



Programmazione 1 (Canale O-Z)

Lezione 16: Strutture Dati 2: Pile, Code e Liste Doppie

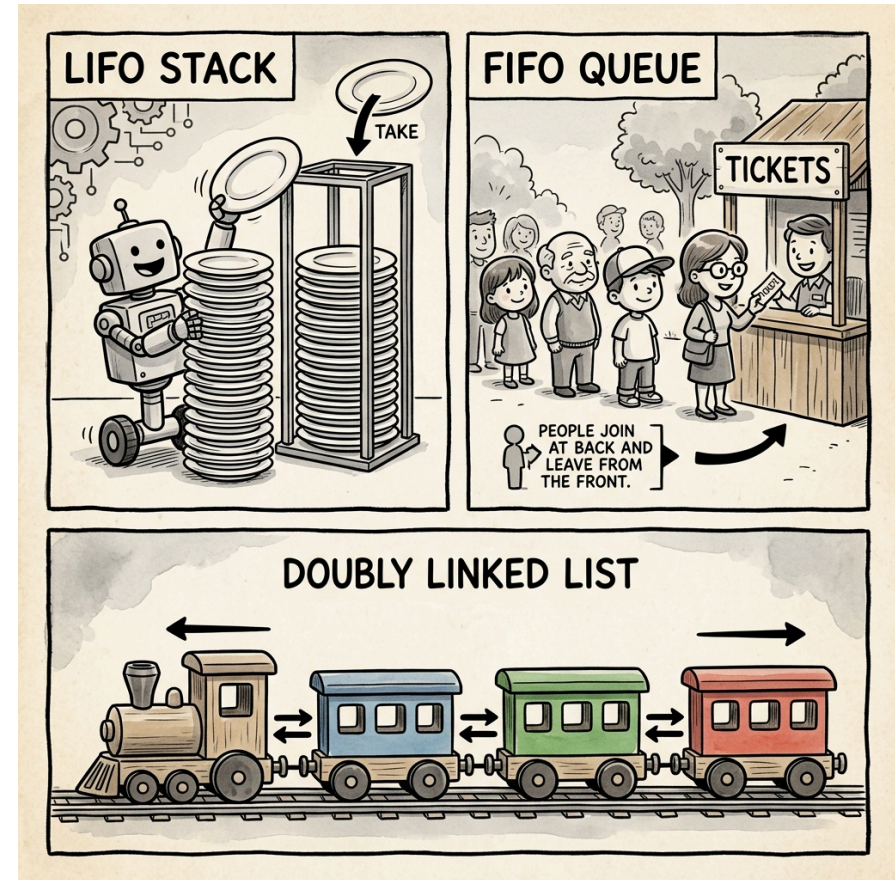
Docente: Prof. Antonino Furnari

Email: antonino.furnari@unict.it

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

Dalla Rappresentazione alle Regole

- Finora: **come** disporre i dati in memoria (array contigui, liste di nodi sparsi nello Heap).
- Oggi separiamo la **rappresentazione fisica** dalle **regole logiche** di accesso:
 - Tipo di Dato Astratto (**ADT**);
 - **Pila** (LIFO) e **Coda** (FIFO);
 - **Liste Doppiaemente Concatenate**.



Tipo di Dato Astratto (ADT)

- Un **ADT** (*Abstract Data Type*) è una descrizione **logica** di una struttura dati che definisce:
 - i. i dati memorizzati;
 - ii. le operazioni consentite su di essi;
 - iii. le regole con cui è permesso accedervi.
- Conta **cosa** fa la struttura, non **come** lo fa fisicamente in memoria.
- L'implementazione (array o lista) è nascosta dietro un'**interfaccia pubblica** (un insieme di funzioni): **incapsulamento** e **astrazione dei dati**.

? Quando usate un file in C, vi interessa in quale settore del disco finiscono i caratteri? O vi bastano `fopen`, `fprintf` e `fclose`?



