

strutture dinamiche: le liste

introduzione alla programmazione

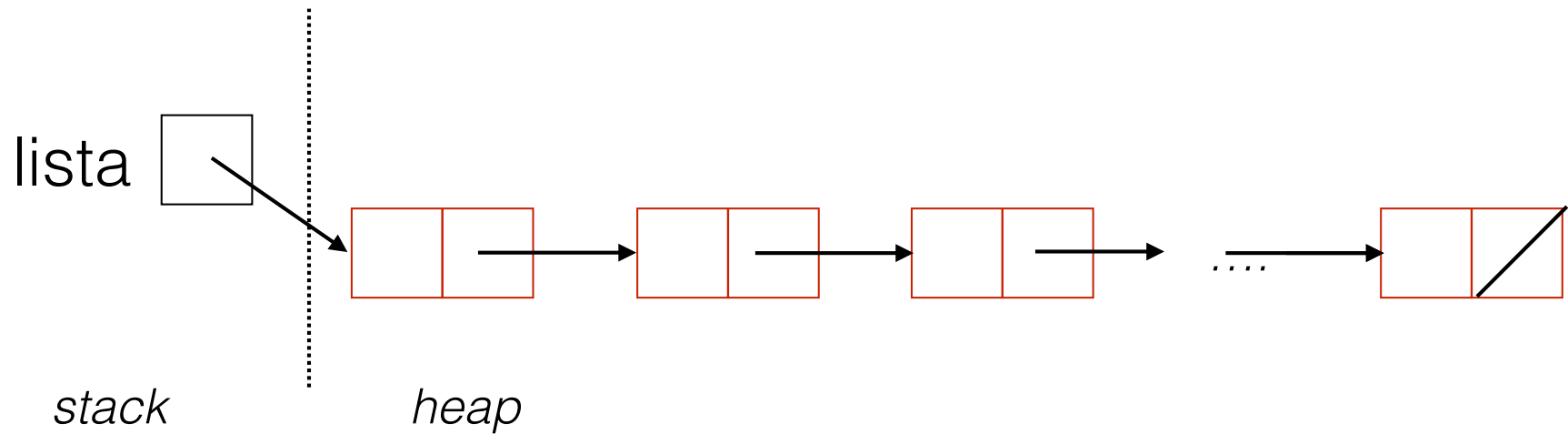
introduzione

- 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

introduzione

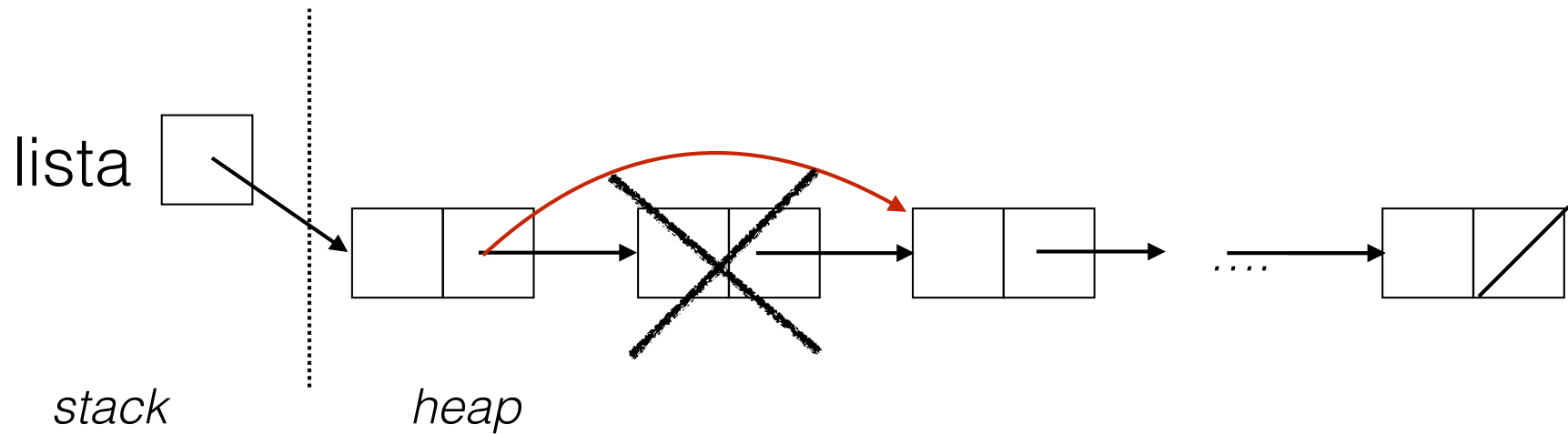
- il punto B è gestito piuttosto bene dai vector
- in entrambi i casi un'alternativa efficace sono le *liste*, che si realizzano mediante concatenazione attraverso puntatori di singole celle allocate separatamente (ossia in posizioni non necessariamente contigue)
- notare però che le liste perdono un'importante proprietà degli array: l'accesso diretto a locazioni arbitrarie tramite indici

