Lezione 19
9–10 dicembre 1999

Argomenti trattati
- Puntatori e variabili dinamiche.
- Stack e heap.
- Introduzione alle liste.

19.1 I puntatori

Variabili statiche e variabili dinamiche

Le variabili che abbiamo utilizzato sinora sono dette *variabili statiche*. Una variabile statica viene dichiarata nel programma principale o in un sottoprogramma ed è denotata da un identificatore. La variabile esiste per tutta la durata dell'esecuzione del blocco in cui è stata dichiarata. Quando l'esecuzione del blocco ha termine, la variabile statica viene distrutta insieme al record di attivazione del blocco stesso.

Le *variabili dinamiche*, al contrario, possono essere create e distrutte dinamicamente durante l'esecuzione, indipendentemente dalla struttura statica del programma. Esse non vengono dichiarate esplicitamente e non sono accessibili tramite identificatori propri, ma vengono create e distrutte mediante le procedure predefinite `new` e `dispose` e sono accessibili tramite *puntatori*.

Mentre con le variabili statiche si possono creare solo strutture di dimensione fissa (con l'unica eccezione dei file), mediante le variabili dinamiche è possibile creare strutture dati molto flessibili, la cui dimensione può variare durante l'esecuzione del programma.

Puntatori e variabili dinamiche

In Pascal un *puntatore* è una variabile che contiene o il riferimento ad una variabile dinamica oppure il valore `NIL`. In altre parole, un puntatore indica dove si trova una certa variabile dinamica; un puntatore contentente `NIL`, invece, non indica alcuna variabile. La variabile indicata (o *puntata*) da un puntatore può essere creata dinamicamente utilizzando la procedura `new`.

Per introdurre un tipo puntatore occorre prima di tutto specificare il tipo delle variabili a cui il puntatore può puntare. Ad esempio, per definire il tipo "puntatore a `integer`" e dichiarare due variabili `p` e `q` di tale tipo, possiamo scrivere:

```
TYPE
```

```
   punt = ^integer;
VAR
   p, q: punt;
```

In questo modo, ognuna delle variabili `p` e `q` è in grado di contenere un riferimento ad una variabile di tipo `integer`, creata dinamicamente utilizzando la procedura `new`. Inoltre, `p` e `q` possono contenere il valore `NIL` per indicare che non puntano ad alcuna variabile.

Inizialmente, come per tutte le variabili, il valore dei puntatori `p` e `q` è indefinito. Per assegnare a `q` il valore `NIL`, è sufficiente scrivere:

```
q := NIL
```

Per creare una variabile dinamica di tipo `integer` e far puntare `p` ad essa utilizziamo la chiamata:

```
new(p)
```

L'effetto di questa chiamata è quello di

- creare (o *allocare*) una variabile dinamica del tipo a cui punta `p`, in questo caso `integer`;
- far puntare `p` a tale variabile.

La variabile dinamica di tipo `integer` puntata da `p` può essere indicata con `p^`. Ad esempio, per assegnare il valore `8` alla variabile puntata da `p`, possiamo scrivere

```
p^ := 8
```

Per incrementare di `1` il valore di `p^` scriviamo:

```
p^ := p^ + 1
```

Se un puntatore contiene `NIL`, cioè se non punta ad alcun oggetto, l'uso dell'operatore `^` per accedere alla variabile puntata provoca un errore in esecuzione. Ad esempio, se `q` contiene `NIL`, l'esecuzione di un'istruzione contenente `q^` provocherà errore.

Come per le altre variabili, è possibile copiare un puntatore nell'altro, con un semplice assegnamento, come:

```
q := p
```

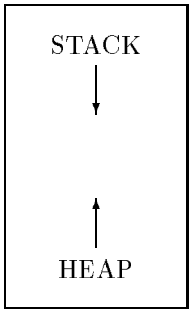
Dopo l'esecuzione di tale assegnamento, `q` punterà alla *stessa* variabile a cui punta `p`. Scrivendo invece

```
q^ := p^
```

il valore della variabile a cui punta `p` viene copiato nella variabile a cui punta `q` (chiaramente `p` e `q` non devono contenere `NIL`). Gli esempi presentati successivamente evidenziano questa differenza fondamentale.

Mentre le variabili statiche sono memorizzate nello stack, le variabili dinamiche vengono memorizzate in un'altra area di memoria detta *heap*. In termini di memoria, un puntatore può essere visto come l'indirizzo di una variabile nello heap. Ad esempio, se `p` punta alla variabile che si trova all'indirizzo 1000 dello heap, il contenuto di `p` sarà appunto `1000`. Pertanto, una variabile puntatore contiene l'indirizzo a cui reperire una variabile dinamica.

Concettualmente lo stack e lo heap sono due aree di memoria differenti. Solitamente vengono realizzati utilizzando la medesima area, dai due estremi opposti: ad esempio, lo stack può iniziare nella parte piú alta della memoria e crescere verso il basso, mentro lo heap può iniziare nella parte piú bassa e crescere verso l'alto. Quando le due aree si incontrano, l'esecuzione viene interrotta per esaurimento della memoria disponibile, segnalando un messaggio d'errore come "stack overflow".



