Lista, pila, coda.

12.1 Introduzione.

I tipi che analizzeremo in questo capitolo non sono direttamente disponibili in Pascal. Dovremo allora realizzarli definendo, con gli strumenti del nostro linguaggio, strutture dati opportune ed algoritmi che implementino le operazioni elementari.

12.2 Lista.

Il concetto di lista compare spesso in matematica inteso come sequenza ordinata di elementi. Tutti gli elementi, escluso l'ultimo, hanno un successivo, così come tutti gli elementi, escluso il primo, hanno un precedente. In generale una lista può essere vuota.
In matematica la lista è rappresentata come una successione a1, a2, ... an di elementi: l'ordinamento è rappresentato dall'ordine con cui compaiono gli elementi nella successione.

12.2.1 Operazioni.

Le operazioni che si devono poter compiere su una lista sono:

1. inserimento di un nuovo elemento nella lista in una determinata posizione. Sia a1, a2, a3, a4, a5, a6 la successione degli elementi di una lista. Se inseriamo b alla posizione 3 la nuova lista sarà a1, a2, b, a3, a4, a5, a6

2. cancellazione dell'elemento di posizione p.
Sia a1, a2, a3, a4, a5, a6 una lista di elementi. Se cancelliamo l'elemento 2 la nuova lista sarà a1, a3, a4, a5, a6.

3. ricerca dell'elemento di posizione p.
Sia a1, a2, a3, a4, a5, a6 la successione degli elementi di una lista. Data una posizione p si deve poter conoscere l'elemento ap.

4. stampa di tutti gli elementi della lista.
Data una lista si deve poter conoscere tutti gli elementi che la compongono.

12.3 Realizzazione.

Il Pascal, a differenza di altri linguaggi, non fornisce il tipo lista. Risolviamo allora il problema della costruzione del tipo lista definendo con precisione una struttura dati e alcune funzioni che permettano di realizzare le operazioni sulle liste.

12.3.1 Lista realizzata con i vettori.

Una struttura dati che si può utilizzare per realizzare la lista è il vettore. La lista a0, a1, a2, ..., an può essere memorizzata in un vettore a, ponendo a[0]=a0, a[1]=a1, a[2]=a2, ..., a[n]=an.

Il vettore è così definito:

```
TYPE
     lista=RECORD
          d:ARRAY[0..dim] OF tipo_elemento;
          n:INTEGER;
     END;
     posizione=INTEGER;
```

dove

- n indica il numero degli elementi del vettore a effettivamente occupati
- tipo_elemento è il tipo del singolo elemento che compare nella lista
- posizione è il tipo dell'indice della lista

Per realizzare le operazioni elementari sulle liste implementiamo alcune funzioni che, avuto in ingresso la lista, compiono le operazioni richieste. Le operazioni elementari sono:

- creazione: creare la lista vuota significa porre il campo n uguale a zero.

```
PROCEDURE l_crea( VAR l:lista )
BEGIN
     l.n:=0;
END;
```

-inserimento: per poter inserire un nuovo elemento nella lista occorre indicare la lista, l'elemento che deve essere inserito e la posizione dove inserirlo.

```
PROCEDURE l_inserisci( VAR l:lista; x:tipo_elemento;
p:posizione );
{ inserisce l'elemento x alla posizione p della lista l }
VAR
     i:posizione;
BEGIN
     IF (p<0) OR (p>l.n) THEN
           errore( 'inserisci: indice fuori lista' );
     FOR i:=l.n DOWNTO p+1 DO      { libera la posizione p }
          l.d[i]:=l.d[i-1];
     l.d[p]:=x;              {   inserisci l'elemento nuovo }
     l.n:=l.n+1               {  aggiorna numero elementi della
lista }
END;
```

- cancellazione: per cancellare un elemento dalla lista occorre indicarne la posizione nella lista.

```
PROCEDURE l_cancella( VAR l:lista; p:posizione );
{ cancella l'elemento di posizione p dalla lista l }
VAR
     i:posizione;
BEGIN
     IF (p<0) OR (p>=l.n) THEN
           errore( 'l_cancella: indice fuori lista' );
     FOR i:=p TO l.n-2 DO { ricopri l'elemento di posizione p }
          l.d[i]:=l.d[i+1];
     l.n:=l.n-1;                  { aggiorna numero elementi  }
END;
```

- ricerca di un elemento: l'elemento in una lista è individuato dalla sua posizione.

```
FUNCTION l_elemento( l:lista; p:posizione ):tipo_elemento;
{ restituisce l'elemento di posizione p della lista l }
BEGIN
     IF (p<0) OR (p>=l.n) THEN errore( 'l_elemento: indice
fuori lista' );
     l_elemento:=l.d[p];
END;
```

- percorrere tutta la lista: sarà necessario poter percorrere tutta la lista sia per ricercare un determinato elemento, sia per stamparla. Sviluppiamo allora tre funzioni elementari che ci permetteranno di scandire tutti gli elementi della lista e precisamente:

- primo: è una funzione che restituisce l'indice del primo elemento della lista.

```
FUNCTION l_primo( l:lista ):posizione;
{ restituisce la posizione del primo elemento della lista l }
BEGIN
     l_primo:=0
END;
```

- ultimo: restituisce la posizione dell'elemento successivo all'ultimo.

```
FUNCTION l_ultimo( l:lista ):posizione;
{ restituisce la posizione dell'elemento successivo all'ultimo
nella lista l }
BEGIN
     l_ultimo:=l.n
END;
```

- successivo: data una lista ed una posizione restituisce la posizione dell'elemento successivo.

```
FUNCTION l_successivo( l:lista; p:posizione  ):posizione;
{ restituisce la posizione successiva  a p }
BEGIN
     l_successivo:=p+1
END;
```

Esempio 1: gestire una sequenza di numeri.

Utilizzando la lista la struttura dati può essere così definita:

STRUTTURA DATI:

```
CONST
     dim=100;
TYPE
     tipo_elemento=REAL;

     lista=RECORD
          d:ARRAY[0..dim] OF tipo_elemento;
          n:INTEGER;
     END;
     posizione=INTEGER;

     sequenza=lista;
VAR
     s:sequenza;          { lista dei numeri }
```

La sequenza s potrà contenere 100 elementi al massimo.

ALGORITMO:

Utilizzando le funzioni elementari definite sulla lista realizziamo la stampa e l'inserimento ordinato.

- stampa: per stampare gli elementi della sequenza occorre conoscere tutti gli elementi della lista partendo dal primo.

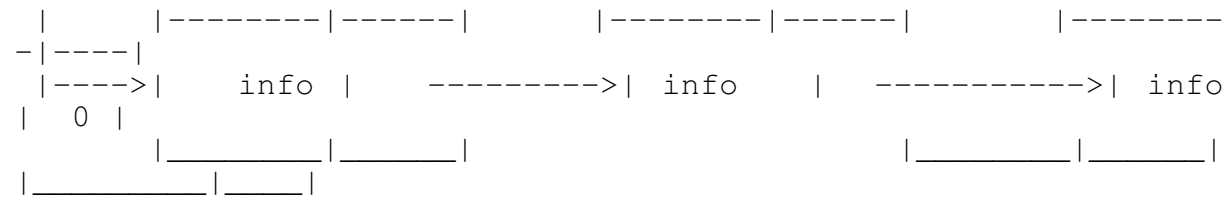
```
PROCEDURE stampa( s:sequenza );
VAR
     i:posizione;
BEGIN
     i:=l_primo( s );
     WHILE i<>l_ultimo( s ) DO BEGIN
          WRITELN( l_elemento(s, i) );
          i:=l_successivo( s, i )
     END
END;
```

- inserimento ordinato: supponendo che la sequenza sia ordinata in ordine crescente si dovrà ricercare la posizione giusta nella quale inserire il nuovo elemento.

```
FUNCTION ricerca( VAR s:sequenza; d:tipo_elemento ):posizione;
VAR
     p   :posizione;
     stop:BOOLEAN;
BEGIN
     p:=l_primo(s); stop:=FALSE;
     WHILE p<>l_ultimo(s) AND NOT stop DO
          IF l_elemento(s, p)<=d THEN p:=l_successivo(s, p)
          ELSE alt:=true;
     ricerca:=p
END;

PROCEDURE inserimento( VAR s:sequenza; d:tipo_elemento );
BEGIN
     l_inserisci( s, d, ricerca(s, d) );
END;
```

12.3.2 Realizzazione con le variabili dinamiche.

Una lista può essere rappresentata graficamente così:



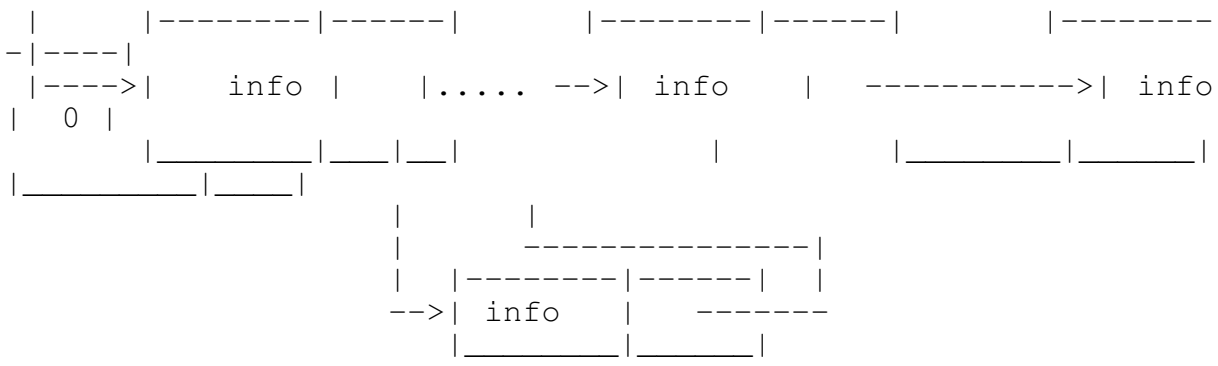
dove

- root è un puntatore che indica la posizione del primo elemento della lista
- ogni elemento della lista è diviso in due parti: l'informazione e un puntatore all'elemento successivo
- l'ultimo elemento contiene un segnale di fine lista

