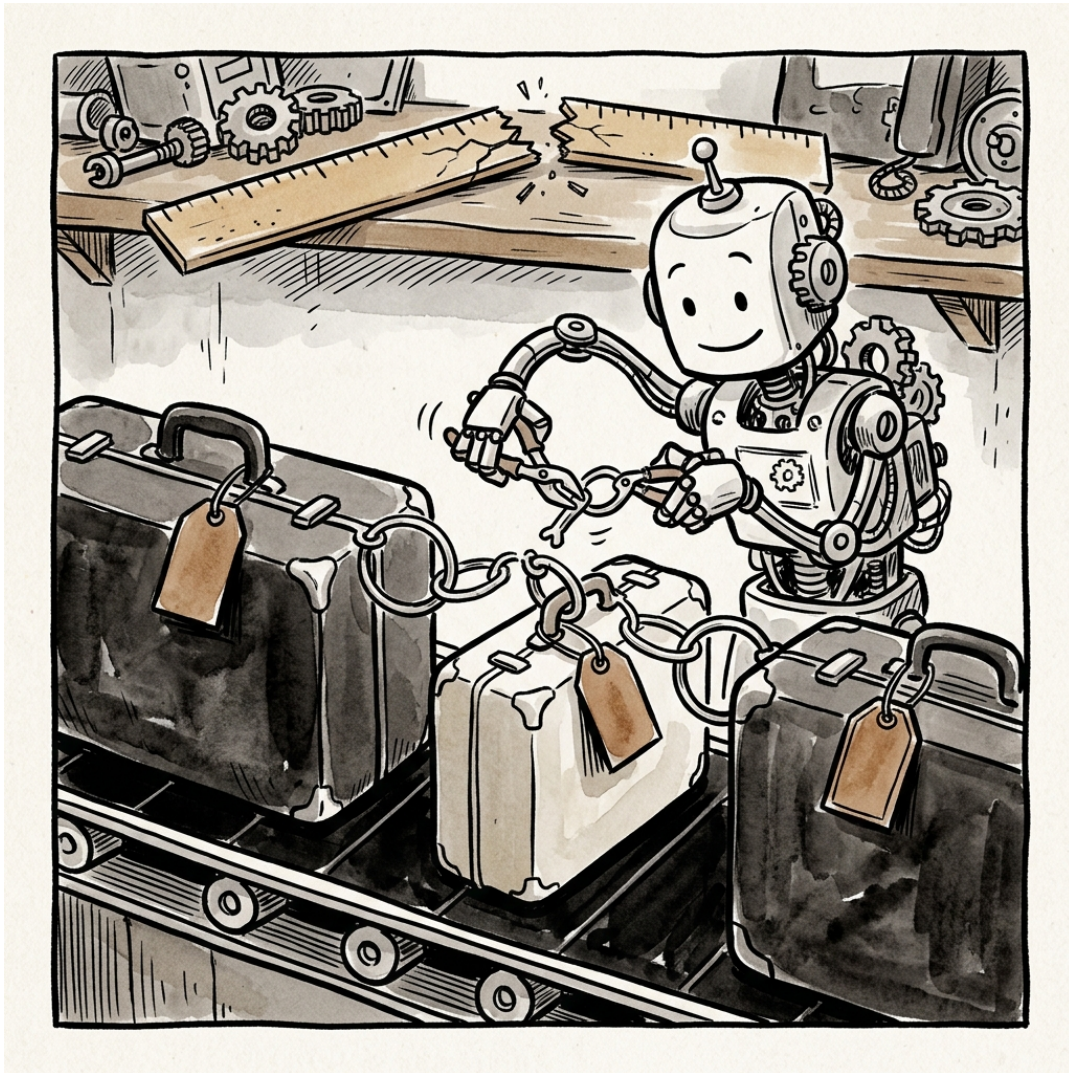


Liste Concatenate Semplici



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 1

Liste Concatenate Semplici

Fino ad oggi abbiamo memorizzato collezioni di elementi contigui in memoria fisica (RAM) attraverso gli array statici e dinamici. Sebbene gli array garantiscano tempi di accesso immediati in tempo costante grazie al calcolo dell'offset dall'indice di partenza, essi presentano rigidi limiti strutturali:

1. **Costo di ridimensionamento:** Se lo spazio allocato nello Heap tramite `malloc` o `realloc` si esaurisce, l'ingrandimento dell'array può comportare la ricerca di una nuova area di memoria contigua sufficientemente capiente, costringendo la CPU a copiare interamente i dati esistenti ($O(N)$).
2. **Costo di inserimento/eliminazione in posizione intermedia:** Inserire o rimuovere un valore all'indice 0 richiede lo scorrimento (*shifting*) di tutti gli elementi successivi verso destra o verso sinistra, con una complessità pari a $O(N)$.

La **Lista Concatenata Semplice** (Singly Linked List) supera questi limiti rinunciando alla contiguità fisica degli elementi.

1.1 La Struttura Logica della Lista

In una lista concatenata, gli elementi (chiamati **Nodi**) sono distribuiti liberamente all'interno dello Heap. Essi sono logicamente collegati tramite indirizzi di memoria (puntatori).

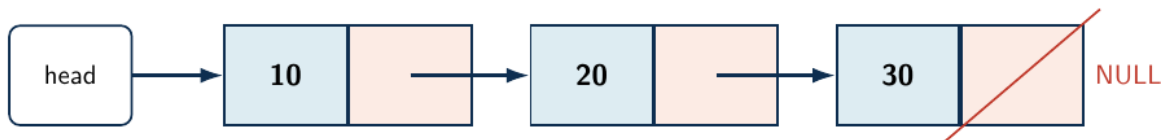
1.1.1 L'Anatomia di un Nodo

Ogni nodo è un record (`struct`) costituito da due campi fondamentali:

1. **Payload (Dato):** Le informazioni effettivamente memorizzate (ad esempio, un intero `int`, un carattere o una struttura nidificata).
2. **Puntatore al Successore (next):** Un puntatore contenente l'indirizzo di memoria del nodo successivo nella sequenza.

Per gestire una lista concatenata, il programmatore deve mantenere una variabile locale nello Stack chiamata **Puntatore di Testa (head)**, la quale contiene l'indirizzo di memoria del primo nodo.

