Informatica - Mod. Programmazione
Lezione 09

Prof. Giuseppe Psaila

Laurea Triennale in Ingegneria Informatica
Università di Bergamo

Strutture Dati Dinamiche

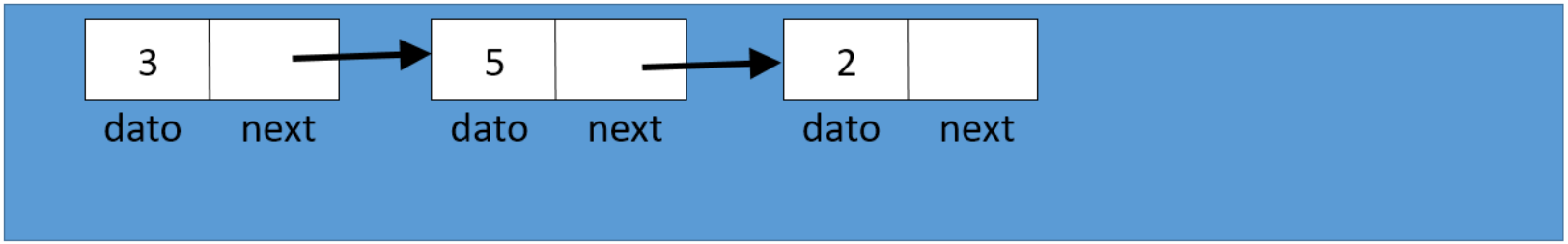
- Le **Strutture Dati Dinamiche** nascono per affrontare gli scenari in cui non è possibile stabilire in nessun momento quanti dati dovranno essere memorizzati.
- Sono basate sempre sull'allocazione dinamica
- Le più frequenti:
  Lista Semplice
  Albero (li vedrete nel corso di Programmazione a Oggetti).

Lista (Semplice)

- L'elemento fondamentale di una lista è il **NODO**
- È una struttura che contiene due parti:
  - una parte **Informativa**, con il dato da memorizzare
  - un campo denominato **next**, che punta al nodo successivo.
- Esempio: lista di interi

```
struct NODO
{
    int dato;
    NODO *next;
};
```

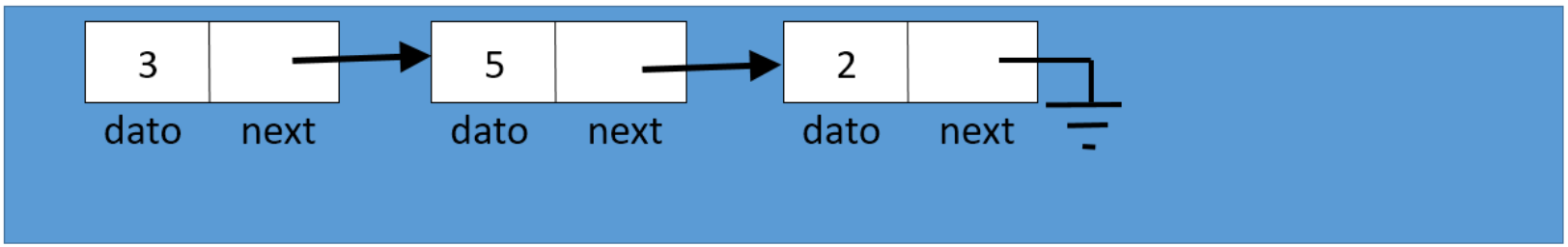
Esempio



Lista (Semplice)

- Come dire che la lista è finita?
- Il valore del campo `next` dell'ultimo nodo vale `NULL`
- Convenzionalmente, si usa il simbolo di **messa a terra**

Esempio



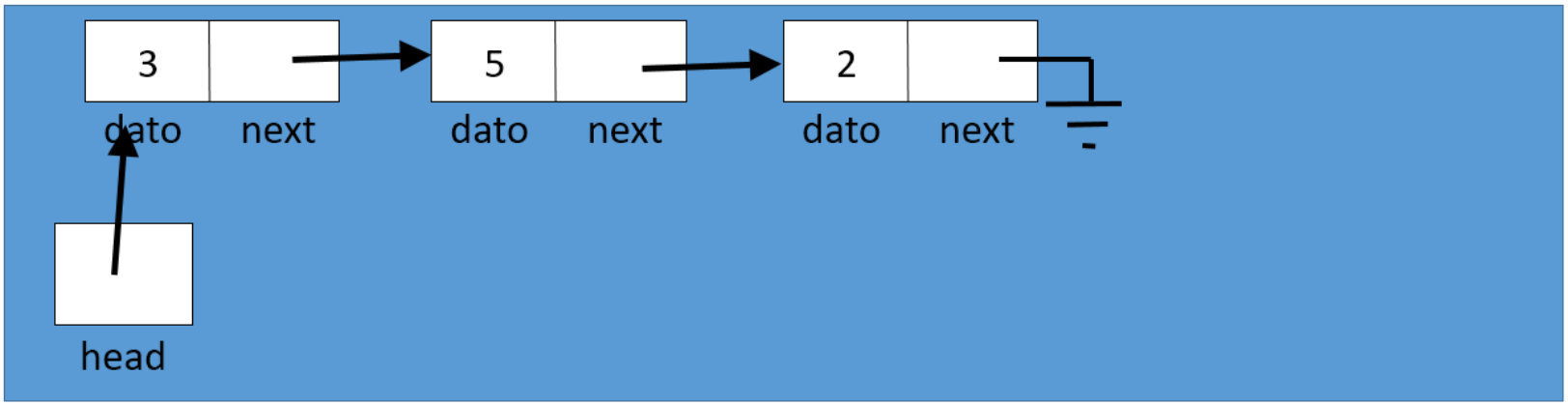
Lista (Semplice)

- Il Primo nodo della lista viene chiamato **Testa (Head)**
- l'Ultimo nodo della lista viene chiamato **Coda (Tail)**
- I nodi vengono allocati dinamicamente, quando serve, quindi sono nello Heap
- Serve una variabile per memorizzare l'indirizzo della testa

  `NODO *head=NULL;`

  deve essere inizializzata a `NULL`, perchè all'inizio la lista è vuota.

Esempio



Operazioni sulle Liste

- **Inserimento in Testa** - da sapere
- **Inserimento in Coda** - da sapere
- **Inserimento Ordinato** - da guardare sul libro
- **Cancellazione dalla Testa** - da sapere
- **Cancellazione dalla Coda** - da sapere
- **Cancellazione di un Valore** - da sapere
- **Scansione** - fondamentale

Inserimento in Testa

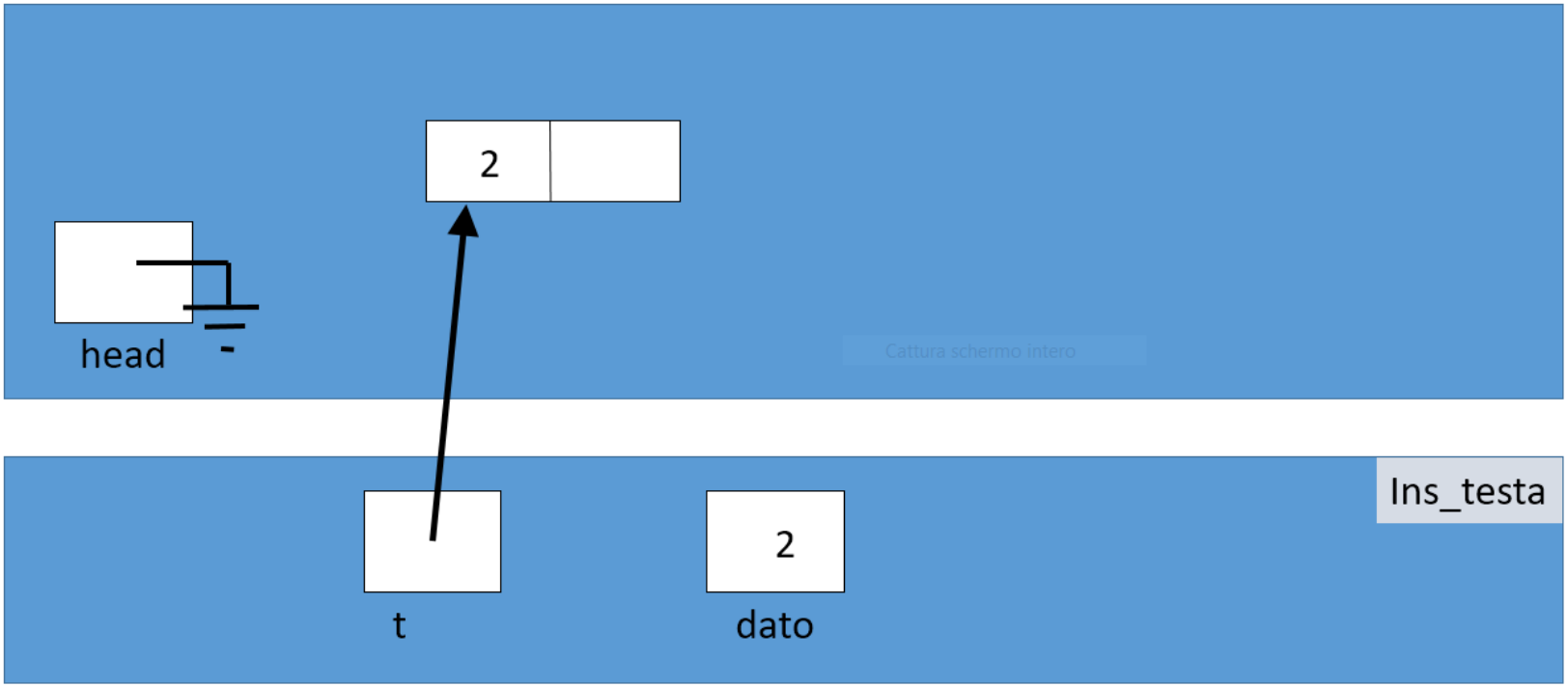
```
int ins_testa(NODO *&head, int dato)
{
    NODO *t;

    t = new NODO;
    if(t == (NODO *)NULL)
    {
        cout << "Memoria Esaurita";
        return 1;
    }

    t->dato = dato;
    t->next= head;
    head = t;

    return 0;
}
```

Inserimento in Testa - Fasi

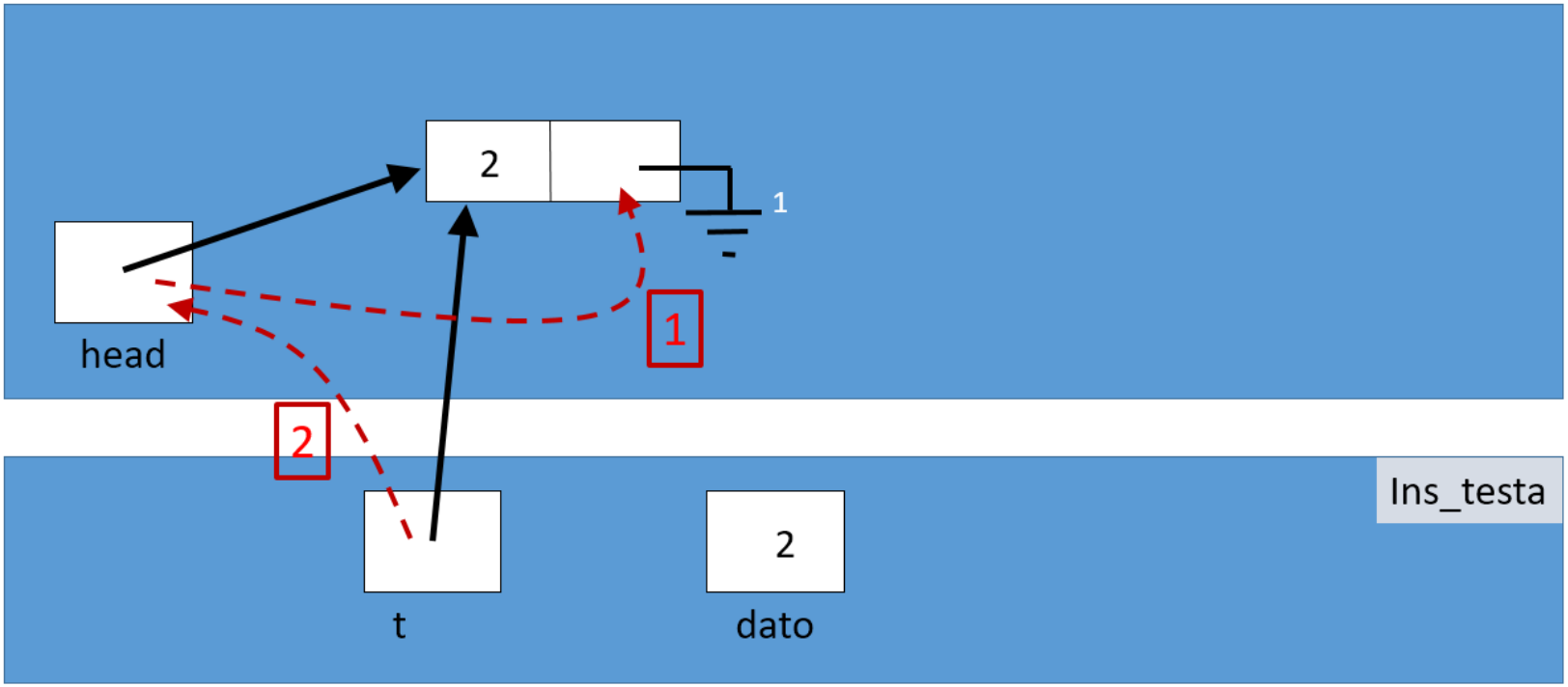
- Allocazione del nodo e verifica dell'allocazione
- Inizializzazione del campo `dato` nel nodo
- Il nuovo nodo punta alla vecchia testa
- Il nuovo nodo diventa la nuova testa