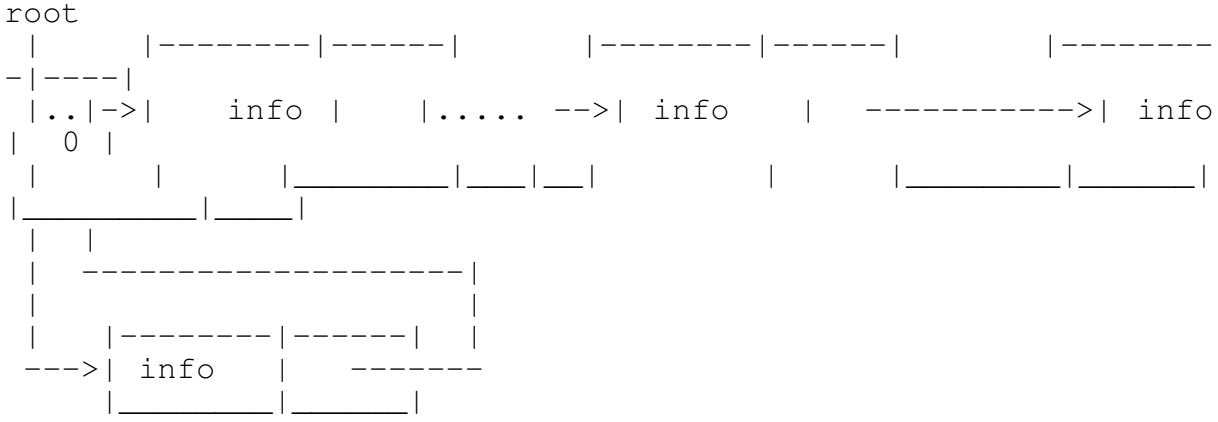
L'inserimento di un elemento ad una posizione p si realizza modificando il puntatore dell'elemento precedente.

Esempio: inserire un elemento alla seconda posizione.

Conoscendo la posizione del primo elemento si modifica il puntatore che indica l'elemento successivo facendo riferimento al nuovo elemento.



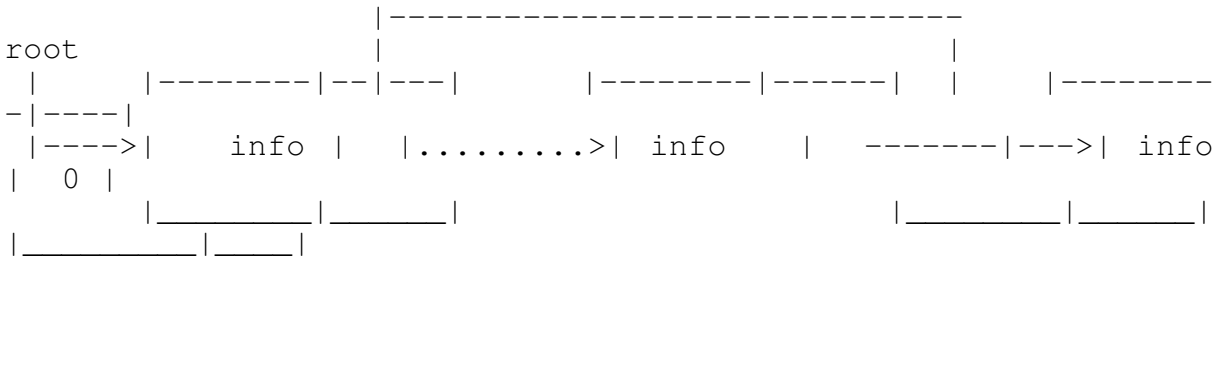
Tutti gli elementi hanno un precedente escluso il primo. L'inserimento in testa alla lista comporta allora la modifica di root.



La cancellazione di un elemento di posizione p comporta la modifica del puntatore dell'elemento precedente.

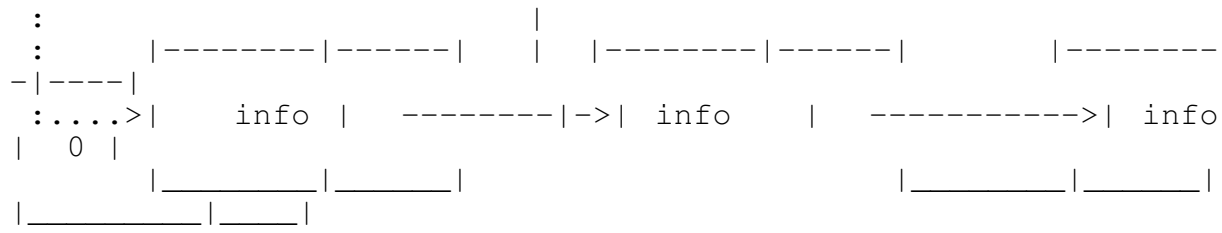
Esempio: cancellare il secondo elemento di una lista.

Conoscendo la posizione del primo si modifica il suo riferimento al successivo.



Il secondo elemento non sarà più raggiungibile da root.

Se l'elemento da cancellare è il primo si deve modificare root.



Possiamo realizzare tale struttura dati utilizzando per i singoli elementi della lista un record con due campi: il primo conterrà il dato, il secondo sarà una variabile di tipo puntatore che indicherà la posizione dell'elemento successivo. Il tipo lista sarà un puntatore al primo record. Da ogni record si potrà raggiungere il successivo attraverso il campo puntatore.

```
STRUTTURA DATI:
TYPE
     ptelemento=^elemento;
     elemento=RECORD
          d   :tipo_elemento;
          next:ptelemento;
     END;

     posizione=ptelemento;
     lista    =ptelemento;

VAR
     root:lista;
```

Per realizzare completamente il tipo lista dobbiamo implementare le operazioni elementari descritte in precedenza.

a. creazione: creare la lista vuota significa inizializzare la variabile statica che indica il primo elemento della lista.

```
PROCEDURE l_crea( VAR l:lista )
BEGIN
     l:=NIL
END;
```

b.inserimento: per poter inserire un nuovo elemento nella lista occorre indicare la lista, l'elemento che deve essere inserito e la posizione dove inserirlo. Negli esempi grafici abbiamo visto come per operare sull'elemento p sia necessario conoscere la posizione dell'elemento p-1. Allora facciamo l'ipotesi che la posizione indicata alla procedura sia quella dell'elemento

su cui si devono apportare modifiche. Se la posizione indicata è NIL si opererà direttamente sulla radice.

```
PROCEDURE   l_inserisci(   VAR   l:lista;   x:tipo_elemento;
p:posizione );
{ inserisce l'elemento x alla posizione p della lista l }
VAR
     i:posizione;
BEGIN
     NEW( i );
     i^.d:=x;                 { prepara il dato  }

     IF p=NIL THEN BEGIN   { se deve essere inserito in testa }
          i^.next:=l;
          l:=i             { bisogna modificare root          }
     END
     ELSE BEGIN
               i^.next:=p^.next;{altrimenti bisogna modificare }
               p^.next:=i      { il campo next dell'elemeto p  }
     END;
END;
```

c. cancellazione: per cancellare un elemento dalla lista occorre indicarne la posizione nella lista. Ricordiamo che la posizione sarà quella dell'elemento su cui si dovrà operare: NIL se l'elemento da cancellare è il primo.

```
PROCEDURE l_cancella( VAR l:lista; p:posizione );
{ cancella l'elemento di posizione p dalla lista l }
VAR
     i:posizione;
BEGIN
     IF p=NIL THEN BEGIN    { se si deve cancellare il primo  }
          i:=l;
          l:=l^.next               { modifica root }
     END
     ELSE BEGIN
           i:=p^.next;
           p^.next:=i^.next{altrimenti modifica il campo next }
     END;
     DISPOSE( i );
END;
```

d. ricerca di un elemento: l'elemento in una lista è individuato dalla sua posizione. Se p=NIL è il primo della lista altrimenti è il successivo a quello indicato.

```
FUNCTION l_elemento( l:lista; p:posizione ):tipo_elemento;
{ restituisce l'elemento di posizione p della lista l }
BEGIN
     IF p=NIL THEN                 { se è il primo }
        l_elemento:=l^.d
     ELSE
        l_elemento:=p^.next^.d { l'elemento successivo a p }
```

```
END;
```

e. percorrere tutta la lista: per percorrere tutti gli elementi della lista sono necessarie tre funzioni elementari e precisamente:

e.1 primo: è una funzione che restituisce l'indice del primo elemento della lista. Nella nostra realizzazione restituisce sempre NIL

```
FUNCTION l_primo( l:lista ):posizione;
{ restituisce la posizione del primo elemento della lista l }
BEGIN
     l_primo:=NIL
END;
```

e.2 ultimo: restituisce la posizione dell'elemento successivo all'ultimo.

```
FUNCTION l_ultimo( l:lista ):posizione;
{ restituisce la posizione dell'elemento successivo all'ultimo
nella lista l }
VAR
     i:posizione;
BEGIN
     i:=l; l_ultimo:=NIL;
     WHILE i<>NIL DO BEGIN
          l_ultimo:=i;
          i:=i^.next;
     END;
END;
```

La posizione dell'elemento successivo all'ultimo è l'indirizzo dell'elemento che ha il campo next=NIL.

e.3 successivo: data una lista ed una posizione restituisce la posizione dell'elemento successivo.

```
FUNCTION l_successivo( l:lista; p:posizione  ):posizione;
{ restituisce la posizione successiva  a p }
BEGIN
     IF p=NIL THEN l_successivo:=l   { il successivo al primo è
root }
     ELSE l_successivo:=p^.next;
END;
```

Esempio 2: gestire una sequenza di numeri.