Memoria

Le variabili dinamiche possono essere distrutte utilizzando la procedura `dispose`. In particolare, `dispose(p)` distrugge (o *dealloca*) la variabile puntata da `p`: il valore di `p` diventa indefinito; l'area di memoria in cui si trovava la variabile puntata da `p` diventa disponibile al sistema che può utilizzarla successivamente, quando venga richiesta l'allocazione di nuove variabili dinamiche. La procedura `dispose` va usata con particolare attenzione. Infatti, se `p` e `q` puntano alla stessa variabile, `dispose(p)` distrugge tale variabile. Pertanto, anche l'oggetto `q^` cessa di esistere.

Esempi

Si consideri il seguente programma:

```
PROGRAM p1 (output);

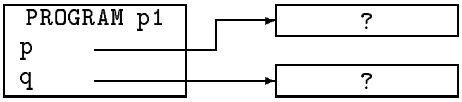
   TYPE
      punt = ^integer;
   VAR
      p, q: punt;

BEGIN {p1}
   new(p);
   new(q);
   p^ := 5;
   q^ := p^; {*}
   writeln(p^, q^);
   p^ := 10;
   writeln(p^, q^)
END. {p1}
```

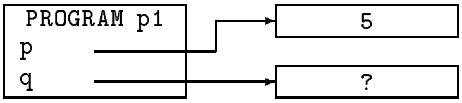
Rappresentiamo il record di attivazione di `p1` in cui trovano spazio le variabili statiche `p` e `q` di tipo `punt`, che inizialmente hanno valore indefinito (visto che il programma non contiene sottoprogrammi, per brevità evitiamo di rappresentare i campi `ris.fz.` e `ritorno`).

PROGRAM p1	
p	?
q	?

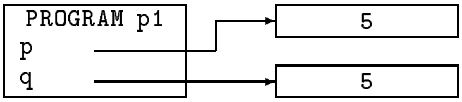
L'effetto delle prime due istruzioni del programma è quello di creare due aree di memoria nello heap, di dimensione tale da contenere valori di tipo `integer`, e di far puntare ciascuna delle variabili `p` e `q` a una di queste aree. Il valore contenuto nelle due aree è indefinito. Rappresentiamo le variabili nello heap separatamente da quelle nello stack:



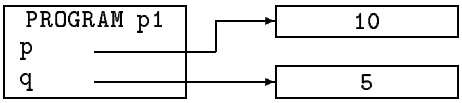
L'istruzione successiva è l'assegnamento `p^ := 5`, che va a modificare il contenuto della variabile dinamica puntata da `p`:



Viene poi effettuato l'assegnamento `q^ := p^`, nel quale il valore della variabile `p^` è assegnato alla variabile `q^`. Pertanto, il contenuto della memoria diventa:



La successiva istruzione di stampa, `writeln(p^, q^)`, scriverà in output i valori `5 5`. Viene poi effettuato l'assegnamento `p^ := 10`, che modifica il contenuto della variabile dinamica puntata da `p`:



Dunque, la seconda istruzione di scrittura produrrà in output i valori `10 5`.

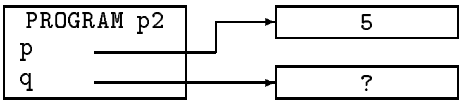
Consideriamo ora il seguente programma, che differisce dal precedente solo nell'assegnamento marcato con `{*}`.

```
PROGRAM p2 (output);

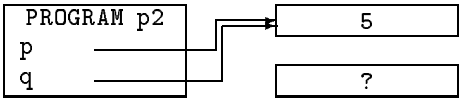
   TYPE
     punt = ^integer;
   VAR
     p, q: punt;

BEGIN {p2}
   new(p);
   new(q);
   p^ := 5;
   q := p; {*}
   writeln(p^, q^);
   p^ := 10;
   writeln(p^, q^)
END. {p2}
```

Prima dell'esecuzione di tale assegnamento il contenuto della memoria sarà:



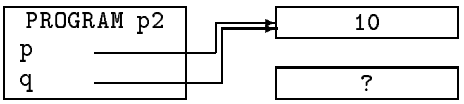
L'effetto dell'assegnamento `q := p` è quello di copiare in `q` il contenuto di `p` (cioè il puntatore memorizzato in `p`). Dunque, dopo la sua esecuzione, `q` *punterà alla stessa variabile dinamica* a cui punta `p`. L'altra variabile dinamica non risulta piú accessibile:



Osserviamo che, mentre dopo l'esecuzione dell'istruzione marcata con `{*}` del programma `p1`, le variabili dinamiche `p^` e `q^` hanno lo stesso valore, ma sono distinte, dopo l'esecuzione dell'istruzione `{*}` di `p2`, `p^` e `q^` *sono la stessa variabile*.

