Strutture di dati

Strutture di dati

- Abbiamo studiato strutture di dati di **dimensione fissa** come
  - Array unidimensionali
  - Array bidimensionali
  - Strutture (`struct`)
- Vedremo strutture di dati **dinamiche** che possono crescere e ridursi durante l'esecuzione
  - **Liste collegate** (`linked list`)
    - Collezioni di elementi di dati «allineati in una riga»
    - Permettono di inserire ed eliminare elementi da qualsiasi punto
  - **Pile** (`stack`)
    - Utilizzate nei compilatori e nei sistemi operativi
    - Permettono di inserire ed eliminare elementi solo dalla cima della pila
  - **Code** (`queue`)
    - Rappresentano le file di attesa
    - Permettono di inserire elementi solo alla fine della coda ed eliminarli solo dalla testa
  - **Alberi binari** (`binary tree`)
    - Facilitano la ricerca e l'ordinamento dei dati ad alta velocità
    - Permettono l'eliminazione efficiente dei dati duplicati
    - Utilizzati per la compilazione di espressioni nel linguaggio macchina
- Ogni struttura di dati ha molte applicazioni interessanti

Strutture autoreferenziali

- Una **struttura autoreferenziale** contiene un membro puntatore che punta a una struttura dello stesso tipo

```
struct node {
    int data;
    struct node *nextPtr;
};
```

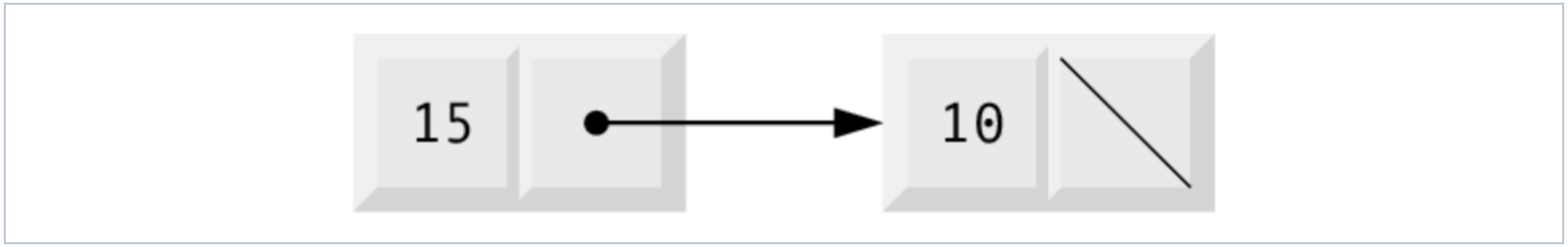
- La struttura `struct node` ha due membri
  - `data`, un membro di tipo `int`
  - `nextPtr`, un membro di tipo puntatore a `struct node`
- `nextPtr` punta a un'altra struttura `struct node`
- È chiamato link (collegamento) e permette di legare una struttura a un'altra dello stesso tipo

Strutture autoreferenziali

- La struttura struct node è un esempio di struttura autoreferenziale
- Le strutture autoreferenziali possono essere collegate per formare
  - Liste
  - Code
  - Pile
  - Alberi

Strutture autoreferenziali

- Due oggetti aventi lo stesso tipo collegati per formare una lista



Fonte: Deitel & Deitel

- La barra nell'ultimo nodo rappresenta un puntatore `NULL`
- Il puntatore `NULL` indica la fine della struttura di dati (es. lista)

Gestione dinamica della memoria

- Strutture dinamiche necessitano una gestione dinamica della memoria
- La gestione dinamica della memoria ha due componenti
  - Ottenere ulteriore spazio di memoria per contenere nuovi nodi
  - Liberare spazio non più necessario
- Per la gestione della memoria sono essenziali:
  - La funzione `malloc`
  - La funzione `free`
  - L'operatore `sizeof`

`malloc`

- La funzione `malloc` richiede memoria in fase di esecuzione
- Passate come parametro a `malloc` il numero di byte da allocare
- In caso di successo, `malloc` restituisce un puntatore `void *` alla memoria allocata
- Il puntatore `void *` può essere assegnato a una variabile di qualsiasi tipo di puntatore
- Normalmente si usa l'operatore `sizeof` per determinare i byte necessari

```
struct node* newPtr = malloc(sizeof(struct node));
```

- Non è garantito che la memoria sia inizializzata
- Se non è disponibile memoria, `malloc` restituisce `NULL`

`free`

- La funzione `free` ritorna la memoria al sistema quando non è più necessaria
- Per liberare la memoria della chiamata a `malloc` usate

```
free(newPtr);
```

- Dopo impostate il puntatore a NULL
  - Previene che il programma faccia riferimento alla memoria liberata
- Non liberare memoria allocata dinamicamente quando non è più necessaria può esaurire la memoria (**memory leak**)
- Fare riferimento alla memoria che è stata liberata è un errore fatale che normalmente provoca l'arresto del programma

Intervallo

Fonte: chatGPT

Liste collegate

- Una lista collegata è una collezione con organizzazione lineare di oggetti `struct` autoreferenziali chiamati nodi
  - Sono connessi da collegamenti di puntatori
- Si accede a una lista collegata tramite un puntatore al suo primo nodo
- Si accede ai nodi successivi tramite il puntatore di collegamento memorizzato in ogni nodo
- I dati sono memorizzati dinamicamente in una lista collegata creando ogni nodo quando necessario
- Un nodo può contenere dati di ogni tipo
- Le pile e le code sono **strutture lineari** di dati
  - Sono versioni vincolate di liste collegate

Array vs liste collegate

- Le liste di dati possono essere memorizzate in array
- Le liste collegate presentano diversi vantaggi
  - Una lista collegata è appropriata quando il numero di elementi da rappresentare nella struttura di dati non è noto a priori
    - Le liste collegate sono dinamiche, la lunghezza può aumentare o diminuire quando necessario
    - Gli array hanno una dimensione fissa
  - Un array può essere dichiarato con più elementi di quelli attesi, provocando uno **spreco** di memoria
  - L'uso di liste collegate e dell'allocazione dinamica di memoria per strutture che crescono e si riducono durante l'esecuzione può far risparmiare memoria
- Alcuni svantaggi
  - I puntatori ai nodi occupano memoria aggiuntiva
  - L'allocazione dinamica espone al rischio di un sovraccarico di chiamate a funzioni

Array vs liste collegate

- Le liste collegate possono essere mantenute in ordine inserendo ogni nuovo elemento nel punto corretto della lista
- Inserimenti e cancellazioni di dati in un array ordinato richiedono tempo
  - tutti gli elementi che seguono l'elemento inserito o cancellato devono essere opportunamente spostati

Array vs liste collegate

- L'accesso agli elementi di un array è più rapido
- Gli elementi di un array sono registrati in memoria in modo contiguo
- Accesso immediato ad un elemento di un array perché l'indirizzo può essere calcolato in base alla posizione
- Le liste collegate non offrono accesso immediato ai loro elementi

Fonte: Deitel & Deitel

Implementare una lista collegata

- Deitel code example `fig12_01.c`
- Definizione della struttura autoreferenziale `struct listNode`, usata per costruire una lista collegata
- Definizione di `typedef` per rendere il codice più leggibile
- Funzione `main` consente di:
  - Inserire caratteri nella lista (righe 34-39)
  - Cancellare elementi dalla lista (righe 40-58)
  - Terminare il programma.
- `startPtr` inizializzato a NULL per indicare una lista vuota
- Funzioni principali:
  - `insert` per l'inserimento di elementi.
  - `delete` per la cancellazione di elementi.

