di Sergio Polini

Liste sequenziali e concatenate, semplici e doppie, lineari e circolari

La volta scorsa abbiamo cominciato a scrivere QUED, il «line editor» che vi ho proposto come esempio di uso delle liste. Abbiamo definito le caratteristiche dell'input e dell'output, ovvero di due delle tre fasi fondamentali di ogni programma. Ci rimane la fase di elaborazione. Abbiamo già accennato alle principali operazioni che il programma dovrà compiere sul testo: aggiunta di nuove righe, cancellazione e spostamento di brani, ricerca e sostituzione di stringhe, inserimento nel testo in memoria di un altro testo da un file, ecc. Così facendo abbiamo introdotto un argomento molto importante: il Pascal ci aiuta a cominciare il progetto di un programma dalla definizione dei suoi dati, dalla scelta delle strutture di dati più adatte; nel far questo, tuttavia, non bisogna dimenticare che non si deve solo badare alla «natura» delle informazioni, ma anche (e forse soprattutto) alle operazioni che su di esse vogliamo compiere

Un editor manipola testi, un testo è una successione di righe, una riga è una successione di caratteri. Di qui si fa presto a pensare ad un array come ad una successione di elementi dello stesso tipo, e quindi ad una rappresentazione del testo in memoria mediante un array di array di caratteri, cioè un array di stringhe.

L'array non è comunque l'unico modo di rappresentare dati la cui caratteristica fondamentale sia quella di stare «l'uno dopo l'altro», e può anche non essere il più efficiente. Se definissimo un array di 1000 stringhe di 80 caratteri ognuna, ad esempio, sprecheremmo memoria per tutte le righe lunghe meno di 80 caratteri e per ogni testo comprendente meno di 1000 righe, saremmo invece un po' in crisi se avessimo bisogno di più di 1000 righe. In realtà questi sono tutti problemi per i quali, se proprio si deve, una soluzione si trova. Ciò che più conta è che, se ragionassimo in quel modo, rischieremmo di trovarci poi in difficoltà al momento di scrivere le varie routine; conviene pensare subito a come vogliamo operare sui nostri dati, conviene

```
type
  NPtr = ^Nodo;
  Nodo = record
    Next: NPtr;
    Riga: string[80]
  end;
procedure InsDopo(A, C: NPtr);
begin
  C^.Next := A^.Next;
  A^.Next := C
end;
procedure InsPrima(B, C: NPtr);
var
  R: string[80];
begin
  C^.Next := B^.Next;
  B^.Next := C;
  R := B^.Riga;
  B^.Riga := C^.Riga;
  C^.Riga := R
end;
```

Figura 1
Inserimento di un nodo dopo o prima di un altro in una lista semplice.

chiedersi fin dal principio se altre strutture di dati possono rendere più facile o più efficiente l'implementazione delle operazioni sui dati.

Liste sequenziali e liste concatenate

Cominciamo con una generalizzazione del concetto di array. Una lista, nel suo significato più ampio, è semplicemente un insieme ordinato di elementi, dove «ordinato» vuol dire che ha senso parlare di elemento precedente o successivo ad un altro. Da questo punto di vista anche un array è una lista: ad ogni elemento è associato un indice, in modo che si può dire che a[i] viene dopo a[i-1] e prima di a[i+1]. L'implementazione di un array è molto semplice: i suoi elementi sono conservati in memoria in locazioni l'una consecutiva all'altra (se «a» è un array di interi e a[i] si trova all'indirizzo 260, a[i+1] si trova all'indirizzo 262). È per questo che possiamo dire che gli array sono *liste sequenziali*.

Nella puntata di aprile abbiamo però visto che possiamo creare strutture di dati ordinate anche in un altro modo: ogni elemento (o nodo) «punta» al successivo, in modo che è possibile, seguendo la «catena» di puntatori, passare da un elemento A al successivo B anche se A e B non hanno indirizzi consecutivi. L'indirizzo di a[i+1] è immediatamente deducibile se conosciamo l'indirizzo (e la dimensione) di a[i]; nel caso delle *liste concatenate*, invece, l'informazione relativa all'indirizzo di B va aggiunta al contenuto di A. È chiaro che, così facendo, occupiamo memoria: un array di due stringhe di 80 caratteri ognuna occupa 162 byte; una lista di due record, ognuno con un campo string[80] e un altro di tipo puntatore, occupa 170 byte. Ma anche con gli array, come abbiamo visto sopra, ci può essere un problema di spreco di memoria.