una lista semplice - intuizione



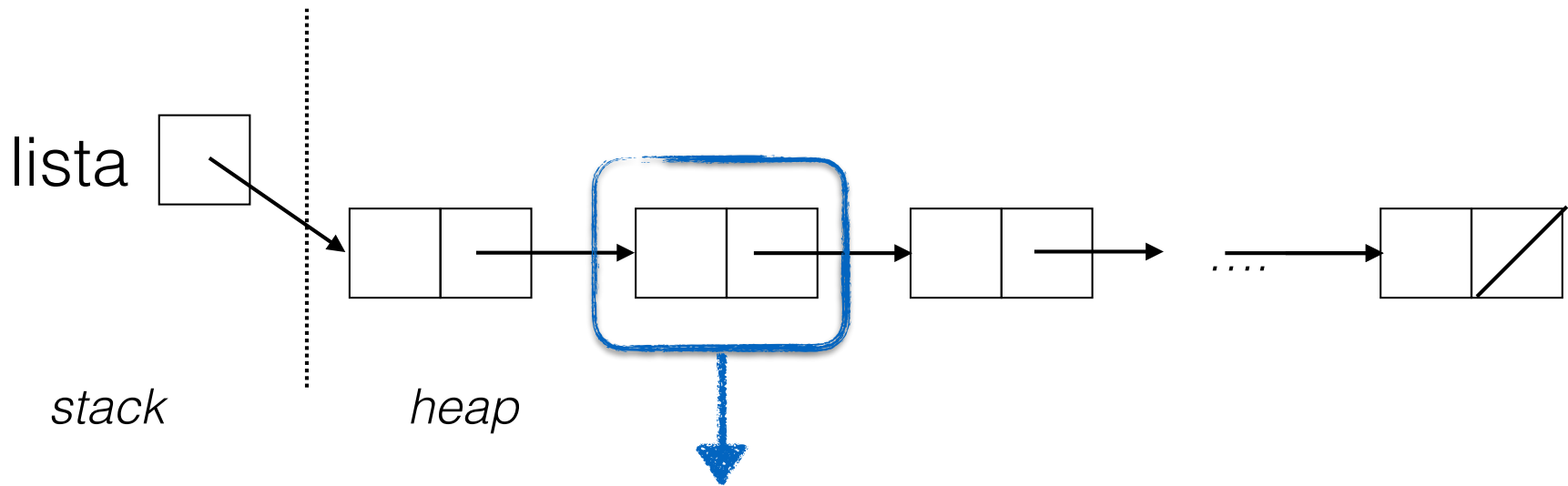
- ogni cella (in rosso) memorizza il contenuto e un puntatore alla cella successiva

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

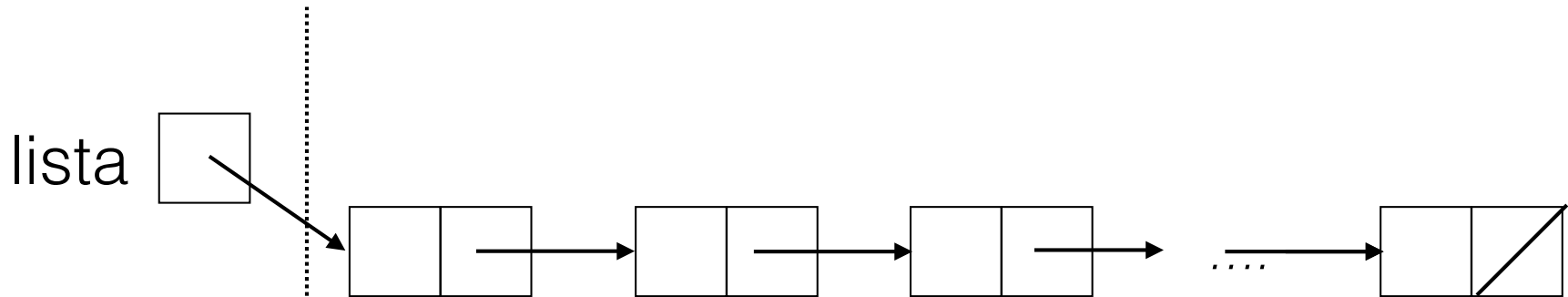
```
struct cell {
 T info; //T tipo noto a priori
 cell *next;
};
typedef cell*list;
```

