1.4 Esercizi

Esercizio 1.1 (Specifica numeri complessi). Si fornisca una specifica del tipo di dato numero complesso, che includa i seguenti operatori: parte reale, parte immaginaria, argomento, modulo, costruzione a partire dalla rappresentazione cartesiana, costruzione a partire dalla rappresentazione trigonometrica, addizione, sottrazione, moltiplicazione, coniugato, divisione.

Esercizio 1.2 (Realizzazione numeri complessi). Si fornisca una realizzazione in Pascal del tipo di dato numero complesso specificato nell'Esercizio 1.1, utilizzando record e puntatori.

Esercizio 1.3 (Memorizzazione matrice). Una matrice w a tre dimensioni può essere memorizzata "per righe" in un vettore v, facendo variare più in fretta il terzo indice, indi il secondo, ed infine il primo. Si riscriva la formula che dà la corrispondenza tra $w[i, j, h]$ e $v[p]$.

Esercizio 1.4 (Memorizzazione matrice). Si riscriva la formula che dà la corrispondenza tra un generico elemento $w[i_1, i_2, \ldots, i_k]$ di una matrice a k dimensioni e $v[p]$, assumendo di memorizzare w nel vettore v "per colonne" e che ciascun indice i_j possa variare nell'intervallo min_j e max_j, per $1 \leq j \leq k$.

Esercizio 1.5 (Memoria associativa). Una memoria associativa (o mapping) è una corrispondenza $M : D \rightarrow C$ tra elementi del dominio D e del codominio C, in cui sono ammesse le operazioni:

CREA: $() \rightarrow$ mapping
ASSEGNA: (mapping, dominio, codominio) $\rightarrow$ mapping
CALCOLA: (mapping, dominio) $\rightarrow$ codominio $\times$ booleano

dove:

CREA(M) = Λ (Λ indica mapping vuoto con $M(d)$ *indefinito* per ogni d);
ASSEGNA(M, d, c) = M', con $M'(d) = c$ ed $M'(x) = M(x)$ per $x \neq d$;
CALCOLA(M, d) = (c, b), con $b = falso$, se $M(d)$ è indefinito,
$c = M(d)$ e $b = vero$, altrimenti.

Si fornisca una realizzazione per il tipo di dato mapping che fa uso di un vettore, e si discuta la complessità di ciascuna operazione, assumendo che il dominio D sia un insieme finito.

Capitolo 2

Liste

Una lista è una sequenza di elementi di un certo tipo, in cui è possibile aggiungere o togliere elementi. Per far questo, occorre specificare la posizione relativa all'interno della sequenza nella quale il nuovo elemento va aggiunto o dalla quale il vecchio elemento va tolto. A differenza del vettore, che è una struttura a dimensione fissa dove è possibile accedere direttamente ad ogni elemento specificandone l'indice, la lista è a dimensione variabile e si può accedere direttamente solo ad un ristretto sottoinsieme di elementi (di solito il primo o l'ultimo). Per accedere ad un generico elemento, occorre scandire sequenzialmente gli elementi della lista: partendo da un elemento accessibile direttamente, ci si sposta via via da un elemento ad uno adiacente nella sequenza, fino a raggiungere l'elemento desiderato.

Esempio 2.1 (Liste). Uno schedario contenente le informazioni degli impiegati di un'azienda, in cui è possibile ricercare, aggiungere e togliere schede, è un esempio di lista. □

Sia $L = a_1, a_2, \ldots, a_n$ una lista. La lunghezza di L è pari ad n, il numero dei suoi elementi. Se la lista è vuota, cioè non contiene alcun elemento, allora $n = 0$, e la lista è indicata con Λ. Ogni elemento è caratterizzato da una posizione in L, indicata con pos_i, e da un valore a_i. Anche se viene immediato immaginare che pos_i sia uguale ad i, si considera la posizione come un tipo di dato a sé stante la cui realizzazione, che varia a seconda della realizzazione adottata per il tipo di dato lista, può non essere uguale all'intero i. Per comodità, si assume inoltre l'esistenza di una posizione pos_0 che precede quella del primo elemento e di una posizione pos_{n+1} che segue quella dell'ultimo elemento.

2.1 Specifica

Un tipico insieme di operazioni per il tipo di dato lista comprende l'operazione di creazione CREALISTA, per inizializzare una lista ad un particolare valore (di solito la sequenza vuota), quelle di selezione PRIMOLISTA ed ULTIMOLISTA, per accedere direttamente al primo o all'ultimo elemento della sequenza, e SUCCLISTA e PREDLISTA, per passare da un elemento al successivo o al precedente nella sequenza, quelle di interrogazione LISTAVUOTA e FINELISTA, per verificare che la sequenza sia vuota o che sia stato oltrepassato un estremo della lista, quella di lettura LEGGILISTA per leggere il valore di un elemento, e quelle di modifica SCRIVILISTA per cambiare il valore di un elemento, INSLISTA per inserire un nuovo elemento nella lista, e CANCLISTA per cancellare un vecchio elemento della lista.