Il problema è identico al precedente. Realizzeremo la lista utilizzando le variabili dinamiche.

STRUTTURA DATI:

```
TYPE
```

```
     tipo_elemento=REAL;

     ptelemento=^elemento;
     elemento=RECORD
          d   :tipo_elemento;
          next:ptelemento;
     END;

     posizione=ptelemento;
     lista    =ptelemento;
     sequenza =lista;
VAR
     s:sequenza;          { lista dei numeri }
```

ALGORITMO:

Le procedure di stampa e di inserimento ordinato sono quelle dell'esempio 1.

12.4 Considerazioni.

Abbiamo presentato due realizzazioni del tipo lista in Pascal. Ovviamente non sono le sole possibili, ma prima di presentarne altre confrontiamole attraverso l'analisi della complessità delle operazioni elementari.
Per quanto riguarda la rappresentazione 12.3.1 possiamo osservare che le funzioni di creazione, primo, successivo, ultimo ed elemento hanno complessità costante. L'inserimento e la cancellazione hanno complessità media e massima lineare poichè è necessario spostare, in caso di operazione su un elemento, tutti gli elementi successivi.

Per quanto riguarda la realizzazione 12.3.2 possiamo osservare che tutte le operazioni elementari escluso ultimo hanno complessità costante. ultimo ha complessità minima, media e massima lineari. Per liste con molti elementi, la seconda realizzazione è migliore della prima se si riuscisse a far diventare costante anche ultimo.

Possiamo ancora osservare come per utilizzare i vettori sia necessario conoscere in precedenza il numero massimo di elementi che dovranno essere gestiti, mentre con le variabili dinamiche il numero è dato dalla versione di Pascal utilizzato.

12.5 Nuove implementazioni.

Le nuove rappresentazioni che forniremo hanno tutte come riferimento l'implementazione 12.3.2.

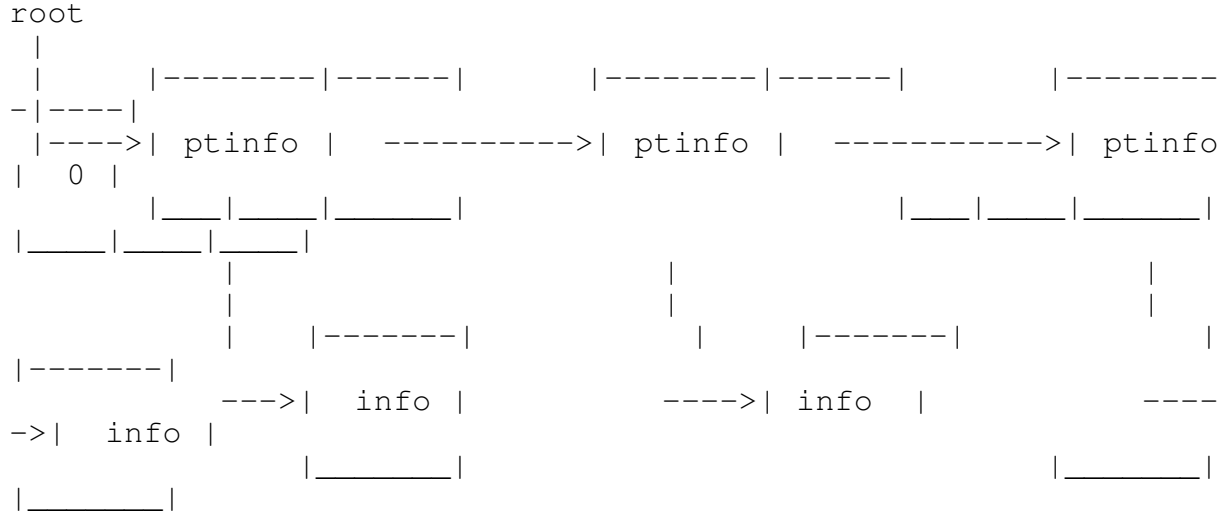
12.5.1 Lista di puntatori.
Tra le funzioni elementari che abbiamo sviluppato esiste la l_elemento che restituisce il dato di una data posizione. Se il dato non è di tipo scalare ma strutturato la funzione l_elemento è scorretta non potendo una funzione restituire una

variabile strutturata. A questo inconveniente si può ovviare facilmente modificando la struttura dati.

Supponiamo di avere una lista di persone caratterizzate da un cognome e un nome. La struttura opportuna per memorizzare il tutto è il record. Allora avremo:

```
TYPE
     str30=PACKED ARRAY[1..30] OF CHAR;
     dato=RECORD
          cognome,
          nome   :str30;
     END;
    tipo_elemento=^dato;{ tipo_elemento è un puntatore a dato }

     ptelemento=^elemento;
     elemento=RECORD
          d   :tipo_elemento;
          next:ptelemento;
     END;

     posizione=ptelemento;
     lista    =ptelemento;

VAR
     l:lista;
```

Possiamo rappresentare graficamente la lista in questo modo:



Tutte le primitive implementate per 13.3.2 operano anche per questa struttura: l'unica che dovrà essere modificata sarà la primitiva l_cancella che dovrà cancellare anche il dato:

```
PROCEDURE l_cancella( VAR l:lista; p:posizione );
{ cancella l'elemento di posizione p dalla lista l }
VAR
     i:posizione;
BEGIN
     IF p=NIL THEN BEGIN
          i:=l; l:=l^.next
     END
     ELSE BEGIN
               i:=p^.next; p^.next:=i^.next
     END;
     DISPOSE( i^.d );              { elimina il dato }
     DISPOSE( i );
END;
```

Esempio 3: scrivere una procedura che legga dall'esterno dei nomi e li inserisca nella lista l alla prima posizione.

STRUTTURA DATI:

La struttura dati è quella presentata all'inizio del paragrafo.

ALGORITMO:

Poichè la lista utilizza come dati puntatori a record dovremo creare una variabile dinamica per ogni dato da inserire nella lista.

```
PROCEDURE carica_lista( VAR l:lista );
VAR
     e:tipo_elemento;
BEGIN
     NEW( e );           { crea il dato per contenere cognome e
nome }
     WRITE('Cognome ');
     READLN( e^.cognome );
     WHILE e^.cognome <> '*' DO BEGIN    { cognome=* --> fine
inserimento }
          WRITE('Nome ');
          READLN( e^.nome );
          l_inserisci( l, e, l_primo(l) );
          NEW( e );                   { dopo ogni inserimento è
necessario  }
                                      {  allocare  nuovo  spazio
per il nuovo  }
                                      {              inserimento
}
          WRITE('Cognome ');
          READLN( e^.cognome );
     END;
     DISPOSE( e );
END;
```

Esempio 4: sempre con la stessa struttura stampiamo tutti gli elementi della lista.

```
PROCEDURE stampa( l:lista );
VAR
     i:posizione;
BEGIN
     i:=l_primo( l );
     WHILE i<>l_ultimo( l ) DO BEGIN
          WRITELN(  l_elemento(l,  i)^.cognome,  l_elemento(l,
i)^.nome );
          i:=l_successivo( s, i )
     END
END;
```

12.5.2 Modifica della radice.

Nel paragrafo 12.4 abbiamo studiato la complessità degli algoritmi relativi alla implementazione 12.3.2, scoprendo che l'unica funzione che non ha complessità costante è l_ultimo. Possiamo allora modificare la struttura in modo che anche questa funzione abbia complessità costante.
Osserviamo che la funzione l_ultimo restituisce la posizione dell'ultimo elemento della lista. Per avere complessità costante dobbiamo costruire una struttura dati che ci permetta di conoscere la posizione dell'ultimo senza percorrere tutta la lista: dobbiamo memorizzare tale posizione in testa alla lista ed aggiornarla ogni volta che serve.

```
TYPE
     tipo_elemento=REAL;

     ptelemento=^elemento;
     elemento=RECORD
          d   :tipo_elemento;
          next:ptelemento;
     END;
     posizione=ptelemento;

     radice=RECORD
          root,
          last:posizione
     END;
     ptradice=^radice;
     lista   =ptradice;

VAR
     l:lista;
```

La lista è un puntatore ad un record che contiene due puntatori, il primo alla testa della lista vera e propria, il secondo alla coda. Le funzioni di inserimento e di cancellazione dovranno controllare se modificano l'ultimo

elemento della lista: in tal caso dovranno modificare anche il campo last di radice.

Sviluppiamo alcune delle primitive con la nuova struttura.

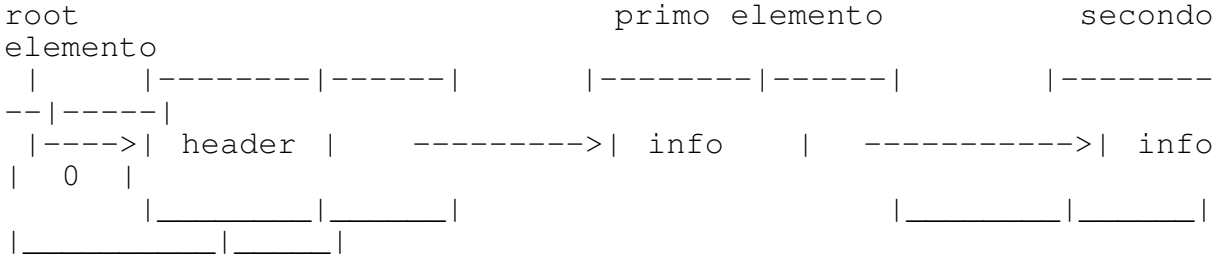
```
PROCEDURE l_crea( VAR l:lista )
BEGIN
     NEW( l );
     l^.root:=NIL;        { primo e ultimo conterranno NIL per
}
     l^.last:=NIL            { indicare che la lista è vuota
}
END;


PROCEDURE   l_inserisci(   VAR   l:lista;   x:tipo_elemento;
p:posizione );
{ inserisce l'elemento x alla posizione p della lista l }
VAR
     i:posizione;
BEGIN
     NEW( i );
     i^.d:=x;

     IF p=NIL THEN BEGIN
          i^.next:=l^.root; l^.root:=i
     END
     ELSE BEGIN
               i^.next:=p^.next; p^.next:=i
     END;

     IF p=l^.last THEN{ se l'elemento inserito è l'ultimo     }
          l^.last:=i  { si deve modificare last               }
END;

PROCEDURE l_cancella( VAR l:lista; p:posizione );
{ cancella l'elemento di posizione p dalla lista l }
VAR
     i:posizione;
BEGIN
     IF p=NIL THEN BEGIN
          i:=l^.root; l^.root:=l^.root^.next
     END
     ELSE BEGIN
               i:=p^.next; p^.next:=i^.next
     END;

     IF i=l^.last THEN  { se l'elemento eliminato è l'ultimo  }
          l^.last:=p    { modifica il campo last di radice    }
     DISPOSE( i );
END;


FUNCTION l_ultimo( l:lista ):posizione;
```

```
{ restituisce la posizione dell'elemento successivo all'ultimo
nella lista l }
BEGIN
     l_ultimo:=l^.last
END;
```

Le rimanenti primitive rimangono quelle presentate in 12.3.2.