La successiva istruzione di scrittura produrrà in output i valori `5 5`.

Consideriamo ora l'assegnamento `p^ := 10`. Esso va a modificare la variabile puntata da `p`:



Poiché `p` e `q` puntano alla stessa variabile, la successiva `writeln(p^, q^)` produrrà in output i valori `10 10`, sebbene non si sia eseguita alcuna operazione in cui sia stato utilizzato esplicitamente il nome `q^`.

19.2 Introduzione alle strutture dati dinamiche

Come abbiamo detto, mediante le variabili dinamiche è possibile creare strutture dati estremamente flessibili, la cui dimensione può variare durante l'esecuzione. Presentiamo ora un esempio in cui utilizziamo una struttura di questo tipo.

Vogliamo costruire un programma che legga da `input` una sequenza di numeri interi e la riscriva in `output` al contrario. Per convenzione, l'inserimento di `0` indica la fine della sequenza. Richiediamo inoltre che il programma sia in grado di operare con sequenze di lunghezza arbitraria, senza alcun limite prefissato.

Ad alto livello, il programma sarà costituito da due fasi fondamentali: la prima di acquisizione della sequenza da `input`, la seconda di scrittura della sequenza rovesciata in `output`. Costruiremo dunque due procedure di nome `lettura` e `scrittura`, corrispondenti a queste due fasi. Piú precisamente, supponendo di avere definito un tipo di nome `tiposequenza`, mediante il quale rappresentare la sequenza, possiamo definire le seguenti intestazioni:

```
PROCEDURE lettura (VAR s: tiposequenza);
{riceve da input una sequenza di numeri e la memorizza nel parametro s}
```

e

```
PROCEDURE scrittura (s: tiposequenza);
{scrive in output il contenuto della sequenza memorizzata in s}
```

Il programma principale avrà dunque la seguente struttura:

```
PROGRAM seq (input, output);

   TYPE
      tiposequenza = ...;

   VAR
      sequenza: tiposequenza;

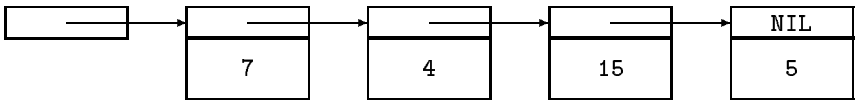
   ...sottoprogrammi...
```

```
BEGIN {seq}
   lettura(sequenza);
   scrittura(sequenza)
END. {seq}
```

Non è stato ancora chiarito in quale fase la sequenza letta debba essere rovesciata. È infatti possibile memorizzare la sequenza nell'ordine in cui viene ricevuta e scriverla poi partendo dal fondo, oppure memorizzare la sequenza in ordine inverso e scriverla in `output` così come è stata memorizzata. Sceglieremo la seconda soluzione. Questa scelta è legata alla rappresentazione che adotteremo per il tipo `tiposequenza`.

Osserviamo che la richiesta di poter elaborare sequenze di lunghezza arbitraria, senza alcun limite prefissato, non consente l'uso di strutture statiche come gli array. Il problema può essere risolto facendo ricorso a *strutture dati dinamiche*, cioè strutture il cui numero di elementi può variare durante l'esecuzione. Tali strutture vengono realizzate organizzando, con opportuni collegamenti, variabili dinamiche.

In questo specifico caso, possiamo creare una variabile dinamica per rappresentare ciascun elemento della sequenza, collegando tra loro le variabili così create in una struttura a *lista*. La creazione di nuove variabili e il relativo collegamento alla lista avvengono dinamicamente man mano si leggono gli interi da input. Ad esempio, dopo avere letto la sequenza di interi `5 15 4 7`, si dovrà avere in memoria una struttura come quella riportata nella seguente figura, dove rappresentiamo con `NIL` la fine della lista:



La lista è costituita da quattro nodi con la medesima struttura: uno spazio per il numero intero da memorizzare nel nodo e un'indicazione che permette di accedere al nodo successivo. Quest'ultima informazione può essere rappresentata mediante un puntatore. Ciascun nodo sarà dunque un record costituito dai campi sopra indicati. Piú precisamente, introduciamo le seguenti definizioni di tipo:

```
TYPE
   tiposequenza = ^nodo;
   nodo = RECORD
          info: integer;
          pros: tiposequenza
      END;
```

Si osservi che la definizione è ciclica: il tipo `tiposequenza` è definito in termini del tipo `nodo` e viceversa. In questo caso il linguaggio Pascal permette di utilizzare un nome di tipo (`nodo`), prima di darne la definizione. Come già anticipato, la struttura dati del programma sarà costituita da una variabile di tipo `tiposequenza`:

```
VAR
   sequenza: tiposequenza;
```

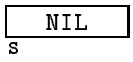
Esaminiamo ora come avviene la costruzione della lista. In particolare, sviluppiamo la procedura `lettura`.