Formalmente, la sintassi degli operatori è la seguente.

CREALISTA: () → lista
LISTAVUOTA: (lista) → booleano
PRIMOLISTA: (lista) → posizione
ULTIMOLISTA: (lista) → posizione
SUCCLISTA: (posizione, lista) → posizione
PREDLISTA: (posizione, lista) → posizione
FINELISTA: (posizione, lista) → booleano
LEGGILISTA: (posizione, lista) → tipoelem
SCRIVILISTA: (tipoelem, posizione, lista) → lista
INSLISTA: (tipoelem, posizione, lista) → lista
CANCLISTA: (posizione, lista) → lista

Indicando col termine lista l'insieme di tutte le sequenze di lunghezza arbitraria di elementi di tipo "tipoelem" e con L una generica lista, la semantica degli operatori è data dalle seguenti funzioni, con relative precondizioni e postcondizioni.

CREALISTA() = L'
Post: $L' = \Lambda$, la sequenza vuota

LISTAVUOTA(L) = b
Post: b = vero, se $L = \Lambda$; b = falso, altrimenti

PRIMOLISTA(L) = p
Post: $p = pos_1$ (e quindi $pos_1 = pos_{n+1}$ se $L = \Lambda$)

ULTIMOLISTA(L) = p
Post: $p = pos_n$ (e quindi $pos_n = pos_0$ se $L = \Lambda$)

SUCCLISTA(p, L) = q
Pre: $L = a_1, a_2, \ldots, a_n$ e $p = pos_i$ per un i, $1 \leq i \leq n$
Post: $q = pos_{i+1}$

PREDLISTA(p, L) = q
Pre: $L = a_1, a_2, \ldots, a_n$ e $p = pos_i$ per un i, $1 \leq i \leq n$
Post: $q = pos_{i-1}$

FINELISTA(p, L) = b
Pre: $L = a_1, a_2, \ldots, a_n$ e $p = pos_i$ per un i, $0 \leq i \leq n+1$
Post: b = vero, se $p = pos_0$ oppure se $p = pos_{n+1}$; b = falso, altrimenti

LEGGILISTA(p, L) = a
Pre: $L = a_1, a_2, \ldots, a_n$ e $p = pos_i$ per un i, $1 \leq i \leq n$
Post: $a = a_i$

SCRIVILISTA(a, p, L) = L'
Pre: $L = a_1, a_2, \ldots, a_n$ e $p = pos_i$ per un i, $1 \leq i \leq n$
Post: $L' = a_1, \ldots, a_{i-1}, a, a_{i+1}, \ldots, a_n$

INSLISTA(a, p, L) = L'
Pre: $L = a_1, a_2, \ldots, a_n$ e $p = pos_i$ per un i, $1 \leq i \leq n+1$
Post: $L' = a_1, \ldots, a_{i-1}, a, a_i, a_{i+1}, \ldots, a_n$ se $1 \leq i \leq n$,
$L' = a_1, \ldots, a_n, a$, se $i = n+1$ (e quindi $L' = a$, se $i = 1$ e $L = \Lambda$)

CANCLISTA(p, L) = L'
Pre: $L = a_1, a_2, \ldots, a_n$ e $p = pos_i$ per un i, $1 \leq i \leq n$
Post: $L' = a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$ (e quindi $L' = \Lambda$ se $i = n = 1$)

Dalla specifica emerge che per accedere ad un elemento occorre conoscerne la posizione. Ciò è possibile solo conoscendo a priori la posizione dell'elemento precedente (o seguente) ed applicando quindi l'operazione SUCCLISTA (o PREDLISTA). Le uniche operazioni che danno per risultato una posizione, senza che gliene venga fornita una in ingresso, sono le operazioni PRIMOLISTA e ULTIMOLISTA. Per accedere ad un generico elemento occorre pertanto scandire la lista in avanti a partire dal primo elemento, o a ritroso partendo dall'ultimo. Se viene oltrepassato un estremo della lista, SUCCLISTA e PREDLISTA restituiscono, rispettivamente, pos_{n+1} e pos_0. È possibile accorgersi di questo fatto con l'interrogazione FINELISTA, che in tal caso restituirà il valore booleano vero.

