



Programmazione 1 (Canale O-Z)

Lezione 15: Strutture Dati Dinamiche (Le Liste Concatenate)

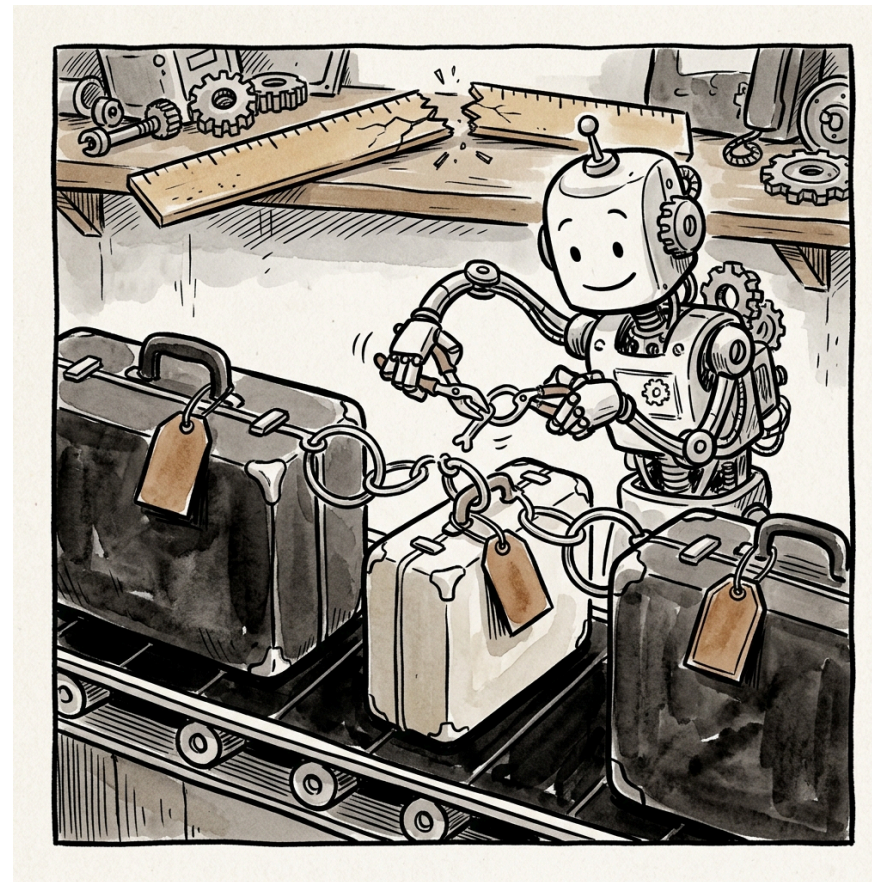
Docente: Prof. Antonino Furnari

Email: `antonino.furnari@unict.it`

Sito Web: antoninofurnari.github.io/prog1-2627

I Limiti degli Array

- Accesso immediato in tempo costante (offset dall'indice), ma:
 - i. **Ridimensionamento:** `realloc` può dover copiare tutto altrove ($O(N)$).
 - ii. **Inserimento/eliminazione in mezzo:** *shifting* di tutti gli elementi successivi ($O(N)$).
- La **Lista Concatenata Semplice** rinuncia alla contiguità fisica.



? Array di 100.000 elementi: cosa deve fare la CPU per inserire un nuovo numero all'indice 0?

VS Array vs Liste Concatenate

Array

- **Contiguità Fisica:** elementi adiacenti in RAM.
- **Accesso Casuale:** tempo costante per accedere all'indice `i`.
- **Inserimento/Rimozione:** costoso ($O(N)$) per lo shifting degli elementi.
- **Dimensione:** da riallocare (anche costosamente) per crescere.

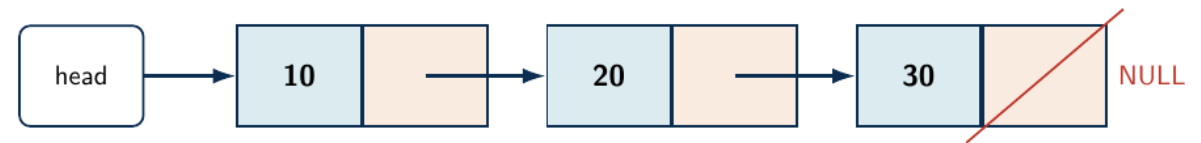
Lista Concatenata

- **Frammentazione Fisica:** nodi sparsi nello Heap.
- **Accesso Sequenziale:** niente `lista[4]`, bisogna scorrere i nodi uno ad uno.
- **Inserimento/Rimozione:** tempo costante ($O(1)$) una volta trovato il punto di aggancio.
- **Dimensione:** cresce/decrece a runtime, un nodo alla volta.

L'Anatomia di una Lista

- **Nodo:** record (`struct`) con due campi:
 - i. **Payload (dato):** l'informazione memorizzata (`int`, `char`, una `struct`...).
 - ii. **Puntatore al successore:** l'indirizzo del nodo successivo.
- **Puntatore di Testa:** variabile locale nello Stack con l'indirizzo del primo nodo. Se perdiamo la testa, perdiamo **l'intera lista**.
- **Sentinella NULL:** l'ultimo nodo punta a `NULL`.