Funzione `insert`

- La funzione `insert` riceve come argomenti l'indirizzo del puntatore al primo nodo della lista e un carattere da inserire
- Permette ad `insert` di modificare il puntatore al primo nodo della lista della funzione chiamante
- Permette di puntare ad un nuovo nodo quando un elemento viene inserito all'inizio della lista
- Passa il puntatore per riferimento
- Passare l'indirizzo di un puntatore crea un **puntatore a puntatore** (indirezione doppia)

Implementare `insert`

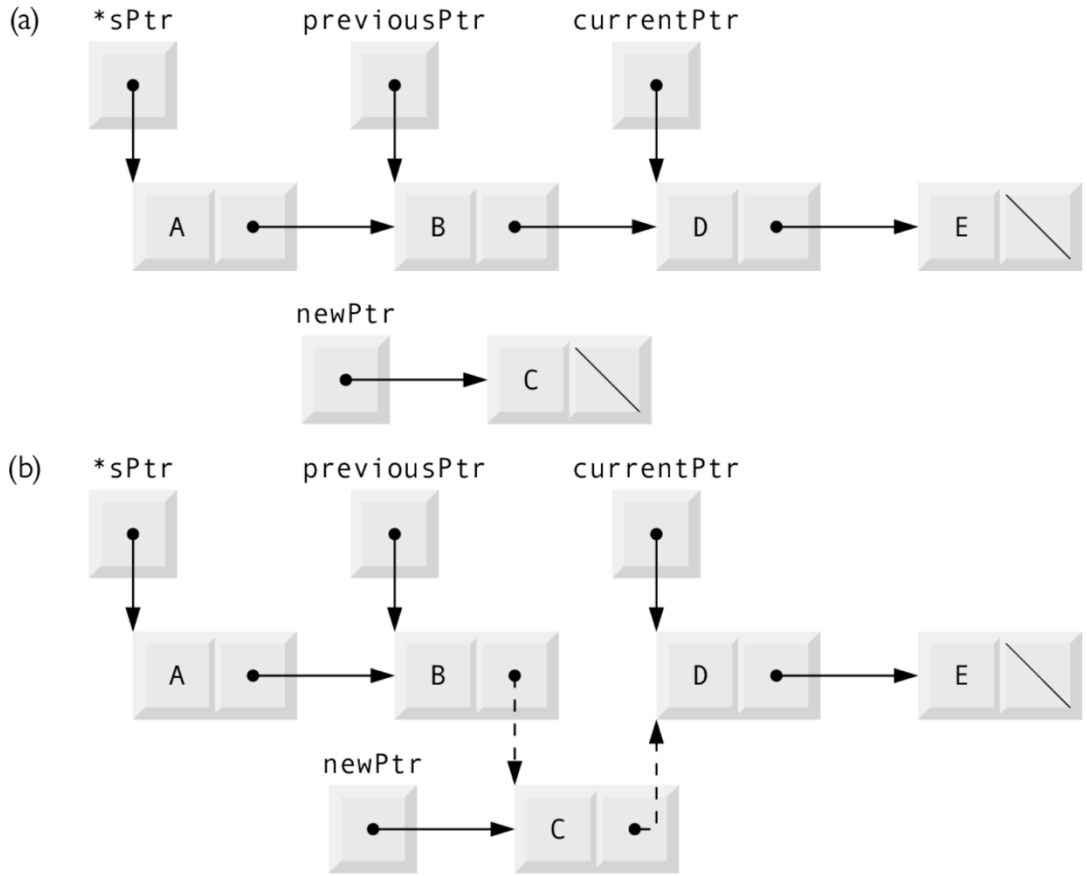
```
68  // inserisci un nuovo valore nella lista ordinata
69  void insert(ListNodePtr *sPtr, char value) {
70     ListNodePtr newPtr = malloc( sizeof (ListNode)); // crea il nodo
70
71     if (newPtr != NULL) { // c'e' spazio disponibile?
72        newPtr->data = value; // inserisci value nel nodo
73        newPtr->nextPtr = NULL; // il nodo non è collegato ad altri nodi
73
74        ListNodePtr previousPtr = NULL;
75        ListNodePtr currentPtr = *sPtr;
75
76        // ripeti il ciclo per trovare la posizione corretta nella lista
77        while  (currentPtr !=  NULL  && value > currentPtr->data) {
78           previousPtr = currentPtr; // va avanti ...
79           currentPtr = currentPtr->nextPtr; // ... al nodo successivo
80        }
80
81        // inserisci il nuovo nodo all'inizio della lista
82        if (previousPtr == NULL) {
83           newPtr->nextPtr = *sPtr;
84           *sPtr = newPtr;
85        }
86        else { // inserisci il nuovo nodo tra previousPtr e currentPtr
87           previousPtr->nextPtr = newPtr;
88           newPtr->nextPtr = currentPtr;
89        }
90     }
91     else {
92        printf("%c not inserted. No memory available.\n", value);
93     }
94  }
```

Fonte: Deitel & Deitel

Meglio un `int` come tipo di ritorno per segnalare se non c'è abbastanza memoria per l'inserimento

Implementare `insert`

- La parte (a) mostra la lista e il nuovo nodo prima dell'inserimento
- La parte (b) mostra il risultato dell'inserimento



Fonte: Deitel & Deitel

Funzione `delete`

- Se il carattere da cancellare corrisponde al carattere nel primo nodo, dobbiamo eliminare il primo nodo
- Adesso `startPtr` in `main` punta al secondo nodo della lista
- Altrimenti si avanza finché non si localizza il carattere da cancellare
- Se i carattere è nella lista, deallocare la memoria puntata da `tempPtr` e restituire il carattere che è stato cancellato

