FONDAMENTI DI INFORMATICA C

FRANCO ZAMBONELLI

STRUTTURE DATI E LISTE

STRUTTURE DATI

Tipi array e record descrivono:
- strutture dati di *forma* e *dimensione* e *modalitá di accesso,* prefissate;

E i file
- dimensione non prefissata, ma forma e modalità di accesso si'.

Array, record, file, puntatori sono:
- tipi di dati *concreti,* cioè resi disponibili *direttamente dal linguaggio*

Serve poter gestire strutture dati piú complesse:
- non gestite direttamente dal linguaggio!

Si deve introdurre il concetto di tipo di dato astratto **(ADT = Abstract Data Type)**

Si tratta di definire strutture dati
- Con "forma" e dimensioni definibili a piacere
- Con operazioni definibili

Definizione operazionale:
- corredata da un insieme di procedure per operare su tali tipi,

STRUTTURE DATI

Le strutture tipi di dato che vedremo rappresentano modi di organizzare insiemi di informazione

I tipi di dato astratto che vedremo sono definiti dai seguenti cinque componenti:
- un insieme di *atomi,* determinati dalla informazione elementare da registrare e organizzare insieme ad altre,
- una *relazione strutturale*, che identifica la struttura globale della organizzazione
- un insieme di *posizioni,* che indicano dove sono contenuti gli atomi nella struttura,
- una *funzione* di valutazione, che associa atomi a posizioni
- un insieme di *operazioni* (*procedure* e *funzioni*), che specificano *cosa* si può fare sui dati e sulla struttura.

Le procedure e funzioni possono essere *elementari* oppure ottenute per combinazione di operazioni elementari.

Esempio: mano di bridge
1. *atomi:* carte con 4 semi e 13 valori
2. *posizioni:* da 1 a 13 per ciascuno di 4 giocatori, nessuna relazione tra le posizioni
3. *operazioni:* creazione di una mano, giocata di una carta da parte di un giocatore, ...

Scrittura modulare del software

Descriveremo:
- Strutture dati *classiche*
- procedure che implementano le operazioni *base*.

Le procedure *base* non esauriscono le possibili manipolazioni → possono essere *combinate* per realizzare operazioni più complesse.

La tecnica della combinazione di moduli ha numerosi vantaggi:
- scrittura più rapida dei programmi → problema complesso come combinazione di problemi semplici
- programmi più leggibili
- programmi più verificabili: i moduli semplici sono già collaudati.

Le strutture dati sono *astratte,* cioè svincolate da specifiche applicazioni, ma possono essere rapidamente adattate a problemi concreti (messaggio interno → messaggio esterno).

Problema degli errori e della robustezza delle procedure:
- trattare anche condizioni di errore (p.e. procedure con parametri non corretti)
- le procedure mostrate hanno una individuazione interna degli errori
- gestione di un parametro di errore, per permettere al programma chiamante di prendere eventuali contromisure.

Le procedure e funzioni
- organizzate in *librerie*
- suddivise per tipo di dato e tipo di implementazione
- costituite dai file, *header* (`<nomelibreria>.h`) e file con il codice (`<nomelibreria>.c`)

STRUTTURE LINEARI: LISTA

Una ampia classe di problemi può essere risolta con strutture lineari, cioè strutture i cui elementi possono essere messi in relazione di ordinamento totale (isomorfismo con i numeri naturali!)

Lista semplice e, con riferimento alla definizione di tipo di dato astratto possiamo individuare atomi, posizioni ed operazioni nel modo seguente:

- *atomi* può essere un insieme qualunque: interi, reali, caratteri, stringhe, record, ...
- *posizioni* è un insieme dotato di una *relazione d'ordine lineare:* esistono un primo ed un ultimo elemento; tutti gli altri hanno un predecessore ed un successore esempio: i numeri naturali {1,2,...,n}
- *operazioni* di base