Nella procedura, dopo avere inizializzato la lista come vuota, si effettua un ciclo, in cui viene letto di volta in volta un numero, che va inserito *all'inizio* della lista. Il ciclo ha termine quando viene letto `0`:

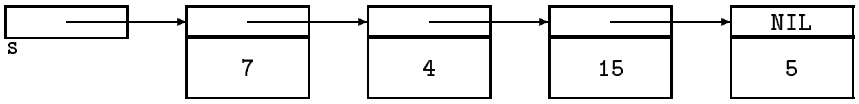
```
s := lista vuota
leggi un numero x
```

```
WHILE x <> 0 DO
   BEGIN
      inserisci x all'inizio della sequenza
      leggi un numero x
    END
```

L'inizializzazione avviene assegnando a `s` il valore `NIL`, per indicare che, inizialmente, la sequenza è vuota:



Esaminiamo ora come effettuare l'inserimento all'inizio della lista del valore contenuto in una variabile `x`. Supponiamo che `s` punti ad una lista già costruita come ad esempio



Le operazioni da effettuare sono:

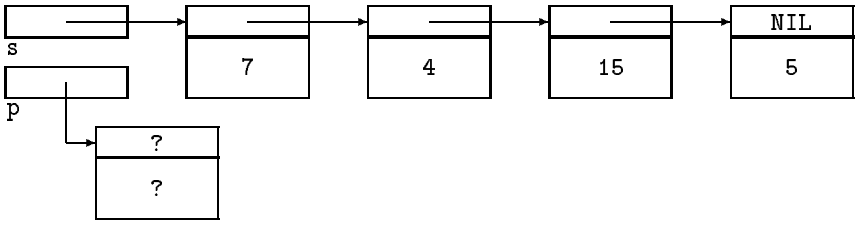
- creazione di un nuovo nodo;
- memorizzazione del valore della variabile `x` nel campo `info` del nuovo nodo;
- inserimento del nuovo nodo all'inizio della lista.

Dichiariamo localmente alla procedura `lettura` una variabile `p` di tipo `tiposequenza`, che utilizziamo per la creazione del nuovo nodo (si noti che le variabili `s` e `p` si trovano nello stack, i nodi della lista, invece, essendo variabili dinamiche, si trovano nello heap).

Per creare il nodo, è sufficiente scrivere:

```
new(p)
```

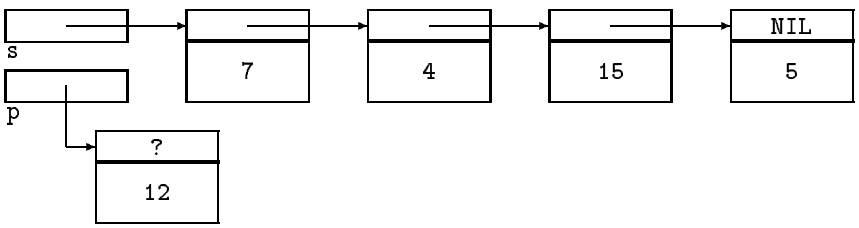
Dopo l'esecuzione di questa istruzione, la memoria conterrà:



Il nodo creato è accessibile, tramite il puntatore `p`, utilizzando il nome `p^`. Dunque, `p^` è di tipo `nodo`. Per assegnare il valore della variabile `x` al campo `info` è necessario utilizzare il selettore di campo, scrivendo:

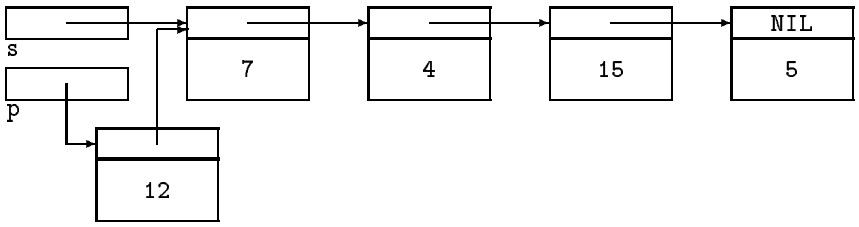
```
p^.info := x
```

Supponendo che la variabile `x` contenga il valore `12`, il contenuto della memoria diviene:

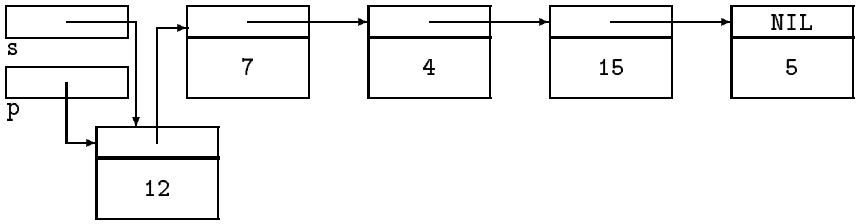


A questo punto occorre collegare il nuovo nodo alla lista già esistente. Questa operazione viene effettuata in due fasi:

- si fa puntare il campo `pros` del nuovo nodo all'inizio della lista esistente, mediante l'assegnamento `p^.pros := s`:



- si fa puntare l'inizio della lista, cioè il puntatore `s`, al nuovo nodo, mediante l'assegnamento `s := p`:



Osservando ora il contenuto della lista a partire da `s`, si ottiene la sequenza `12 7 4 15 5`. Il codice completo della procedura è dunque:

```
PROCEDURE lettura (VAR s: tiposequenza);

{riceve da input una sequenza di numeri e la memorizza, rovesciata, nel parametro s}

   VAR
      x: integer;
      p: tiposequenza;

BEGIN {lettura}
   {inizializzazione sequenza vuota}
   s := NIL;

   {lettura e memorizzazione sequenza}
   writeln('Inserire una sequenza di numeri interi (0 per terminare)');
   readln(x);
   WHILE x <> 0 DO
      BEGIN
         {inserisci x all'inizio della sequenza}
            {crea un nuovo nodo in cui memorizzare x}
         new(p);
         p^.info := x;

            {inserisci il nuovo nodo all'inizio della lista}
         p^.pros := s;
         s := p;
```

```
         {leggi il numero successivo}
         readln(x)
      END {while}
END; {lettura}
```

Sviluppiamo ora la procedura di scrittura. Compito di questa procedura è quello di scandire la lista (il cui puntatore iniziale è passato come parametro per valore) e di scriverne in `output` il contenuto.

La procedura può essere realizzata con un semplice ciclo:

```
WHILE la lista non e' vuota DO
   BEGIN
      scrivi l'informazione contenuta nel primo nodo
      sposta il puntatore iniziale sul nodo successivo
   END
```

Il primo nodo della lista è quello puntato da `s`, cioè il record `s^`. Per scriverne il contenuto è sufficiente utilizzare l'istruzione `write(s^.info)`.

Il nodo successivo al primo è quello puntato da `s^.pros`. Pertanto, per spostare il puntatore `s` sul nodo successivo al primo, possiamo scrivere `s := s^.pros`.

Il codice della procedura è dunque:

```
PROCEDURE scrittura (s: tiposequenza);

{scrive in output il contenuto della sequenza memorizzata in s}

BEGIN {scrittura}
   WHILE s <> NIL DO
      BEGIN
         writeln(s^.info);
         s := s^.pros
      END {while}
END; {scrittura}
```

Si noti che l'inserimento di un elemento in fondo a una lista e la scrittura del contenuto di una lista a partire dal fondo sono operazioni piú complicate rispetto a quelle utilizzate qui. Per questo motivo abbiamo scelto di sviluppare il programma memorizzando nella lista la sequenza già rovesciata. Per fare questo abbiamo effettuato gli inserimenti all'inizio della lista.

Riportiamo ora il codice completo del programma, in cui la procedura `scrittura` è modificata in modo che venga stampato un messaggio quando la sequenza è vuota e in modo che i numeri siano stampati sulla stessa riga.

```
PROGRAM seq (input, output);

   TYPE
      tiposequenza = ^nodo;
      nodo = RECORD
            info: integer;
            pros: tiposequenza
         END;

   VAR
      sequenza: tiposequenza;
```

```
PROCEDURE lettura (VAR s: tiposequenza);

{riceve da input una sequenza di numeri e la memorizza, rovesciata, nel parametro s}

   VAR
      x: integer;
      p: tiposequenza;

BEGIN {lettura}
   {inizializzazione sequenza vuota}
   s := NIL;

   {lettura e memorizzazione sequenza}
   writeln('Inserire una sequenza di numeri interi (0 per terminare)');
   readln(x);
   WHILE x <> 0 DO
      BEGIN
         {inserisci x all'inizio della sequenza}
            {crea un nuovo nodo in cui memorizzare x}
         new(p);
         p^.info := x;

            {inserisci il nuovo nodo all'inizio della lista}
         p^.pros := s;
         s := p;

         {leggi il numero successivo}
         readln(x)
      END {while}
END; {lettura}


PROCEDURE scrittura (s: tiposequenza);

{scrive in output il contenuto della sequenza memorizzata in s}

BEGIN {scrittura}
   IF s = NIL THEN
      writeln('La sequenza ricevuta e'' vuota')
   ELSE
      BEGIN
         write('La sequenza rovesciata e'' ');
         REPEAT
            {scrivi l'intero contenuto del record puntato da s}
            write(s^.info : 1, ' ');

           {sposta il puntatore sul record successivo}
            s := s^.pros
         UNTIL s = NIL;
         writeln
```

```
        END {else}
   END; {scrittura}



BEGIN {seq}
   lettura(sequenza);
   scrittura(sequenza)
END. {seq}
```

19.3 Liste

Utilizzando variabili dinamiche e puntatori è possibile, come abbiamo visto nell'esempio precedente, creare strutture a *lista*. Una lista viene realizzata mediante una collezione di record, che chiamiamo anche *nodi*, ognuno dei quali contiene, oltre alle informazioni da memorizzare, un puntatore al record successivo della lista. Si accede alla lista tramite il puntatore al primo elemento.