Lista con tre nodi



Ogni cella "dato" contiene il valore; ogni cella "next" contiene l'indirizzo del nodo successivo.



Il Nodo in C e l'Allocazione Sicura

```
typedef struct Nodo {  
    int dato;  
    struct Nodo *prossimo;  
} Nodo;
```

Una funzione **helper** per non ripetere `malloc` e controllo errori:

```
Nodo* crea_nodo(int valore) {  
    Nodo *nuovo = (Nodo*) malloc(sizeof(Nodo));  
    if (nuovo == NULL) {  
        fprintf(stderr, "Errore fatale: memoria Heap esaurita.\n");  
        exit(EXIT_FAILURE);  
    }  
    nuovo->dato = valore;  
    nuovo->prossimo = NULL; // Nasce scollegato  
    return nuovo;  
}
```

- Servono `<stdio.h>`, `<stdlib.h>` (per `malloc`, `exit`) e `<stdbool.h>` (per `bool`).

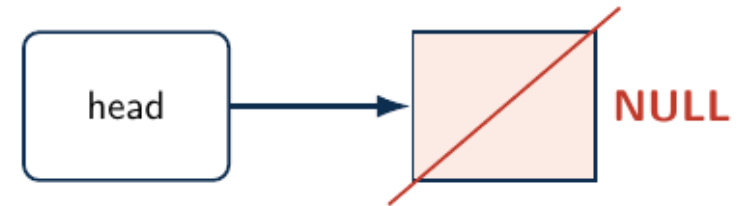
● La Lista Vuota

- Appena creata, la lista è solo la **testa**, che vale `NULL`:

```
Nodo *testa = NULL;
```

- Verifica della lista vuota:

```
bool is_empty(Nodo *testa) {  
    return testa == NULL;  
}
```



head punta a NULL: nessun nodo esiste, la lista non ha elementi.

? Senza indici, come facciamo a visitare tutti i nodi della lista?



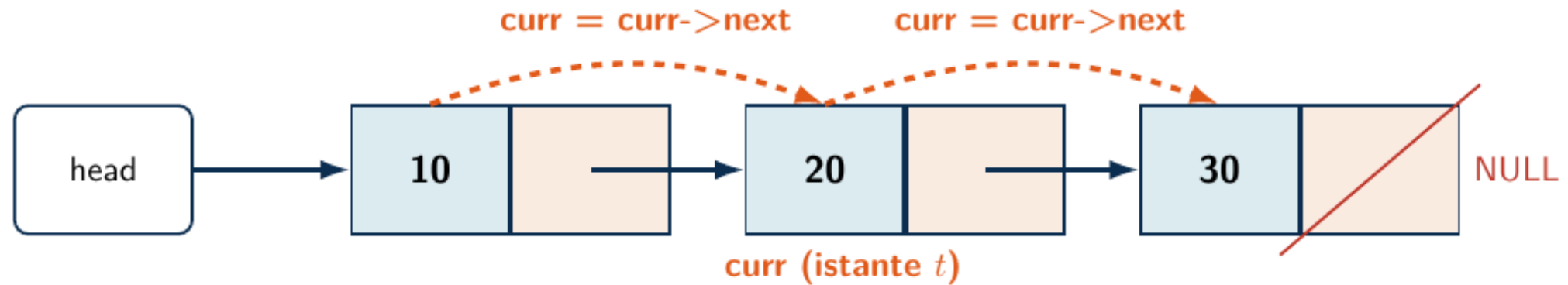
Scorrimento Sequenziale e Stampa

```
void stampa_lista(Nodo *testa) {  
    Nodo *corrente = testa;  
    printf("Testa → ");  
    while (corrente ≠ NULL) {  
        printf("[%d] → ", corrente→dato);  
        corrente = corrente→prossimo; // Passo di avanzamento  
    }  
    printf("NULL\n");  
}
```

- Niente indici numerici: un puntatore ausiliario `corrente` parte dalla testa e avanza copiando il campo `prossimo`.
- `testa` non viene alterata.



Visualizzazione: Attraversamento



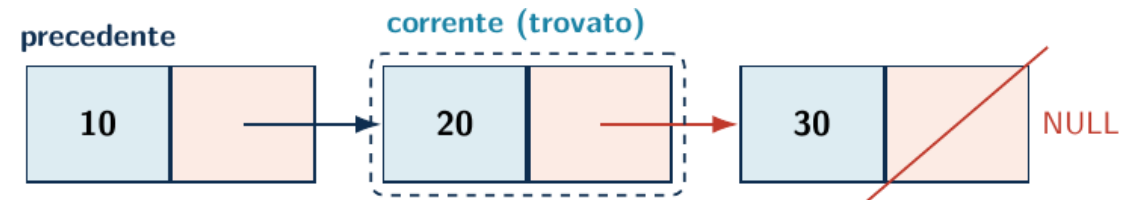
Il ciclo termina quando *curr* diventa NULL: tutti i nodi sono stati visitati.