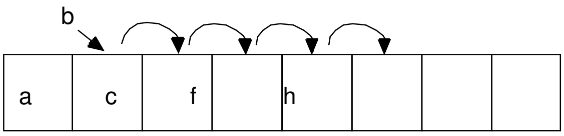
Operazioni Base su una Lista

operazione	*Esempio di possibile intestazione delle funzioni in C*
Crea una lista vuota	`void ListaCrea(Lista *L);`
vero se la lista è vuota	`boolean ListaVuota(Lista L);`
restituisce la prima posizione	`Posiz Primo(Lista L);`
restituisce l'ultima posizione	`Posiz Ultimo(Lista L);`
restituisce la posizione successiva a P	`Posiz SuccL(Posiz P, Lista L);`
restituisce la posizione precedente a P	`Posiz PrecL(Posiz P, Lista L);`
restituisce l'atomo della posizione P	`int Recupera(Posiz P, Lista L, Atomo *A);`
modifica l'atomo della posizione P in A	`int Aggiorna (Atomo A, Posiz P, Lista *L);`
cancella l'atomo della posizione P	`int Cancella(Posiz P, Lista *L);`
inserisce un nuovo atomo prima della posizione P	`int InserPrima(Atomo A, Posiz P, Lista *L);`
inserisce un nuovo atomo dopo la posizione P	`int InserDopo(Atomo A, Posiz P, Lista *L);`
restituisce la lunghezza della lista	`int Lungh(Lista L);`

Implementazione di lista con array

Gli elementi della lista sono memorizzati in un array, con elemento base Atomo e le posizioni sono costituite da valori dell'indice compresi fra 1 e il numero di elementi presenti.

- per motivi di efficienza, si memorizza la lunghezza L della lista: le posizioni saranno da 1 a L
- si usa la pseudo-posizione 0, quando la lista è vuota
- la capienza della lista è determinata dal dimensionamento dell'array
- l'operazione di inserimento dopo una specifica posizione P richiede:
  1. creazione dello spazio, spostando in avanti gli elementi successivi a P, secondo lo schema di figura.
  2. inserimento effettivo



Implementazione di liste con array in C

Si noti che, per i motivi di efficienza, qui viene usato il parametro variabile `*L` anche quando non si devono effettuare modifiche sulla lista. Ad esempio, si usa:

```
Posiz SuccL(Posiz P, Lista *L);
```

invece di

```
Posiz SuccL(Posiz P, Lista L);
```

in quanto, in questo secondo caso, nel record di attivazione di `SuccL` si dovrebbe predisporre per il parametro `L` uno spazio, in byte, pari a

```
sizeof(int) + MaxDim*sizeof(Atomo)
```

ATOMI

I dettagli dell'atomo sono racchiusi in una unità, `InfoBase`, utilizzata per la compilazione sia del programma chiamante che della unità delle liste. Si noti che:

- l'elemento in posizione P della lista sarà memorizzato nella posizione P-1 dell'array.

```
/* Infobase.h  */
#include <stdio.h>
#define MaxDim 10
#define Null 0  /* elemento particolare */

typedef int boolean;
typedef int Atomo;

extern void Acquisisci(Atomo *A);
extern void Visualizza(Atomo A);/

/* Infobase.c  */
#include <stdio.h>
#include "InfoBase.h"

void Acquisisci(Atomo *A){
  ...
} /* end Acquisisci */

void Visualizza(Atomo A){
  ....
} /*   end  Visualizza */
```


FILE HEADER DELLA LISTA

NOTA: per la gestione degli errori, si memorizza nella lista:

- un campo `ListaStato` per contenere le informazioni sullo stato raggiunto dopo l'esecuzione delle varie operazioni fatte sulla lista
- una funzione `ListaErrore` che genera una stringa informativa in caso di errore.

```
/* ListaArr.h  */
#define ListaNoSucc 1 /* Codici di stato    */
#define ListaNoPrec 2 /* Assegnati a
ListaStato come */
#define ListaPosErr 3 /* risultato delle
operazioni  */
#define ListaPiena 4  /* soggette ad errore
   */
#define ListaOK 0    /* Terminazione senza
errori */
#define NoPosiz 0     /* Posizione non valida
   */

typedef int Posiz;/*0, pseudo-posiz. per lista
vuota */

typedef struct {
   Posiz Lungh;
   Atomo Dati[MaxDim];
} Lista;
```



```
extern void ListaCrea(Lista *L);
extern boolean ListaVuota(Lista *L);
extern Posiz Primo(Lista *L); /* prima
posizione */
extern Posiz Ultimo(Lista *L); /* ultima
posizione */
extern Posiz SuccL(Posiz P, Lista *L);
      /* restituisce la posizione successiva a
P */
extern Posiz PrecL(Posiz P, Lista *L);
      /* restituisce la posizione precedente a
P */
extern int Recupera(Posiz P, Lista *L, Atomo
*A);
      /* restituisce in A l'atomo della
posizione P */
extern int Aggiorna (Atomo A, Posiz P, Lista
*L);
      /* modifica l'atomo della posizione P in
A */
extern int Cancella(Posiz P, Lista *L);
      /* cancella l'atomo della posizione P */
extern int InserDopo(Atomo A, Posiz P, Lista
*L);
    /* inserisce un nuovo atomo dopo la
posizione P */
extern int InserPrima(Atomo A, Posiz P, Lista
*L);
/* inserisce un nuovo atomo prima della
posizione P */
extern int Lungh(Lista *L); /*lunghezza della
lista */
extern char *ListaErrore (Lista *L);
extern int ListaStato;
```


