Rappresentazione di liste mediante puntatori in linguaggio C

- Specifica del tipo degli elementi

```
typedef ... TipoElemLista;

struct StructLista {
  TipoElemLista key;
  struct StructLista *next;
};

typedef struct StructLista TipoNodoLista;
typedef TipoNodoLista *TipoLista;
```

- *key* contiene l'elemento della lista
- *next* contiene il puntatore alla struttura che rappresenta l'elemento successivo
- Una lista vuota è rappresentata dal puntatore NULL, una lista non vuota è rappresentata dal puntatore al suo elemento iniziale

Implementazione di operazioni elementari

- Inizializzazione (la variabile lis è un puntatore doppio)

```
void InitLista(TipoLista *lis)
{
  *lis = NULL;
}
```

- Inserimento di un elemento in testa

```
/* pseudo-code like */
TipoLista Inserisci(TipoLista lis, TipoLista nodo) {
  nodo->next = lis;
  return nodo;
}

/*free-style */
void InserisciTestaLista(TipoLista *lis, TipoElemLista elem) {
  TipoLista paux;

  paux = malloc(sizeof(TipoNodoLista));
  paux->key = elem; paux->next = *lis;
  *lis = paux;
}
```

Implementazione di operazioni elementari

- Inserimento di un elemento in testa non distruttivo

```
TipoLista InserisciTestaListaND(TipoLista *lis, TipoElemLista elem)
{
  TipoLista paux;

  paux = malloc(sizeof(TipoNodoLista));
  paux->info = elem;
  paux->next = *lis;
  return paux;
}
```

- Test di lista vuota

```
bool TestListaVuota(TipoLista lis)
{ return (lis == NULL); }
```

- Restituzione dell'elemento in testa

```
void TestaLista(TipoLista lis, TipoElemLista *elem)
{
  if (lis != NULL)
    *elem = lis->info;
}
```

Implementazione di operazioni elementari

- Resto e Resto non distruttivo

```
void RestoLista(TipoLista *lis)
{
  if (*lis != NULL)
    *lis = (*lis)->next;
}

TipoLista RestoListaND(TipoLista *lis)
{
  TipoLista paux;
  if (*lis != NULL)
    paux = (*lis)->next;
  else paux = NULL;

  return paux;
}
```

Implementazione di operazioni elementari

- Rimozione del primo elemento

```
void CancellaPrimoLista(TipoLista *lis)
{
  TipoLista paux;
  if (*lis != NULL) {
    paux = *lis;
    *lis = (*lis)->next;
    free(paux);
  }
}
```

- Tutte le funzioni distruttive modificano la lista su cui operiamo e quindi per mantenere una lista integra operiamo su una copia
- Esigenze del problema sono determinanti

Funzioni supplementari

- Copia lista
  Provate a farla per esercizio.

```
void CopiaLista(TipoLista lis, TipoLista *copia);
```

- Inserire in coda e stampare in ordine inverso

```
void InserisciCodaLista(TipoLista *lis, TipoElemLista elem)
{
  TipoLista ultimo;
  TipoLista paux;

  paux = malloc(sizeof(TipoNodoLista));
  paux->info = elem; paux->next = NULL;
  if (*lis == NULL)
    *lis = paux;
  else {
    ultimo = *lis;
    while (ultimo->next != NULL)
      ultimo = ultimo->next;
    ultimo->next = paux;
  }
}
```

```
void StampaListaInversaRic(TipoLista lis)
{
  TipoLista suc; suc = lis;

  if (suc != NULL)
    {
      TipoElemLista x;
      x = suc->info;
      suc = (suc)->next;
      StampaListaInversaRic(&suc);
      printf(" %d ",x);
    }
}
```

- Calcola la lunghezza di una lista (Fate la versione ricorsiva per esercizio)

```
int LunghezzaLista(TipoLista lis) {
  int length=0;
  TipoLista papp=lis;

  while (papp != NULL)
    {
      length++;  papp = papp->next;
    }
  return length;
}
```

- Reverse ricorsivo

```
void List_Invert_Ric(TipoLista *lis)
{
  TipoLista papp;

  if (*lis != NULL)
    {
      int appo = (*lis)->info;

      RestoLista(lis);
      List_Invert_Ric(lis);
      InserisciCodaLista(lis, appo);
    }
}
```

- La lista viene distrutta e ricostruita.
- Non è necessario lavorare su una copia.

