


# strutture dinamiche: le liste

introduzione alla programmazione

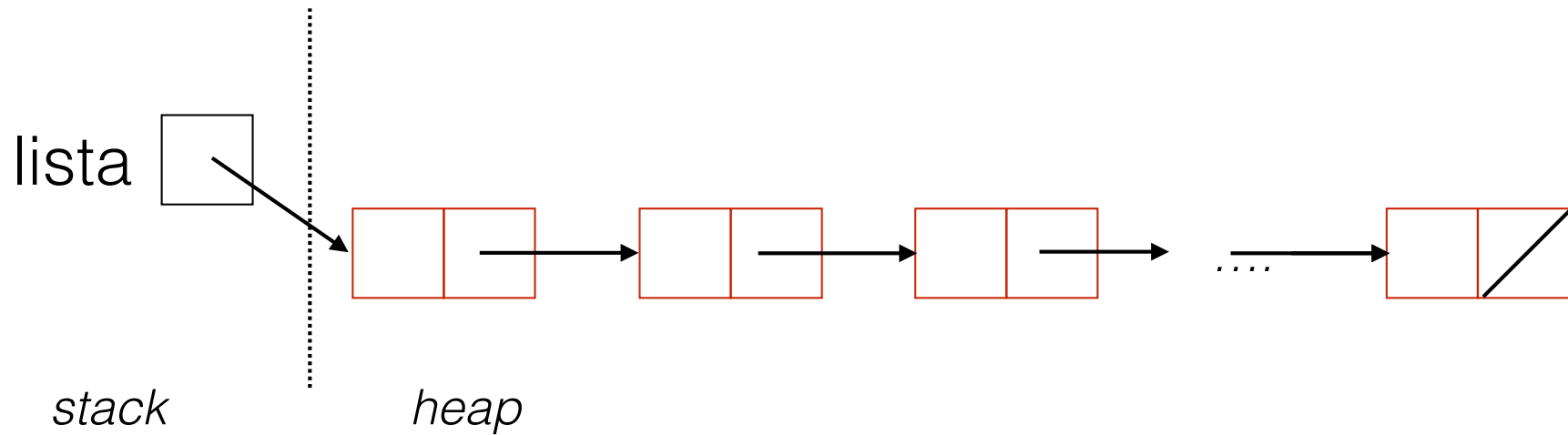
# introduzione

- gli array o i vector sono gli strumenti di riferimento quando vogliamo trattare *sequenze di dati omogenei*
  - in alcuni casi sono inefficienti:
    - A. se prevediamo frequenti inserimenti o cancellazioni in posizioni intermedie
    - B. se le dimensioni della sequenza variano molto spesso nel corso dell'esecuzione del programma
- Questo i vector lo gestiscono piuttosto bene
- 

# introduzione

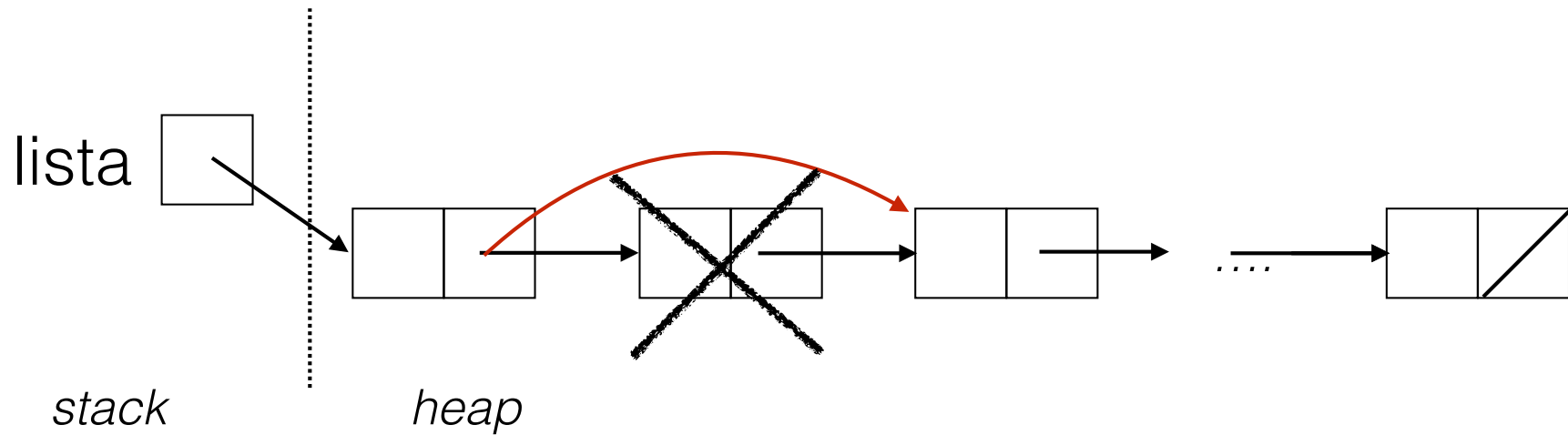
- Per entrambi i casi un'alternativa efficace sono le *liste*, che si realizzano mediante concatenazione attraverso puntatori di singole celle allocate separatamente (ossia in posizioni non necessariamente contigue)
- Notate però che le liste perdono un'importante proprietà degli array: l'accesso diretto a locazioni arbitrarie tramite indici
- Questo perché **gli elementi della sequenza non sono memorizzati in locazioni di memoria contigue**