Lista con tre nodi



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

Figura 1.1: Struttura di una lista concatenata

1.2 Rappresentazione in C ed Ispezione

Definiamo il nodo in C mediante una struttura auto-referenziale (una struct che contiene un puntatore a un'altra istanza della stessa struct):

```
1 typedef struct Nodo {  
2     int dato;  
3     struct Nodo *prossimo;  
4 } Nodo;
```

1.2.1 Funzione di Allocazione Sicura

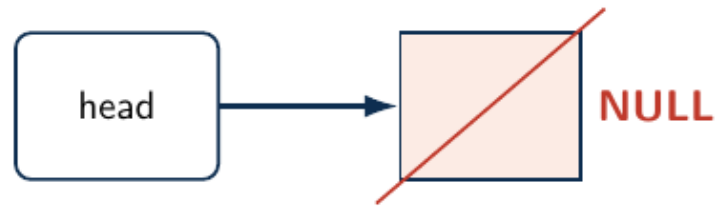
Per evitare di scrivere ripetutamente codice di allocazione nello Heap con gestione degli errori, implementiamo una funzione helper per creare nodi isolati:

```
1 Nodo* crea_nodo(int valore) {  
2     Nodo *nuovo = (Nodo*) malloc(sizeof(Nodo));  
3     if (nuovo == NULL) {  
4         fprintf(stderr, "Errore fatale: memoria Heap esaurita.\n");  
5         exit(EXIT_FAILURE);  
6     }  
7     nuovo->dato = valore;  
8     nuovo->prossimo = NULL; // Nasce scollegato  
9     return nuovo;  
10 }
```

Appena creata, la lista è composta solo della testa ed è dunque vuota, come mostrato nell'immagine seguente:

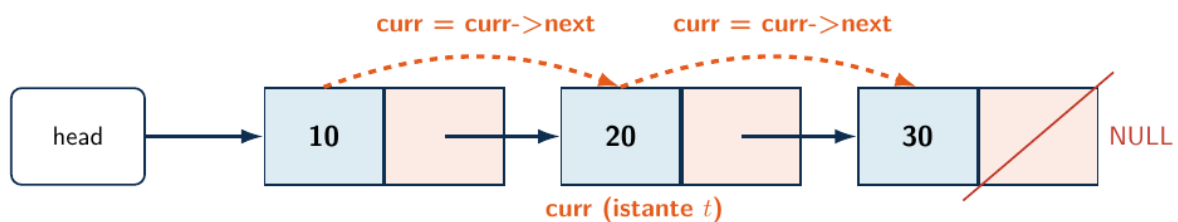
Possiamo verificare se una lista è vuota mediante la seguente funzione:

```
1 bool is_empty(Nodo *testa) {  
2     return testa == NULL;  
3 }
```



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

Figura 1.2: Lista vuota



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

Figura 1.3: Attraversamento della lista

1.2.2 Scorrimento Sequenziale e Stampa

Per scorrere la lista, non possiamo ricorrere ad indici numerici diretti. Dobbiamo usare un puntatore ausiliario (convenzionalmente chiamato *corrente* o *curr*) posizionato inizialmente sulla testa, facendolo avanzare progressivamente aggiornando il suo indirizzo con il valore salvato nel campo *prossimo*.

```

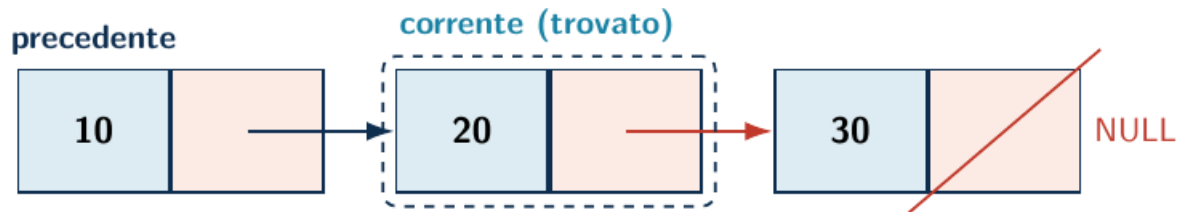
1 void stampa_lista(Nodo *testa) {
2     Nodo *corrente = testa;
3     printf("Testa -> ");
4     while (corrente != NULL) {
5         printf("[%d] -> ", corrente->dato);
6         corrente = corrente->prossimo; // Passo di avanzamento
7     }
8     printf("NULL\n");
9 }
  
```

1.2.3 Ricerca di un Elemento

La ricerca in una lista concatenata non ordinata è intrinsecamente lineare e richiede di scorrere i nodi uno ad uno:

```

1 bool cerca_elemento(Nodo *testa, int chiave, Nodo **nodo_trovato) {
2     Nodo *corrente = testa;
  
```



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

Figura 1.4: Ricerca del nodo

```

3  while (corrente != NULL) {
4      if (corrente->dato == chiave) {
5          if (nodo_trovato != NULL) {
6              *nodo_trovato = corrente;
7          }
8          return true;
9      }
10     corrente = corrente->prossimo;
11 }
12 if (nodo_trovato != NULL) {
13     *nodo_trovato = NULL;
14 }
15 return false;
16 }

```

Quando non si conosce già la posizione del nodo, si scorre la lista con due puntatori, precedente e corrente, finché `corrente->dato` non corrisponde al valore cercato; poi si applica la stessa logica di detach.

1.3 La Chirurgia dei Puntatori: Inserimenti

L'inserimento di elementi in una lista non richiede mai lo spostamento fisico dei nodi in memoria, bensì la riconfigurazione locale delle connessioni dei puntatori.

1.3.1 Inserimento in Testa

Inserire in testa è un'operazione che richiede tempo costante:

1. Si alloca un nuovo nodo.
2. Si fa puntare il campo `prossimo` del nuovo nodo alla vecchia testa.
3. Si aggiorna il puntatore di testa affinché contenga l'indirizzo del nuovo nodo.

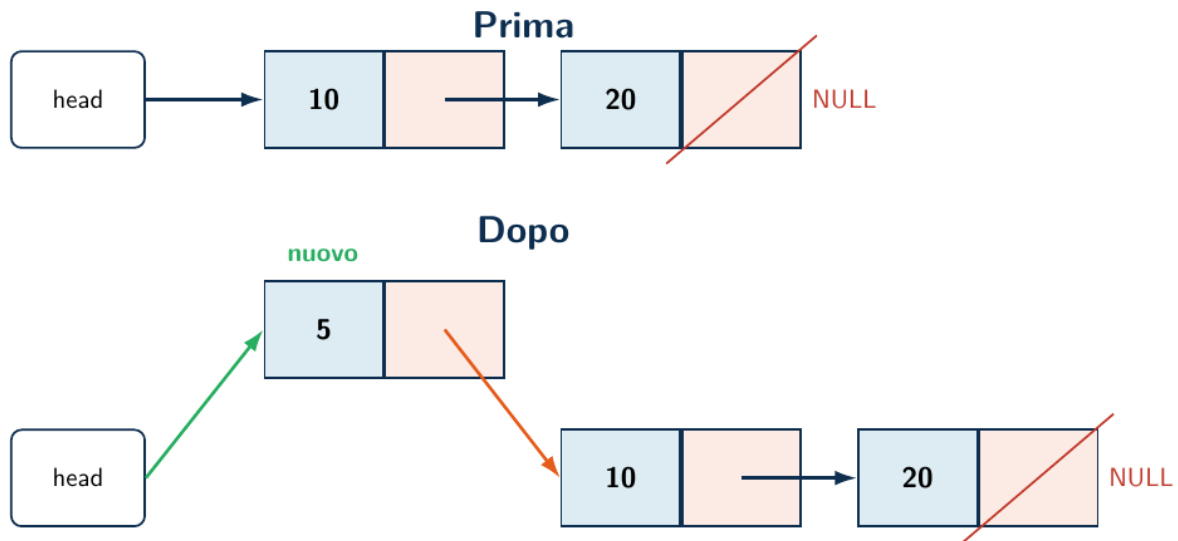


Figura 1.5: Inserimento in testa

```

1  Nodo* inserisci_testa(Nodo *testa, int valore) {
2      Nodo *nuovo = crea_nodo(valore);
3      nuovo->prossimo = testa;
4      return nuovo; // Diventa la nuova testa
5  }

```

Nelle figure sottostanti viene mostrato il passaggio dall'allocazione isolata del nuovo nodo all'aggancio definitivo.