- Dichiarazione e inizializzazione

```
list lista=NULL;
```

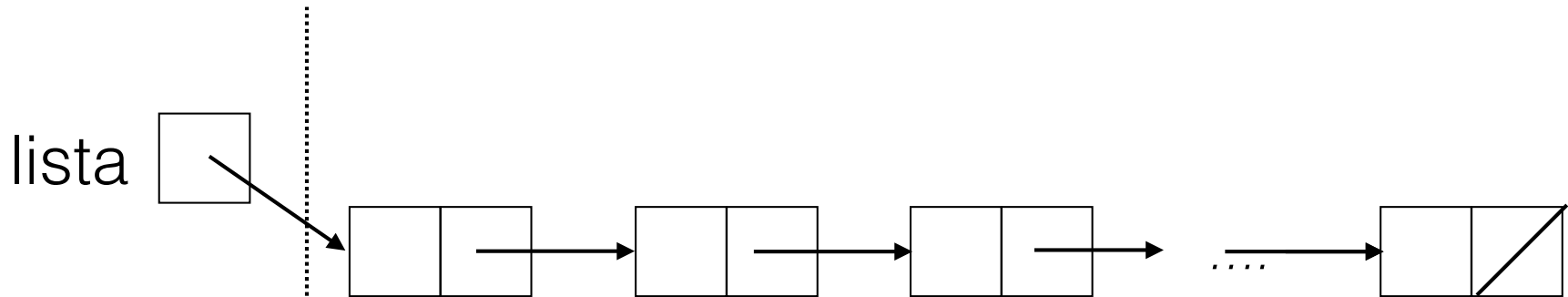


# liste semplici



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

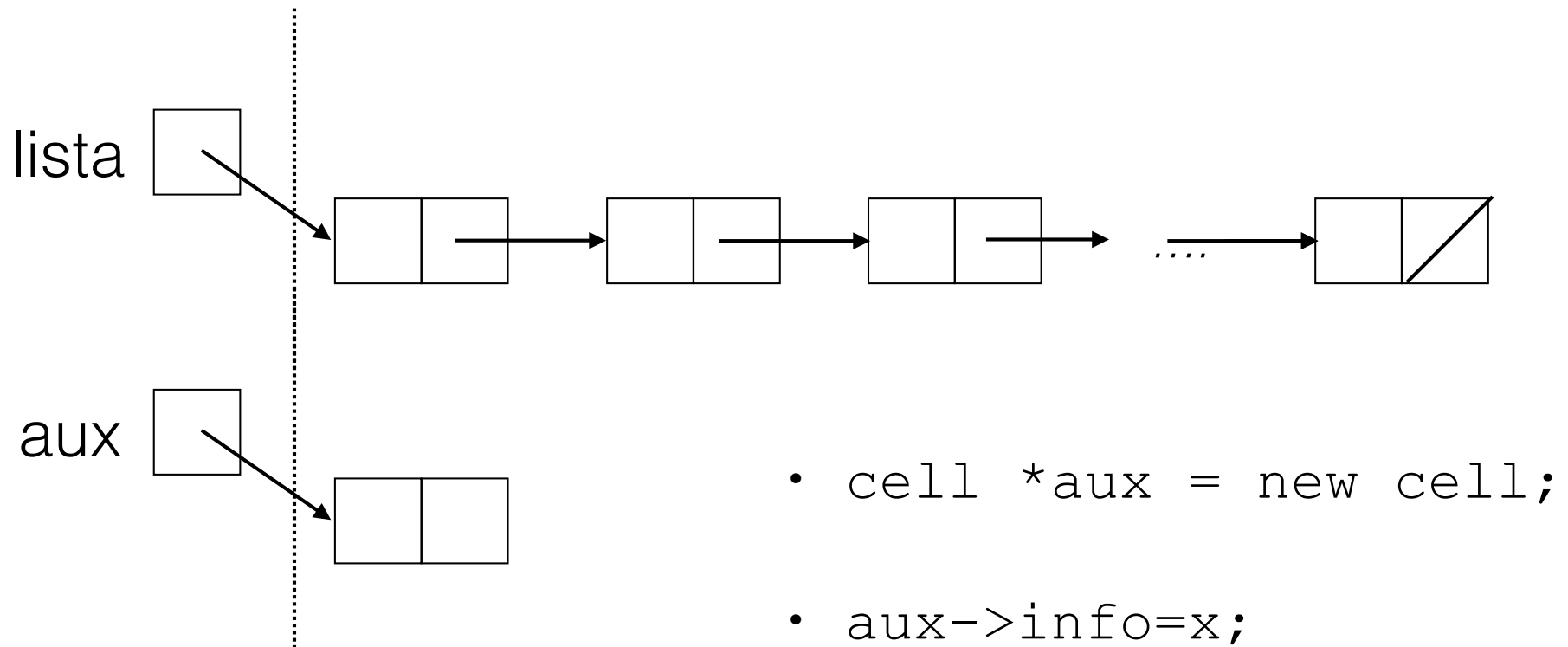
# liste semplici - inserimento in testa



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

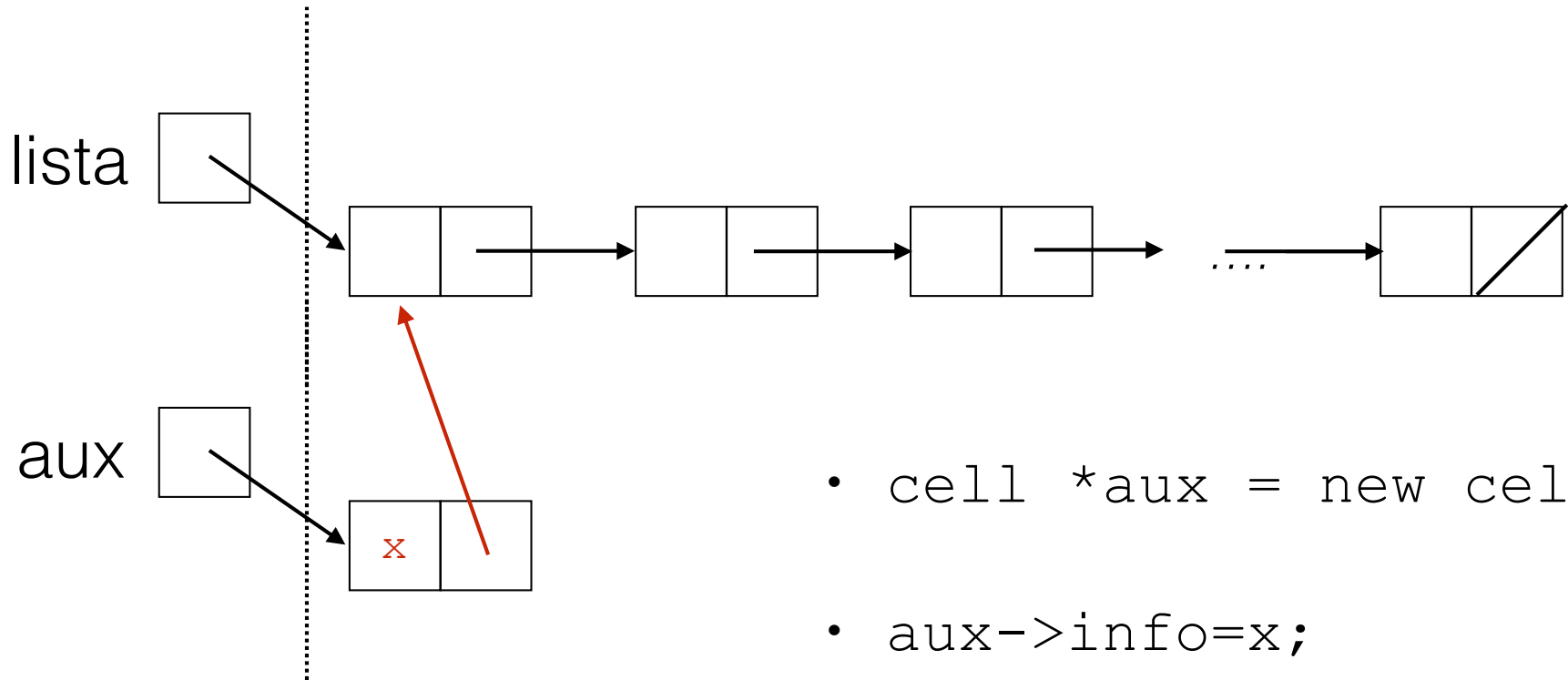
- vogliamo inserire un elemento di tipo T, sia x, in testa alla lista