# una lista semplice - intuizione



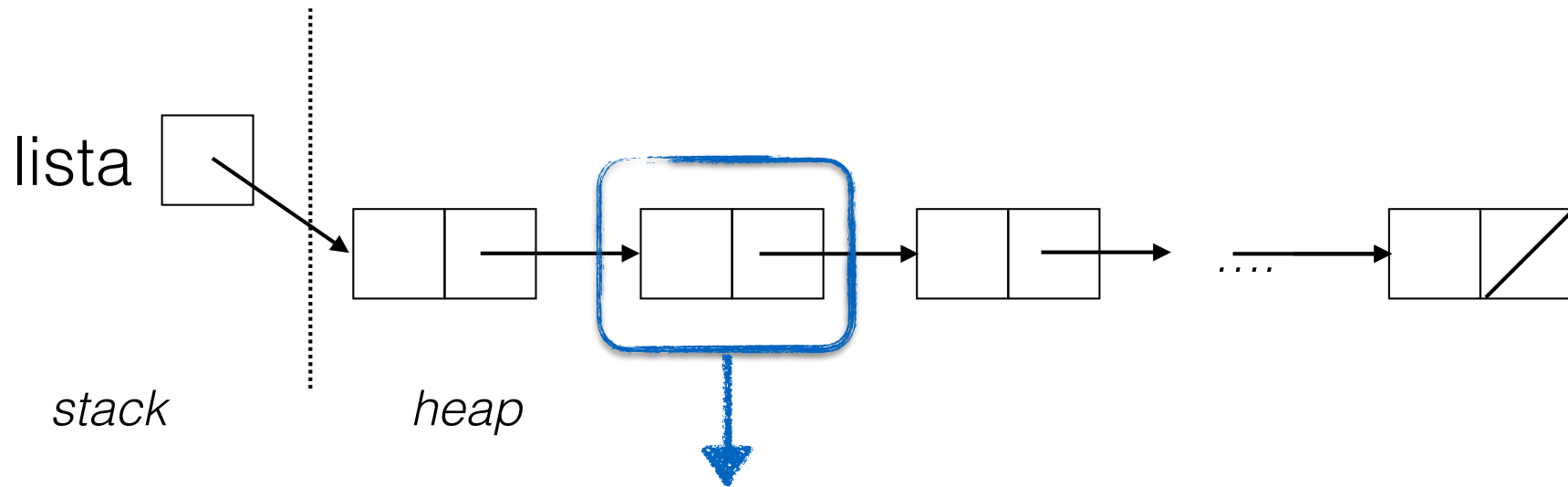
- ogni cella (in rosso) memorizza il contenuto e un puntatore alla cella successiva

# una lista semplice - intuizione



cancellare un elemento

# una lista semplice



Ogni singolo elemento o *cella*  
aggrega due dati non omogenei:

- un tipo di dato T
- e un puntatore a cella

# le celle di una lista di tipo T

- ```
struct cell {  
    T info;    //T tipo noto a priori  
    cell *next;  
};
```
- la dichiarazione  

```
cell c;
```

dichiara sullo stack una variabile `c` di tipo `cell`  
per accedere ai suoi campi:  
`c.info`  
`c.next`
- per usare la struct in un contesto dinamico devo usare un puntatore  

```
cell *lista;
```

per accedere ai suoi campi  
`(*lista).info` e `(*lista).next`  
una notazione più comune e più suggestiva è  
`lista->info` e `lista->next` (che si legge “puntato”)

# liste semplici

- Definizione di tipo

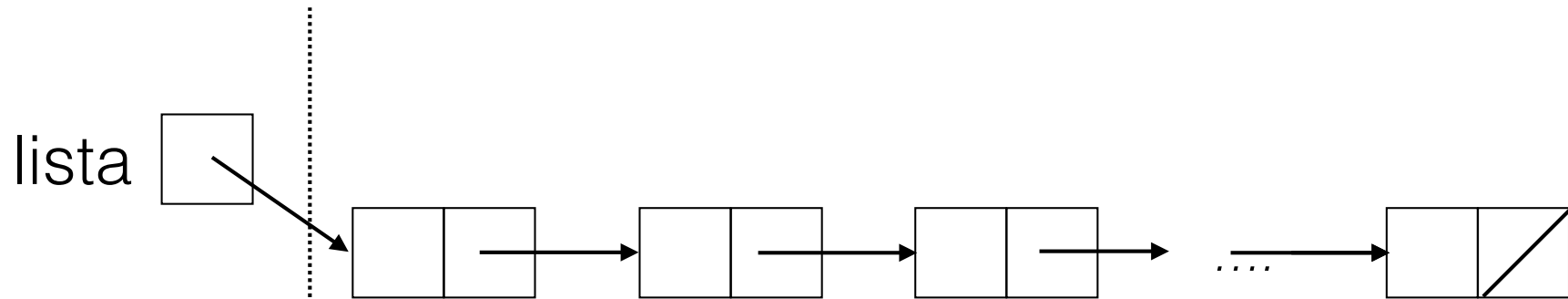
```
struct cell {  
    T info; //T tipo noto a priori  
    cell *next;  
};  
typedef cell*list;
```

- Dichiarazione e inizializzazione

```
list lista=NULLPTR;
```

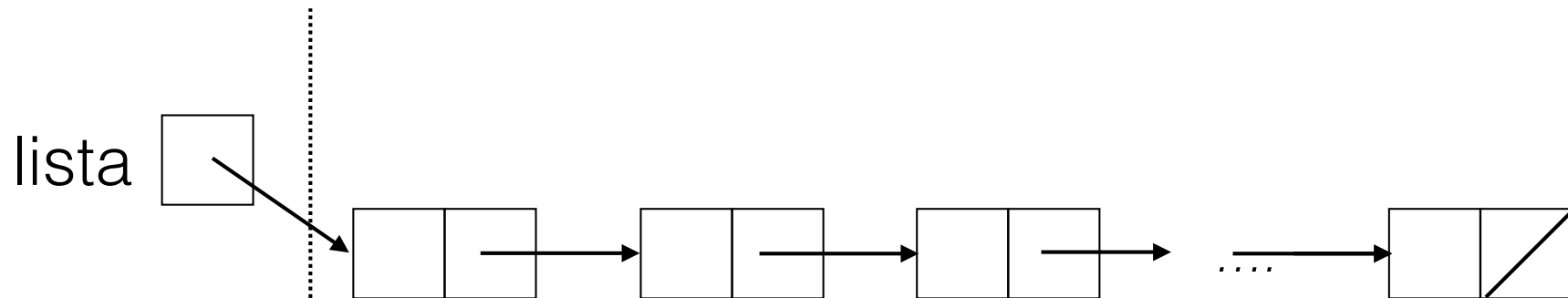


# liste semplici



```
struct cell {  
    T info;  //T tipo noto a priori  
    cell *next;  
};  
typedef cell*list;
```

