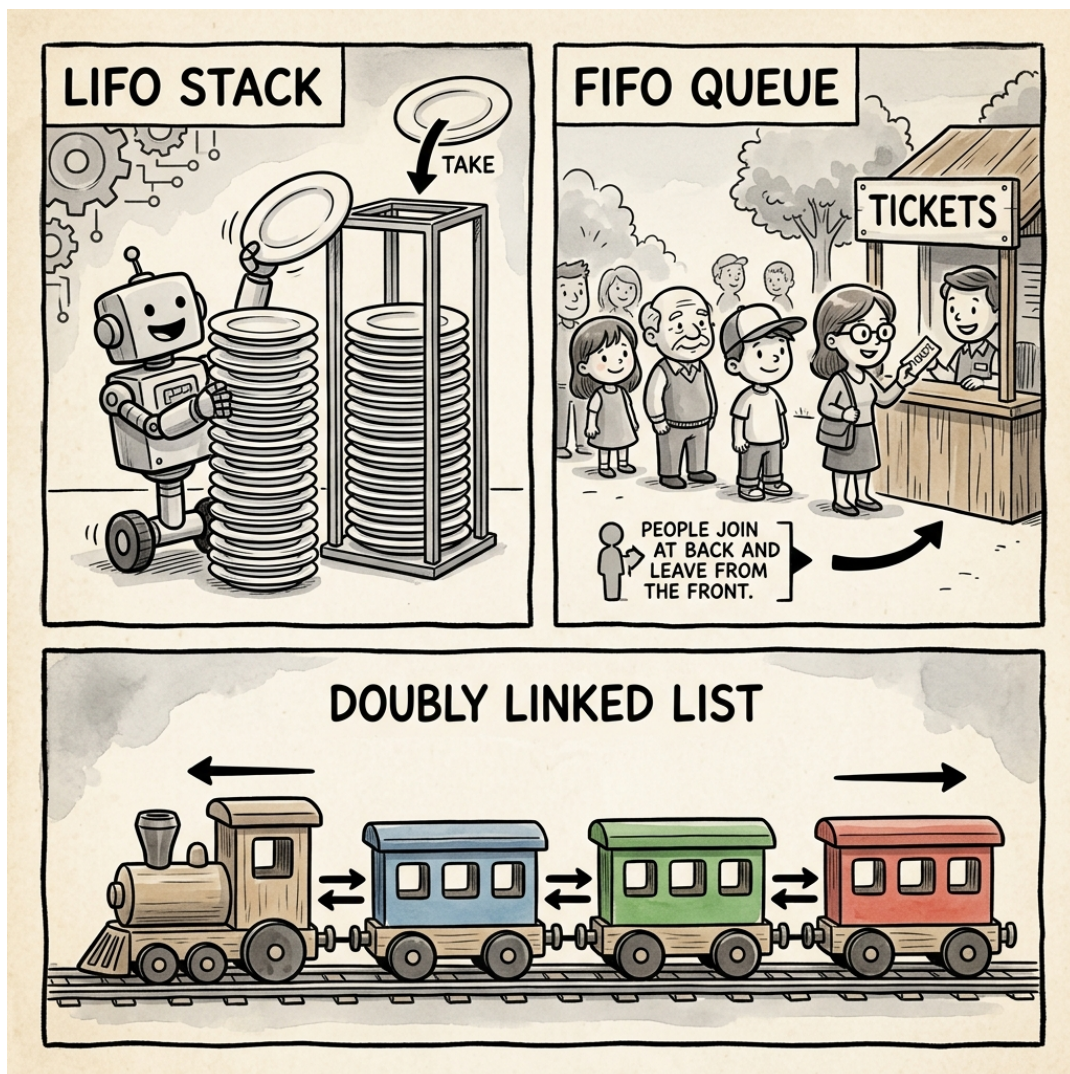


Pile, Code e Liste Doppie



Lezione 1

Pile, Code e Liste Doppie

Nelle lezioni precedenti abbiamo visto come memorizzare dati all'interno di strutture lineari come gli array (allocazione contigua in RAM) e le liste concatenate semplici (nodi sparsi nello Heap collegati in avanti). In questa lezione facciamo un passo avanti: separiamo la rappresentazione fisica dei dati dalle regole logiche con cui vi accediamo, introducendo il concetto di **Tipo di Dato Astratto (ADT)**. Studieremo poi come implementare in C due delle strutture dati più importanti dell'informatica, la **Pila (Stack)** e la **Coda (Queue)**, concludendo con un'introduzione alle **Liste Doppiaemente Concatenate**.

1.1 Il Concetto di Tipo di Dato Astratto (ADT)

Fino ad ora abbiamo manipolato i dati preoccupandoci di come disporli fisicamente in memoria. Ad esempio, per rappresentare una sequenza di interi, abbiamo usato un array (contiguo) o una lista (frammentata).

