Laboratorio di Informatica
Algoritmi e Strutture Dati
Note

Roberto Battiti   Marco Tomazzoni [1]

5 maggio 2000

Avvertenza: Le note sono in versione provvisoria: lavori in corso!

[1] Si ringrazia Alessandro Villani i cui appunti sono stati ampiamente saccheggiati.

Indice

Capitolo 1

LISTE, PILE E CODE

1.1 Liste collegate

Una semplice struttura dati è la *lista collegata circolare*, una raffigurazione della quale è mostrata in Fig. 1.1 [1]. Una lista collegata è la rappresentazione di un insieme ordinato dinamico (dinamico = che può aumentare o diminuire i suoi membri).

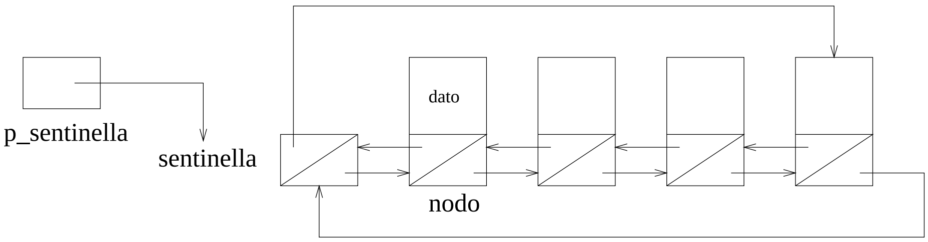


Figura 1.1: Schema di una lista collegata circolare.

Ogni elemento della lista collegata, detto *nodo*, contiene due puntatori, uno al prossimo nodo ed uno al nodo precedente della lista, ed un dato (il dato da caricare nella lista). Spesso il primo nodo della lista è chiamato testa, l'ultimo coda. Un nodo particolare, *sentinella*, contiene due puntatori, uno al primo ed uno all'ultimo nodo della lista, ma non consente di caricare nessun dato. La sentinella sta tra la testa e la coda. La sua utilitá è piú nella chiarezza del codice che nell'incremento delle prestazioni.

Definiamo una classe `Lista` la cui struttura rispecchia quella mostrata in Fig. 1.1 e i cui oggetti rappresentano liste collegate di oggetti

[1] Questo non è l'unico modo per implementare una lista. Vi sono versioni che, ad esempio, non fanno uso del nodo *sentinella*. Esse prevedono che l'ultimo nodo contenga, oltre al puntatore al nodo precedente, il puntatore nullo ed introducono una variabile puntatore, che chiamiamo *testa*, che punta al primo nodo della lista.

di tipo nodo. In Fig. 1.2 è mostrato il file di intestazione (header file) per le classi nodo e Lista.

```
//
// File Lista1.H
// Header File per le liste collegate
//


class nodo {

public:
  nodo* next;       Puntatore al prossimo nodo
  nodo* prev;       Puntatore al nodo precedente
  int data;         Dato contenuto nel nodo
};


class Lista {
  nodo* sentinel;  // Sentinella
  nodo* current;   // Nodo corrente
  int len;         // Lunghezza della lista

public:
  Lista();       // Costruttore
  ~Lista();      // Distruttore

  void Next();     // Sposta il nodo corrente sul nodo successivo
  void Previous(); // Sposta il nodo corrente sul nodo precedente

  void GoToFirst()       // Sposta il nodo corrente sul primo nodo della lista
    { current = sentinel->next; }
  void GoToEndOfList()        // Sposta il nodo corrente oltre l'ultimo nodo
    { current = sentinel; }   // della lista

  int IsEmpty()             // Controllo se la lista e` vuota
    { return (sentinel == sentinel->next); }
  int IsFirst() const       // Controllo se il nodo corrente e` sul primo nodo
    { return (current == sentinel->next); }
  int IsEndOfList() const                // Controllo se il nodo corrente e`
    { return (current == sentinel); }    // oltre l'ultimo nodo

  void Write(int d);         // Cambia il dato del nodo corrente
  int Read();                // Legge il dato del nodo corrente

  void Insert(int d);        // Inserisce un nodo nella posizione corrente
  int Delete();              // Cancella un nodo dalla posizione corrente

  int GetLength()            // Ritorna la lunghezza della lista
    { return len; }
};
```

Figura 1.2: Header file per la classe Lista.

Come abbiamo detto in precedenza, la classe nodo contiene due variabili di istanza next e prev di tipo nodo* (puntatore ad oggetti di tipo nodo) e una variabile data di tipo mytype, il dato che vogliamo caricare nel nodo. La classe Lista è dichiarata friend class in Node Tali variabili di istanza sono dichiarate pubbliche in modo che a tutte le funzioni membro della classe `Lista` sia permesso di accedervi.

La classe `Lista`, come abbiamo già accennato, rappresenta una lista collegata di oggetti di tipo `nodo`. La variabile di istanza più importante è la variabile `sentinel`, puntatore al nodo sentinella. Un'altra variabile di istanza di tipo `nodo*`, chiamata `current`, designa il nodo della lista su cui attualmente si è posizionati.