Poiché inserzioni e cancellazioni "allungano" ed "accorciano" la lista, esse provocano la modifica di tutte le posizioni degli elementi che seguono quello inserito o cancellato. In particolare, INSLISTA(a, p, L) con $p = pos_i$ inserisce il nuovo elemento di valore a come nuovo i-esimo elemento della sequenza, mentre il vecchio i-esimo elemento diventa l'$(i+1)$-esimo, il vecchio $(i+1)$-esimo diventa l'$(i+2)$-esimo, ecc. Se $p = pos_{n+1}$, INSLISTA inserisce il nuovo elemento in fondo alla sequenza, lasciando inalterate le posizioni dei vecchi elementi. Se la lista è vuota e $p = pos_1 = pos_{n+1}$, il nuovo elemento diviene il primo (ed unico) elemento della lista. Invece, CANCLISTA(p, L) con $p = pos_i$ cancella il vecchio i-esimo elemento della sequenza, mentre il vecchio $(i+1)$-esimo elemento diventa l'i-esimo, il vecchio $(i+2)$-esimo diventa l'$(i+1)$-esimo, ecc.

Si noti infine che l'operazione LISTAVUOTA è ridondante, poiché LISTAVUOTA(L) = FINELISTA(PRIMOLISTA(L), L) = FINELISTA(ULTIMOLISTA(L), L).

Esempio 2.2 (Il catalogo di Don Giovanni). Il famoso catalogo delle conquiste di Don Giovanni comprende italiane, tedesche, spagnole, francesi e turche. Tra queste ci sono contadine, cameriere, cittadine, duchesse, baronesse, marchesine e principesse. Il fedele servitore Leporello ha strutturato il catalogo del padrone con una lista ordinata per data di conquista, vorrebbe cancellare dal catalogo tutte le duchesse e copiare in un'altra lista L tutte le spagnole, mantenendone lo stesso ordine che avevano nel catalogo.

```
type tiponazione = (italiana, tedesca, spagnola, francese, turca);
    tiporango = (contadina, cameriera, citttadina, duchessa, baronessa,
        marchesa, principessa);
    conquista = record
        data: tipodata;
        nome: tiponome;
        rango: tiporango;
        nazionalità: tiponazione
    end;

procedure DONGIOVANNI (var catalogo, L: lista);
    var p, q: posizione; temp: conquista;
    begin
        CREALISTA(L);
        q := PRIMOLISTA(catalogo); p := PRIMOLISTA(L);
        while not FINELISTA(q, catalogo) do begin
            temp := LEGGILISTA(q, catalogo);
            if temp.nazionalità = spagnola then
                begin
                    INSLISTA(temp, p, L);
```

```
            p := SUCCLISTA(p, L)
        end;
    if temp.rango = duchessa
        then CANCLISTA(q, catalogo)
        else q := SUCCLISTA(q, catalogo)
  end
end
```

Nella procedura DONGIOVANNI sono utilizzate due posizioni p e q per scandire, rispettivamente, L e *catalogo*. L è inizializzata alla lista vuota, q alla prima posizione del catalogo, e p alla prima posizione di L. Poiché all'inizio L è vuota, $p = pos_1 = pos_{|L|+1}$, la posizione che segue l'ultimo elemento. All'interno del ciclo **while**, la posizione p viene mantenuta sempre uguale a quella che segue l'ultimo elemento di L, in modo da inserire un nuovo elemento in fondo alla sequenza. In questo modo, gli elementi inseriti in L mantengono lo stesso ordine che avevano nel catalogo. Nel ciclo **while**, *prima* è verificata la nazionalità e *poi* è controllato il rango. Così, se una conquista è al tempo stesso sia duchessa che spagnola, viene prima inserita nella lista L e poi cancellata dal catalogo. In accordo alla specifica del tipo di dato lista, la cancellazione di una duchessa dal catalogo non richiede di fare avanzare la posizione q, poiché q assume per definizione la posizione dell'elemento che seguiva quello cancellato. □

Per valutare la complessità di una procedura (come per esempio la DONGIOVANNI) che utilizzi le operazioni appena definite, occorre prima conoscere la complessità di ogni operazione impiegata. In altri termini, occorre conoscere la realizzazione della struttura di dati di tipo "lista".

La prima realizzazione che viene in mente è quella di memorizzare gli elementi della lista in un vettore. In tal caso, la posizione di un elemento è un indice del vettore e pos_i è proprio uguale ad i, per $1 \leq i \leq n$. In altri termini, alla contiguità tra elementi della lista corrisponde una contiguità di memorizzazione. Questa realizzazione permette agevolmente, cioè in tempo O(1), di passare da un elemento al successivo o al precedente, di accorgersi se si tenta di superare un estremo della lista, di leggere o modificare il valore di un elemento. Purtroppo presenta due seri svantaggi. Innanzitutto, il vettore è una struttura a dimensione fissa, mentre la lista è a dimensione variabile. Il numero di elementi della lista non può quindi superare la dimensione del vettore, che è fissata una volta per tutte. Inoltre, l'inserzione di un nuovo elemento o la cancellazione di un vecchio elemento richiedono un inaccettabile tempo $O(n)$, poiché è necessario scalare di una posizione a destra, in caso di inserzione, o a sinistra, in caso di cancellazione, tutti gli elementi che seguono quello inserito o cancellato. Pertanto, non è conveniente realizzare una lista con un vettore.