# liste semplici - inserimento in testa



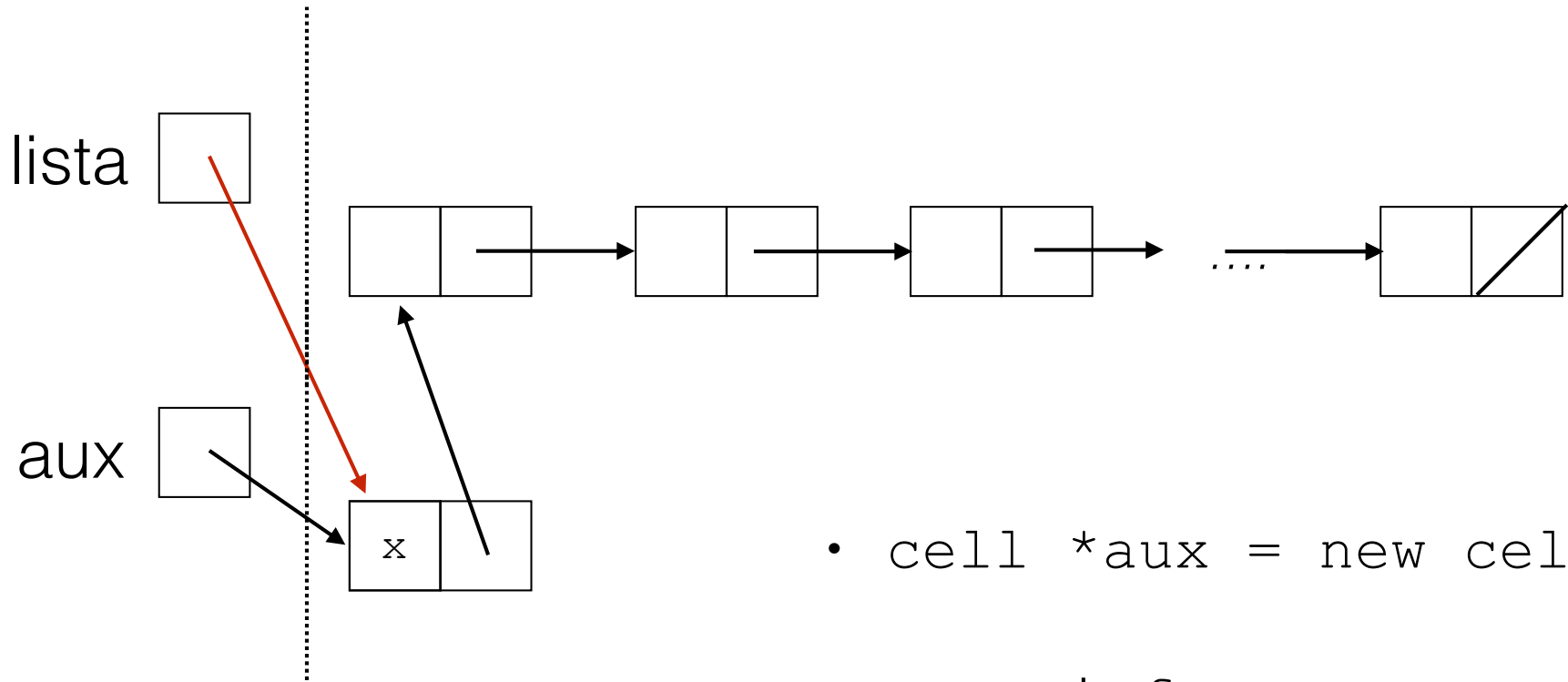
- costruiamo una variabile ausiliaria di tipo puntatore a cell e inseriamo x nel campo info (che memorizza il contenuto della cella)

# liste semplici - inserimento in testa



- `cell *aux = new cell;`
- `aux->info=x;`
- `aux->next=lista;`
- “aggiustiamo” i puntatori (appendiamo la vecchia lista in fondo ad aux)

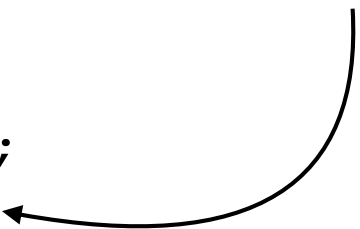
# liste semplici - inserimento in testa



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

# liste semplici - scansione sequenziale

*verifico di non essere arrivato in fondo  
se aux==NULL non posso accedere ai suoi campi*



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

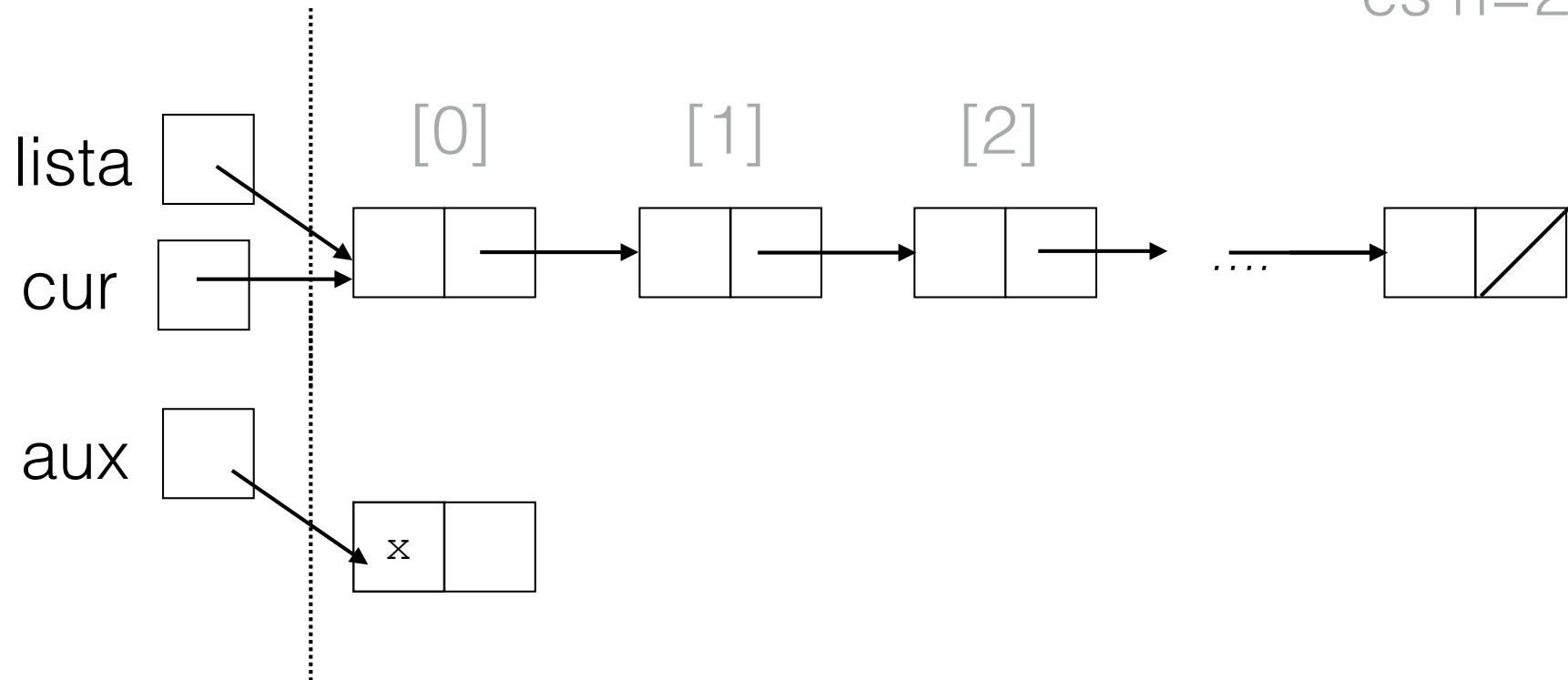
# Liste semplici - accesso all'elemento n