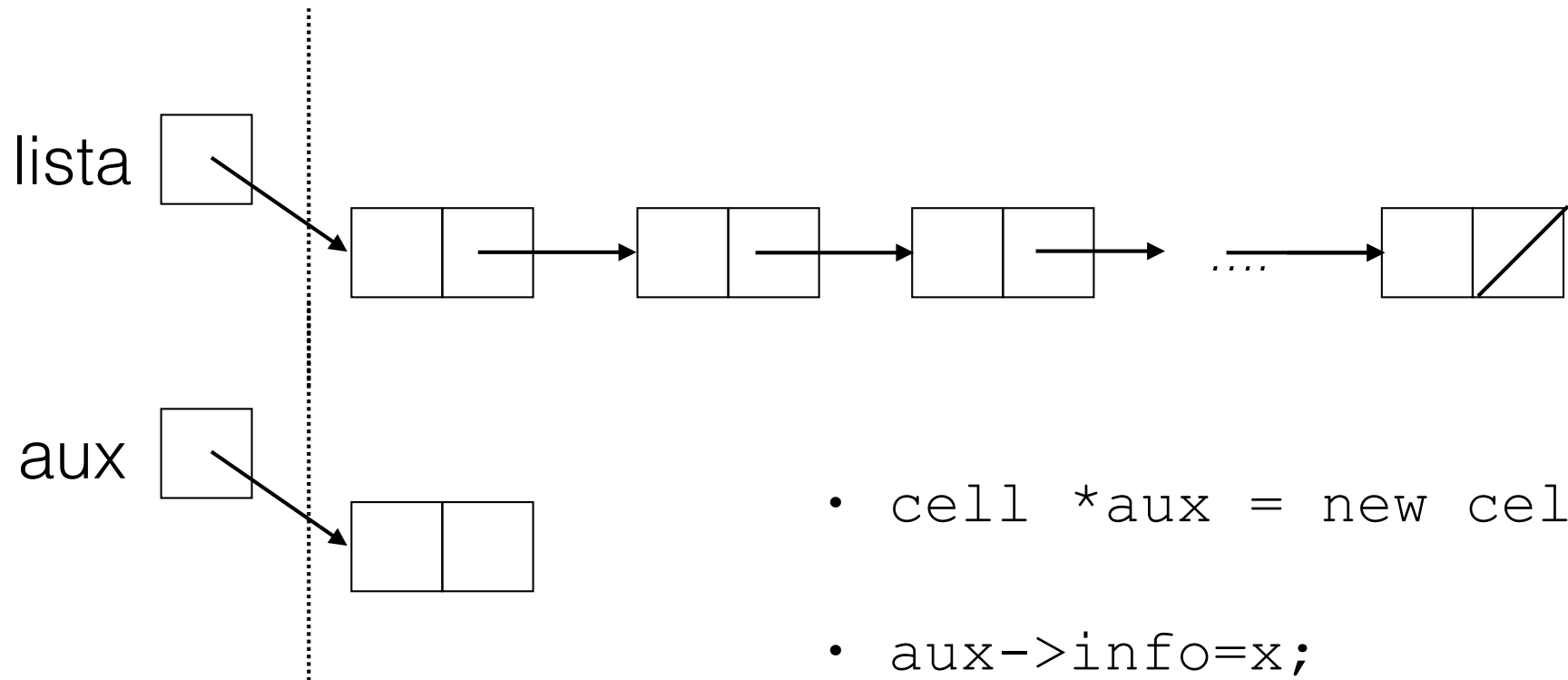
# liste semplici - inserimento in testa



```
struct cell {  
    T info;  //T tipo noto a priori  
    cell *next;  
};  
typedef cell*list;
```

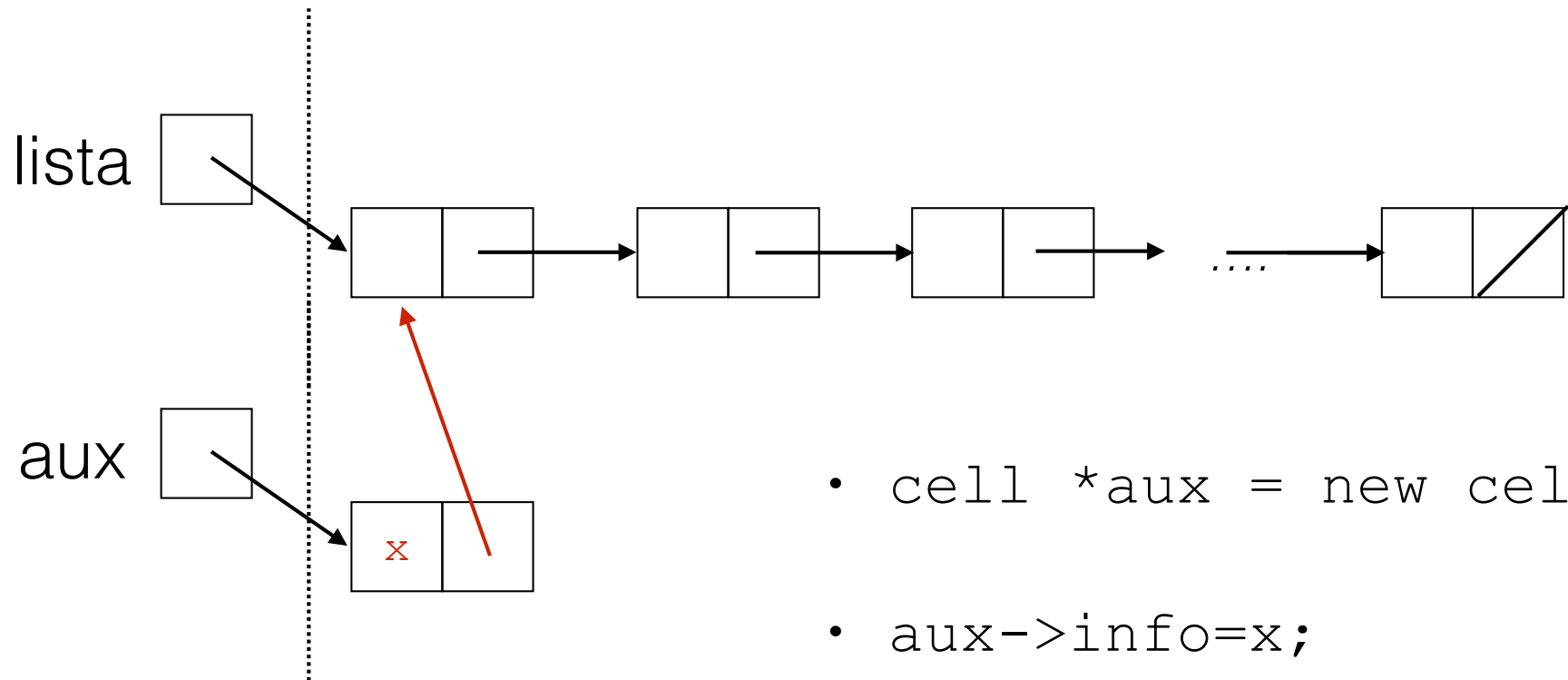
- vogliamo inserire un elemento di tipo T, sia x, in testa alla lista

# liste semplici - inserimento in testa



- costruiamo una variabile ausiliaria di tipo puntatore a cell e inseriamo x nel campo info (che memorizza il contenuto della cella)