12.5.3 Header.

Tutte le funzione che abbiamo implementato sulle liste operano in modo differente se elaborano il primo elemento della lista o uno degli altri. Per semplificare gli algoritmi possiamo modificare la struttura dati in modo tale che non sia necessaria questa distinzione perchè tutti gli elementi, compreso il primo, hanno un record che li precede. Graficamente possiamo rappresentare la situazione così:



La struttura dati per questa nuova implementazione di lista rimane quella di 12.3.2. Quello che cambia sono gli algoritmi. Vediamone alcuni

- creazione: la creazione dovrà creare la testata.

```
PROCEDURE l_crea( VAR l:lista );
BEGIN
     NEW( l );
     l^.next:=NIL;       { lista vuota }
END;
```

- primo: la funzione l_primo non dovrà più restituire NIL ma il puntatore alla testata.

```
FUNCTION l_primo( l:lista ):posizione;
BEGIN
     l_primo:=l;
END;
```

- inserimento: la funzione di inserimento non dovrà controllare se si deve operare sul primo poichè anche il primo ha un precedente.

```
PROCEDURE   l_inserisci(   VAR   l:lista;   x:tipo_elemento;
p:posizione );
```

```
{ inserisce un elemento x alla posizione p della lista l}
VAR
     i:posizione;
BEGIN
     NEW( i );
     i^.d:=x;
     i^.next:=p^.next;
     p^.next:=i
END;
```

La modifica delle altre funzioni è lasciata al lettore.

Problema 1. Definire un tipo di dato per la gestione dei polinomi in una sola variabile.

STRUTTURA DATI: Utilizzando le liste la struttura dati per il polinomio è

```
TYPE
     lettera = 'A'..'Z';

     termine=RECORD
          grado :INTEGER;
          coef  :REAL;
     END;

     tipo_elemento=^termine;
     posizione    =^elemento;

     elemento=RECORD
          d   :tipo_elemento;
          next:posizione;
     END;

     lista=posizione;

     poly= RECORD
         variabile : lettera;
         termini   : lista;
     END;

     polinomio =^poly;
```

Il polinomio è un puntatore ad un record che contiene la variabile e la lista dei termini che lo compongono. Ogni termine è caratterizzato da un coefficiente e da un esponente.

- la procedura stampa_lista è la stampa dei singoli termini.

```
PROCEDURE stampa_lista( s:lista );
VAR
     i:posizione;
BEGIN
   i:=l_primo( s );
```

```
   WHILE i<>l_ultimo( s ) DO BEGIN
      WRITE('(',l_elemento(s,i)^.grado:3,'                    ',
l_elemento(s,i)^.coef:6:2,')');
      WRITE(' ');
      i:=l_successivo( s, i )
   END
END;
```

- la funzione ricerca è la funzione di ricerca in una lista ordinata in modo decrescente rispetto al campo grado.

```
FUNCTION ricerca( VAR l:lista; t:tipo_elemento; VAR p:posizione
):BOOLEAN;
VAR
     trovato,
     trovabile:BOOLEAN;
        u  : posizione;
BEGIN
     trovato:=FALSE; trovabile:=TRUE;
     p:=l_primo( l );
     u:=l_ultimo( l );
     WHILE ( p<>u ) AND NOT trovato AND trovabile DO
     IF ( l_elemento(l, p)^.grado=t^.grado ) THEN
                trovato:=TRUE
     ELSE IF ( l_elemento(l, p)^.grado<t^.grado) THEN
                trovabile:=FALSE
          ELSE p:=l_successivo(l, p);
     ricerca:=trovato
END;
```

- la procedura add_term aggiunge un elemento alla lista in modo che la lista rimanga ordinata: se l'elemento è presente somma i due coefficienti.
Se la somma da 0 cancella il nodo.

```
PROCEDURE add_term( VAR l:lista; t:tipo_elemento );
VAR
     p:posizione;
BEGIN
    IF  t^.coef <> 0 THEN
        IF ricerca(l, t, p) THEN BEGIN
           l_elemento(l,  p)^.coef:=l_elemento(l,  p)^.coef  +
t^.coef;
           IF l_elemento(l,p)^.coef=0 THEN l_cancella(l, p)
     END ELSE l_inserisci(l, t, p)
END;
```

- la funzione crea_poly crea un polinomio in una determinata variabile, inizializzando la lista dei termini a NIL.

```
FUNCTION crea_poly(x:lettera):polinomio;
VAR
   p:polinomio;
BEGIN
     NEW(p);
     p^.variabile:=x;
     l_crea(p^.termini);
     crea_poly:=p;
END;
```

- la funzione var_poly restituisce la variabile in cui è espresso il polinomio.

```
FUNCTION var_poly(p:polinomio):lettera;
BEGIN
    var_poly:=p^.variabile
END;
```

- la funzione add_poly1 è un primo modo per sommare due polinomi: crea il polinomio somma inserendo nel polinomio risultato prima gli elementi del primo polinomio, poi quelli del secondo. Date le caratteristiche della funzione add_term il risultato è la somma dei due polinomi.

```
FUNCTION add_poly1(p1,p2:polinomio):polinomio;
VAR
  p3:polinomio;
  l,l1 :lista;
  t:tipo_elemento;
BEGIN
    IF var_poly(p1) = var_poly(p2) THEN BEGIN
        p3:=crea_poly(var_poly(p1));
        l:=l_primo(p1^.termini);
        l1:=l_ultimo(p1^.termini);
        WHILE l <> l1 DO BEGIN
             NEW(t);
             t^:=l_elemento(p1^.termini,l)^;
             add_term(p3^.termini,t);
             l:=l_successivo(p1^.termini,l)
        END;
        l:=l_primo(p2^.termini);
        l1:=l_ultimo(p2^.termini);
        WHILE l <> l1 DO BEGIN
            NEW(t);
            t^:=l_elemento(p2^.termini,l)^;
            add_term(p3^.termini,t);
            l:=l_successivo(p2^.termini,l)
        END
    END
    ELSE p3:=NIL;
    add_poly1:=p3
END;
```

- la funzione add_poly è un secondo modo per sommare due polinomi: poichè le due liste dei termini sono ordinate per grado l'algoritmo è nella sostanza il merge (la fusione) tra due liste ordinate.

```
FUNCTION add_poly(p1,p2:polinomio):polinomio;
VAR
  p3:polinomio;
  l,l1,l2,l3,l4 :lista;
  t:tipo_elemento;
BEGIN
   IF var_poly(p1) = var_poly(p2) THEN BEGIN
      p3:=crea_poly(var_poly(p1));
      l:=l_primo(p1^.termini);
      l1:=l_ultimo(p1^.termini);
      l2:=l_primo(p2^.termini);
      l3:=l_ultimo(p2^.termini);
      l4:=l_primo(p3^.termini);
      WHILE (l <> l1) AND (l2 <> l3) DO BEGIN
         NEW(t);
         IF l_elemento(p1^.termini,l)^.grado >
                        l_elemento(p2^.termini,l2)^.grado THEN
BEGIN
             t^:=l_elemento(p1^.termini,l)^;
             l:=l_successivo(p1^.termini,l)
         END ELSE
             IF l_elemento(p1^.termini,l)^.grado <
                        l_elemento(p2^.termini,l2)^.grado THEN
BEGIN
                 t^:=l_elemento(p2^.termini,l2)^;
                 l2:=l_successivo(p2^.termini,l2)
             END
             ELSE BEGIN
                    t^:=l_elemento(p1^.termini,l)^;
                    t^.coef:=t^.coef +
l_elemento(p2^.termini,l2)^.coef;
                    l:=l_successivo(p1^.termini,l);
                    l2:=l_successivo(p2^.termini,l2)
                  END;
         IF t^.coef <> 0 THEN BEGIN
               l_inserisci(p3^.termini,t,l4);
               l4:=l_successivo(p3^.termini,l4)
         END ELSE DISPOSE(t);
      END;
      WHILE l <> l1 DO BEGIN
         NEW(t);
         t^:=l_elemento(p1^.termini,l)^;
         l:=l_successivo(p1^.termini,l);
         l_inserisci(p3^.termini,t,l4);
         l4:=l_successivo(p3^.termini,l4)
      END;
      WHILE l2 <> l3 DO BEGIN
         NEW(t);
```

```
          t^:=l_elemento(p2^.termini,l2)^;
          l2:=l_successivo(p2^.termini,l2);
          l_inserisci(p3^.termini,t,l4);
          l4:=l_successivo(p3^.termini,l4)
       END;
    END
    ELSE p3:=NIL;
     add_poly:=p3
END;
```

