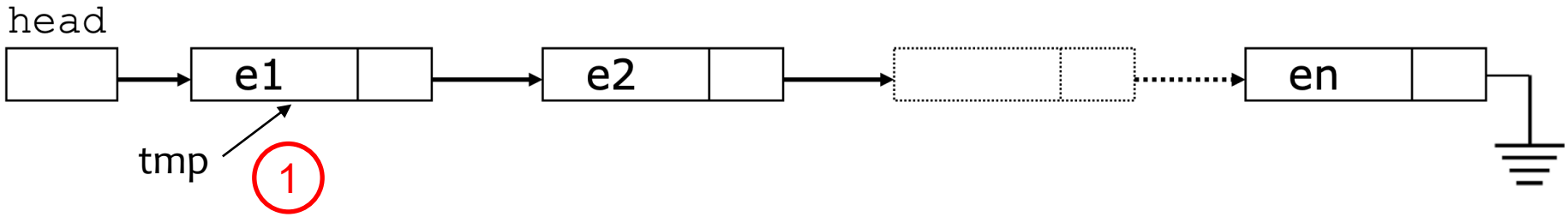
- Realizzazione ricorsiva:

```
nodo_t* inserisciDopoElemento(nodo_t * l, int cerc, int num){
  if (l == NULL || l->num == cerc)
    return inserisciInTesta(l->next, num);
  l->next = inserisciDopoElemento(l->next, cerc, num);
  return l;
}
```

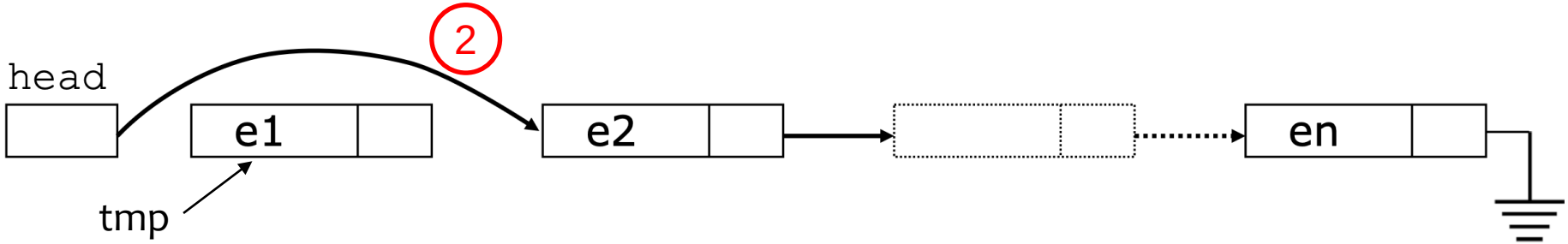
- Caso base: se la lista è vuota si effettua un inserimento in testa e si restituisce la testa
- Passo iterativo: si effettua l'inserimento in coda sulla lista che parte dall'elemento successivo e la si concatena al nodo corrente (cioè la testa della lista della chiamata ricorsiva corrente)

Liste concatenate – eliminazione testa

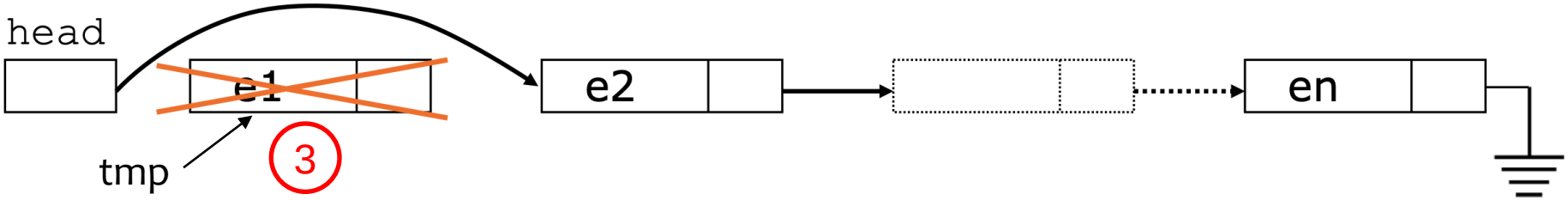
1. Si assegna la testa ad un puntatore temporaneo