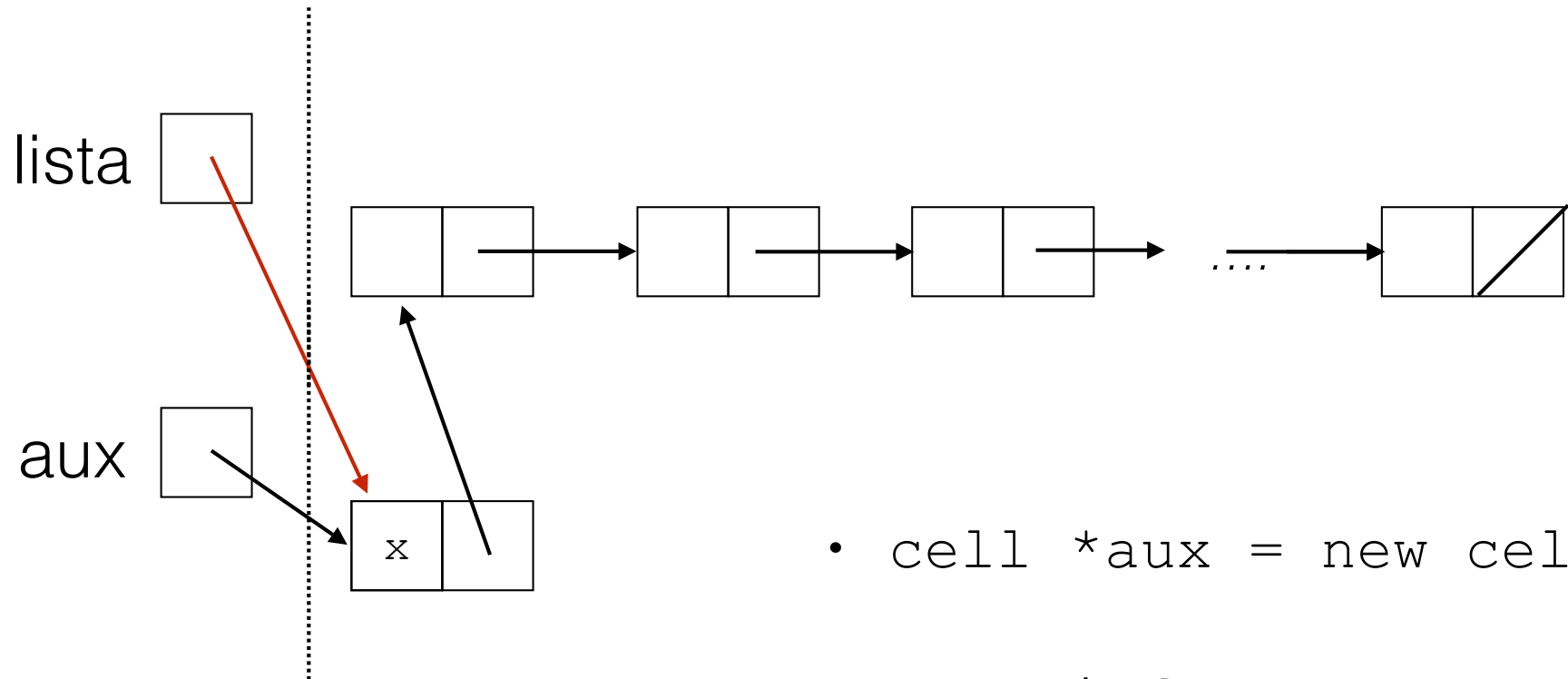
# liste semplici - inserimento in testa



- `cell *aux = new cell;`
- `aux->info=x;`
- `aux->next=list;`

- “aggiustiamo” i puntatori (appendiamo la vecchia lista in fondo ad aux)

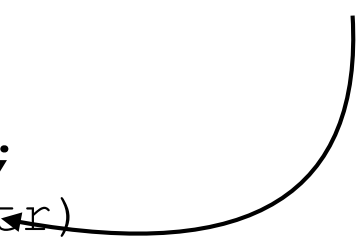
# liste semplici - inserimento in testa



- `cell *aux = new cell;`
- `aux->info=x;`
- `aux->next=lista;`
- `lista = aux;`

# liste semplici - scansione sequenziale

verifico di non essere arrivato in fondo  
se `aux==nullptr` non posso accedere ai suoi campi



- ```
cell *aux = lista;
while (aux!=nullptr)
{
    leggi(aux->info); // leggi è una funzione che
                      // legge un tipo T
    aux=aux->next;
}
```

Notare l'uso del puntatore ausiliario (cursore)  
Serve per non dover spostare il  
puntatore iniziale (perche'?)

# Liste semplici - accesso all'elemento n

Non è immediato come negli array  
PERCHE'?...

- ```
int cont=0;
cell *aux = lista;
while ((aux!=nullptr)&& (cont<n))
{
    aux=aux->next;
    cont++;
}
if (aux!=nullptr)
    cout << n <<"-esimo elemento e'" << aux->info << endl;
```