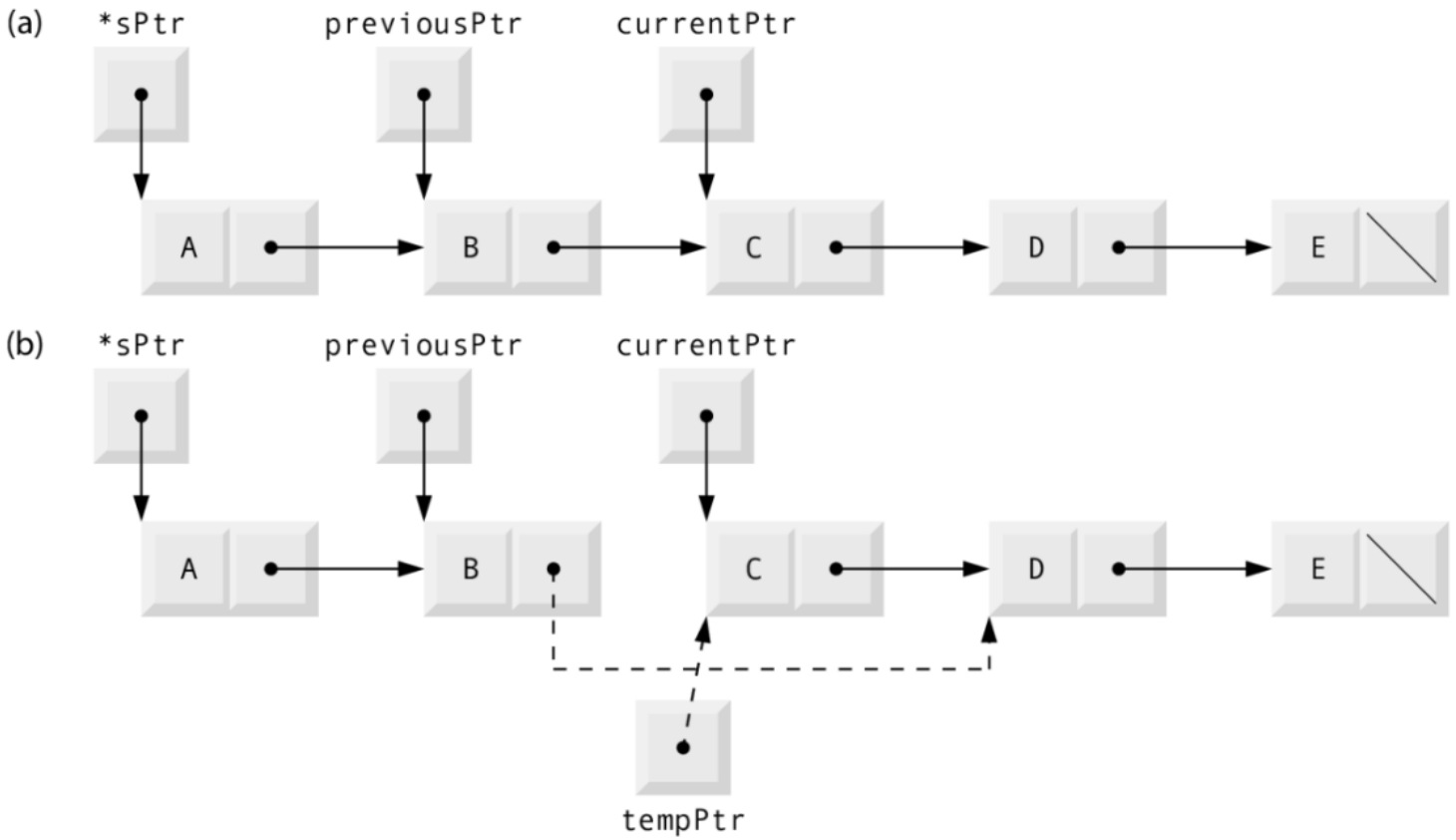
Implementare `delete`

```
112  // elimina un elemento della lista
113  char delete(ListNodePtr *sPtr, char value) {
114     // elimina il primo nodo se viene trovata una corrispondenza
115     if (value == (*sPtr)->data) {
116        ListNodePtr tempPtr = *sPtr; // aggancia il nodo da rimuovere
117        *sPtr = (*sPtr)->nextPtr; // sfila il nodo
118        free(tempPtr); // libera il nodo sfilato
119        return value;
120     }
121     else {
122        ListNodePtr previousPtr = *sPtr;
123        ListNodePtr currentPtr = (*sPtr)->nextPtr;
123
124        // ripeti il ciclo per trovare la posizione corretta nella lista
125        while (currentPtr != NULL && currentPtr->data != value) {
126           previousPtr = currentPtr; // va avanti ...
127           currentPtr = currentPtr->nextPtr; // ... al nodo successivo
128        }
128
129        // elimina il nodo a currentPtr
130        if (currentPtr != NULL) {
131           ListNodePtr tempPtr = currentPtr;
132           previousPtr->nextPtr = currentPtr->nextPtr;
133           free(tempPtr);
134           return value;
135        }
136     }
136
137     return '\0';
138  }
```

Fonte: Deitel & Deitel

Implementare `delete`

- La parte (a) mostra la lista prima dell'eliminazione
- La parte (b) mostra il risultato dell'eliminazione di `c`



Fonte: Deitel & Deitel

Implementare `isEmpty` e `printList`

```
143   // restituisci 1 se la lista e' vuota, altrimenti 0
144   int isEmpty(ListNodePtr sPtr) {
145      return sPtr == NULL;
146   }
146
147   // stampa la lista
148   void printList(ListNodePtr currentPtr) {
149      // se la lista e' vuota
150      if (isEmpty(currentPtr)) {
151         puts("List is empty.\n");
152      }
153      else {
154         puts("The list is:");
154
155         // finche' non si raggiunge la fine della lista
156         while (currentPtr != NULL) {
157            printf("%c --> ", currentPtr->data);
158            currentPtr = currentPtr->nextPtr;
159         }
159
160         puts("NULL\n");
161      }
162   }
```