La Pila (Stack) e la Logica LIFO

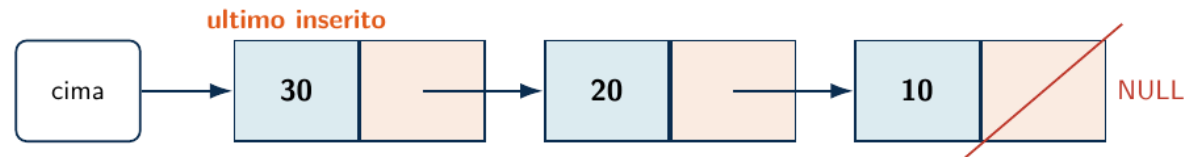
- Governata dalla logica **LIFO** (*Last-In, First-Out*): l'ultimo elemento inserito è sempre il primo ad essere rimosso.
- **Metafora**: una pila di piatti in un dispenser a molla: si appoggia e si preleva **solo dalla cima**.
- **Applicazioni informatiche**:
 - lo Stack dei Record di Attivazione delle funzioni;
 - "Annulla" (*Undo, Ctrl+Z*) nei programmi di videoscrittura;
 - la cronologia del browser (il tasto "Indietro").



Le Operazioni della Pila

- **push**: inserisce un elemento in cima.
- **pop**: estrae l'elemento in cima.
- **peek**: legge l'elemento in cima senza rimuoverlo.

Pila con tre elementi (inseriti in ordine 10, 20, 30)



L'elemento 30, inserito per ultimo, è il primo raggiungibile da cima: sarà anche il primo ad essere rimosso.

? Implementiamo la pila con una lista concatenata: la cima conviene in testa o in coda alla lista?

Pila in C: Definizione e Inizializzazione

- **Cima della pila = testa della lista:** `push` e `pop` in testa, in tempo costante $O(1)$, senza scorrere i nodi.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

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

typedef Nodo* Pila;    // Il tipo astratto Pila

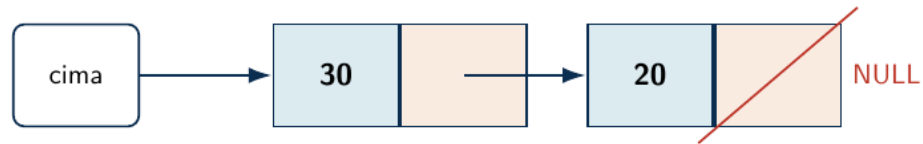
Pila crea_pila(void) {    // Inizializza una pila vuota
    return NULL;
}

bool is_empty_pila(Pila p) {    // Verifica se la pila è vuota
    return p == NULL;
}
```

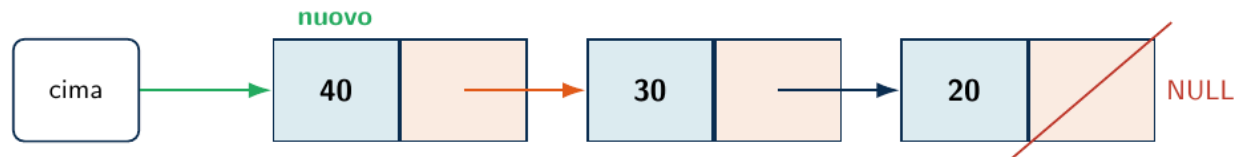


Push e Pop in Figura

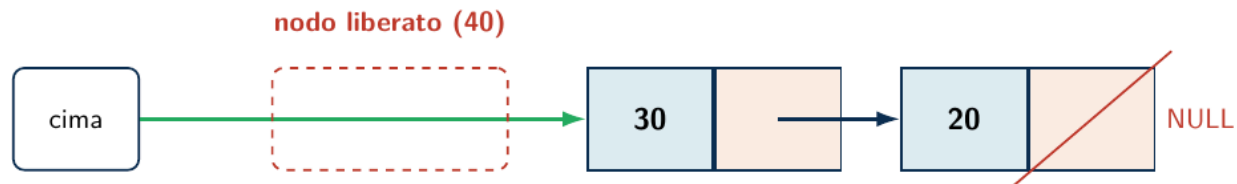
prima



push(40)



pop()





Operazione Push

Inserisce un elemento in cima alla pila (equivale a un **inserimento in testa** alla lista).