Inserimento in Testa

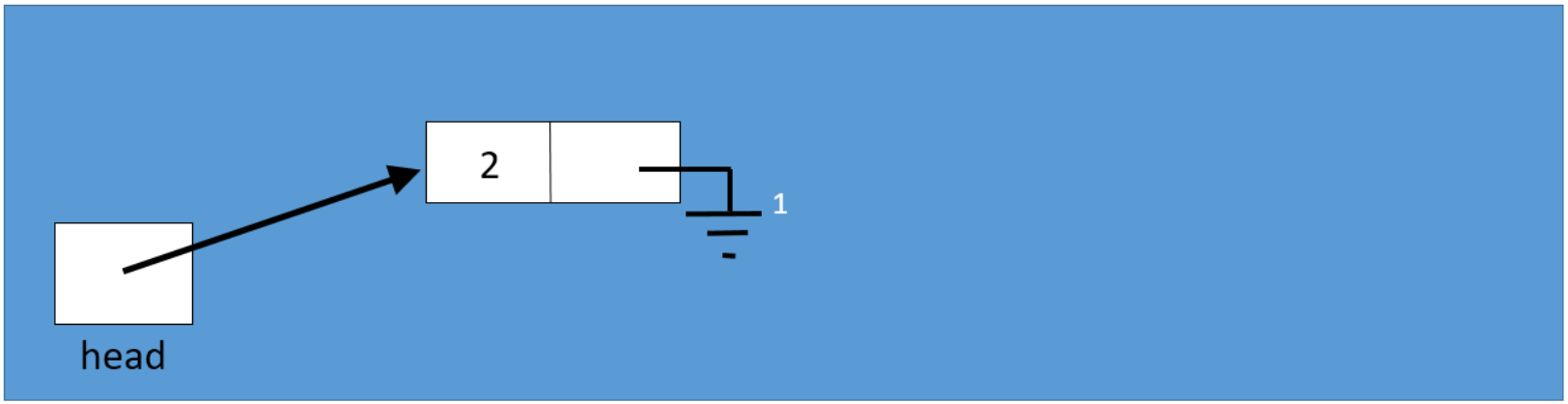
Inserimento in Testa



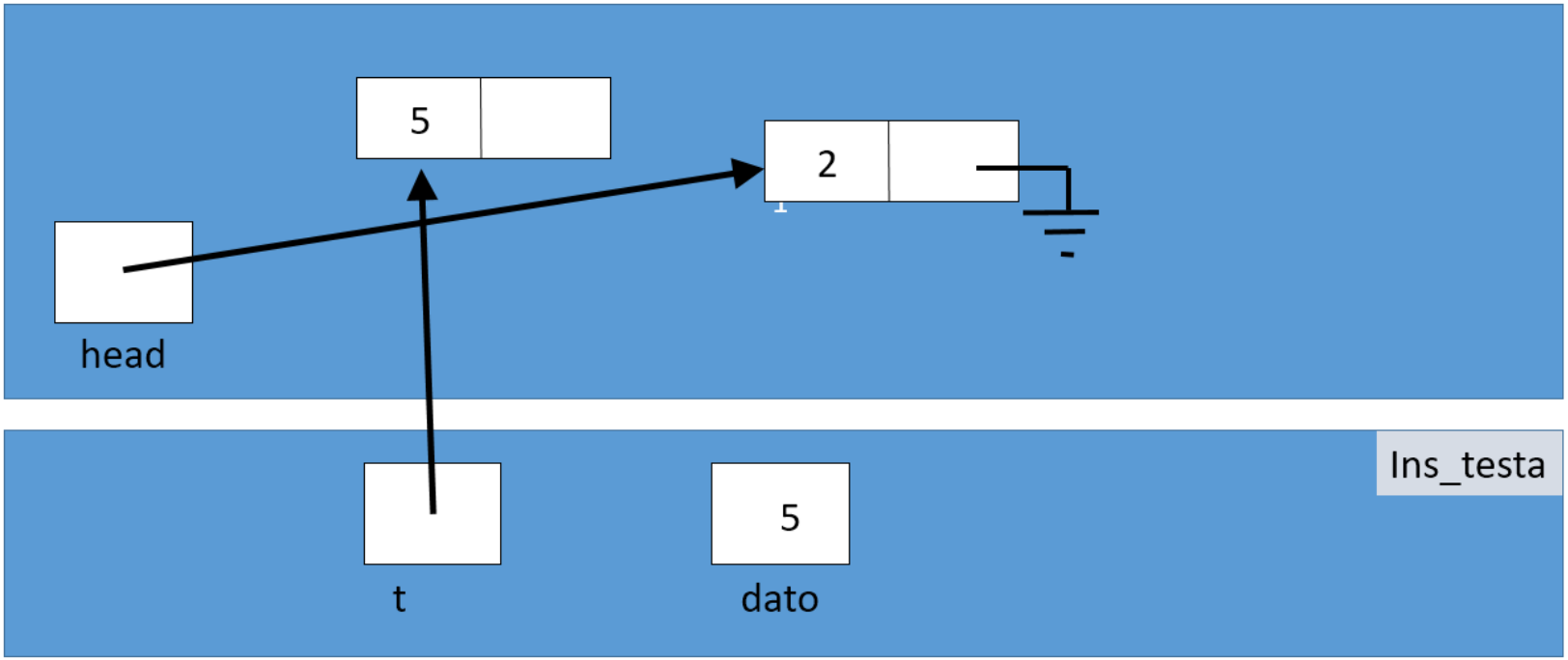
Inserimento in Testa



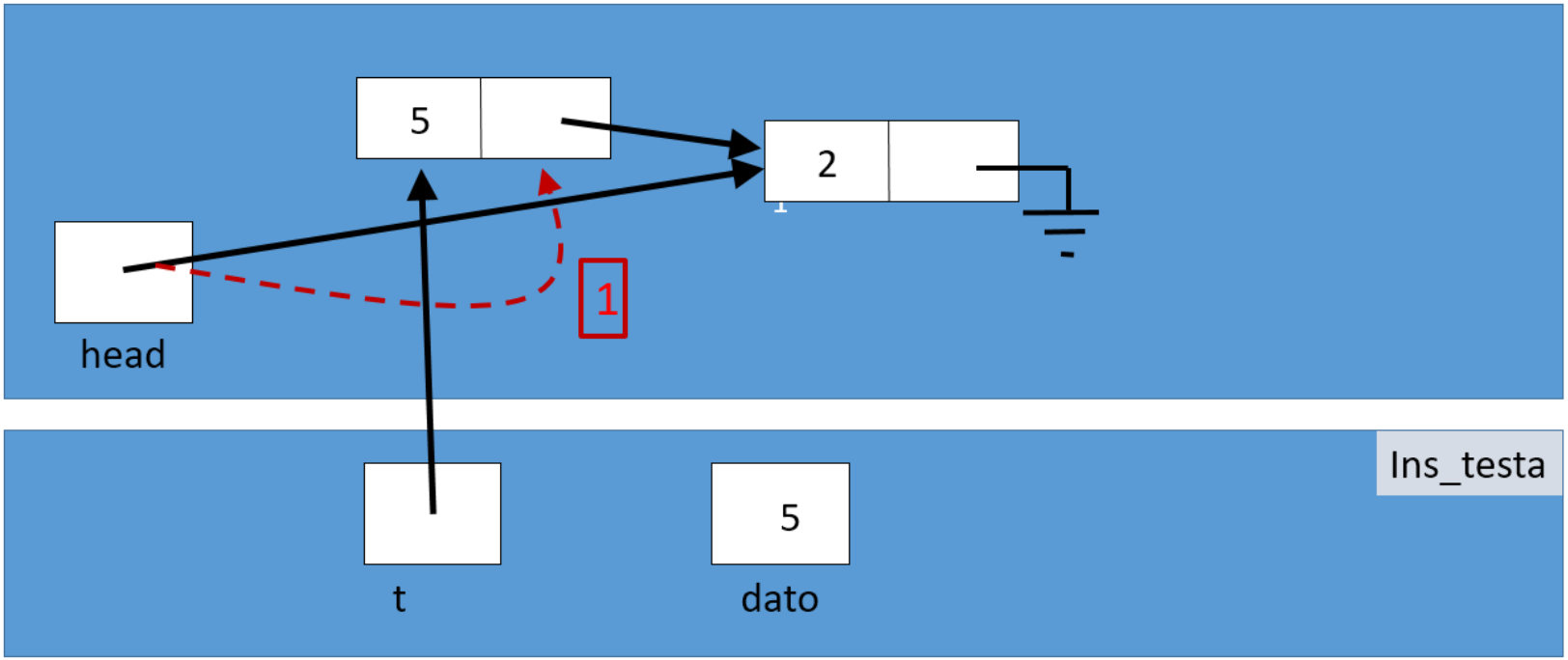
Inserimento in Testa



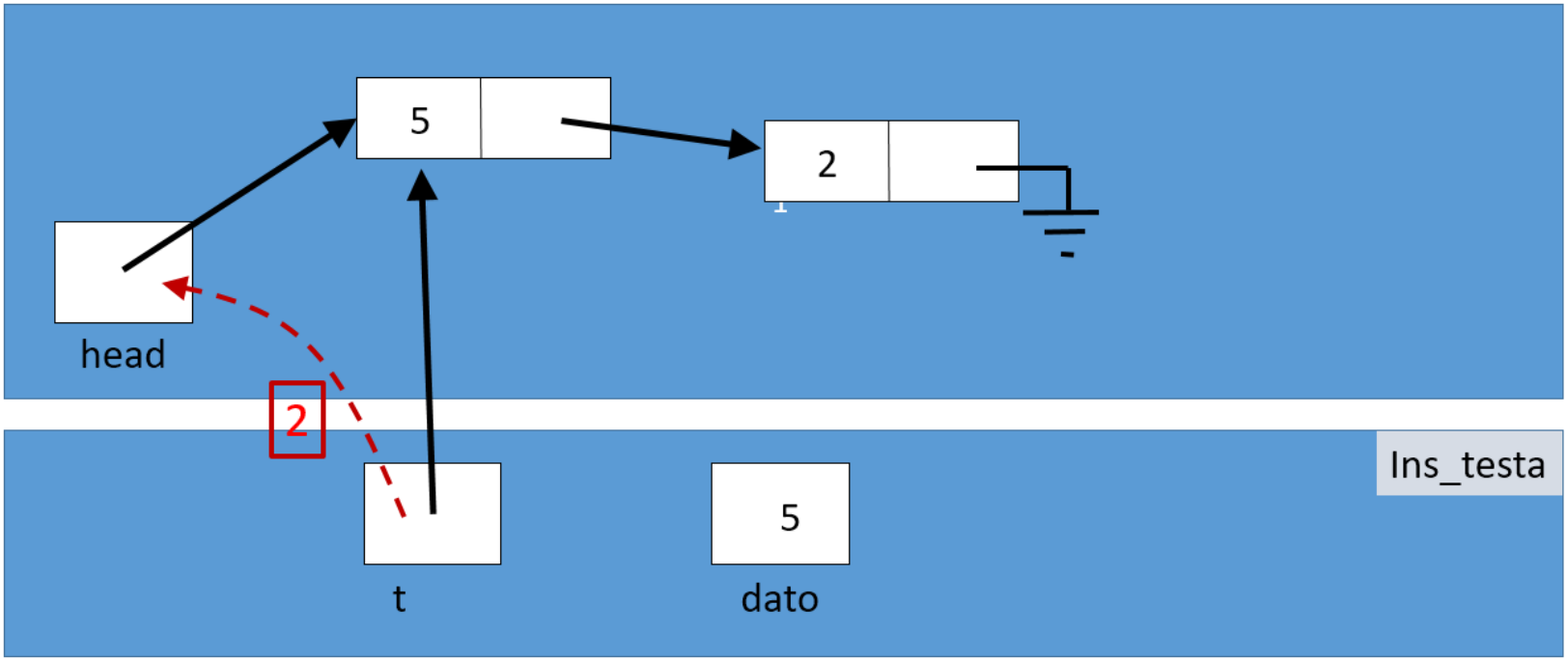
Inserimento in Testa



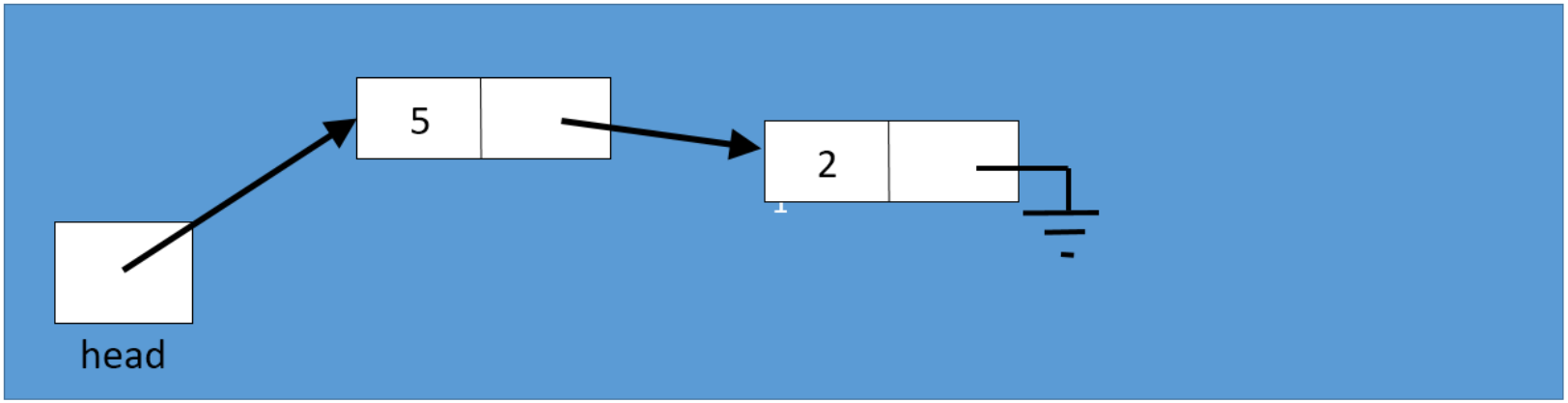
Inserimento in Testa



Inserimento in Testa



Inserimento in Testa



Inserimento in Coda

```
int ins_coda(NODO *&head, int dato)
{
    NODO *t;
    NODO *p;

    t = new NODO;
    if(t == (NODO *)NULL)
    {
        cout << "Memoria Esaurita";
        return 1;
    }
```

Inserimento in Coda

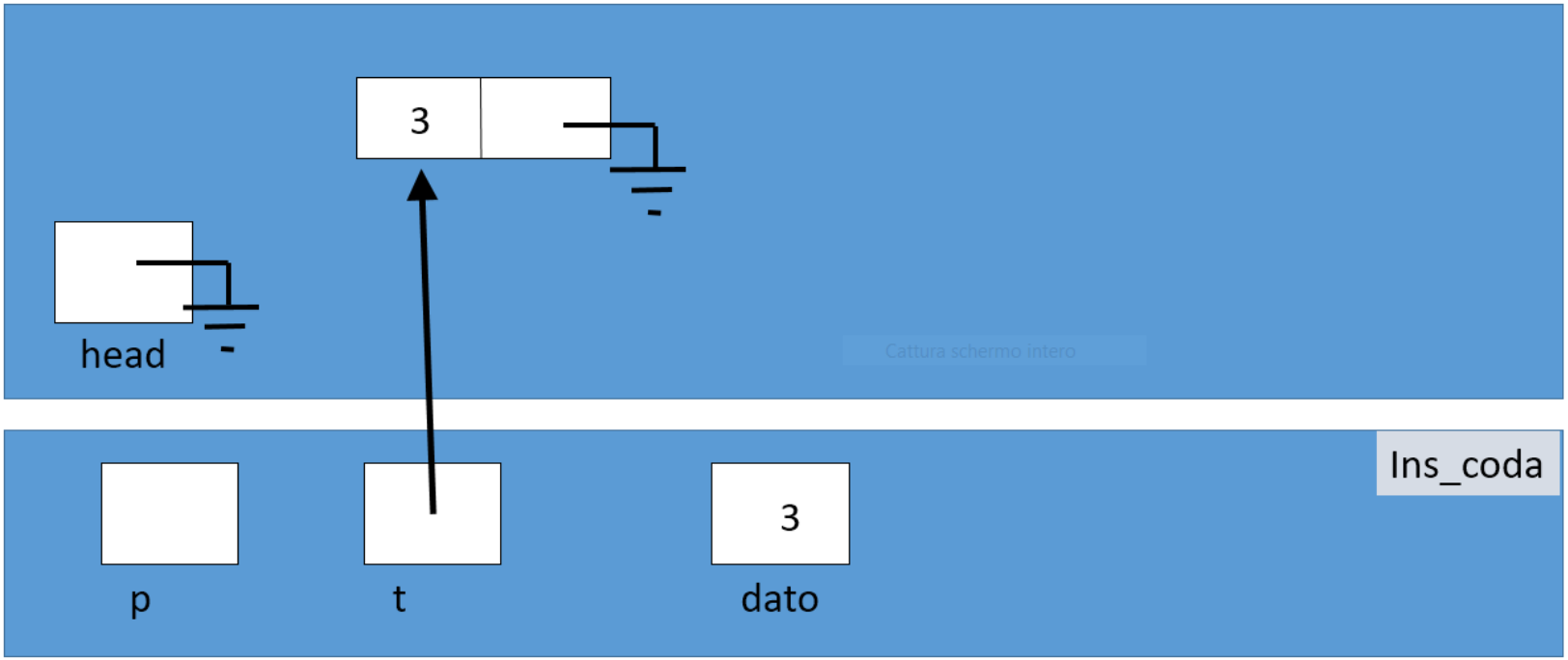
```
        t->dato = dato;
        t->next= NULL;

        if(head == NULL )
        {
            head = t;
            return 0;
        }

        p = head;
        while(p->next != NULL)
            p = p->next;
        p->next = t;

        return 0;
}
```

Inserimento in Coda - Fasi

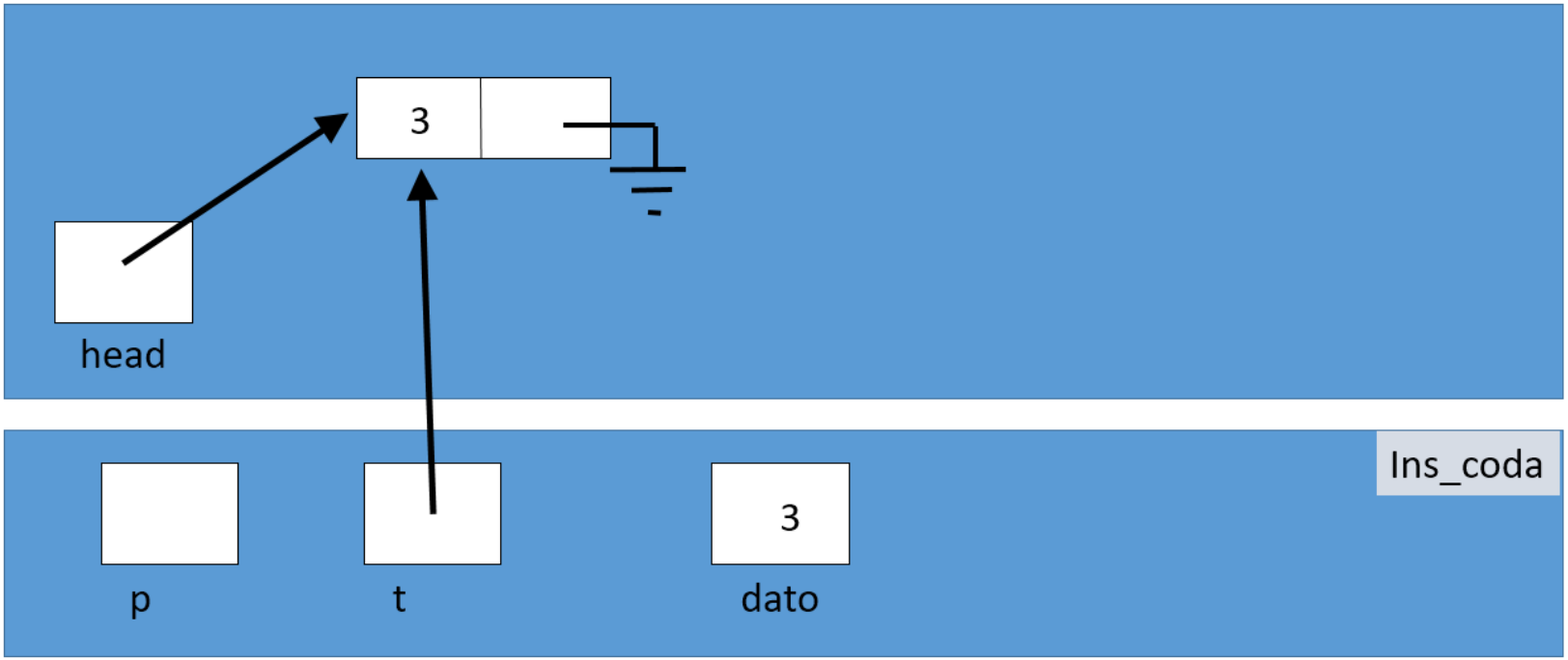
- Allocazione del nodo e verifica dell'allocazione
- Inizializzazione del nodo
- Se la lista è vuota:
  il nuovo nodo è il primo nodo, si inserisce e si termina
- Se la lista contiene un solo nodo:
  il nuovo nodo si appende a questo e si termina
- Altrimenti:
  si cerca la coda (ultimo nodo) e si appende il nuovo nodo.