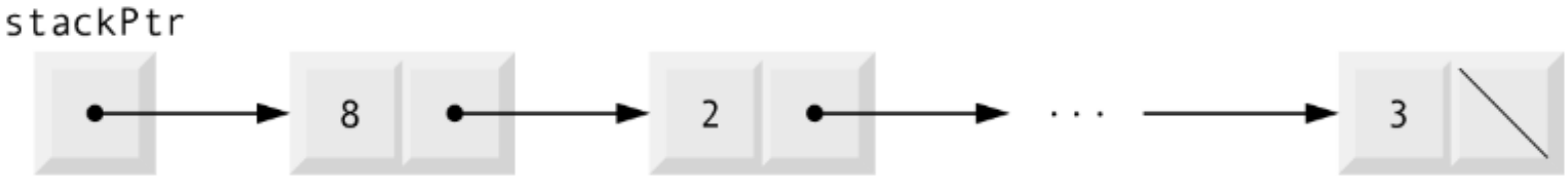
Fonte: Deitel & Deitel

Pile

- Una pila (o **stack**) può essere implementata come versione vincolata di una lista collegata
- Possibile aggiungere nuovi nodi ed eliminare quelli esistenti solo in cima
- Struttura di dati **Last-In, First-Out (LIFO)**
  - Il primo elemento a uscire è l'ultimo entrato
- Si fa riferimento ad una pila tramite un puntatore all'elemento in cima
- Il membro link nell'ultimo nodo della pila è impostato a `NULL`, indicando il fondo della pila
- Non terminare una pila con `NULL` può causare errori in fase di esecuzione

Pile

- `stackPtr` punta all'elemento in cima alla pila
- Lista collegata in cui inserimenti e cancellazioni possono avvenire solo in cima alla pila



Fonte: Deitel & Deitel

Operazioni con pile

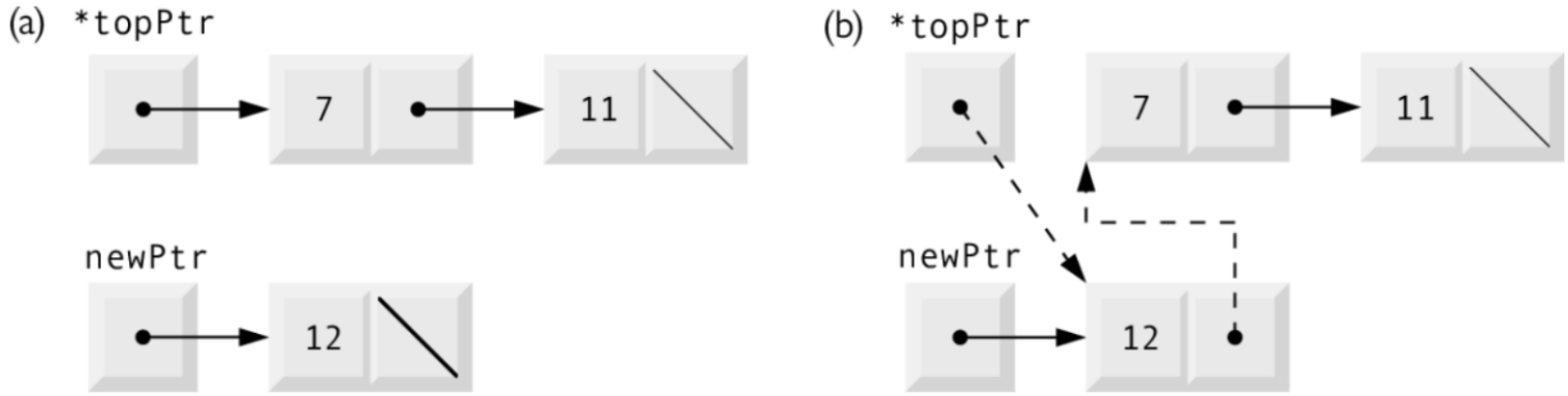
- Le principali funzioni per manipolare una pila sono
  - **`push`**
  - **`pop`**
- `push` crea un nuovo nodo e lo colloca in cima alla pila
- `pop` rimuove il nodo in cima alla pila, libera la memoria occupata dal nodo rimosso, e restituisce il valore del nodo

Implementare una pila

- Deitel code example `fig12_02.c`

Funzione `push`

- La funzione `push` inserisce un nodo in cima alla pila

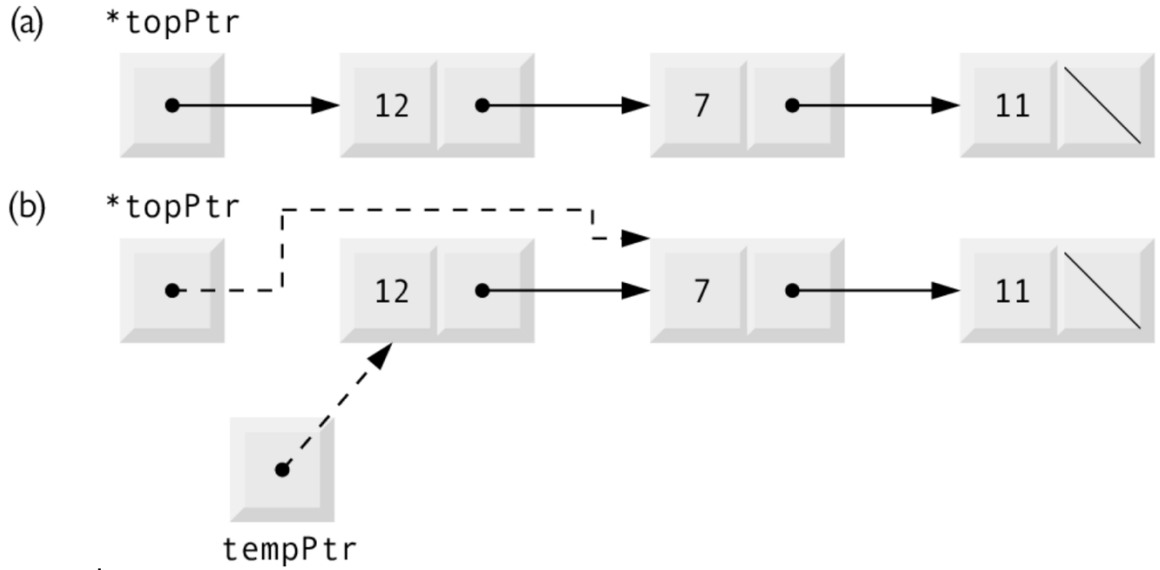


Fonte: Deitel & Deitel

- (a) pila prima dell'inserimento
- (b) pila dopo l'inserimento

Funzione pop

- La funzione `pop` estrae un nodo dalla cima della pila



Fonte: Deitel & Deitel

- (a) pila prima dell'eliminazione
- (b) pila dopo l'eliminazione

Applicazioni delle pile

- Le pile hanno molte applicazioni
- Quando viene effettuata una chiamata a funzione
  - la funzione chiamata dovrà ritornare alla funzione chiamante
  - l'indirizzo di ritorno è inserito con un push in una pila
  - Se si ha una serie di chiamate ordine last-in, first-out
- Le pile mantengono lo spazio creato per le variabili locali in ogni invocazione
  - Quando una funzione ritorna, lo spazio viene eliminato con un pop dalla pila
- Le pile possono essere usate dai compilatori per valutare espressioni e generare codice in linguaggio macchina