- la funzione molt_poly realizza la moltiplicazione di due polinomi: inserisce in modo corretto (usando la funzione add_term) nel polinomio risultato i prodotti dei singoli termini.

```
FUNCTION molt_poly(p1,p2:polinomio):polinomio;
VAR
  p3:polinomio;
  l,l1,l2,l3 :lista;
  t      :tipo_elemento;
  t1,t2  :termine;
BEGIN
    IF var_poly(p1) = var_poly(p2) THEN BEGIN
        p3:=crea_poly(var_poly(p1));
        l:=l_primo(p1^.termini);
        l1:=l_ultimo(p1^.termini);
        l3:=l_ultimo(p2^.termini);
        WHILE l <> l1 DO BEGIN
            t1:=l_elemento(p1^.termini,l)^;
            l2:=l_primo(p2^.termini);
            WHILE l2 <> l3 DO BEGIN
                NEW(t);
                t2:=l_elemento(p2^.termini,l2)^;
                t^.grado:=t1.grado + t2.grado;
                t^.coef:=t1.coef * t2.coef;
                add_term(p3^.termini,t);
                l2:=l_successivo(p2^.termini,l2)
            END;
            l:=l_successivo(p1^.termini,l)
        END
    END
    ELSE p3:=NIL;
    molt_poly:=p3
END;
```

- la procedura stampa_poly stampa la variabile in cui è espresso il polinomio e poi la lista dei termini.

```
PROCEDURE stampa_poly(p:polinomio);
BEGIN
     IF p <> NIL THEN BEGIN
         WRITE(var_poly(p),' ');
         stampa_lista(p^.termini);
         WRITELN
```

```
     END
END;
```

- la procedura leggi_poly permette l'inserimento di un polinomio.

```
PROCEDURE leggi_poly(VAR p:polinomio);
VAR
  x:lettera;
  t:tipo_elemento;
BEGIN
        WRITE('leggi incognita = ');
        READLN(x);
        p:=crea_poly(x);
     NEW(t);
     WRITE('grado ');
     READLN( t^.grado );
     WHILE( t^.grado<>-1 ) DO BEGIN
          WRITE('coeff ' );
          READLN( t^.coef );
          add_term( p^.termini, t);
          NEW(t);
          WRITE('grado ');
          READLN( t^.grado );
     END;
        DISPOSE(t);
END;
```

Problema 2. Gestire una tabella con organizzazione hash.

In molte applicazioni con tabelle abbiamo dovuto ricercare un elemento attraverso uno o più campi. Supponiamo ora di avere parecchi accessi ad una tabella in ordine imprevedibile e contemporaneamente ci siano parecchi inserimenti e cancellazioni. Non è pensabile mantenere ordinata la tabella perchè costa troppo. Contemporaneamente la ricerca sequenziale di un elemento comporta tempi di risposta troppo elevati. In questi casi è possibile operare direttamente sul campo che serve per la ricerca, detto chiave di accesso, utilizzando una funzione che trasforma la chiave in un numero. L'elemento ricercato, se esiste, si trova alla posizione indicata dal numero. Una gestione di tabella basata sulla trasformazione della chiave si dice hash.
Se la funzione hash è iniettiva, cioè se dette k1 e k2 due chiavi distinte h(k1)<>h(k2), per la ricerca non ci sono problemi. Si applica la funzione alla chiave di ricerca. Se la posizione ricavata non è libera allora abbiamo trovato l'elemento ricercato altrimenti possiamo concludere che non esiste in tabella. Ad esempio supponiamo di avere come chiave una stringa di tre caratteri, con una lettera maiuscola ed un numero con due cifre. Una possibile funzione hash, iniettiva e crescente, è la seguente:

```
TYPE
```

```
     str3=PACKED ARRAY[1..3] OF CHAR;

FUNCTION hash( chiave:str3 ):INTEGER;
VAR
     h : INTEGER;
BEGIN
     h:=( ord(chiave[1]) - ord('A') ) * 100;
     h:=h + ( ord(chiave[2]) - ord('0') ) * 10 + ord(chiave[3])
- ord('0');
     hash:=h;
END;
```

I possibili valori generati da tale funzione variano tra 0 e 2599.

I problemi nascono quando non è possibile scrivere una funzione iniettiva. Ad esempio supponiamo di avere come chiave il cognome e il nome degli alunni di una scuola. Supponiamo inoltre che i caratteri utilizzati siano solo quelli maiuscoli. Una funzione hash iniettiva sarà:

```
TYPE
     str30=PACKED ARRAY[1..30] OF CHAR;
     str60=PACKED ARRAY[1..60] OF CHAR;
     chiave=RECORD
          cognome,
          nome   :str30;
     END;

FUNCTION hash( c:chiave ):INTEGER;
VAR
     i, h, k : INTEGER;
     cc      : str60;
BEGIN
     i:=1;
     WHILE c.cognome[i]<>'*' DO BEGIN
          cc[i]:=c.cognome[i];
          i:=i+1;
     END;
     k:=1;
     WHILE c.nome[k]<>'*' DO BEGIN
          cc[i]:=c.nome[k];
          k:=k+1;
          i:=i+1;
     END;
     h:=ord(cc[1])-ord('A');
     FOR k:=2 TO i-1 DO
          h:=h*25+ord(cc[k])-ord('A');
     hash:=h;
END;
```

Si può facilmente notare che il numero h generato da tale funzione supera la capacità degli INTEGER. Inoltre se la scuola ha 1000 alunni lo spreco di spazio per la tabella è enorme. In

questo caso si deve utilizzare una funzione hash non iniettiva. Allora applicandola a due chiavi diverse si può ottenere lo stesso valore. Abbiamo una collisione. Per risolvere questo problema si può utilizzare la lista. La tabella sarà un vettore di liste. Dato un elemento si risale alla lista che dovrebbe contenerlo. La ricerca avviene confrontando tutti gli elementi della lista giusta con l'elemento da ricercare. Naturalmente più lunga è la lista più lenta sarà la ricerca.

Per gestire gli alunni di una scuola con una tabella hash definiamo la seguente struttura dati:

```
CONST
     dim=2000;
TYPE
     str30=PACKED ARRAY[1..30] OF CHAR;
     dato=RECORD
          cognome,
          nome   : str30;
     END;
     tipo_elemento=^dato;                { tipo_elemento è un
puntatore a dato }

     ptelemento=^elemento;
     elemento=RECORD
          d   :tipo_elemento;
          next:ptelemento;
     END;

     posizione=ptelemento;
     radice=RECORD
          root,
          last:posizione
     END;
     ptradice=^radice;
     lista   =ptradice;

     tabella=ARRAY[0..dim] OF lista;
{ tabella è il vettore delle liste di collisione }

VAR
     l   :tabella;
```

- La funzione hash che utilizziamo è la seguente:

```
FUNCTION hash( d:tipo_elemento ):INTEGER;
VAR
     i, h, k : INTEGER;
     cc      : PACKED ARRAY[1..60] OF CHAR;
BEGIN
     i:=1;
     WHILE d^.cognome[i]<>'*' DO BEGIN  { cognome e nome sono
terminati da * }
          cc[i]:=d^.cognome[i];
```

```
          i:=i+1;
     END;
     k:=1;
     WHILE d^.nome[k]<>'*' DO BEGIN
          cc[i]:=d^.nome[k];
          k:=k+1;
          i:=i+1;
     END;

     h:=ord(cc[1]);
     FOR k:=2 TO i-1 DO
          h:=(h*2+ord(s[k])) MOD (dim+1);
     hash:=h;
END;
```

Per gestire la tabella dobbiamo implementare una funzione di inizializzazione, una funzione di ricerca, una di inserimento ed una di cancellazione. Come esempio sviluppiamo l'inizializzazione e la ricerca mentre l'inserimento e la cancellazione sono lasciate al lettore.

- inizializzazione:

```
PROCEDURE h_crea( VAR t:tabella );
VAR
     i:INTEGER;
BEGIN
     FOR i:=0 TO dim DO
          l_crea( t[i] );
END;
```

- ricerca:

```
FUNCTION     ricerca_in_lista(     l:lista;     d:tipo_elemento
):tipo_elemento;
VAR
     p      :posizione;
     trovato:BOOLEAN;
BEGIN
     p:=l_primo(l); trovato:=FALSE;
     WHILE (p<>l_ultimo(l)) AND NOT trovato DO
          IF (l_elemento(l,p)^.cognome=d^.cognome) AND
               (l_elemento(l,p)^.nome=d^.nome)             THEN
trovato:=TRUE
          ELSE p:=l_successivo(l,p);
     IF trovato THEN ricerca_in_lista:=l_elemento(l,p)
     ELSE ricerca_in_lista:=NIL
END;

FUNCTION h_ricerca( t:tabella; d:tipo_elemento ):tipo_elemento;
VAR
     p:INTEGER;
BEGIN
     p:=hash(d);
```

```
     h_ricerca:=ricerca_in_lista( t[p], d)
END;
```

Esercizi.

1- Scrivere una procedura che, data una lista di reali, restituisca la somma dei valori della lista.

2- Scrivere una procedura che, data una lista di interi, restituisca la lista degli elementi pari e la lista degli elementi dispari.

3- Scrivere una procedura che, date due liste di reali di uguale lunghezza, restituisca il prodotto scalare delle due liste.

4- Scrivere una procedura che, data una lista di reali, restituisca la media degli elementi della lista.

5- Scrivere una procedura che, data una lista di interi, restituisca una lista formata dagli stessi numeri ordinati in modo crescente.

6- Scrivere una procedura che, data una lista di interi, restituisca il valore massimo.

7- Scrivere una procedura che, data una lista di reali, stampi gli elementi della lista in ordine inverso al modo in cui compaiono nella lista.

8- Scrivere una procedura che, data una lista di reali, restituisca una lista contenente i valori che nella prima lista compaiono almeno due volte.