Inserimento in Coda

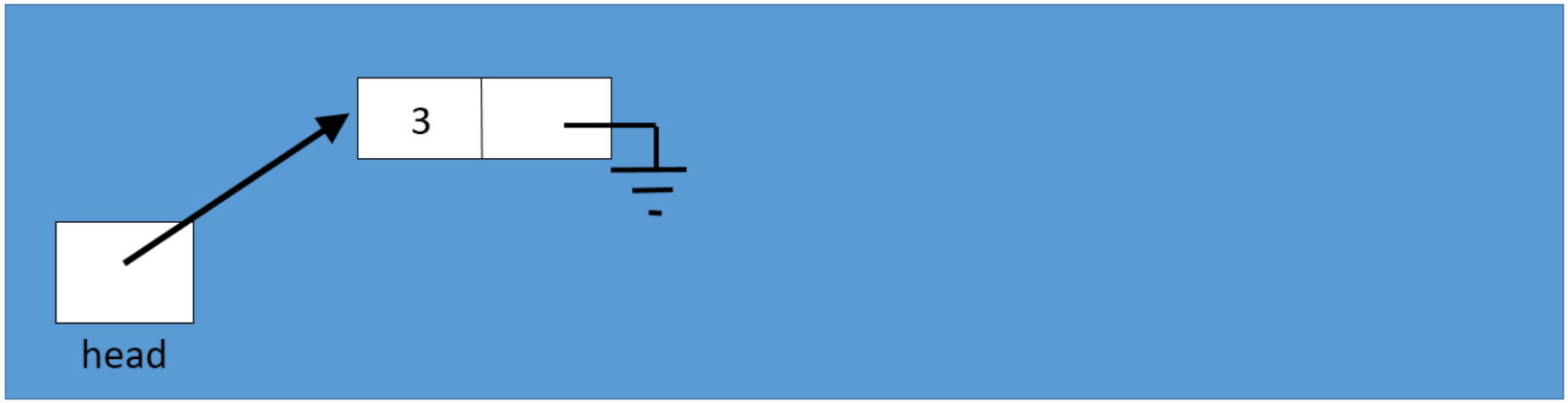
Inserimento in Coda



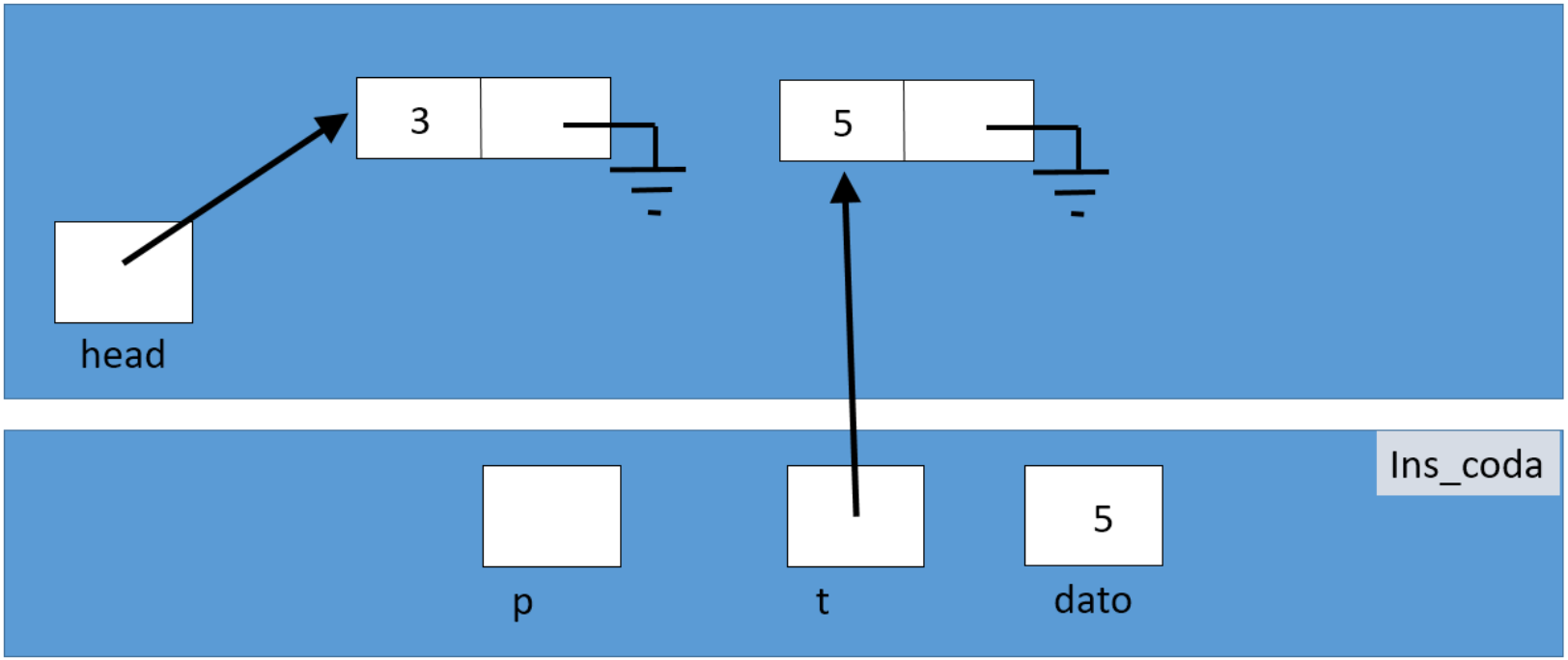
Inserimento in Coda



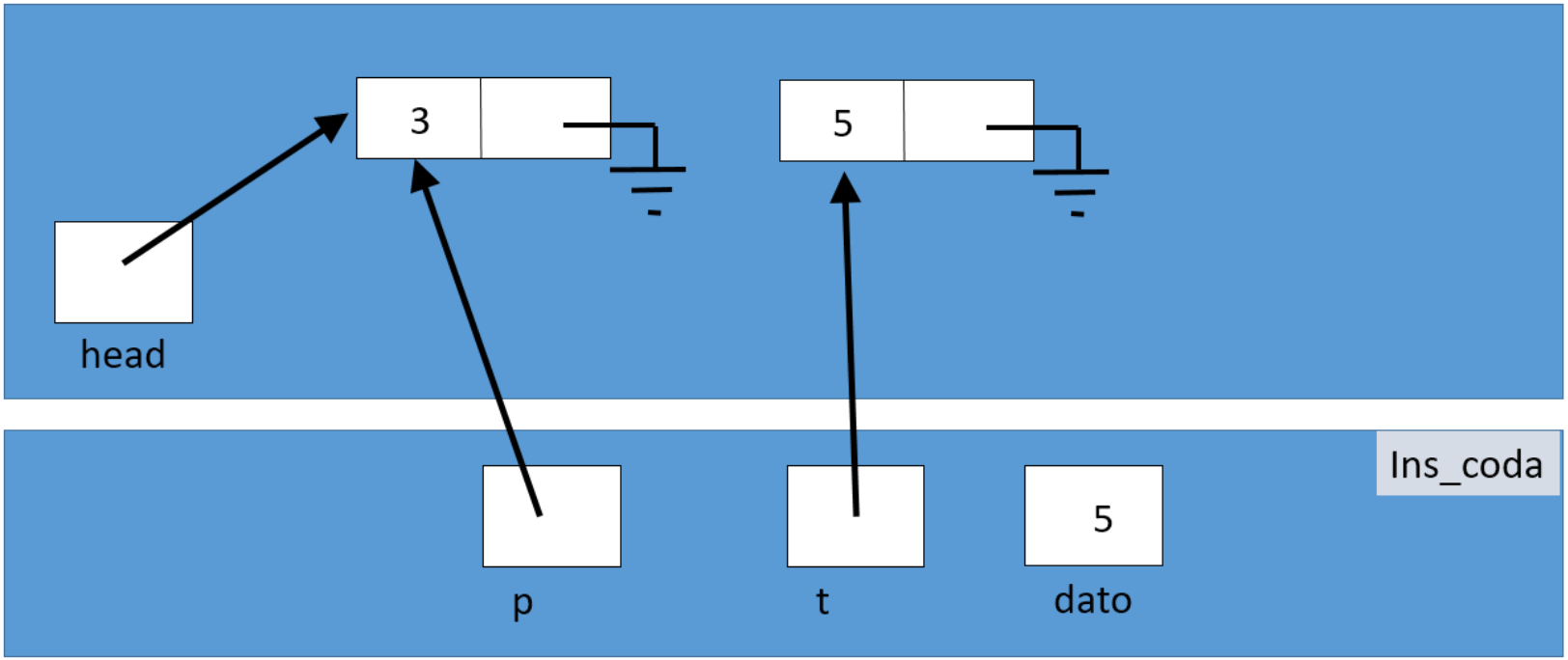
Inserimento in Coda



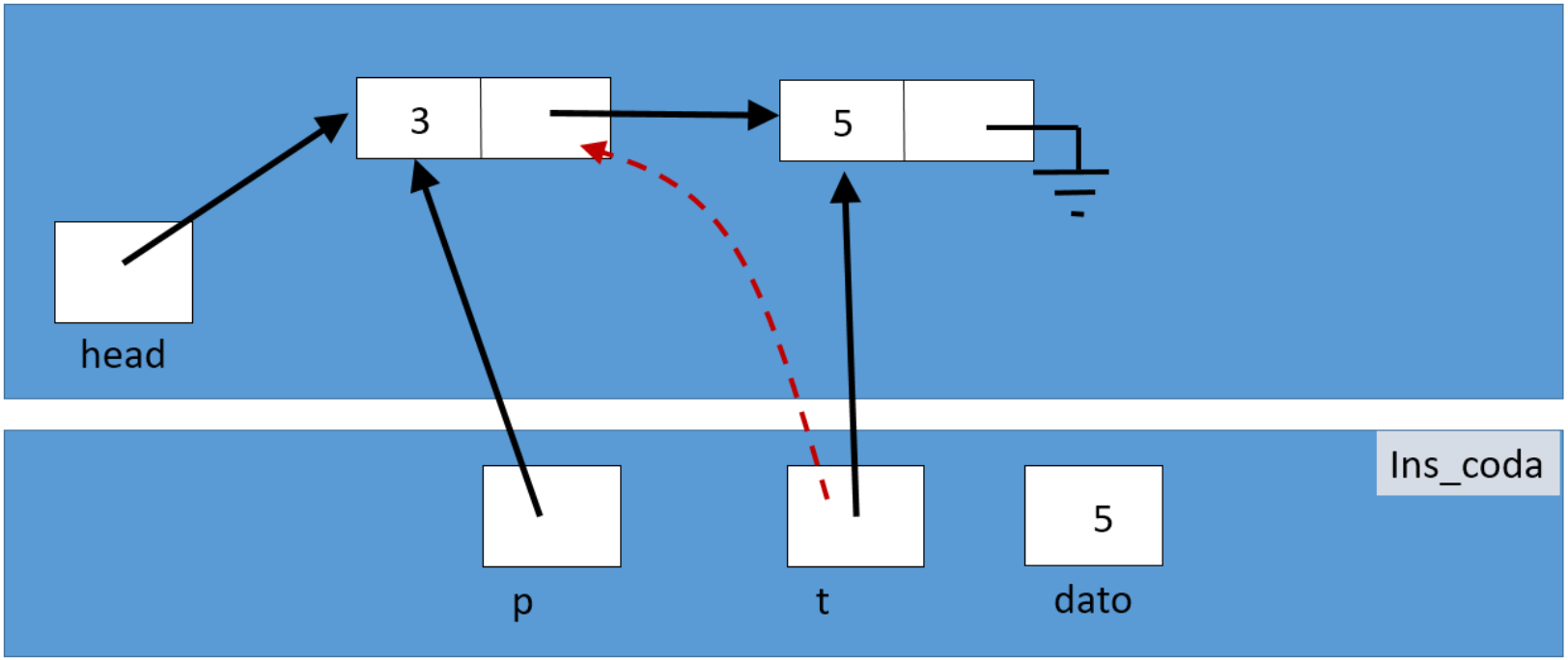
Inserimento in Coda



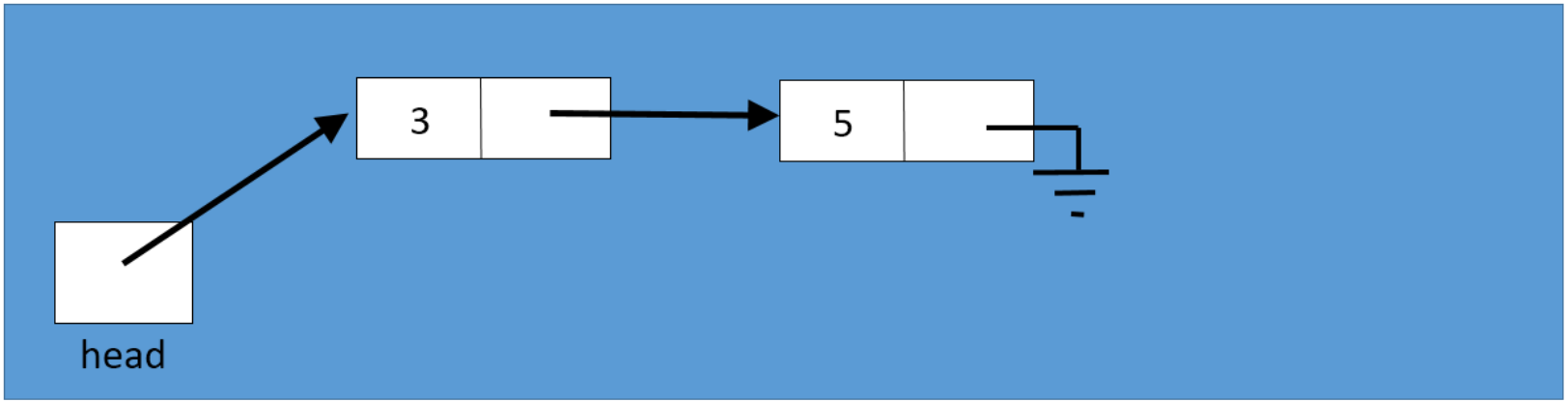
Inserimento in Coda



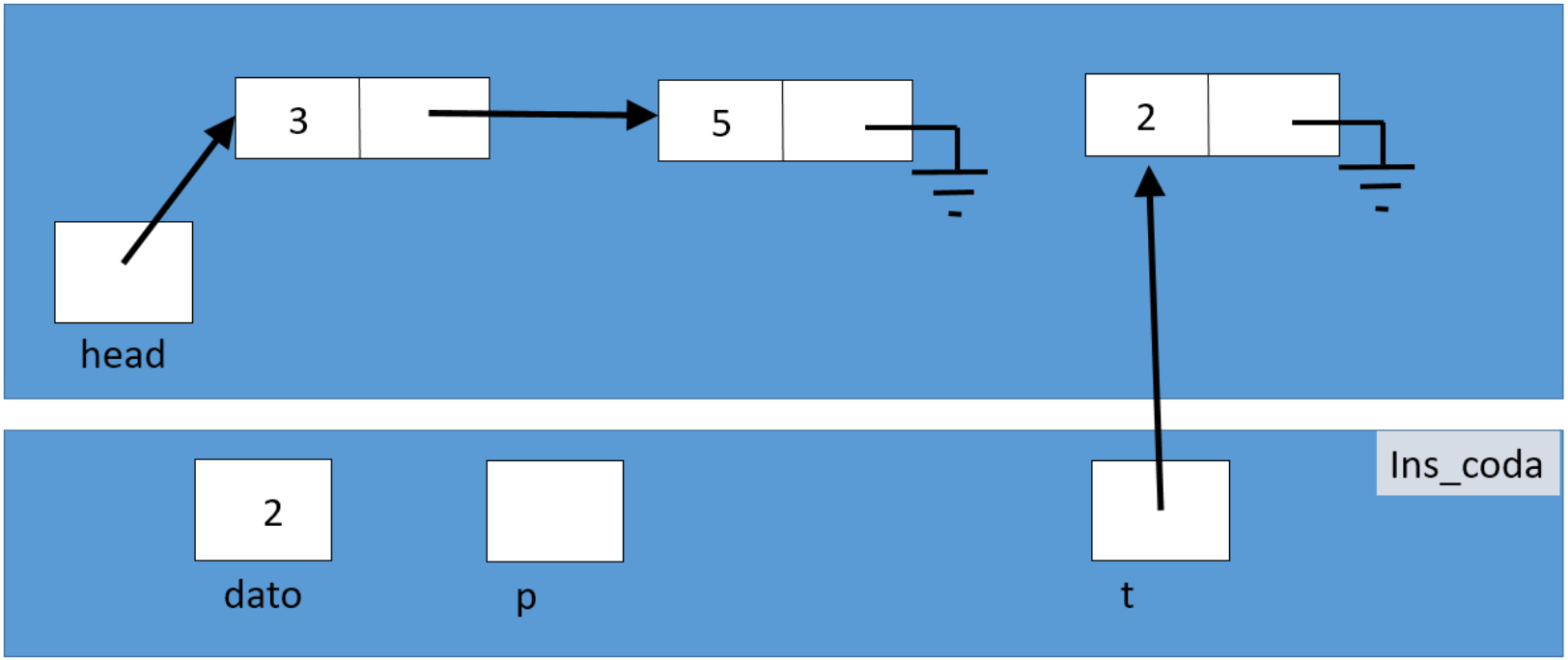
Inserimento in Coda



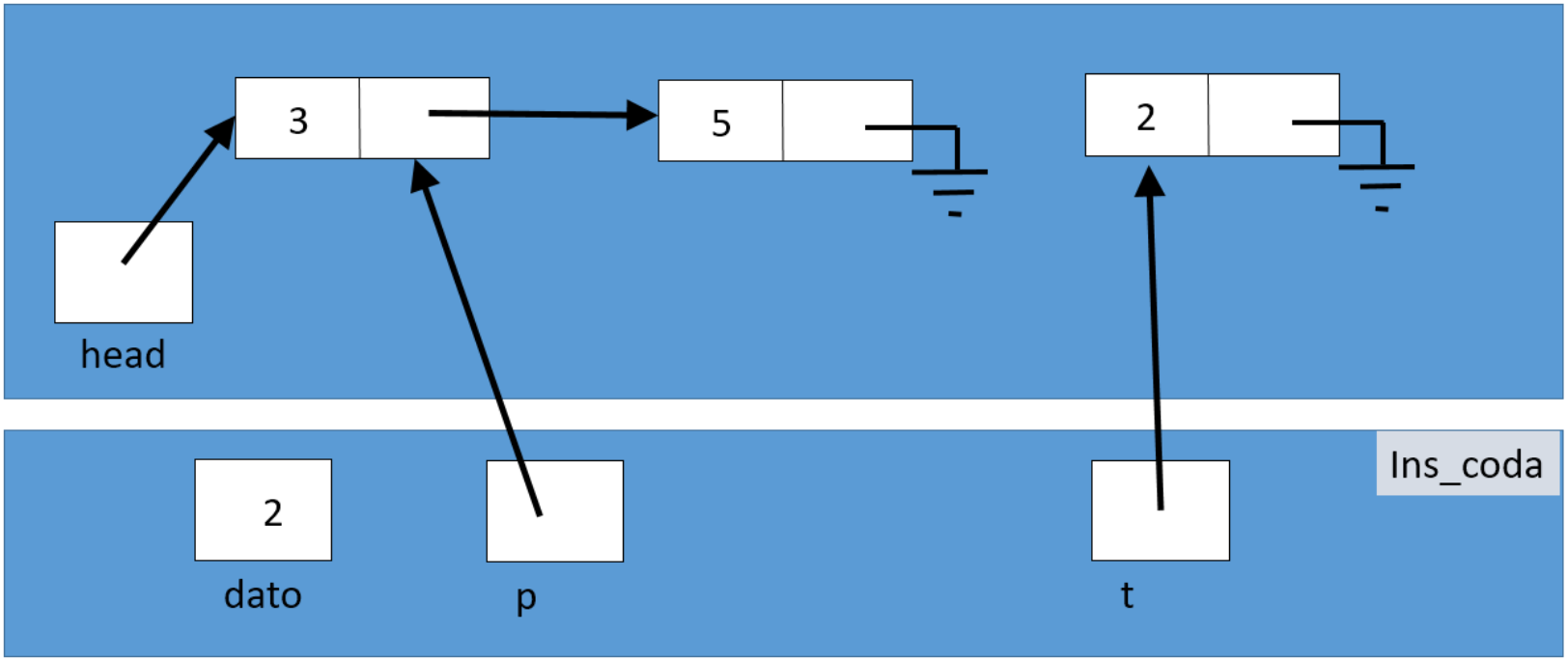
Inserimento in Coda



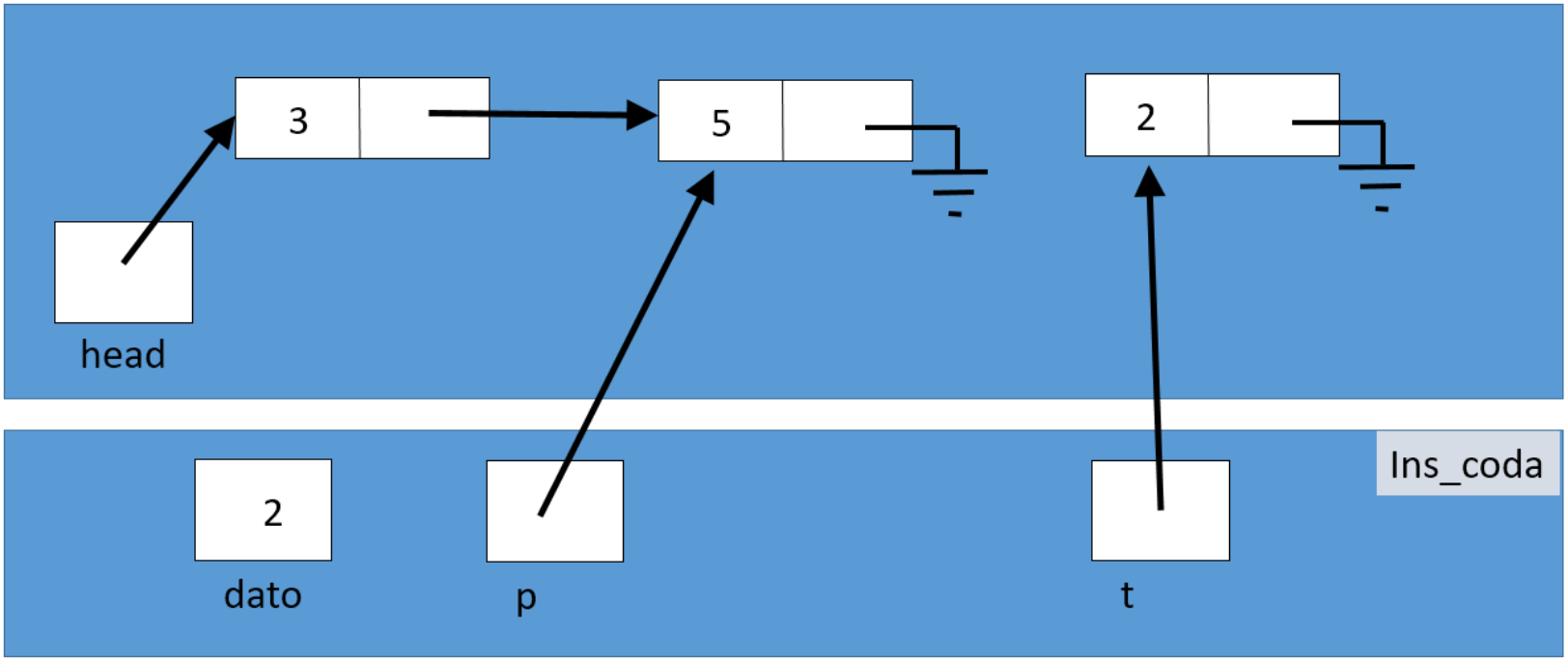
Inserimento in Coda



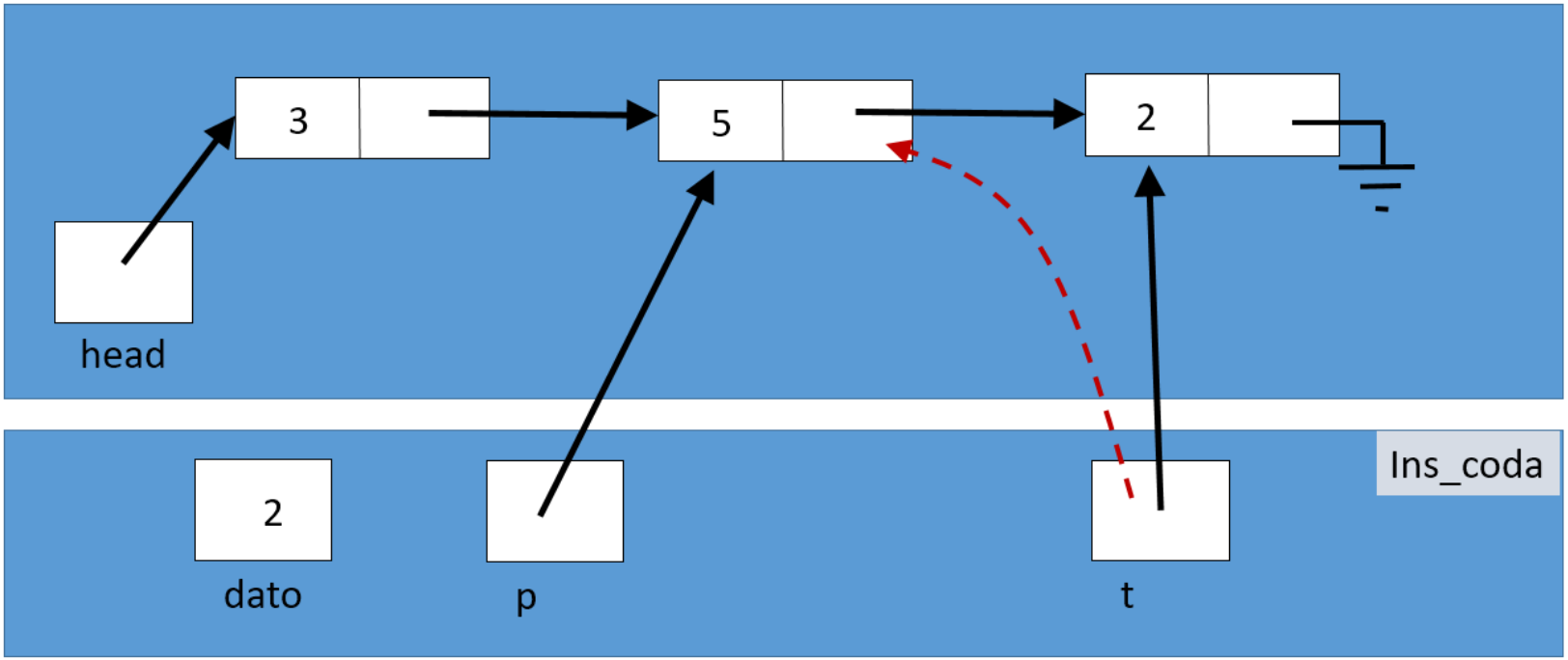
Inserimento in Coda



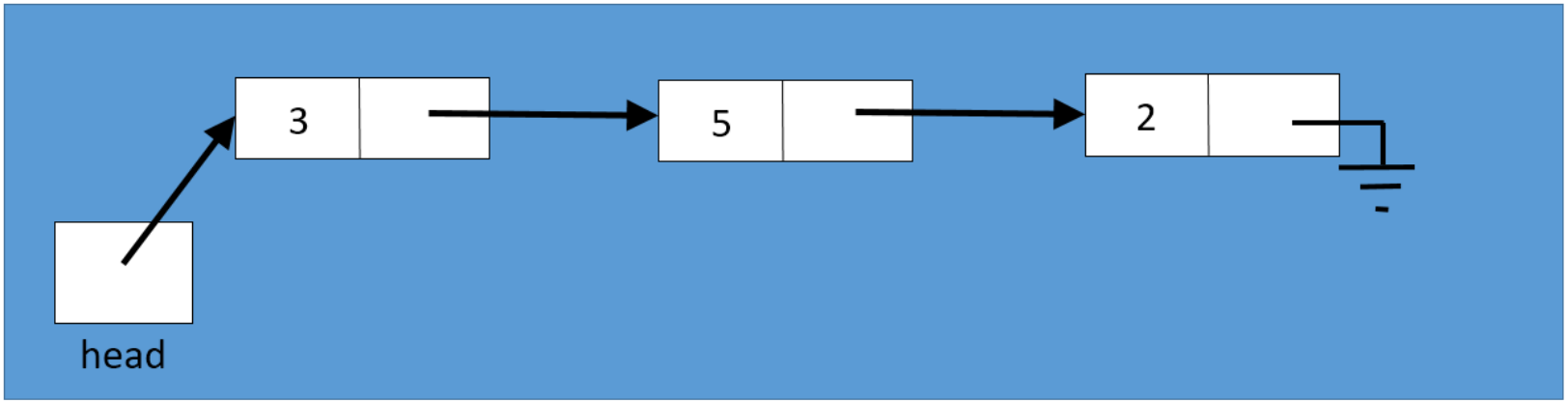
Inserimento in Coda



Inserimento in Coda



Inserimento in Coda



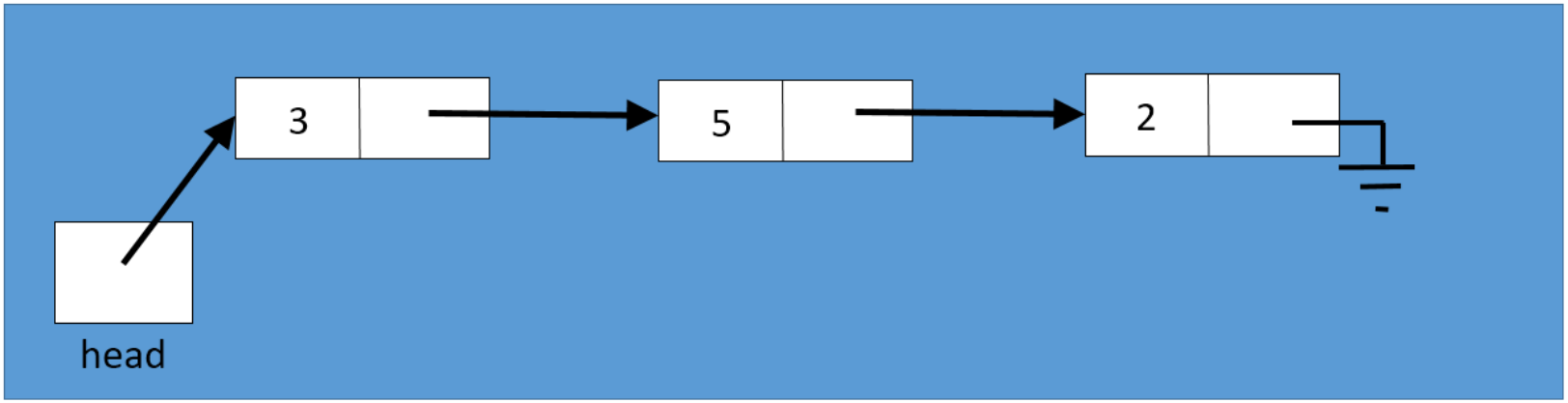
Cancellazione dalla Testa

```
int canc_testa(NODO *&head)
{
    NODO *t;

    if(head == NULL)