```
Pila push(Pila p, int valore) {  
    Nodo *nuovo = (Nodo*) malloc(sizeof(Nodo));  
    if (nuovo == NULL) {  
        fprintf(stderr, "Errore: Memoria esaurita\n");  
        exit(1);  
    }  
    nuovo->dato = valore;  
    nuovo->prossimo = p; // Il nuovo nodo punta alla vecchia cima  
    return nuovo; // Il nuovo nodo diventa la nuova cima della pila  
}
```

- Si usa come `p = push(p, 42);`



Operazione Pop

Estrae la cima, ne restituisce il valore **tramite puntatore** e dealloca il nodo.

```
bool pop(Pila *p, int *valore_estratto) {  
    if (is_empty_pila(*p)) {  
        return false; // Underflow della pila (pila vuota)  
    }  
  
    Nodo *da_eliminare = *p;  
    *valore_estratto = da_eliminare->dato;  
  
    *p = (*p)->prossimo; // Sposta la cima sul nodo successivo  
    free(da_eliminare); // Dealloca la memoria  
    return true;  
}
```

- `Pila *p`: la funzione deve **modificare** la cima del chiamante. Il `bool` segnala l'**underflow**.



Operazione Peek

Ispeziona il valore in cima **senza rimuoverlo**.

```
bool peek(Pila p, int *valore_cima) {  
    if (is_empty_pila(p)) {  
        return false;  
    }  
    *valore_cima = p->dato;  
    return true;  
}
```

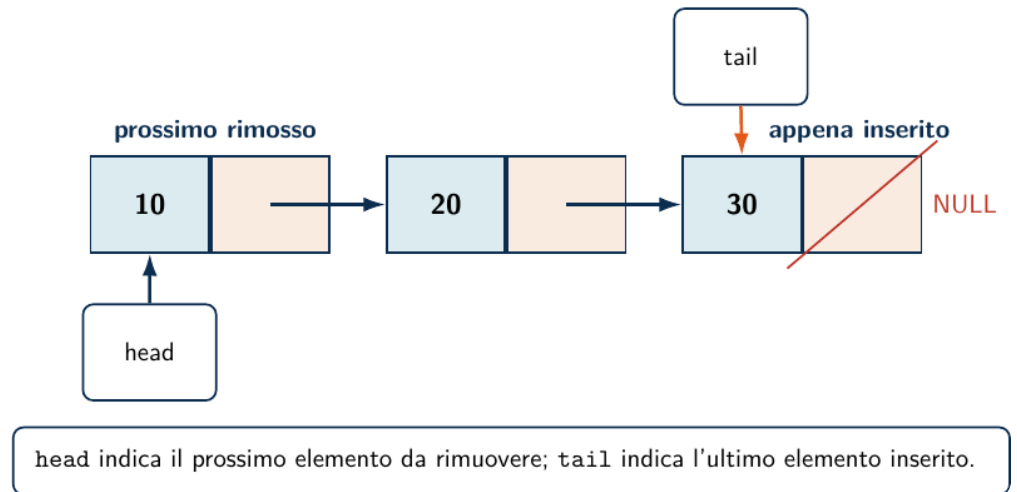
- Qui basta `Pila p` (non `Pila *p`): la cima non cambia.

 **Live:** nel main, `Pila p = crea_pila();` poi `p = push(p, 10); p = push(p, 20); p = push(p, 30);` e `while (pop(&p, &v)) printf("%d ", v);`. Compilare con `gcc -Wall pila.c -o pila` ➡ stampa 30 20 10 (LIFO).

La Coda (Queue) e la Logica FIFO

- Logica **FIFO** (*First-In, First-Out*): il primo inserito è il primo ad essere rimosso.
- Metafora**: la fila alle casse del supermercato.
- Applicazioni**: coda di stampa del sistema operativo, buffer dei pacchetti nei router.
- `enqueue` : aggiunge in fondo; `dequeue` : rimuove dall'inizio.