- Con la lista `10 → 20 → 30` stampa: `Testa → [10] → [20] → [30] → NULL`

Ricerca di un Elemento

```
bool cerca_elemento(Nodo *testa, int chiave,
                   Nodo **nodo_trovato) {
    Nodo *corrente = testa;
    while (corrente != NULL) {
        if (corrente->dato == chiave) {
            if (nodo_trovato != NULL) {
                *nodo_trovato = corrente;
            }
            return true;
        }
        corrente = corrente->prossimo;
    }
    if (nodo_trovato != NULL) {
        *nodo_trovato = NULL;
    }
    return false;
}
```

- Ricerca **lineare**: si scorrono i nodi uno ad uno.
- Restituisce un `bool` e scrive l'indirizzo del nodo nel puntatore fornito dal chiamante (`Nodo **`).



Il ciclo di ricerca si ferma su `corrente->dato == 20`.



Chirurgia dei Puntatori: Inserimento in Testa

- Nessun nodo si sposta in memoria: si **riconfigurano i collegamenti**.
- Tempo costante:
 - i. Si alloca un nuovo nodo.
 - ii. Il suo `prossimo` punta alla vecchia testa.
 - iii. La testa diventa il nuovo nodo.

```
Nodo* inserisci_testa(Nodo *testa, int valore) {  
    Nodo *nuovo = crea_nodo(valore);  
    nuovo->prossimo = testa;  
    return nuovo; // Diventa la nuova testa  
}
```

- Nel `main`: `testa = inserisci_testa(testa, 42);`

Inserimento in Testa — Passo 1

Lista di partenza `10 → 20`, inseriamo `5`:

```
Nodo *nuovo = crea_nodo(5);
```

```
testa → [ 10 | ● ] → [ 20 | NULL ]
```

```
nuovo → [ 5 | NULL ]           (nasce scollegato)
```



Inserimento in Testa — Passo 2

```
nuovo→prossimo = testa;
```

```
nuovo → [ 5 | ● ] —┐  
                    ▼  
testa —————> [ 10 | ● ] —> [ 20 | NULL ]
```

- Il nuovo nodo **eredita** il resto della lista come suo seguito.

Inserimento in Testa — Passo 3

```
return nuovo;           // nella funzione  
testa = inserisci_testa(testa, 5); // nel main
```

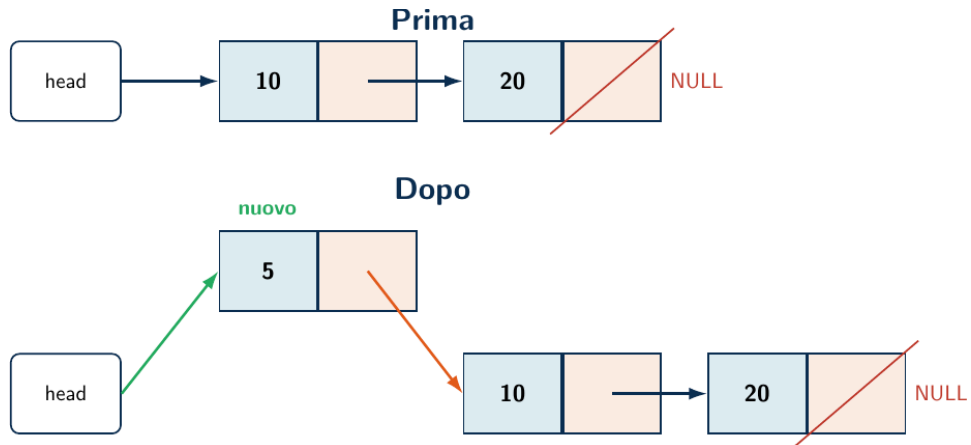
testa → [5 | ●] → [10 | ●] → [20 | NULL]

? E se la lista era vuota (`testa = NULL`)? Il codice funziona ancora?

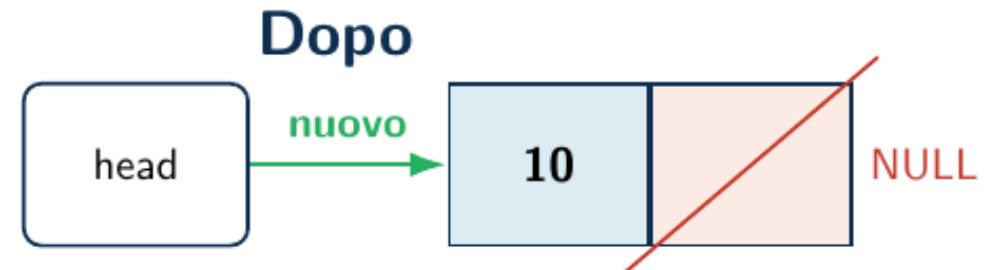
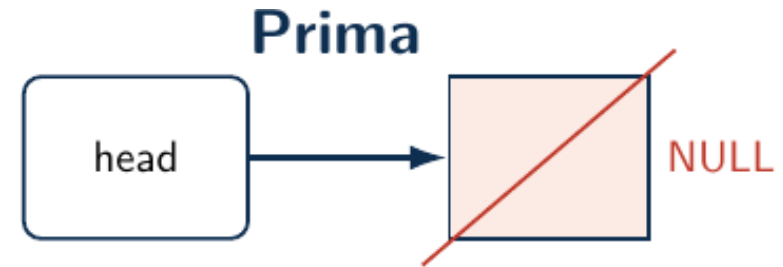


Visualizzazione: Inserimento in Testa

Lista Non Vuota



Lista Vuota



- Con `testa = NULL`: `nuovo → prossimo = NULL` e il nuovo nodo diventa l'unico elemento.