IMPLEMENTAZIONE FUNZIONI

```
/* ListaArr.c */
#include "Infobase.h"
#include "ListaArr.h"

int ListaStato=0;

void ListaCrea(Lista *L){
 L->Lungh=0;
}          /* end ListaCrea */

boolean ListaVuota(Lista *L){ /* *L per
economia */
 return (L->Lungh==0);
}          /* end ListaVuota */

Posiz Primo(Lista *L){ /* *L per economia */
 if (L->Lungh==0)
   return NoPosiz;
 else
   return 1;
}         /* end Primo */

Posiz Ultimo(Lista *L){ /* *L per economia */
 if (L->Lungh==0)
   return NoPosiz;
 else
   return L->Lungh;
}         /* end Ultimo */
```

```
Posiz SuccL(Posiz P, Lista *L){ /* *L per
economia */
 if ( (P<1) || (P>=L->Lungh))/* P<1 non e`
valida */
 {   /* l'ultimo non ha successore     */
  ListaStato = ListaNoSucc;
  return NoPosiz;
 }
 else{
  ListaStato = ListaOK;
  return (++P); /* !! (P++) NON VA BENE
PERCHE'.. */
 }
}     /* end SuccL */

Posiz PrecL(Posiz P, Lista *L){
 if ( (P<=1) || (P>L->Lungh)) /* P=1 non è
valida  */
 {      /* il primo non ha precedenti  */
  ListaStato = ListaNoPrec;
  return NoPosiz;
 }
 else{
  ListaStato = ListaOK;
  return (--P);
 }
}     /* end SuccL */
```



```
int Recupera(Posiz P, Lista *L, Atomo *A){
                              /* *L per econ. */
 if ( (P<1) || (P>(L->Lungh))) /* pos. non
valida */
    ListaStato = ListaPosErr;
 else{
   ListaStato = ListaOK;
   *A=L->Dati[P-1];
 }
 return ListaStato;
}     /* end Recupera */

int Aggiorna(Atomo A, Posiz P, Lista *L){
 if ((P<1) || (P>L->Lungh)) /* pos. non
valida */
    ListaStato = ListaPosErr;
 else{
    ListaStato = ListaOK;
    L->Dati[P-1]=A;
 }
 return ListaStato;
}      /* end Aggiorna */

int Cancella(Posiz P, Lista *L){
 Posiz I;
 if ( (P<1) || (P>L->Lungh)) /* pos. non
valida */
    ListaStato = ListaPosErr;
 else{
    ListaStato = ListaOK;
    for (I=P; I<L->Lungh;I++) /*
compattamento */
      L->Dati[I-1]=L->Dati[I];
    L->Lungh--; /* decremento di lunghezza */
 }
 return ListaStato;
}      /* end Cancella */
```



```
int InserDopo(Atomo A, Posiz P, Lista *L){
 Posiz I;
 if ( (P< 0) || (P>L->Lungh)|| ((L-
>Lungh)==MaxDim))
   if ((L->Lungh)==MaxDim)
     ListaStato = ListaPiena;
   else
     ListaStato = ListaPosErr;
 else{
  ListaStato = ListaOK;
  for (I=L->Lungh;I>P;I--) /* crea spazio */
         L->Dati[I]=L->Dati[I-1];
  L->Dati[I] = A;
  L->Lungh++; /* incremento di lunghezza */
 }
 return ListaStato;
}      /* end InserDopo */
```



```
int InserPrima (Atomo A, Posiz P, Lista *L){
 Atomo Temp;
 if ( (P< 0) || (P>L->Lungh)|| ((L-
>Lungh)==MaxDim))
   if ((L->Lungh)==MaxDim)
      ListaStato = ListaPiena;
   else
    ListaStato = ListaPosErr;
 else{    /* la posizione e' accettabile */
   ListaStato = ListaOK;
   if (ListaVuota(L))
      InserDopo(A,P,L);
   else{ /* inserisce dopo e scambia i due
atomi */
    InserDopo(A,P,L);
    Recupera(P,L,&Temp);
    Aggiorna(A,P,L);
    Aggiorna(Temp,SuccL(P,L),L);
   }
 } /* end if la posizione e' accettabile */
 return ListaStato;
} /* end InserPrima */
```

```
char *ListaErrore (){
  switch(ListaStato){
    case  ListaNoSucc :
      return "Posizione errata per SuccL";
    break;
    case  ListaNoPrec :
      return "Posizione errata per PrecL";
    break;
    case  ListaPosErr :
      return "Posizione errata per lista";
    break;
    case  ListaPiena  :
      return "Lista Piena";
  }
  return "Stato errato";
}     /* end ListaErrore */

/* Lunghezza lista */
int Lungh(Lista *L) {
  return L->Lungh;
} /* end Lungh */
```

