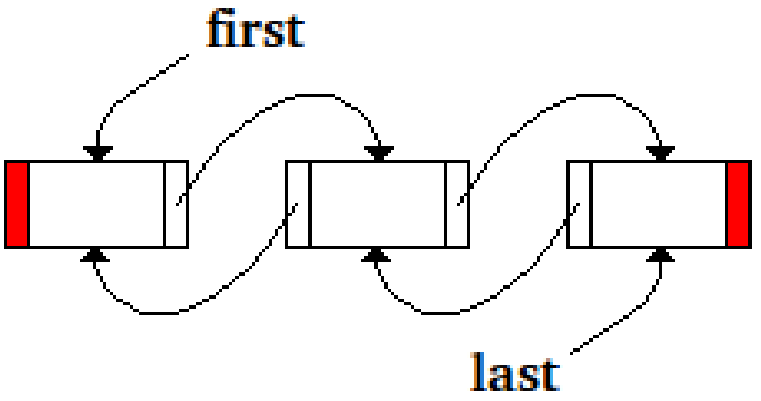
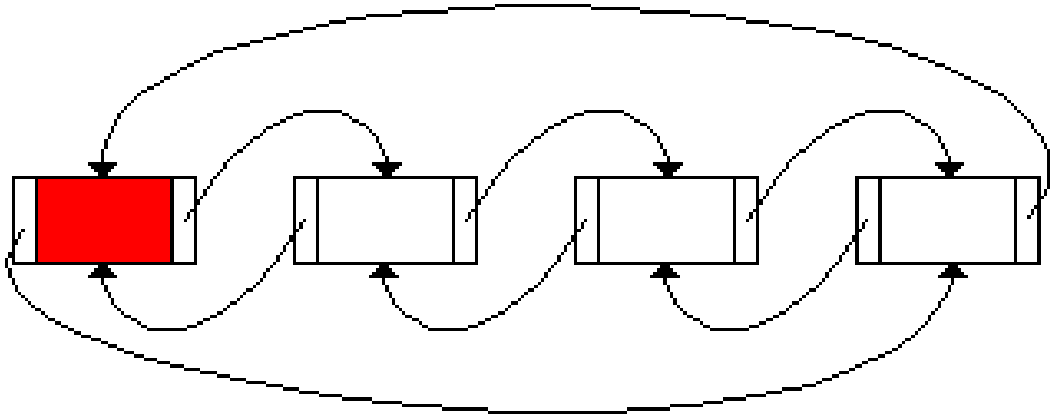
Liste circolari e con sentinella



Using NULL to mark end of list



Using a special dummy node

Variabili di tipo puntatore passate per riferimento

```
struct cell {
    DataType payload;
    cell* next;
};

typedef cell* basic_list;
```

Variabili di tipo puntatore passate per riferimento

```
#include <iostream>
#include "BasicList.h"

void head_insert(basic_list& list, DataType new_value) {
    cell* aux = new cell;
    aux->next = list;
    aux->payload = new_value;
    list = aux;
}
```

Variabili di tipo puntatore passate per riferimento

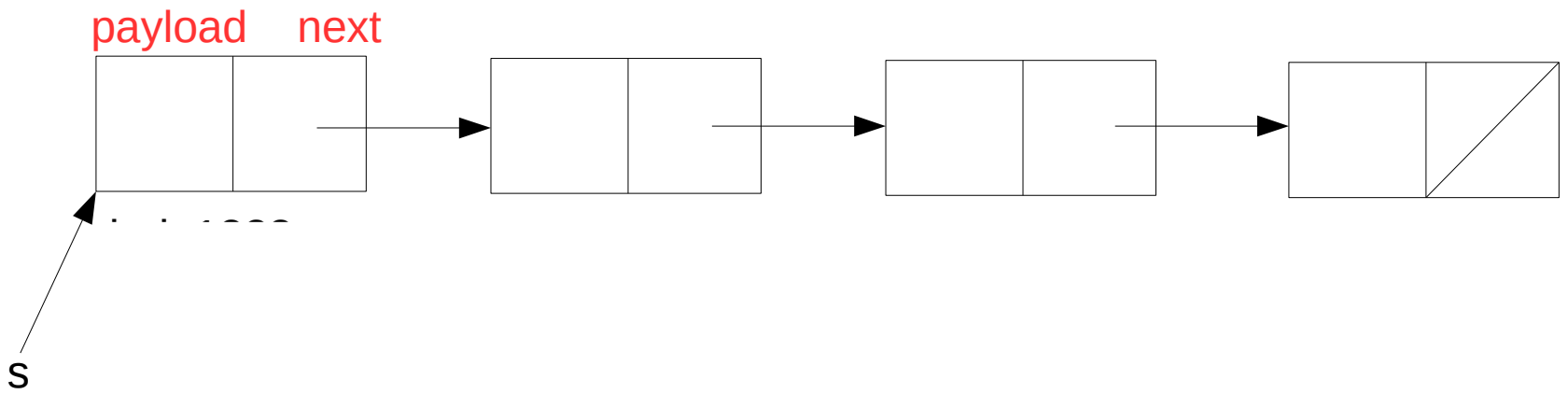
```
#include <iostream>
#include "BasicList.h"
```

