Esercizi Liste

Prof. Alessandro Campi

Esercizio

- Si consideri una lista dinamica di interi, i cui elementi sono del tipo definito come di seguito riportato:

```
typedef struct El {
  int dato;
  struct El *next;
} ELEMENTO;
```

- Si codifichi in C la funzione **somma** avente il seguente prototipo:

  int somma(ELEMENTO *Testa, int M)

- Tale funzione riceve come parametro la testa della lista e un intero M. Quindi, restituisce la somma dei soli valori della lista che sono multipli di M. Se la lista è vuota, la funzione restituisce il valore -1.

```
int somma(ELEMENTO *Testa, int M) {
    int sum=0;

    if (Testa==NULL)
        return(-1);
    else {
        while(Testa!=NULL) {
            if (Testa->dato%M==0)
                sum=sum+Testa->dato;

            Testa=Testa->next;
        }

        return sum;
    }
}
```

```
int somma(ELEMENTO *Testa, int M) {
    if (Testa==NULL)
        return(-1);
    if (Testa->next==NULL)
        if(Testa->dato%M==0)
            return Testa->dato;
        return 0;
    else
        if(Testa->dato%M==0)
            return Testa->dato+somma(Testa->next,M);
        else
            return somma(Testa->next,M);

}
```

Esercizio

- Trovare con una funzione ricorsiva l'elemento massimo in una lista

```
typedef struct Nodo {
        int valore;
        struct Nodo *prox;
} nodo;
typedef nodo *lista;

lista radice;
```

```
nodo * max(lista lis) {
   nodo * e;
   if(lis==NULL){//può accadere solo 1° chiamata
        printf("lista vuota");
        exit(1);//termina l'esecuzione del programma
   }
   if(lis->prox == NULL)
        return lis;
    e = max(lis->prox);
   if ( e->valore < lis->valore )
        return lis;
   else
        return e;
}
```

```
int max(lista lis) {
   int e;
   if(lis==NULL){//può accadere solo 1° chiamata
        printf("lista vuota");
        exit(1);//termina l'esecuzione del programma
   }
   if(lis->prox == NULL)
        return lis->valore;
    e = max(lis->prox);
   if ( e < lis->valore )
        return lis->valore;
   else
        return e;
}
```

Esercizio

- Supponendo date le seguenti definizioni:

```
typedef struct El {int s;
                   struct El *next;} ElementoLista;
typedef ElementoLista *ListaDiInteri;
```

- definire una funzione FirstEven che, data una ListaDiInteri restituisce la posizione (puntatore) del primo elemento pari nella lista (restituisce NULL se la lista non contiene elementi pari).

```
ListaDiInteri FirstEven(ListaDiInteri lis) {
    while (lis != NULL) {
        if (lis->s % 2 == 0)
                return lis;
                 /* il primo pari è in lis */
        else
                lis = lis -> next;
    }
    return NULL;
}
```

In versione ricorsiva

```
ListaDiInteri FirstEven(ListaDiInteri lis) {
    ListaDiInteri ris = NULL;
    if (lis != NULL){
        if ((lis -> s) % 2 == 0)
            ris = lis;
        else
            ris = FirstEven(lis -> next);
    }
    return ris;
}
```

Esercizio

- Supponendo date le seguenti definizioni:

```
typedef struct El {int s;
                   struct El *next;}Elemento;
typedef Elemento *ListaDiInteri;
```

- definire una funzione MinEven che, data una ListaDiInteri restituisce la posizione (puntatore) del **minimo** elemento pari nella lista (restituisce NULL se la lista non contiene elementi pari).

```
ListaDiInteri MinEven(ListaDiInteri lis){
    ListaDiInteri ris;
    ris = FirstEven(lis);
    if (ris != NULL) {
        lis = ris -> next;
        while (lis !=NULL) {
            if (((lis->s) % 2 = =0) && (lis->s < ris->s))
                ris = lis;
            lis = lis -> next;
        }
    }
    return ris;
}
```

- **Definiamo una funzione MinEl che restituisce il minimo tra due elementi**

```
ListaDiInteri MinEl(ListaDiInteri p, ListaDiInteri q) {
        if ((p->s) < (q->s))
                return p;
        else
                return q;
}
```

```
ListaDiInteri MinEven (ListaDiInteri lis) {
    ListaDiInteri p;
    if (lis == NULL)
        p = NULL;
    else
        if ((lis->s) % 2 !=0)
            p = MinEven(lis->next);
        else  {
            p = MinEven(lis->next);
            if (p != NULL)  p = MinEl(lis,p);
            else  p = lis;
        }
        return p;
}
```

Esercizio

- Sia data una struttura dati dinamica di tipo lista semplicemente concatenata. Ogni nodo della lista contiene un numero intero come valore.
- Si scriva la funzione che, dato un vettore di N numeri interi, restituisce la lista contenente gli N elementi del vettore; l'elemento di indice 0 va in testa alla lista, ecc.
- Si scriva la funzione che stampa a terminale i valori contenuti nella lista in ordine inverso rispetto a quello della lista stessa (leggendoli dalla lista, non dal vettore).

```
#include <stdio.h>
#define N 10
typedef struct Node {
  int val;
  struct Nodo *prox;
} Nodo;
typedef Nodo * lista;

lista crea (int n, int V[]);
void stampaInverso(lista lis)

int main() {
    int v[N] = {0,1,2,3,4,5,6,7,8,9};
    stampaInverso(crea(N,v));
    return 0;
}
```

```
lista crea (int n, int V[]) {
  lista testa;
  if (n == 0)
        return NULL;
  testa = (lista)malloc(sizeof(Nodo));//il cast può essere omesso
  testa->val = V[0];
  testa->prox = crea(n-1, &V[1]);
  return testa;
}

void stampaInverso(lista lis) {
  if (lis == NULL)
        return;
  stampaInverso (lis->prox);
  printf ("%d\n", lis->val);
}
```