Il vero e proprio confronto tra liste sequenziali e liste concatenate va infatti condotto sulla base di quello che vogliamo fare con i nostri dati. Si possono distinguere diversi tipi di operazioni: accedere ad un elemento per leggerlo o modificarlo, inserire o cancellare un elemento, percorrere tutta la lista nei due sensi, determinarne la lunghezza, fondere più liste in una, scomporre una lista in più sotto-liste, cercare l'elemento che abbia un particolare valore, ordinare la lista. Basta un attimo di riflessione per vedere che con una lista sequenziale sono molto rapide operazioni come l'accesso all'ennesimo elemento (ad esempio: a[n] := 4), ma macchinosi l'inserimento e la cancellazione, in quanto si rende necessario far scorrere di un posto tutti gli elementi successivi a quello inserito o cancellato; con una lista concatenata, invece, abbiamo visto ad aprile quanto siano semplici l'inserimento e la cancellazione (basta cambiare il valore di pochi puntatori), ma può essere necessario passare attraverso i precedenti n-1 elementi per accedere all'ennesimo.

Ecco perché bisogna valutare quali tipi di operazioni richiede il nostro programma per poter poi scegliere la rappresentazione più adatta. La scelta non è facile nel caso di un editor orientato alla riga, in quanto abbiamo bisogno sia di accedere alla riga ennesima che di inserire, cancellare, muovere e copiare righe; decidere è difficile anche perché sarà l'utente a operare in un modo piuttosto che in un altro, e tuttavia la presumibile frequenza delle operazioni per le quali meglio si prestano le liste concatenate può far optare per queste.

Nel nostro caso inoltre, come avevamo detto la volta scorsa, scegliamo le liste concatenate anche perché queste, sicuramente più adatte per la realizzazione di editor a tutto schermo, vengono ampiamente utilizzate nell'editor Toolbox.

Non vi è poi un solo tipo di lista concatenata, e le possibili variazioni sul tema consentono di adattare la struttura di dati alle particolari necessità del singolo programma.

Liste semplici e liste doppie

La lista che abbiamo visto ad aprile era percorribile solo in un senso. Talvolta ciò è sufficiente (c'è una piccola lista «unidirezionale» anche in QUED, nella procedura CopiaSubLista del file QLIST.INC), ma spesso è necessario poter procedere nei due sensi; in QUED, ad esempio, dobbiamo poter cercare una stringa contenuta in una riga successiva (con il comando «/stringa/») o precedente (con «?stringa?»). Per ottenere ciò basta aggiungere ad ogni elemento, accanto al puntatore (che chiameremo Next) all'elemento successivo, anche un puntatore (che chiameremo Prev) a quello precedente; si ottengono così anche altri vantaggi.

L'inserimento di un nodo in una lista concatenata è in generale molto semplice: se B viene dopo A, A contiene un puntatore a B; per inserire C tra i due basta assegnare al puntatore di C l'indirizzo di B (contenuto nel puntatore di A) e al puntatore di A l'indirizzo di C.

Figura 2 Inserimento di un nodo dopo o prima di un altro in una lista doppia.

```
type
  NPtr = ^Nodo;
  Nodo = record
    Next, Prev: NPtr;
    Riga      : string[80]
  end;
procedure InsDopo(A, C: NPtr);
begin { A^.Next e' B }
  A^.Next^.Prev := C;
  C^.Next       := A^.Next;
  C^.Prev       := A;
  A^.Next       := C
end;
procedure InsPrima(B, C: NPtr);
begin { B^.Prev e' A }
  B^.Prev^.Next := C;
  C^.Prev       := B^.Prev;
  C^.Next       := B;
  B^.Prev       := C
end;
```

Così però si inserisce un elemento *dopo* un altro: dobbiamo cioè avere accesso ad A per operare; se fossimo invece in B non potremmo con la stessa facilità inserire C *prima* di B. La complicazione non è molta: basta inserire C dopo B e poi scambiare il contenuto degli altri campi di B e C. Probabilmente sarà però anche necessario cambiare il valore di un puntatore al nodo corrente; una lista è infatti spesso accompagnata da variabili «ausiliarie» di tipo puntatore: una può contenere l'indirizzo del nodo corrente, un'altra quello dell'ultimo nodo della lista (per potervi accedere senza dover passare per tutti i precedenti) ecc.