Il codice sopra funziona anche nel caso speciale in cui inseriamo un elemento in una lista vuota:

1.3.2 Inserimento in Coda

Per inserire un elemento in fondo alla lista, è necessario individuare l'ultimo nodo (ovvero il nodo che punta a NULL), modificando il suo campo `prossimo` affinché punti al nuovo nodo.

- Se la lista è vuota (`head == NULL`), il nuovo nodo diventa direttamente la testa.
- Se la lista contiene elementi, si effettua una scansione lineare per posizionarsi sull'ultimo elemento.

```

1  Nodo* inserisci_coda(Nodo *testa, int valore) {
2      Nodo *nuovo = crea_nodo(valore);
3      if (testa == NULL) {
4          return nuovo;
5      }
6      Nodo *corrente = testa;
7      while (corrente->prossimo != NULL) {
8          corrente = corrente->prossimo;
9      }
10     corrente->prossimo = nuovo;

```

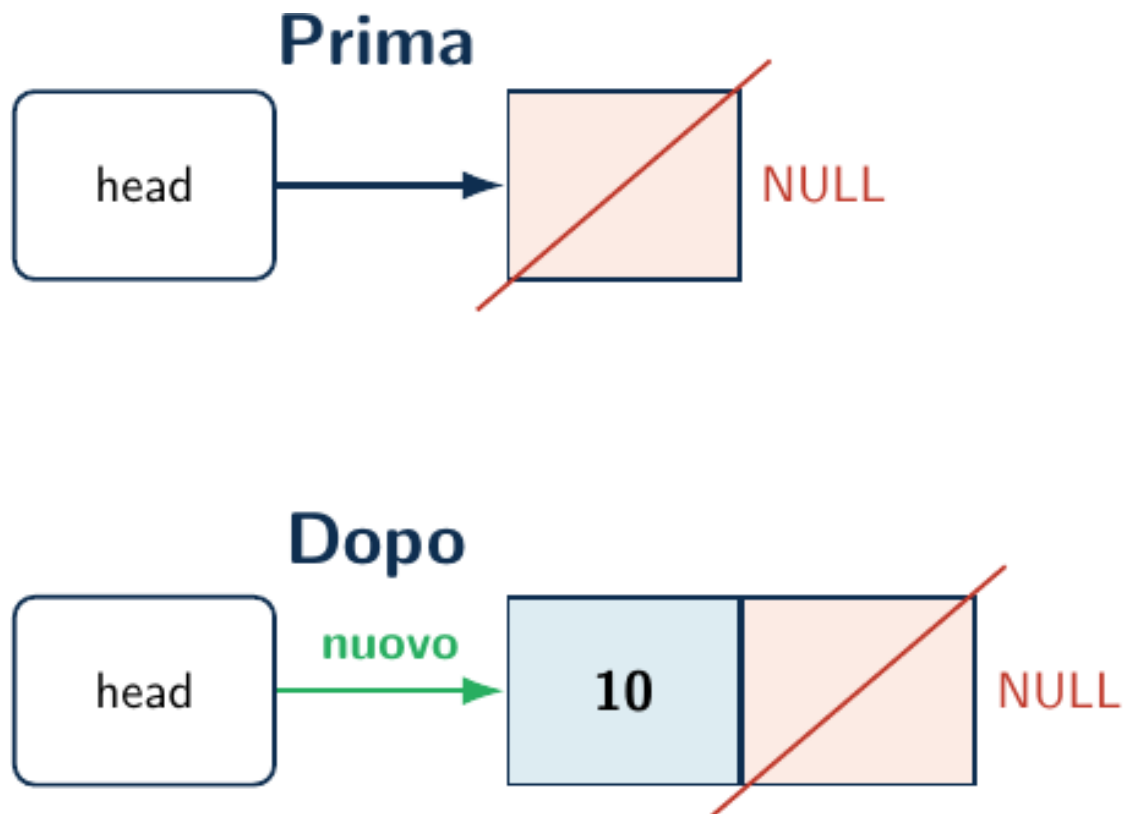


Figura 1.6: Inserimento in lista vuota

```
11  return testa;  
12 }
```

1.3.3 Inserimento Ordinato

L'inserimento ordinato in una lista crescente richiede di trovare la posizione esatta in mezzo a due nodi esistenti. Poiché la lista semplice è a senso unico, una volta superato il punto di inserimento non è possibile tornare indietro per aggiornare il puntatore del nodo che precede. Per risolvere questo problema, si utilizza la **tecnica dei due puntatori sfalsati** (precedente e corrente).

```
1  Nodo* inserisci_ordinato(Nodo *testa, int valore) {  
2      Nodo *nuovo = crea_nodo(valore);  
3  
4      // Caso 1: Lista vuota o inserimento in testa  
5      if (testa == NULL || valore < testa->dato) {  
6          nuovo->prossimo = testa;  
7          return nuovo;  
8      }  
9  
10     // Caso 2: Ricerca della posizione intermedia  
11     Nodo *precedente = testa;  
12     Nodo *corrente = testa->prossimo;  
13 }
```