# Liste semplici - accesso all'elemento n

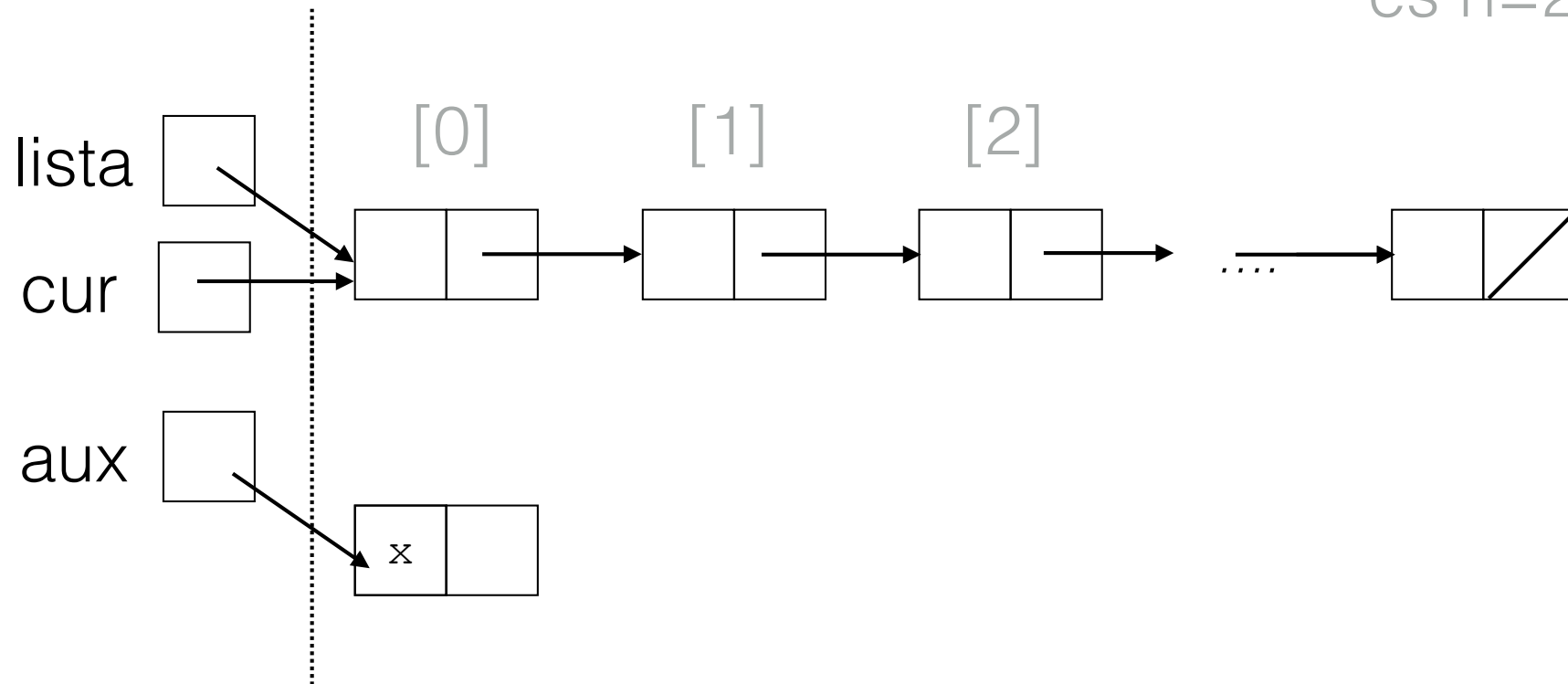
Non è immediato come negli array (non possiamo basarci sul fatto che le celle di memoria occupate sono adiacenti)

- ```
int cont=0;
cell *aux = lista;
while ((aux!=nullptr)&& (cont<n))
{
    aux=aux->next;
    cont++;
}
if (aux!=nullptr)
    cout << n <<"-esimo elemento e'" << aux->info << endl;
```



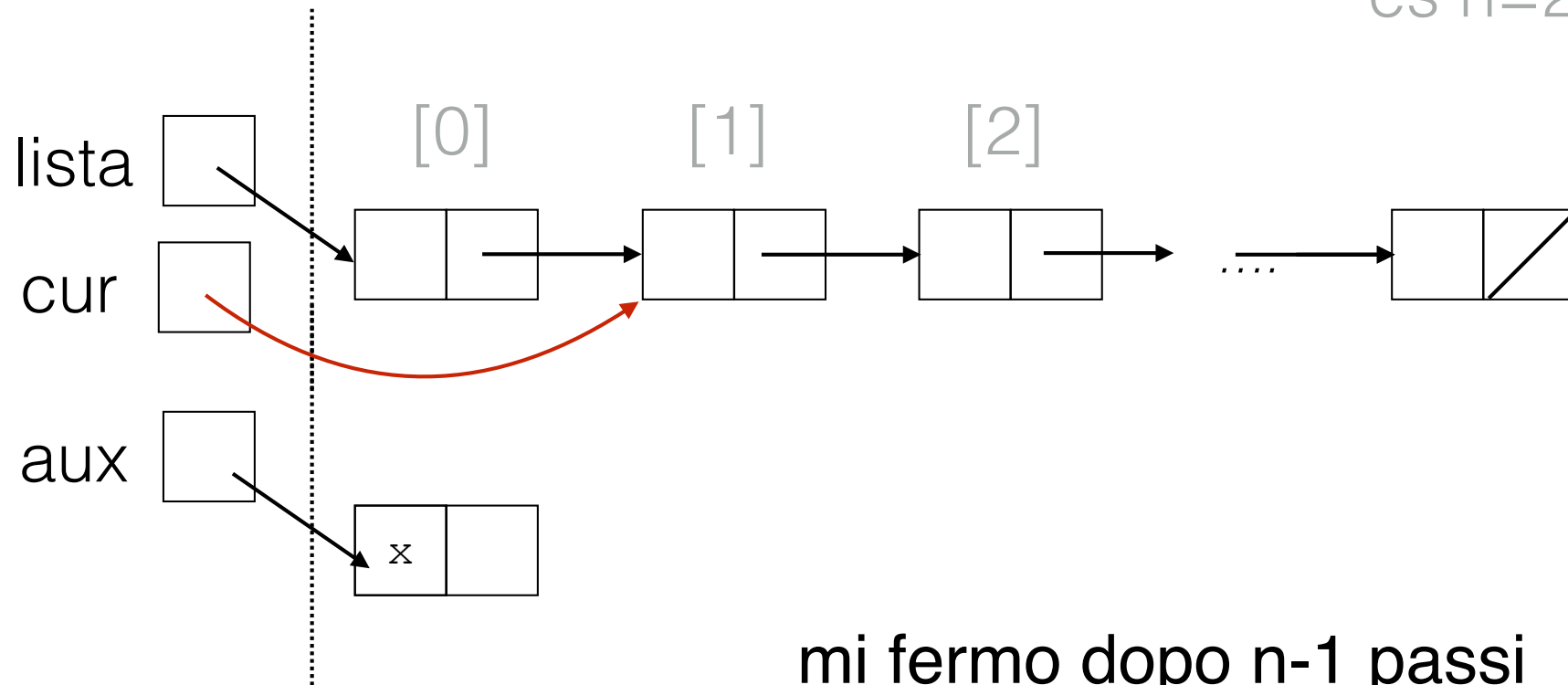
# liste semplici - inserimento di un elemento in posizione n