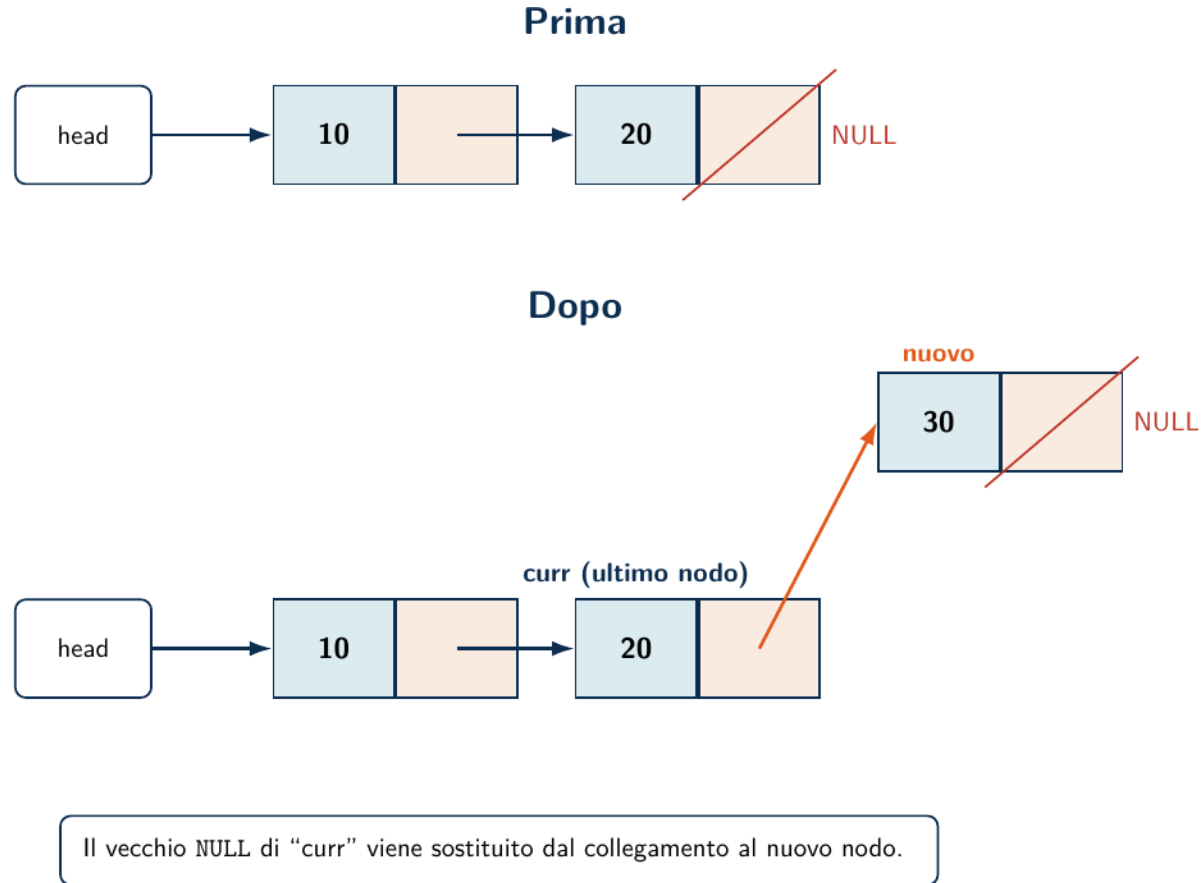
Inserimento in Coda

- Bisogna trovare l'**ultimo** nodo (quello che punta a `NULL`) e collegarlo al nuovo:
 - lista vuota ➡ il nuovo nodo diventa la testa;
 - altrimenti scansione lineare fino all'ultimo nodo.

```
Nodo* inserisci_coda(Nodo *testa, int valore) {
    Nodo *nuovo = crea_nodo(valore);
    if (testa == NULL) {
        return nuovo;
    }
    Nodo *corrente = testa;
    while (corrente->prossimo != NULL) { // ci fermiamo ALL'ULTIMO, non dopo!
        corrente = corrente->prossimo;
    }
    corrente->prossimo = nuovo;
    return testa;
}
```



Visualizzazione: Inserimento in Coda



? Lista ordinata $10 \rightarrow 20 \rightarrow 30$: come inseriamo 25 mantenendo l'ordine?



Inserimento Ordinato: l'Idea

- La lista semplice è **a senso unico**: superato il punto di inserimento non si torna indietro per aggiornare il nodo che precede.
- **Tecnica dei due puntatori sfalsati** di un passo: `precedente` e `corrente`.
- Casi da gestire:
 - i. lista vuota o valore minore della testa ➡ inserimento in testa;
 - ii. altrimenti si avanza finché `corrente→dato < valore`, poi si inserisce **tra** `precedente` e `corrente`.
- Con `25` in `10 → 20 → 30`: ci si ferma con `precedente` su 20 e `corrente` su 30.