Coda con tre elementi (inseriti in ordine 10, 20, 30)



? Con una lista semplice e il solo puntatore di testa, dove conviene inserire e dove prelevare?

⚙ Il Dilemma dell'Efficienza della Coda

Lista semplice con il solo puntatore di testa:

- `dequeue` in testa: tempo costante $O(1)$.
- `enqueue` in coda: scorrere **tutta** la lista ad ogni inserimento, $O(N)$.
 - *Con 1 milione di elementi, ogni inserimento percorre 1 milione di nodi.*

La Soluzione Ingegneristica

Una **struttura di controllo** con due puntatori:

- `head`: punta alla testa della lista (per prelevare);
- `tail`: punta all'ultimo nodo (per inserire).

➡ Entrambe le operazioni in $O(1)$.



Coda in C: Struttura e Inizializzazione

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

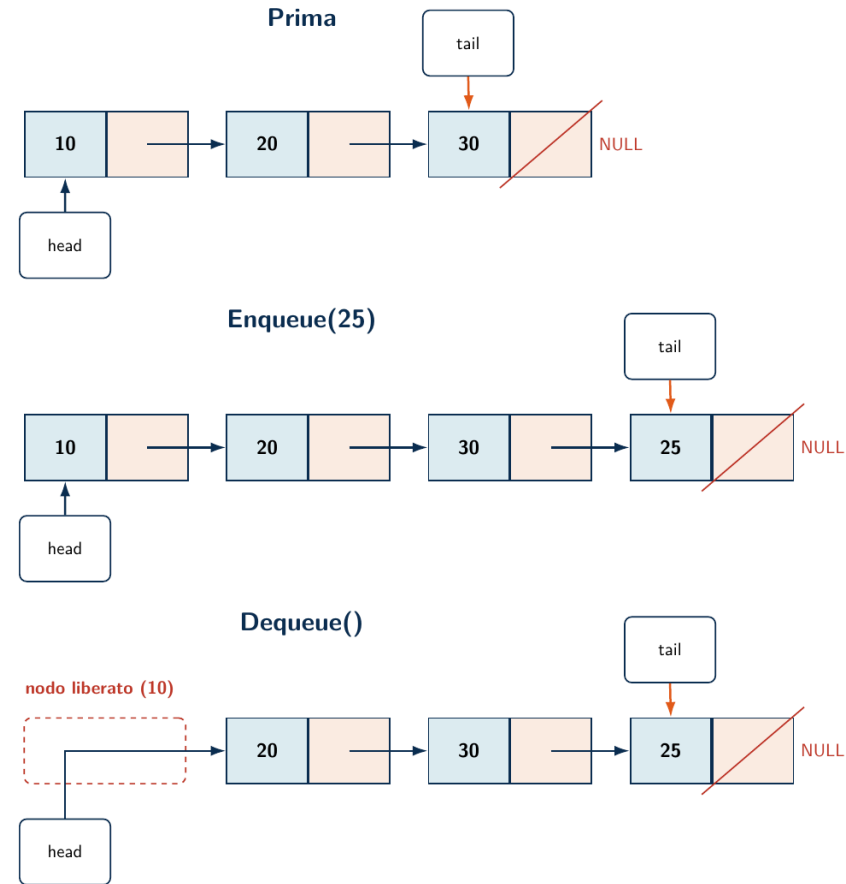
// Struttura di controllo con doppio puntatore
typedef struct {
    NodoCoda *head; // Accesso all'inizio della coda (per dequeue)
    NodoCoda *tail; // Accesso alla fine della coda (per enqueue)
} Coda;

void inizializza_coda(Coda *c) {
    c->head = NULL;
    c->tail = NULL;
}

bool is_empty_coda(const Coda *c) {
    return c->head == NULL;
}
```



Enqueue e Dequeue in Figura





Operazione Enqueue

Aggiunge in fondo in $O(1)$ sfruttando `tail`.

```
void enqueue(Coda *c, int valore) {
    NodoCoda *nuovo = (NodoCoda*) malloc(sizeof(NodoCoda));
    if (nuovo == NULL) {
        fprintf(stderr, "Errore: Memoria esaurita\n");
        exit(1);
    }
    nuovo->dato = valore;
    nuovo->prossimo = NULL;

    if (is_empty_coda(c)) {
        // Se la coda è vuota, il nuovo nodo è sia testa che coda
        c->head = nuovo;
        c->tail = nuovo;
    } else {
        c->tail->prossimo = nuovo; // Il vecchio ultimo nodo viene collegato al nuovo
        c->tail = nuovo;           // Il puntatore di coda si sposta sul nuovo nodo
    }
}
```