In Fig. 1.3 è mostrato il file sorgente per la classe `Lista`. Analizziamo in dettaglio le funzioni `Insert(int d)` e `Delete()`:

- la funzione `Insert(int d)` inserisce un nodo contenente il dato `d` immediatamente prima del nodo designato da `current` (in Fig. 1.5 è mostrata una schematizzazione dell'implementazione della funzione). A tale scopo viene creato, con un'istruzione `new`, un oggetto `newnode` di tipo `node` contenente il dato `d`; ai valori dei puntatori `newnode->next` e `newnode->prev` vengono assegnati rispettivamente i valori `current` e `current->prev` (l'indirizzo del nodo che precede `current`). Poi viene aggiornato il valore del `current->prev` (in Fig. 1.5 rappresentato da una freccia tratteggiata) e il valore del puntatore `next` del nodo che precedeva `current` viene posto uguale a `newnode`; dopo tutte queste operazioni di assegnazione ed aggiornamento, il valore di `len` (la lunghezza attuale della lista, ovvero il numero di nodi escluso `sentinel`) viene incrementato di 1;

- la funzione `Delete()` rimuove il nodo designato da `current`, operazione permessa solo nel caso in cui `current` non coincida con `sentinel`, e restituisce il dato contenuto nel nodo rimosso. A tale scopo `current->data` viene salvato in una variabile temporanea `tmp` e `current->next` viene salvato in un puntatore temporaneo `tmpp` in modo che, alla fine, `current` sia il nodo che seguiva il vecchio `current`İn seguito vengono aggiornati i valori di `current->next->prev` e di `current->prev->next`, puntatori dei nodi rispettivamente successivo e precedente a `current` (le frecce tratteggiate) e `current` viene distrutto con un'istruzione `delete`.

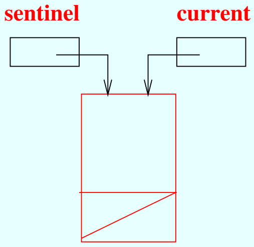
```
//
// File Lista1.C
// File di implementazione della classe Lista
//

#include <iostream.h>
#include "Lista1.H"

Lista::Lista()
{
  sentinel = new nodo;

  sentinel->next = sentinel;
  sentinel->prev = sentinel;
  current = sentinel;
  len = 0;
}
```

Costruttore:
crea una lista vuota




```
Lista::~Lista()
{
  GoToFirst();
  while (!IsEndOfList())
    Delete();
  delete sentinel;
}
```

Distruttore

```
void Lista::Next()
{
  if (IsEndOfList()) {
    cerr << "Errore in Next: sei oltre l'ultimo nodo!"
         << endl;
    exit(-1);
  }
  else
    current = current->next;
}

void Lista::Previous()
{
  if (IsFirst()) {
    cerr << "Errore in Previous: sei sul primo nodo!"
         << endl;
    exit(-1);
  }
  else
    current = current->prev;
}

void Lista::Write(int d)
{
  if (IsEndOfList()) {
    cerr << "Errore in Write: sei oltre l'ultimo nodo!" << endl;
    exit(-1);
  }
  else
    current->data = d;
}
```

Continua...

```
int Lista::Read()
{
  if (IsEndOfList()) {
    cerr << "Errore in Read: sei oltre l'ultimo nodo!" << endl;
    exit(-1);
  }
  else
    return (current->data);
}

void Lista::Insert(int d)
{
  node* newnode = new nodo;

  newnode->next = current;
  newnode->prev = current->prev;
  newnode->data = d;
  current->prev = newnode;
  newnode->prev->next = newnode;
  len++;
}

int Lista::Delete()
{
  if (IsEndOfList()) {
    cerr << "Errore in Delete: sei oltre l'ultimo nodo!" << endl;
    exit(-1);
  }
  int tmp = current->data;
  node* tmpp = current->next;

  current->next->prev = current->prev;
  current->prev->next = current->next;
  delete current;
  len--;
  current = tmpp;
  return tmp;
}
```

Figura 1.3: File di implementazione della classe `Lista`

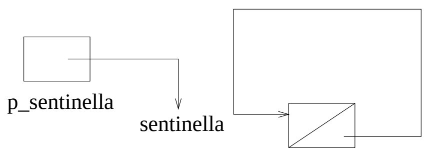


Figura 1.4: Schema di una lista vuota.

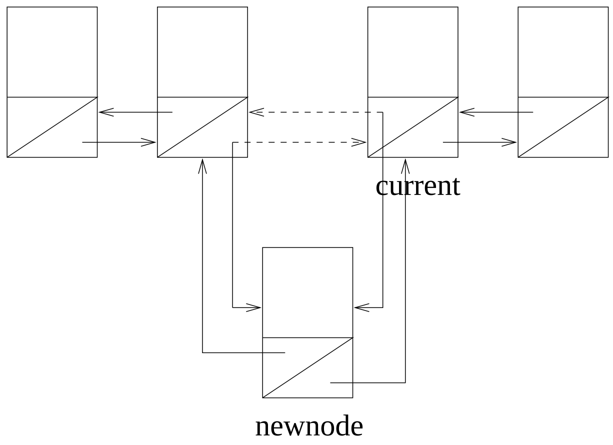


Figura 1.5: La funzione `Insert(int d)`

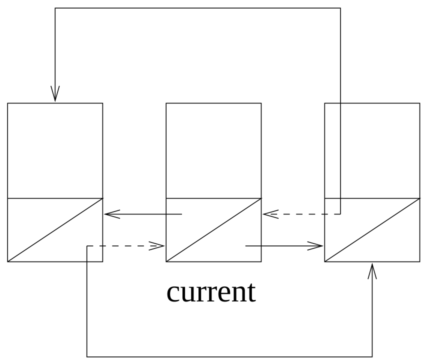


Figura 1.6: La funzione `Remove()`