Alcune funzioni sono “ovvie”→ utile isolare completamente la funzione realizzata rispetto all'implementazione ed alla struttura dati utilizzata.

Schema generale per la gestione degli errori:

1. le funzioni che restituiscono una posizione, in caso di errore restituiscono la posizione *nulla* (`NoPosiz`) e aggiornano la variabile di stato;
2. le altre funzioni restituiscono direttamente il valore della variabile di stato, che può essere immediatamente esaminato dal programma chiamante.

Complessità computazionale delle operazioni di lista su array

Le operazioni `InserDopo` e `Cancella` contengono cicli il cui numero di ripetizioni nel caso peggiore è uguale al numero di elementi della lista (a meno di costanti additive).

Queste due operazioni e `InserPrima` (che usa `InserDopo`) hanno complessità asintotica *O(n)*. Tutte le altre operazioni non hanno cicli o ricorsioni, quindi hanno complessità costante.

Implementazione di lista con puntatori

Le posizioni costituite da *puntatori*.

Ogni elemento è un record che contiene:

- un atomo
- il puntatore all'elemento successivo

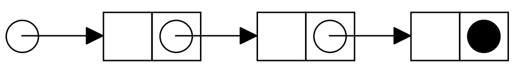
Occorrerà memorizzare a parte la posizione del primo elemento (o, per comodità, anche la posizione dell'ultimo e la lunghezza della lista).

Il puntatore realizza la relazione di *successore* che sussiste tra due elementi consecutivi.

L'ultimo elemento ha come successivo, la pseudo-posizione `NULL.`

Non c’e’ limite di dimensionamento a priori:

- tutti i dati, a parte il puntatore al primo elemento, risiedono nella parte dinamica della memoria.



Strutture Ricorsive

Nella implementazione di una lista con puntatori è necessario definire un tipo struttura *ricorsivo*:

- un tipo struttura (Cella) in cui un campo (Prox) è un puntatore alla struttura stessa.

Bisogna definire l’elemento di una lista cosí:

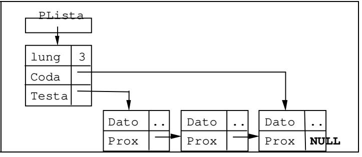
```
typedef struct Tag_Cella{
  struct Tag_Cella *Prox;
  Atomo Dato;
}Cella;
```

In questa implementazione viene memorizzato, oltre al puntatore al primo elemento (testa della lista) anche il puntatore all’ultimo elemento della lista (coda della lista) e il numero di elementi che costituisce la lista (lunghezza della lista).

Non sono necessari ma fanno comodo.

```
typedef  Cella *Posiz;

typedef struct{
 int Lungh;
 Posiz Coda,Testa;
} Lista;
```

```
Lista *PLista;
```



Per individuare il dato del primo elemento della lista, partendo dal puntatore

```
(*(*PLista).Testa).Dato
```

oppure

```
PLista->Testa->Dato
```


FILE HEADER DELLA LISTA

```
/* ListaPunt.h */
#define ListaNoSucc 1 /* Codici di stato */
#define ListaNoPrec 2
#define ListaPosErr 3
#define NoPosiz NULL
#define ListaOK 0

typedef struct  TCella{
 struct TCella  *Prox;
 Atomo Dato;
}Cella;

typedef  Cella *Posiz;

typedef struct{
 int Lungh;
 Posiz Coda,Testa;
} Lista;
```



```
extern void ListaCrea(Lista *L);
extern boolean ListaVuota (Lista *L);

extern Posiz Primo(Lista *L);
extern Posiz Ultimo(Lista *L);
extern Posiz SuccL(Posiz P, Lista *L);
extern Posiz PrecL(Posiz P, Lista *L);
extern int Recupera(Posiz P, Lista *L, Atomo
*A);
extern int Aggiorna (Atomo A, Posiz P, Lista
*L);
extern int Cancella(Posiz P, Lista *L);
extern int InserDopo(Atomo A, Posiz P, Lista
*L);
extern int InserPrima (Atomo A, Posiz P, Lista
*L);
extern int Lungh(Lista *L);
extern void InserOrdinato(Atomo A, Lista *L);
extern char *ListaErrore (Lista *L);
extern void CreaListaIterativo(Cella *PIniz,
int N);

extern int ListaStato; /* Variabile di stato
*/
```

NOTA: abbiamo mantenuto la stessa “interfaccia” di uso. E’ possibile scrivere un programma che fa uso di liste e scegliere una delle due implementazioni compilando senza modificare il programma stesso.