cell*

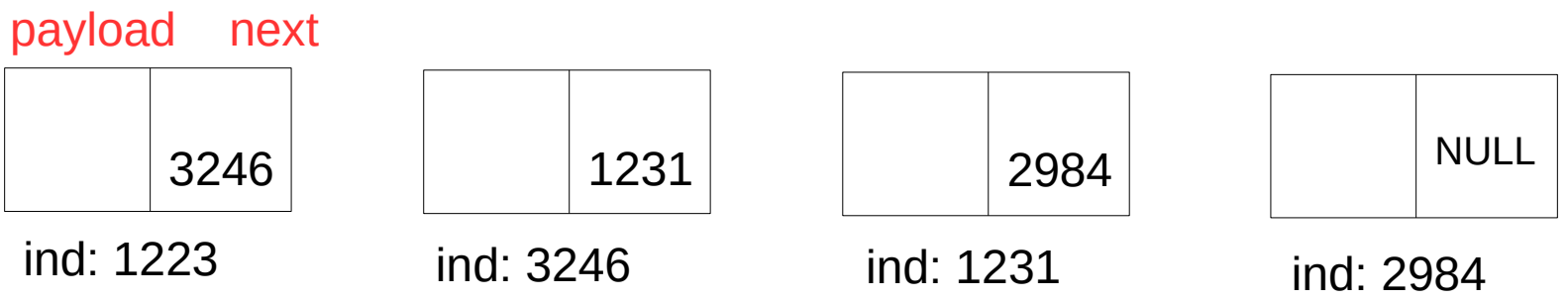
```
void head_insert(basic_list& list, DataType new_value) {
    cell* aux = new cell;
    aux->next = list;
    aux->payload = new_value;
    list = aux;
}
```

list è uno dei parametri formali di head_insert; è di tipo cell*, ovvero di tipo puntatore, passata per riferimento (&)
Perché?

Variabili di tipo puntatore passate per riferimento



Variabili di tipo puntatore passate per riferimento



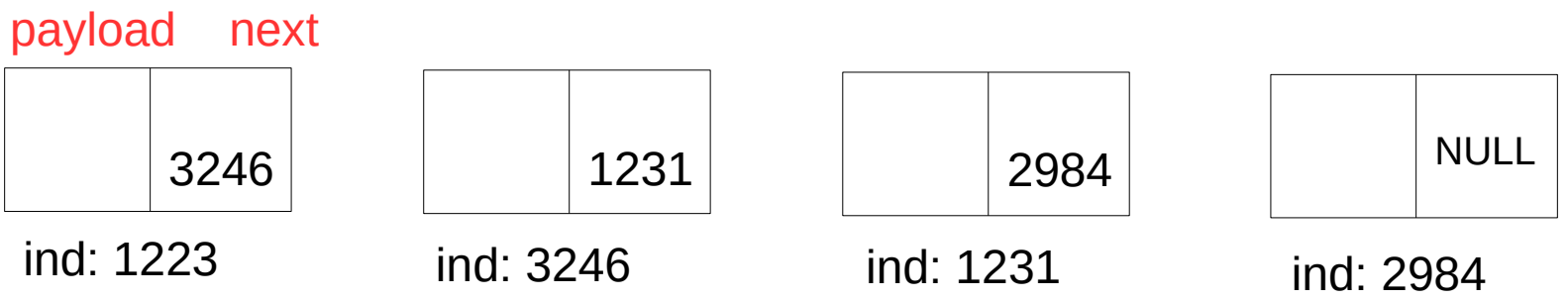
s = 1223

Cosa succede quando dal main viene chiamata `head_insert(s, 13)`?

il **parametro attuale s** sostituisce il **parametro formale list**

```
void head_insert(basic_list& list, DataType new_value) {
    cell* aux = new cell;
    aux->next = list;
    aux->payload = new_value;
    list = aux;
}
```