Un **Tipo di Dato Astratto** (*Abstract Data Type - ADT*) è una descrizione logica di una struttura dati che definisce:

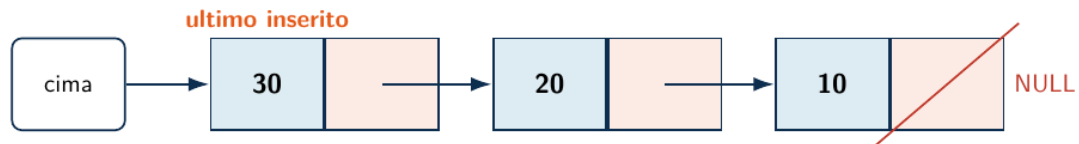
1. I dati memorizzati.
2. Le operazioni consentite su di essi.
3. Le regole con cui è permesso accedere a tali dati.

In un ADT, l'implementazione fisica (array o lista) viene nascosta all'utente esterno dietro un'**interfaccia pubblica** (un insieme di funzioni). Questo principio di programmazione prende il nome di **incapsulamento** ed **astrazione dei dati**.

1.2 La Pila (Stack)

La **Pila** (o *Stack*) è una struttura dati lineare governata dalla logica **LIFO** (*Last-In, First-Out*): l'ultimo elemento inserito è sempre il primo ad essere rimosso.

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.

Figura 1.1: Struttura della Pila

1.2.1 Metafore e Casi d'Uso

- **Metafora reale:** Una pila di piatti in un dispenser a molla: possiamo appoggiare un piatto solo sopra la cima, e possiamo prelevare solo l'ultimo piatto posato in cima.
- **Applicazioni informatiche:**
 - Lo Stack dei Record di Attivazione delle funzioni gestito a livello hardware della CPU.
 - La funzionalità “Annulla” (*Undo* o *Ctrl+Z*) dei programmi di videoscrittura.
 - La cronologia di navigazione dei browser web (il tasto “Indietro” torna all’ultima pagina visitata).

1.2.2 Implementazione di una Pila in C

Per implementare una pila dinamica senza sprechi di memoria, utilizziamo una lista concatenata semplice. Facciamo coincidere la “cima” della pila con la **testa** della lista. In questo modo, sia l’inserimento (*push*) che la rimozione (*pop*) avvengono in testa alla lista in tempo costante $O(1)$, evitando di scorrere i nodi.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4
5 // Definizione del nodo
6 typedef struct Nodo {
7     int dato;
8     struct Nodo *prossimo;
9 } Nodo;
10
11 // Definizione del tipo astratto Pila
12 typedef Nodo* Pila;
13
14 // Inizializza una pila vuota
```

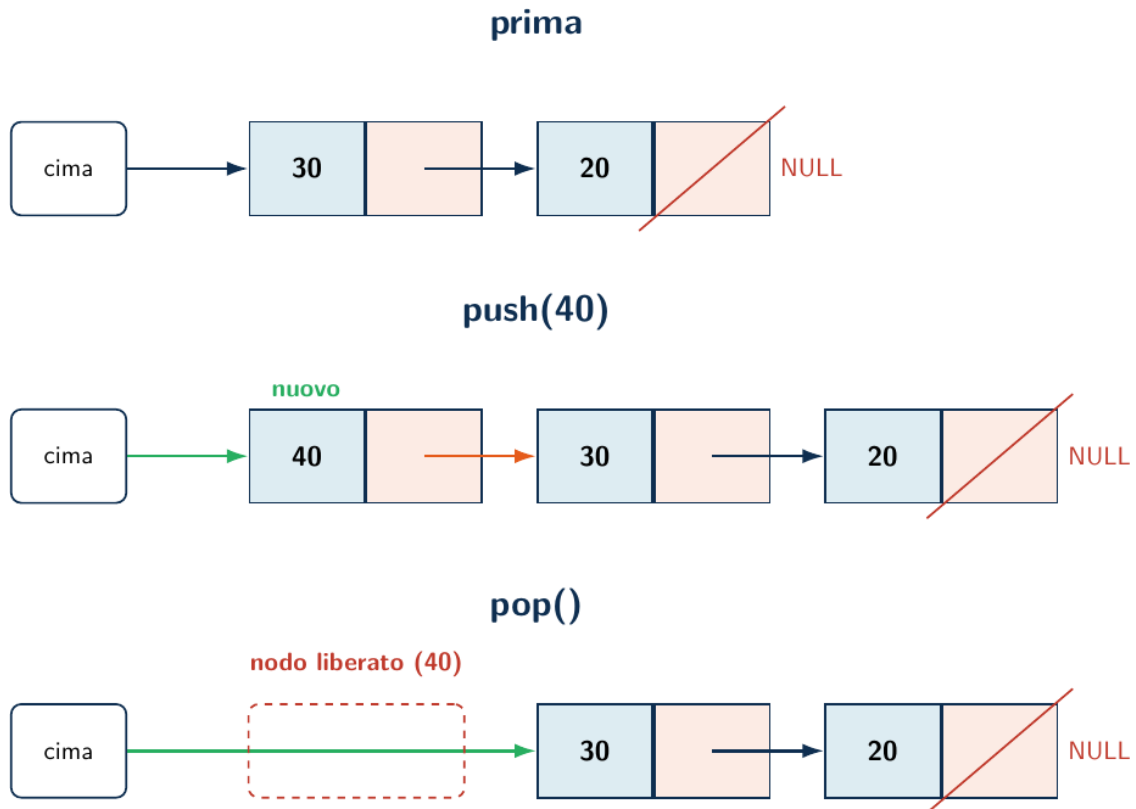


Figura 1.2: Operazioni di Push e Pop