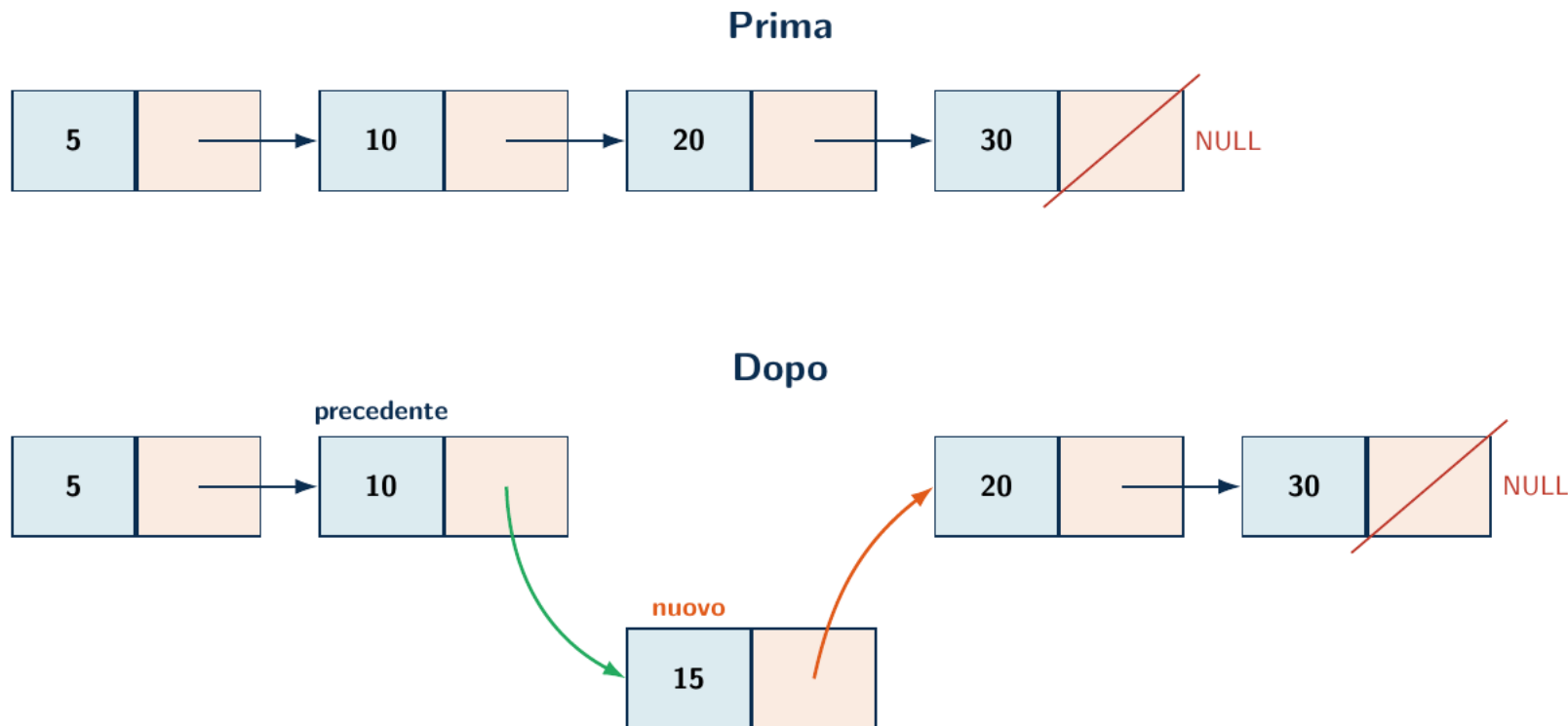


Inserimento Ordinato: il Codice

```
Nodo* inserisci_ordinato(Nodo *testa, int valore) {
    Nodo *nuovo = crea_nodo(valore);
    // Caso 1: Lista vuota o inserimento in testa
    if (testa == NULL || valore < testa->dato) {
        nuovo->prossimo = testa;
        return nuovo;
    }
    // Caso 2: Ricerca della posizione intermedia
    Nodo *precedente = testa;
    Nodo *corrente = testa->prossimo;
    while (corrente != NULL && corrente->dato < valore) {
        precedente = corrente;
        corrente = corrente->prossimo;
    }
    // Chirurgia di collegamento
    precedente->prossimo = nuovo;
    nuovo->prossimo = corrente;
    return testa;
}
```

Visualizzazione: Inserimento Ordinato

- Il valore 15 deve essere inserito tra 10 (precedente) e 20 (corrente).
- precedente→prossimo viene indirizzato a nuovo, e il prossimo del nuovo nodo a corrente.



Il valore 15 va tra 10 e 20: si inserisce quindi tra precedente e corrente.



La Chirurgia delle Cancellazioni

- Rimuovere un nodo significa:
 - **scavalcarlo** modificando il puntatore del nodo precedente;
 - eliminarlo fisicamente dallo Heap con `free`.
- **! Pericolo di Memory Leak:** salvare l'indirizzo del nodo da eliminare in un puntatore di supporto **prima** di modificare le connessioni.
- Una `free` senza aver preservato il campo `prossimo` rende **orfani e inaccessibili** tutti i nodi successivi.