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!=NULL) && (cont<n))
{
    aux=aux->next;
    cont++;
}
if (aux!=NULL)
    cout << n <<"-esimo elemento e'" << aux->info << endl;
```

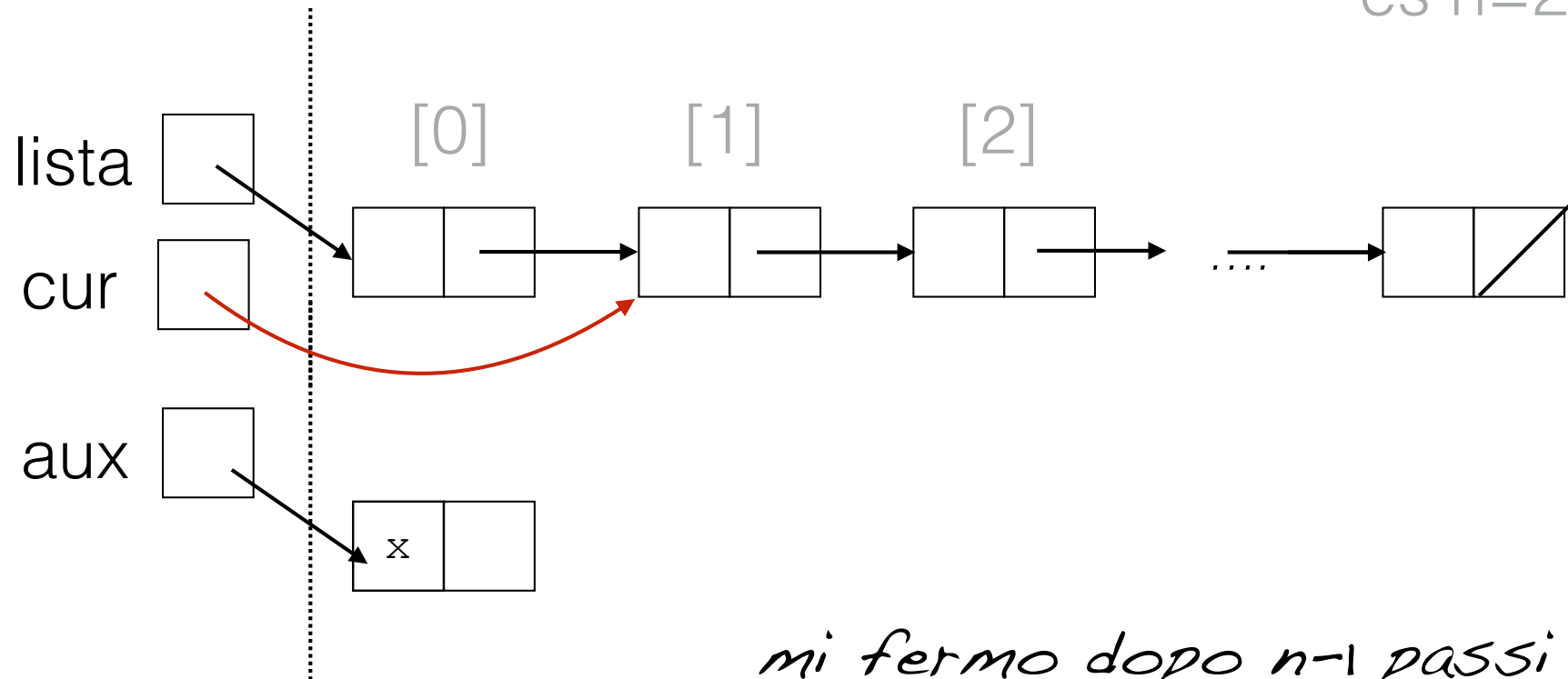

liste semplici - inserimento di un elemento in posizione n

es n=2



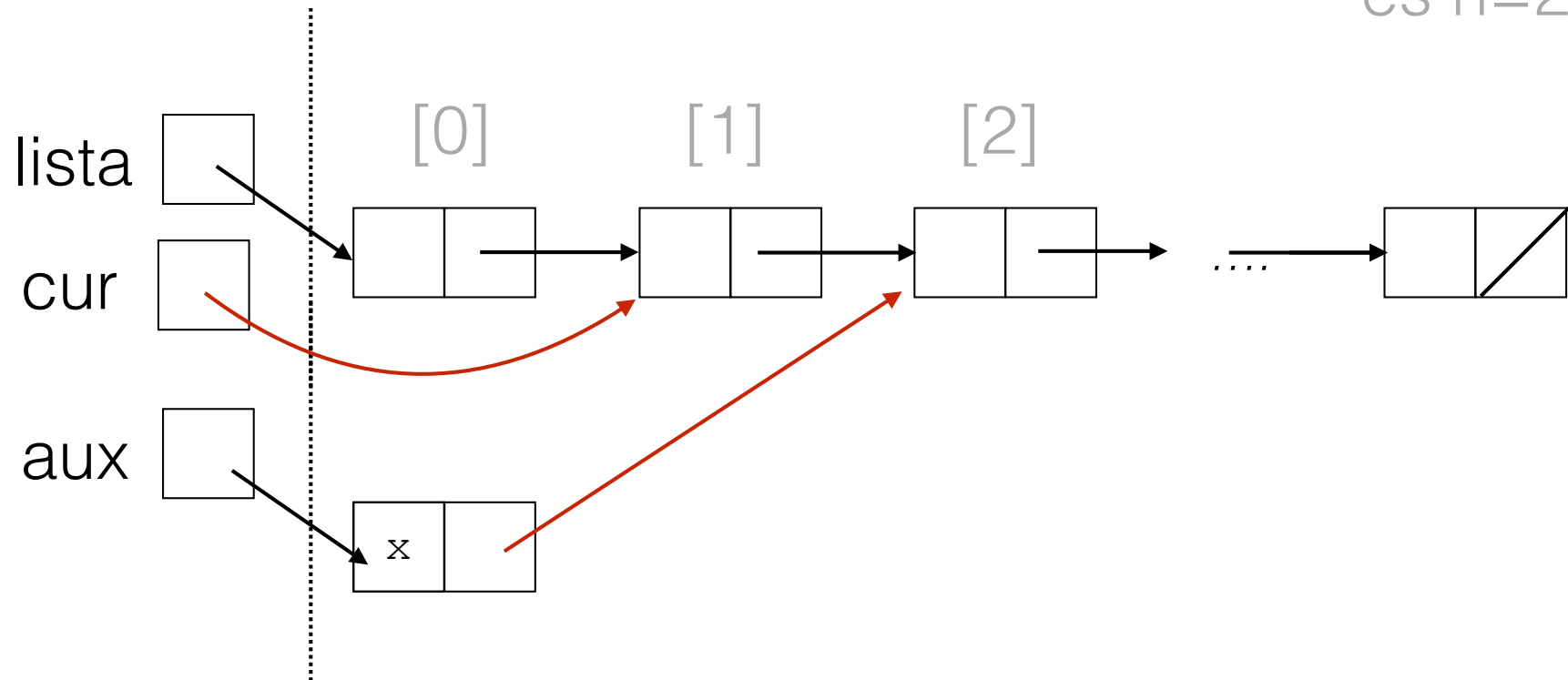