9- Scrivere una procedura che, data una lista di reali, restituisca una lista contenente i valori che nella prima lista compaiono due volte.

10- Scrivere una procedura che, date due liste di interi ordinate, restituisca la lista ordinata formata dagli elementi delle due liste. Si richiede che l'algoritmo abbia complessità lineare.

11- Scrivere una procedura che, data una lista ordinata alfabeticamente di cognomi e nomi di persone e il cognome e nome di una nuova persona, inserisca il nuovo valore nella lista, rispettando l'ordinamento.

12- Scrivere una procedura che, data una lista di interi L e una posizione P e un numero N, restituisca la lista che si ottiene eliminando dalla lista L, partendo dalla posizione P, N interi.

13- Implementare il tipo INSIEME di interi tramite liste ordinate e in particolare le seguenti funzioni:

a- FUNCTION make_null:insieme;
   restituisce l'insieme vuoto.

b- FUNCTION appartiene(x : INTEGER; i : insieme):BOOLEAN;
   restituisce vero se x appartiene all'insieme, falso altrimenti

c- FUNCTION insert (x : INTEGER; i : insieme):insieme;
   inserisce x nell'insieme i

d- FUNCTION estrai (x : INTEGER; i : insieme):insieme;
   restituisce l'insieme i privato dell'elemento x

e- FUNCTION is_null(i : insieme): BOOLEAN;
   restituisce vero se l'insieme è vuoto, falso altrimenti

14- Implementare il tipo MATRICE_SPARSA (cap. 10) utilizzando le liste.

12.6 Pila.

Possiamo considerare la pila come un caso particolare della lista nella quale inserimenti e cancellazioni avvengono solo da un estremo. Per questo la pila è detta struttura lifo per indicare che l'ultimo elemento inserito sarà il primo ad essere estratto (last in first out). Un modello intuitivo della pila è una pila di dischi su un tavolo, una pila di piatti nel lavandino, una pila di riviste sullo scaffale, dove per aggiungere o togliere un oggetto conviene operare sulla cima.

12.6.1 Operazioni

Le operazioni che si devono poter compiere su una pila sono le seguenti:

- creazione: si deve poter creare la struttura vuota.

- inserimento: l'inserimento di un nuovo oggetto nella pila deve avvenire sempre in cima.

- cancellazione: la cancellazione di un elemento avviene sempre in cima.

12.6.2 Realizzazione.

Poichè la pila è una caso particolare della lista lineare possiamo realizzarla partendo dalla lista ed implementando alcune funzioni elementari.

STRUTTURA DATI:

```
TYPE
     ..........
     pila=lista;
VAR
     p:pila;
```

Possiamo utilizzare il tipo lista sviluppato in 12.3.2. Le operazioni elementari possono essere facilmente sviluppate utilizzando le primitive della lista e ricordando di compiere inserimenti ed estrazioni in testa.

ALGORITMI:

- creazione: la creazione avviene inizializzando la lista.

```
PROCEDURE crea( VAR p:pila );
BEGIN
     l_crea( p );
END;
```

-inserimento: per inserire in una pila si utilizza l'inserimento in una lista indicando come posizione la prima.

```
PROCEDURE push( VAR p:pila; x:tipo_elemento );
BEGIN
     l_inserisci( p, x, l_primo(p) );
END;
```

c. cancellazione: si cancella sempre il primo elemento della lista.

```
PROCEDURE pop( VAR p:pila );
BEGIN
     l_cancella( p, l_primo(p) );
END;
```

- per poter estrarre tutti gli elementi di una pila per visualizzarli sono necessarie due funzioni elementari che ci dicano quando la pila diventa vuota e che restituiscano il primo elemento della pila.

- restituzione dell'elemento in testa. Utilizzando la funzione elemento della lista restituisce il primo elemento della pila.

```
FUNCTION top( VAR p:pila ):tipo_elemento;
BEGIN
     top:=l_elemento( p, l_primo(p) );
END;
```

- pila vuota: una pila è vuota se primo=ultimo.

```
FUNCTION vuota( VAR p:pila ):BOOLEAN;
BEGIN
     vuota:=l_primo(p)=l_ultimo(p);
```

```
END;
```

Esercizi:

1. Implementare la pila con la lista di 12.3.1 mantenendo costante la complessità delle primitiva.

Problema 3. Realizzare una semplice calcolatrice che utilizza come ingresso un'espressione scritta in notazione polacca postfissa.

La notazione polacca postfissa permette di scrivere un'espressione senza utilizzare le parentesi. Ad esempio

```
(3 + 2) * (4 + 6)        diventa    3 2 + 4 6 + *
((5 + 7 + 9) * 2) / 8    diventa    5 7 + 9 + 2 * 8 /
5 + 7 + 9                diventa    5 7 9 + +
```

Per valutare l'espressione è molto utile utilizzare una pila. Ad esempio sia

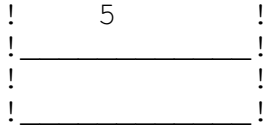
```
3 2 + 4 6 + *
```

l'espressione da valutare. Utilizzando una pila si procede in questo modo:

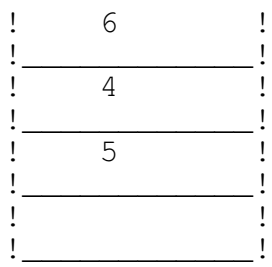
analizzando l'espressione a partire da sinistra si inseriscono nella pila tutti i numeri che si incontrano.

Quando si incontra un simbolo di operazione vengono estratti dalla pila gli ultimi due valori, si procede alla somma e si inserisce il risultato nella pila

```
3  +  2  = 5
```

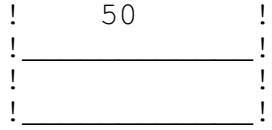


si continua sino a quando l'espressione è finita



4 + 6 = 10

5 * 10 = 50



Alla fine nella pila è memorizzato il risultato dell'espressione

OBIETTIVO: valutare un'espressione scritta in notazione polacca postfissa
RISULTATI: risultato espressione
DATI: stringa contenente numeri e i simboli +, *, -, /.
LIMITI: per inserire nella stringa il numero -3 bisogna scrivere 0 3 -.

STRUTTURE DATI:

```
TYPE
     stringa=PACKED ARRAY[1..80] OF CHAR;   { la stringa è
terminata da $ }
```

La pila è così definita:

```
     dato=REAL;
     tipo_elemento=^dato;                { tipo_elemento è un
puntatore a dato }

     ptelemento=^elemento;
     elemento=RECORD
```

```
          d   :tipo_elemento;
          next:ptelemento;
     END;

     posizione=ptelemento;

     radice=RECORD
          root,
          last:posizione
     END;
     ptradice=^radice;
     lista   =ptradice;
     pila=lista;

VAR
     str : stringa;
```

ALGORITMO: la parte fondamentale dell'algoritmo consiste nello scrivere una funzione di valutazione che, avuto in ingresso una stringa, la valuta e restituisce il risultato.

```
          FUNCTION valuta( str:stringa ):REAL;
```

La valutazione avviene così:

```
     Per tutti i caratteri di str diversi da ' '
          classifica il carattere
          esegui operazione corrispondente
     restituisci l'unico valore presente nella pila
```

I caratteri ammessi sono:

```
TYPE
     token=(numero, piu, per, diviso, meno, virgola, blank );
```

Le operazioni sono di tre tipi: se si tratta di piu, meno, per, diviso si deve eseguire il calcolo visto in precedenza, se si tratta di blank si deve passare al nuovo carattere, se si tratta di numero bisogna convertirlo in REAL e inserirlo nella pila. Le operazioni terminano quando il carattere esaminato è $. La funzione sarà:

```
FUNCTION valuta( str:str80 ):REAL;
TYPE
     token=(numero, piu, per, diviso, meno, virgola, blank );
     str20=PACKED ARRAY[1..20] OF CHAR;
VAR
     p:INTEGER;                        { indica il carattere da
esaminare  }
     tk:token;                    { indica il tipo di carattere
}
     operazione:SET OF token; { insieme che indica i simboli di
calcolo  }
     s:pila;
```

```
PROCEDURE elimina_blank;
BEGIN
     WHILE  str[p]=' ' DO p:=p+1
END;

PROCEDURE stop( s:str20 );
{ in caso di errore si indica la posizione nella stringa dove è
avvenuto e il tipo di errore }

BEGIN
     WRITELN( str );
     WRITELN( '^':p, s );
     halt;
END;


FUNCTION tipo_tk( x:CHAR ):token;
{ classifica il carattere della stringa  }

BEGIN
     CASE x OF
          '0'..'9':tipo_tk:=numero;
          '+'     :tipo_tk:=piu;
          '-'     :tipo_tk:=meno;
          '*'     :tipo_tk:=per;
          '/'     :tipo_tk:=diviso;
          '.'     :tipo_tk:=virgola;
          ' '     :tipo_tk:=blank;
          ELSE stop('simbolo sconosciuto ');
     END;
END;

PROCEDURE calcola;
{  esegue  il  calcolo  in  base  al  simbolo  contenuto  in  tk,
prelevando gli operandi dalla pila e inserendo il risultato
sulla pila *

VAR
     op1, op2 :REAL;
     op:tipo_elemento;
BEGIN
     IF NOT p_vuota( s ) THEN op1:=top(s)^
     ELSE stop( 'manca argomento ');
     pop( s );
     IF NOT p_vuota( s ) THEN op2:=top(s)^
     ELSE stop( 'manca argomento ');
     pop( s );
     NEW(op);
     CASE tk OF
          piu :   BEGIN op^:=op2+op1; push( s, op );
                  END;
          meno:   BEGIN op^:=op2-op1; push( s, op );
                  END;
```

```
          per:    BEGIN op^:=op2*op1; push( s, op );
                  END;
          diviso: BEGIN op^:=op2/op1; push( s, op );
                  END;
     END;
     p:=p+1;
END;

FUNCTION converti_numero:tipo_elemento;
{ se il simbolo è un numero viene convertito in REAL  }

VAR
     e    :tipo_elemento;
     a,b,c:INTEGER;
BEGIN
     NEW(e); a:=0; b:=0; c:=1;
     WHILE tipo_tk(str[p])=numero DO BEGIN
          a:=a*10+ord(str[p])-ord('0'); p:=p+1;
     END;
     e^:=a;
     IF tipo_tk(str[p])=virgola THEN BEGIN
          p:=p+1;
          WHILE tipo_tk(str[p])=numero DO BEGIN
               b:=b*10+ord(str[p])-ord('0');
               c:=c*10; p:=p+1;
          END;
          e^:=e^+b/c;
     END;
     converti_numero:=e;
END;

BEGIN          { valuta }
     p_crea(s);
     p:=1;
     operazione:=[piu, meno, per, diviso];
     elimina_blank;
     WHILE (str[p]<>'$') DO BEGIN
          tk:=tipo_tk(str[p]);
          IF tk IN operazione THEN calcola
          ELSE push(s,converti_numero);
          elimina_blank;
     END;
     IF NOT p_vuota( s ) THEN valuta:=top(s)^
     ELSE stop( 'manca argomento ');
     pop( s );
     IF NOT p_vuota( s ) THEN stop( 'manca operazione ');
END;
```