Variabili di tipo puntatore passate per riferimento



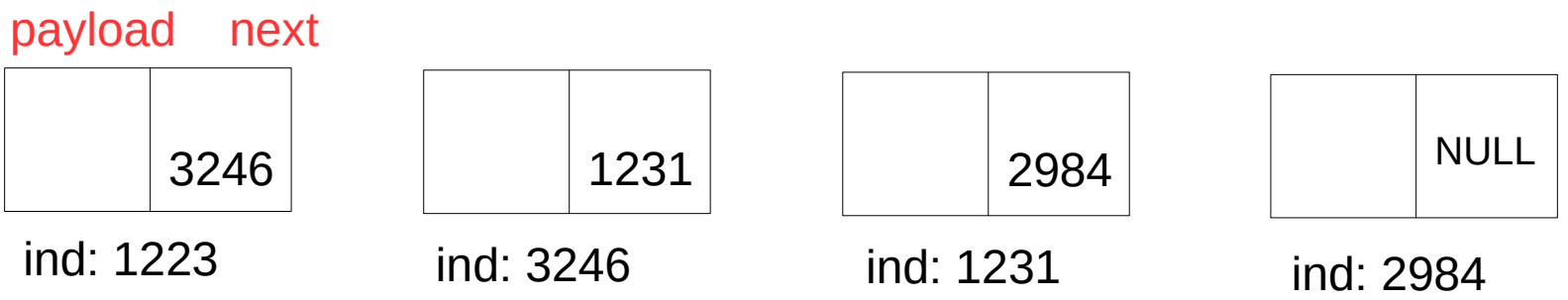
s = 1223

Cosa succede quando dal main viene chiamata `head_insert(s, 13)`?

```
void head_insert(basic_list& list, DataType new_value) {
    cell* aux = new cell;
    aux->next = list;
    aux->payload = new_value;
    list = aux;
}
```

però il **parametro attuale è passato per riferimento**: significa che **non** stiamo passando il **valore contenuto in s** ma **l'indirizzo della variabile s**; qualunque modifica faremo al parametro attuale nel corpo della funzione, permarrà anche dopo la chiamata

Variabili di tipo puntatore passate per riferimento



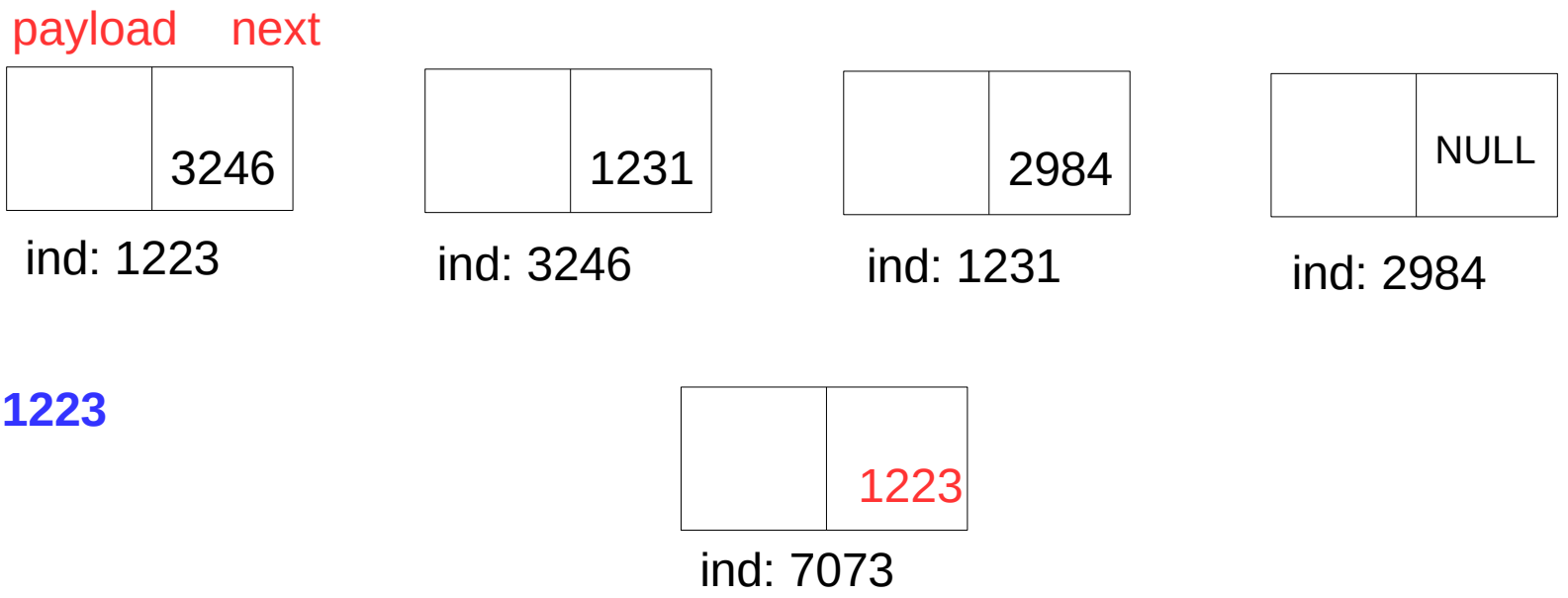
s = 1223


```
void head_insert(basic_list& list, DataType new_value) {
    cell* aux = new cell;
    aux->next = list;
    aux->payload = new_value;
    list = aux;
}
```