? Per rimuovere la testa, possiamo fare subito `free(testa)` ? Cosa perderemmo?



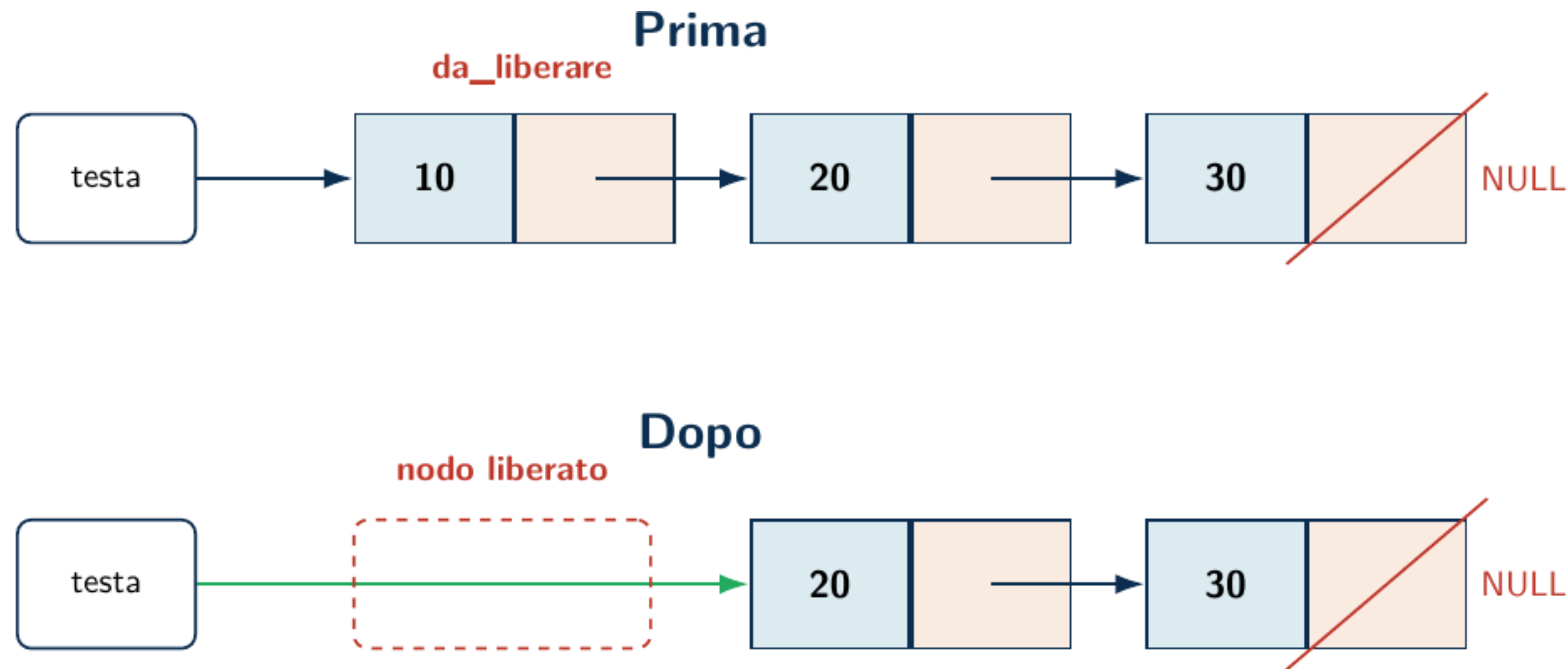
Cancellazione in Testa

```
Nodo* rimuovi_testa(Nodo *testa) {  
    if (testa == NULL) return NULL;  
  
    Nodo *da_liberare = testa; // Salvataggio riferimento  
    testa = testa->prossimo;   // Spostamento testa  
    free(da_liberare);        // Rimozione sicura  
    return testa;  
}
```

Prima: testa → [10 | ●] → [20 | ●] → [30 | NULL]
Dopo: testa → [20 | ●] → [30 | NULL] ([10] liberato)

Visualizzazione: Cancellazione in Testa

- Salviamo il riferimento a `testa` in `da_liberare`, spostiamo `testa` in avanti, e infine liberiamo la memoria.



testa ora punta direttamente al secondo nodo: il primo nodo è stato deallocato.



