Algoritmi (modulo di laboratorio)

Corso di Laurea in Matematica

Roberto Cordone

DI - Università degli Studi di Milano

Lezioni:	Martedì 8.30 - 10.30 in aula 8 Mercoledì 10.30 - 13.30 in aula 309
	Giovedì 16.30 - 18.30 in aula 307 Venerdì 10.30 - 12.30 in aula 4
Ricevimento:	su appuntamento (Dipartimento di Informatica)
E-mail:	roberto.cordone@unimi.it
Pagina web:	http://homes.di.unimi.it/~cordone/courses/2024-algo/2024-algo.html
Sito Ariel:	https://mgoldwurmasd.ariel.ctu.unimi.it

Lezione 7: Liste — Milano, A.A. 2023/24

Limitazioni di vettori e tabelle

Un vettore rappresenta una n-upla di elementi con n fissato

Una tabella rappresenta una n-upla di elementi con n variabile ($\leq k$), ma

- l'inserimento di nuovi elementi avviene
  - in posizione terminale in tempo $\Theta(1)$
  - in una posizione desiderata in tempo $\Theta(n)$
    (*si scalano in avanti tutti gli elementi successivi*)
- la cancellazione avviene
  - riordinando gli elementi in tempo $\Theta(1)$
    (*l'ultimo elemento sostituisce quello cancellato*)
  - mantenendo l'ordine originale in tempo $\Theta(n)$
    (*si scalano all'indietro tutti gli elementi successivi*)

Si sente la mancanza di una struttura

- totalmente dinamica
- che consenta inserimenti e cancellazioni efficienti mantenendo l'ordine originale

Liste: struttura dati astratta

Una lista L su un insieme U ha una definizione ricorsiva: può essere

- un insieme vuoto Λ detto lista vuota (caso base) oppure
- una coppia ordinata (a_1, L_1) con $a_1 \in U$ e L_1 lista su U

La definizione non è tautologica perché il caso base arresta la ricorsione, ma il numero degli elementi non è soggetto ad alcun limite

Per accedere agli elementi della lista, anziché gli indici $\{1, \ldots, n\}$, le operazioni di proiezione e sostituzione usano un insieme di posizioni P

Gli insiemi delle posizioni e degli indici sono in corrispondenza biunivoca

$$\{p_1, \ldots, p_n\} \leftrightarrow \{1, \ldots, n\}$$

Indicheremo per comodità una lista L non vuota come $(a_1, \ldots, a_n)$, ma

- in genere le posizioni non coincidono con gli indici
- data la lista L, solo la posizione p_1 è nota (a volte anche p_n)
- da una posizione p_i, si può ricavare solo la posizione seguente p_{i+1} (a volte anche p_{i-1})

Liste: operazioni

Sia $\mathcal{L}$ l'insieme di tutte le possibili liste su U

Le liste ammettono tipicamente le seguenti operazioni

- proiezione: data una lista e una posizione, fornisce l'elemento corrispondente

$$\text{leggelista} : \mathcal{L} \times P \to U$$

- sostituzione: data una lista, una posizione e un elemento inserisce l'elemento nella lista sostituendo quello puntato dalla posizione

$$\text{scrivelista} : \mathcal{L} \times P \times U \to \mathcal{L}$$

- verifica di vuotezza: data una lista, indica se è vuota

$$\text{listavuota} : \mathcal{L} \to \mathbb{B} \qquad \text{(ovvero } \{0,1\}\text{)}$$

- accesso alla testa: data una lista, ne fornisce la prima posizione

$$\text{primolista} : \mathcal{L} \to P$$

E se la lista è vuota?

- accesso alla coda: data una lista, ne fornisce l'ultima posizione

$$\text{ultimolista} : \mathcal{L} \to P$$

E se la lista è vuota?