12.7 Coda.

Anche la coda può essere considerata come un caso particolare della lista nella quale gli inserimenti avvengono ad un estremo e le estrazioni all'altro. La coda è detta struttura fifo (first in first out) perchè il primo elemento ad essere inserito è anche il primo ad essere estratto. Un modello intuitivo della coda è la coda che si determina allo sportello di un ufficio o all'uscita di un'autostrada.

12.7.1 Operazioni.

Le operazioni che si compiono su una coda sono:

- creazione: la coda deve essere creata vuota.

- inserimento: l'inserimento di un nuovo elemento deve essere effettuato in fondo alla coda

- estrazione: l'elemento estratto è quello in testa

- cancellazione: l'elemento da cancellare è sempre quello in testa

- vuota: è necessario sapere quando la coda è vuota per evitare operazioni scorrette.

12.7.2 Realizzazione.

Abbiamo detto che la coda può essere considerata come un caso particolare della lista, con le operazioni di inserimento che avvengono in fondo e le operazioni di cancellazione che avvengono in testa. Utilizziamo la lista come in 12.3.2 ma con la modifica apportata in 12.5.2 per rendere costanti tutte le operazioni.

STRUTTURA DATI:

La struttura dati è la stessa di 12.5.2. La coda sarà:

```
TYPE
     ...........
     coda=lista;
VAR
     c:coda;
```

Le operazioni elementari saranno:

- creazione: si crea la coda utilizzando la funzione l_crea.

```
PROCEDURE c_crea( VAR c:coda );
BEGIN
     l_crea( c );
END;
```

- inserimento: l'inserimento avviene sempre in fondo alla lista.

```
PROCEDURE in_coda( VAR c:coda; x:tipo_elemento );
BEGIN
     l_inserisci( c, x, l_fine(c) );
END;
```

- cancellazione: gli elementi vengono eliminati dalla testa della coda.

```
PROCEDURE da_coda( VAR c:coda );
BEGIN
     l_cancella( c, l_primo(c) );
END;
```

- elemento: l'elemento che deve essere restituito è sempre quello in testa.

```
FUNCTION testa( VAR c:coda ):tipo_elemento;
BEGIN
     testa:=l_elemento( c, l_primo(c) );
END;
```

- vuota: la coda è vuota quando primo=ultimo.

```
FUNCTION vuota( VAR c:coda ):BOOLEAN;
BEGIN
     vuota:=l_primo(c)=lultimo(c);
END;
```

12.7.3 Realizzazione con i vettori.

La realizzazione 12.3.1 della lista non è conveniente per la coda poichè non si riesce a rendere costanti sia l'inserimento in coda sia la cancellazione. Vediamo allora come implementare direttamente con i vettori un tipo coda.
La strutura dati sarà:

```
TYPE
     coda=RECORD
          elementi    :ARRAY[1..max] OF tipo_elemento;
          testa, fondo:INTEGER;
     END;
```

dove testa indica l'estremo della coda da dove vengono fatte le estrazioni e fondo l'estremo nel quale inserire il nuovo elemento. Consideriamo il vettore circolare, cioè la posizione successiva all'ultima sarà la prima. Il problema che dobbiamo risolvere sarà riconoscere quando la coda è vuota o piena. Possiamo pensare che fondo indichi sempre una posizione occupata mentre testa quella libera. Gli algoritmi di base sono:

- creazione: la coda viene creata con testa=fondo=1. La condizione testa=fondo indica lista vuota.

```
PROCEDURE c_crea( VAR c:coda);
BEGIN
     c.testa:=1; c.fondo:=1
END;
```

- vuota: la coda è vuota quando testa=fondo

```
FUNCTION vuota( VAR c:coda ):BOOLEAN;
BEGIN
     vuota:=c.testa=c.fondo
END;
```

- inserisci: l'inserimento avviene in fondo. Ricordiamo che fondo indica sempre la posizione libera. La coda è piena se il successivo di fondo=testa.

```
PROCEDURE in_coda( VAR c:coda; x:tipo_elemento );
BEGIN
     c.elementi[c.fondo]:=x;
     c.fondo:=successivo( c.fondo );
     IF c.fondo=c.testa THEN errore( 'coda piena' );
END;
```

- cancella: se testa=fondo allora la coda è vuota

```
PROCEDURE da_coda( VAR c:coda );
BEGIN
     IF c.testa=c.fondo THEN errore( 'coda vuota' );
     c.testa:=successivo(c.testa);
END;
```

- testa: se testa=fondo allora la coda è vuota.

```
FUNCTION testa( VAR c:coda ):tipo_elemento;
BEGIN
     IF c.testa=c.fondo THEN errore( 'coda vuota ' );
     testa:=c.elementi[c.testa];
END;
```

La funzione successivo determina il successivo di un numero modulo max

```
FUNCTION successivo( i:INTEGER ):INTEGER;
BEGIN
     successivo:= (i MOD max) + 1;
END;
```

Problema 4. Valutare le code che si formano agli sportelli di una banca.

Il problema presentato è un problema di ricerca operativa che può essere risolto utilizzando approcci diversi: l'analisi e lo studio delle problematiche relative non rientra negli scopi di questo libro. Come per ogni altro problema, il punto critico è la costruzione di un modello stocastico in grado di descrivere il fenomeno che si vuole analizzare: come abbiamo già osservato nel capitolo 2, non è possibile la costruzione di un modello senza la conoscenza adeguata del fenomeno da descrivere.
Nella soluzione proposta verranno operate notevoli semplificazioni e non verranno assolutamente trattati i problemi di tipo statistico.

DEFINIZIONE DEL PROBLEMA: Supponiamo di avere una banca con n cassieri. Durante una giornata lavorativa arrivano diversi clienti che, se trovano tutte le casse occupate, si inseriranno nella coda più corta. Supponiamo che ci siano n code, una per cassiere.

OBIETTIVO: misurare l'efficienza del servizio offerto dalla banca. L'efficienza viene valutata sia dal punto di vista della banca, sia dal punto di vista del cliente. La banca è interessata al tempo di utilizzo di ciascun cassiere e alle lunghezze medie delle code, il clente è interessato al tempo medio di permanenza in coda.

RISULTATI: per ogni cassiere è necessario conoscere:

- il numero totale di clienti serviti
- il tempo effettivo di lavoro del cassiere

per ogni coda è necessario conoscere:

- la lunghezza massima della coda nell'arco della giornata
- la dimensione media della coda
- il tempo medio di attesa dei clienti
- il tempo massimo di attesa

DATI: numero dei cassieri.

Per poter costruire un modello adeguato è necessario conoscere la distribuzione dei clienti nell'arco di una giornata e il servizio richiesto da ciascuno. In questo problema supponiamo che, dopo analisi adeguate, si sia giunti a queste conclusioni:

- distribuzione dei clienti: i tempi che intercorrono tra l'arrivo di due clienti possono essere 3, 5, 8, 12, tutti con uguale probabilità.

- il servizio richiesto influenza la simulazione per quanto riguarda l'occupazione del cassiere. I tempi di servizio per ogni cassiere sono 7, 10, 20 tutti con eguale probabilità. Supponiamo che i tempi di servizio del cassiere siano indipendenti dal cassiere stesso.

I tempi sono espressi in minuti. La simulazione avverrà nell'arco di una giornata: il tempo di simulazione sarà 480.

STRUTTURE DATI: ogni cassiere sarà identificato da un record così definito:

```
status=(libero, occupato);    { possono essere introdotti altri stati per
                                il                    cassiere
}
cassiere=RECORD
     clock       : INTEGER;     { indica il tempo del prossimo evento      }
     stato        : status;       { ogni cassiere può essere libero/occupato }
     t_utilizzo,                  { tempo effettivo di lavoro }
     n_cli_ser   : INTEGER;      { numero di clienti serviti }
     cli         : ptcli;       { puntatore al cliente servito }
END;
```

dove con evento si intende la fine del servizio di un cassiere per un cliente.
I clienti sono caratterizzati dal tempo di arrivo e dal tempo richiesto per il servizio presso una cassa.

```
ptcli  =^cliente;
cliente=RECORD
     t_servizio,
     t_arrivo  : INTEGER;
END;
```

Per ogni cassiere esiste una coda così definita:

```
coda_cliente=RECORD
     tempo_ultimo_evento,                    { tempo ultimo arrivo/partenza in/da coda }
     massimo_coda,               { massima lunghezza raggiunta dalla coda  }
     lung_coda,                     { lunghezza attuale della coda            }
     tempo_attesa_totale,          { somma di tutti i tempi di attesa       }
     dimensione_media    : INTEGER;
     coda                : lista;
END;
```

la dimensione media viene calcolata come

     sommatoria (lunghezza coda * tempo in cui è rimasta tale lunghezza)

la coda è così definita:

```
tipo_elemento=ptcli;
ptelemento   =^elemento;
elemento     =RECORD
     d   :tipo_elemento;
     next:ptelemento;
END;
posizione=ptelemento;
radice   =RECORD
     root,
     last:posizione
END;
ptradice=^radice;
lista   =ptradice;
```

per quanto riguarda le operazioni sulla coda utilizziamo l'implementazione vista nei paragrafi precedenti.