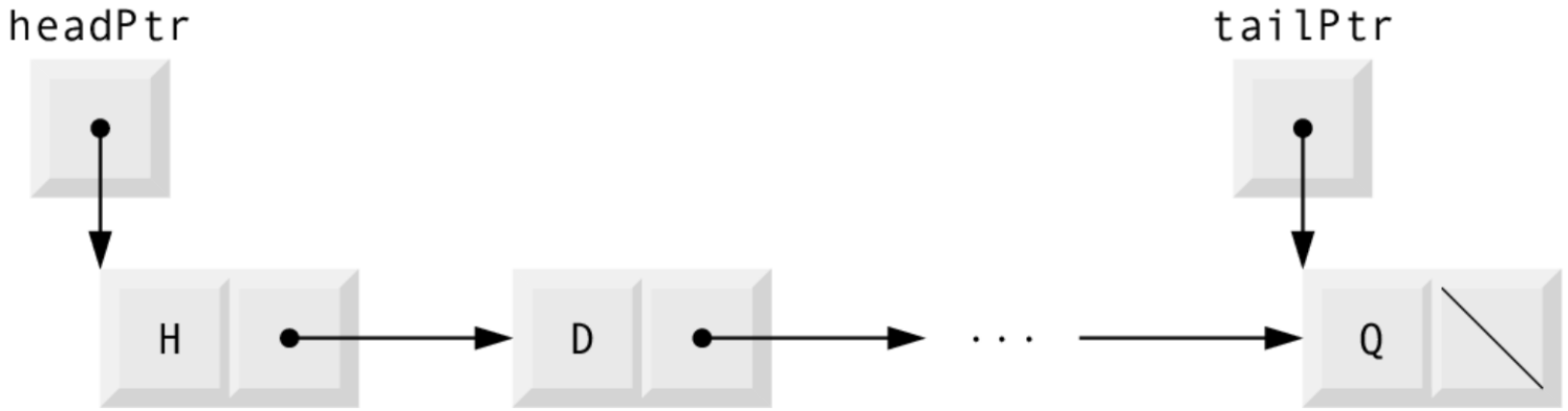
Code

- Una **coda** è simile alla fila davanti alla cassa del supermercato
  - La prima persona in fila è servita per prima
  - I nuovi clienti si aggiungono alla fine della coda
- I nodi della coda possono essere **estratti** solo dalla **testa della coda**
- I nodi possono essere **inseriti** solo alla **fine della coda**
- Struttura **First-In, First-Out (FIFO)**
- Le operazioni di inserimento ed estrazione sono chiamate
  - `enqueue`
  - `dequeue`

Applicazioni delle code

- Nei computer con un solo processore
  - è possibile servire solo un utente alla volta
  - Richieste di altri utenti sono poste in una coda
  - Ogni richiesta avanza verso la testa della coda quando una richiesta che la precede è soddisfatta
- I sistemi multicore odierni
  - I programmi vengono collocati in una coda finché uno dei processori è disponibile
- Le code supportano lo spooling di stampa
  - Un ufficio può avere un'unica stampante
  - Se vengono generati documenti da stampare mentre la stampante è occupata, vengono inseriti in una coda
- I pacchetti che viaggiano su internet aspettano in coda
  - Ogni volta che arriva un pacchetto al nodo di una rete, deve essere instradato al nodo successivo
  - Il nodo inoltra un pacchetto alla volta, mentre gli altri pacchetti sono messi in coda

Esempio di coda



Fonte: Deitel & Deitel

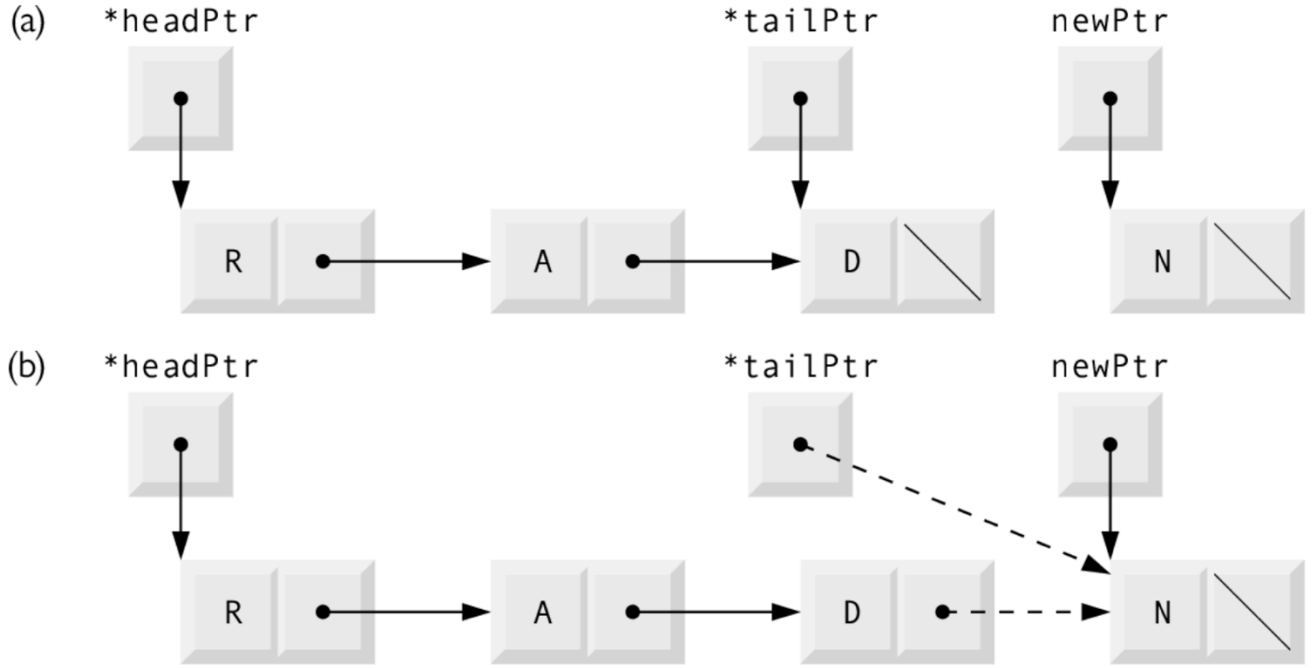
- Puntatore sia al nodo iniziale che finale
- Il collegamento nell'ultimo nodo della coda è impostato a NULL