Liste: operazioni

Sia $\mathcal{L}$ l'insieme di tutte le possibili liste su U

Le liste ammettono tipicamente le seguenti operazioni

- posizione successiva: data una lista e una posizione, fornisce la posizione successiva nella lista

$$\text{succlista} : \mathcal{L} \times P \to P$$

E se la posizione è l'ultima?

- posizione precedente: data una lista e una posizione, fornisce la posizione precedente nella lista

$$\text{predlista} : \mathcal{L} \times P \to P$$

E se la posizione è la prima?

- verifica di appartenenza: data una lista e una posizione, indica se la posizione appartiene o no alla lista

$$\text{finelista} : \mathcal{L} \times P \to \mathbb{B}$$

Liste: operazioni

Sia $\mathcal{L}$ l'insieme di tutte le possibili liste su U

Le liste ammettono tipicamente le seguenti operazioni

- inserimento: data una lista, una posizione e un elemento, aggiunge alla lista l'elemento nella posizione data

$$\text{inslista} : \mathcal{L} \times P \times U \to \mathcal{L}$$

L'elemento che stava nella posizione data e i seguenti scalano avanti

- cancellazione: data una lista, una posizione e un elemento, cancella dalla lista l'elemento nella posizione data

$$\text{canclista} : \mathcal{L} \times P \to \mathcal{L}$$

Gli elementi successivi scalano indietro

Alcune operazioni generano o ricevono posizioni non appartenenti a L

- la testa o la coda di una lista vuota
- la posizione successiva all'ultima o precedente alla prima

Indichiamo le posizioni esterne alla lista col simbolo aggiuntivo $\perp$

Liste: operazioni

In matematica basta definire un oggetto per crearlo

Nelle implementazioni concrete, però questo non sempre vale: potrebbe occorrere qualche inizializzazione o allocazione dinamica

Per motivi tecnici, quindi è opportuno definire anche

- creazione: crea una lista vuota

$$\text{crealista} : () \to \mathcal{L}$$

- distruzione: distrugge una lista (rendendola vuota)

$$\text{distruggelista} : \mathcal{L} \to ()$$

Liste: implementazione con puntatori

L'idea base è di rappresentare le posizioni con indirizzi di memoria

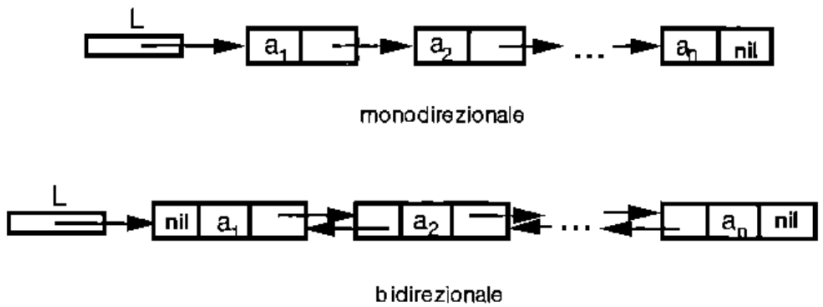
- l'intera lista corrisponde allora a
  - posizione del primo elemento
  - eventualmente, posizione dell'ultimo elemento
- ogni elemento a_i della lista corrisponde a una struttura con
  - il dato a_i
  - la posizione successiva p_{i+1} ($\perp$ se a_i è in coda)
  - eventualmente, la posizione precedente p_{i-1} ($\perp$ se a_i è in cima)

```
#define NO_ELEMENT NULL                    (posizione esterna alla lista)

typedef elemento *lista;                   (la lista è l'indirizzo della testa)
typedef elemento *posizione; (la posizione dell'elemento è il suo indirizzo)

typedef struct _elemento elemento;
struct _elemento {
   U a;                                    (U è il tipo dell'elemento generico)
   posizione succ;
   posizione pred;                         (questo campo può mancare)
};
```

Varianti delle liste a puntatori

Vi sono tre aspetti implementativi fondamentali nelle liste a puntatori

1. direzionalità, cioè la presenza o assenza della posizione precedente
   - le liste monodirezionali hanno solo la posizione successiva
   - le liste bidirezionali hanno la posizione successiva e la precedente