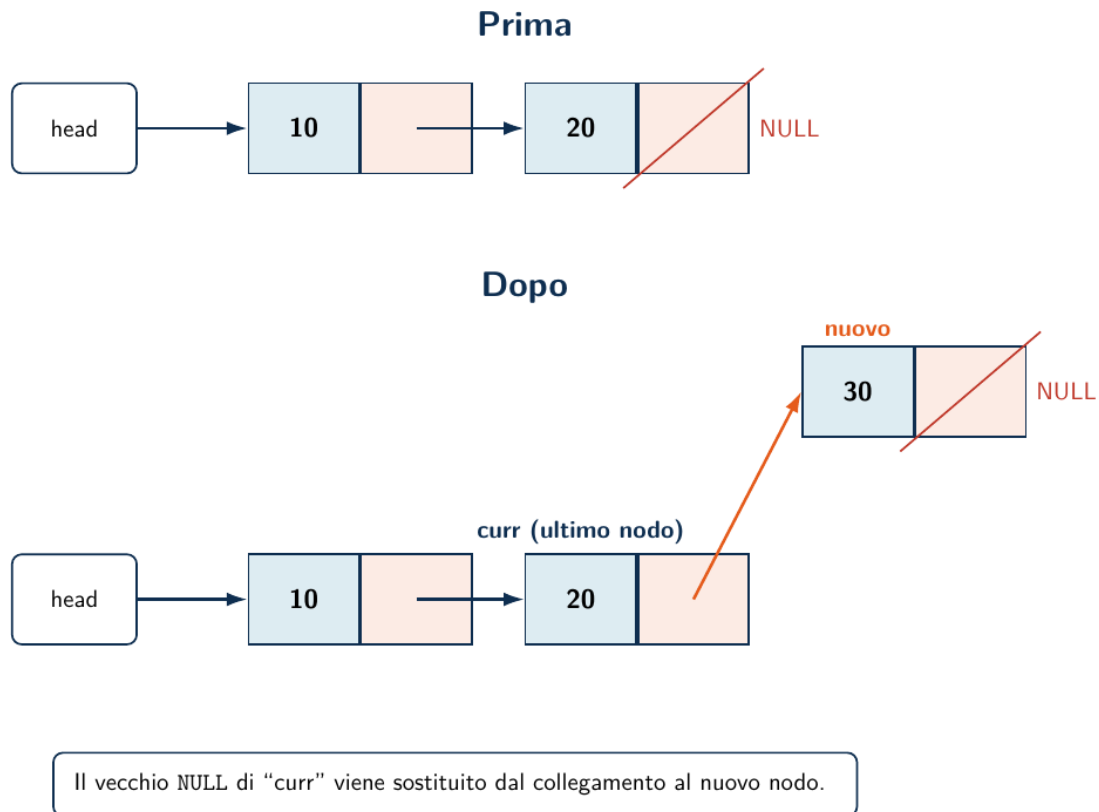


Figura 1.7: Inserimento in coda

```

14  while (corrente != NULL && corrente->dato < valore) {
15      precedente = corrente;
16      corrente = corrente->prossimo;
17  }
18
19  // Chirurgia di collegamento
20  precedente->prossimo = nuovo;
21  nuovo->prossimo = corrente;
22
23  return testa;
24  }

```

1.4 La Chirurgia dei Puntatori: Cancellazioni

La rimozione di un nodo richiede di bypassarlo (scavalcarlo) modificando il puntatore del nodo precedente, ed eliminando fisicamente l'elemento dallo Heap tramite free.

Attenzione

Pericolo di Memory Leak È tassativo salvare l'indirizzo del nodo da eliminare in un puntatore di supporto *prima* di modificare le connessioni. Se si effettua la free del nodo senza preservare il suo campo *prossimo*, si perde il collegamento con tutti i

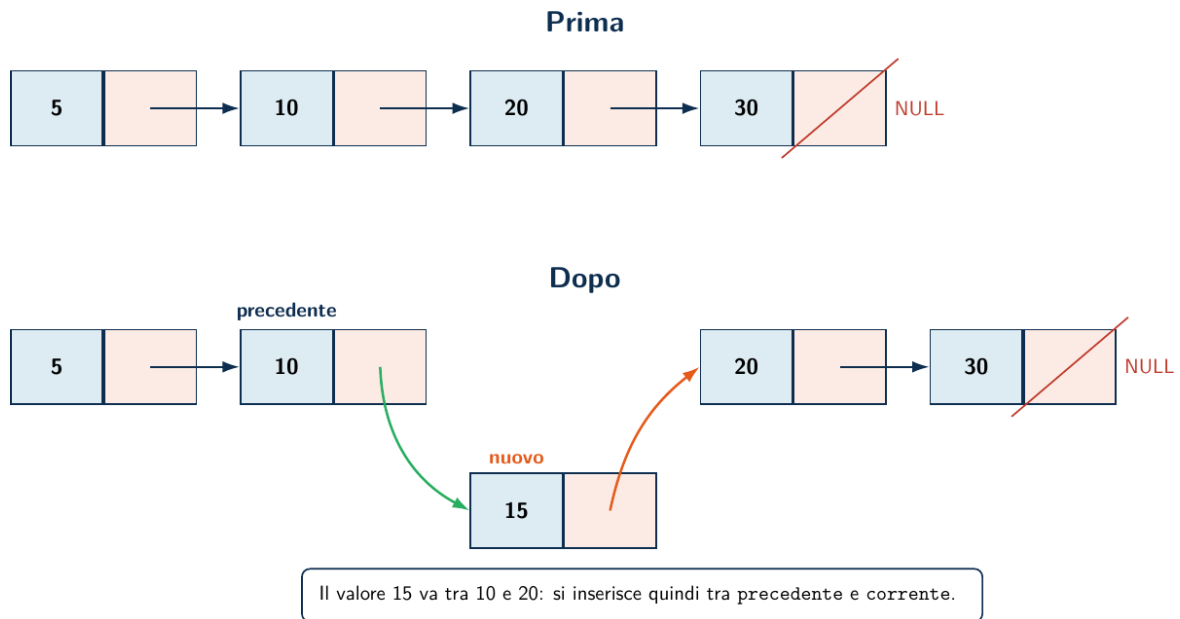


Figura 1.8: Inserimento ordinato

nodì successivi, rendendoli orfani e inaccessibili nello Heap.

1.4.1 Cancellazione in Testa

```

1  Nodo* rimuovi_testa(Nodo *testa) {
2      if (testa == NULL) return NULL;
3
4      Nodo *da_liberare = testa; // Salvataggio riferimento
5      testa = testa->prossimo;   // Spostamento testa
6      free(da_liberare);        // Rimozione sicura
7      return testa;
8  }