Implementare una coda

- Deitel code example `fig12_03.c`

Funzione enqueue

- La funzione `enqueue` inserisce un nodo nella coda

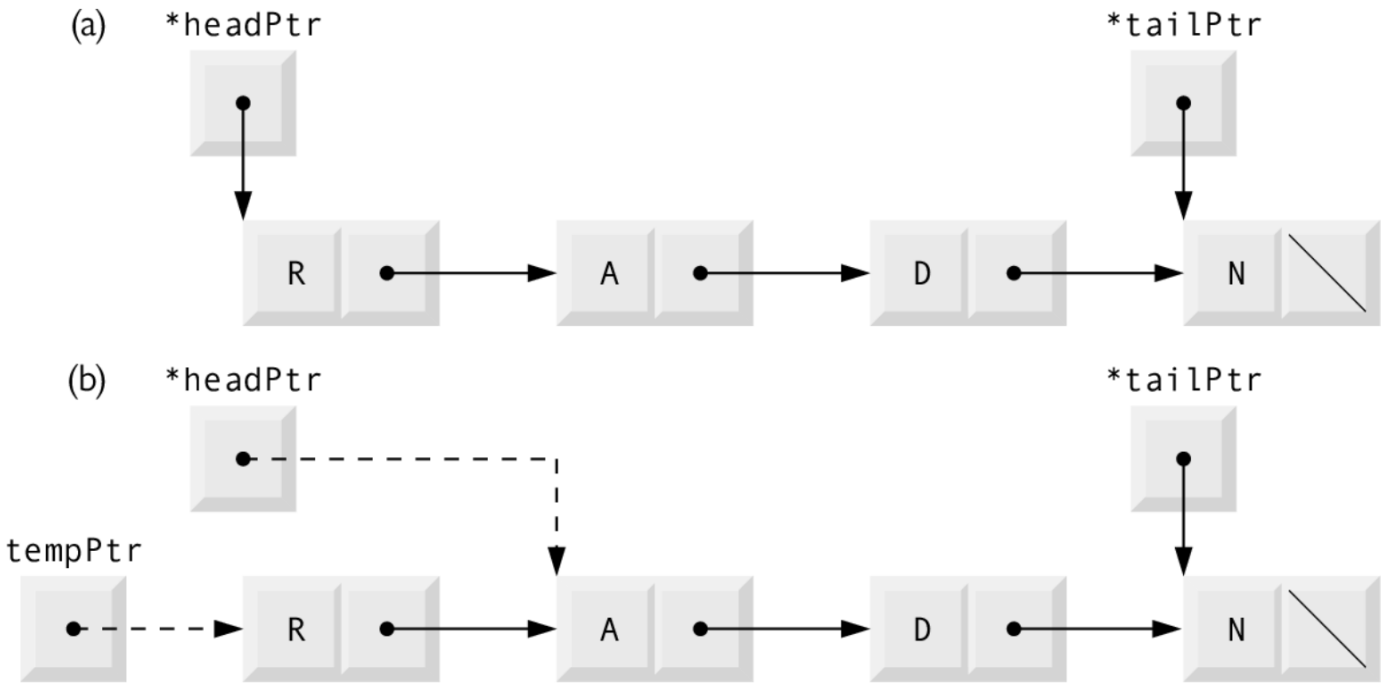


Fonte: Deitel & Deitel

- (a) coda prima dell'inserimento
- (b) coda dopo l'inserimento

Funzione dequeue

- La funzione `dequeue` elimina un nodo nella coda

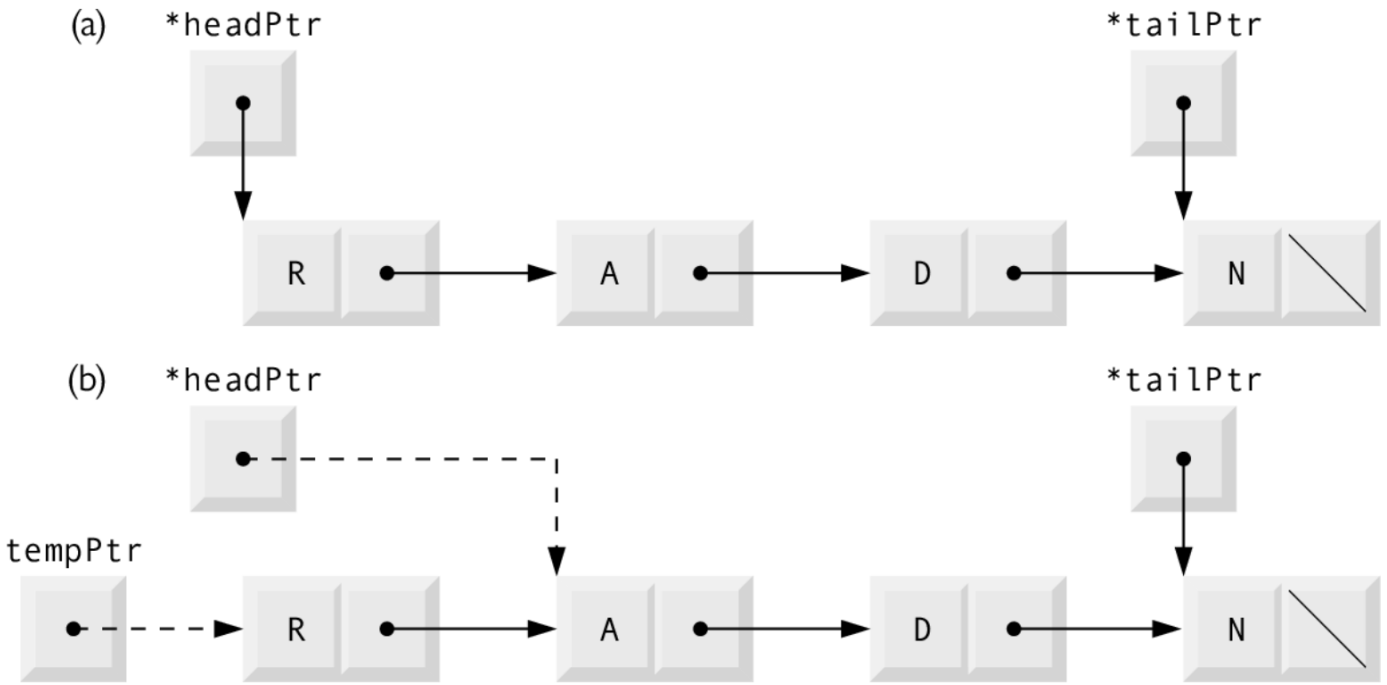


Fonte: Deitel & Deitel

- (a) coda prima dell'eliminazione
- (b) coda dopo l'eliminazione

Funzione dequeue

- La funzione `dequeue` elimina un nodo nella coda



Fonte: Deitel & Deitel

- (a) coda prima dell'eliminazione
- (b) coda dopo l'eliminazione

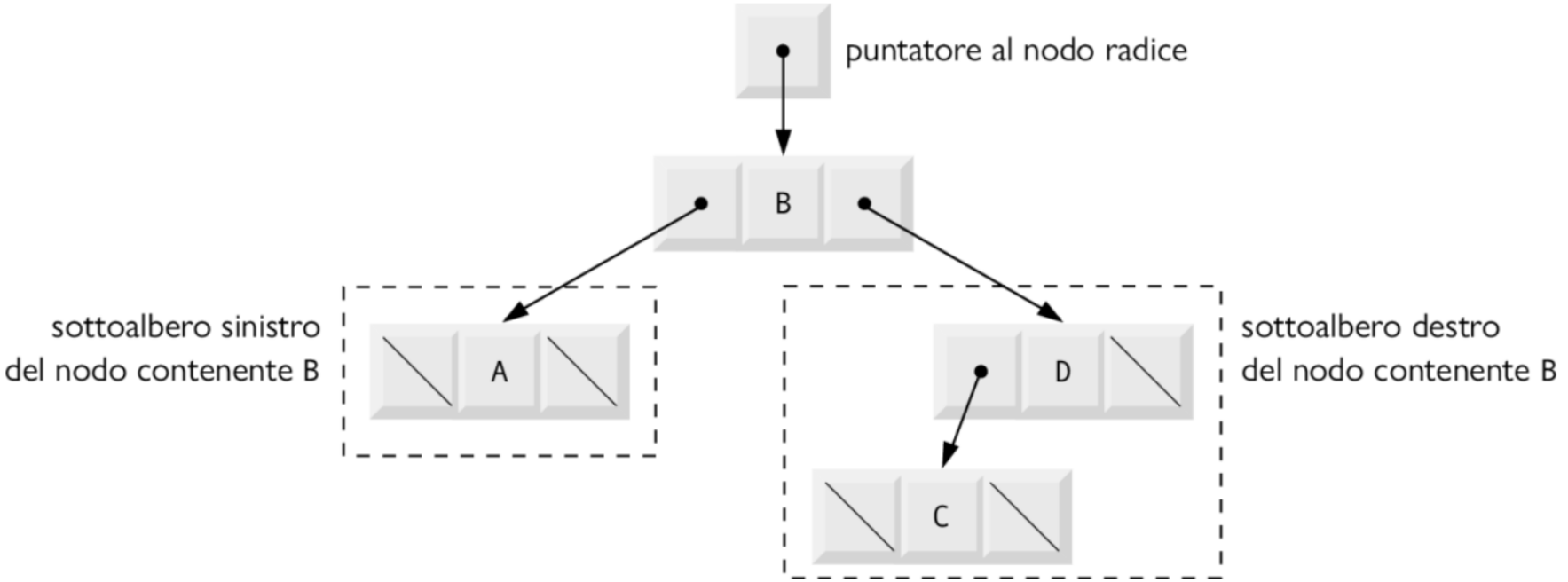
Intervallo

Fonte: chatGPT

Alberi

- Le strutture che abbiamo visto finora sono lineari
  - Liste
  - Pile
  - Code
- I nodi degli **alberi** contengono due o più collegamenti