liste semplici - inserimento di un elemento in posizione n

es $n=2$



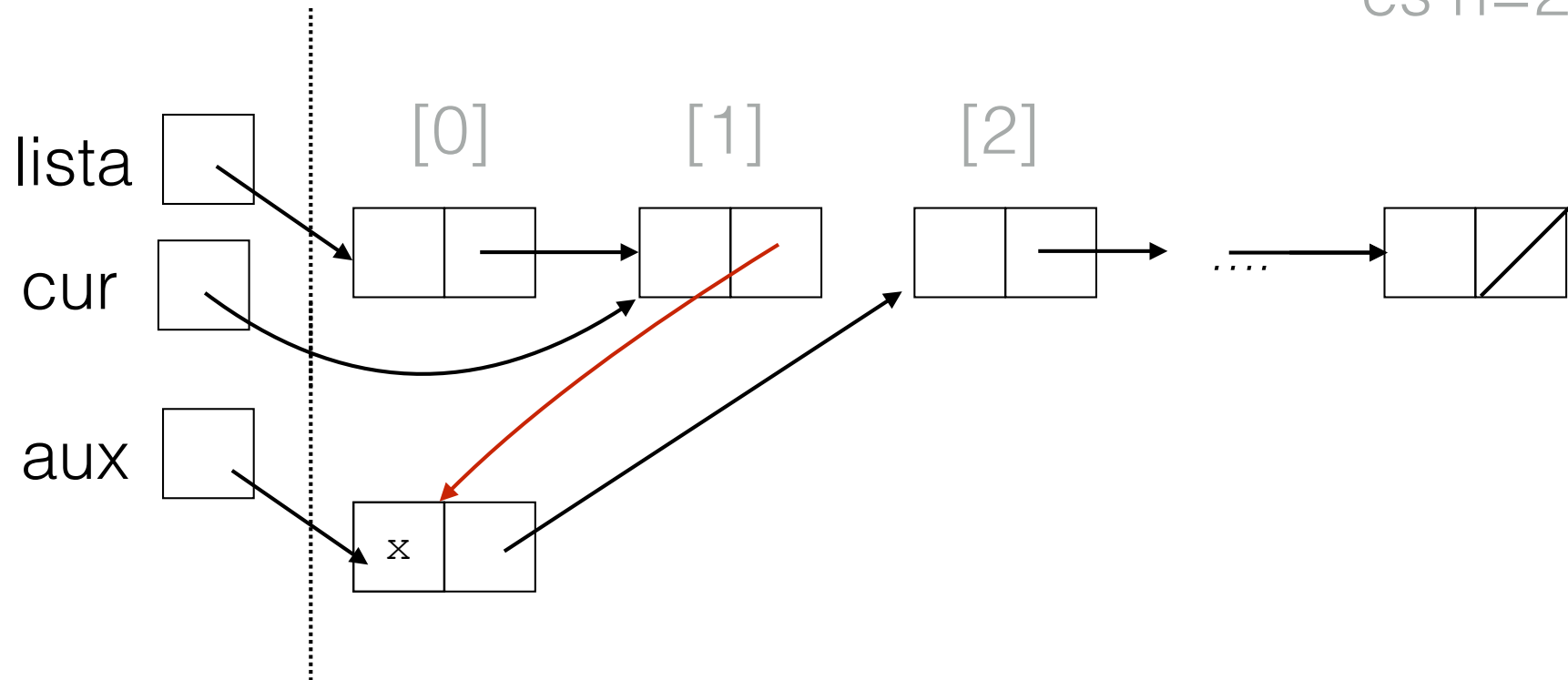
liste semplici - inserimento di un elemento in posizione n

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!=NULL)&& (cont<n))  
{  
    prev=cur;  
    cur=cur->next;  
    cont++;  
}
```