Per definire il tipo *lista di interi*, cioè una lista nei cui nodi sono memorizzati valori di tipo `integer`, possiamo scrivere:

```
TYPE
   tipolista = ^nodolista;
   nodolista = RECORD
         info: integer;
         pros: tipolista
      END;
```

Una variabile che venga dichiarata di `tipolista` potrà dunque puntare a una lista contenente numeri interi (o alla lista vuota).

Inserimento di un elemento all'inizio di una lista

Costruiamo una procedura di nome `InserInizio`, che ricevendo come parametri il puntatore ad una lista e un valore di tipo `integer`, inserisca all'inizio della lista indicata dal primo parametro un nuovo nodo, contente il valore specificato nel secondo parametro.

L'intestazione della procedura sarà:

```
PROCEDURE InserInizio (VAR l: tipolista; x: integer);
```

La tecnica per effettuare l'inserimento di un nodo all'inizio di una lista è stata esaminata nell'esempio precedente e consiste dei seguenti passi:

1. creazione di un nuovo nodo mediante un puntatore ausiliario;
2. memorizzazione delle informazioni nel nuovo nodo;
3. collegamento del nodo al resto della lista.

Per collegare il nuovo nodo alla lista già esistente occorre:

- collegare la lista precedentemente esistente al nuovo nodo (facendo puntare il campo `pros` del nuovo nodo al primo nodo della lista precedentemente esistente);
- spostare il puntatore iniziale della lista sul nuovo nodo.

Le operazioni sopra indicate sono effettuate dalla seguente procedura:

```
PROCEDURE InserInizio (VAR l: tipolista; x: integer);

{Inserisce il valore di x all'inizio della lista puntata da l}

   VAR
      p: tipolista;

BEGIN {InserInizio}
   new(p);
   p^.info := x;
   p^.pros := l;
   l := p
END; {InserInizio}
```

Scansione di una lista

La scansione di una lista consiste nell'esaminare tutti i nodi di una lista, dal primo all'ultimo, effettuando determinate operazioni su ciascun nodo, come ad esempio la scrittura in `output` delle informazioni contenute nel nodo.

Una procedura di scansione si basa su un ciclo come il seguente, in cui si esamina la lista partendo dal primo nodo:

```
WHILE non e' stata esaminata tutta la lista DO
   BEGIN
      esegui le operazioni sul nodo corrente
      spostati sul nodo successivo
   END
```

Dato un puntatore `l`, che punti inizialmente al primo nodo della lista, il ciclo precedente può essere riscritto come:

```
WHILE l <> NIL DO
   BEGIN
      esegui le operazioni sul nodo l^
      l := l^.pros
   END
```

Possiamo ad esempio usare lo schema di scansione per costruire la seguente procedura che riceva come parametro il puntatore ad una lista e scriva in `output` i valori contenuti in essa:

```
PROCEDURE ScriviLista (l: tipolista);

{Scrive in output il contenuto della lista puntata da l}
{versione iterativa}

BEGIN {ScriviLista}
   WHILE l <> NIL DO
      BEGIN
         writeln(l^.info);
         l := l^.pros
      END {while}
END; {Scrivilista}
```

Si osservi che il parametro formale `l` è passato per valore. Di conseguenza, al termine dell'esecuzione della procedura, il parametro attuale con il quale la procedura è stata chiamata continuerà a

puntare al primo elemento della lista, sebbene all'interno della procedura il valore di `l` sia stato modificato.

Ricerca di un elemento in una lista

Costruiamo ora una funzione che, ricevendo il puntatore al primo elemento di una lista e un valore x di tipo `integer`, restituisca il puntatore al primo nodo della lista contenente x, se presente, oppure `NIL`.

L'intestazione della funzione sarà:

```
FUNCTION trova (l: tipolista; x: integer): tipolista;
```

La funzione può essere realizzata effettuando una scansione della lista, che viene interrotta quando viene trovato l'elemento cercato. Per uscire dal ciclo quando si incontra nella lista il valore di `x`, potremmo modificare il ciclo di scansione scritto sopra, riscrivendolo come:

```
WHILE (l <> NIL) AND (l^.info <> x) DO
   l := l^.pros
```

Il ciclo appena scritto *è scorretto* e *può provocare problemi* in esecuzione. Infatti, se la condizione `l <> NIL` è falsa, cioè se `l` contiene `NIL`, la condizione `l^.info <> x` non ha alcun senso in quanto la variabile `l^` non esiste.[1]

Per risolvere questo problema (che ricorre spesso quando si trattano i puntatori) possiamo far uso di una variabile di tipo `boolean`, che indica quando si sia trovato l'elemento cercato. Il confronto tra il contenuto del nodo, `l^.info`, e la variabile `x` viene effettuato all'interno del ciclo, dove sicuramente `l` è diverso da `NIL`:

```
trovato := false;
WHILE (l <> NIL) AND NOT trovato DO
   IF l^.info = x THEN
      trovato := true
   ELSE
      l := l^.pros
```