Per quanto riguarda l'insieme dei cassieri e delle code definiamo due vettori di n_max_casse componenti:

```
cassieri=ARRAY[0..n_max_casse] OF cassiere;
code    =ARRAY[0..n_max_casse] OF coda_cliente;
```

deve poi esistere un orologio generale che permette la sincronizzazione dei vari eventi

```
VAR
     orologio : INTEGER;
     banca    : cassieri;
     c        : code;
     n_casse  : INTEGER;
```

ALGORITMO: la simulazione che faremo è ad eventi, cioè si individuano gli eventi e si fa avanzare l'orologio globale del sistema simulato da un evento all'altro, supponendo che tra due eventi non succeda nulla. In precedenza abbiamo affermato che gli eventi sono determinati dalla terminazione del lavoro di un cassiere con un cliente. Non abbiamo però affrontato l'evento "arrivo cliente nuovo". Supponiamo allora l'esistenza di un cassiere fittizio, il cassiere 0, che quando termina di lavorare genera il cliente che entra nella banca. La ricerca del prossimo evento sarà allora:

```
PROCEDURE prossimo_evento;

     tra tutte le macchine occupate ricerca quella che
     ha l'orologio con valore minimo, libera la cassa
     ed assegna il valore dell'orologio della cassa
     all'orologio globale.
```

Codificando:

```
PROCEDURE prossimo_evento;
VAR
     min:INTEGER;

FUNCTION minimo( VAR i_min:INTEGER ):BOOLEAN;
VAR
     i, min:INTEGER;
BEGIN
     min:=MAXINT;
     FOR i:=0 TO n_casse DO
          IF (banca[i].stato=occupato) AND ( banca[i].clock<min
) THEN BEGIN
               min:=banca[i].clock; i_min:=i
          END;
     minimo:=min<>MAXINT;
END;

BEGIN
     IF minimo( min ) THEN BEGIN    { ricerca minimo tra le
casse occupate }
          libera_cassa( min );
          orologio:=banca[min].clock;
     END;
END;
```

Per quanto riguarda la liberazione della cassa dobbiamo distinguere tra la cassa fittizia 0 e le altre casse.

```
PROCEDURE libera_cassa( i:INTEGER );
BEGIN
     IF i=0 THEN libera_0
     ELSE libera_n( i );
END;
```

Il cliente che esce dalla cassa 0 è il cliente che effettivamente entra nella banca. Lo si inserirà nella coda più corta.

```
PROCEDURE libera_0;
VAR
     i, min, i_min :INTEGER;
BEGIN
     min:=MAXINT;                              { cerca coda più
corta }
     FOR i:=1 TO n_casse DO
          IF c[i].lung_coda<min THEN BEGIN
               min:=c[i].lung_coda; i_min:=i;
          END;

     inserisci_in_coda( banca[0].cli, c[i_min] ); { inserisce
nella coda }
     banca[0].stato:=libero;                  { libera la cassa
0 }
```

```
     banca[0].cli:=NIL;                          { nessun cliente
assegnato }

END;
```

Il cliente che esce dalle altre casse esce definitivamente dalla banca.

```
PROCEDURE libera_n( i:INTEGER );
BEGIN
 { aggiorna il tempo di utilizzo e il numero di clienti serviti
della cassa }


banca[i].t_utilizzo:=banca[i].cli^.t_servizio+banca[i].t_utiliz
zo;
     banca[i].n_cli_ser:=banca[i].n_cli_ser+1;

     banca[i].stato:=libero;            { libera la cassa }
     DISPOSE( banca[i].cli );
END;

PROCEDURE inserisci_in_coda( cl:ptcli; VAR cd:coda_cliente );
BEGIN
     in_coda( cd.coda, cl );  { inserisce in coda  }

{ aggiorna i descrittori della coda }

     cd.dimensione_media:=cd.dimensione_media + cd.lung_coda
                          *          (orologio          -
cd.tempo_ultimo_evento);

     cd.lung_coda:=cd.lung_coda + 1;
     IF            cd.lung_coda>cd.massimo_coda           THEN
cd.massimo_coda:=cd.lung_coda;
     cd.tempo_ultimo_evento:=orologio;
END;
```

Quando si verifica un evento bisogna determinare quali operazioni possono effettuarsi. Per esempio se tutte le casse sono libere, all'arrivo di un cliente una di queste incomincerà a lavorare. Le operazioni possibili nel nostro sistema sono due: se la cassa 0 è libera si deve generare un nuovo cliente e assegnarlo alla cassa. Se una generica cassa è libera e la coda di attesa non è vuota, si deve assegnare il primo cliente in attesa alla cassa.
La procedura di scelta dell'operazione da effettuare sarà:

```
PROCEDURE operazione;
VAR
     i:INTEGER;
BEGIN
     IF banca[0].stato=libero THEN operazione_0;
     FOR i:=1 TO n_casse DO
```

```
          IF (banca[i].stato=libero) AND ( c[i].lung_coda>0 )
THEN
               operazione_n( i );
END;
```

La procedura operazione_0 genera un cliente e lo assegna alla cassa 0.

```
PROCEDURE operazione_0;
BEGIN
     IF orologio<tempo_simulazione THEN BEGIN
          banca[0].cli:=genera_cliente;
          banca[0].clock:=banca[0].cli^.t_arrivo;
          banca[0].stato:=occupato;
     END;
END;
```

La procedura operazione_n inizializza la cassa libera con il primo cliente in coda.

```
PROCEDURE operazione_n( i:INTEGER );
BEGIN
     banca[i].cli:=estrai_da_coda( c[i] );

{ la cassa si libererà quando scadrà il tempo di servizio per
il cliente  }
     banca[i].clock:=orologio+banca[i].cli^.t_servizio;
     banca[i].stato:=occupato;
END;
```

Rimane da affrontare il problema della generazione di un cliente.

```
FUNCTION genera_cliente:ptcli;
VAR
     p_cli:ptcli;

FUNCTION prossimo_arrivo:INTEGER;
VAR
     a:ARRAY[0..3] OF INTEGER;
BEGIN
     a[0]:=3; a[1]:=5; a[2]:=8; a[3]:=12
     prossimo_arrivo:=a[ RANDOM(4) ];
END;

FUNCTION tempo_servizio:INTEGER;
VAR
     s:ARRAY[0..2] OF INTEGER;
BEGIN
     s[0]:=7; s[1]:=10; s[2]:=20;
     tempo_servizio:=s[ RANDOM(3) ];
END;

BEGIN
     NEW(p_cli);
```

```
     p_cli^.t_arrivo:=orologio+prossimo_arrivo;
     p_cli^.t_servizio:=tempo_servizio;
     genera_cliente:=p_cli;
END;
```

Le funzioni prossimo_arrivo e tempo_servizio influenzano in modo decisivo i risultati della simulazione. L'implementazione proposta è molto semplice e si basa sull'utilizzo della funzione RANDOM.

Il programma principale sarà:

```
BEGIN
     inizializza;
     WHILE NOT finita simulazione DO BEGIN
          prossimo_evento;
          operazione;
     END;
     stampa_risultati
END.
```

La simulazione è finita se è scaduto il tempo e non ci sono clienti nella banca.

```
     WHILE (orologio < tempo_simulazione)
               OR NOT code_vuote OR NOT casse_libere
```

dove

```
FUNCTION code_vuote:BOOLEAN;
VAR
     i:INTEGER;
BEGIN
     code_vuote:=TRUE;
     FOR i:=1 TO n_casse DO
          IF c[i].lung_coda>0 THEN code_vuote:=FALSE
END;

FUNCTION casse_libere:BOOLEAN;
VAR
     i:INTEGER;
BEGIN
     casse_libere:=TRUE;
     FOR i:=1 TO n_casse DO
          IF banca[i].stato=occupato THEN casse_libere:=FALSE;
END;
```

inizializzazione delle strutture e stampa dei risultati:

```
PROCEDURE inizializza;
VAR
     i:INTEGER;
BEGIN
     REPEAT
```

```
          WRITE('Numero casse ');
          READLN( n_casse );
     UNTIL (n_casse>0) AND (n_casse<=n_max_casse);

     orologio:=0;

     FOR i:=0 TO n_casse DO BEGIN
          banca[i].clock:=0;
          banca[i].stato:=libero;
          banca[i].n_cli_ser:=0;
          banca[i].t_utilizzo:=0;
          banca[i].cli:=NIL;

          c[i].tempo_ultimo_evento:=0;
          c[i].massimo_coda:=0;
          c[i].lung_coda:=0;
          c[i].tempo_attesa_totale:=0;
          c[i].dimensione_media:=0;
          l_crea( c[i].coda );
     END;
END;

PROCEDURE stampa_risultati;
VAR
     i:INTEGER;
BEGIN
     FOR i:=1 TO n_casse DO BEGIN
          WRITE('MACCHINA ', i );
          WRITE( ' N. clienti serviti ', banca[i].n_cli_ser );
          WRITELN( ' T. utilizzo ', banca[i].t_utilizzo );
     END;
     FOR i:=1 TO n_casse DO BEGIN
          WRITE('CODA ', i );
          WRITELN( 'Massimo coda ', c[i].massimo_coda );
          WRITELN( ' Dimensione media ', c[i].dimensione_media
);
          WRITELN(     '     Tempo     attesa     totale     ',
c[i].tempo_attesa_totale );
     END;
END;
```