Le liste doppie consentono di evitare il ricorso a trucchi del genere: è possibile inserire un nodo prima o dopo di un altro con la stessa immediatezza, oppure anche prevedere la sola routine di «inserimento dopo» e passare a questa, nel caso di inserimento di C prima di B, B.^ Prev invece di B: sarà proprio questa la soluzione che adotteremo per QUED, quando si tratterà di implementare i comandi «a» (append: aggiunta di righe dopo un'altra) e «i» (insert: inserimento di righe prima di un'altra).

In conclusione, una lista doppia consente operazioni di inserimento più facili, e in generale una maggiore flessibilità.

Liste lineari e liste circolari

Molte volte può bastare percorrere una lista fino al primo o fino all'ultimo nodo, cioè fino all'uno o all'altro dei suoi estremi. Negli editor del Toolbox, ad esempio, come anche nel WordStar, i comandi di ricerca di stringhe partono alla riga corrente ed effettuano la ricerca fino alla prima o all'ultima riga del testo, secondo che si sia scelta o no l'opzione «b» (backwards).

QUED funziona un po' diversamente: il comando «/pippo/» cerca pippo «in avanti», cioè dalla riga successiva a quella corrente verso la fine del testo; se però non la trova prosegue con la prima riga, ritornando per questa via fino alla riga corrente. Anche altri programmi possono aver bisogno di percorrere tutta una lista quale che sia il nodo di partenza.

Per congiungere la «testa» e la «coda» di una lista basta assegnare al puntatore Next nell'ultimo nodo l'indirizzo del primo, e, se la lista è doppia, al puntatore Prev del primo nodo l'indirizzo dell'ultimo. Con ciò non si rischia di perdere la nozione di dove la lista inizia e finisce, in quanto, come dicevamo sopra, una lista è sempre accompagnata da variabili ausiliarie di tipo puntatore, di cui una può puntare al primo o all'ultimo nodo.