IMPLEMENTAZIONE DELLE FUNZIONI

```
#include "Infobase.h"
#include "ListaPunt.h "

void ListaCrea(Lista *L){
 L->Lungh=0;
 L->Testa=NULL;
 L->Coda=NULL;
}  /* end ListaCrea */

boolean ListaVuota (Lista *L){
   return (L->Lungh==0);
}  /* end ListaVuota */

Posiz Primo(Lista *L){
   if (L->Lungh==0)
     return NoPosiz;
   else
     return L->Testa;
} /* end Primo */

Posiz Ultimo(Lista *L){
   if (L->Lungh==0)
     return NoPosiz;
   else
     return L->Coda;
}    /* end Ultimo */
```

```
Posiz SuccL(Posiz P, Lista *L){
  if ((P==L->Coda) || (P==NULL))
      { /* l'ultimo non ha successore */

        ListaStato = ListaNoSucc;
        return NoPosiz;
      }
  else
      {
        ListaStato = Lista0K;
        return P->Prox;
      }
}     /* end SuccL */

Posiz PrecL(Posiz P, Lista *L){
  Posiz Q;
  if ((P==L->Testa) || (P==NULL))
      { /* il primo non ha precedenti */

       ListaStato = ListaNoPrec;
       return NoPosiz;
      }
  else
      { /* cerca posiz. prec. a partire dalla
testa */
       ListaStato = Lista0K;
       Q=L->Testa;
       while (1) {
        if (Q==L->Coda)
          { /* e' giunto in coda senza trovare
*/
           ListaStato = ListaNoPrec;
           return NoPosiz;
          }
          else
           if (Q->Prox==P)
             return Q;
          Q=Q->Prox; /* itera la ricerca */
       } /* end while */
      } /* end if */
} /* end PrecL */
```



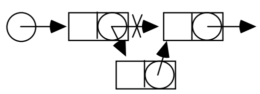
```
int Recupera(Posiz P, Lista *L, Atomo *A){
/* il controllo di validita' di P e' limitato
*/
/* ma un controllo completo avrebbe costo O(n)
*/
  if (P==NULL)
       ListaStato = ListaPosErr;
  else
    {
     ListaStato = Lista0K;
     *A=P->Dato;
    }
  return ListaStato;
}     /* end Recupera */

int Aggiorna(Atomo A, Posiz P, Lista *L){
  if ((L->Lungh==0) || (P==NULL)) /* pos. non
valida */
     ListaStato = ListaPosErr;
  else {
     ListaStato = Lista0K;
     P->Dato=A;
    }
  return ListaStato;
}     /* end Aggiorna */
```


Inserimento e cancellazione in lista a puntatori

Le operazioni di inserimento e cancellazione non hanno la necessità di *creare lo spazio* o di *compattare*:

- i vari atomi sono contigui soltanto **logicamente** tramite i riferimenti dei puntatori;

L'inserimento *dopo* una data posizione viene effettuato modificando il puntatore al prossimo atomo situato in quella posizione, come mostrato dalla figura




```
int InserDopo(Atomo A, Posiz P, Lista *L){
/* si suppone P valida se la lista non è
vuota*/
  Posiz Q;                if (L->Testa==NULL)
/* ins. in testa, ignorando P */
      {
        L->Testa=malloc(sizeof(Cella));
        L->Testa->Dato = A;
        L->Testa->Prox = NULL;
        L->Coda=L->Testa;
      }
  else
      {
        Q=malloc(sizeof(Cella));
        Q->Dato=A;     /* inserisce nuovo */
        Q->Prox=P->Prox;  /* sistema i
puntatori */
        P->Prox=Q;
        if (P==L->Coda)
           L->Coda=Q; /* inserimento in coda */
      }
  L->Lungh++;
  ListaStato = Lista0K;
  return ListaStato;
}     /* end InserDopo */
```