In questo modo, se si esce dal ciclo avendo trovato l'elemento, cioè con la variabile `trovato` contenente `true`, `l` punta al nodo contenente l'elemento, altrimenti, se si esce perché la lista è finita e dunque l'elemento cercato non c'è, `l` punta a `NIL`. Dopo il ciclo è dunque sufficiente restituire all'ambiente chiamante il valore di `l`:

```
FUNCTION trova (l: tipolista; x: integer): tipolista;

{Restituisce il puntatore al primo nodo della lista l contenente il }
{valore di x, se presente, oppure NIL}
{versione iterativa}

   VAR
      trovato: boolean;
```

[1] Si conderi una condizione del tipo `c1 AND c2`, cioè l'AND di due condizioni `c1` e `c2`. Alcuni compilatori Pascal, generano il codice in modo tale che, quando `c1` risulta falsa, l'intera condizione venga subito valutata come falsa, senza calcolare il valore di `c2` (valutazione "cortocircuitata"). Altri compilatori, invece, generano il codice in modo che, in ogni caso, entrambe le condizioni siano valutate. In quest'ultimo caso, una condizione come `(x <> 0) AND (y DIV x = 1)` provocherà un errore in esecuzione (divisione per zero) quando `x` contiene zero. Per evitare questi errori ed ottenere programmi indipendenti dal compilatore utilizzato, è necessario scrivere le condizioni di un `AND` in modo tale che entrambe siano ben definite e non provochino effetti collaterali, come potrebbe ad esempio succedere richiamando funzioni che modifichino variabili globali (un discorso analogo vale per l'`OR` in cui, se la prima condizione risulta vera, si potrebbe evitare di valutare la seconda).

```
BEGIN {trova}
   trovato := false;
   WHILE (l <> NIL) AND NOT trovato DO
      IF l^.info = x THEN
         trovato := true
      ELSE
         l := l^.pros;
   trova := l
END; {trova}
```

Nel caso la lista sia *ordinata* in maniera non decrescente, la ricerca di un elemento può essere arrestata non appena si trovi un nodo contenente un valore maggiore o uguale a quello desiderato. In questo caso, la procedura di ricerca può essere costituita da un ciclo simile al precedente, in cui si ricerca il primo elemento maggiore o uguale al valore voluto, e da una fase in cui si determina il risultato da restituire. Il ciclo sarà:

```
finito := false;
WHILE (l <> NIL) AND NOT finito DO
   IF l^.info >= x THEN
      finito := true
   ELSE
      l := l^.pros
```

In uscita dal ciclo si deve restituire il valore del puntatore `l` nel caso la ricerca abbia avuto successo, cioè nel caso in cui si sia usciti dal ciclo con `finito` a `true`, avendo in `l` il puntatore ad un nodo contente lo stesso valore di `x`. Negli altri casi andrà restituito `NIL`.

Si potrebbe dunque scrivere

```
IF finito AND (l^.info = x) THEN
   trova := l
ELSE
   trova := NIL
```

Anche *questa scrittura è scorretta* e può provocare malfunzionamenti. Se infatti si è usciti dal ciclo avendo raggiunto la fine della lista (cioè con `l` uguale a `NIL`), il nodo `l^` non esiste. D'altra parte, se `finito` vale `true`, sicuramente `l` punta ad un nodo il cui contenuto va confrontato con il valore di `x`. Pertanto, possiamo riscrivere correttamente il precedente frammento di codice separando le due condizioni e valutandole in due strutture `IF` innestate:

```
IF finito THEN
   IF l^.info = x THEN
      trova := l
   ELSE
      trova := NIL
ELSE
   trova := NIL
```

Il codice completo della `FUNCTION` è:

```
FUNCTION trova (l: tipolista; x: integer): tipolista;

{Restituisce il puntatore al primo nodo della lista l contenente il}
{valore di x, se presente, oppure NIL}
{Si suppone che la lista sia ordinata in maniera non descrescente}
```

```
{versione iterativa}

   VAR
      finito: boolean;

BEGIN {trova}
   {ricerca il primo elemento maggiore o uguale a x}
   finito := false;
   WHILE (l <> NIL) AND NOT finito DO
      IF l^.info >= x THEN
         finito := true
      ELSE
         l := l^.pros;

   {restituisce il risultato}
   IF finito THEN
      IF l^.info = x THEN
         trova := l
      ELSE
         trova := NIL
   ELSE
      trova := NIL
END; {trova}
```

Esercizi

1. Per ognuno dei seguenti programmi, disegnare le variabili presenti nello stack e nello heap durante l'esecuzione, con i rispettivi contenuti. Indicare poi l'output prodotto.