       return 1;

    t = head;
    head = t->next;
    delete t;

    return 0;
}
```

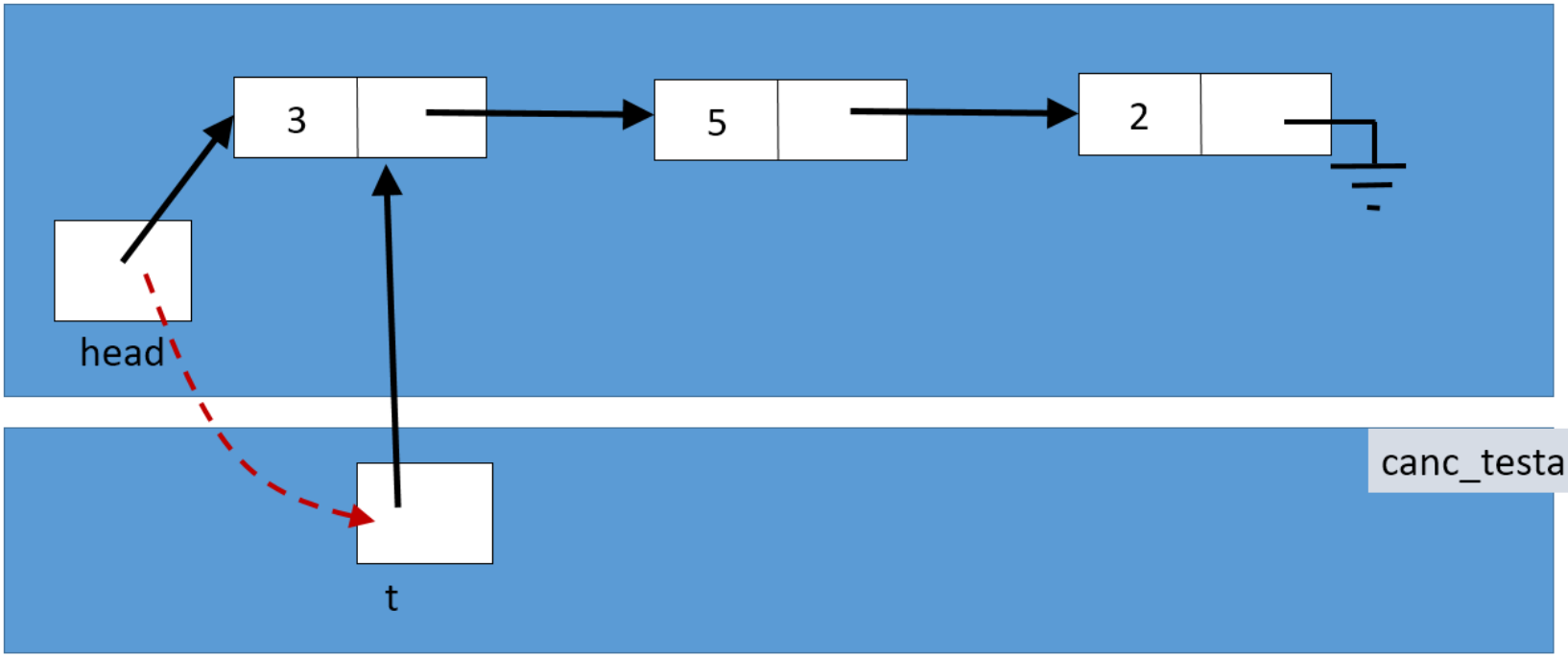
Cancellazione dalla Testa - Fasi

- Se la lista è vuota, si termina
- Si copia l'indirizzo della test a in `t`
- Il secondo nodo diventa la nuova testa
- Si de-alloca la vecchia testa (usando `t`)

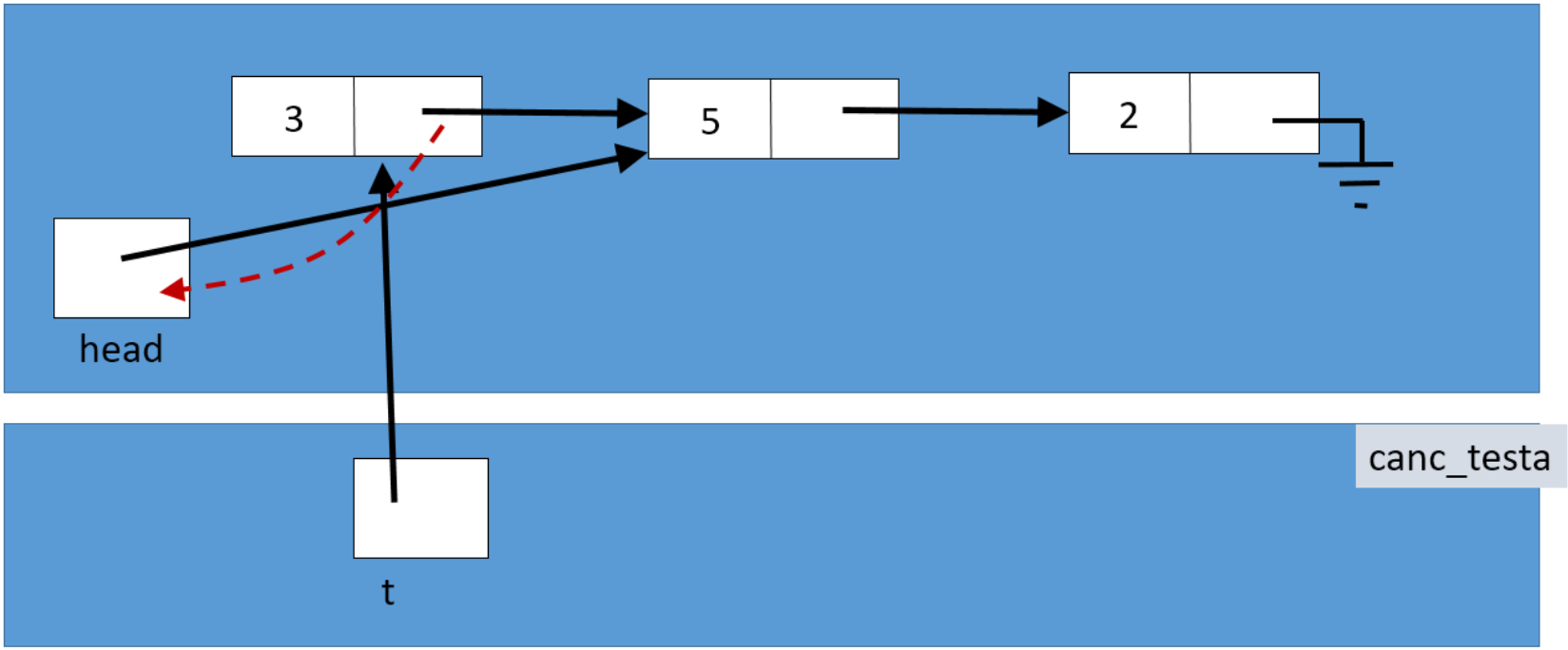
Cancellazione dalla Testa



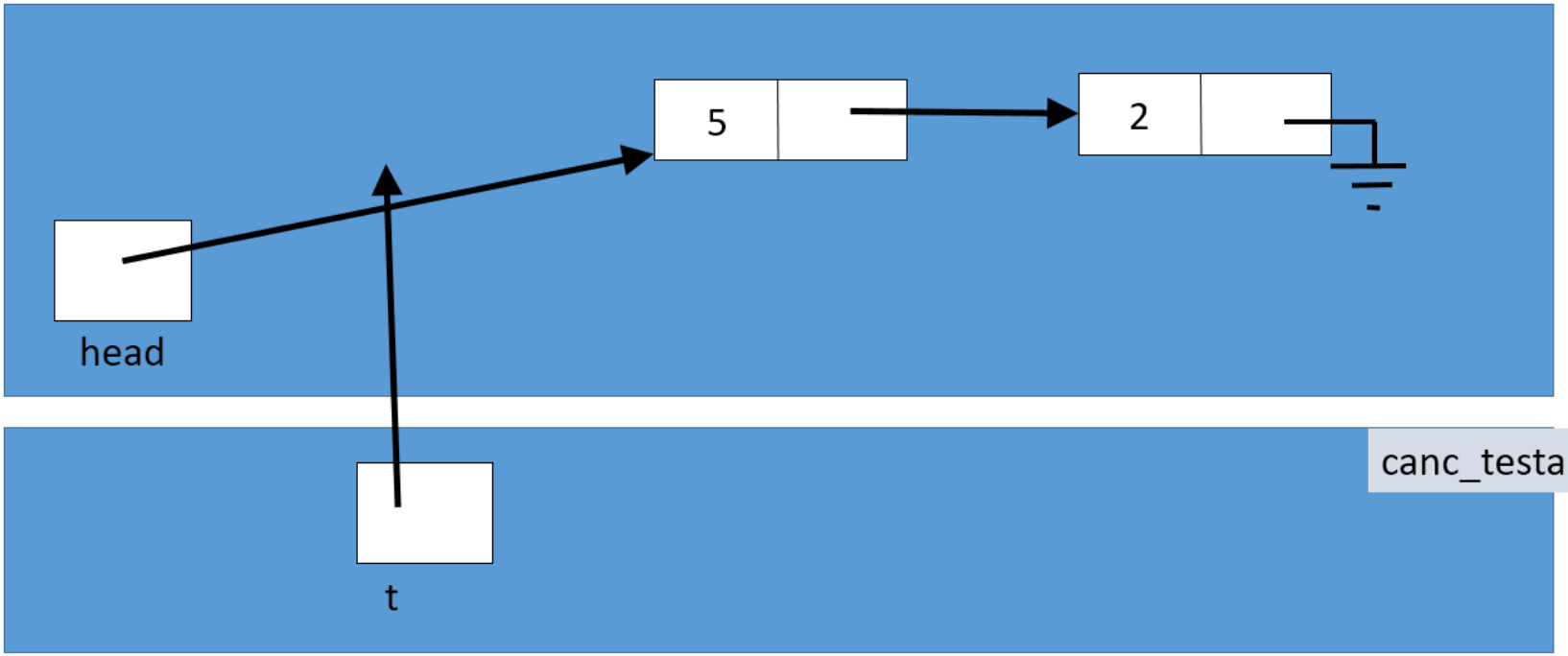
Cancellazione dalla Testa



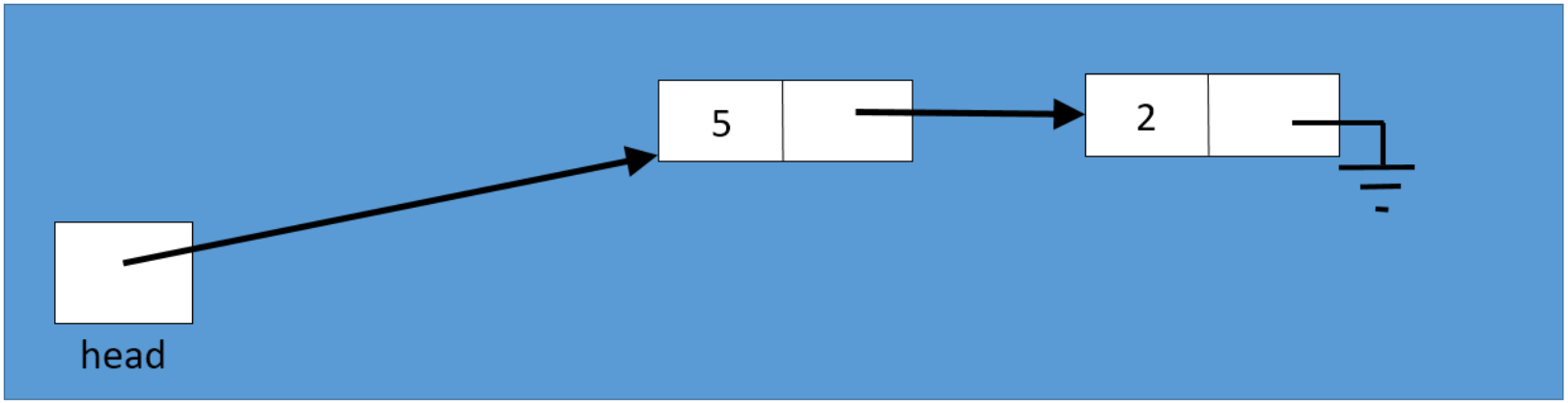
Cancellazione dalla Testa



Cancellazione dalla Testa



Cancellazione dalla Testa



Cancellazione dalla Coda

```
int canc_coda(NODO *&head)
{
    NODO *t;
    NODO *p;
    NODO *prec;

    if(head == NULL)
       return 1;

    if(head->next == NULL)
    {
        t = head;
        head = NULL;
    }
    else
```