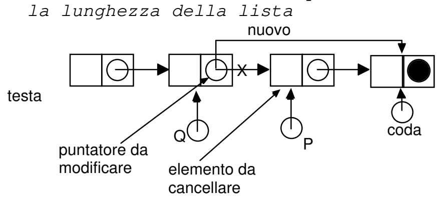
```
int InserPrima (Atomo A, Posiz P, Lista *L){
 Atomo Temp;
 if (ListaVuota(L))
    InserDopo(A,P,L);
 else { /* inserisce dopo e scambia i due
atomi */
    InserDopo(A,P,L);
    Recupera(P,L,&Temp);
    Aggiorna(A,P,L);
    Aggiorna(Temp,SuccL(P,L),L);
 }
 ListaStato = Lista0K;
 return ListaStato;
}     /* end InserPrima */
```


CANCELLAZIONE ELEMENTO

La cancellazione dell'elemento di posizione P, richiede la modifica dell'elemento precedente, come mostrato in figura

NOTA: la lista con puntatori ha un verso di percorrenza preferenziale: l'elemento precedente ad una posizione data si trova ripercorrendo la lista dall'inizio

```
- se la lista non è vuota e la posizione P data è
  valida
  - se l'elemento da cancellare è quello di testa
    - se la lista aveva lunghezza unitaria
                (viene vuotata)
      - vuota la lista aggiornando testa e coda
      altrimenti
      - il secondo elemento diventa quello di testa
    altrimenti
   - cerca la posizione Q precedente all'elemento
     da cancellare
    - aggiorna il campo prossimo di Q con il campo
     prossimo di P
    - se P era la coda, Q diventa la nuova coda
  - rilascia l'elemento della posiz. P e decrementa
    la lunghezza della lista
```



L'algoritmo di cancellazione in C

```
int Cancella(Posiz P, Lista *L){
 Posiz Q;
 if ((L->Lungh==0) || (P==NULL))
   /* posizione non e` valida */
   ListaStato = ListaPosErr;
 else { /* cancella: lista non vuota e pos.
valida */
    ListaStato = Lista0K;
    if (P==L->Testa)
       if (L->Lungh==1)
          { /* eliminazione dell'unico
elemento */
            L->Testa=NULL; /* lista diventa
vuota */
            L->Coda=NULL;
          }
       else
          L->Testa=L->Testa->Prox;
    else   /* elim. elemento qualunque su   */
       {  /* lista di due o piu' */
          Q=PrecL(P,L);
          Q->Prox=P->Prox;
          if (L->Coda==P)
          L->Coda=Q;
       }
    free(P);
    L->Lungh--;
 } /* end cancellazione */
 return ListaStato;
} /* end Cancella */
```


NOTE:

Come casi particolari:

- inserimento del primo elemento in una lista vuota
- inserimento in testa alla lista.

Il primo caso viene trattato automaticamente dalla procedura `InserDopo` ed è sufficiente specificare, come posizione, il primo elemento, con `InserDopo(A,Primo(L),L)`.

Il secondo caso invece richiede un inserimento in *prima posizione*, con `InserPrima(A,Primo(L),L)`.

Le procedure `InserPrima` , `Lungh` e `ListaErrore` sono implementate come in `ListeArr`.

Complessità computazionale

L'operazione `PrecL` ha un ciclo al suo interno il cui numero di ripetizioni nel caso è uguale al numero di elementi nella lista (a meno di costanti additive), quindi `PrecL` e `Cancella` (che fanno uso di `PrecL`) hanno complessità asintotica $O(n)$. Tutte le altre operazioni hanno complessità costante.

LISTA ORDINATA

Si vuole che la relazione fra le posizioni corrisponda ad una *relazione d'ordine totale* fra gli elementi.

Si suppone di disporre di una funzione `Minore` che prende come parametri due atomi e restituisce il valore *vero* se il primo precede il secondo, secondo la relazione d'ordine voluta.

Si può scrivere una unità di programma generalizzata facilmente adattabile a qualunque tipo di confronto si voglia effettuare su qualunque tipo di dato.