Cancellazione in Coda

```
Nodo* rimuovi_coda(Nodo *testa) {
    if (testa == NULL) return NULL;

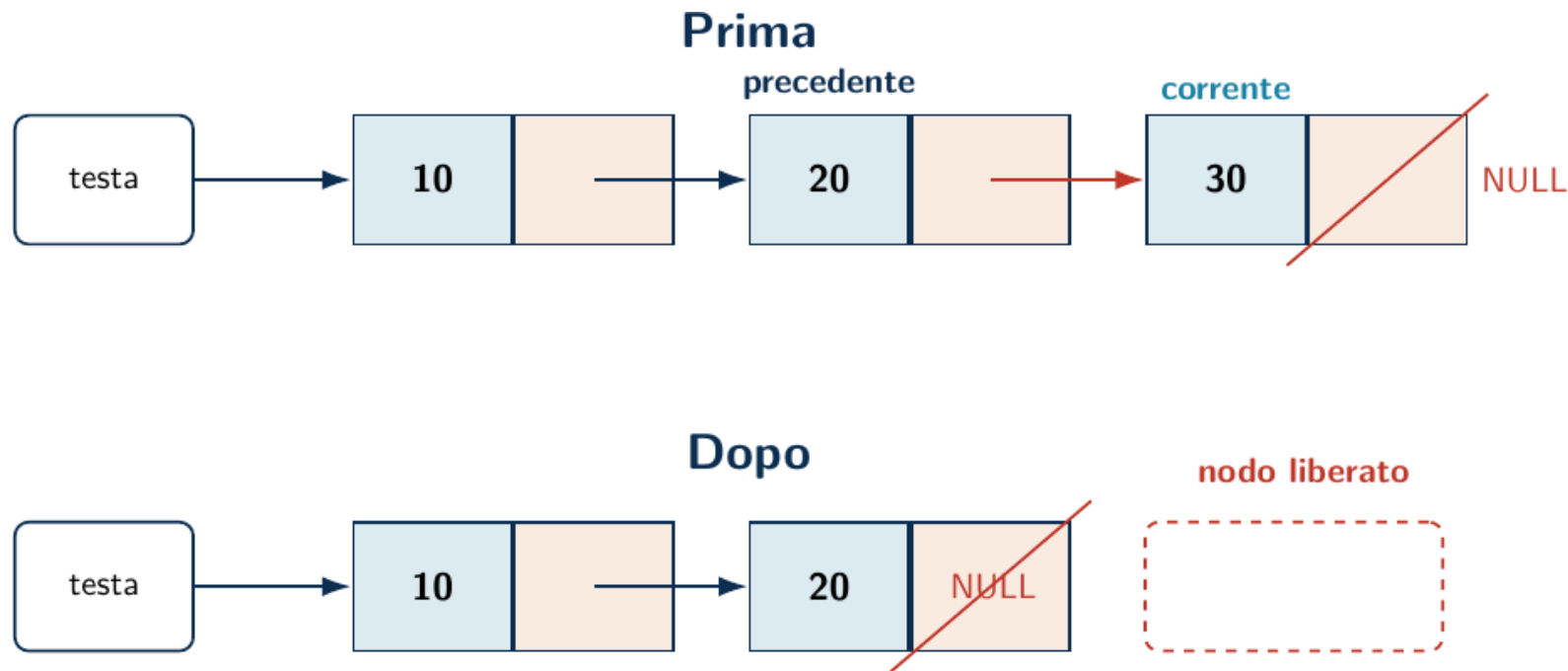
    // Se c'è solo un nodo
    if (testa->prossimo == NULL) {
        free(testa);
        return NULL;
    }

    Nodo *precedente = NULL;
    Nodo *corrente = testa;
    while (corrente->prossimo != NULL) {
        precedente = corrente;
        corrente = corrente->prossimo;
    }
    precedente->prossimo = NULL;
    free(corrente);
    return testa;
}
```

- Bisogna posizionarsi sul **penultimo** nodo: si scorre tutta la lista ($O(N)$).
- A fine ciclo: **corrente** è l'ultimo, **precedente** il penultimo.
- Il penultimo diventa l'ultimo (**prossimo = NULL**), poi si libera l'ultimo.
- Caso speciale: un **solo nodo** ➡ la lista diventa vuota.

Visualizzazione: Cancellazione in Coda

- Una volta scansionata la lista, il puntatore `prossimo` di `precedente` viene reciso e impostato a `NULL`. Infine, si dealloca `corrente`.



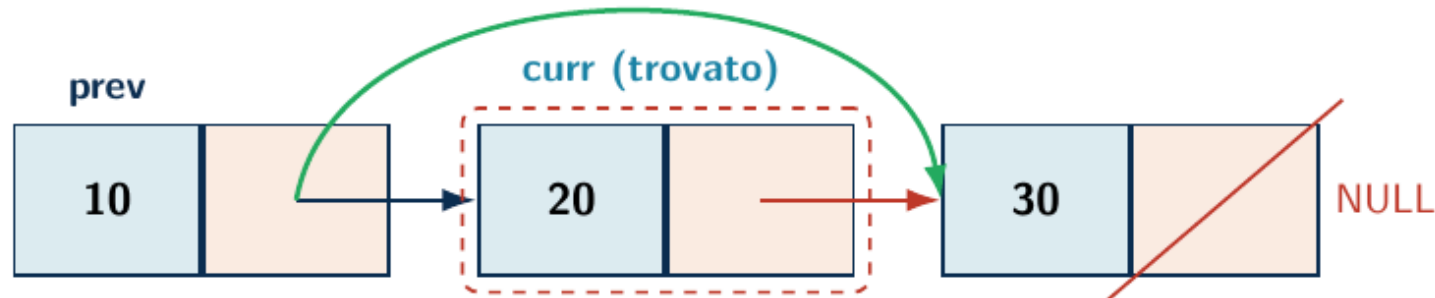
Il puntatore `prossimo` di `precedente` diventa `NULL`: il vecchio ultimo nodo è stato deallocato.