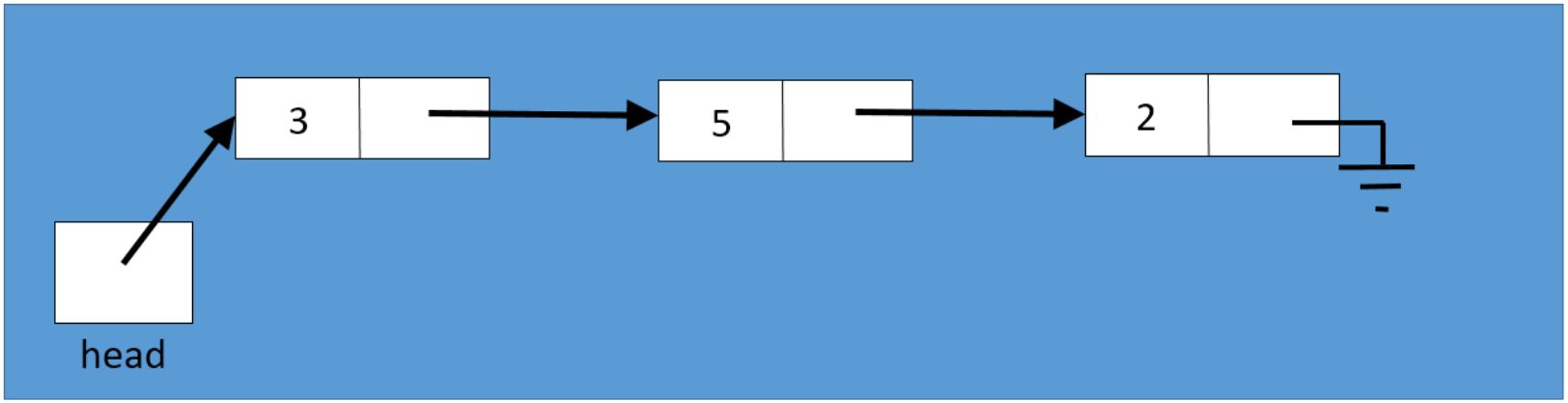
Cancellazione dalla Coda

```
    else
    {
        p = head->next;
        prec = head;
        while(p->next != NULL)
        {
            prec = p;
            p = p->next;
        }
        t = p;
        prec->next = NULL;
    }

    delete t;

    return 0;
}
```

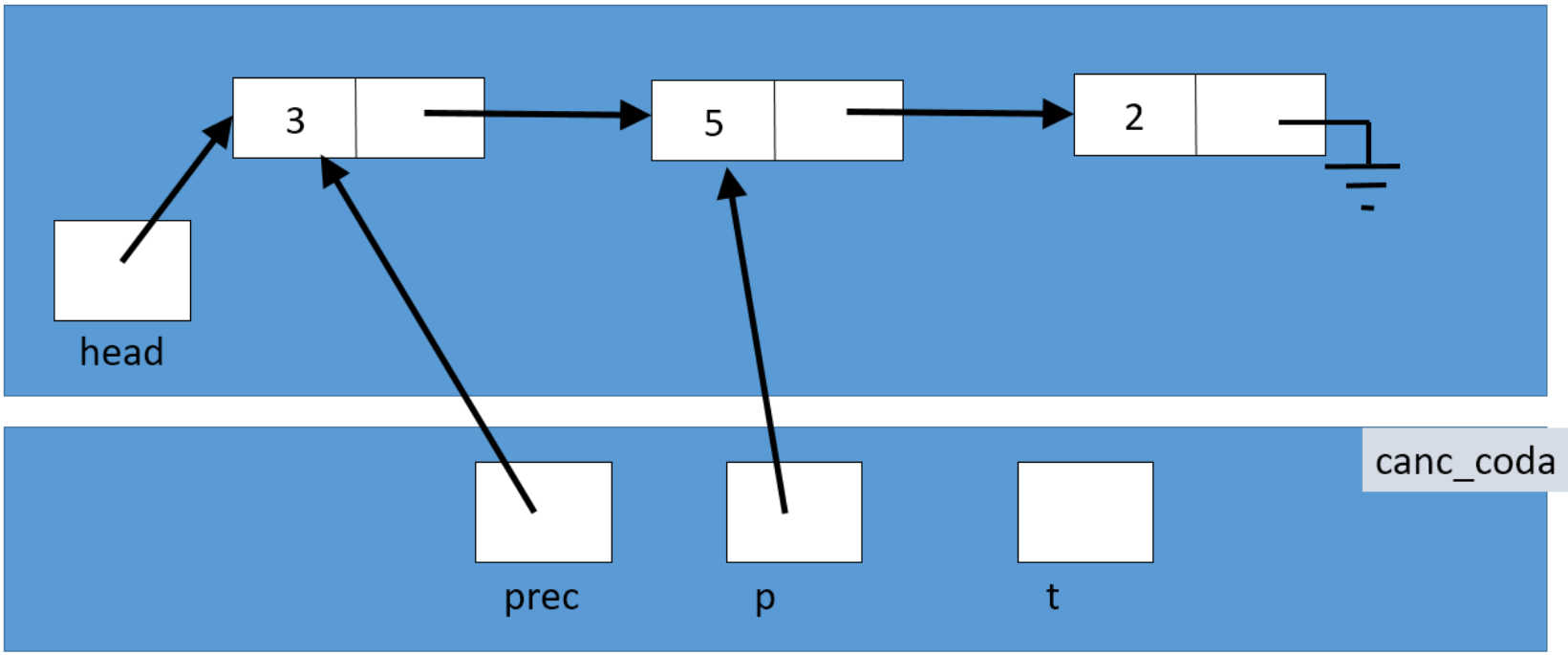
Cancellazione dalla Coda - Fasi

- Se la lista è vuota, si termina
- Se la testa è l'unico nodo:
  - si assegna l'indirizzo a `t`
  - si mette la testa a `NULL`
- altrimenti si deve cercare la coda, mantenendo l'indirizzo del penultimo nodo:
  si assegna a `t` l'indirizzo della coda e si assegna il valore `NULL` al campo `next` del penultimo nodo
- Si de-alloca il nodo puntato da `t`