- INSERISCO

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

Lista semplice - cancellazione

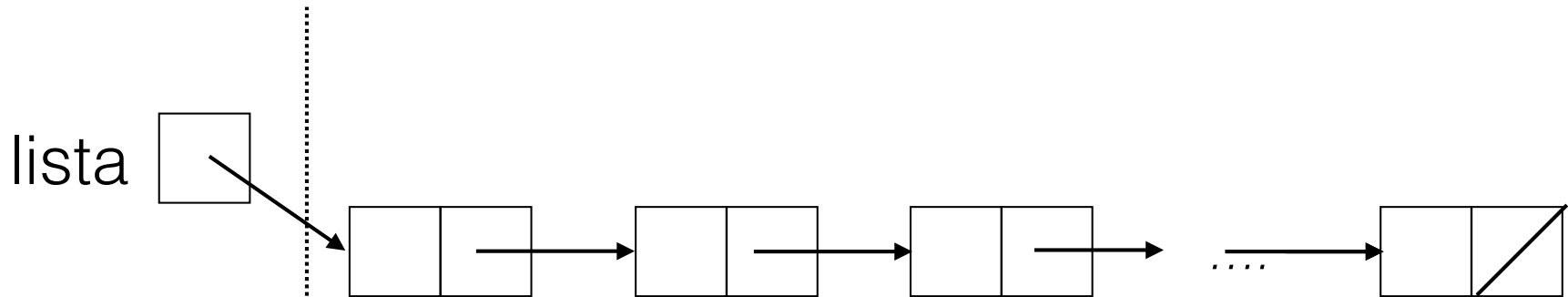
```
void CancellaElem(list &l, T elem)
{
    cell *cur=l;
    cell *prev;
    while (cur!=NULL && cur->info!=elem)
    {
        prev=cur;
        cur=cur->next;
    }
    if (cur!=NULL) //altrimenti non faccio nulla
    {
        if (cur==l)
            l=l->next; //inizio
        else
            prev->next=cur->next;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

Liste semplici ordinate - Inserimento

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

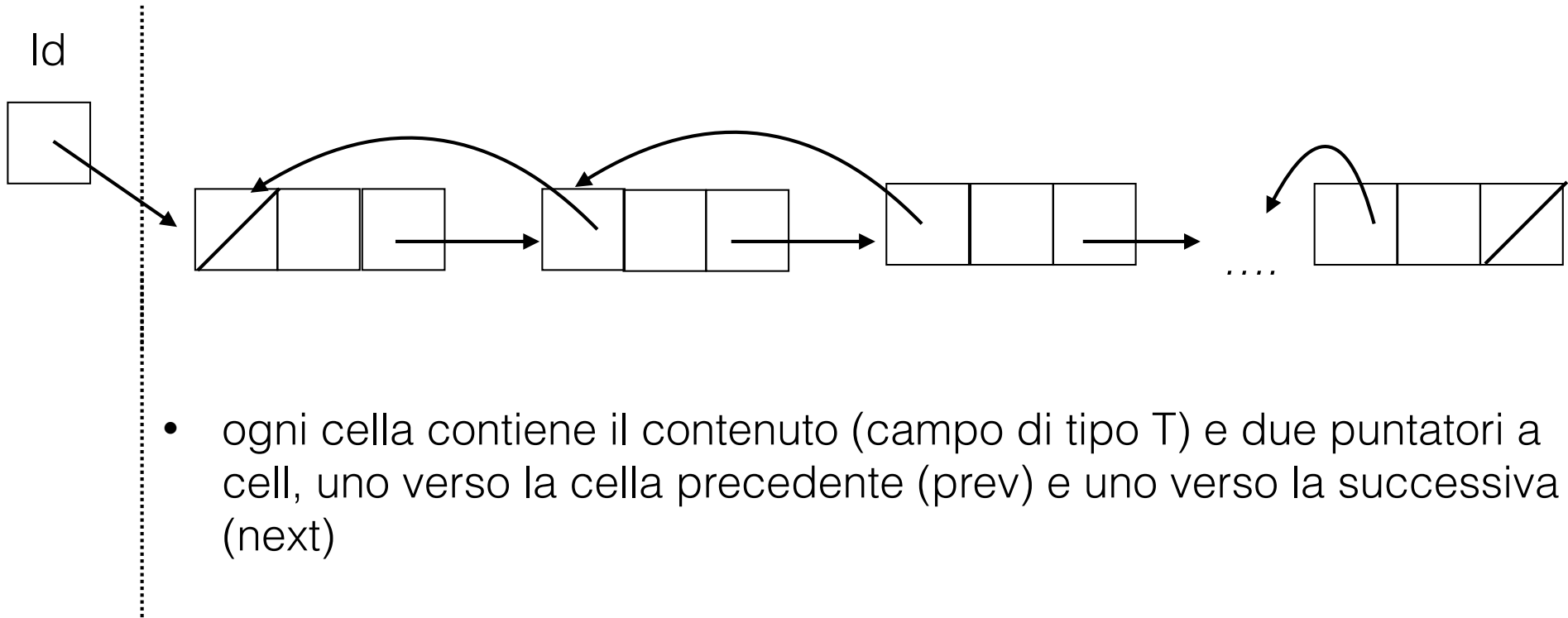
    cell *cur = l;
    cell *prev;
    while ((cur!=NULL)&& (cur->info<=elem))
    {
        prev=cur;
        cur=cur->next;
    }
    aux->next=cur;
    if (cur==l)
        l=aux;
    else
        prev->next=aux;
}
```

liste semplici



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


liste doppiamente collegate



```
typedef int T;
```

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

```
lista_doppia ld; //variabile
```