aux = 7073

`head_insert(s, 13)`

Variabili di tipo puntatore passate per riferimento

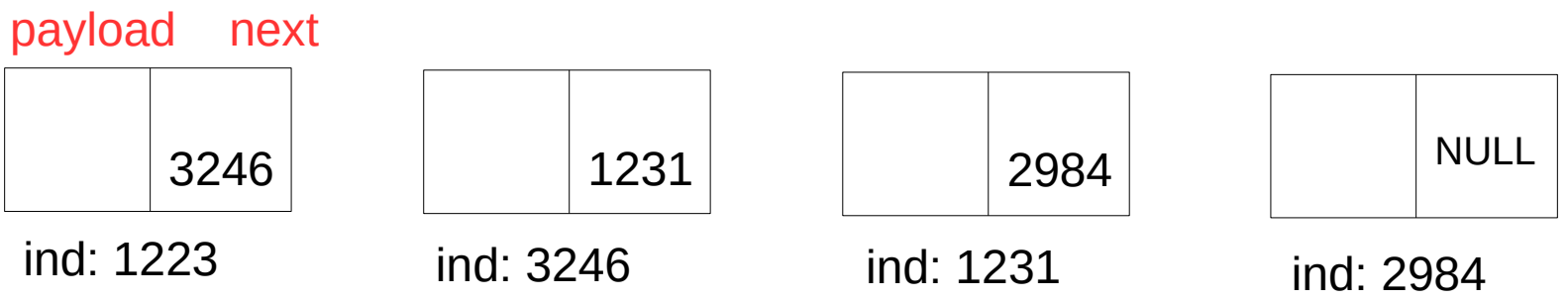



```
void head_insert(basic_list& list, DataType new_value) {
    cell* aux = new cell;
    aux->next = list;
    aux->payload = new_value;
    list = aux;
}
```

aggiorno la componente next della struct appena allocata, mettendoci il valore contenuto in s, ovvero 1223

```
head_insert(s, 13)
```

Variabili di tipo puntatore passate per riferimento

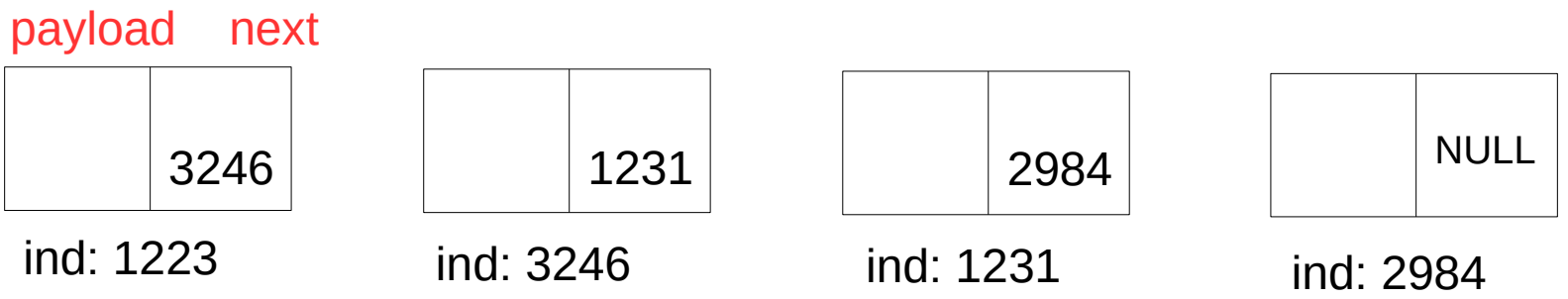



```
void head_insert(basic_list& list, DataType new_value) {
    cell* aux = new cell;
    aux->next = list;
    aux->payload = new_value;
    list = aux;
}
```

aggiorno la componente payload della struct appena allocata, mettendoci 13

`head_insert(s, 13)`

Variabili di tipo puntatore passate per riferimento

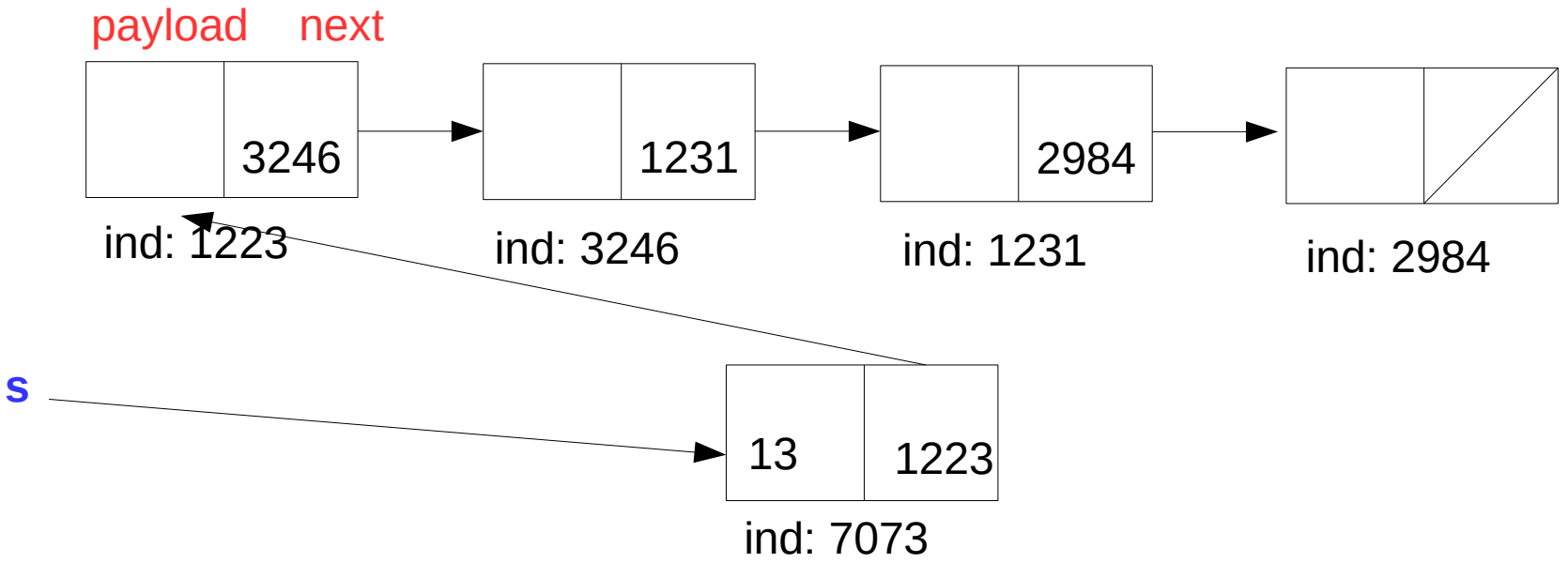



```
void head_insert(basic_list& list, DataType new_value) {
    cell* aux = new cell;
    aux->next = list;
    aux->payload = new_value;
    list = aux;
}
```

