Lezione 25
18–19 gennaio 2000

Argomenti trattati
– Sottoprogrammi iterativi e ricorsivi per il trattamento di liste e alberi.

25.1 Esempi

In questa lezione presentiamo alcuni sottoprogrammi che trattano liste e alberi binari contenenti valori di tipo `integer`. In tutti gli esempi faremo riferimento alle seguenti definizioni di tipo:

```
TYPE
   tipolista = ^nodolista;       {puntatore a un nodo della lista}
   nodolista = RECORD            {nodo della lista}
                  info: integer; {informazione contenuta nel nodo}
                  pros: tipolista{puntatore all'elemento successivo della lista}
               END;

   tipoalbero = ^nodoalbero;
   nodoalbero = RECORD
                  info: integer;     {informazione contenuta nel nodo}
                  sx, dx: tipoalbero {puntatori ai sottoalberi sinistro e destro}
                END;
```

Eliminazione da una lista di tutti i valori pari

Vogliamo scrivere una procedura che riceva come parametro il puntatore ad una lista di interi ed elimini da tale lista tutti i nodi contenenti valori pari. Ad esempio, se la lista contiene inizialmente `2 8 1 6 4 3 2`, dopo l'esecuzione del sottoprogramma dovrà contenere `1 3`.

Scriviamo prima di tutto l'intestazione della procedura, che riceverà, come parametro per riferimento, il puntatore al primo elemento della lista:

```
PROCEDURE eliminapari (VAR l: tipolista);
```

In base alla definizione ricorsiva, una lista è o la lista vuota, oppure un elemento seguito da una lista. Stabiliamo le operazioni da compiere nei due casi.

- La *lista vuota* rimane vuota, quindi in questo caso non va eseguita alcuna operazione.
- La lista costituita da *un elemento seguito da una lista*, può essere trasformata come segue:
  - se l'elemento contiene un valore pari allora va eliminato;
  - in ogni caso vanno eliminati gli elementi pari nella lista che segue l'elemento.

  In questo caso possiamo dunque compiere le seguenti operazioni:

  ```
  elimina gli elementi pari dalla lista puntata da l^.pros
  IF il primo elemento e' pari THEN eliminalo dalla lista
  ```

  La prima operazione può essere realizzata mediante la chiamata ricorsiva

  ```
  eliminapari(l^.pros)
  ```

  Per eliminare il primo nodo dalla lista occorre far puntare `l` alla parte di lista che segue il primo elemento, cioè alla lista puntatata da `l^.pros`. Ciò può essere realizzato mediante l'assegnamento `l := l^.pros`. Per rilasciare l'area di memoria occupata dal nodo eliminato, si utilizza un puntatore ausiliario, su cui poi viene chiamata la procedura `dispose`.

Il codice completo della procedura costruita secondo lo schema indicato è:

```
PROCEDURE eliminapari (VAR l: tipolista);

{elimina dalla lista passata come parametro tutti i nodi contenenti valori pari}

   VAR
      q: tipolista;

BEGIN {eliminapari}
   IF l <> NIL THEN
      BEGIN
         eliminapari(l^.pros);
         IF l^.info MOD 2 = 0 THEN
            BEGIN
               q := l;
               l := l^.pros;
               dispose(q)
            END
      END
END; {eliminapari}
```

Scambio del valore minimo con il valore massimo

Vogliamo ora costruire una procedura che riceva il puntatore a una lista di interi e trasformi tale lista scambiando il valore minimo con il valore massimo. Ad esempio, la lista contenente `2 8 1 6 4 13 2` deve essere trasformata nella lista contenente `2 8 13 6 4 1 2`. Per semplicità supponiamo che il valore minimo e massimo appaiano nella lista solo una volta.

Scriviamo prima di tutto l'intestazione della procedura:

```
PROCEDURE scambiaMinMax (l: tipolista);
```

Si noti che, poiché la procedura deve modificare i dati contenuti nei nodi della lista, e non i puntatori, è sufficiente il passaggio del puntatore per valore.

La procedura può essere sviluppata in due fasi fondamentali: nella prima fase si scandisce la lista cercando il valore massimo e il valore minimo. Nella seconda fase si effettua lo scambio. Per semplificare la fase di scambio, nella fase di ricerca è utile memorizzare, oltre ai valori del massimo e del minimo, anche i puntatori ai due nodi che li contengono.