```

15 Pila crea_pila(void) {
16     return NULL;
17 }
18
19 // Verifica se la pila ? vuota
20 bool is_empty_pila(Pila p) {
21     return p == NULL;
22 }

```

Operazioni Push, Pop e Peek

Le operazioni di push e pop permettono di inserire ed eliminare elementi dalla pila, come mostrato di seguito:

L'operazione peek permette di leggere l'elemento in cima alla pila senza eliminarlo dalla struttura.

Operazione Push Inserisce un elemento in cima alla pila (equivalente a un inserimento in testa alla lista).

```

1 Pila push(Pila p, int valore) {
2     Nodo *nuovo = (Nodo*) malloc(sizeof(Nodo));
3     if (nuovo == NULL) {
4         fprintf(stderr, "Errore: Memoria esaurita\n");

```

```
5     exit(1);
6 }
7 nuovo->dato = valore;
8 nuovo->prossimo = p; // Il nuovo nodo punta alla vecchia cima
9 return nuovo;        // Il nuovo nodo diventa la nuova cima della
    ⇨ pila
10 }
```

Operazione Pop Estrae l'elemento in cima alla pila, ne restituisce il valore attraverso un puntatore e dealloca il nodo rimosso.

```
1 bool pop(Pila *p, int *valore_estratto) {
2     if (is_empty_pila(*p)) {
3         return false; // Underflow della pila (pila vuota)
4     }
5
6     Nodo *da_eliminare = *p;
7     *valore_estratto = da_eliminare->dato;
8
9     *p = (*p)->prossimo; // Sposta la cima sul nodo successivo
10    free(da_eliminare); // Dealloca la memoria
11    return true;
12 }
```

Operazione Peek Consente di ispezionare il valore in cima alla pila senza rimuoverlo.

```
1 bool peek(Pila p, int *valore_cima) {
2     if (is_empty_pila(p)) {
3         return false;
4     }
5     *valore_cima = p->dato;
6     return true;
7 }
```

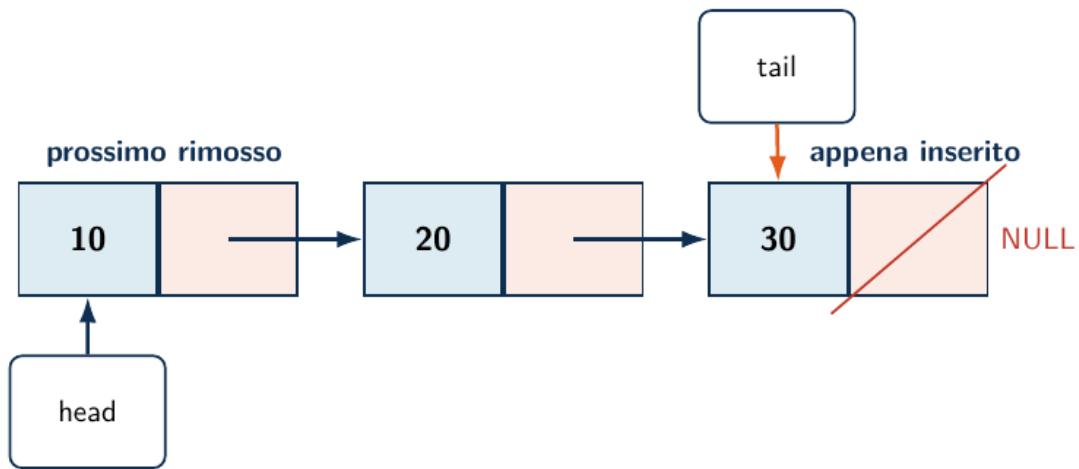
1.3 La Coda (Queue)

La **Coda** (o *Queue*) è una struttura dati lineare governata dalla logica **FIFO** (*First-In, First-Out*): il primo elemento inserito è il primo ad essere rimosso.

1.3.1 Metafore e Casi d'Uso

- **Metafora reale:** La fila ordinata alle casse di un supermercato: chi arriva per primo viene servito per primo, chi si accoda per ultimo deve attendere il proprio turno.
- **Applicazioni informatiche:**
 - La coda di stampa del sistema operativo.
 - La trasmissione dei pacchetti di dati nei buffer di memoria dei router di rete.

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



head indica il prossimo elemento da rimuovere; tail indica l'ultimo elemento inserito.

Figura 1.3: Struttura della Coda

1.3.2 Testa, Coda ed Efficienza

Se implementiamo una coda usando una normale lista semplice memorizzando solo il puntatore di testa:

- Se estraiamo (dequeue) in testa alla lista, l'operazione richiede tempo costante $O(1)$.
- Se inseriamo (enqueue) in coda alla lista, siamo costretti a scorrere l'intera lista fino all'ultimo elemento ad ogni inserimento, con complessità temporale lineare $O(N)$.

Per ovviare a questo problema ed eseguire entrambe le operazioni in tempo costante $O(1)$, ingegnerizziamo una struttura di controllo che mantiene **due puntatori**:

1. Un puntatore head alla testa della lista (per prelevare).
2. Un puntatore tail all'ultimo nodo della lista (per inserire).

1.3.3 Implementazione di una Coda in C