cambio il valore di s: non è più 1223 ma 7073; poiché s mi indica l'inizio della lista, questa modifica deve essere visibile anche al chiamante

```
head_insert(s, 13)
```

Variabili di tipo puntatore passate per riferimento



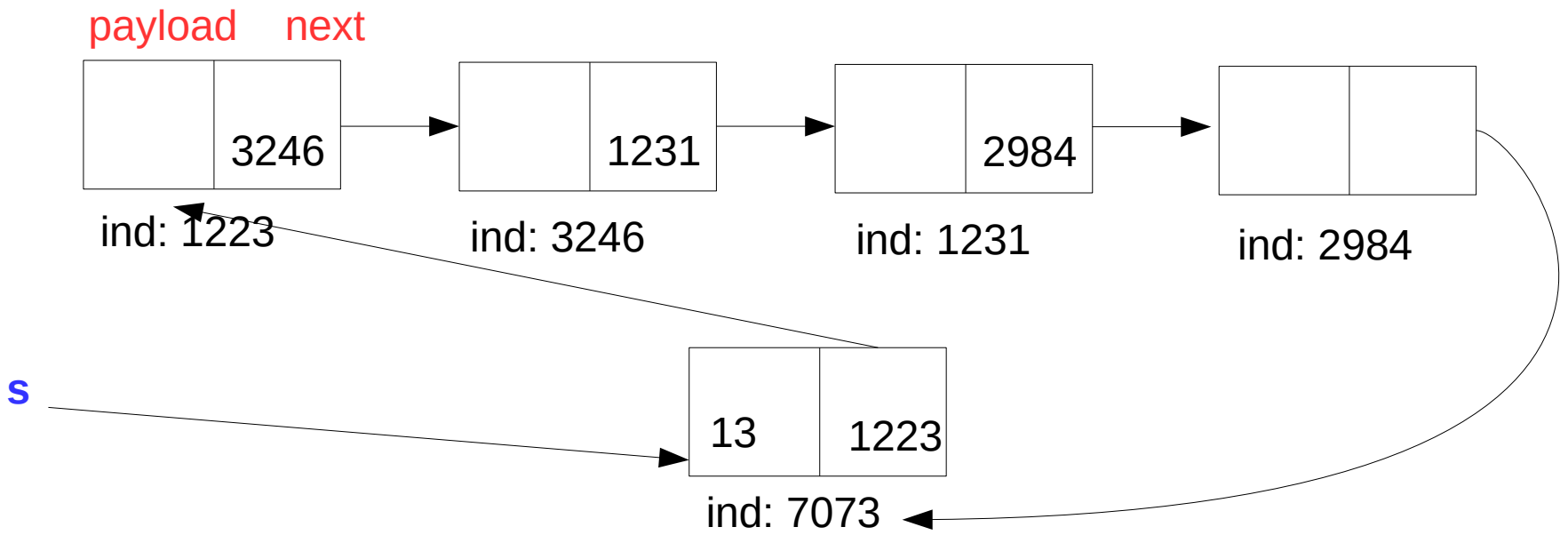

```
void head_insert(basic_list& list, DataType new_value) {
    cell* aux = new cell;
    aux->next = list;
    aux->payload = new_value;
    list = aux;
}
```

`head_insert(s, 13)`

Variabili di tipo puntatore passate per riferimento

```
void print_list(std::ostream& output_stream, basic_list list) {
    cell* aux = list;
    while (aux != nullptr) {
        WriteData(output_stream, aux->payload);
        aux = aux->next;
        output_stream << std::endl;
    }
}
```