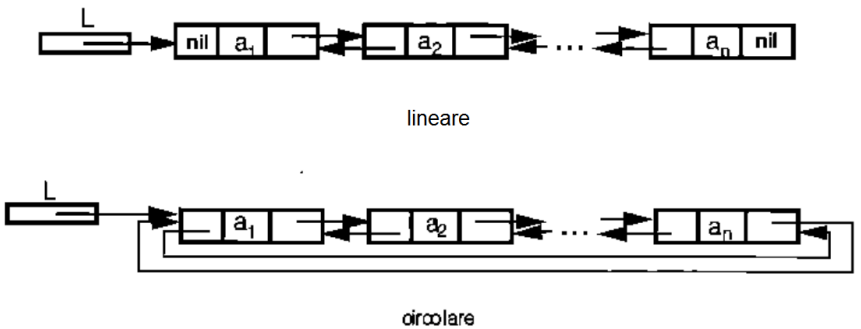


Si tratta di efficienza spaziale contro efficienza temporale

Varianti delle liste a puntatori

Vi sono tre aspetti implementativi fondamentali nelle liste a puntatori

2. linearità o circolarità
   - le liste lineari terminano nella posizione esterna $\perp$
   - le liste circolari dopo l'ultimo elemento tornano al primo

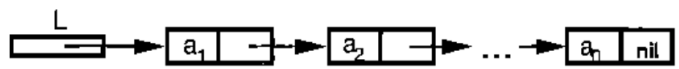


Si può scorrere ripetutamente la lista, ma come arrestarsi?

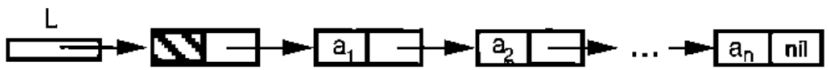
Varianti delle liste a puntatori

Vi sono tre aspetti implementativi fondamentali nelle liste a puntatori

3. presenza o assenza di una sentinella, cioè un elemento fittizio
   - le liste senza sentinella occupano meno spazio
   - le liste con sentinella semplificano alcune operazioni

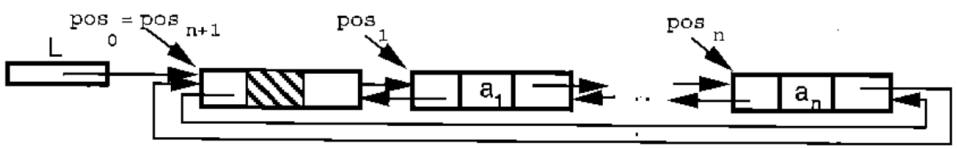


monodirezionale



monodirezionale con sentinella

I tre aspetti sono indipendenti e danno luogo a $2^3 = 8$ combinazioni

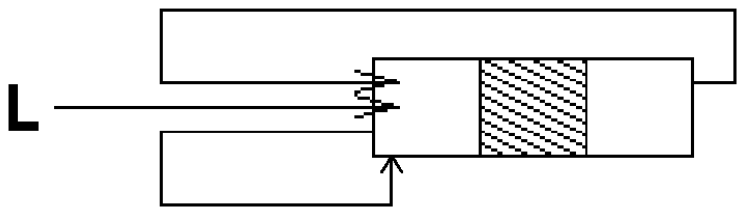


bidirezionale circolare con sentinella

Alcune operazioni fondamentali

Consideriamo l'implementazione bidirezionale, circolare con sentinella (come nel codice)