- Alberi **binari** (binary tree)
  - Il nodo radice è il primo nodo dell'albero (`root`)
  - Ogni collegamento di un nodo si riferisce a un figlio (`child`)
  - Due figli dello stesso nodo sono chiamati fratelli
  - Un nodo senza figli è una foglia (`leaf`)
  - I nodi foglia hanno i collegamenti ai nodi figli impostati a NULL

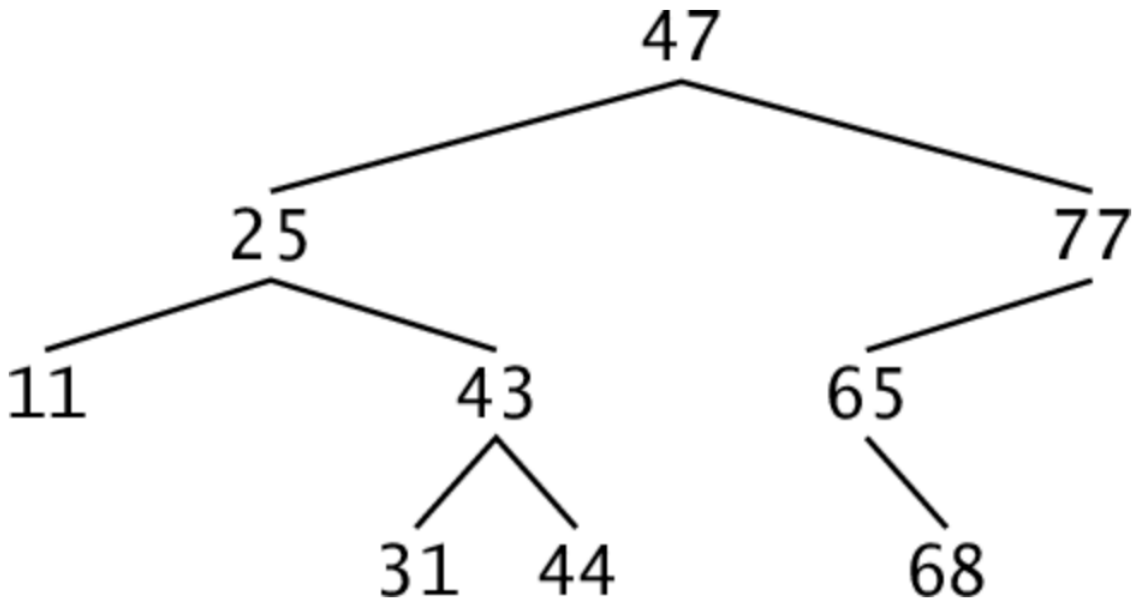
Alberi



Fonte: Deitel & Deitel

Esempio di Albero

- Albero in cui per ogni nodo
  - il valore di ogni nodo nel sottoalbero sinistro è minore
  - Il valore di ogni nodo nel sottoalbero destro è maggiore



Fonte: Deitel & Deitel

Implementare un albero di ricerca binaria

- Il seguente programma
  - crea un albero di ricerca binaria
  - I duplicati non vengono inseriti
  - attraversa i nodi dell'albero in tre modi
    - In ordine
    - In pre-ordine
    - In post-ordine