Liste circolari



Liste circolari

```
void head_insert(basic_list& list, DataType new_value) {
    cell* aux = new cell;
    aux->next = list;   // se list è EMPTY_SEQ (sto inserendo il primo elemento), il next della struct appena creata è EMPTY_SEQ; se la lista è circolare, l'ultimo elemento deve puntare al primo; questa istruzione non va bene per liste circolari
    aux->payload = new_value;
    list = aux;
}
```

Liste circolari

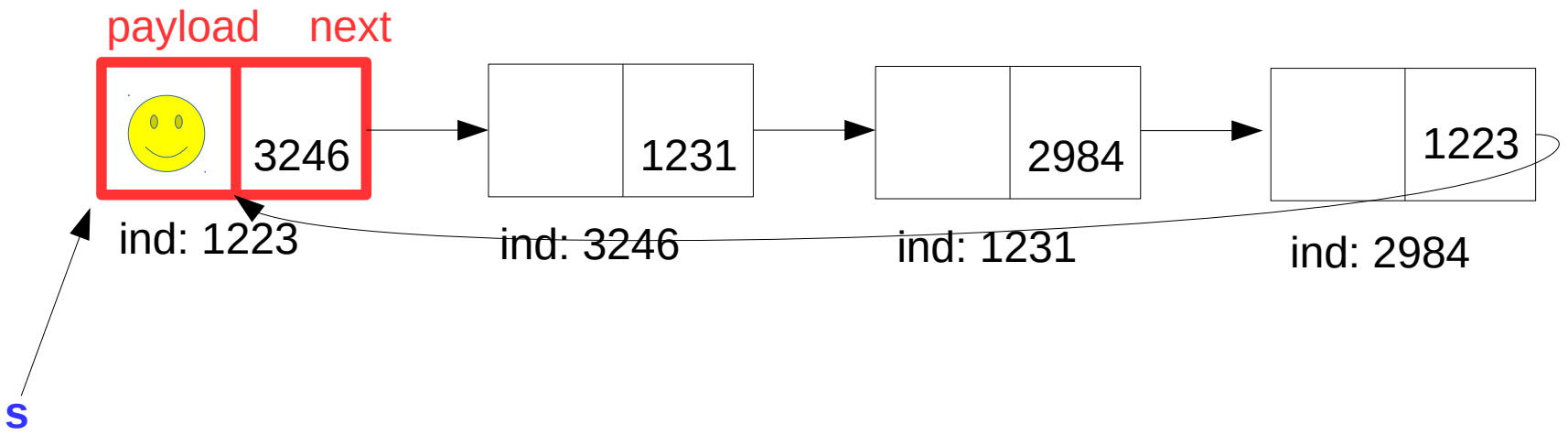
```
void head_insert(basic_list& list, DataType new_value) {
     cell* aux = new cell;
     aux->payload = new_value;
     if (list == emptyList)
          aux->next = aux;
     else
          {cell* tmp = list;
           while (tmp->next != list)
                tmp = tmp->next;
          tmp->next = aux;
          aux->next = list;
          }

    list = aux;
}
```

Liste circolari

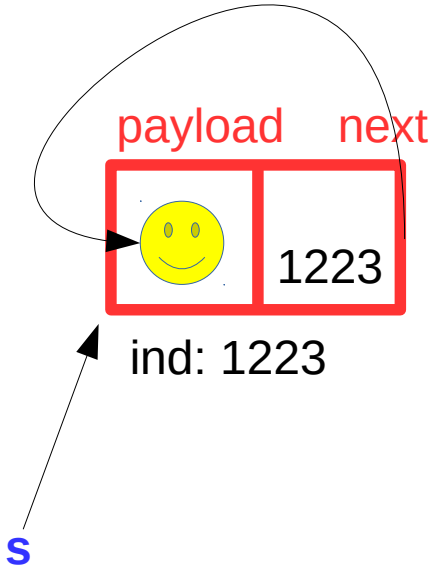
```
void print_list(std::ostream& output_stream, basic_list list) {
    if (list != emptyList)
    {
        cell* aux = list;
        while (aux->next != list) {
            WriteData(output_stream, aux->payload);
            aux = aux->next;
            output_stream << std::endl;
        }
        WriteData(output_stream, aux->payload);
    }
}
```

Liste circolari con sentinella



Il primo elemento della lista non porta informazione, è un elemento fittizio, chiamato “sentinella”, che serve solo a semplificare le operazioni evitando di dover considerare la lista emptySeq come caso base. Le informazioni iniziano da s->next (se diversa da s)

Liste circolari con sentinella



La lista vuota **non è più rappresentata da emptySeq**, ma dalla lista con solo la sentinella; il next punta alla sentinella stessa

Una volta creata una lista vuota, l'indirizzo del suo primo elemento non cambia più!!!

Se aggiungo un elemento in testa, lo metto dopo la sentinella!