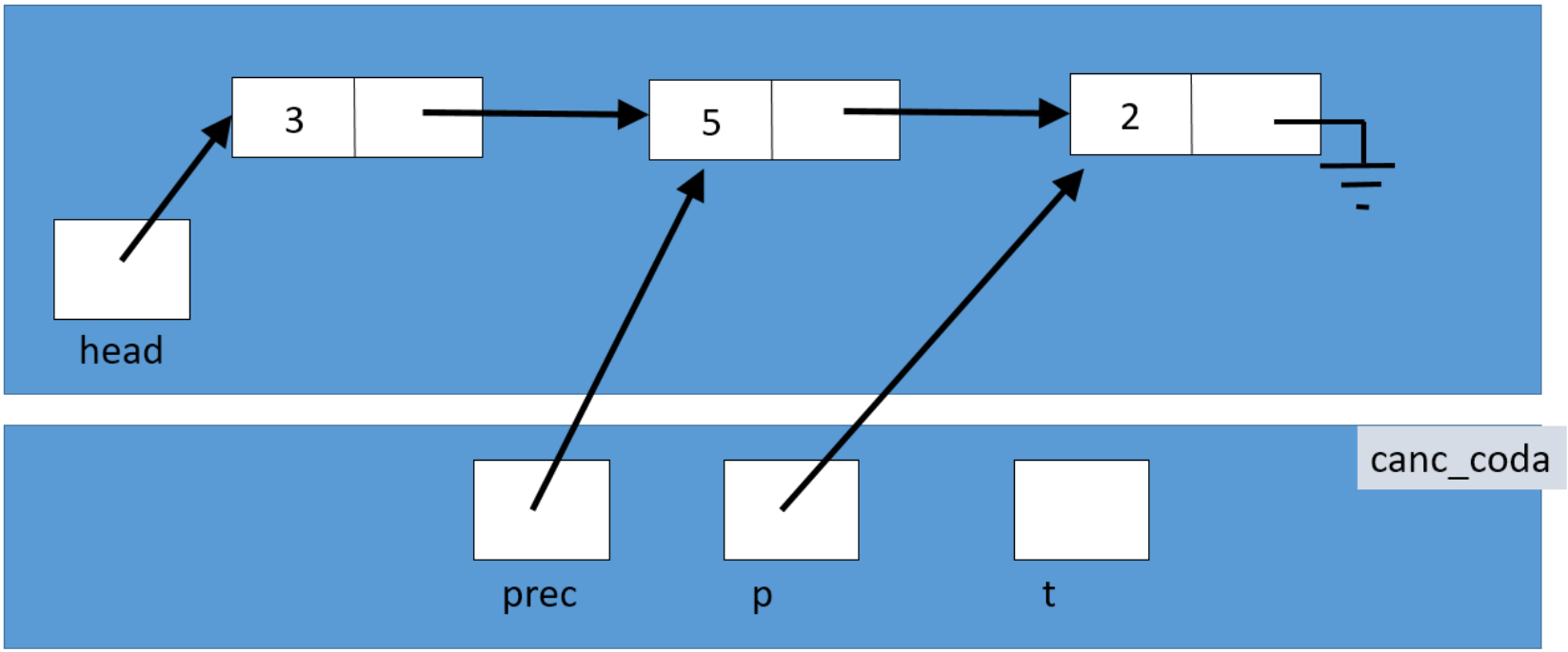
Cancellazione dalla Coda



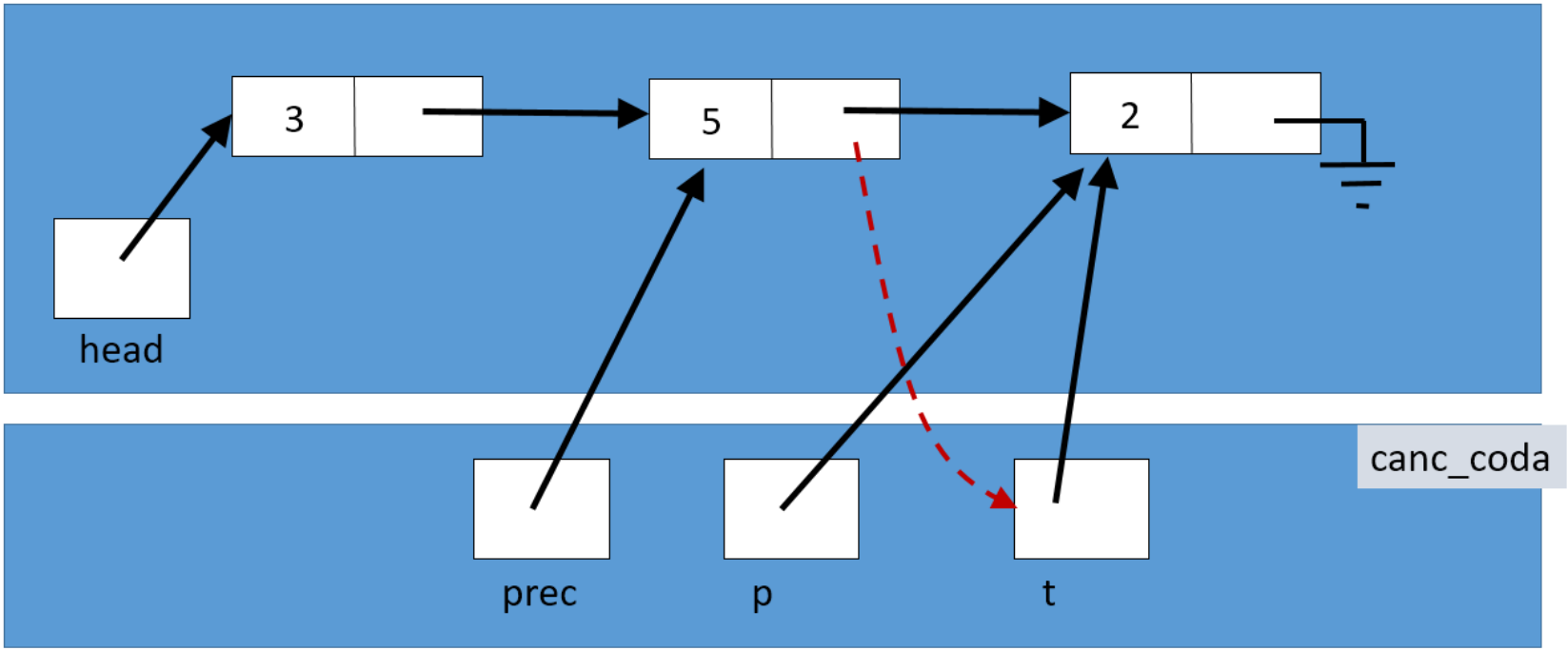
Cancellazione dalla Coda



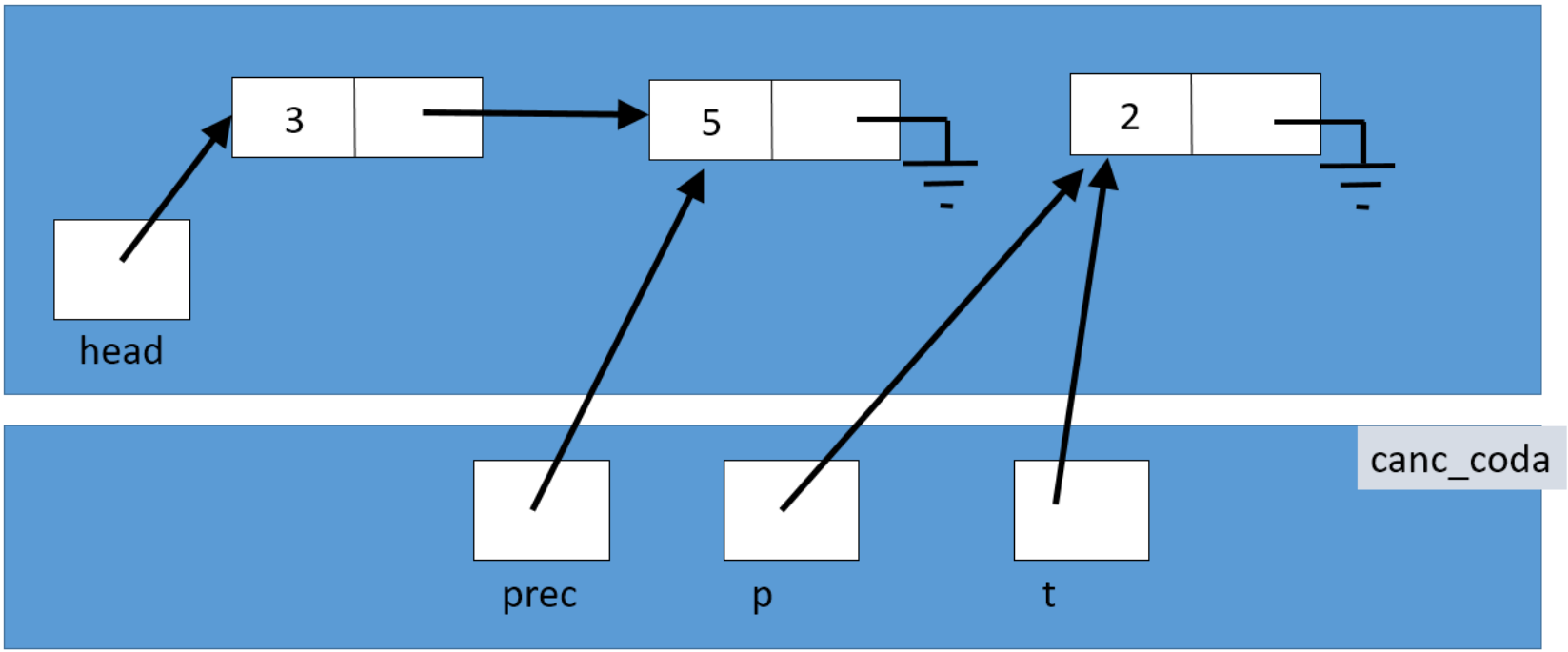
Cancellazione dalla Coda



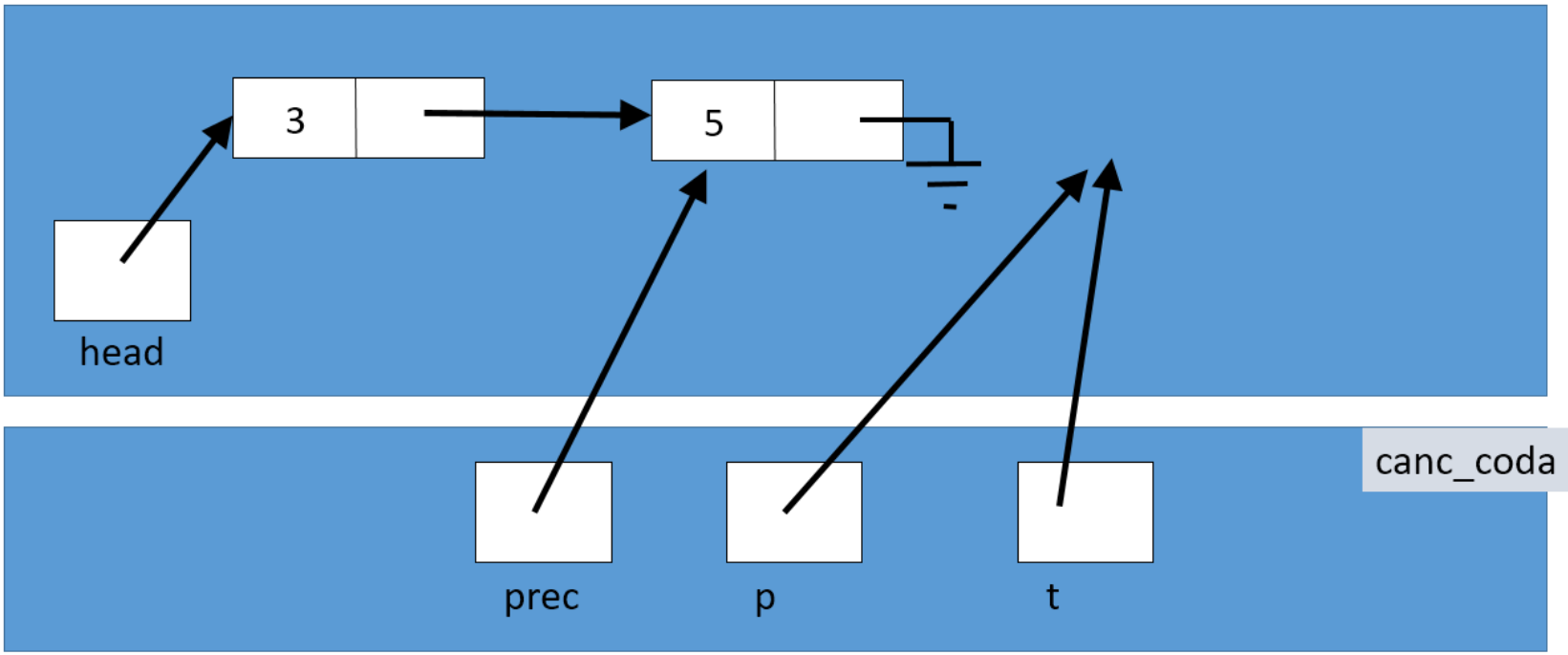
Cancellazione dalla Coda



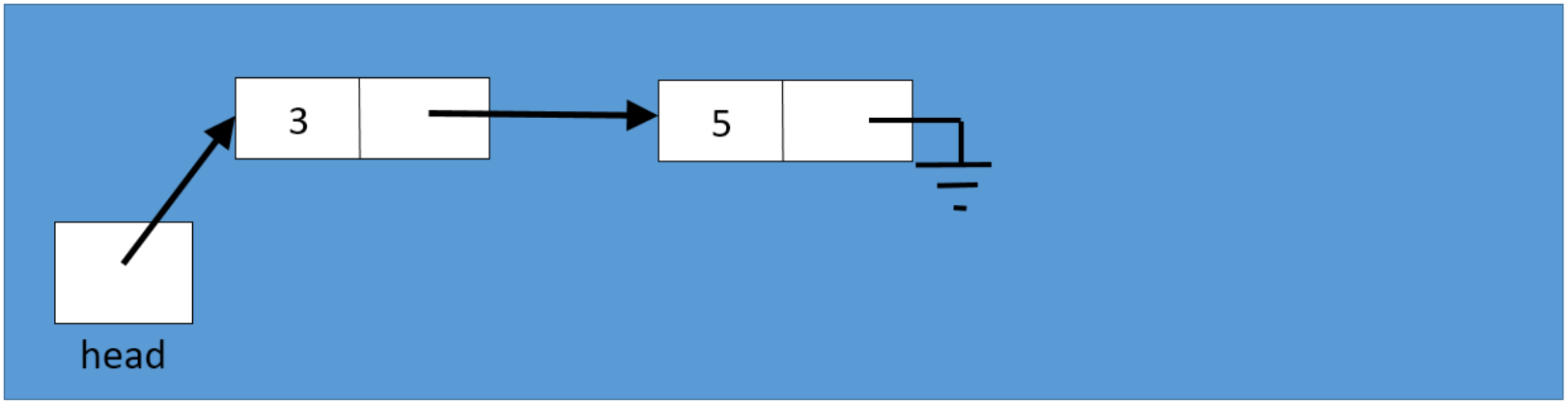
Cancellazione dalla Coda



Cancellazione dalla Coda



Cancellazione dalla Coda



Cancellazione di un vValore

```
int canc_valore(NODO *&head, int dato)
{
    NODO *t;
    NODO *p;
    NODO *prec;

    if(head == NULL)
        return 1;
    if(head->dato == dato)
    {
        t = head;
        head=head->next;
        delete t;
        return 0;
    }
```

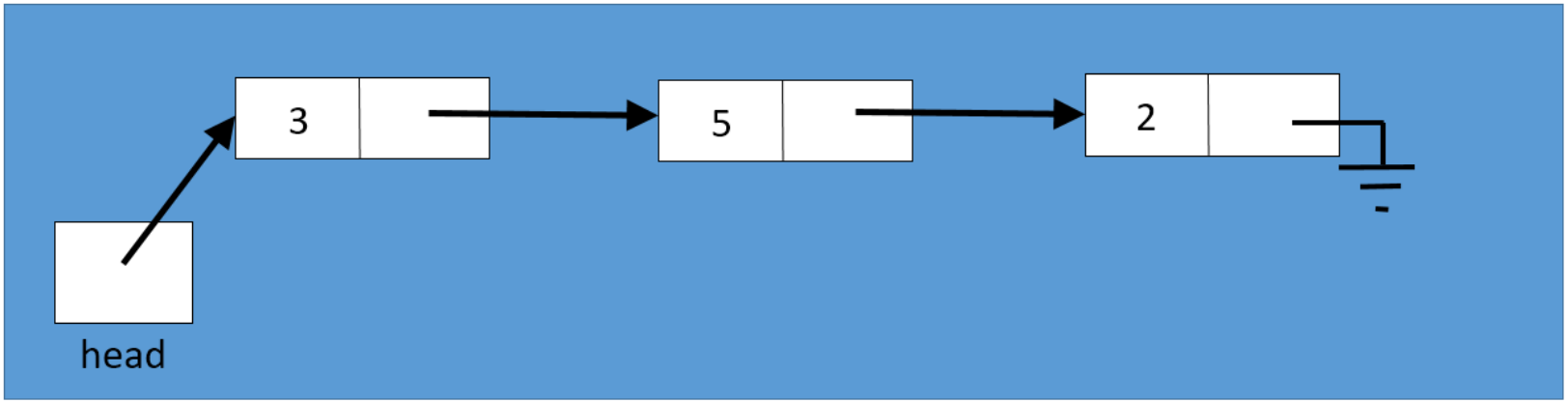
Cancellazione di un vValore

```
p = head->next;
prec = head;
while(p!=NULL)
{
    if(p->dato == dato)
    {
        t = p;
        prec->next = p->next;
        delete t;
        return 0;
    }
    prec = p;
    p = p->next;
}
return 1;
}
```

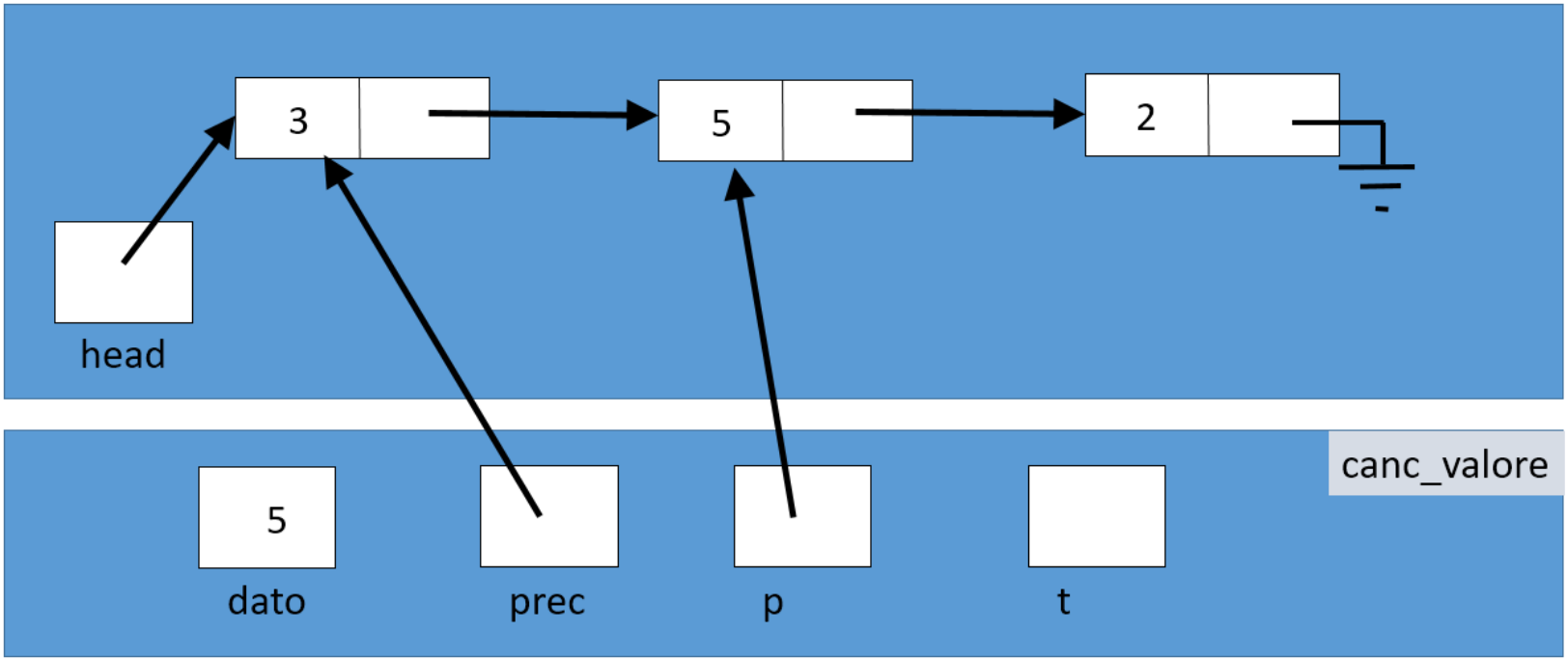
Cancellazione di un Valore - Fasi

- Se la lista è vuota, si termina
- Se la testa contiene il valore cercato:
  si procede con una cancellazione dalla testa e si termina
- altrimenti si deve cercare il nodo che contiene il valore cercato: si scandisce la lista con la coppia di puntatori `p` e `prec`
- se si trova il valore, si fa puntare il precedente al nodo puntato dal nodo trovato e si de-alloca il nodo trovato
- Se il valore non viene trovato, si termina senza fare niente.