   - ```
     PROGRAM p1 (output);
     VAR r, s: ^integer;
     BEGIN
        new (r);
        new (s);
        s^:= 10;
        r^:= 20;
        r:= s;
        s^:= 50;
        writeln (r^, s^)
     END.
     ```
   - ```
     PROGRAM p2 (output);
     VAR r, s: ^integer;
     BEGIN
        new (r);
        new (s);
        s^:= 10;
        r^:= 20;
        r^:= s^;
        s^:= 50;
        writeln (r^, s^)
     END.
     ```

- ```
  PROGRAM p3 (output);
  VAR r, s: ^integer;
  BEGIN
     new (r);
     new (s);
     s^:= 10;
     r^:= 20;
     r:= s;
     new (s);
     s^:= 50;
     writeln (r^, s^)
  END.
  ```

2. Scrivere l'output prodotto da ciascuno dei seguenti programmi su input `4`.

   - ```
     PROGRAM p1 (input, output);
        VAR
           p, q: ^integer;
           i, x: integer;
     BEGIN {p1}
        readln(x);
        new(p);
        new(q);
        p^ := -2;
        q^ := p^;
        FOR i := 1 TO x - 1 DO
           p^ := p^ + q^;
        writeln(p^)
     END. {p1}
     ```
   - ```
     PROGRAM p2 (input, output);
        VAR
           p, q: ^integer;
           i, x: integer;
     BEGIN {p2}
        readln(x);
        new(p);
        new(q);
        p^ := -2;
        q := p;
        FOR i := 1 TO x - 1 DO
           p^ := p^ + q^;
        writeln(p^)
     END. {p2}
     ```

3. Simulare "manualmente" l'esecuzione del programma `seq` su alcune sequenze scelte a piacere (tra cui la sequenza vuota, ottenibile inserendo subito `0`), disegnando l'evoluzione dello stack e dello heap. In particolare, si osservi che nella `PROCEDURE lettura` la variabile `p` è locale, mentre la variabile `s` è un parametro per riferimento, che si riferisce (nell'unica chiamta utilizzata) alla variabile `sequenza` del programma principale. Dunque le operazioni effettuate su `s` andranno in realtà ad operare sulla variabile globale `sequenza`, che conterrà sempre il puntatore all'inizio della sequenza. Nella `PROCEDURE scrittura`, il parametro `s` è invece passato per valore. In questo caso, al momento del passaggio dei parametri, il puntatore `sequenza`
     ```

(variabile globale) viene copiato nel puntatore `s` (variabile locale). Entrambi dunque punteranno alla stessa lista, i cui nodi si trovano nello heap. Gli assegnamenti `s := s^.pros`, effettuati nel corso della procedura, modificano la variabile locale `s`, facendola via via puntare ai vari nodi della lista. La variabile globale `sequenza` continua invece a puntare all'inizio della lista.

4. Simulare "manualmente" l'esecuzione dei sottoprogrammi presentati, evidenziando l'evoluzione dello stack e dello heap.

5. Costruire una procedura che data in ingresso una lista di interi, la trasformi nella lista ottenuta facendo scorrere circolarmente all'indiero tutti gli elementi: tutti gli elementi vengono spostati all'indietro di una posizione, eccetto il primo che passa in ultima posizione. Ad esempio, la lista contenente `54 23 654 12 26` dovrà essere trasformata nella lista contenente `23 654 12 26 54`.

6. La `FUNCTION trova` restituisce il puntatore al nodo contenente l'elemento cercato, oppure `NIL` nel caso l'elemento non sia presente. Costruire una `FUNCTION posizione`, che restituisca la posizione di un elemento in una lista, o zero se l'elemento non è presente. Se ad esempio la lista contiene `5 8 10 4 9`, e viene cercato `10`, la funzione dovrà restituire `3`.

7. (Dal tema d'esame del 31 gennaio 1997.) Si vuole costruire un ambiente per lo studio e la prova di sottoprogrammi che manipolano sequenze di interi. In base ad un menù, l'utente può scegliere di volta in volta quali operazioni effettuare manipolando una sequenza, inizialmente vuota. Le operazioni previste sono:

   (a) Inserimento di un nuovo elemento *all'inizio* della sequenza.

   (b) Stampa del contenuto della sequenza.

   (c) Ricerca di un elemento all'interno della sequenza: dato in ingresso un intero, il programma ne indica la posizione, se presente. Ad esempio, se la sequenza contiene `24 3 7 98 46 22` e l'utente inserisce per la ricerca `98`, il programma dovrà rispondere che l'elemento si trova in quarta posizione.

   (d) Rimozione di un elemento dalla sequenza: dato in ingresso un intero, il programma lo elimina dalla sequenza, se presente.

   (e) Scambio dell'elemento minimo della sequenza con quello massimo. Ad esempio, la sequenza contenente `24 3 7 98 46 22` dovrà essere trasformata nella sequenza contenente `24 98 7 3 46 22`.

   Il programma *deve essere privo* di vincoli sulla lunghezza massima della sequenza trattata e deve risultare facilmente espandibile nel caso si vogliano introdurre nuove funzionalità.