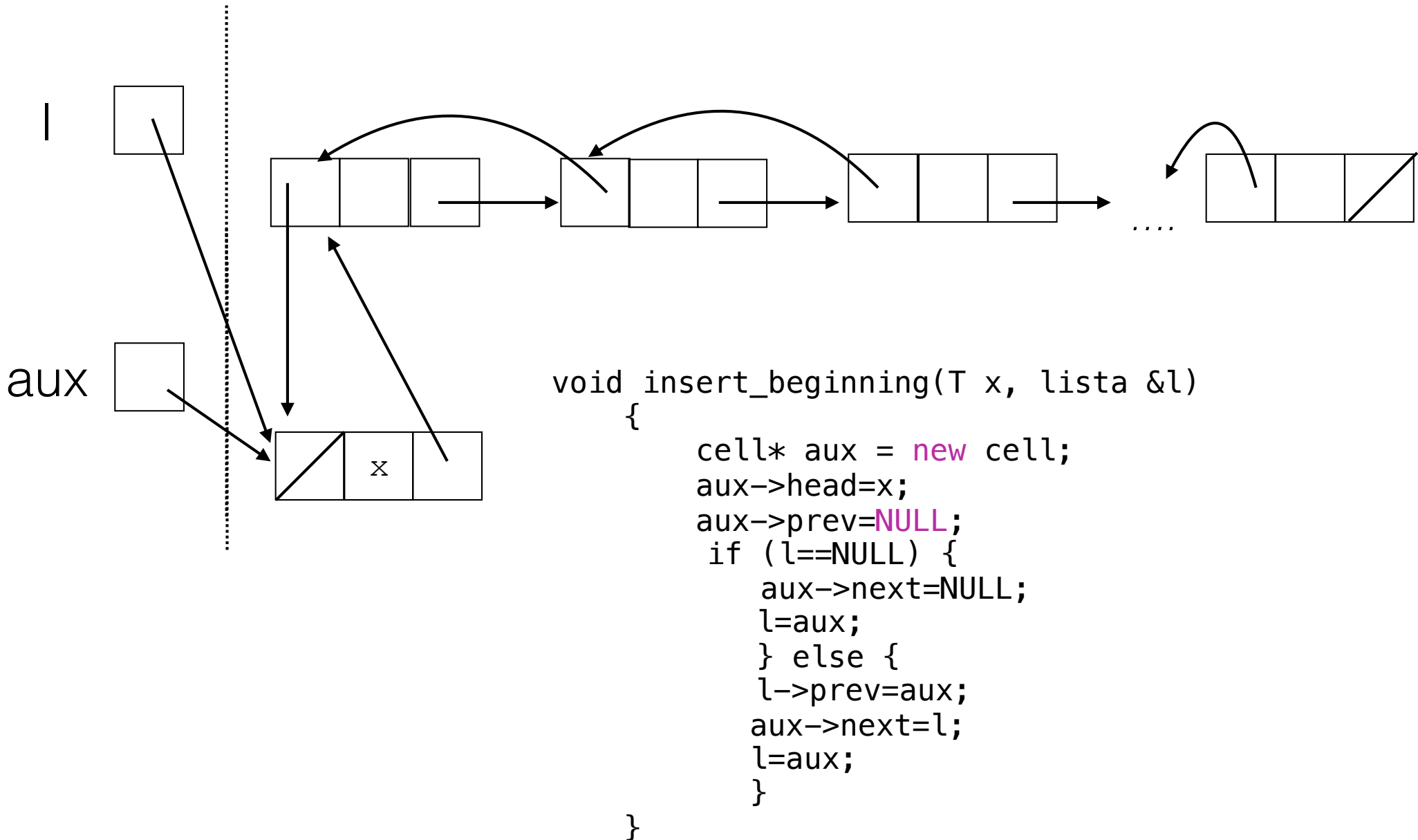
oppure

```
typedef int T;
```

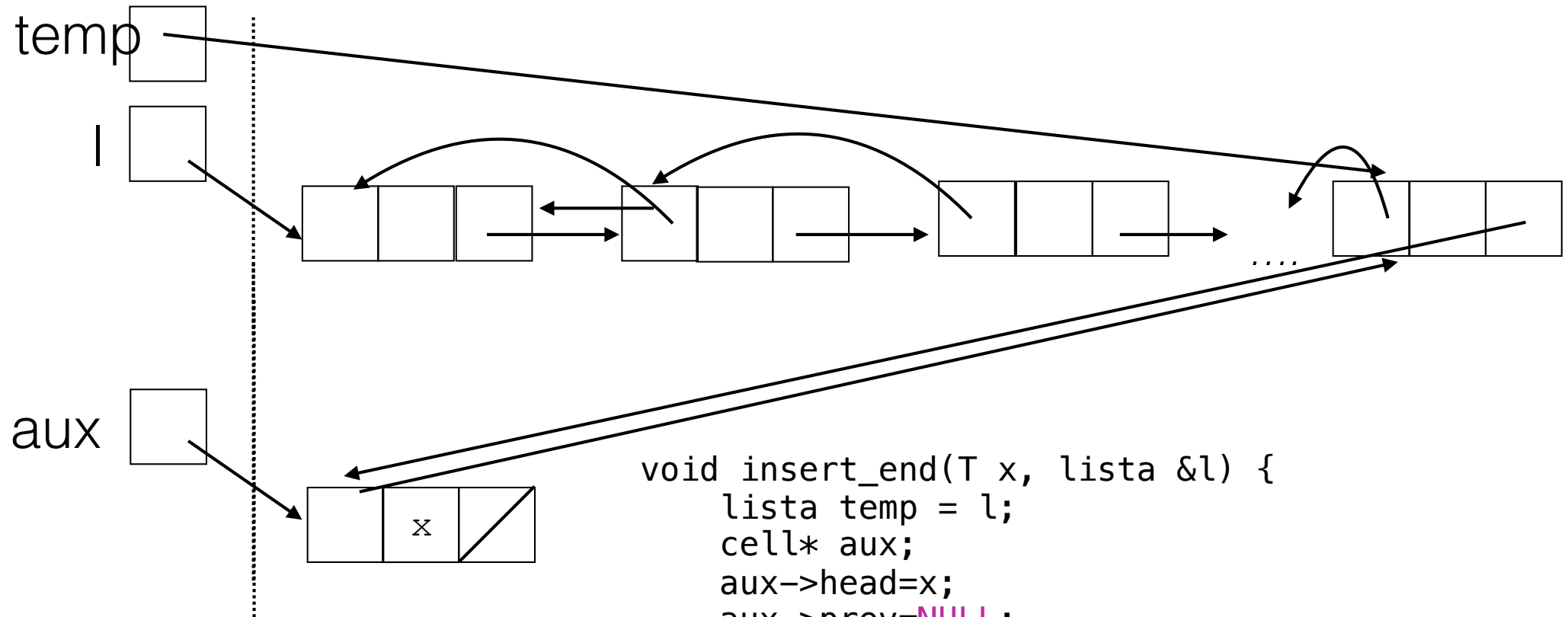
```
struct cell {  
    T head;  
    cell *next;  
    cell *prev;  
};  
typedef cell * lista_doppia;
```

```
lista_doppia ld;
```

liste doppiamente collegate inserimento in testa



liste doppiamente collegate inserimento in coda



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