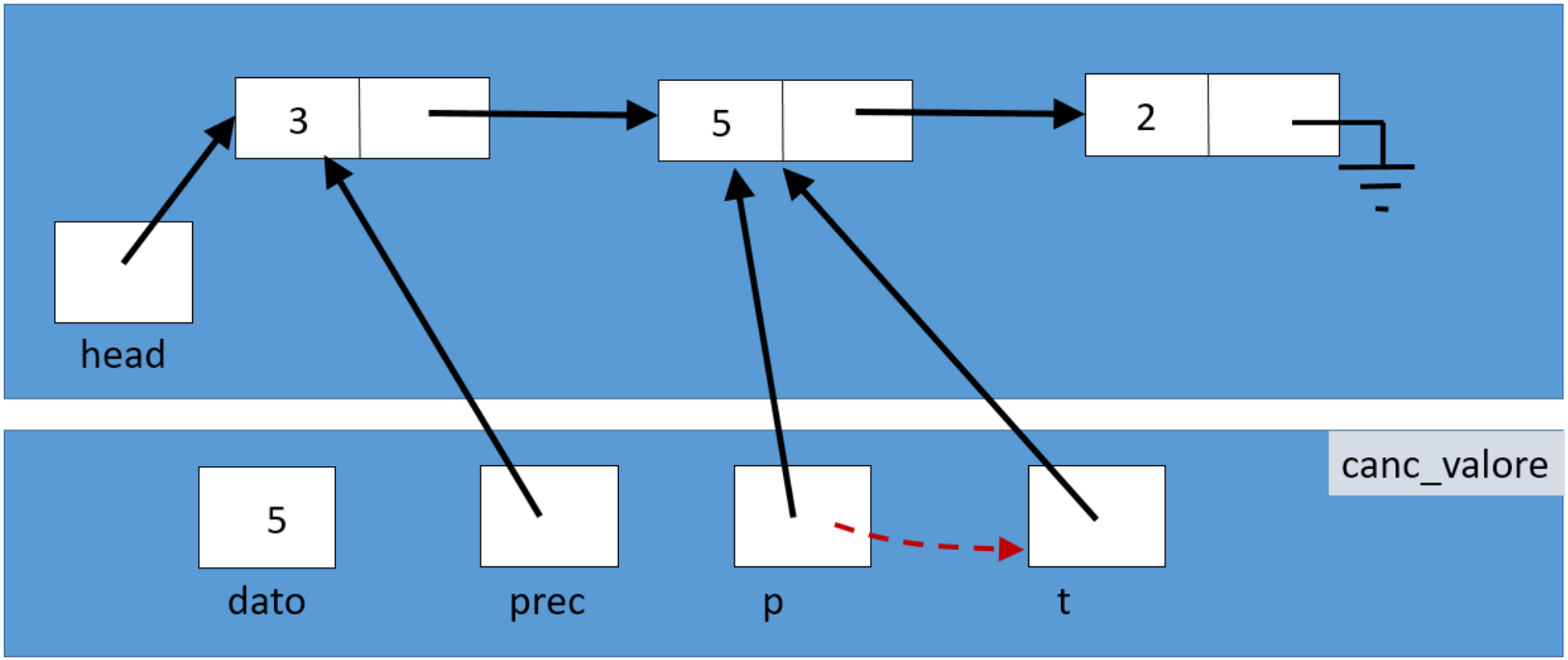
Cancellazione di un Valore: 5



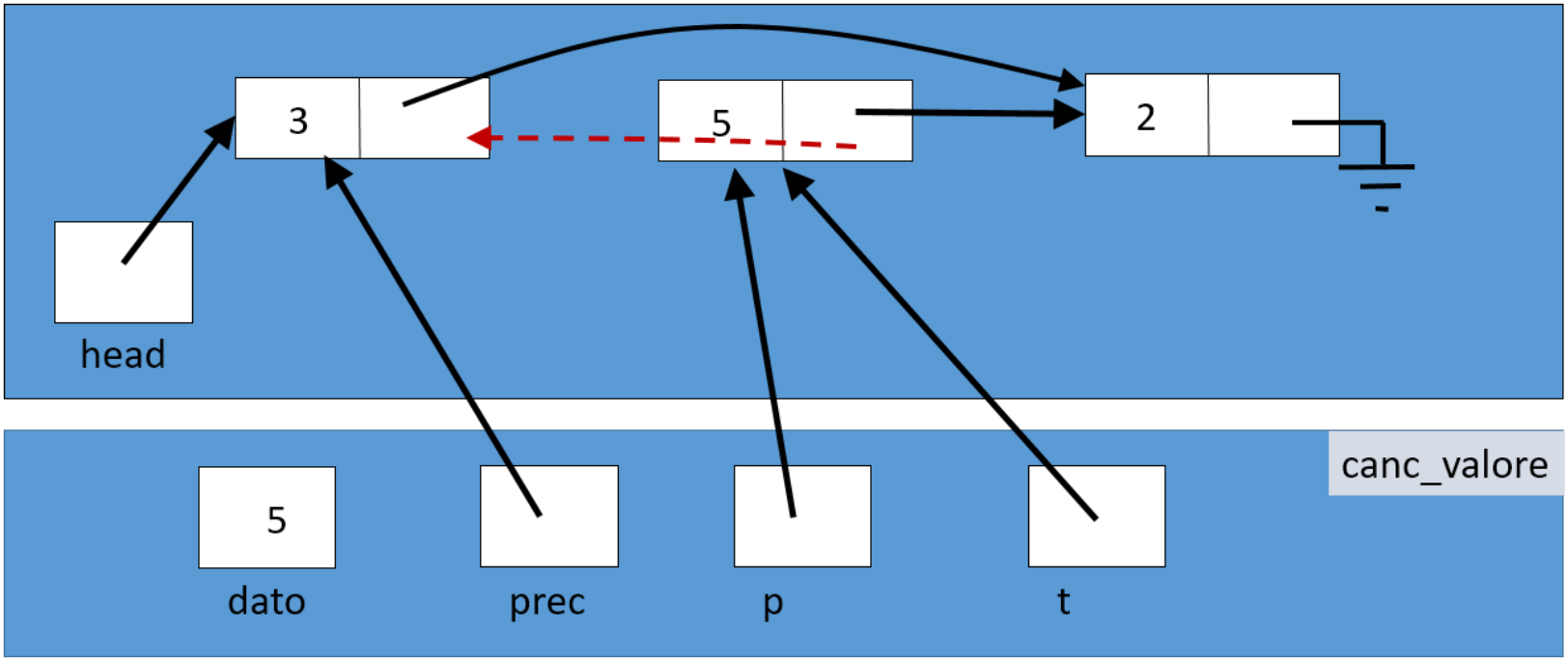
Cancellazione di un Valore: 5



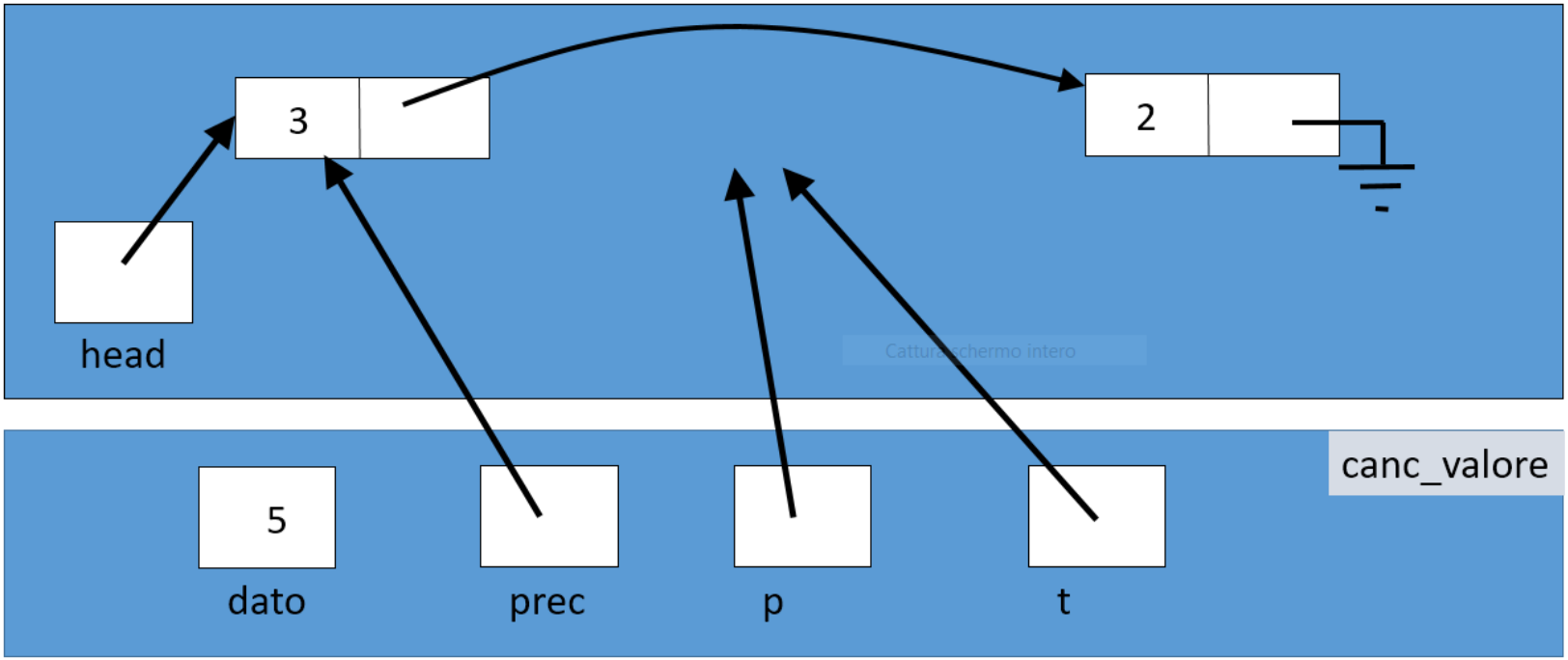
Cancellazione di un Valore: 5



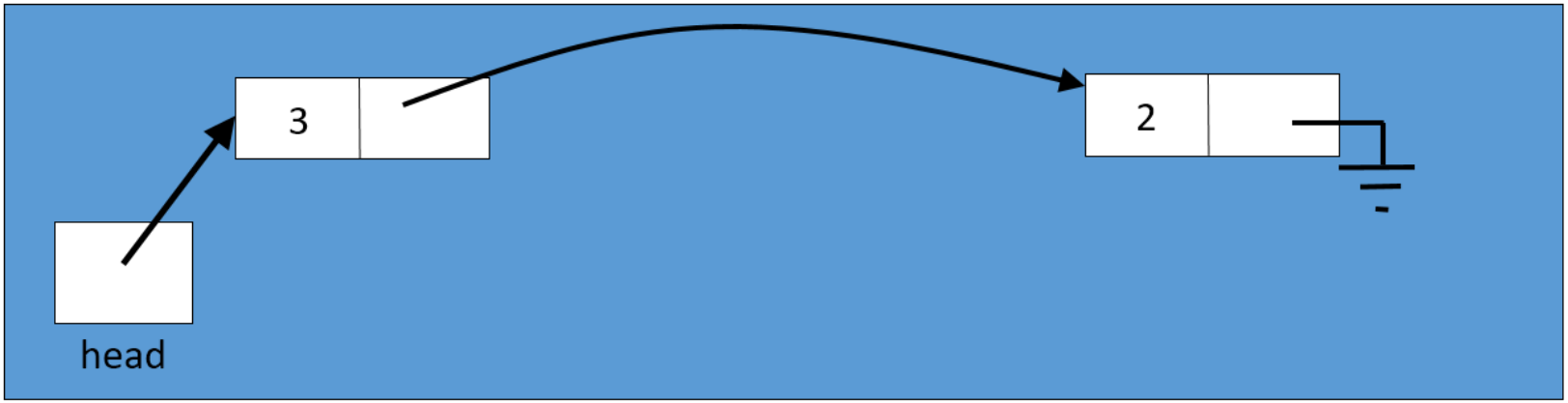
Cancellazione di un Valore: 5



Cancellazione di un Valore: 5



Cancellazione di un Valore: 5



Scansione

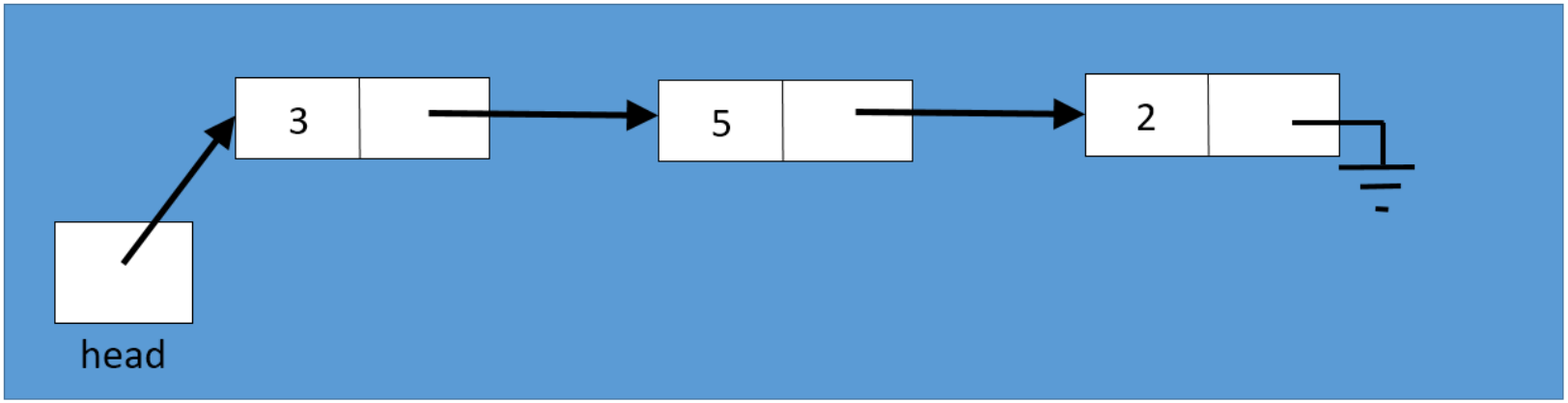
```
void stampa(NODO *head)
{
    NODO *p;

    cout << "---------------------------------" << endl;
    p = head;
    while(p != NULL)
    {
        cout << "Valore: " << p->dato << endl;
        p = p->next;
    }
    cout << "---------------------------------" << endl;
}
```

Schema della Scansione

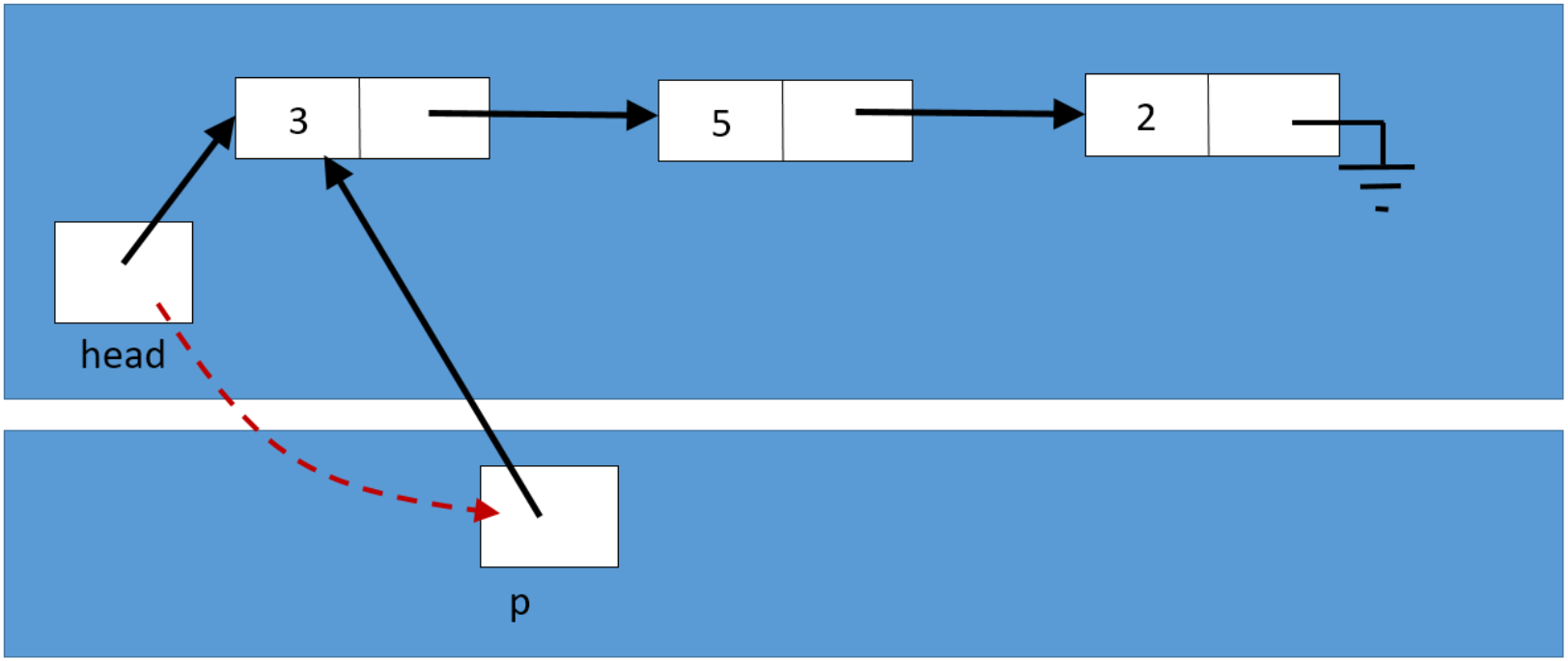
- Inizializzazione del puntatore di scansione `p`
- Ciclo `while` con condizione `p != NULL`
- Nel corpo dell ciclo:
  - uso del nodo
  - scorrimento del puntatore `p = p->next;`

```
P = head;
while( p != NULL)
{
     Uso del Nodo Corrente
     p = p->next;
}
```