- IN GENERALE: Uso della ricorsione è più semplice da un punto di vista concettuale

Esercizio

- Date n liste collegate contenenti m interi distinti e un intero x, scrivere lo pseudo-codice di una funzione che conta le occorrenze di x nelle n liste.
- Strutture dati utilizzate: Una lista L che contiene i puntatori alle n liste e n liste contenenti gli m interi.
- L'algoritmo prevede che:

```
PROBLEMA(L,x)
   occ <- 0
   conto le occorrenze nelle singole liste e le sommo in occ
   ritorno il valore della variabile occ
```

- Devo scorrere L e per ogni suo elemento conto le occorrenze di x nella lista corrispondente

```
PROBLEMA(L,x)
   occ <- 0
   WHILE (L <> NIL)
      occ <- occ + (calcolo le occorrenze di x nella lista
                    puntata dall'elemento corrente di L)
      L <- NEXT[L]

   ritorno il valore della variabile occ
```

Esercizio

- mi serve una funzione che conti le occorrenze di x in una lista di interi

```
CALCOLAOCCORRENZE(L,x)
    cont <- 0
    WHILE (L <> NIL)
       IF (INFO[L] = x)
       THEN
          cont <- cont + 1

       L <- NEXT[L]

    RETURN cont
```

- quindi lo pseudo-codice di PROBLEMA diventa:

```
PROBLEMA(L,x)
    occ <- 0
    WHILE (L <> NIL)
       occ <- occ + CALCOLAOCCORRENZE(PINFO[L],x)
       L <- NEXT[L]

    RETURN occ
```

Variazione sul tema

- Se le n liste fossero ordinate ($<$), avremmo modo di rendere più efficiente il nostro algoritmo? Se si, come?

Variazione sul tema

- Se le n liste fossero ordinate ($<$), avremmo modo di rendere più efficiente il nostro algoritmo? Se si, come?
- Sapendo che gli interi sono ordinati possiamo aggiungere una condizione alla ricerca in CALCOLAOCCORRENZE:

  se l'elemento da valutare è maggiore di quello che cerchiamo è inutile proseguire a cercare.

  In pseudo-codice:

```
CALCOLAOCCORRENZE(L,x)
   cont <- 0
   WHILE (L <> NIL AND INFO[L] <= x)
      IF (INFO[L] = x)
      THEN
         cont <- cont + 1

      L <- NEXT[L]

   RETURN cont
```

Codifica C

- Innanzitutto dobbiamo pensare alle strutture per le liste. Per le liste di interi sappiamo come fare:

```
typedef int TipoElemLista;

struct StructLista {
  TipoElemLista info;
  struct StructLista *next; };

typedef struct StructLista TipoNodoLista;
typedef TipoNodoLista *TipoLista;
```

- Per la lista L dobbiamo ridefinire le strutture, sfruttando sempre la definizione generica

```
typedef TipoLista pTipoElemLista;

struct pStructLista {
  TipoLista pinfo;
  struct pStructLista *pnext;
};

typedef struct pStructLista pTipoNodoLista;
typedef pTipoNodoLista *pTipoLista;
```

```
int calcolaOcc(TipoLista l, int val)
{
  int cont = 0;

  while ((l != NULL) && (l->info <= val))
    {
      if (l->info == val) cont++;
      l = l->next;
    }
  return cont;
}

int Problema(pTipoLista L, int val)
{
  int occorrenze = 0;

  while (L != NULL)
    {
      occorrenze = occorrenze + calcolaOcc(L->pinfo,val);
      L = L->pnext;
    }
  return occorrenze;
}
```

Funzione principale

- Per usare le funzioni appena definite, scriviamo lo pseudo-codice per la creazione di n liste di m interi e quindi traduciamolo in codice C.

```
funzione(N,M,X)
  pInitLista(L)

  FOR I = 1 TO N
    InitLista(APP)
    FOR J = 1 TO M
      ;;LEGGI DA TASTIERA UN INTERO (VAL)
      INSERISCITESTALISTA(APP,VAL)

    PINSERISCITESTALISTA(L,APP)

  RISULTATO <- Problema(L,X)
```

- INSERISCITESTALISTA e PINSERISCITESTALISTA eseguono la stessa operazione di inserimento, ma su tipi di dato differente

Funzione principale in C

```
funzione(int n, int m,int x)
{ pTipoLista plista1,papp;
  int i,j,res;

  pInitLista(&plista1);

  for(i=0;i<n;i++)
    { TipoLista app;
      InitLista(&app);

      for (j=0;j<m;j++)
        { int val;

          printf("Inserisci elem in lista %d ",(i+1));
          scanf("%d",&val);
          InserisciTestaLista(&app,val);
        }
      pInserisciTestaLista(&plista1,app);
    }
  printf("%d è presente %d volte\n",x,Problema(plista1,x));
}
```

Modificare lo pseudo-codice della generazione delle liste nel caso in cui n e m non siano noti a priori.