```

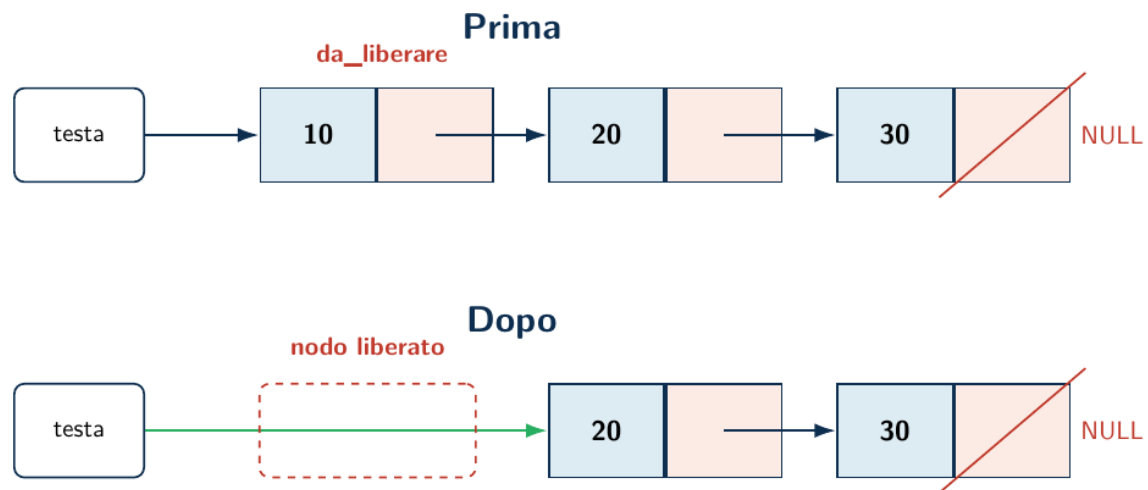
1.4.2 Cancellazione in Coda

Per rimuovere l'ultimo elemento della lista, occorre scorrere l'intera lista per posizionarsi sul penultimo nodo. Una volta individuato il penultimo nodo, si imposta il suo campo `prossimo` a `NULL` e si libera la memoria dell'ultimo nodo.

```

1  Nodo* rimuovi_coda(Nodo *testa) {
2      if (testa == NULL) return NULL;
3
4      // Se c'è solo un nodo
5      if (testa->prossimo == NULL) {
6          free(testa);
7          return NULL;
8      }

```



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

Figura 1.9: Cancellazione in testa

```

9
10     Nodo *precedente = NULL;
11     Nodo *corrente = testa;
12     while (corrente->prossimo != NULL) {
13         precedente = corrente;
14         corrente = corrente->prossimo;
15     }
16     precedente->prossimo = NULL;
17     free(corrente);
18     return testa;
19 }

```

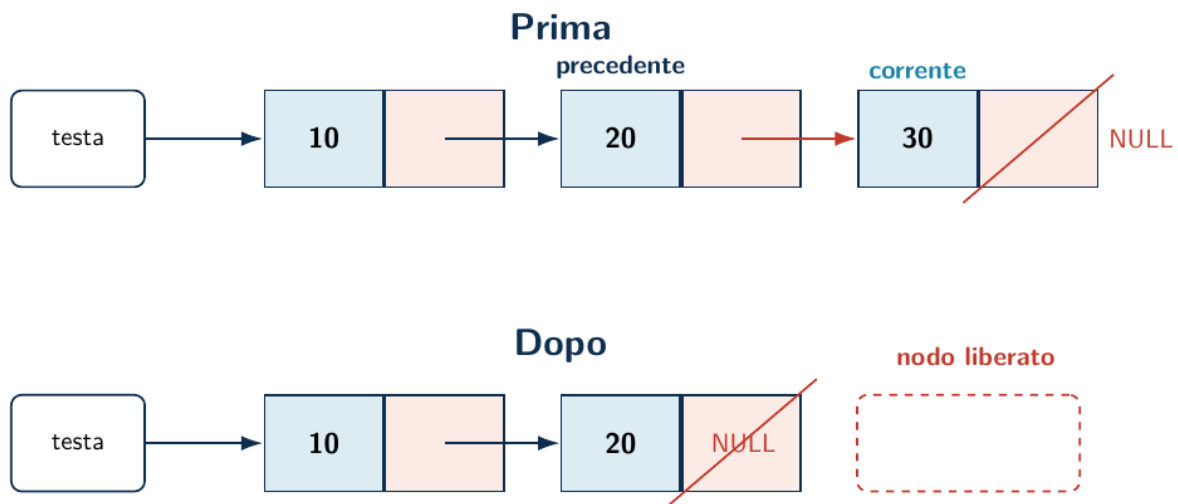
1.4.3 Cancellazione di un Elemento Specifico

Per rimuovere un nodo contenente un dato valore, si esegue una scansione lineare con due puntatori sfalsati. Una volta individuato il valore, si esegue lo scavallo del puntatore (precedente->prossimo = corrente->prossimo).

```

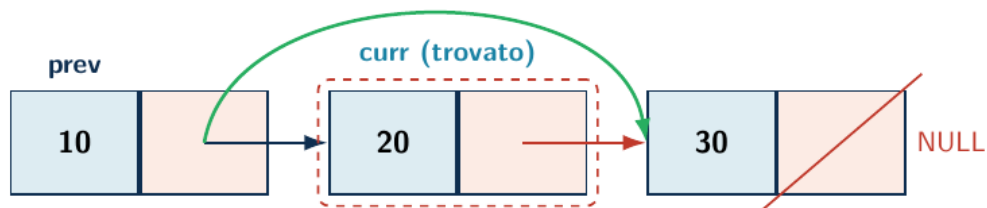
1  Nodo* rimuovi_valore(Nodo *testa, int valore) {
2      if (testa == NULL) return NULL;
3
4      // Rimozione in testa
5      if (testa->dato == valore) {
6          Nodo *nuova_testa = testa->prossimo;
7          free(testa);
8          return nuova_testa;
9      }
10
11     Nodo *precedente = testa;
12     Nodo *corrente = testa->prossimo;

```



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

Figura 1.10: Cancellazione in coda



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

Figura 1.11: Ricerca e cancellazione

```

13
14  while (corrente != NULL && corrente->dato != valore) {
15      precedente = corrente;
16      corrente = corrente->prossimo;
17  }
18
19  if (corrente != NULL) {
20      precedente->prossimo = corrente->prossimo; // Detach
21      free(corrente); // Deallocazione
22  }
23
24  return testa;
25  }
```

1.4.4 Deallocazione dell'Intera Lista

Sebbene i moderni sistemi operativi liberino in automatico lo spazio di indirizzamento utilizzato da un programma, di fatto liberando lo spazio delle free non effettuate al momento della chiusura del programma, in ambienti di programmazione di basso livello (es. microcontrollori), non liberare la memoria può dare vita a memory leak persistenti anche alla chiusura del programma. Prima del termine del programma, è pertanto buona norma distruggere ogni lista allocata dinamicamente:

```

1 void svuota_lista(Nodo *testa) {
2     Nodo *corrente = testa;
3     while (corrente != NULL) {
4         Nodo *successore = corrente->prossimo; // Preserva il futuro
5         free(corrente);                       // Dealloca il presente
6         corrente = successore;                 // Avanza
7     }
8 }

```

1.4.5 Ordinamento su Liste (Bubble Sort)

Ora che sappiamo navigare e modificare una lista, come facciamo ad ordinarne una nata disordinata? Negli array usavamo il Bubble Sort scambiando gli elementi con un algoritmo di swap. Sulle liste abbiamo due opzioni:

- **Opzione A (Swap dei Dati):** Copiamo e scambiamo unicamente il valore contenuto nei campi dato dei nodi, lasciando i nodi fisicamente dove si trovano nello Heap.
- **Opzione B (Swap dei Puntatori):** Sganciamo fisicamente i nodi e riallineiamo i loro puntatori prossimo per scambiarne l'ordine.

