Strutture dati

Matteo Spanio Giulio Pitteri

8 maggio 2025

In questa lezione si propone un ripasso alle strutture dati già viste aggiungendo l'implementazione di base degli alberi.

Liste

Fino ad ora abbiamo visto alcune implementazioni di liste:

- array list
- linked list
- doubly linked list
- xor linked list

Le liste sono strutture dati lineari in cui gli elementi sono collegati tra loro tramite puntatori. Una delle varianti più comuni è la lista doppiamente concatenata.

Implementazione

Vediamo di seguito un esempio di implementazione di una lista doppiamente concatenata. La lista è composta da nodi, ognuno dei quali contiene un valore e due puntatori: uno al nodo successivo e uno al nodo precedente. In questo modo, è possibile navigare sia in avanti che all'indietro nella lista.

```c
typedef struct node {
    int value;
    struct node *next;
    struct node *prev;
} Node;

typedef struct list {
    Node *head;
    Node *tail;
    int size;
} List;
```

Ogni nodo contiene:

- un valore intero (int value)
- un puntatore al nodo successivo (*next)
- un puntatore al nodo precedente (*prev)
- La lista mantiene un puntatore alla testa e alla coda, oltre alla dimensione.

Funzioni principali

```
Node *new_node(int value);
List *new_list(void);
void append(List *list, int value);
void prepend(List *list, int value);
void insert(List *list, int index, int value);
void free_list(List *list);
```

Funzione	Descrizione	Complessità
new_node	Crea un nuovo nodo con il valore specificato.	O(1)
new_list	Crea una nuova lista vuota.	O(1)
append	Aggiunge un nuovo nodo alla fine della lista.	O(1)
prepend	Aggiunge un nuovo nodo all'inizio della lista.	O(1)
insert	Inserisce un nuovo nodo in una posizione specificata.	O(n)
free_list	Libera la memoria allocata per la lista.	O(n)

Vedi un esempio di implementazione di una lista doppiamente concatenata.

Stack (Pila)

Le liste possono essere usate per implementare uno stack, una struttura dati LIFO (Last In First Out).

Implementazione

```c
typedef struct node {
    int value;
    struct node *next;
} Node;

typedef struct stack {
    Node *top;
    int size;
} Stack;
```

Ogni nodo contiene:

- un valore intero (int value)
- un puntatore al nodo successivo (*next)
- La lista mantiene un puntatore alla testa, che rappresenta l'elemento in cima allo stack.

Funzioni principali

```c
void push(Stack *stack, int value);
int pop(Stack *stack);
int is_empty(Stack *stack);
```

Funzione	Descrizione	Complessità
push	Aggiunge un nuovo nodo in cima allo stack.	O(1)
pop	Rimuove e restituisce il nodo in cima allo stack.	O(1)
is_empty	Controlla se lo stack è vuoto.	O(1)

Vedi un esempio di implementazione di uno stack.

Queue

Le liste possono essere usate per implementare una coda, una struttura dati FIFO (First In First Out).

Implementazione

```c
typedef struct node {
    int value;
    struct node *next;
} Node;

typedef struct queue {
    Node *front;
    Node *rear;
    int size;
} Queue;
```

Ogni nodo contiene:

- un valore intero (int value)
- un puntatore al nodo successivo (*next)
- La lista mantiene due puntatori: uno alla testa (front) e uno alla coda (rear).

Funzioni principali

```c
void enqueue(Queue *queue, int value);
int dequeue(Queue *queue);
int is_empty(Queue *queue);
```

Funzione	Descrizione	Complessità
enqueue	Aggiunge un nuovo nodo alla fine della coda.	O(1)
dequeue	Rimuove e restituisce il nodo in testa alla coda.	O(1)
is_empty	Controlla se la coda è vuota.	O(1)

Vedi un esempio di implementazione di una queue.

Alberi

Gli alberi sono una struttura dati gerarchica in cui ogni nodo ha un valore e può avere zero o più figli. Gli alberi sono usati per rappresentare strutture dati complesse come i file system, i database e le strutture di dati in memoria.

Implementazione

```c
typedef struct node {
    int value;
    struct node *parent;
    struct node *left;
    struct node *right;
} Node;
```

Ogni nodo contiene:

- un valore intero (int value)
- un puntatore al nodo padre (*parent)
- un puntatore al nodo sinistro (*left)
- un puntatore al nodo destro (*right)

Funzioni principali

```c
Node *new_node(int value);
Node *insert_left(Node *parent, int value);
Node *insert_right(Node *parent, int value);
void inorder_traversal(Node *node);
void preorder_traversal(Node *node);
void postorder_traversal(Node *node);
```

Funzione	Descrizione	Complessità
new_node	Crea un nuovo nodo con il valore specificato.	O(1)
insert_left	Inserisce un nuovo nodo a sinistra del nodo padre.	O(1)
insert_right	Inserisce un nuovo nodo a destra del nodo padre.	O(1)
inorder_traversal	Visita i nodi in ordine (sinistra, radice, destra).	O(n)
preorder_traversal	Visita i nodi in preordine (radice, sinistra, destra).	O(n)
postorder_traversal	Visita i nodi in postordine (sinistra, destra, radice).	O(n)

Vedi un esempio di implementazione di un albero.