es n=2



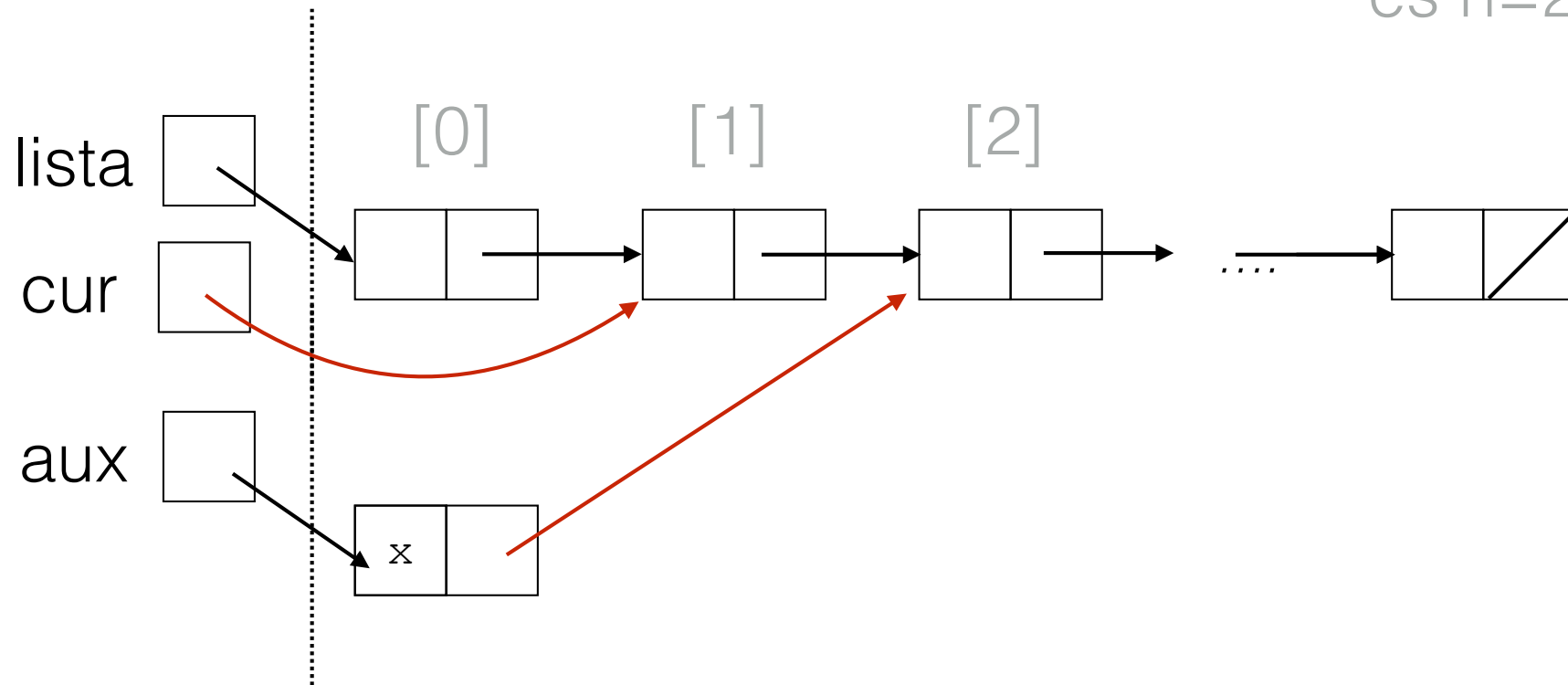
# liste semplici - inserimento di un elemento in posizione n

es  $n=2$



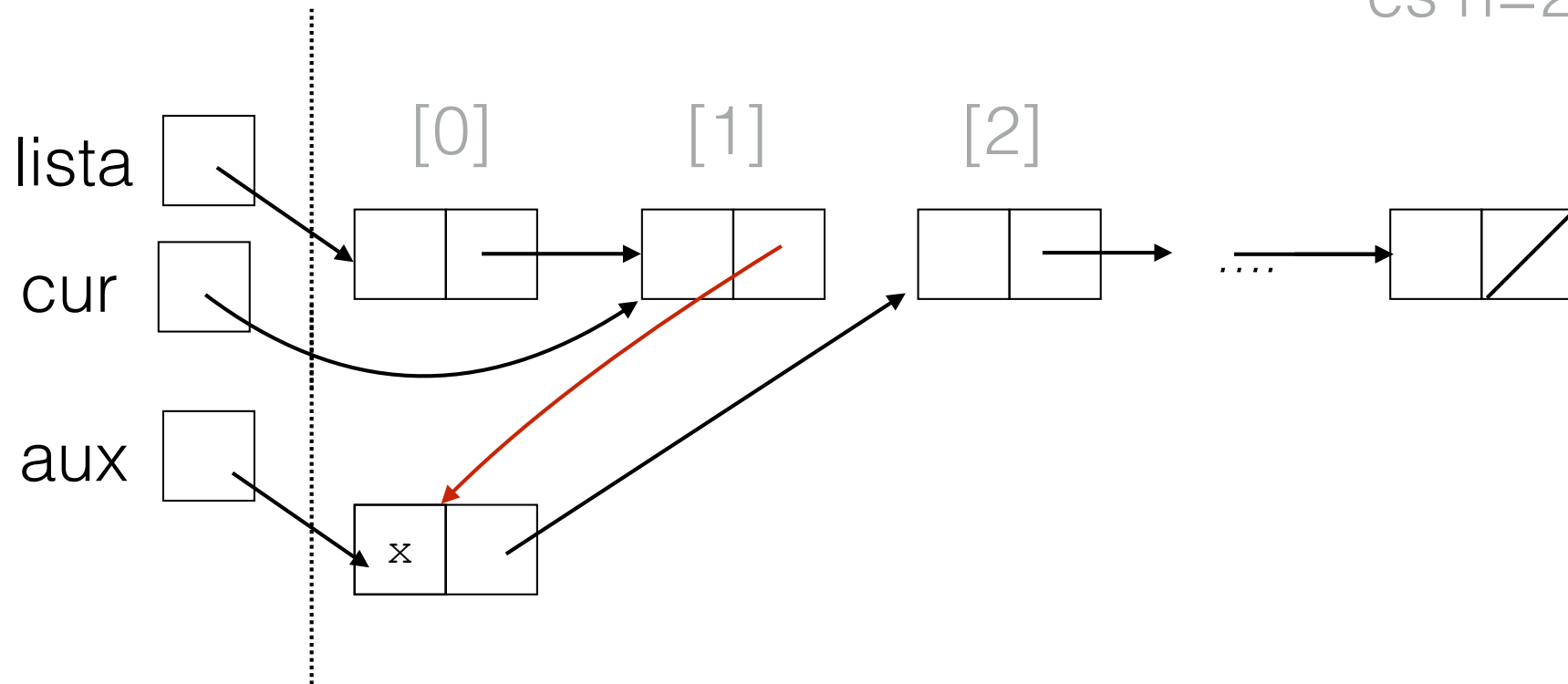
# liste semplici - inserimento di un elemento in posizione n

es n=2



# liste semplici - inserimento di un elemento in posizione n

es n=2



# Lista semplice - inserimento di un elemento in posizione n

- PREPARO ELEMENTO

```
cell *aux = new cell;  
aux->info=x;
```

- SCORRO

```
int cont=0;  
cell *cur = lista;  
cell *prev;  
while ((cur!=nullptr)&& (cont<n))  
{  
    prev=cur;  
    cur=cur->next;  
    cont++;  
}
```