```

1 typedef struct NodeCoda {
2     int dato;
3     struct NodeCoda *prossimo;
4 } NodeCoda;
5
6 // Struttura di controllo con doppio puntatore
7 typedef struct {
8     NodeCoda *head; // Accesso all'inizio della coda (per dequeue)
9     NodeCoda *tail; // Accesso alla fine della coda (per enqueue)

```

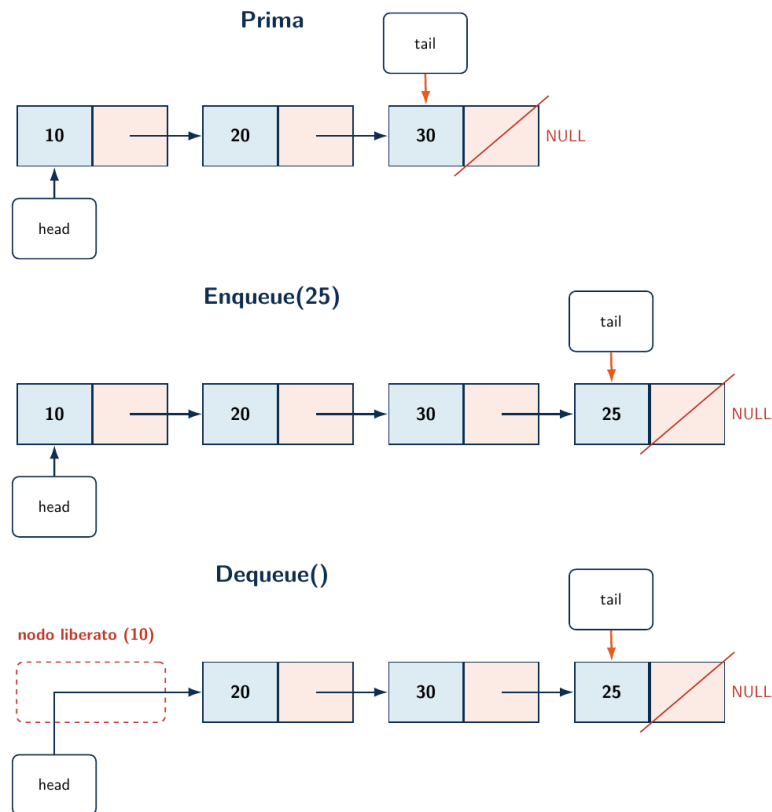


Figura 1.4: Operazioni di Enqueue e Dequeue

```

10 } Coda;
11
12 // Inizializza la coda
13 void inizializza_coda(Coda *c) {
14     c->head = NULL;
15     c->tail = NULL;
16 }
17
18 // Verifica se la coda ? vuota
19 bool is_empty_coda(const Coda *c) {
20     return c->head == NULL;
21 }

```

Operazioni di Enqueue e Dequeue

Le operazioni di enqueue e dequeue permettono di inserire e rimuovere elementi dalla coda, come mostrato di seguito:

Operazione Enqueue Aggiunge un elemento alla fine della coda in tempo $O(1)$ sfruttando il puntatore tail.

```

1 void enqueue(Coda *c, int valore) {
2     NodoCoda *nuovo = (NodoCoda*) malloc(sizeof(NodoCoda));
3     if (nuovo == NULL) {

```

```

4         fprintf(stderr, "Errore: Memoria esaurita\n");
5         exit(1);
6     }
7     nuovo->dato = valore;
8     nuovo->prossimo = NULL;
9
10    if (is_empty_coda(c)) {
11        // Se la coda ? vuota, il nuovo nodo ? sia testa che coda
12        c->head = nuovo;
13        c->tail = nuovo;
14    } else {
15        // Il vecchio ultimo nodo viene collegato al nuovo
16        c->tail->prossimo = nuovo;
17        // Il puntatore di coda si sposta sul nuovo nodo
18        c->tail = nuovo;
19    }
20 }

```

Operazione Dequeue Estrae l'elemento in testa alla coda in tempo $O(1)$ sfruttando il puntatore head.