La Scelta Ingegneristica

Scambiare i puntatori (Opzione B) è un'operazione complessa che richiede di aggiornare fino a 4 indirizzi diversi per ogni singolo swap, aumentando drasticamente il rischio di perdere nodi o creare cicli infiniti. Per garantire la stabilità del codice e la semplicità didattica, l'**Opzione A (Swap dei Dati)** è la scelta ottimale, in quanto ricalca in modo pulito ed intuitivo la logica del Bubble Sort sugli array!

L'algoritmo di Bubble Sort applicato alla Lista

Sostituiamo gli indici numerici $[i]$ e $[i+1]$ con i puntatori di navigazione corrente e `corrente->prossimo`:

```

1 void ordina_lista(Nodo *testa) {
2     // Caso banale: lista vuota o con un solo elemento (gia' ordinata)
3     if (testa == NULL || testa->prossimo == NULL) {
4         return;
5     }
6
7     bool scambiato;

```

```
8  Nodo *corrente;
9  Nodo *limite = NULL; // Ottimizzazione: delimita la fine della
   ↪ scansione
10
11  do {
12      scambiato = false;
13      corrente = testa;
14
15      // Scorriamo la lista fino al limite degli elementi già
   ↪ consolidati
16      while (corrente->prossimo != limite) {
17          // Confrontiamo elementi adiacenti sequenziali
18          if (corrente->dato > corrente->prossimo->dato) {
19              // LA CHIRURGIA DEL PAYLOAD: Scambio in-place dei dati
20              int temp = corrente->dato;
21              corrente->dato = corrente->prossimo->dato;
22              corrente->prossimo->dato = temp;
23
24              scambiato = true; // Segnaliamo che e' avvenuto uno
   ↪ scambio
25          }
26          corrente = corrente->prossimo; // Passo in avanti
27      }
28      // L'ultimo elemento esaminato e' ora sicuramente al suo posto
   ↪ corretto
29      limite = corrente;
30
31  } while (scambiato); // Se non ci sono stati scambi, l'intera lista e
   ↪ 'ordinata!
32 }
```

1.5 Quiz di Autovalutazione

Quiz 1

Domanda

1. Come si inizializza correttamente una lista vuota in C?

- A) Dichiarando un puntatore a Nodo e assegnandogli il valore NULL (es. `Nodo *testa = NULL;`).
- B) Allocando subito un nodo vuoto tramite `malloc`.
- C) Dichiarando una variabile `Nodo` locale nello Stack senza inizializzarla.
- D) Impostando il valore del primo nodo a 0.

Soluzione**Risposta corretta: A**

Spiegazione: Per indicare che una lista non contiene alcun elemento, il puntatore di testa (testa) deve essere impostato esplicitamente a NULL. Questo evita che contenga indirizzi spazzatura e consente alle funzioni di controllo di rilevarne lo stato vuoto.

Quiz 2**Domanda****2. Qual è l'effetto di perdere l'indirizzo memorizzato nella variabile di testa head?**

- A) La lista si inverte automaticamente in memoria.
- B) Un memory leak irreversibile di tutti i nodi della lista.
- C) I nodi vengono deallocati automaticamente dal Garbage Collector di C.
- D) Il programma continua a funzionare regolarmente accedendo tramite indici.

Soluzione**Risposta corretta: B**

Spiegazione: Nello Heap, i nodi sono accessibili unicamente conoscendo l'indirizzo del primo nodo (head). Se questa variabile viene persa o sovrascritta senza prima aver deallocato la memoria, gli indirizzi dei nodi diventano intrinsecamente irraggiungibili per la CPU, bloccando quella porzione di RAM fino al termine del processo.

Quiz 3**Domanda****3. Cosa contiene il campo prossimo dell'ultimo nodo di una lista ben formata?**

- A) L'indirizzo della testa (head).
- B) Un valore casuale sporco di memoria (*garbage value*).
- C) Il valore speciale NULL.
- D) L'indirizzo del nodo stesso.

Soluzione**Risposta corretta: C**

Spiegazione: In C, NULL indica l'assenza di un indirizzo di memoria valido ed è utilizzato come sentinella standard per arrestare i cicli di scorrimento delle liste.

Quiz 4

Domanda

4. Quale operazione sui puntatori è necessaria per inserire un nuovo nodo in testa a una lista non vuota?

- A) Far puntare il campo `prossimo` del nuovo nodo a `NULL` e deallocare la testa.
- B) Far puntare il campo `prossimo` del nuovo nodo al nodo che era precedentemente in testa, e poi aggiornare il puntatore di testa per farlo puntare al nuovo nodo.
- C) Spostare tutti i valori della lista nei nodi adiacenti verso destra.
- D) Modificare l'indirizzo del puntatore di testa lasciando inalterato il campo `prossimo` del nuovo nodo.

Soluzione
Risposta corretta: B
Spiegazione: Per inserire un nodo in testa senza perdere il collegamento con i nodi già esistenti, il nuovo nodo deve prima "agganciarsi" alla vecchia testa salvandone l'indirizzo nel suo campo `prossimo`. Solo a questo punto possiamo muovere il puntatore di testa sul nuovo nodo.

Quiz 5

Domanda

5. Perché nell'allocazione dinamica di un nodo è importante verificare il valore di ritorno della `malloc`?

- A) Per evitare memory leaks.
- B) Perché la `malloc` restituisce `NULL` in caso di memoria Heap esaurita, provocando crash (Segmentation Fault) in caso di mancato controllo.
- C) Per verificare che il dato inserito sia di tipo numerico.
- D) Per liberare automaticamente il nodo deallocato in precedenza.

Soluzione
Risposta corretta: B
Spiegazione: Tentare di dereferenziare un puntatore `NULL` (es. `nuovo->dato = valore` quando nuovo è `NULL`) provoca un crash immediato del programma (Segmentation Fault).

Quiz 6

Domanda

6. Nella funzione `inserisci\{}_coda`, perché il ciclo di scorrimento si ferma quando

corrente->prossimo != NULL anziché corrente != NULL?

- A) Perché altrimenti si verificherebbe un ciclo infinito.
- B) Per evitare di allocare due nodi contemporaneamente.
- C) Perché abbiamo bisogno di fermarci esattamente sull'ultimo nodo per poterne modificare il campo prossimo.
- D) Perché l'ultimo nodo ha il puntatore al dato pari a 0.

Soluzione
Risposta corretta: C
Spiegazione: Se ci fermassimo quando corrente == NULL, avremmo superato la lista e non avremmo più un nodo valido su cui innestare il puntatore al nuovo elemento.