- INSERISCO

```
if (cont==n) {  
    aux->next=cur;  
    prev->next=aux;  
}
```

# Lista semplice - cancellazione

```
void CancellaElem(list &l, T elem)
{
    cell *cur=l;
    cell *prev;
    while (cur!=nullptr && cur->info!=elem)
    {
        prev=cur;
        cur=cur->next;
    }
    if (cur!=nullptr) //altrimenti non faccio nulla
    {
        if (cur==l)
            l=l->next; //inizio
        else
            prev->next=cur->next; //in mezzo
        delete cur;
    }
}
```

Variante: cancellare tutte le occorrenze di elem.

# Liste semplici ordinate - Inserimento

```
void InserisciInOrdine(list &l, T elem)
{
    cell *aux = new cell;
    aux->info=elem;

    cell *cur = l;
    cell *prev;
    while ((cur!=nullptr)&& (cur->info<=elem))
    {
        prev=cur;
        cur=cur->next;
    }
    aux->next=cur;
    if (cur==l)
        l=aux;
    else
        prev->next=aux;
}
```

Precondizione: la lista deve essere ordinata

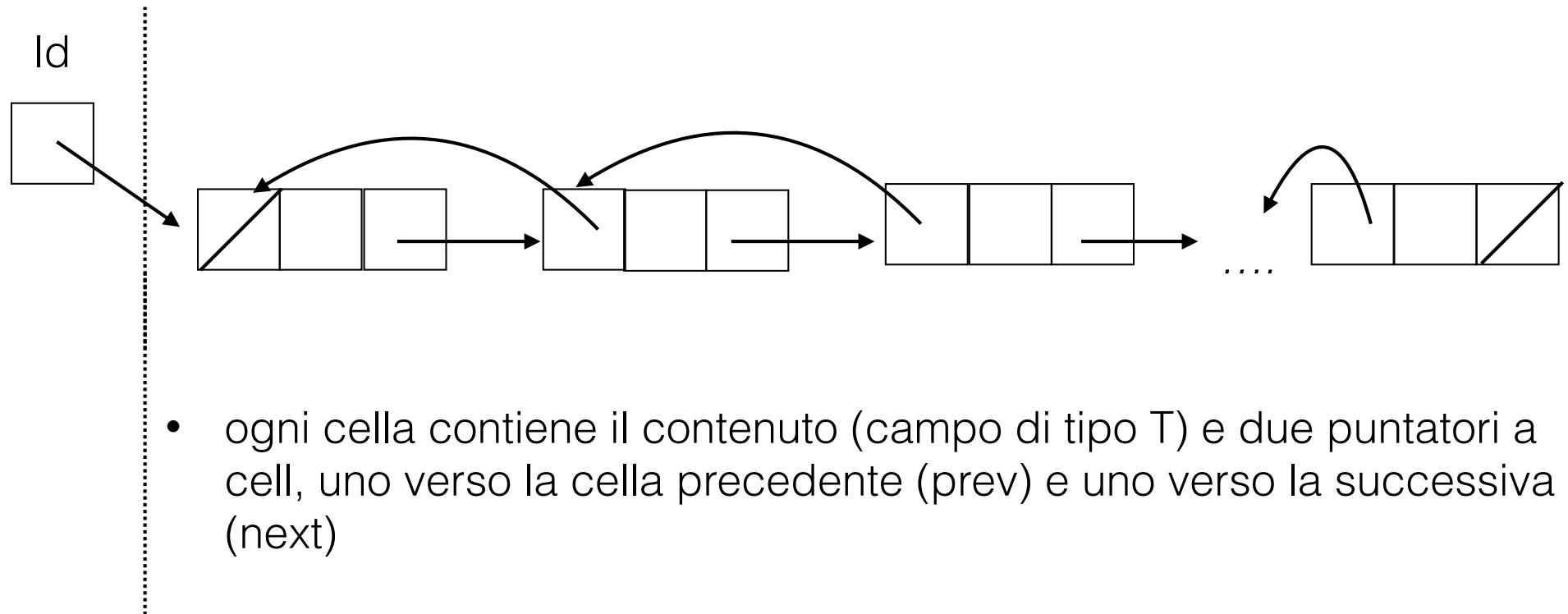
Postcondizione: la lista rimarrà ordinata

# Esercizi

- Come realizzare liste ordinate? (Hint: prevedendo solo una modalità di inserimento degli elementi, quella ordinata)
- Ragioniamo su come realizzare funzioni che manipolano coppie di liste collegati semplici, per es
  - `bool are_equal(const list& l1, const list& l2);`
  - `list cat(const list& l1, const list& l2);`
  - `list intersect(const list& l1, const list& l2);`
  - `list union(const list& l1, const list& l2);`
- Come realizzare funzioni di questo tipo senza perdere la proprietà dell'ordinamento degli elementi?



# liste doppiamente collegate



- ogni cella contiene il contenuto (campo di tipo T) e due puntatori a cell, uno verso la cella precedente (prev) e uno verso la successiva (next)

```
typedef int T;

typedef struct cell {
    T head;
    cell *next;
    cell *prev;
} *lista_doppia;

lista_doppia ld; //variabile
```