```

1 bool dequeue(Coda *c, int *valore_estratto) {
2     if (is_empty_coda(c)) {
3         return false; // Underflow della coda
4     }
5
6     NodoCoda *da_eliminare = c->head;
7     *valore_estratto = da_eliminare->dato;
8
9     c->head = c->head->prossimo; // La testa avanza al nodo successivo
10
11    // Se la coda si ? svuotata, azzeriamo anche il puntatore tail
12    if (c->head == NULL) {
13        c->tail = NULL;
14    }
15
16    free(da_eliminare); // Deallocazione memoria
17    return true;
18 }

```

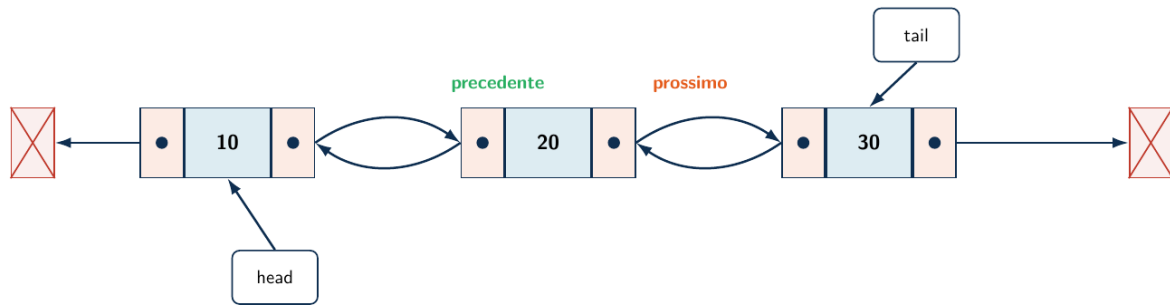
1.4 Liste Doppiaemente Concatenate

Nelle liste semplici, ogni nodo contiene solo un puntatore in avanti (`prossimo`). Questo rende impossibile retrocedere o effettuare scansioni all'indietro.

1.4.1 La Struttura dei Nodi Doppi

Per navigare in entrambe le direzioni, ogni nodo ospita **due puntatori**:

- Un puntatore `prossimo` che punta in avanti.



head punta sempre al primo nodo, tail punta sempre all'ultimo: insieme permettono accesso immediato a entrambe le estremità della lista.

Figura 1.5: Lista Doppiaemente Concatenata

- Un puntatore precedente che punta all'indietro.

```

1  typedef struct NodoDoppio {
2      int dato;
3      struct NodoDoppio *precedente; // Collegamento all'indietro
4      struct NodoDoppio *prossimo;   // Collegamento in avanti
5  } NodoDoppio;
6
7  // Struttura di controllo per la gestione ottimizzata
8  typedef struct {
9      NodoDoppio *head; // Inizio lista
10     NodoDoppio *tail; // Fine lista
11 } ListaDoppia;

```

```

1  // Inizializzazione della lista doppia
2  void inizializza_lista_doppia(ListaDoppia *l) {
3      l->head = NULL;
4      l->tail = NULL;
5  }
6
7  // Allocazione del singolo nodo doppio
8  NodoDoppio* crea_nodo_doppio(int valore) {
9      NodoDoppio *nuovo = (NodoDoppio*) malloc(sizeof(NodoDoppio));
10     if (nuovo == NULL) {
11         fprintf(stderr, "Errore: RAM esaurita!\n");
12         exit(1);
13     }
14     nuovo->dato = valore;
15     nuovo->precedente = NULL;
16     nuovo->prossimo = NULL;
17     return nuovo;
18 }

```

1.4.2 Inserimenti Bilaterali

Grazie alla presenza del puntatore `tail` e del collegamento precedente, possiamo inserire elementi sia in testa che in coda alla lista in tempo costante $O(1)$ senza mai dover effettuare scansioni lineari dei nodi.

Inserimento in Testa

```

1 void inserisci_testa_doppia(ListaDoppia *l, int valore) {
2     NodoDoppio *nuovo = crea_nodo_doppio(valore);
3
4     if (l->head == NULL) {
5         // Caso lista vuota: il nuovo nodo ? sia testa che coda
6         l->head = nuovo;
7         l->tail = nuovo;
8     } else {
9         nuovo->prossimo = l->head;    // Il nuovo punta in avanti alla
        ↪ vecchia testa
10        l->head->precedente = nuovo;  // La vecchia testa punta all'
        ↪ indietro al nuovo
11        l->head = nuovo;              // Aggiorniamo la testa di
        ↪ controllo
12    }
13 }
```

Inserimento in Coda

```

1 void inserisci_coda_doppia(ListaDoppia *l, int valore) {
2     NodoDoppio *nuovo = crea_nodo_doppio(valore);
3
4     if (l->head == NULL) {
5         // Caso lista vuota
6         l->head = nuovo;
7         l->tail = nuovo;
8     } else {
9         nuovo->precedente = l->tail; // Il nuovo si allaccia all'
        ↪ indietro al vecchio ultimo
10        l->tail->prossimo = nuovo;    // Il vecchio ultimo punta in
        ↪ avanti al nuovo arrivato
11        l->tail = nuovo;              // Aggiorniamo la coda di
        ↪ controllo
12    }
13 }
```

1.4.3 Rimozione Istantanea

Il grande vantaggio delle liste doppie risiede nell'eliminazione di un nodo. Se si possiede già il puntatore esatto al nodo da eliminare (da `{}_eliminare`), non occorre alcuna scansione della lista per trovare il predecessore. L'operazione avviene in tempo costante $O(1)$ ricollegando direttamente i puntatori dei nodi adiacenti.