Operazione Dequeue

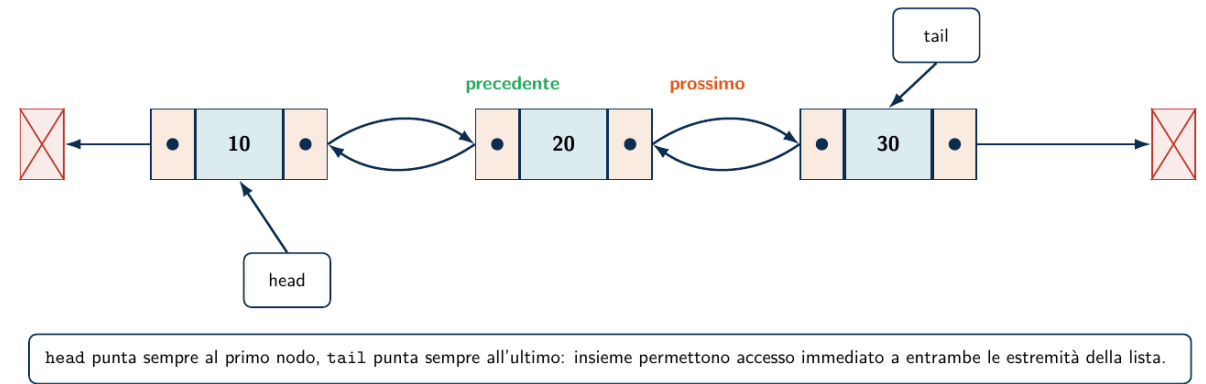
Estrae dalla testa in $O(1)$ sfruttando `head`.

```
bool dequeue(Coda *c, int *valore_estratto) {
    if (is_empty_coda(c)) {
        return false; // Underflow della coda
    }
    NodoCoda *da_eliminare = c->head;
    *valore_estratto = da_eliminare->dato;
    c->head = c->head->prossimo; // La testa avanza al nodo successivo
    // Se la coda si è svuotata, azzeriamo anche il puntatore tail
    if (c->head == NULL) {
        c->tail = NULL;
    }
    free(da_eliminare); // Deallocazione memoria
    return true;
}
```

 **Live:** Coda c; inizializza_coda(&c);, enqueue di 10, 20, 30, poi while (dequeue(&c, &v)) printf("%d ", v); con gcc -Wall coda.c -o coda  10 20 30 (FIFO).

Liste Doppiaemente Concatenate

- Nelle liste semplici c'è solo **prossimo**: impossibile retrocedere o scandire all'indietro.
- Nelle **Liste Doppie** ogni nodo ha **due puntatori**:
 - **prossimo** (in avanti);
 - **precedente** (all'indietro).
- Scansione **bidirezionale** e rimozione **istantanea**.





Liste Doppie in C: Strutture

```
typedef struct NodoDoppio {  
    int dato;  
    struct NodoDoppio *precedente; // Collegamento all'indietro  
    struct NodoDoppio *prossimo;   // Collegamento in avanti  
} NodoDoppio;  
  
// Struttura di controllo per la gestione ottimizzata  
typedef struct {  
    NodoDoppio *head; // Inizio lista  
    NodoDoppio *tail; // Fine lista  
} ListaDoppia;
```

- Come per la coda, `head` e `tail` evitano di scorrere la lista per raggiungere gli estremi.

Liste Doppie in C: Inizializzazione e Nodo

```
// Inizializzazione della lista doppia
void inizializza_lista_doppia(ListaDoppia *l) {
    l→head = NULL;
    l→tail = NULL;
}

// Allocazione del singolo nodo doppio
NodoDoppio* crea_nodo_doppio(int valore) {
    NodoDoppio *nuovo = (NodoDoppio*) malloc(sizeof(NodoDoppio));
    if (nuovo == NULL) {
        fprintf(stderr, "Errore: RAM esaurita!\n");
        exit(1);
    }
    nuovo→dato = valore;
    nuovo→precedente = NULL;
    nuovo→prossimo = NULL;
    return nuovo;
}
```