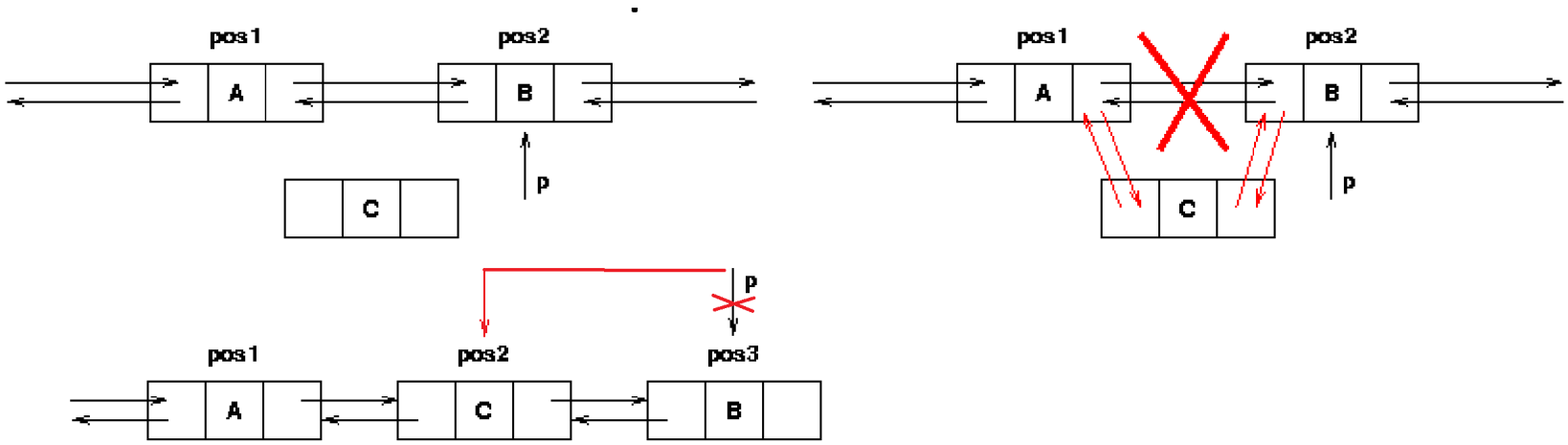
`crealista/listavuota`



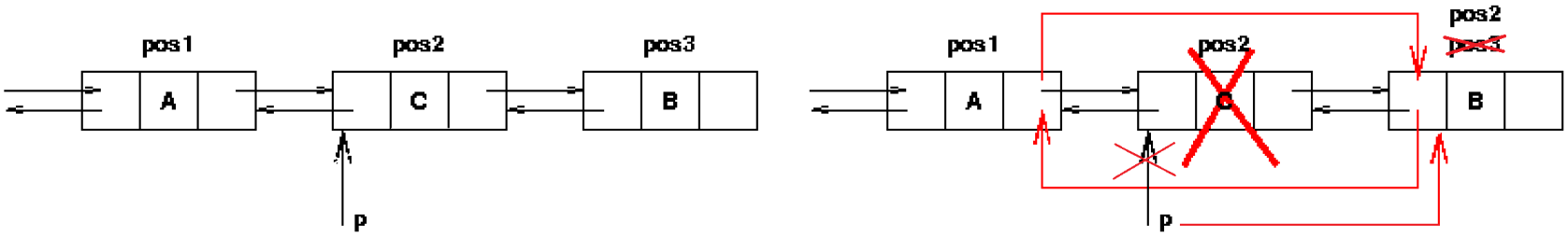
Alcune operazioni fondamentali

Consideriamo l'implementazione bidirezionale, circolare con sentinella (come nel codice)