```
1 void rimuovi_nodo_doppio(ListaDoppia *l, NodoDoppio *da_eliminare) {
2     if (l->head == NULL || da_eliminare == NULL) return;
3
4     // Se il nodo da eliminare ? la testa
5     if (l->head == da_eliminare) {
6         l->head = da_eliminare->prossimo;
7     }
8
9     // Se il nodo da eliminare ? la coda
10    if (l->tail == da_eliminare) {
11        l->tail = da_eliminare->precedente;
12    }
13
14    // Ricuciamo il nodo precedente
15    if (da_eliminare->precedente != NULL) {
16        da_eliminare->precedente->prossimo = da_eliminare->prossimo;
17    }
18
19    // Ricuciamo il nodo successivo
20    if (da_eliminare->prossimo != NULL) {
21        da_eliminare->prossimo->precedente = da_eliminare->precedente;
22    }
23
24    free(da_eliminare);
25 }
```

1.4.4 I Trade-off Ingegneristici

1. **Consumo di memoria:** Ogni nodo deve memorizzare due puntatori (8 byte ciascuno su sistemi a 64 bit), aumentando l'overhead strutturale.
2. **Complessità del codice:** Ogni inserimento o cancellazione richiede l'aggiornamento simultaneo di un massimo di 4 puntatori, aumentando il rischio di bug e puntatori orfani in RAM.

1.5 Quiz di Autovalutazione

Quiz 1

Domanda

1. Che cos'è un Tipo di Dato Astratto (ADT)?

- A) Una struttura dati non ancora implementata dal compilatore.
- B) Un modello logico definito dalle operazioni consentite e dalle regole d'accesso, indipendente dall'implementazione fisica.
- C) Un tipo di puntatore generico `void*` che può indirizzare qualsiasi area dello Heap.
- D) Una macro definita con `\{\}\#define` per memorizzare variabili intere.

Risposta corretta: B
Spiegazione: Gli ADT definiscono il comportamento logico (l'interfaccia pubblica), nascondendo i dettagli implementativi fisici (l'incapsulamento sotto il cofano).

Soluzione

Quiz 2

Domanda

2. Quale logica regola l'accesso agli elementi in una Pila (Stack)?

- A) FIFO (First-In, First-Out)
- B) LIFO (Last-In, First-Out)
- C) LILO (Last-In, Last-Out)
- D) Accesso casuale tramite indice numerico

Risposta corretta: B
Spiegazione: In uno Stack, l'ultimo elemento inserito (cima) è il primo a essere rimosso, proprio come in una pila fisica di piatti.

Soluzione

Quiz 3

Domanda

3. Nell'implementazione di una Pila tramite lista concatenata semplice, perché facciamo coincidere la cima con la testa della lista?

- A) Per risparmiare memoria escludendo il puntatore NULL.
- B) Per effettuare l'inserimento (push) e la rimozione (pop) in tempo costante $O(1)$ senza scorrere la lista.
- C) Perché la testa della lista risiede nello Stack frame invece che nello Heap.
- D) Per ordinare automaticamente gli elementi inseriti.

Risposta corretta: B
Spiegazione: Operare in testa a una lista semplice richiede solo la modifica locale dei puntatori senza dover scorrere i nodi intermedi.

Soluzione

Quiz 4

Domanda

4. Cosa si intende per "Underflow" di una Pila?

- A) L'esaurimento della memoria fisica Heap a seguito di una `malloc` fallita.
- B) Il tentativo di rimuovere un elemento (`pop`) da una pila che è attualmente vuota.
- C) La perdita del puntatore alla testa della lista.
- D) L'accumulo di record di attivazione causato da ricorsione infinita.

Soluzione
Risposta corretta: B
Spiegazione: L'underflow si verifica quando un'operazione di rimozione fallisce perché la struttura dati è priva di elementi.

Quiz 5

Domanda

5. Quale logica regola l'accesso agli elementi in una Coda (Queue)?

- A) FIFO (First-In, First-Out)
- B) LIFO (Last-In, First-Out)
- C) Accesso casuale indicizzato
- D) Ordinamento alfabetico automatico

Soluzione
Risposta corretta: A
Spiegazione: In una Coda, il primo elemento inserito è il primo a uscire, rispecchiando il comportamento di una fila ordinata di persone.

Quiz 6

Domanda

6. Perché una semplice lista concatenata con il solo puntatore head non è efficiente per implementare una Coda?

- A) Perché l'estrazione richiede di scorrere tutti gli elementi.
- B) Perché l'inserimento in coda alla lista richiede uno scorrimento lineare $O(N)$ di tutti i nodi.
- C) Perché non permette la memorizzazione di dati di tipo intero.
- D) Perché causa costanti errori di segmentation fault a runtime.

Soluzione**Risposta corretta: B**

Spiegazione: Senza un riferimento diretto all'ultimo nodo, per inserire in coda occorre scorrere l'intera lista partendo dalla testa, un'operazione costosa all'aumentare dei nodi.

Quiz 7**Domanda**

7. In una Coda ottimizzata con puntatori head e tail, qual è la complessità dell'operazione di accodamento (enqueue)?

- A) Tempo lineare $O(N)$
- B) Tempo costante $O(1)$
- C) Tempo logaritmico $O(\log N)$
- D) Tempo quadratico $O(N^2)$

Soluzione**Risposta corretta: B**

Spiegazione: Sfruttando il puntatore tail che referencia direttamente l'ultimo nodo, l'aggiungimento del nuovo elemento avviene all'istante senza dover effettuare alcuna scansione.

Quiz 8**Domanda**

8. Che cos'è una Lista Doppia Concatenata?

- A) Una lista che può contenere solo coppie di valori uguali.
- B) Una lista formata da nodi contenenti due puntatori, uno al nodo successivo e uno al nodo precedente.
- C) Una lista memorizzata contemporaneamente in due settori diversi della RAM.
- D) Una lista che supporta due puntatori di testa separati.

Soluzione**Risposta corretta: B**

Spiegazione: La bidirezionalità è consentita dalla presenza del puntatore precedente in ogni nodo, permettendo la scansione a ritroso.

Quiz 9

Domanda

9. Nelle liste doppie, qual è la complessità temporale per eliminare un nodo se possediamo già il suo puntatore esatto?

- A) Tempo lineare $O(N)$
- B) Tempo costante $O(1)$
- C) Tempo quadratico $O(N^2)$
- D) L'operazione non è permessa a runtime

Soluzione
Risposta corretta: B
Spiegazione: Poiché il nodo contiene già gli indirizzi sia del successore che del predecessore, non serve effettuare alcuna scansione della lista per ricucire i collegamenti.

Quiz 10

Domanda

10. Qual è il principale svantaggio delle liste doppie rispetto alle liste semplici?

- A) L'impossibilità di allocarle nello Heap.
- B) Un consumo di memoria maggiore per ogni nodo e una maggiore complessità nella gestione del codice (chirurgia dei puntatori).
- C) La mancanza di supporto per l'allocazione dinamica tramite `malloc`.
- D) La limitazione a contenere unicamente payload di tipo float.

Soluzione
Risposta corretta: B
Spiegazione: I due puntatori per nodo raddoppiano lo spazio di RAM dedicato ai collegamenti, e le operazioni sui puntatori richiedono molta attenzione per non perdere i legami bilaterali.

1.6 Esercizi Pratici Risolti

Esercizio 1: Controllo Pila Vuota

Domanda

Scrivere una funzione in C che verifichi se una pila è vuota.