2. Si assegna a `head` il valore in `next` del primo nodo



3. Si libera la memoria



Liste concatenate – eliminazione testa

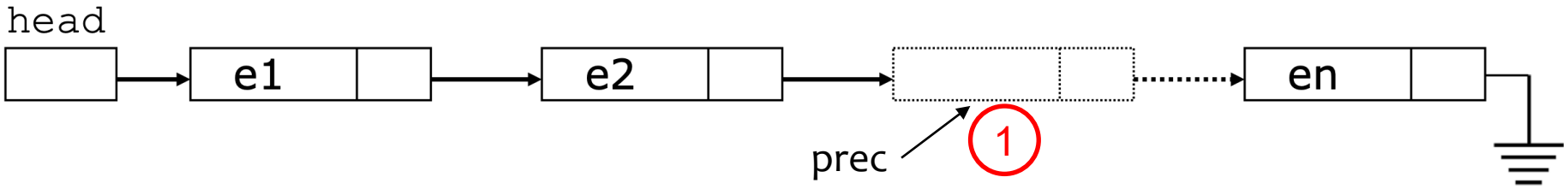
```
nodo_t* cancellaTesta(nodo_t* l){
   nodo_t *tmp;
   if(l != NULL){
  1 tmp = l;
  2 l = l->next;
  3 free(tmp);
   return l;
}
```

Se la lista è vuota non c'è niente da deallocare

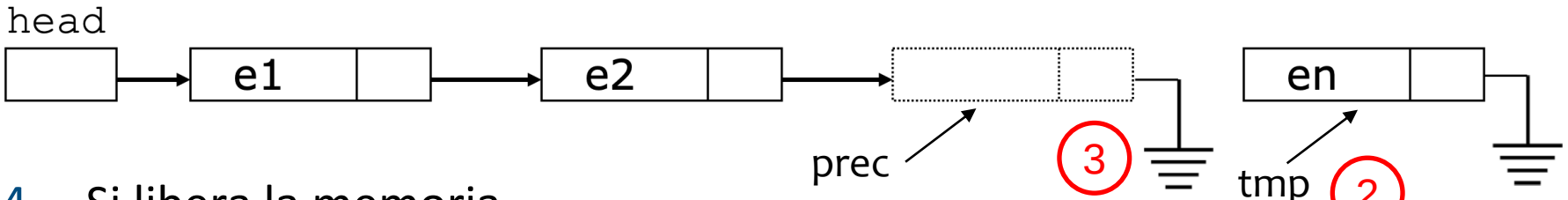
Si restituisce la testa della lista visto che è stata modificata

Liste concatenate – eliminazione coda

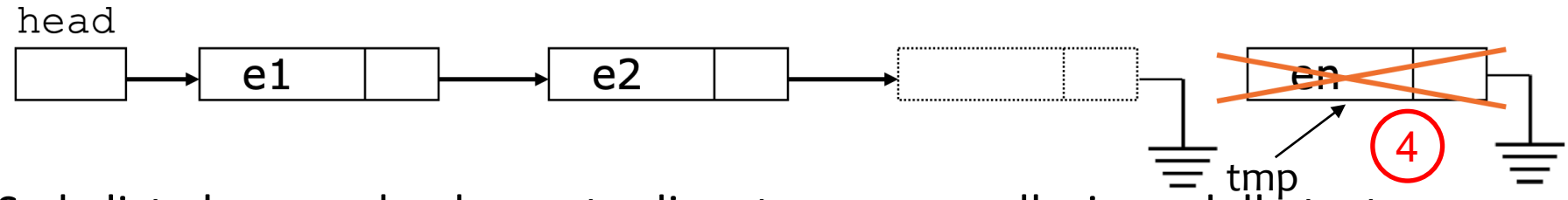
1. Si scorre la lista fino ad arrivare al penultimo elemento



2. Si assegna la coda ad un puntatore temporaneo
3. Si assegna al puntatore `next` del penultimo elemento `NULL`



4. Si libera la memoria



Se la lista ha un solo elemento diventa una cancellazione della testa

Liste concatenate – eliminazione coda

- Realizzazione iterativa:

```
nodo_t* cancellaCoda(nodo_t* l){
  nodo_t *tmp, *prec;
  if(l != NULL){
    if(l->next == NULL){
      free(l);
      l = NULL;
    } else {
    1 for(prec=l; prec->next->next != NULL; prec=prec->next);
    2 tmp = prec->next;
    3 prec->next = NULL;
    4 free(tmp);
    }
  }
  return l;
}
```