Vediamo due possibili realizzazioni diverse, che sono basate sull'uso di puntatori e di cursori, dove alla contiguità tra elementi della lista non corrisponde necessariamente una contiguità di memorizzazione. Queste realizzazioni permettono sia di ottenere complessità O(1) per tutte le operazioni, sia di non limitare il massimo numero di elementi presenti nella lista.

2.2 Realizzazione con puntatori

Come mostrato nella Fig. 2.1, ci sono vari modi possibili di realizzare le liste mediante puntatori. L'idea di base è di memorizzare una lista di n elementi in n record, usualmente detti celle, tali che l'i-esima cella contenga il valore dell'i-esimo

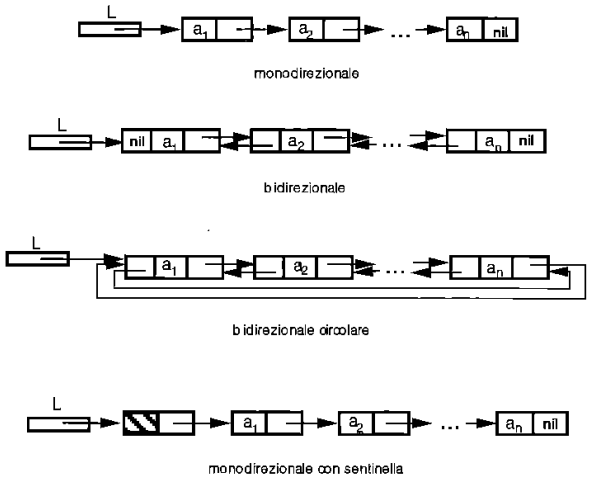

Figura 2.1: Alcune realizzazioni di una lista con puntatori.

elemento della lista e l'indirizzo (puntatore) della cella contenente l'elemento successivo (realizzazione monodirezionale) o gli indirizzi sia della cella successiva che precedente (realizzazione bidirezionale). La prima cella è indirizzata da una variabile L di tipo puntatore, mentre l'ultima cella contiene il valore convenzionale **nil** come indirizzo della cella successiva (ovviamente, nel caso bidirezionale, la prima cella contiene **nil** come indirizzo della cella precedente). Ricordiamo che gli indirizzi delle celle sono noti soltanto alla macchina, ma non lo sono al programmatore.

Entrambe le realizzazioni possono essere rese circolari, facendo sì che l'ultima cella contenga l'indirizzo della prima cella e, nel caso bidirezionale, che la prima cella contenga l'indirizzo dell'ultima. Per rendere più leggibile il codice Pascal delle operazioni INSLISTA e CANCLISTA quando operano sugli elementi estremi della lista, può essere conveniente introdurre una cella in più, detta sentinella, che è direttamente indirizzata da L, ma non contiene il valore di alcun elemento. Si hanno così otto realizzazioni possibili, a seconda che ci siano o no bidirezionalità, circolarità, e sentinella. La Fig. 2.1 illustra quattro di queste otto possibilità. Tutte le otto realizzazioni richiedono $\Theta(n)$ spazio di memoria.

Nel seguito, verrà utilizzata la realizzazione bidirezionale circolare con sentinella che permette di ottenere complessità O(1) per ciascuna operazione semplificandone nel contempo il codice Pascal, a scapito di un lieve spreco di memoria. Va però notato che è possibile ottenere complessità O(1) per quasi tutte le operazioni (con l'esclusione dei soli ULTIMOLISTA e PREDLISTA che hanno complessità $O(n)$) anche con una semplice realizzazione monodirezionale, senza circolarità né sentinelle, usando quindi la minore quantità di memoria possibile ma complicando il codice Pascal delle operazioni.

Con la realizzazione bidirezionale circolare con sentinella, la posizione pos_i è uguale all'indirizzo della cella contenente a_i, per $1 \leq i \leq n$, mentre pos_0 e pos_{n+1}

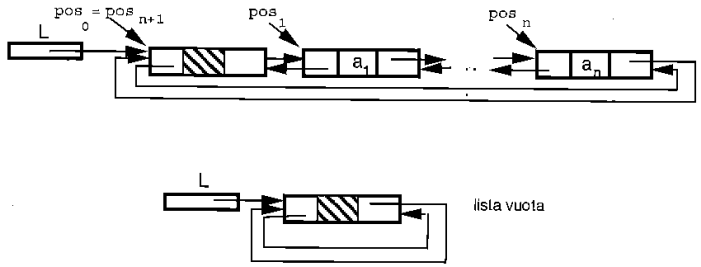

Figura 2.2: Realizzazione bidirezionale circolare con sentinella.

sono uguali all'indirizzo della sentinella (si veda la Fig. 2.2). Si noti che una lista di n elementi è formata da $n + 1$ celle (che contengono gli n elementi più la sentinella) e che quindi una lista vuota contiene esattamente una cella (la sentinella) che ha il proprio indirizzo sia come indirizzo della cella precedente che della cella successiva.