Quiz 7

Domanda

7. Nella rimozione di un nodo intermedio, perché sono necessari due puntatori (precedente e corrente)?

- A) Per raddoppiare la velocità di esecuzione della cancellazione.
- B) Perché la lista semplice è monodirezionale e serve precedente per scavalcare corrente una volta trovato il valore.
- C) Per gestire contemporaneamente l'inserimento e la rimozione.
- D) Perché non è possibile usare variabili locali all'interno della funzione.

Soluzione
Risposta corretta: B
Spiegazione: Una volta posizionati sul nodo da cancellare (corrente), non abbiamo alcun modo di risalire al nodo che lo precede per fare in modo che quest'ultimo "scavalchi" l'elemento rimosso.

Quiz 8

Domanda

8. Che cosa accade se si invoca free(testa) come prima istruzione di una funzione di deallocazione totale della lista?

- A) La memoria viene liberata correttamente e head viene aggiornato.
- B) Si perde l'indirizzo del nodo successivo, impedendo di scorrere e deallocare il resto della lista (causando leak).
- C) Il compilatore segnala un errore di sintassi bloccante.

- D) Tutti i nodi successivi vengono automaticamente spostati nello Stack.

Soluzione
Risposta corretta: B
Spiegazione: Liberare il nodo distrugge il campo prossimo. Bisogna sempre salvare l'indirizzo del successore in una variabile temporanea prima di deallocare il nodo corrente.

Quiz 9

Domanda

9. Quale delle seguenti dichiarazioni definisce correttamente un tipo nodo autoreferenziale?

- A) `typedef struct \{\}\ int dato; Nodo *prossimo; \{\}\ Nodo;`
- B) `typedef struct Nodo \{\}\ int dato; struct Nodo *prossimo; \{\}\ Nodo;`
- C) `typedef struct Nodo \{\}\ int dato; Nodo prossimo; \{\}\ Nodo;`
- D) `struct Nodo \{\}\ int dato; struct Nodo prossimo; \{\}\;`

Soluzione
Risposta corretta: B
Spiegazione: Il compilatore ha bisogno di conoscere l'esistenza di `struct Nodo` per dichiarare il puntatore interno prima che la definizione del tipo alias `Nodo` sia completata.

Quiz 10

Domanda

10. Qual è il principale svantaggio delle liste concatenate rispetto agli array statici?

- A) Il fatto che non possono memorizzare valori negativi.
- B) Mancanza dell'accesso casuale e consumo extra di memoria per memorizzare i puntatori.
- C) L'impossibilità di usare tipi strutturati (es. `struct`) come payload del nodo.
- D) Il rischio costante di esaurire lo Stack di sistema a causa dell'allocazione dinamica.

Soluzione
Risposta corretta: B
Spiegazione: Per ogni elemento salvato in lista, occorre consumare dello spazio aggiuntivo in memoria (4 o 8 byte) per salvare l'indirizzo del nodo successivo.

1.6 Esercizi Pratici Risolti

Esercizio 1: Conteggio dei Nodi

Domanda

Scrivere una funzione `int conta_nodi(Nodo *testa)` che restituisca il numero totale di elementi presenti nella lista.

```

1  #include <stdio.h>
2
3  typedef struct Nodo {
4      int dato;
5      struct Nodo *prossimo;
6  } Nodo;
7
8  int conta_nodi(Nodo *testa) {
9      int conteggio = 0;
10     Nodo *corrente = testa;
11     while (corrente != NULL) {
12         conteggio++;
13         corrente = corrente->prossimo;
14     }
15     return conteggio;
16 }
```

Soluzione

Esercizio 2: Somma e Media dei Nodi

Domanda

Scrivere una funzione `double media_lista(Nodo *testa)` che calcoli la media aritmetica dei valori interi contenuti nei nodi di una lista semplice. Nel caso di lista vuota, la funzione deve restituire `0.0`.

```

1  #include <stdio.h>
2
3  typedef struct Nodo {
4      int dato;
5      struct Nodo *prossimo;
6  } Nodo;
7
8  double media_lista(Nodo *testa) {
9      if (testa == NULL) {
10         return 0.0;
11     }
12     int somma = 0;
13     int conteggio = 0;
14     Nodo *corrente = testa;
15     while (corrente != NULL) {
16         somma += corrente->dato;
17         conteggio++;
18         corrente = corrente->prossimo;
19     }
20     return (double)somma / conteggio;
21 }

```

Soluzione**Esercizio 3: Verifica Presenza Elemento (Ricerca Lineare)****Domanda**

Scrivere una funzione `bool contiene({}_valore(Nodo *testa, int target)` che restituisca `true` se il valore `target` è presente in almeno uno dei nodi della lista, e `false` altrimenti.

```

1  #include <stdbool.h>
2  #include <stdio.h>
3
4  typedef struct Nodo {
5      int dato;
6      struct Nodo *prossimo;
7  } Nodo;
8
9  bool contiene_valore(Nodo *testa, int target) {
10     Nodo *corrente = testa;
11     while (corrente != NULL) {
12         if (corrente->dato == target) {
13             return true; // Trovato, interruzione anticipata
14         }
15         corrente = corrente->prossimo;
16     }
17     return false; // Elemento non presente
18 }

```

Soluzione**Esercizio 4: Inserimento in Testa In-Place (Doppio****Domanda**

Puntatore)

Implementare la funzione `void inserisci\{\}_testa\{\}_doppio\{\}_ptr(Nodo **testa\{\}_ptr, int valore)` che inserisce un elemento in testa alla lista modificando direttamente la variabile di testa del chiamante passata tramite riferimento.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Nodo {
5      int dato;
6      struct Nodo *prossimo;
7  } Nodo;
8
9  void inserisci_testa_doppio_ptr(Nodo **testa_ptr, int valore) {
10     Nodo *nuovo = malloc(sizeof(Nodo));
11     if (nuovo == NULL) {
12         exit(EXIT_FAILURE);
13     }
14     nuovo->dato = valore;
15
16     // Il nuovo nodo punta al nodo attualmente referenziato dalla
17     // testa
18     nuovo->prossimo = *testa_ptr;
19
20     // Aggiorniamo il valore del puntatore originale nello Stack
21     *testa_ptr = nuovo;
22 }

```

Soluzione**Esercizio 5: Deallocazione Totale di una Lista****Domanda**

Scrivere una funzione `void distruggi_liste(Nodo **testa_ptr)` che liberi tutta la memoria occupata dai nodi della lista e, al termine, imposti il puntatore originale del chiamante a NULL per evitare puntatori pendenti (*dangling pointers*).

Analisi algoritmica: Si liberano individualmente gli N nodi allocati dinamicamente. Complessità temporale: $O(N)$. Azzerando `*testa\{}_ptr` garantiamo la sicurezza del codice chiamante che non tenterà di accedere a memoria non più valida.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Nodo {
5      int dato;
6      struct Nodo *prossimo;
7  } Nodo;
8
9  void distruggi_lista(Nodo **testa_ptr) {
10     if (*testa_ptr == NULL) return;
11     Nodo *corrente = *testa_ptr;
12     while (corrente != NULL) {
13         Nodo *prossimo = corrente->prossimo; // Conserviamo il link
14         // Deallocazione sicura
15         free(corrente); // del presente
16         corrente = prossimo; // Avanzamento
17     }
18     *testa_ptr = NULL; // Azzerare la variabile originale del
19                             ↪ chiamante
20 }