Implementare un albero di ricerca binaria

```c
// fig12_04.c
// Creazione e attraversamento di un albero binario
// in pre-ordine, in ordine e in post-ordine
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// struttura autoreferenziale
struct  treeNode {
    struct  treeNode *leftPtr; // puntatore al sottoalbero sinistro
     int  data; // valore del nodo
     struct  treeNode *rightPtr; // puntatore al sottoalbero destro
 };

 typedef struct treeNode TreeNode; // sinonimo per struct treeNode
 typedef TreeNode *TreeNodePtr; // sinonimo per TreeNode*

 // prototipi
 void insertNode(TreeNodePtr *treePtr, int value);
 void inOrder(TreeNodePtr treePtr);
 void preOrder(TreeNodePtr treePtr);
 void postOrder(TreeNodePtr treePtr);
```

Fonte: Deitel & Deitel

Implementare un albero di ricerca binaria

```
20   int main(void) {
21      TreeNodePtr rootPtr = NULL; // albero inizialmente vuoto
21
22      srand(time(NULL));
23      puts("The numbers being placed in the tree are:");
23
24      // inserisci nell'albero valori a caso tra 0 e 14
25      for (int i = 1; i <= 10; ++i) {
26         int item = rand() % 15;
27         printf("%3d", item);
28         insertNode(&rootPtr, item);
29      }
29
30      // attraversa l'albero in pre-ordine
31      puts("\n\nThe preOrder traversal is:");
32      preOrder(rootPtr);
32
33      // attraversa l'albero in ordine
34      puts("\n\nThe inOrder traversal is:");
35      inOrder(rootPtr);
35
36      // attraversa l'albero in post-ordine
37      puts("\n\nThe postOrder traversal is:");
38      postOrder(rootPtr);
39   }
```

Fonte: Deitel & Deitel

Implementare un albero di ricerca binaria

```
40   // inserisci un nodo nell'albero
41   void insertNode(TreeNodePtr *treePtr, int value) {
42      if (*treePtr == NULL) { // se l'albero e' vuoto
43         *treePtr = malloc(sizeof(TreeNode));
44
45         if (*treePtr != NULL) { // se la memoria e' stata allocata, assegna i dati
46            (*treePtr)->data = value;
47            (*treePtr)->leftPtr = NULL;
48            (*treePtr)->rightPtr = NULL;
49         }
50         else {
51            printf("%d not inserted. No memory available.\n", value);
52         }
53      }
54      else { // l'albero non e' vuoto
55         if ( value < (*treePtr)->data ) { // il valore va nel sottoalbero sinistro
56            insertNode(&((*treePtr)->leftPtr), value);
57         }
58         else if ( value > (*treePtr)->data ) { // il valore va nel sottoalbero destro
59            insertNode(&((*treePtr)->rightPtr), value);
60         }
61         else { // i valori dei dati duplicati vengono ignorati
62            printf("%s", "dup");
63         }
64      }
   }
```

Fonte: Deitel & Deitel

Implementare un albero di ricerca binaria

```
65   // inizia l'attraversamento in ordine dell'albero
66   void inOrder(TreeNodePtr treePtr) {
67      // se l'albero non e' vuoto, allora attraversalo
68      if  (treePtr !=  NULL ) {
69         inOrder(treePtr->leftPtr);
70         printf( "%3d" , treePtr->data);
71         inOrder(treePtr->rightPtr);
72      }
73   }
```

Fonte: Deitel & Deitel

Implementare un albero di ricerca binaria

```
74   // inizia l'attraversamento in pre-ordine dell'albero
75   void preOrder(TreeNodePtr treePtr) {
76      // se l'albero non e' vuoto, allora attraversalo
77      if  (treePtr !=  NULL ) {
78         printf( "%3d" , treePtr->data);
79         preOrder(treePtr->leftPtr);
80         preOrder(treePtr->rightPtr);
81      }
82   }
```

Fonte: Deitel & Deitel

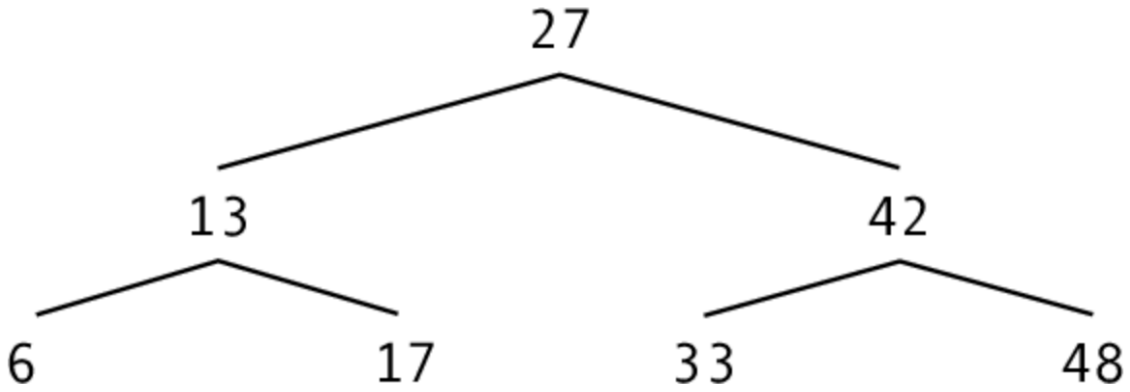
Implementare un albero di ricerca binaria

```
83   // inizia l'attraversamento in post-ordine dell'albero
84   void postOrder(TreeNodePtr treePtr) {
85      // se l'albero non e' vuoto, allora attraversalo
86      if  (treePtr !=  NULL ) {
87         postOrder(treePtr->leftPtr);
88         postOrder(treePtr->rightPtr);
89         printf( "%3d" , treePtr->data);
90      }
91   }
```

Fonte: Deitel & Deitel

Implementare un albero di ricerca binaria

- Albero in input



- Output
  - Ordine: 6 13 17 27 33 42 48
  - Pre-ordine: 27 13 6 17 42 33 48
  - Post-ordine: 6 17 13 33 48 42 27

Fonte: Deitel & Deitel

Ricerca in alberi binari

- La ricerca in un albero binario è veloce
- Se l'albero è «ben organizzato», ogni livello contiene il doppio degli elementi del livello precedente
- Una ricerca in un albero con n elementi avrà al più log n livelli
- Per trovare il nodo (o determinare che non esiste) si effettuano al massimo log n confronti
- In un albero con 1.000.000 di nodi, sono necessari 20 confronti: $2^{20} > 1.000.000$