Utilizzando i puntatori del Pascal, si ottiene la seguente realizzazione, dove gli operatori che inizializzano e modificano la lista sono scritti tramite procedure, mentre quelli che restituiscono un valore sono scritti tramite function.

```
type
    lista = ↑cella;
    posizione = lista;
    cella = record
        precedente: posizione;
        elemento: tipoelem;
        successivo: posizione
    end;

procedure CREALISTA(var L: lista);
    begin
        new(L);
        L↑.successivo := L; L↑.precedente := L
    end;

function LISTAVUOTA(var L: lista): boolean;
    begin
        LISTAVUOTA := (L↑.successivo = L) and (L↑.precedente = L)
    end;

function PRIMOLISTA(var L: lista): posizione;
    begin
        PRIMOLISTA := L↑.successivo
    end;

function ULTIMOLISTA(var L: lista): posizione;
    begin
        ULTIMOLISTA := L↑.precedente
    end;
```



```
function SUCCLISTA(p: posizione; var L: lista): posizione;
    begin
        SUCCLISTA := p↑.successivo
    end;

function PREDLISTA(p: posizione; var L: lista): posizione;
    begin
        PREDLISTA := p↑.precedente
    end;

function FINELISTA(p: posizione; var L: lista): boolean;
    begin
        FINELISTA := (p = L);
    end;

function LEGGILISTA(p: posizione; var L: lista): tipoelem;
    begin
        LEGGILISTA := p↑.elemento
    end;

procedure SCRIVILISTA(a: tipoelem; p: posizione; var L: lista);
    begin
        p↑.elemento := a
    end;

procedure INSLISTA(a: tipoelem; var p: posizione; var L: lista);
    var temp: posizione;
    begin
        new(temp);
        temp↑.precedente := p↑.precedente;
        temp↑.successivo := p;
        p↑.precedente↑.successivo := temp;
        p↑.precedente := temp;
        temp↑.elemento := a;
        p := temp
    end;

procedure CANCLISTA(var p: posizione; var L: lista);
    var temp: posizione;
    begin
        temp := p;
        p↑.precedente↑.successivo := p↑.successivo;
        p↑.successivo↑.precedente := p↑.precedente;
        p := p↑.successivo;
        dispose(temp)
    end
```

Si noti che nelle procedure INSLISTA e CANCLISTA la posizione p è passata per variabile ed è modificata all'interno della procedura stessa. Infatti, in accordo alla specifica di INSLISTA, se $p = pos_i$, il nuovo elemento inserito diventa l'i-esimo, mentre quello che era il vecchio i-esimo diventa l'$(i + 1)$-esimo, ecc. Analogamente,

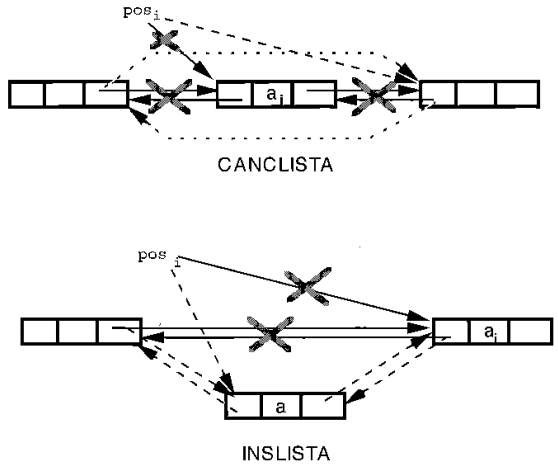

Figura 2.3: Aggiornamento dei puntatori nelle procedure CANCLISTA e INSLISTA

nel caso di CANCLISTA, il vecchio $(i+1)$-esimo elemento diventa l'i-esimo, ecc. In entrambi i casi, p deve contenere l'indirizzo della cella che contiene il nuovo i-esimo elemento (si veda la Fig. 2.3).

Si noti inoltre che utilizzando i puntatori è molto gravoso, se non addirittura impossibile, verificare che siano soddisfatte le condizioni sugli input di ogni operazione. Per esempio, la specifica di LEGGILISTA(p, L) prevede che p sia una posizione di un elemento della lista L; essendo p realizzato con un puntatore, la verifica di questa precondizione richiederebbe la scansione della lista! Per ragioni di efficienza, la condizione non viene verificata, ed il corretto uso delle operazioni è quindi piena responsabilità del programmatore.

2.3 Realizzazione con cursori

Nei linguaggi che non hanno i puntatori come tipi di dato primitivi (p.e. in Fortran o in Algol) è possibile simulare i puntatori con "cursori", cioè con variabili intere il cui valore è interpretato come un indice di un vettore. Questo vettore simula la memoria disponibile per i puntatori, che deve essere gestita esplicitamente nella realizzazione. Per far questo, si definisce un vettore SPAZIO che:

1. contiene tutte le liste, ognuna individuata da un proprio cursore iniziale;
2. contiene tutte le celle libere, organizzate anch'esse in una lista, detta "lista libera".