Può tuttavia convenire «marcare» diversamente gli estremi di una lista, definendo un nodo «zero» destinato ad essere inserito tra l'ultimo e il primo (dopo l'ultimo e prima del primo). In questo modo si risolve anche un altro problema. Abbiamo prima parlato di inserimento di un nodo in una lista facendo sempre riferimento a nodi preesi-

```
{ QLIST.INC }
{
Insieme di routine per la creazione e manipolazione di liste circolari
doppie. Le routine possono essere utilizzate per liste i cui elementi
vengano dichiarati come segue:
   type
     NPtr = ^Nodo;          (* puntatore ad un "nodo" *)
     Nodo = record
       Prev, Next: NPtr;    (* puntatori ai nodi precedente e successivo   *)
       .......              (* altri campi; in QUED un campo Txt = ^AnyStr *)
     end;                   (* dove AnyStr = string[255] *)
Le istruzioni specifiche di Qued (in quanto relative agli "altri campi")
sono affiancate da un commento: (* QUED *).
I nomi delle procedure specifiche di Qued iniziano con "Q_" e sono ambedue
relative all'"altro campo" Txt.
NB: le routine richiedono anche la precedente dichiarazione di una costante
MINMEM e di costanti per i codici di errore.
}
procedure CreaNodo(var P: NPtr);
begin
  if (maxavail > 0) and (maxavail < MINMEM) then P := nil
  else new(P)
end;

procedure InsNodo(P, Q: NPtr);
{ inserisce un nodo P nella lista subito dopo il nodo Q }
begin
  Q^.Next^.Prev := P;
  P^.Next := Q^.Next;
  P^.Prev := Q;
  Q^.Next := P
end;

procedure Q_CreaRiga(s: AnyStr; RPtr: NPtr; var Esito: integer);
{ Alloca memoria per il campo Txt di un nodo-riga }
var
  ls: integer;
begin
  if (maxavail > 0) and (maxavail < MINMEM) then Esito := ERRNOMEM
  else begin
    ls := length(s) + 1;                { numero di byte da allocare }
    getmem(RPtr^.Txt, ls);
    RPtr^.Txt^ := s;
    Esito := OK
  end
end;


procedure Q_AddRiga(s: AnyStr; var RPtr: NPtr; var Esito : integer);
{ Inserisce un nodo-riga nel testo in memoria }
var
  P: NPtr;
begin
  Esito := ERRNOMEM;
  CreaNodo(P);
  if P <> nil then Q_CreaRiga(s, P, Esito);  { Esito := OK se tutto bene }
  if Esito = OK then begin
    InsNodo(P, RPtr); RPtr := P
  end
end;

procedure MuoviSubLista(P1, P2, Q: NPtr; var Esito: integer);
{ Porta la sublista P1-P2 da dove era a subito dopo Q, dopo aver controllato }
{ che l nodo puntato da Q non sia ne' quello precedente al nodo puntato da  }
{ P1 (movimento "inutile"), ne' compreso nella lista P1-P2 }
var
  P: NPtr;
begin
  Esito := OK; P := P1^.Prev;
  while (Esito = OK) and (P <> P2^.Next) do begin
    if Q = P then Esito := ERRDEST;
    P := P^.Next
  end;
  if Esito = OK then begin
    P1^.Prev^.Next := P2^.Next;         { congiunge il nodo prima di P1 }
    P2^.Next^.Prev := P1^.Prev;                        { a quello dopo P2 }
    P1^.Prev := Q;                      { inserisce la lista P1-P2 tra  }
    P2^.Next := Q^.Next;                                { Q e il nodo dopo Q }
    Q^.Next^.Prev := P2;
    Q^.Next := P1
  end
end;
```

Figura 3 - Il file QLIST.INC, contenente le routine di manipolazione di liste circolari doppie usate