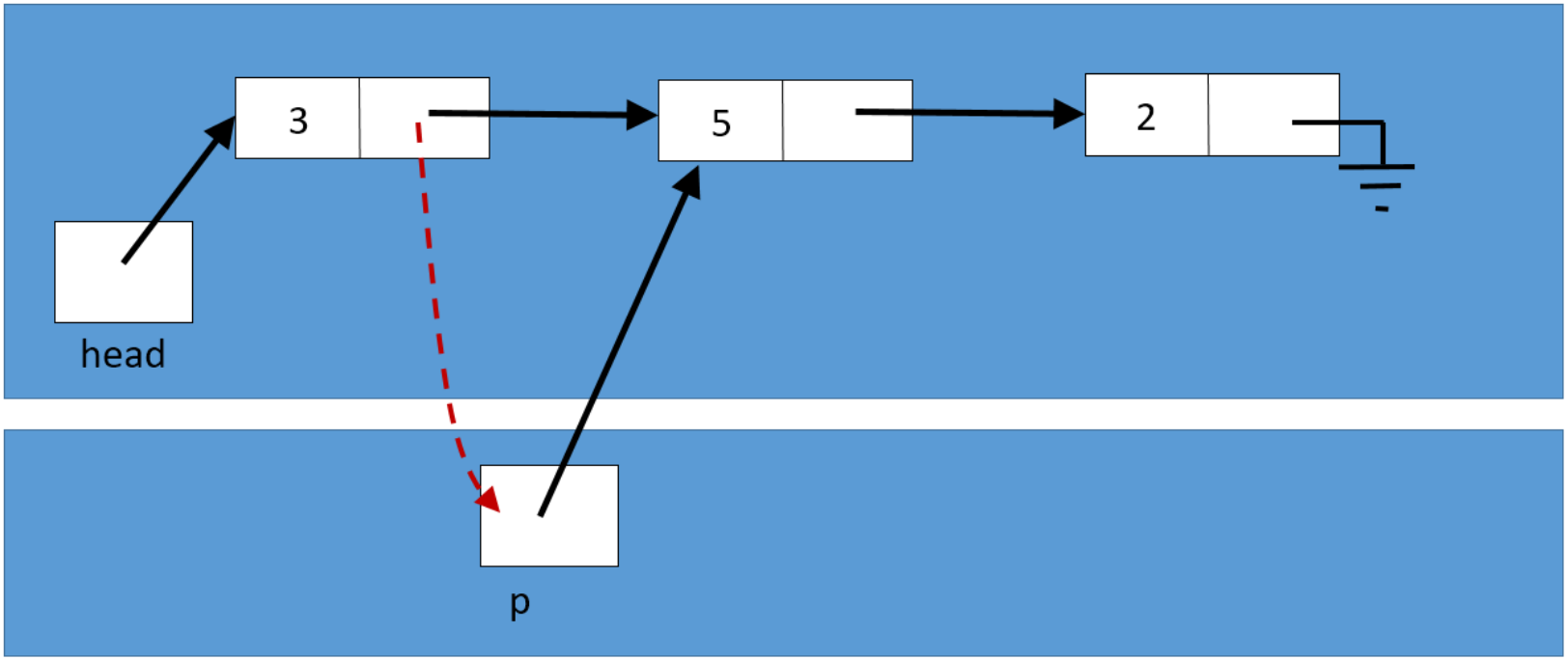
Scansione



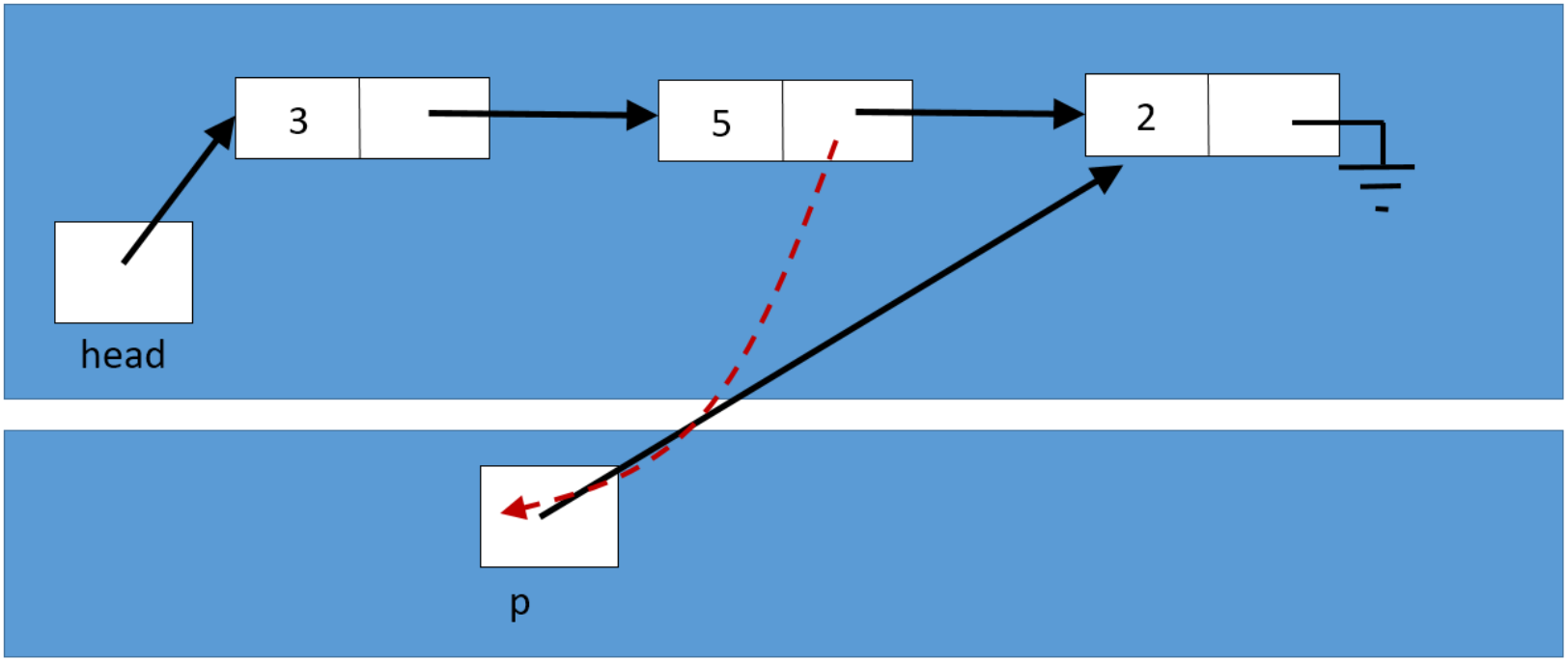
Scansione



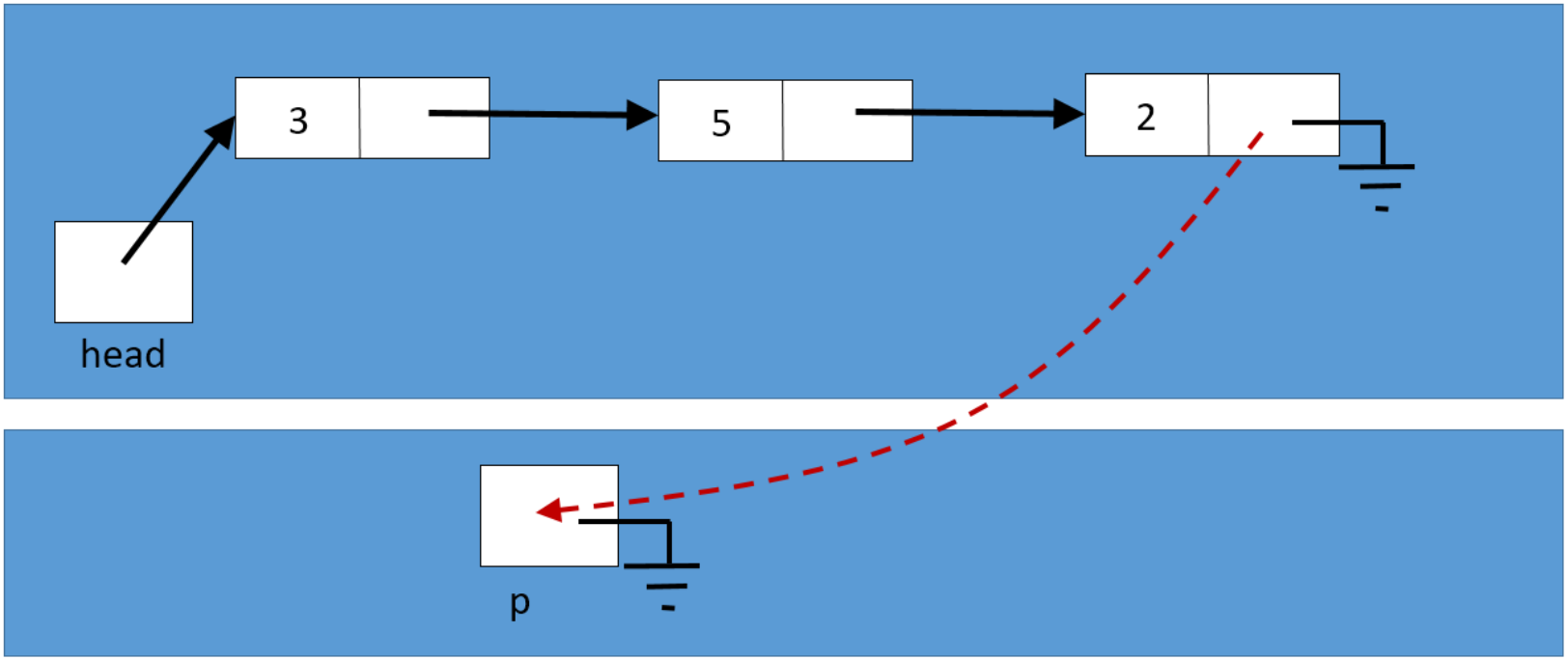
Scansione



Scansione



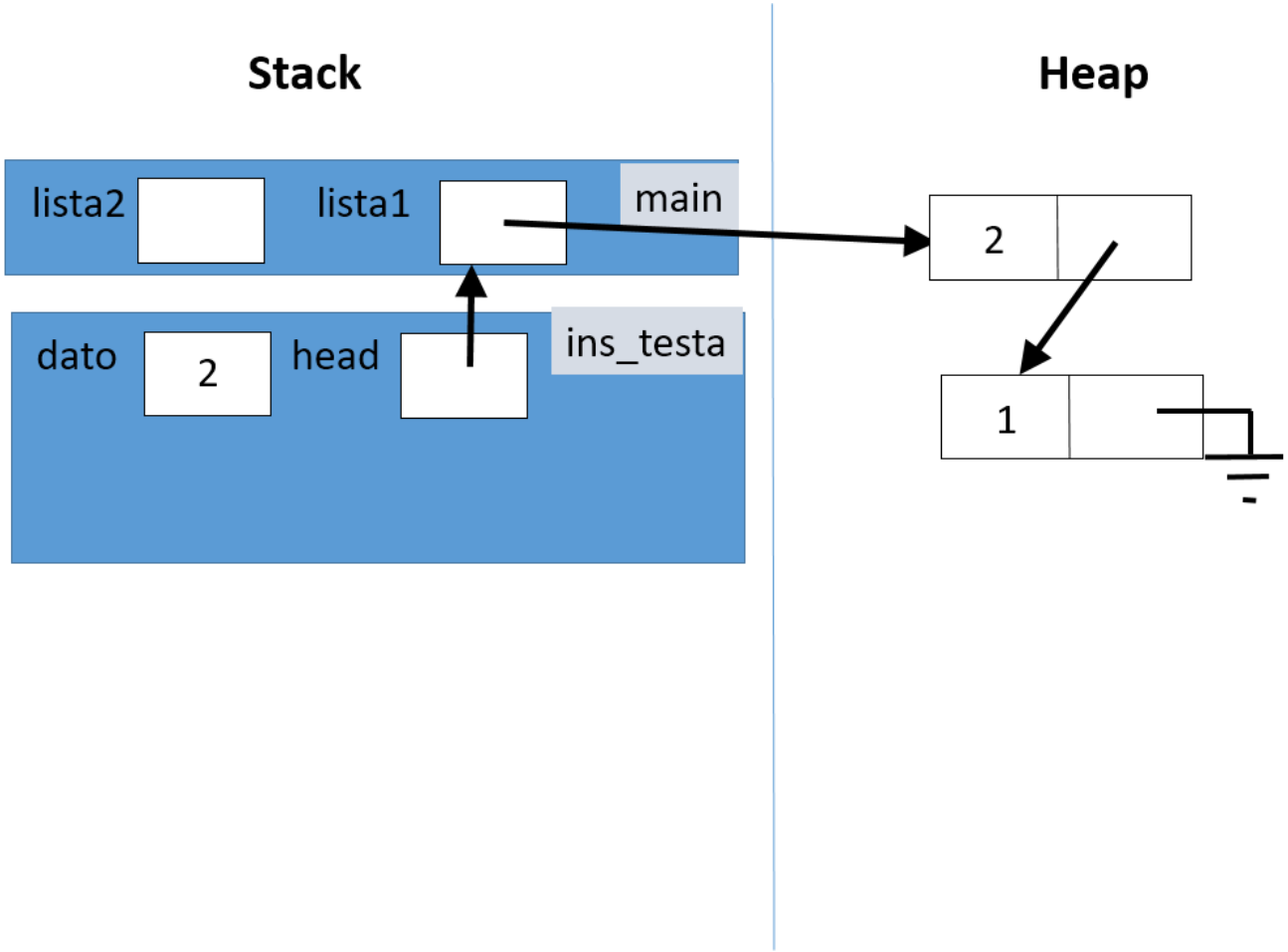
Scansione



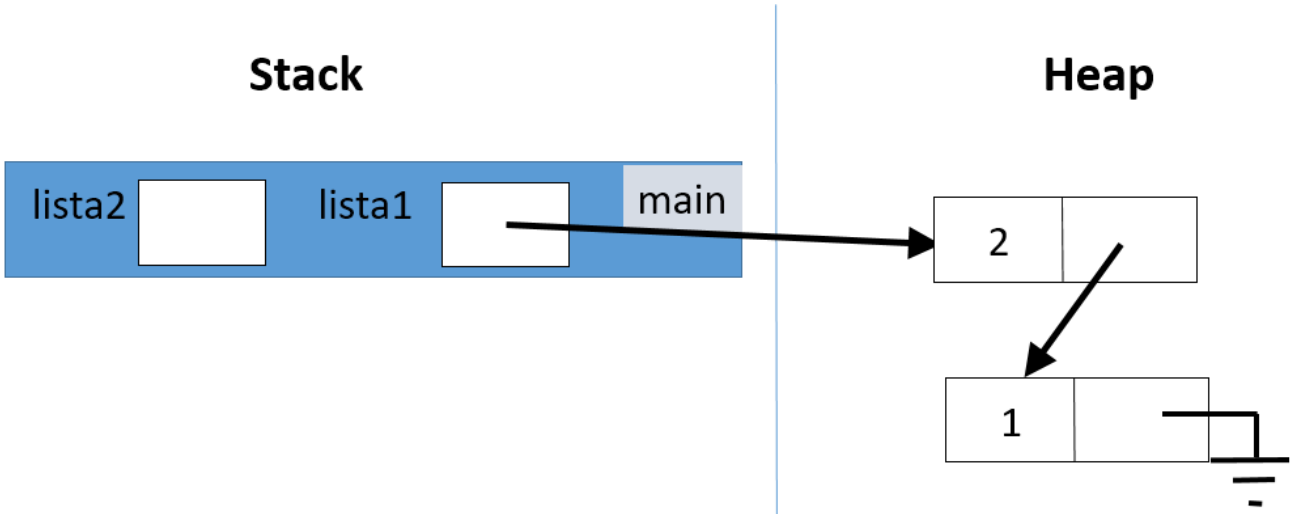
Programma Lista_Interi.cpp

- Le funzioni viste prima sono riportate in questo programma
- Nella funzione `main`, vengono definite due liste, `lista1` e `lista2`
- Un ciclo di inserimento inserisce i valori in testa,
  poi si cancella il valore dalla testa
  poi si chiede un valore da cancellare dalla lista
- Un secondo ciclo di inserimento inserisce i valori in coda,
  poi si cancella dalla coda.

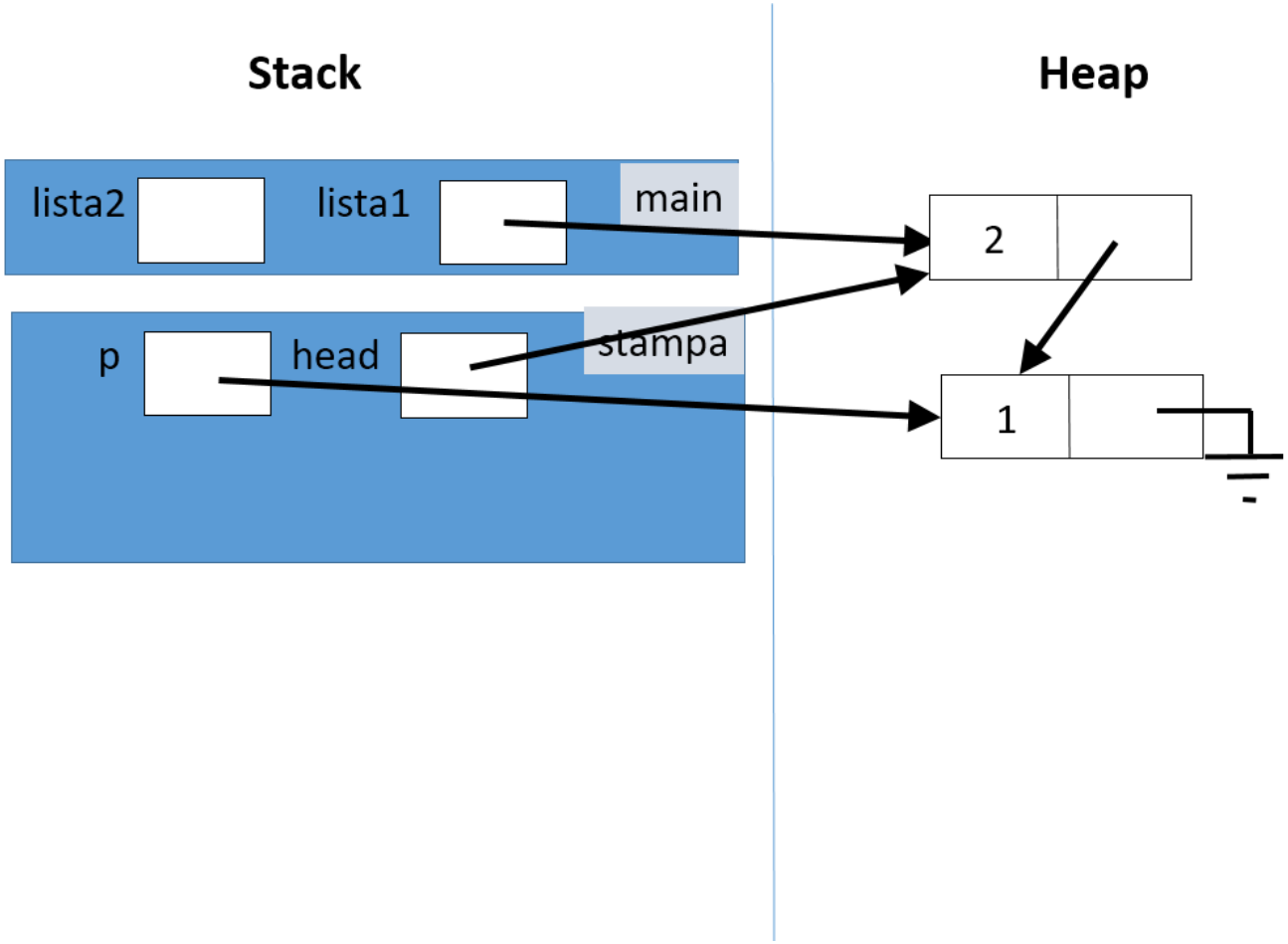
Programma Lista_Interi.cpp



Programma Lista_Interi.cpp



Programma Lista_Interi.cpp



Liste di Strutture Complesse

- Supponiamo di dover gestire una lista di punti del piano cartesiano.
- Soluzione semplice, ma meno pulita:

```
struct NODO
{
    float x;
    float y;
    NODO *next;
};
```

Liste di Strutture Complesse

- Se abbiamo il tipo `PUNTO`, dobbiamo fare una **Lista di PUNTO**.

```
struct NODO
{
    PUNTO dato;
    NODO *next;
};
```

- Come accedere al punto?

```
p->dato.x
```

La Pila (Stack)

- Una tipica struttura dati, molto usata è la **PILA** (o **STACK**, in Inglese)
- Usa una politica di gestione detta **LIFO**
  Last In
  First Out
- La sua caratteristica è quella di invertire l'ordine di estrazione, rispetto all'ordine di inserimento

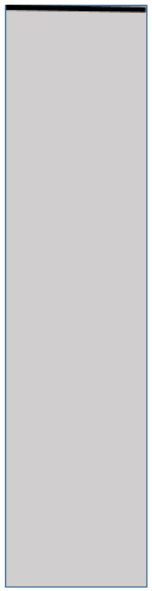
Operazioni sulle Pile

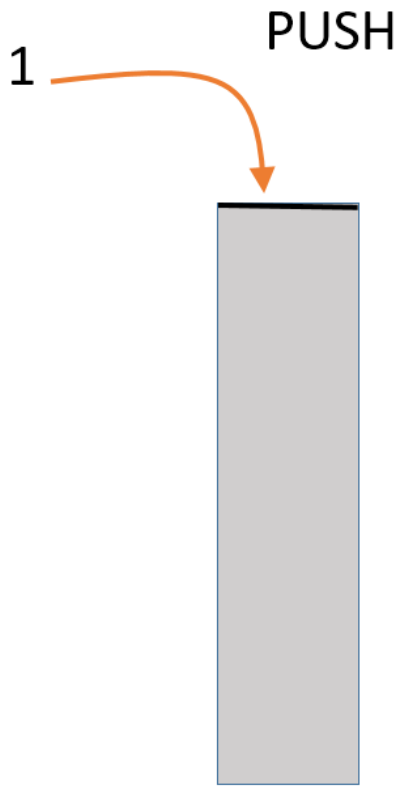
- **PUSH**

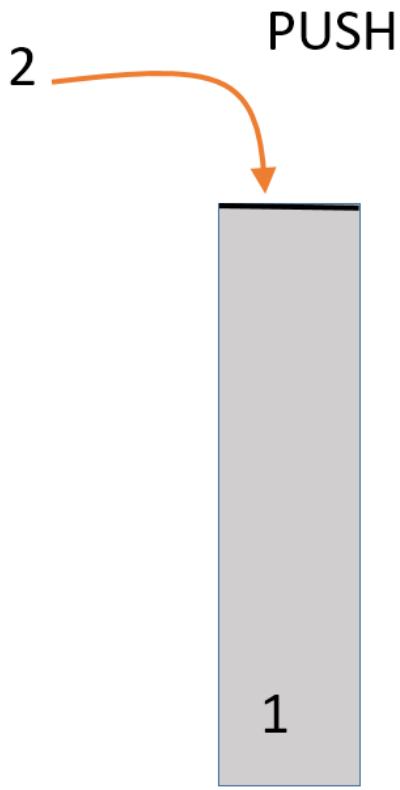
  Inserisce un nuovo valore in cima alla pila
- **TOP**

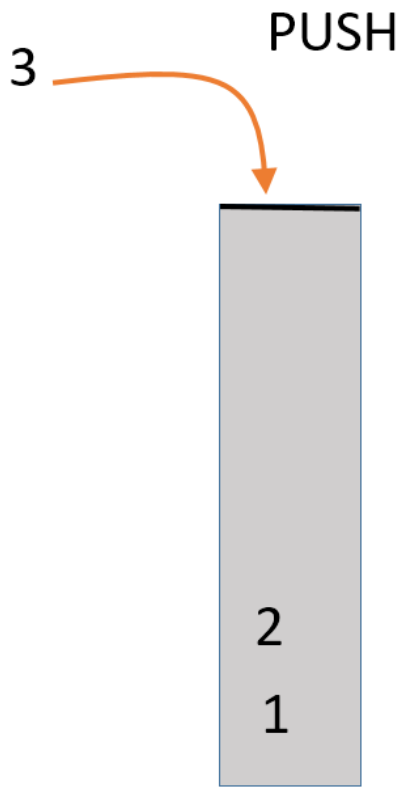
  Ispezione il valore in cima alla pila
- **POP**

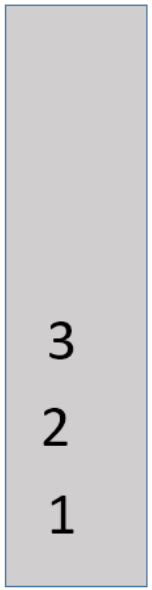
  Estrae il valore in cima alla pila

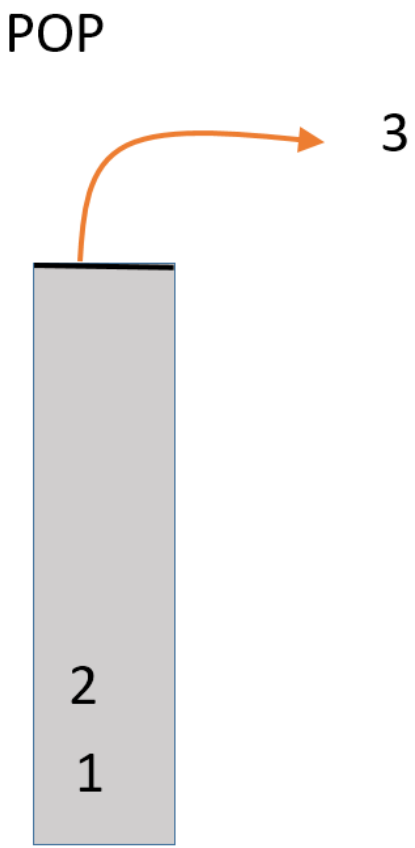



PUSH
1


PUSH
2
1


PUSH
3
2
1


3
2
1


POP
3
2
1

Pile e Liste

- La lista può essere usata per realizzare una pila
- **PUSH**
  Inserimento in Testa
- **POP**
  Cancellazione dalla Testa

La Coda (Queue)

- Un'altra struttura dati, molto usata, è la **CODA** (o **QUEUE**, in Inglese)
- Usa una politica di gestione detta **FIFO**
  First In
  First Out
- La sua caratteristica è quella di mantenere l'ordine di estrazione, rispetto all'ordine di inserimento

Operazioni sulle Code

- **APPEND**
  Inserisce un nuovo valore in fondo alla coda
- **FIRST**
  Ispeziona il valore in cima alla coda
- **EXTRACT**
  Estrae il valore in cima alla coda

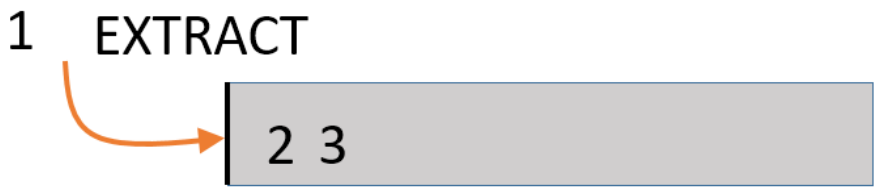
Strutture Dati Dinamiche


1
APPEND


2
APPEND
1


3
APPEND
1 2


1 2 3


1
EXTRACT
2 3

Code e Liste

- La lista può essere usata per realizzare una coda
- **APPEND**
  Inserimento in coda
- **EXTRACT**
  Cancellazione dalla Testa

Sfida

- Scrivere un programma che legge una lista di punti, in modo interattivo (cioè chiede all'utente se vuole inserire un nuovo punto oppure terminare)
- Calcola il baricentro dei punti `b`
- Stampa i punti che sono nell'area compresa tra il punto `(0,0)` e il baricentro `b` b

Sfida 2

- Definire il tipo `NOINATIVO`, come struttura di quattro campi:
  - `Codice`, stringa di 15 caratteri;
  - `Cognome`, stringa di 50 caratteri;
  - `Nome`, stringa di 50 caratteri;
  - `eta`, numero intero.

  Si definisca il tipo `NODO`, per gestire liste di `NOMINATIVO`.

Sfida 2 (Continua)

- Scrivere un programma che legge un elenco di nominativi richiesti all'utente;
  terminata la lettura, chiede all'utente se vuole salvare su file;
  in caso di risposta affermativa, scrive sul file `Nominativi.dat` l'elenco, una riga per nominativo, in modo che ogni riga sia strutturata come:
  15 carattri per il codice, 50 caratteri per il cognome, 50 caratteri per il nomee, 3 caratteri per l'età

Sfida 3

- Scrivere un programma che carica il contenuto del file di nominativi descritto nella sfida 2. Una volta caricato l'elenco di nominativi, lo stampa sullo schermo.
- Quindi, il programma chiede all'utente un numero intero `n`, crea una nuova lista con i nominativi che hanno un'età minore o uguale a `n` e la stampa sullo schermo
  Internamente, la lista viene creata dalla funzione
  `NODO *FiltraLista(NODO *lista, int n);`

Sfida 3 (Continua)

- Il programma chiede all'utente se vuole salvare la lista filtrata su file: in caso affermativo, chiede il nome del file su cui salvare (il formato del file è a vostra discrezione).
  La nuova lista viene deallocata.
- Il programma chiede all'utente se vuole nuovamente filtrare la lista originale di nominativi, altrimenti termina.