Esempio 2.3 (Lista libera). Siano $L = 8, 5$ ed $M = 2, 13, 7$ due liste di interi ed il vettore SPAZIO contenga in tutto 10 celle, ciascuna divisa nei campi "precedente", "elemento" e "successivo". Una possibile configurazione del vettore SPAZIO e delle tre liste L, M e *listalibera* è mostrato nella Fig. 2.4. L è memorizzata nelle celle 1, 7 e 4, M nelle celle 3, 5, 2 e 9, mentre *listalibera* comprende le celle 8, 0 e 6. □

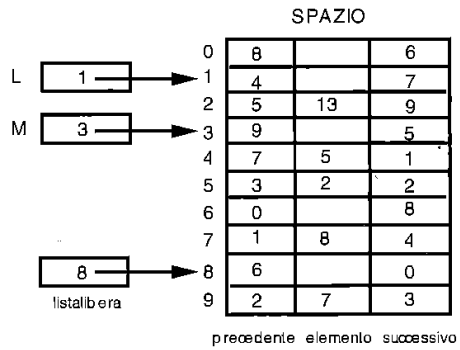

Figura 2.4: Memorizzazione delle liste L = 8, 5 ed M = 2, 13, 7 nel vettore SPAZIO.

Utilizzando i cursori, si può definire la posizione pos_i dell'i-esimo elemento di L uguale al valore del cursore alla cella del vettore SPAZIO che contiene a_i, se $1 \leq i \leq n$, o uguale al valore del cursore alla sentinella, se $i = 0$ oppure $i = n+1$. Assumendo che una opportuna procedura INIZIALIZZALISTALIBERA includa inizialmente tutte le celle del vettore SPAZIO nella lista libera, si ha:

```
const maxlung = ... {opportuna costante intera positiva};
type lista = 0..maxlung − 1;
   posizione = lista;
var listalibera: lista;
   SPAZIO: array[0..maxlung − 1] of record
      precedente: posizione;
      elemento: tipoelem;
      successivo: posizione
   end

procedure INIZIALIZZALISTALIBERA;
   var i: integer;
   begin
      listalibera := 0;
      for i := 0 to maxlung − 1 do begin
         SPAZIO[i].successivo := (i + 1) mod maxlung;
         SPAZIO[i].precedente := (i − 1) mod maxlung
      end
   end
```

La realizzazione delle operazioni usando cursori è analoga a quella mediante puntatori (per esempio, le espressioni $p\uparrow$.*successivo* e $p\uparrow$.*successivo*$\uparrow$.*precedente* diventano SPAZIO[p].*successivo* e SPAZIO[SPAZIO[p].*successivo*].*precedente*). Gli unici cambiamenti si hanno nelle procedure INSLISTA e CANCLISTA, dove occorre "spostare" una cella dalla *listalibera* alla lista L o da L alla *listalibera*.

Per realizzare queste operazioni, è comodo definire una procedura SPOSTA(h, k) che trasferisce la cella puntata da h, spostandola prima della cella puntata da k.

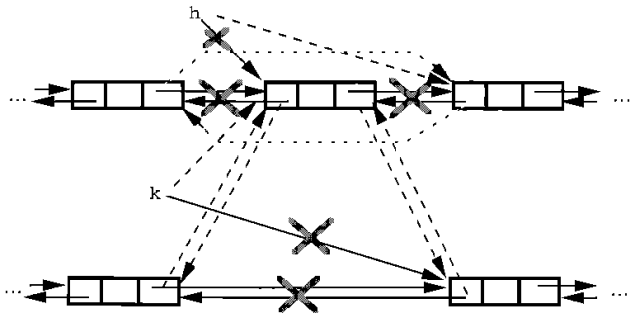

Figura 2.5: Aggiornamento dei cursori nella procedura SPOSTA.

La procedura aggiorna il cursore h in modo che punti alla cella che seguiva quella spostata e il cursore k in modo che punti alla cella spostata (si veda la Fig. 2.5).