stenti: è ovvio che non si possono usare le tecniche che abbiamo descritto per creare una lista, cioè per inserire nodi in una lista vuota (potete trovare nella procedura CopiaSubLista un esempio di gestione di una lista senza un tale nodo; prima dell'aggiunta di un nodo si deve vedere se IniCopia vale **nil**, cioè se la lista è vuota o no, e comportarsi diversamente nei due casi). Se però per prima cosa creiamo un nodo «zeresimo», assegnando ai suoi puntatori Next e Prev l'indirizzo di questo stesso nodo, la lista non sarà mai vuota e non vi saranno più problemi. Potremo sempre riconoscere l'ultimo nodo della lista: nel caso di lista lineare ultimo sarà quel nodo in cui Next vale **nil**, nel caso di lista circolare con nodo «zero» ultimo sarà il nodo il cui Next è uguale a NodoZeroPtr, quello cioè che contiene l'indirizzo del nodo «zero».

```
procedure CopiaSubLista(P1, P2, Q: NPtr; var Esito: integer);
{ Crea una copia della sublista P1-P2 e la inserisce subito dopo Q }
var
  IniCopia, FinCopia: NPtr;      { ptr al primo e all'ultimo nodo della copia }
  NodoOrig          : NPtr;      { ptr al nodo "originale" che viene copiato }
  Temp              : NPtr;
  St: integer;                                              (* QUED *)
procedure CancellaCopia;
  begin
    Temp := IniCopia;
    while Temp <> nil do begin
      Temp := Temp^.Next;
      freemem(IniCopia^.Txt, length(IniCopia^.Txt^)+1);     (* QUED *)
      dispose(IniCopia); IniCopia := Temp
    end;
    Esito := ERRNOMEM
  end;
begin
  IniCopia := nil; Temp := nil; NodoOrig := P1;
  while NodoOrig <> P2^.Next do begin          { crea una copia della sublista }
    CreaNodo(FinCopia);
    if FinCopia = nil then begin               { se non c'e' memoria sufficiente }
      CancellaCopia; exit
    end;
    FinCopia^ := NodoOrig^;
    Q_CreaRiga(NodoOrig^.Txt^, FinCopia, St);               (* QUED *)
    if St = OK then begin                                   (* QUED *)
      if IniCopia = nil then begin                 { prima linea della copia }
        IniCopia := FinCopia; Temp := FinCopia;
        FinCopia^.Prev := nil; FinCopia^.Next := nil
      end
      else begin
        Temp^.Next := FinCopia; FinCopia^.Prev := Temp;
        FinCopia^.Next := nil; Temp := Temp^.Next
      end;
      NodoOrig := NodoOrig^.Next
    end                                                     (* QUED *)
    else begin                                              (* QUED *)
      CancellaCopia; exit                                   (* QUED *)
    end                                                     (* QUED *)
  end;
  IniCopia^.Prev := Q;                         { inserisce la copia della lista }
  FinCopia^.Next := Q^.Next;                       { dopo il nodo puntato da Q }
  Q^.Next^.Prev := FinCopia;
  Q^.Next := IniCopia;
  Esito := OK
end;



procedure DelSubLista(P1, P2: NPtr);
{ Cancella la sublista P1-P2 }
var
  M, N: NPtr;
begin
  P1^.Prev^.Next := P2^.Next;                  { congiunge il nodo prima di P1 }
  P2^.Next^.Prev := P1^.Prev;                                { a quello dopo P2 }
  M := P1; N := P1;
  while M <> nil do begin
    if N <> P2 then N := N^.Next
    else N := nil;
    freemem(M^.Txt, length(M^.Txt^)+1);                     (* QUED *)
    dispose(M); M := N
  end
end;

procedure CreaLista(var P: NPtr);
begin
  CreaNodo(P);
  if P <> nil then with P^ do begin
    Prev := P; Next := P;
  end
end;
```

dal programma QUED. La lista per il testo da elaborare viene creata con CreaLista (NodoZeroPtr).

New e Getmem

Il file QLIST.INC contiene le routine di gestione delle liste usate dal programma QUED; tutte le routine tranne due possono essere agevolmente inserite in altri programmi, seguendo le indicazioni poste nel commento all'inizio del file. Le due routine specifiche di QUED servono ad eliminare un possibile spreco di memoria.

Supponiamo di dichiarare i nodi della nostra lista come record contenenti, oltre ai puntatori Next e Prev, anche un campo Riga di tipo stringa: dovremmo determinare la lunghezza massima di questa stringa, e così sprecare memoria per ogni riga più corta. Se, come per altro verso conviene, fissiamo in 255 caratteri la lunghezza massima di una riga, lo spreco può essere notevole.

La soluzione ci viene offerta dalla procedura Getmem. La procedura New alloca tanta memoria quanta ne richiede il tipo dell'oggetto puntato dalla variabile passata alla procedura, mentre Getmem consente di controllare la quantità di memoria allocata.

Se P è una variabile di tipo puntatore a un nodo che contiene anche un campo di tipo string [255], New(P) alloca sempre 256 byte oltre quelli necessari per gli altri campi del nodo, indipendentemente dalla effettiva lunghezza della stringa.

Può convenire quindi sostituire quel campo con un altro di tipo puntatore a stringhe (chiamato Txt in QUED); quando poi si deve allocare memoria per un nodo contenente una stringa di 43 caratteri, possiamo usare prima New(P) per il nodo e poi Getmem (P^.Txt,44) per la stringa (i 43 caratteri più il byte di lunghezza). Questo richiede un po' d'attenzione quando si rilascia il nodo: bisogna prima liberare con Freemem la memoria allocata con Getmem (quella puntata dal campo-puntatore), poi con Dispose quella occupata da tutto il nodo. Ma ne può valere la pena.

Per ora ci fermiamo qui. Non abbiamo esaurito l'argomento «elaborazione»: ci mancano le routine di esecuzione dei comandi dati dall'utente, che cominceremo a vedere il mese prossimo. MC