Dichiariamo dunque due variabili `max` e `min` di tipo `integer` destinate a contenere il valore del massimo e del minimo, e due variabili `pmax` e `pmin` di tipo `tipolista` destinate a contenere i puntatori ai nodi contenenti il massimo e il minimo. Nel caso di lista non vuota inizializziamo tali variabili con i valori corrispondenti al primo nodo della lista:

```
min := l^.info;
pmin := l;
max := l^.info;
pmax := l
```

La ricerca avviene partendo dal secondo elemento e confrontando, di volta in volta, i valori incontrati con `max` e `min`.

```
l := l^.pros;
WHILE l <> NIL DO
   BEGIN
      IF l^.info < min THEN
         BEGIN
            min := l^.info;
            pmin := l
         END;
      IF l^.info > max THEN
         BEGIN
            max := l^.info;
            pmax := l
         END;
      l := l^.pros
   END; {while}
```

Alla fine della ricerca è sufficiente porre il valore `min` nel nodo puntato da `pmax` e il valore `max` nel nodo puntato da `pmin`:

```
pmin^.info := max;
pmax^.info := min
```

Si noti che se il valore minimo o massimo sono presenti piú di una volta, con questo procedimento vengono scambiate le prime occorrenze dei due valori.

Nel caso di lista vuota la procedura non dovrà operare alcuna trasformazione. Pertanto racchiudiamo il codice precedente nel ramo `THEN` di un'istruzione `IF`:

```
PROCEDURE scambiaMinMax (l: tipolista);

{scambia nella lista l il valore minimo con il valore massimo}
{se il minimo o il massimo sono presenti piu' di una volta, sostituisce solo la}
{loro prima occorrenza}

   VAR
      min, max: integer;     {minimo e massimo}
      pmin, pmax: tipolista; {puntatori ai nodi contenenti minimo e massimo}
```

```
BEGIN {scambiaMinMax}
   IF l <> NIL THEN
      BEGIN

         {ricerca del minimo e del massimo}
         min := l^.info;
         pmin := l;
         max := l^.info;
         pmax := l;
         l := l^.pros;
         WHILE l <> NIL DO
            BEGIN
               IF l^.info < min THEN
                  BEGIN
                     min := l^.info;
                     pmin := l
                  END;
               IF l^.info > max THEN
                  BEGIN
                     max := l^.info;
                     pmax := l
                  END;
               l := l^.pros
            END; {while}

         {sostituzione}
         pmin^.info := max;
         pmax^.info := min
      END {if}
END; {scambiaMinMax}
```

Calcolo del massimo tra la somma degli elementi di posizione pari e la somma degli elementi di posizione dispari

Vogliamo costruire una funzione che riceva come parametro il puntatore a una lista di interi e restituisca il valore maggiore tra la somma degli elementi di posizione pari e la somma degli elementi di posizione dispari. Ad esempio, se la lista contiene `2 8 1 6 4 13 2`, la somma degli elementi che si trovano nelle posizioni pari è `27`, la somma di quelli nelle posizioni dispari è `9`. Pertanto la funzione dovrà restituire `27`.

L'intestazione della funzione sarà:

```
FUNCTION maxPosPariDispari (l: tipolista): integer;
```

La funzione sarà costituita da un ciclo principale in cui si attraversa la lista, del tipo:

```
WHILE lista non e' finita DO
BEGIN
   operazioni su elemento corrente
   spostamento su elemento successivo
END
```

Le operazioni da effettuare dipendono dalla posizione (pari o dispari) dell'elemento. Raffiniamo dunque il ciclo precedente introducendo una variabile `pospari` di tipo `boolean`, che indica se l'elemento considerato è in posizione pari o dispari:

```
pospari := false
WHILE lista non e' finita DO
BEGIN
   IF pospari
      THEN
         aggiungi alla somma degli elementi di posizione pari
         l'elemento corrente
      ELSE
         aggiungi alla somma degli elementi di posizione dispari
         l'elemento corrente
   spostamento su elemento successivo
   pospari := NOT pospari
END
```

Mediante l'assegnamento `pospari := NOT pospari`, la variabile `pospari` assume alternativamente valori `false` e `true`, nelle varie iterazioni.

In uscita dal ciclo è sufficiente selezionare il valore piú grande tra la somma dei pari e dei dispari e restituirlo all'ambiente chiamante:

```
FUNCTION maxPosPariDispari (l: tipolista): integer;

{restituisce il massimo tra la somma degli elementi di posto pari e la somma}
{degli elementi di posto dispari della lista l}

   VAR
      pospari: boolean; {true quando si esaminano elementi di posto pari}
      sommapari, sommadispari: integer;

BEGIN {maxPosPariDispari}
   {inizializzazioni}
   sommapari := 0;
   sommadispari := 0;
   pospari := false;

   {calcolo somme}
   WHILE l <> NIL DO
      BEGIN
         IF pospari THEN
            sommapari := sommapari + l^.info
         ELSE
            sommadispari := sommadispari + l^.info;
         l := l^.pros;
         pospari := NOT pospari
      END; {while}

   {determina il massimo tra le due somme e lo restituisce}
   IF sommapari > sommadispari THEN
      maxPosPariDispari := sommapari
   ELSE
      maxPosPariDispari := sommadispari
END; {maxPosPariDispari}
```

Calcolo della profondità di una albero binario

Ricordiamo che in un albero binario il livello della radice è 1, mentre il livello dei figli di un nodo di livello i è $i+1$. La profondità dell'albero è il massimo livello dei suoi nodi (vedere Lezione 21).