```

1  #include <stdio.h>
2  #include <stdbool.h>
3
4  typedef struct Nodo {
5      int dato;
6      struct Nodo *prossimo;
7  } Nodo;
8
9  typedef struct Pila;
10
11 bool is_empty_pila(Pila p) {
12     // La pila è vuota se il puntatore di testa è NULL
13     return (p == NULL);
14 }

```

Soluzione**Esercizio 2: Conteggio Nodi in Pila****Domanda**

Scrivere una funzione che conti gli elementi all'interno di una pila senza distruggerla (senza svuotarla).

```

1  #include <stdio.h>
2
3  typedef struct Nodo {
4      int dato;
5      struct Nodo *prossimo;
6  } Nodo;
7
8  typedef struct Pila;
9
10 int conta_elementi_pila(Pila p) {
11     int contatore = 0;
12     Nodo *corrente = p; // Usiamo un puntatore temporaneo per non
13     // modificare la testa originale
14     while (corrente != NULL) {
15         contatore++;
16         corrente = corrente->prossimo;
17     }
18     return contatore;
19 }

```

Soluzione

Esercizio 3: Ispezione in Cima (Peek)**Domanda**

Scrivere una funzione peek che legga il valore in cima alla pila restituendo true se l'operazione ha successo, e false se la pila è vuota.

```
1 #include <stdio.h>
2 #include <stdbool.h>
3
4 typedef struct Nodo {
5     int dato;
6     struct Nodo *prossimo;
7 } Nodo;
8
9 typedef Nodo* Pila;
10
11 bool peek(Pila p, int *valore) {
12     if (p == NULL) {
13         return false; // Pila vuota
14     }
15     *valore = p->dato; // Assegna il valore in cima
16     return true;
17 }
```

Soluzione**Esercizio 4: Inizializzazione Coda****Domanda**

Implementare la struttura dati di controllo e la funzione per inizializzare una coda dinamica a vuota.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct NodeCoda {
5      int dato;
6      struct NodeCoda *prossimo;
7  } NodeCoda;
8
9  typedef struct {
10     NodeCoda *head;
11     NodeCoda *tail;
12 } Coda;
13
14 void inizializza_coda(Coda *c) {
15     c->head = NULL;
16     c->tail = NULL;
17 }