L'inserimento ordinato di un atomo `A` può essere ricavato con una combinazione di operazioni base, secondo il seguente algoritmo:

```
- se la lista è vuota effettua un normale
  inserimento
   altrimenti
   - cerca posizione del primo elemento non
  minore di A
       (se l'elemento non esiste trova la coda)
   - se l'elemento della posizione trovata è
  minore
     - inserisci dopo la posizione trovata (in
  coda)
     altrimenti
     - inserisci prima della posizione trovata
```


L'algoritmo si traduce in C con la seguente procedura:

```
int InserOrdinato(Atomo A, Lista *L){
/* Inserimento ordinato in lista    */
/* Indipendente dal tipo di implement. della
lista*/
/* Usa la funzione "Minore" dipendente da
Atomo */

 Atomo Temp;
 Posiz P,U;
 P=Primo(L);
 U=Ultimo(L);

 if (ListaVuota(L))
    InserDopo(A,P,L);
 else
   {/* cerca la pos. del primo elem. non minore di A
*/
      Recupera(P,L,&Temp);
      while ((Minore(Temp,A)) && P!=U)
      { /* se non trova el. > Temp. arresta con P=U
*/
         P=SuccL(P,L);
         Recupera(P,L,&Temp);
      }
      if (Minore(Temp,A))
         InserDopo(A,P,L);
      else
         InserPrima(A,P,L);
 } /* cerca la pos. del primo elem. non minore di A
*/
}    /* end InserOrdinato */
```


Lista ordinata su array

L'inserimento ordinato è indipendente dal tipo di implementazione della lista, facendo uso soltanto delle operazioni base di lista

Possibilità di sfruttare la conoscenza di una specifica implementazione per giungere ad una soluzione più efficiente. Nel caso dell'implementazione con array si possono sfruttare due fatti:

- per effetto dell'ordinamento, dato un indice `i` dell'array, tutti gli atomi corrispondenti a indici inferiori a `i` sono minori di quello corrispondente a `i` (e viceversa per quelli superiori)
- il tipo di dato array ammette *accesso diretto* ai componenti sulla base dell'indice.

È cosí possibile ricercare la posizione di inserimento tenendo conto dell'implementazione, anzichè in termini di operazioni di base.

```
- inizializza estremi sequenza con P=Primo e
  U=Ultimo
- finchè P < U
  - calcola la posizione centrale C
  - recupera l'elemento Temp corrispondente a C
  - se Temp è minore di A
     - ricerca nella metà superiore, ponendo P=C+1
    altrimenti
    - ricerca nella metà inferiore, ponendo U=C-1
```



```
int InserOrdinato(Atomo A, Lista *L){
/* Inserimento ord. in lista: implem. su array
 */
/* Cerca il primo elem. non < con tecnica
dicotomica*/
/* Usa la funz. "Minore" dipendente dal tipo
di Atomo */
 Atomo Temp;
 Posiz Inizio,Fine,C;

 if (ListaVuota(L))
   InserDopo(A,NoPosiz,L);
 else {  /* cerca la pos. del primo elem. non
< di A */
     /* la ricerca inizia tra Primo(L) e
Ultimo(L) */
   Inizio=Primo(L);
   Fine=Ultimo(L);
   while (Inizio<=Fine) {
     C=(Inizio+Fine)/2;
     Recupera(C,L,&Temp);
     if (!Minore(Temp,A)&&!Minore(A,Temp)){
       InserDopo(A,C,L);
       return ListaOK;
     }
     else
       if (Minore(Temp,A))
         Inizio=C+1;
       else
         Fine=C-1;
   }
```

```
  /*  se l'elemento della posizione trovata è
minore */
 /*   - inserisci dopo la pos. trovata (in
coda)     */
 /*    altrimenti inserisci prima della pos.
trovata */
 /* Inizio e' uguale alla pos. del primo
elem. non   */
 /* < di A  o, se tale elem. non c'e', a
Ultimo(L)+1 */

 if (Inizio<=Ultimo(L))
    InserPrima(A,Inizio,L);
   else
    InserDopo(A,Ultimo(L),L);
 }
 return ListaOK;
}     /* end InserOrdinato */
```

La complessità asintotica di questa procedura ha un termine logaritmico per la ricerca, cui si somma un termine lineare (predominante) per l'inserimento.

Esempio di utilizzo della struttura dati Lista

Il programma è costituito da un ciclo di presentazione di un *menu* e dall'esecuzione della relativa operazione sulla lista.