`inslista`



`canclista`



Liste: implementazione con vettori e cursori

L'idea base è di rappresentare le posizioni con indici in un vettore

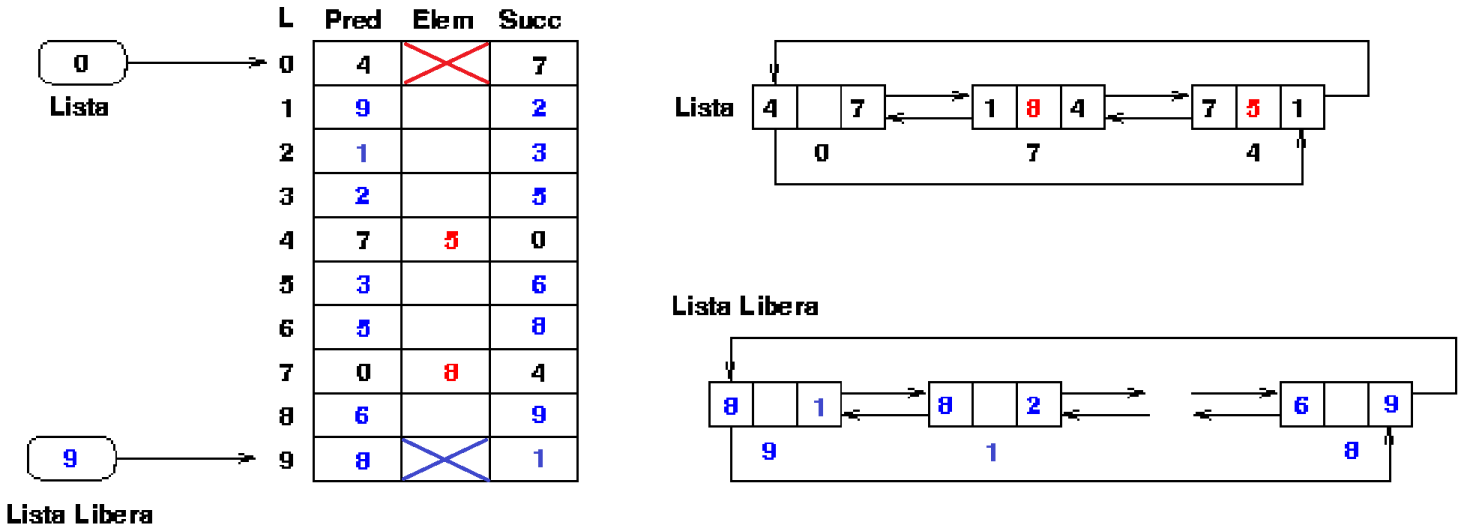
- l'intera lista corrisponde allora a
  - un vettore (cioè un puntatore alla prima cella)
- ogni elemento a_i della lista corrisponde a una struttura con
  - il dato a_i
  - l'indice successivo p_{i+1} ($\perp$ se a_i è in coda)
  - eventualmente, l'indice precedente p_{i-1} ($\perp$ se a_i è in cima)
- gli elementi inutilizzati del vettore stanno in una "lista libera"

Non occorrono allocazioni e deallocazioni

Ovviamente, si perde la totale dinamicità

Liste: implementazione con vettori e cursori

Consideriamo ancora una lista bidirezionale circolare con sentinella



La lista ha sentinella nell'elemento di indice 0 (*pura convenzione*)

Gli elementi non usati nella lista formano una seconda lista (lista libera) con sentinella nell'elemento di indice 9 (*altra pura convenzione*)

Inserimenti e cancellazioni sono spostamenti fra le due liste

In linea di principio, il vettore può ospitare diverse liste

- rigorosamente disgiunte
- ognuna con la sua sentinella

Liste: implementazione con vettori e cursori

```
#define NO_ELEMENT -1                          (posizione esterna alla lista)
#define NO_LIST NULL

#define LIST_SIZE 8                            (lunghezza del vettore)
#define SENTINELLA 0                           (indice della sentinella)
#define SENTINELLA_LIBERA LIST_SIZE+1          (indice della sentinella
                                                per la lista libera)

typedef elemento *lista;                       (la lista è l'indirizzo del vettore)
typedef int posizione;                         (la posizione dell'elemento è il suo indice)

typedef struct _elemento elemento;
struct _elemento {
   U a;                                        (U è il tipo dell'elemento generico)
   posizione succ;
   posizione pred;                             (questo campo può mancare)
};
```