Liste circolari con sentinella

// const basic_list emptyList = NULL; /* quando la lista è circolare con sentinella, la lista vuota NON E' rappresentata da NULL ma da una lista con una sola cell il cui next punta a sé stessa; bisogna avere una funzione create_empty che crea tale lista, non si puo' usare una const

```
void create_empty(basic_list& list)
{
   cell* aux = new cell;
   aux->next = aux;
   list = aux;
}
```

Liste circolari con sentinella

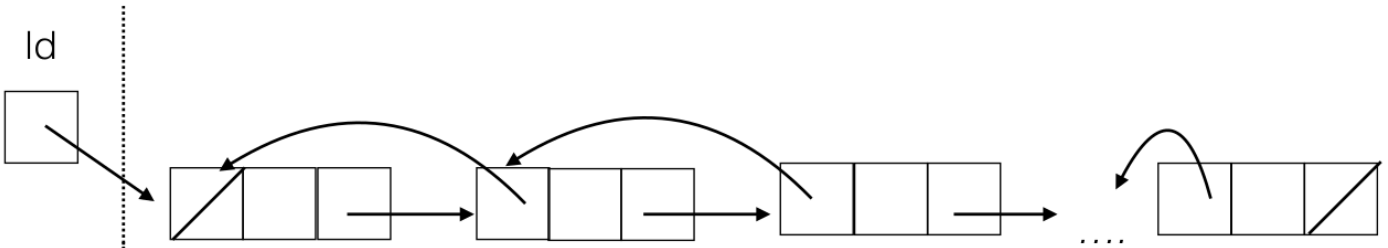
```
void head_insert(basic_list& list, DataType new_value) {
    cell* aux = new cell;
    aux->payload = new_value;
    cell* tmp = list->next;
    list->next = aux;
    aux->next = tmp;
}
```

Liste circolari con sentinella

```
void print_list(std::ostream& output_stream, basic_list list) {
    cell* aux = list->next; // devo saltare la sentinella
    while (aux != list) {
        WriteData(output_stream, aux->payload);
        aux = aux->next;
        output_stream << std::endl;
    }
}
```

Liste doppiamente collegate

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

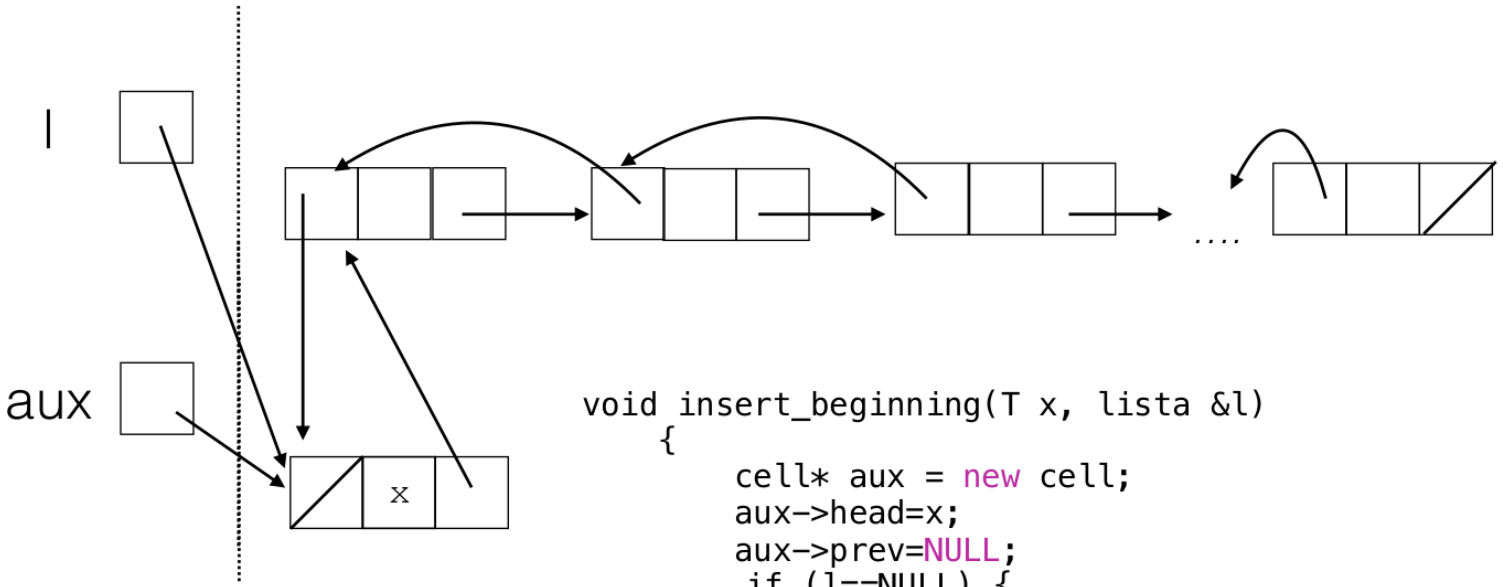
```
typedef int T;

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

lista_doppia ld;
```