L'algoritmo è il seguente:

```
– inizializza la lista
– ripeti
  – prendi un comando da input
  – se il comando è
    – inserimento in testa
      – ripeti
        – prendi un intero da input
         - inseriscilo in testa
         - in caso di insuccesso: messaggio di
             errore
        finché l'intero è diverso da zero
    – inserimento in coda
      – ripeti
        – prendi un intero da input
        – inseriscilo in coda
        – in caso di insuccesso: messaggio di errore
        finché l'intero è diverso da zero
    – inserimento in posizione data
      – prendi da input la posizione N di
inserimento
      – inizializza P alla prima pos. della lista
      – ripeti N–1 volte
        – metti in P il successore di P
      – prendi un intero da input
      – inseriscilo dopo la posizione P
      – in caso di insuccesso: messaggio di errore
    – eliminazione
      – prendi da input la pos. N di cancellazione
      – inizializza P alla prima pos. della lista
      – ripeti N–1 volte
        – metti in P il successore di P
      – cancella la posizione P
      – in caso di insuccesso: messaggio di errore
    – visualizzazione
      – inizializza P alla prima pos. della lista
      – ripeti
        – recupera l'elemento in pos. P
         - visualizzalo
```



```
        - metti in P il successore di P
      finché non si verifica un errore
  finché comando è diverso da fine
```

Struttura infobase

```
/* InfoBase.h */
#define MaxDim 10
#define Null 0  /* elemento particolare */

typedef int boolean;
typedef int Atomo;

extern void Acquisisci(Atomo *A);
extern void Visualizza(Atomo A);
extern int Minore(Atomo A,Atomo B);
extern boolean AtomoNullo(Atomo A);
```



```
/* InfoBase.c */
#include <stdio.h>
#include "InfoBase.h"

void Acquisisci(Atomo *A){
 char linea[80];
 printf("inserire un intero ");
 gets(linea);  /* prende tutta la linea */
 sscanf(linea,"%d", A); /* prende numero da
linea */
   /* la prox. lettura iniziera' da una linea
nuova */
} /* end Acquisisci */

void Visualizza(Atomo A){
 printf("%d\n",A);
}
/*   end  Visualizza */

int Minore(Atomo A,Atomo B){
 return (A<B);
}

boolean AtomoNullo(Atomo A){
 return A==0;
}
```

Il programma di esempio:

```
#include "Infobase.h"
#include "ListaArr.h"
#define InsTesta 'T'
#define InsCoda 'C'
#define InsPos 'P'
#define Elimina 'E'
#define Visualizz 'V'
#define Fine 'F'

int main(){
 Lista L;
 Atomo A;
 int n,i;
 Posiz p;
 char Comando[80];
         /* per una intera linea di Comando
*/
 char C;
```



```
ListaCrea(&L);

 do{
  printf("Comando? ");
  gets(Comando); /* carica la linea di
comando,
          considerando solo il primo
carattere */
  C = Comando[0];
  if (C>='a') C-=32; /* converte in maiuscolo
*/
  switch (C){

   case InsTesta :
        printf("inser. in testa\n");
        do{
         Acquisisci(&A);
         if(!AtomoNullo(A))
          if(InserPrima(A,Primo(&L),&L)
              !=ListaOK)
           printf("non inserito\n");
        } while(!AtomoNullo(A));
        break;

   case InsCoda :
        printf("inserimento in coda\n");
        do{
         Acquisisci(&A);
         if(!AtomoNullo(A))
```



```
 if(InserDopo(A,Ultimo(&L),&L)!=ListaOK)
              printf("non inserito\n");
         } while(!AtomoNullo(A));
         break;

   case InsPos :
         printf("n. d'ordine dopo cui
inserire? ");
         gets(Comando);
         sscanf(Comando,"%d",&n);
         if (n<0){
          printf("solo non negativi,
prego!\n");
          break;
         }
         if (n>Lungh(&L)){
          printf("Lista troppo corta\n");
          break;
         }
         p = Primo(&L);
         for (i=1;i<n;i++) p=SuccL(p,&L);
         Acquisisci(&A);
         if (InserDopo(A,p,&L)!=ListaOK)
          printf("non inserito\n");
         break;
```



```
   case Elimina :
        printf("numero d'ordine da
eliminare? ");
        gets(Comando);
        sscanf(Comando,"%d",&n);
        if (n<0){
         printf("solo non negativi,
prego!\n");
         break;
        }
        if (n>Lungh(&L)){
         printf("Lista troppo corta\n");
         break;
        }
        p = Primo(&L);
        for (i=1;i<n;i++) p=SuccL(p,&L);
        Cancella(p,&L);
        if (ListaStato!=ListaOK)
         printf("non cancellato\n");
        break;
```

```
     case Visualizz:
           if (!ListaVuota(&L)){
             p=Primo(&L);
             do{
               Recupera(p,&L,&A);
               Visualizza(A);
               p=SuccL(p,&L);
             } while (ListaStato==ListaOK);
           }
           break;
     case Fine:    break;
     default : printf("Comando errato\n");
   }
  } while (C!=Fine);

 printf("Fine lavoro\n");
 return 0;
}
```