Il codice di SPOSTA è analogo a quello di una cancellazione e di una inserzione simultanee (si confronti il codice di SPOSTA con quello delle procedure CANCLISTA e INSLISTA viste precedentemente, sostituendo i cursori ai puntatori).

```
procedure SPOSTA(var h, k: posizione);
    var temp: integer;
    begin
        temp := SPAZIO[h].successivo;
        SPAZIO[SPAZIO[h].precedente].successivo := SPAZIO[h].successivo;
        SPAZIO[SPAZIO[h].successivo].precedente := SPAZIO[h].precedente;
        SPAZIO[h].precedente := SPAZIO[k].precedente;
        SPAZIO[SPAZIO[k].precedente].successivo := h;
        SPAZIO[h].successivo := k; SPAZIO[k].precedente := h;
        k := h; h := temp
    end
```

Le procedure INSLISTA e CANCLISTA chiamano SPOSTA al loro interno e utilizzano *listalibera* come variabile globale.

```
procedure INSLISTA(a: tipoelem; var p: posizione; var L: lista);
    begin
        if LISTAVUOTA(listalibera) then
            {messaggio di errore: listalibera vuota}
        else begin
            SPOSTA(listalibera, p);
            SPAZIO[p].elemento := a
        end
    end;

procedure CANCLISTA(varp: posizione; var L: lista);
    begin
        SPOSTA(p, listalibera)
    end
```


In pratica, in un linguaggio di programmazione che fornisca i puntatori come tipi di dato primitivi, i puntatori sono realizzati proprio riservando un vettore SPAZIO per la porzione di memoria ad essi destinata, organizzato come una lista libera. L'operazione *new* non fa altro che staccare una cella dalla lista libera e restituire l'indirizzo (cursore) che la cella ha nel vettore SPAZIO, mentre l'operazione *dispose* riaggancia alla lista libera la cella non più utilizzata. Uno dei vantaggi dei linguaggi con puntatori consiste proprio nello sgravare il programmatore dalla gestione della memoria comune riservata alle liste, che è effettuata automaticamente dal sistema.

La complessità delle operazioni, usando la realizzazione con cursori, è ovviamente la stessa di quella con i puntatori, e l'occupazione di memoria è O(*maxlung*) per tutte le liste.

Si noti che la complessità della procedura DONGIOVANNI, vista nell'Esempio 2.2, è $\Theta(n)$ sia utilizzando i puntatori sia utilizzando i cursori.

2.4 Esercizi

Esercizio 2.1 (Rango). Il rango di un elemento di una lista è definito come la somma del suo valore e dei valori degli elementi che lo seguono. Scrivere una procedura Pascal ricorsiva di complessità ottima che, data una lista L, modifica L in modo che ogni elemento contenga il proprio rango (per esempio, se $L = 3, 2, 5$, allora si vuole ottenere $L = 10, 7, 5$).

Una limitazione inferiore alla complessità del problema è data dalla lunghezza n della lista L, ed è quindi $\Omega(n)$. La soluzione ricorsiva si basa sulla seguente proprietà:

$$\begin{aligned} rango(a_i) &= rango(a_{i+1}) + a_i && \text{se } i < n \\ rango(a_i) &= a_n && \text{se } i = n. \end{aligned}$$

```
procedure RANGO(p: posizione; var L: lista);
    var a: integer; q: posizione;
    begin
        q := SUCCLISTA(p, L);
        if not FINELISTA(q, L) then begin
            RANGO(q, L);
            a := LEGGILISTA(p, L) + LEGGILISTA(q, L);
            SCRIVILISTA(a, p, L)
        end
    end
```

La chiamata della procedura è:

```
if not LISTAVUOTA(L) then RANGO(PRIMOLISTA(L), L);
```

Il tempo $T(n)$ richiesto dalla procedura verifica le seguenti relazioni di ricorrenza, con c e d costanti,

$$\begin{aligned} T(n) &= c && \text{per } n = 1 \\ T(n) &= T(n-1) + d && \text{per } n > 1 \end{aligned}$$

la cui soluzione è $\Theta(n)$. Pertanto la procedura ha complessità ottima.

Esercizio 2.2 (I primi n interi). Si scriva una procedura Pascal ricorsiva di complessità ottima che, dato un intero $n \geq 1$, costruisca due liste L ed M tali che $L = 1, 2, \ldots, n-1, n$ ed $M = n, n-1, \ldots, 2, 1$.

Una limitazione inferiore alla complessità del problema è $\Omega(n)$ perché un qualsiasi algoritmo deve almeno generare gli n elementi della lista L e della lista M.

La procedura PRIMIENNE effettua un inserimento in testa alla lista L ed uno in fondo alla lista M ad ogni chiamata ricorsiva, in modo da memorizzare gli interi in senso crescente in L e decrescente in M.

Prima di effettuare la chiamata della procedura, occorre aver inizializzato le due liste L ed M:

```
CREALISTA(L);
CREALISTA(M);
PRIMIENNE(n, PRIMOLISTA(L), PRIMOLISTA(M), L, M);
```

```
procedure PRIMIENNE(n: integer; p, q: posizione; var L, M: lista);
    begin
        if n ≥ 1 then begin
            INSLISTA(n, p, L);
            INSLISTA(n, q, M);
            q := SUCCLISTA(q, M);
            PRIMIENNE(n − 1, p, q, L, M)
        end
    end
```

La complessità si ottiene risolvendo delle relazioni di ricorrenza simili a quelle dell'Esercizio 2.1. Pertanto la procedura ha complessità $O(n)$ e risulta ottima.