```

Soluzione

Esercizio 5: Stampa Coda

Domanda

Scrivere una funzione che stampi a schermo tutti gli elementi presenti in una coda, dall'inizio alla fine.

```

1  #include <stdio.h>
2
3  typedef struct NodeCoda {
4      int dato;
5      struct NodeCoda *prossimo;
6  } NodeCoda;
7
8  typedef struct {
9      NodeCoda *head;
10     NodeCoda *tail;
11 } Coda;
12
13 void stampa_coda(const Coda *c) {
14     NodeCoda *corrente = c->head;
15     printf("Inizio Coda -> ");
16     while (corrente != NULL) {
17         printf("[%d] -> ", corrente->dato);
18         corrente = corrente->prossimo;
19     }
20     printf("Fine Coda (NULL)\n");
21 }

```

Soluzione

Esercizio 6: Dimensione Coda**Domanda**

Scrivere una funzione che determini la dimensione corrente (numero di elementi) di una coda.

```
1  #include <stdio.h>
2
3  typedef struct NodeCoda {
4      int dato;
5      struct NodeCoda *prossimo;
6  } NodeCoda;
7
8  typedef struct {
9      NodeCoda *head;
10     NodeCoda *tail;
11 } Coda;
12
13 int dimensione_coda(const Coda *c) {
14     int count = 0;
15     NodeCoda *corrente = c->head;
16     while (corrente != NULL) {
17         count++;
18         corrente = corrente->prossimo;
19     }
20     return count;
21 }
```

Soluzione**Esercizio 7: Ricerca in Lista Doppia****Domanda**

Scrivere una funzione che ricerchi un intero in una lista doppia e restituisca l'indirizzo del nodo se trovato, o NULL altrimenti.

```

1  #include <stdio.h>
2
3  typedef struct Nododoppio {
4      int dato;
5      struct Nododoppio *precedente;
6      struct Nododoppio *prossimo;
7  } Nododoppio;
8
9  typedef struct {
10     Nododoppio *head;
11     Nododoppio *tail;
12 } Listadoppia;
13
14 Nododoppio *cerca_lista_doppia(const Listadoppia *l, int target) {
15     Nododoppio *corrente = l->head;
16     while (corrente != NULL) {
17         if (corrente->dato == target) {
18             return corrente; // Trovato
19         }
20         corrente = corrente->prossimo;
21     }
22     return NULL; // Non trovato
23 }

```

Soluzione**Esercizio 8: Inserimento in Testa Lista Doppia****Domanda**

Scrivere la funzione per inserire un elemento in testa a una lista doppiamente concatenata.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Nododoppio {
5      int dato;
6      struct Nododoppio *precedente;
7      struct Nododoppio *prossimo;
8  } Nododoppio;
9
10 typedef struct {
11     Nododoppio *head;
12     Nododoppio *tail;
13 } Listadoppia;
14
15 void inserisci_testa_doppia(Listadoppia *l, int valore) {
16     Nododoppio *nuovo = (Nododoppio*) malloc(sizeof(Nododoppio));
17     if (nuovo == NULL) exit(1);
18     nuovo->dato = valore;
19     nuovo->precedente = NULL;
20     nuovo->prossimo = l->head;
21
22     if (l->head == NULL) {
23         // La lista era vuota
24         l->head = nuovo;
25         l->tail = nuovo;
26     } else {
27         l->tail->prossimo = nuovo;
28         l->head->precedente = nuovo;
29     }
30 }
31

```

Soluzione**Esercizio 9: Inserimento in Coda Lista Doppia****Domanda**

Scrivere la funzione per inserire un elemento in coda a una lista doppiamente concatenata.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Nododoppio {
5      int dato;
6      struct Nododoppio *precedente;
7      struct Nododoppio *prossimo;
8  } Nododoppio;
9
10 typedef struct {
11     Nododoppio *head;
12     Nododoppio *tail;
13 } Listadoppia;
14
15 void inserisci_coda_doppia(Listadoppia *l, int valore) {
16     Nododoppio *nuovo = (Nododoppio*) malloc(sizeof(Nododoppio));
17     if (nuovo == NULL) exit(1);
18     nuovo->dato = valore;
19     nuovo->prossimo = NULL;
20     nuovo->precedente = l->tail;
21
22     if (l->tail == NULL) {
23         // La lista era vuota
24         l->head = nuovo;
25         l->tail = nuovo;
26     } else {
27         l->tail->prossimo = nuovo;
28         l->tail = nuovo;
29     }
30 }
31

```

Soluzione**Esercizio 10: Svuota Pila (Deallocazione)****Domanda**

Scrivere una funzione che svuoti completamente una pila deallocando tutti i nodi occupati nello Heap.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Nodo {
5      int dato;
6      struct Nodo *prossimo;
7  } Nodo;
8
9  typedef struct Pila {
10     void svuota_pila(Pila *p) {
11         while (corrente != NULL) {
12             Nodo *corrente = *p;
13             while (corrente != NULL) {
14                 Nodo *successore = corrente->prossimo; // Salva il
15                 // successore
16                 free(corrente);
17                 corrente = successore;
18                 // Sposta corrente in
19                 // avanti
20             }
21             *p = NULL; // Imposta la testa a NULL
22         }
23     }
24 }

```

Soluzione