Inserimenti Bilaterali: in Testa

Con `tail` e `precedente` si inserisce **ad entrambi gli estremi** in $O(1)$, senza scansioni.

```
void inserisci_testa_doppia(ListaDoppia *l, int valore) {
    NodoDoppio *nuovo = crea_nodo_doppio(valore);

    if (l->head == NULL) {
        // Caso lista vuota: il nuovo nodo è sia testa che coda
        l->head = nuovo;
        l->tail = nuovo;
    } else {
        nuovo->prossimo = l->head;           // Il nuovo punta in avanti alla vecchia testa
        l->head->precedente = nuovo;         // La vecchia testa punta all'indietro al nuovo
        l->head = nuovo;                     // Aggiorniamo la testa di controllo
    }
}
```

Inserimenti Bilaterali: in Coda

```
void inserisci_coda_doppia(ListaDoppia *l, int valore) {
    NodoDoppio *nuovo = crea_nodo_doppio(valore);

    if (l->head == NULL) {
        // Caso lista vuota
        l->head = nuovo;
        l->tail = nuovo;
    } else {
        nuovo->precedente = l->tail;    // Il nuovo si allaccia all'indietro al vecchio ultimo
        l->tail->prossimo = nuovo;      // Il vecchio ultimo punta in avanti al nuovo arrivato
        l->tail = nuovo;                // Aggiorniamo la coda di controllo
    }
}
```

- Simmetrico all'inserimento in testa: si scambiano i ruoli di `head` / `tail` e `prossimo` / `precedente`.

? In una lista semplice, avendo il puntatore a un nodo, cosa serve per eliminarlo? E in una lista doppia?

Il Vero Trionfo: la Rimozione Istantanea

- **Lista semplice:** per eliminare un nodo di cui si ha l'indirizzo bisogna **scorrere** la lista fino al predecessore.
- **Lista doppia:** il predecessore è già lì, `da_eliminare→precedente` ➡ nessuna scansione, tempo costante $O(1)$.
- Si **ricuciono** direttamente i due vicini (se esistono) e si aggiornano `head` / `tail` se il nodo era un estremo.
- Infine `free(da_eliminare)`.



Rimozione Istantanea: il Codice

```
void rimuovi_nodo_doppio(ListaDoppia *l, NodoDoppio *da_eliminare) {
    if (l→head == NULL || da_eliminare == NULL) return;
    // Se il nodo da eliminare è la testa
    if (l→head == da_eliminare) {
        l→head = da_eliminare→prossimo;
    }
    // Se il nodo da eliminare è la coda
    if (l→tail == da_eliminare) {
        l→tail = da_eliminare→precedente;
    }
    // Ricuciamo il nodo precedente
    if (da_eliminare→precedente != NULL) {
        da_eliminare→precedente→prossimo = da_eliminare→prossimo;
    }
    // Ricuciamo il nodo successivo
    if (da_eliminare→prossimo != NULL) {
        da_eliminare→prossimo→precedente = da_eliminare→precedente;
    }
    free(da_eliminare);
}
```

Trade-off Ingegneristici delle Liste Doppie

Vantaggi

1. **Navigazione bidirezionale:** in avanti e all'indietro.
2. **Rimozione rapida:** in tempo costante, noto il puntatore al nodo.

Svantaggi

1. **Consumo di memoria:** ogni nodo memorizza **due** puntatori (8 byte ciascuno a 64 bit). Con payload piccoli (es. `char`), i puntatori pesano più dei dati.
2. **Complessità del codice:** ogni inserimento o cancellazione aggiorna fino a **4** puntatori; dimenticarne uno significa bug e puntatori orfani in RAM.

✓ Cosa Abbiamo Imparato Oggi

- Un **ADT** definisce dati, operazioni e regole d'accesso, nascondendo l'implementazione dietro un'interfaccia.
- **Pila** (LIFO): cima = testa della lista, `push` / `pop` / `peek` in $O(1)$.
- **Coda** (FIFO): struttura di controllo con `head` e `tail` ➡ `enqueue` e `dequeue` in $O(1)$.
- **Lista doppia**: `prossimo` + `precedente`, inserimenti ad entrambi gli estremi e rimozione istantanea, al prezzo di più memoria e più puntatori da gestire.

📖 Per casa: esercizi della Lezione 16 sul sito del corso.

Grazie per l'attenzione!

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