Esercizio 2.3 (Epurazione). Si scriva una procedura Pascal che, data una lista L di interi, restituisca un'altra lista M che contenga solo gli elementi di L i cui valori non compaiono esattamente due volte (p.e., se $L = 5, 7, 3, 2, 2, 1, 2, 3$, allora $M = 5, 7, 2, 2, 1, 2$).

Si effettua una doppia scansione della lista L per selezionare un elemento da L, verificare che non compaia esattamente 2 volte, e inserirlo in M. La posizione p è quella dell'elemento di L da verificare, la posizione q è utilizzata per scandire L alla ricerca delle occorrenze dell'elemento di posizione p, mentre *conta* serve a contare quante volte tale elemento si ripete. La posizione r è quella che segue l'ultimo elemento di M. La complessità è $O(n^2)$.

```
procedure EPURAZIONE(var L, M: lista);
    var p, q, r: posizione; conta: integer;
    begin
        CREALISTA(M); r := PRIMOLISTA(M); p := PRIMOLISTA(L);
        while not FINELISTA(p, L) do begin
            q := PRIMOLISTA(L); conta := 0;
            while not (FINELISTA(q, L)) and (conta ≤ 2) do begin
                if LEGGILISTA(q, L) = LEGGILISTA(p, L) then
                    conta := conta + 1;
                q := SUCCLISTA(q, L)
            end;
            if conta ≠ 2 then begin
                INSLISTA(LEGGILISTA(p, L), r, M);
                r := SUCCLISTA(r, M)
            end;
            p := SUCCLISTA(p, L)
        end
    end
```


Esercizio 2.4. Si supponga di rappresentare un numero decimale D mediante una lista L tale che il valore dell'elemento i-esimo della lista sia uguale a quello dell'i-esima cifra del numero. Si scriva una funzione Pascal che effettua la somma di due numeri. (Esempio: se $D = 190$ e $D' = 17$ allora $L = 1, 9, 0$, $L' = 1, 7$, e il risultato $D'' = D + D'$ è restituito nella lista $L'' = 2, 0, 7$).

Esercizio 2.5 (Tangentopoli). Dopo essere stati inquisiti per tangentopoli, n politici hanno deciso di suicidarsi disponendosi in cerchio e uccidendo via via una persona ogni k (... ma l'ultimo rimasto si suiciderà poi per davvero?). Si scriva una procedura Pascal che, data la lista L degli aspiranti suicidi, e dato k, $1 \leq k \leq n$, stampi l'ordine delle vittime (p.e. se L = Caio, Sempronio, Tizio, Tullio e $k = 3$, allora l'ordine delle vittime è: Tizio, Sempronio, Tullio, Caio). Si assuma di poter cancellare gli elementi da L.

```
procedure TANGENTOPOLI(var L: lista);
    var p: posizione; i: integer;
    begin
        p := PRIMOLISTA(L);
        while not LISTAVUOTA(L) do begin
            for i := 1 to k − 1 do begin
                p := SUCCLISTA(p, L);
                if FINELISTA(p, L) then p := PRIMOLISTA(L)
            end;
            write(LEGGILISTA(p, L));
            CANCLISTA(p, L)
        end
    end
```

Esercizio 2.6. Si scriva una procedura Pascal ricorsiva di complessità ottima che, data una lista L di interi, riordina L in modo che tutti gli elementi dispari precedano, nello stesso ordine che avevano inizialmente in L, tutti gli elementi pari (p.e., se $L = 3, 7, 8, 1, 4$, allora si ottiene $L = 3, 7, 1, 8, 4$).

Esercizio 2.7. Si scriva una procedura Pascal ricorsiva che data una lista L di interi, la modifichi eliminando ogni elemento pari e replicando ogni elemento dispari tante volte quanti sono gli elementi pari che lo precedono (p.e., $L = 4, 6, 7, 3, 2, 5$, allora si ha $L = 7, 7, 7, 3, 3, 3, 5, 5, 5, 5$).

Esercizio 2.8. Si scriva una procedura Pascal che, data una lista L di interi restituisca un'altra lista M che contiene solo gli elementi di L i cui valori sono uguali alla differenza tra i valori dell'elemento immediatamente precedente e quello immediatamente seguente (p.e., se $L = 5, 2, 3, 1, 2$, allora $M = 2, 1$). Si assuma per semplicità che L contenga almeno tre elementi.

Esercizio 2.9. Si scrivano le procedure Pascal per le operazioni delle liste usando la realizzazione con puntatori monodirezionale non circolare e senza sentinella. Tutte le operazioni (tranne ULTIMOLISTA e PREDLISTA) devono avere complessità $O(1)$. (Suggerimento: si definisca pos_i uguale all'indirizzo della cella contenente a_{i-1}, se $i > 0$, e $pos_i =$ **nil**, se $i = 1$).