Cancellazione di un Elemento Specifico

```
Nodo* rimuovi_valore(Nodo *testa, int valore) {  
    if (testa == NULL) return NULL;  
    // Rimozione in testa  
    if (testa->dato == valore) {  
        Nodo *nuova_testa = testa->prossimo;  
        free(testa);  
        return nuova_testa;  
    }  
    Nodo *precedente = testa;  
    Nodo *corrente = testa->prossimo;  
    while (corrente != NULL && corrente->dato != valore) {  
        precedente = corrente;  
        corrente = corrente->prossimo;  
    }  
    if (corrente != NULL) {  
        precedente->prossimo = corrente->prossimo; // Detach  
        free(corrente); // Deallocazione  
    }  
    return testa;  
}
```

Visualizzazione: Rimozione di un Valore

- Due puntatori sfalsati; caso speciale se il valore è in testa.
- Prima lo scavalcamento `precedente → prossimo = corrente → prossimo`, poi `free(corrente)`.
- Se `corrente == NULL` il valore non c'è: lista invariata.



Il ciclo di ricerca si ferma su `curr->dato == 20`; poi si esegue il detach come nella scheda 9.



Deallocazione dell'Intera Lista

- I moderni sistemi operativi liberano la memoria del processo alla sua chiusura, ma in ambienti a basso livello (es. microcontrollori) i leak possono persistere.
- **Buona norma:** distruggere ogni lista allocata prima del termine del programma.

```
void svuota_lista(Nodo *testa) {  
    Nodo *corrente = testa;  
    while (corrente != NULL) {  
        Nodo *successore = corrente->prossimo; // Preserva il futuro  
        free(corrente); // Dealloca il presente  
        corrente = successore; // Avanza  
    }  
}
```

- **Attenzione:** salviamo sempre il successore **prima** di distruggere il nodo corrente!



Ordinamento su Liste: Bubble Sort

- Negli array il Bubble Sort scambiava gli elementi. Sulle liste, due opzioni:
 - i. **Opzione A (Swap dei Dati)**: si scambiano solo i valori dei campi `dato`, i nodi restano dove sono nello Heap.
 - ii. **Opzione B (Swap dei Puntatori)**: si sganciano i nodi e si riallineano i puntatori `prossimo`.
- L'Opzione B richiede di aggiornare **fino a 4 indirizzi** per ogni swap: alto rischio di perdere nodi o creare cicli infiniti.
- **Scelta ottimale: Opzione A (Swap dei Dati)**, ricalca il Bubble Sort sugli array.

Implementazione del Bubble Sort su Lista

```
void ordina_lista(Nodo *testa) {  
    // Lista vuota o con un solo elemento: già ordinata  
    if (testa == NULL || testa->prossimo == NULL) {  
        return;  
    }  
  
    bool scambiato;  
    Nodo *corrente;  
    Nodo *limite = NULL; // delimita la fine della scansione  
  
    do {  
        scambiato = false;  
        corrente = testa;  
  
        while (corrente->prossimo != limite) {  
            if (corrente->dato > corrente->prossimo->dato) {  
                // Scambio in-place dei soli dati (Opzione A)  
                int temp = corrente->dato;  
                corrente->dato = corrente->prossimo->dato;  
                corrente->prossimo->dato = temp;  
  
                scambiato = true;  
            }  
            corrente = corrente->prossimo; // Passo in avanti  
        }  
        // L'ultimo elemento esaminato è al suo posto corretto  
        limite = corrente;  
    } while (scambiato); // Nessuno scambio: lista ordinata!  
}
```

⚙ Bubble Sort su Lista: Osservazioni

- `[i]` e `[i+1]` diventano `corrente` e `corrente→prossimo`.
- `limite`: dopo ogni passata l'ultimo nodo esaminato è al suo posto, la scansione successiva si ferma prima.
- Nessuno scambio in una passata ➡ lista ordinata.
- I nodi **non si muovono** nello Heap: cambiano solo i valori dei campi `dato`.

🎬 **Live:** in un `main`, `inserisci_coda` di 30, 10, 20; `stampa_lista`; `ordina_lista`; di nuovo `stampa_lista`; infine `svuota_lista`. Compilare con `gcc -Wall liste.c -o liste`. Atteso:

Testa → [30] → [10] → [20] → NULL

Testa → [10] → [20] → [30] → NULL

✓ Cosa Abbiamo Imparato Oggi

- La **lista concatenata** rinuncia alla contiguità: nodi sparsi nello Heap, collegati da `prossimo`, chiusi da `NULL`.
- Si scorre con un puntatore `corrente`, mai perdendo la **testa**.
- Inserimenti: in testa ($O(1)$), in coda (scansione), ordinato (**due puntatori sfalsati**).
- Cancellazioni: salvare il nodo **prima** di ricollegare, poi `free`.
- `svuota_lista` salva il successore prima di ogni `free`.
- Bubble Sort su lista: **swap dei dati**, non dei puntatori.

 **Per casa:** esercizi della Lezione 15 sul sito del corso.

Grazie per l'attenzione!

Sito Web del Corso: antoninofurnari.github.io/prog1-2627