Liste doppiamente collegate

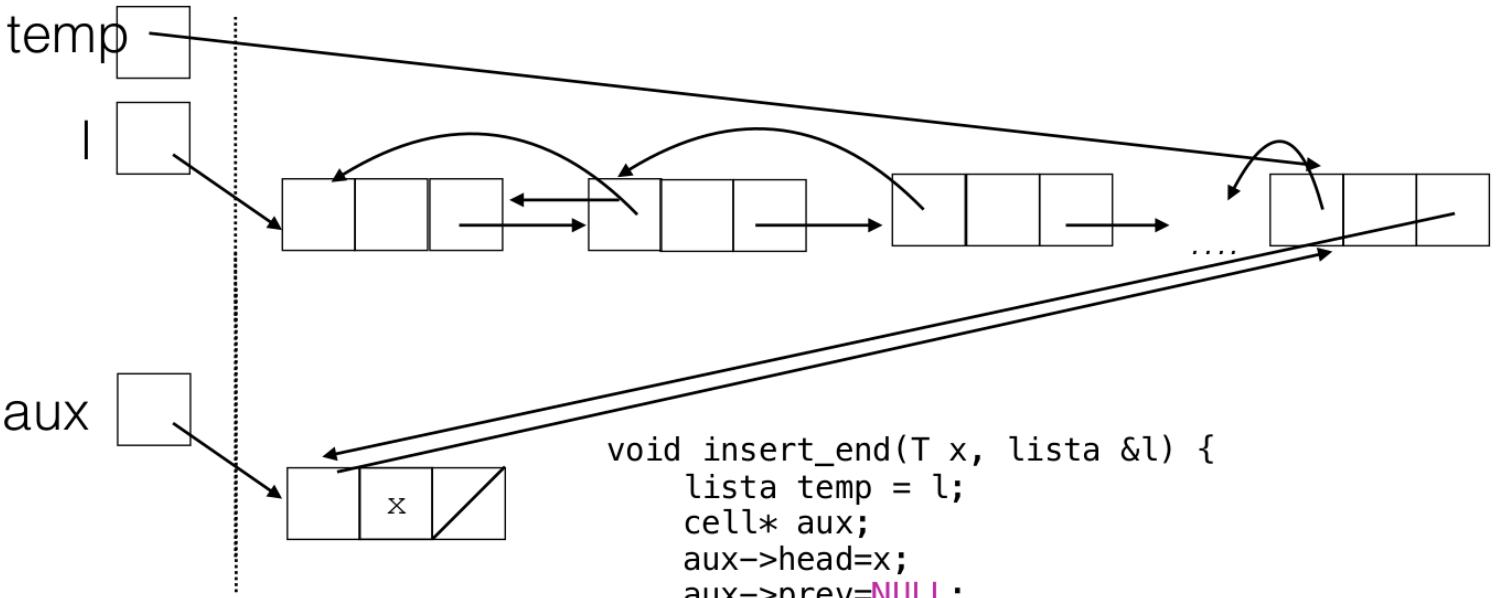
liste doppiamente collegate
inserimento in testa




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

Liste doppiamente collegate

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;
}
```

Laboratorio #1, venerdì 01/03/2019

Progettare le funzionalità delle **liste doppiamente collegate, circolari e con sentinella** presentate oggi e descritte nel file header descritto in seguito

Implementarle (una alla volta, compilando frequentemente, eseguendo e testando le funzionalità una alla volta)

```
/*********************************************************/
/* Implementazione delle operazioni nel namespace */
/*********************************************************/

void list::clear(const List& l)                /* "smantella" la lista svuotandola */
{
}

Elem list::get(int pos, const List& l)         /* restituisce l'elemento in posizione pos se presente, assumendo
- per coerenza con gli array - che la prima posizione sia indicata da 0, la seconda da 1, l'ultima da n-1;
restituisce un elemento che per convenzione si decide che rappresenta l'elemento vuoto altrimenti*/
{
}

void list::set(int pos, Elem e, const List& l)         /* modifica l'elemento in posizione pos, se la posizione e'
ammissibile */
{
}

void list::add(int pos, Elem e, const List& l)         /* inserisce l'elemento in posizione pos, shiftando a
destra gli altri elementi */
{
}

void list::addRear(Elem e,  const List& l)         /* inserisce l'elemento alla fine della lista */
{
}
```

```
void list::addFront(Elem e, const List& l)        /* inserisce l'elemento all'inizio della lista */
{
}

void list::removePos(int pos, const List& l)          /* cancella l'elemento in posizione pos dalla lista */
{
}

void list::removeEl(Elem e, const List& l)         /* cancella l'elemento elem, se presente, dalla lista */
{
}

bool list::isEmpty(const List& l)         /* restituisce true se la lista e' vuota (e' vuota se il next di l e' l
stessa */
{
}

int list::size(const List& l)          /* restituisce la dimensione della lista */
{
}

void list::createEmpty(List& l)          /* crea la lista vuota */
{
}
```