oppure

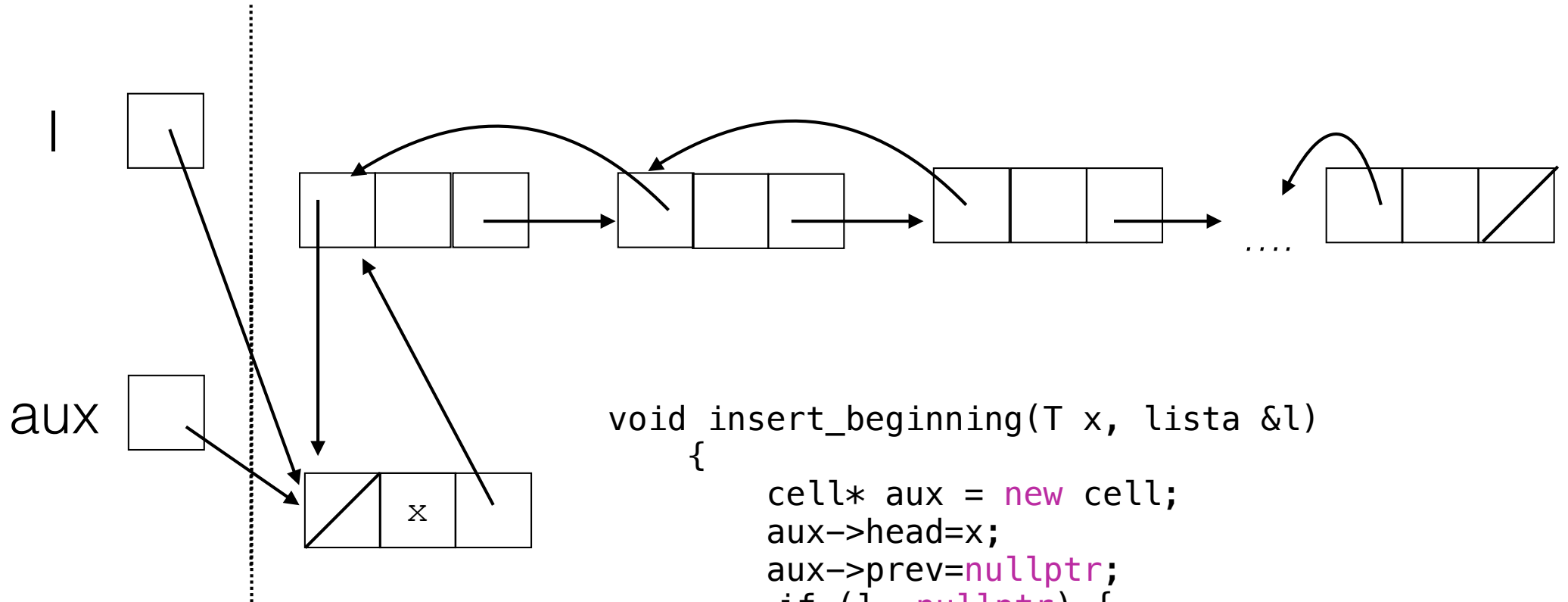
```
typedef int T;

struct cell {
    T head;
    cell *next;
    cell *prev;
};

typedef cell * lista_doppia;

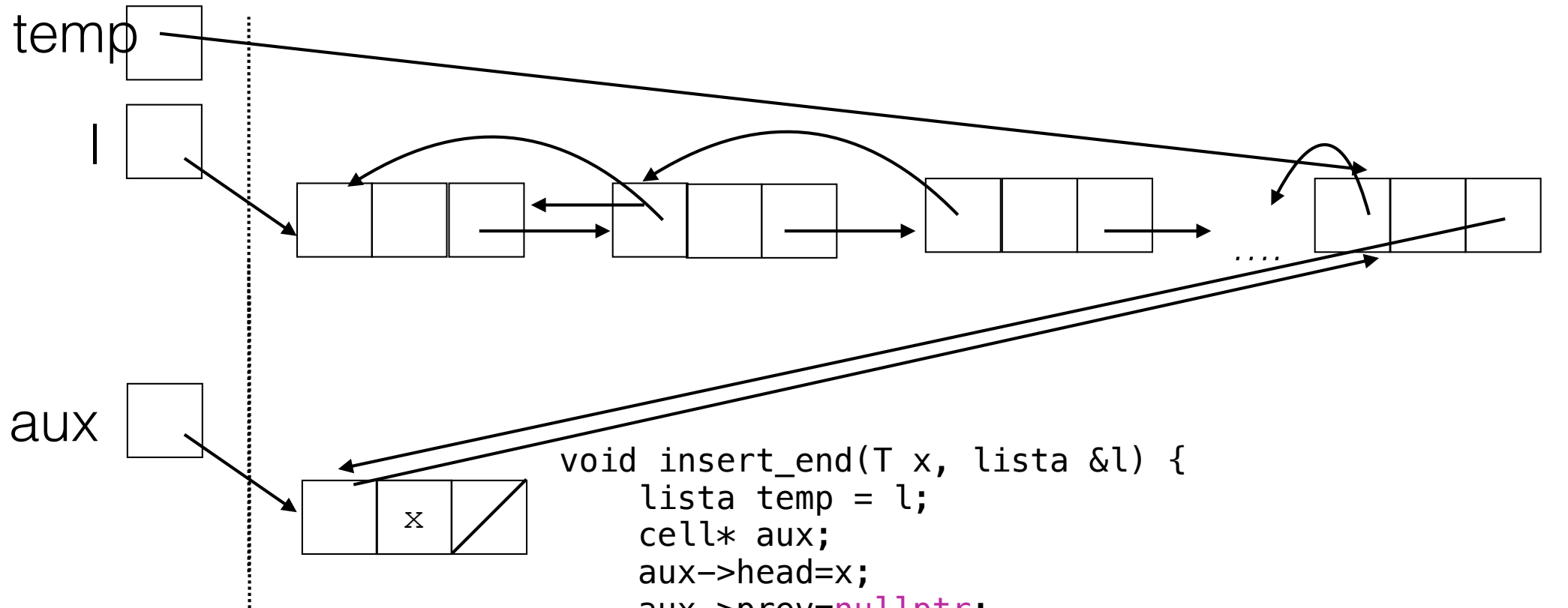
lista_doppia ld;
```

# liste doppiamente collegate inserimento in testa



```
void insert_beginning(T x, lista &l)
{
    cell* aux = new cell;
    aux->head=x;
    aux->prev=nullptr;
    if (l==nullptr) {
        aux->next=nullptr;
        l=aux;
    } else {
        l->prev=aux;
        aux->next=l;
        l=aux;
    }
}
```

# liste doppiamente collegate inserimento in coda



```
void insert_end(T x, lista &l) {  
    lista temp = l;  
    cell* aux;  
    aux->head=x;  
    aux->prev=nullptr;  
    aux->next=nullptr;  
    if(l == nullptr) {  
        l = aux;  
    } else  
        while(temp->next != nullptr) temp = temp->next;  
    temp->next = aux;  
    aux->prev = temp;  
}
```

# liste doppiamente collegate visita al contrario

```
void print_backward(const lista l)
{
    cell* temp = l;
    if(temp == nullptr) return; // empty list, exit
    // Going to last Node
    while(temp->next != nullptr) {
        temp = temp->next;
    }
    // Traversing backward using prev pointer
    cout << "Reverse: ";
    while(temp != nullptr) {
        cout << temp->head;
        temp = temp->prev;
    }
    cout << endl;
}
```