Vogliamo costruire una funzione che riceva come parametro il puntatore alla radice di un albero binario contenente numeri interi e restituisca la profondità dell'albero stesso.

L'intestazione della funzione sarà:

```
FUNCTION profondita (t: tipoalbero): integer;
```

Per scrivere il codice individuiamo una regola ricorsiva per il calcolo della profondità.

- L'*albero vuoto*, essendo privo di nodi, ha profondità zero.
- La profondità dell'albero costituito da una *radice* e due *sottoalberi* T_{sx} e T_{dx} è ottenibile aggiungendo uno alla maggiore tra le profondità di T_{sx} e T_{dx}.

In base a queste regole è immediato scrivere la seguente funzione:

```
FUNCTION profondita (t: tipoalbero): integer;

{restituisce la profondita' dell'albero t}

   VAR
      psx, pdx: integer;

BEGIN {profondita}
   IF t = NIL THEN
      profondita := 0
   ELSE
      BEGIN
         psx := profondita(t^.sx);
         pdx := profondita(t^.dx);
         IF psx < pdx THEN
            profondita := pdx + 1
         ELSE
            profondita := psx + 1
      END
END; {profondita}
```

Calcolo della somma dei valori contenuti in un albero binario di interi

Vogliamo costruire una funzione che riceva come parametro il puntatore a un albero binario contenente numeri interi e restituisca come risultato la somma dei valori contenuti dei nodi dell'albero.

L'intestazione sarà:

```
FUNCTION somma (t: tipoalbero): integer;
```

Anche in questo caso scriviamo il codice basandoci sulla defizione ricorsiva di albero:

- L'albero vuoto ha somma zero.
- Nel caso di un albero non vuoto la somma è data dalla somma del sottoalbero sinistro, piú la somma del sottoalbero destro, piú il valore contenuto nella radice.

In base al precedente schema, otteniamo il seguente codice:

```
FUNCTION somma (t: tipoalbero): integer;

{restituisce la somma dei valori contenuti nell'albero t}

BEGIN {somma}
   IF t = NIL THEN
      somma := 0
   ELSE
      somma := somma(t^.sx) + somma(t^.dx) + t^.info
END; {somma}
```

Calcolo della somma delle foglie di un albero binario di interi

Vogliamo ora costruire una funzione che restituisca la somma dei soli valori contenuti nelle foglie dell'albero.

L'intestazione sarà:

```
FUNCTION sommafoglie (t: tipoalbero): integer;
```

Di nuovo, partiamo dalla definizione ricorsiva di albero:

- L'albero vuoto non ha foglie; dunque la somma zero.
- Nel caso di un albero non vuoto dobbiamo distinguere due casi:
  - se *la radice è una foglia* allora la somma è data semplicemente dal valore della radice;
  - se *la radice non è una foglia* allora la somma è data dalla somma delle foglie del sottoalbero sinistro piú la somma delle foglie del sottoalbero destro.

La procedura, costruita secondo lo schema precedente, è:

```
FUNCTION sommafoglie (t: tipoalbero): integer;

{restituisce la somma delle foglie dell'albero t}

BEGIN {sommafoglie}
   IF t = NIL THEN
      sommafoglie := 0
   ELSE IF (t^.sx = NIL) AND (t^.dx = NIL) THEN
      sommafoglie := t^.info
   ELSE
      sommafoglie := sommafoglie(t^.sx) + sommafoglie(t^.dx)
END; {sommafoglie}
```

Calcolo della media dei valori contenuti in un albero binario

Vogliamo costruire una procedura che riceva come parametro il puntatore a un albero binario di interi e scriva in `output` la media dei valori contenuti nell'albero.

Il sottoprogramma avrà dunque la seguente intestazione:

```
PROCEDURE media (t: tipoalbero);
```

Il calcolo della media non è esprimibile direttamente in maniera ricorsiva. Per ottenere la media dell'albero occorre calcolare la somma dei valori contenuti nei nodi e il numero dei nodi.