Se la lista è vuota non c'è niente da deallocare

Se la lista ha un solo elemento si effettua la cancellazione della testa

Scorro fino al penultimo elemento

Si restituisce la testa della lista visto che è stata potenzialmente modificata

Liste concatenate – eliminazione coda

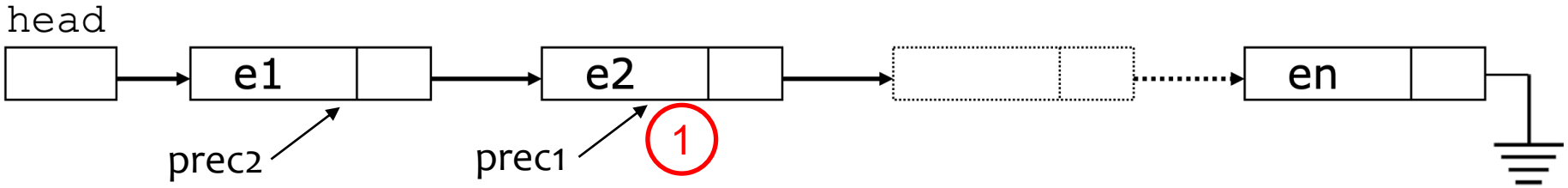
- Realizzazione ricorsiva:

```
nodo_t* cancellaCoda(nodo_t* l){
  if(l == NULL)
    return NULL;
  if(l->next == NULL)
    return cancellaTesta(l);
  l->next = cancellaCoda(l->next);
  return l;
}
```

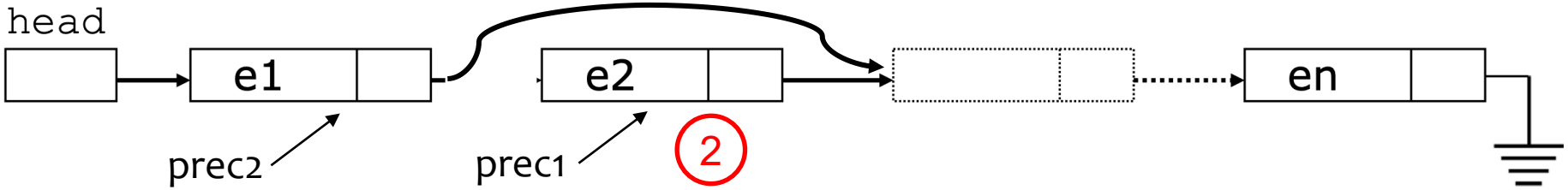
- Caso base:
  - Se la lista è vuota si restituisce NULL
  - Se la lista ha un solo elemento si effettua una cancellazione in testa
- Passo iterativo: si effettua la cancellazione della coda sulla lista che parte dall'elemento successivo e la si concatena al nodo corrente (cioè la testa della lista della chiamata ricorsiva corrente)

Liste concatenate – eliminazione di un dato elemento

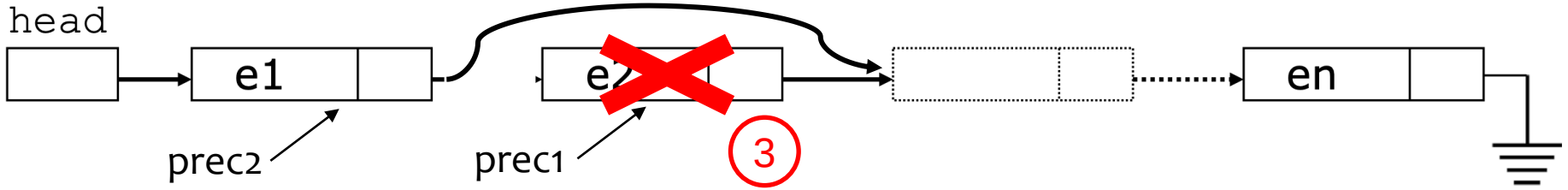
1. Si scorre la lista con due puntatori (prec1 e prec2) fino a che prec1 punta all'elemento da cancellare e prec2 all'elemento precedente oppure fino a che prec2 punta all'ultimo elemento della lista (e prec1 punta a NULL)



2. Si assegna la prec1->next a prec2->next



3. Dealloca l'elemento puntato da prec1



Se l'elemento da eliminare è il primo della lista, diventa un cancellazione dalla testa

Liste concatenate – eliminazione di un dato elemento

- Realizzazione iterativa:

```
nodo_t* cancellaElemento(nodo_t* l, int n){
  nodo_t *prec1, *prec2;
  if(l != NULL){
    if(l->num == n){
      cancellaTesta(l);
    }else if(l->next != NULL){
      prec2=l; prec1=l->next;
      while(prec1 != NULL && prec1->num != n) {
        prec1=prec1->next; prec2=prec2->next;
      }
      if(prec1 != NULL) {
        prec2->next = prec1->next;
        free(prec1);
      }
    }
  }
  return l;
}
```

1

2

3

Se la lista è vuota non c'è niente da deallocare

Se l'elemento da cancellare è in testa si effettua la cancellazione della testa

Se la lista ha almeno due elementi

Se l'elemento d eliminare esiste nella lista

Si restituisce la testa della lista visto che è stata potenzialmente modificata

Liste concatenate – eliminazione di un dato elemento

- Realizzazione ricorsiva:

```
nodo_t* cancellaElemento(nodo_t* l, int n){
  if (l==NULL)
    return l;
  if (l->num == n)
    return cancellaTesta(l);
  else {
    l->next = cancellaElemento(l->next, n);
    return l;
  }
}
```