```

Soluzione

Esercizio 6: Inversione In-Place di una Lista (Reverse)

Domanda

Scrivere una funzione `Nodo* inverti\{}_lista(Nodo *testa)` che inverta l'ordine dei collegamenti dei nodi in modo che l'ultimo nodo diventi la nuova testa, e il primo nodo punti a NULL. L'operazione deve essere eseguita *in-place*, ovvero senza allocare nuovi nodi.

```

1  #include <stdio.h>
2
3  typedef struct Nodo {
4      int dato;
5      struct Nodo *prossimo;
6  } Nodo;
7
8  Nodo* inverti_lista(Nodo *testa) {
9      Nodo *precedente = NULL;
10     Nodo *corrente = testa;
11     Nodo *successore = NULL;
12
13     while (corrente != NULL) {
14         successore = corrente->prossimo; // Salva il resto della
15         ↪ lista
16         corrente->prossimo = precedente; // Inverte la direzione
17         ↪ del puntatore
18         precedente = corrente;
19         corrente = successore;
20     }
21     return precedente; // Nuova testa della lista invertita

```

Soluzione**Esercizio 7: Riconoscimento ed Eliminazione dei Duplicati****Domanda**

in Lista Ordinata

Scrivere una funzione `void rimuovi\_duplicati\_ordinata(Nodo *testa)` che, data una lista ordinata in modo crescente, rimuova tutti i nodi duplicati adiacenti mantenendo una sola copia di ciascun valore.

Soluzione

Analisi algoritmica: Poiché la lista è già ordinata, i duplicati sono necessariamente vicini. La scansione richiede tempo $O(N)$ ed effettua la free dei soli elementi ridondanti.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Nodo {
5      int dato;
6      struct Nodo *prossimo;
7  } Nodo;
8
9  void rimuovi_duplicati_ordinata(Nodo *testa) {
10     if (testa == NULL) return;
11     Nodo *corrente = testa;
12     while (corrente->prossimo != NULL) {
13         if (corrente->dato == corrente->prossimo->dato) {
14             // Rilevato duplicato adiacente
15             Nodo *doppione = corrente->prossimo;
16             corrente->prossimo = doppione->prossimo; // Scavalamento
17             free(doppione);
18         } else {
19             corrente = corrente->prossimo; // Avanziamo solo se non
20             // c'era duplicato
21         }
22     }
23 }
```

Esercizio 8: Unione di Due Liste Ordinate (Merge)

Domanda

Scrivere una funzione `Nodo* fondi({}_liste){}_ordinate(Nodo *l1, Nodo *l2)` che prenda in input due liste ordinate in modo crescente e restituisca una terza lista ordinata contenente l'unione dei nodi delle due liste originali. È possibile staccare e riutilizzare i nodi esistenti.

Soluzione

```

1  #include <stdio.h>
2
3  typedef struct Nodo {
4      int dato;
5      struct Nodo *prossimo;
6  } Nodo;
7
8  Nodo* fondi_liste_ordinate(Nodo *l1, Nodo *l2) {
9      // Casi base se una delle due liste è vuota
10     if (l1 == NULL) return l2;
11     if (l2 == NULL) return l1;
12     Nodo *nuova_testa = NULL;
13
14     // Determina il primo nodo della nuova lista
15     if (l1->dato <= l2->dato) {
16         nuova_testa = l1;
17         l1 = l1->prossimo;
18     } else {
19         nuova_testa = l2;
20         l2 = l2->prossimo;
21     }
22
23     Nodo *corrente = nuova_testa;
24
25     // FONDE confrontando gli elementi testa a testa
26     while (l1 != NULL && l2 != NULL) {
27         if (l1->dato <= l2->dato) {
28             corrente->prossimo = l1;
29             l1 = l1->prossimo;
30         } else {
31             corrente->prossimo = l2;
32             l2 = l2->prossimo;
33         }
34     }
35     corrente = corrente->prossimo;
36
37     // Collega l'eventuale coda rimanente di una delle due liste
38     if (l1 != NULL) {
39         corrente->prossimo = l1;
40     } else {
41         corrente->prossimo = l2;
42     }
43
44     return nuova_testa;
45 }
46

```

Esercizio 9: Individuazione del Nodo Mediano (Puntatori)**Domanda**

Veloce e Lento)

Implementare una funzione `Nodo* trova_{}_mediano(Nodo *testa)` che restituisca l'indirizzo di memoria del nodo centrale di una lista semplice senza calcolarne preventivamente la lunghezza totale. Se la lista ha lunghezza pari, si consideri il primo dei due nodi centrali.

```

1  #include <stdio.h>
2
3  typedef struct Nodo {
4      int dato;
5      struct Nodo *prossimo;
6  } Nodo;
7
8  Nodo* trova_mediano(Nodo *testa) {
9      if (testa == NULL) return NULL;
10     Nodo *lento = testa;
11     Nodo *veloce = testa;
12
13     // Il puntatore veloce cammina al doppio della velocità del
14     // lento
15     while (veloce->prossimo != NULL && veloce->prossimo->prossimo
16            != NULL) {
17         lento = lento->prossimo;
18         veloce = veloce->prossimo->prossimo;
19     }
20     return lento; // Quando veloce arriva alla fine, lento è a metà?
21 }

```

Soluzione**Esercizio 10: Concatenazione in Coda di Due Liste (Append)****Domanda**

Scrivere una funzione `Nodo* concatena_{}_liste(Nodo *l1, Nodo *l2)` che colleghi la lista `l2` in coda alla lista `l1`. La funzione deve restituire la testa della nuova lista unificata.

Soluzione

```

1  #include <stdio.h>
2
3  typedef struct Nodo {
4      int dato;
5      struct Nodo *prossimo;
6  } Nodo;
7
8  Nodo* concatena_liste(Nodo *l1, Nodo *l2) {
9      // Caso speciale: se la prima lista è vuota, restituiamo
10     ↪ direttamente la seconda
11     if (l1 == NULL) {
12         return l2;
13     }
14     Nodo *corrente = l1;
15     // Raggiungiamo l'ultimo nodo di l1
16     while (corrente->prossimo != NULL) {
17         corrente = corrente->prossimo;
18     }
19     // Aggancio del primo nodo di l2 alla coda di l1
20     corrente->prossimo = l2;
21     return l1;
22 }
23
24 }

```