Costruiamo prima di tutto una procedura ricorsiva che calcoli queste quantità. La procedura riceve tre parametri: il puntatore all'albero e due variabili di tipo `integer`. Dopo l'esecuzione, nel

secondo parametro sarà disponibile il valore della somma e nel terzo il numero dei nodi. Utilizzando il solito schema ricorsivo, la procedura è esprimibile come:

```
PROCEDURE m (t: tipoalbero; VAR s, n: integer);

{restituisce nel secondo e nel terzo parametro la somma e il numero di nodi}
{dell'albero t}

   VAR
      s1, s2, n1, n2: integer;

BEGIN {m}
   IF t = NIL THEN
      BEGIN
         s := 0;
         n := 0
      END
   ELSE
      BEGIN
         m(t^.sx, s1, n1);
         m(t^.dx, s2, n2);
         s := s1 + s2 + t^.info;
         n := n1 + n2 + 1
      END
END; {m}
```

La procedura `media` utilizza `m` per calcolare somma e numero dei nodi, effettua dunque un test (per distinguere il caso di albero vuoto), calcola e visualizza la media:

```
PROCEDURE media (t: tipoalbero);

{visualizza la media dei valori contenuti nell'albero t}

   VAR
      somma, nnodi: integer;

   PROCEDURE m (t: tipoalbero; VAR s, n: integer);

   {restituisce nel secondo e nel terzo parametro la somma e il numero di nodi}
   {dell'albero t}

      VAR
         s1, s2, n1, n2: integer;

   BEGIN {m}
      IF t = NIL THEN
         BEGIN
            s := 0;
            n := 0
         END
      ELSE
         BEGIN
            m(t^.sx, s1, n1);
```

```
            m(t^.dx, s2, n2);
            s := s1 + s2 + t^.info;
            n := n1 + n2 + 1
         END
   END; {m}

BEGIN {media}
   m(t, somma, nnodi);
   IF nnodi > 0 THEN
      writeln('La media e'' ', somma / nnodi : 8 : 2)
   ELSE
      writeln('L''albero e'' vuoto')
END; {media}
```

Visita con livelli

Presentiamo ora una procedura che visualizzi il contenuto di un albero binario, passato come parametro, utilizzando la visita in ordine simmetrico. La procedura deve anche indicare, accanto al valore di ogni nodo, il livello a cui si trova.

La procedura ha come intestazione:

```
PROCEDURE scriviConLivelli (a: tipoalbero);
```

Per gestire i livelli, la visualizzazione viene effettuata da una procedura ricorsiva piú interna, con due parametri:

```
PROCEDURE v (t: tipoalbero; liv: integer);
```

Il primo parametro di `v` è il puntatore alla radice del sottoalbero da visualizzare; il secondo parametro è il livello a cui si trova la radice. Quando si effettua una chiamata ricorsiva (cioè si va a considerare un sottoalbero con radice a livello `liv + 1`) è sufficiente passare come secondo parametro `liv + 1`.

```
PROCEDURE scriviConLivelli (a: tipoalbero);

{visualizza i valori contenuti nei nodi dell'albero, indicando il livello di ciascun nodo}

   PROCEDURE v (t: tipoalbero; liv: integer);
   BEGIN
      IF t <> NIL THEN
         BEGIN
            v(t^.sx, liv + 1);
            writeln(t^.info : 10, liv : 10);
            v(t^.dx, liv + 1)
         END
   END;

BEGIN
   writeln('Valore' : 10, 'Livello' : 10);
   v(a, 1)
END;
```

Immagine speculare di un albero

Vogliamo scrivere ora una procedura che riceva come parametro il puntatore a un albero binario e trasformi tale albero nella sua immagine speculare, cioè nell'albero ottenuto scambiando ad ogni livello il sottoalbero sinistro con il sottoalbero destro.

Anche in questo caso conviene procedere basandosi sulla definizione ricorsiva di albero.

- L'*albero vuoto* resta immutato.
- L'*albero non vuoto*, costituito cioè da una radice, da un sottoalbero sinistro e un sottoalbero destro viene trasformato come segue:
  - la radice resta immutata;
  - il nuovo sottoalbero sinistro sarà l'immagine speculare del vecchio sottoalbero destro;
  - il nuovo sottoalbero destro sarà l'immagine speculare del vecchio sottoalbero sinisto.

  Dunque occorre calcolare ricorsivamente le immagini speculari dei due sottoalberi, scambiandoli poi tra loro.

Il codice della procedura costruita seguendo questo schema è:

```
PROCEDURE specchio (t: tipoalbero);

   VAR
      q: tipoalbero;

{trasforma t nella sua immagine speculare}

BEGIN {specchio}
   IF t <> NIL THEN
      BEGIN
         specchio(t^.sx);
         specchio(t^.dx);

         {scambia sottoalberi}
         q := t^.sx;
         t^.sx := t^.dx;
         t^.dx := q
      END
END; {specchio}
```