Esercizio (tde 10-9-2008)

- Si progetti e codifichi una funzione C che riceve in ingresso una lista definita

  typedef struct Node { int numero;

  struct Node * next; } Nodo;

  typedef Nodo * Lista;

- La funzione deve verificare se l'andamento della lista è monotono crescente, cioè se ogni elemento è strettamente superiore al suo predecessore.

```
int monotona(Lista L) {
    Lista Cursore1=L, Cursore2;

    if(L!=NULL)
        Cursore2=L->next;

    while (Cursore2!=NULL) {
        if(Cursore2->numero < Cursore1->numero)
            return 0;
        Cursore1=Cursore2;
        Cursore2=Cursore2->next;
    }

    return 1;
}
```

```
int f(Lista lis) {
    if(lis==NULL )
            return 1;
    if(lis->next==NULL )
            return 1;
    if(lis-> numero < lis->next-> numero)
            return f(lis->next);
    else return 0;
}
```

Esercizio (tde 9-6-2009)

- Si progetti e codifichi una funzione C che riceve in ingresso una lista definita

  typedef struct Node { int numero;

  struct Node * next; } Nodo;

  typedef Nodo * Lista;

  che contiene solo valori positivi.
- Definiamo **picchi** della lista quei valori che sono preceduti e seguiti nella lista da valori più piccoli della loro metà. Il primo e l'ultimo elemento della lista non possono essere picchi.
- Ad esempio nella lista

  4 9 12 36 16 23 87 34 18 64 33

  36 e 87 sono picchi (perché 36/2=18, 18>12 e 18>16 e 87/2=43,5, 43,5>23 e 43,5>34)
- Si progetti e codifichi una funzione C che riceve in ingresso una lista definita come sopra e restituisce il numero di picchi che la lista contiene.

```
int picchi(Lista lis) {
    int n=0;
    while(lis!=NULL && lis->next!=NULL && lis->next->next!=NULL){

        if(((float)(lis->valore) < (float) (lis->next->valore)/2)  &&
                ((float)(lis->next->next->valore) < (float) (lis->next->valore)/2))
                n++;

        lis=lis->next;
    }
    return n;
}
```

```
int picchi(Lista lis) {
    if(! (lis!=NULL && lis->next!=NULL && lis->next->next!=NULL) )
        return 0;
    if(((float)(lis->valore) < (float) (lis->next->valore)/2)  &&
                    ((float)(lis->next->next->valore) < (float) (lis->next->valore)/2))
          return 1 + picchi(lis->next) ;
    else
          return picchi(lis->next) ;
}
```

Esercizio

- Si consideri una lista dinamica di interi (tutti diversi tra loro), i cui elementi sono del tipo definito come di seguito riportato:

```
typedef struct El {
    int dato;
    struct El *next;
} ELEMENTO;
```

- Si codifichi in C la funzione **mediana** avente il seguente prototipo:

int mediana(ELEMENTO *Testa)

```
int contaElementi(ELEMENTO *Testa) {
   int num=0;

   if(Testa==NULL)
         return 0;
    else {
         while(Testa!=NULL) {
              num++;
              Testa=Testa->next;
         }
         return num;
     }
}
```

```
int mediana(ELEMENTO *Testa) {
    ELEMENTO *Cursore1, *Cursore2;
    int num=0, piccoli, grandi;
    num=contaElementi(Testa);
    if (num==0)   return -1;
    else if (num%2==0) return -1;   // se contiene un numero di elementi pari non
                                    // ha senso perché non si ammettono duplicati
    else {
        Cursore1=Testa;
        while(Cursore1!=NULL) {
            Cursore2=Testa;  piccoli=0;  grandi=0;
            while(Cursore2!=null) {
                if(Cursore2->dato > Cursore1->dato)  piccoli++;
                else if(Cursore2->dato < Cursore1->dato) grandi++;
                Cursore2=Cursore2->next;
            }
            if(piccoli==grandi)  return Cursore1->dato;
            Cursore1=Cursore1->next;
        }
        return -1;//qui non si arriva
    }
}
```

Esercizio (tdeB 29-1-2007)

- Due liste di interi si dicono *equipotenti* se sono di uguale lunghezza e, confrontando i valori in posizioni corrispondenti, risulta che i valori della prima lista maggiori dei corrispondenti valori nella seconda sono esattamente in numero uguale ai valori della seconda lista maggiori dei corrispondenti valori nella prima.
- Data la lista definita come:

  typedef struct EL { int valore; struct EL * next; } Nodo;

  typedef Nodo * Lista;
- si descriva un algoritmo e si proponga una codifica per la funzione di prototipo

  int equipotenti (Lista L1, Lista L2);

  che restituisce 1 se le liste sono equipotenti, 0 altrimenti.

- Con una scansione parallela delle due liste, che si arresta appena una delle due è terminata, conto separatamente gli elementi maggiori e quelli minori. Restituisco 1 se al termine i conteggi si equivalgono e le liste terminano contemporaneamente, 0 altrimenti.

```
int equipotenti ( Lista L1, Lista L2 ){
  int maggiori = 0, minori = 0;
  while ( L1 != NULL && L2 != NULL ) {// continuo su entrambe
    if ( L1->valore > L2->valore )
      maggiori++;
    else if ( L1->valore < L2->valore )
      minori++;
    L1 = L1->next; // doppio passaggio ai successivi
    L2 = L2->next;
  }
  return (L1 == L2) && (maggiori == minori);
     // non basta che i due contatori abbiano lo stesso valore: le liste
     // devono anche essere entrambe finite
}
```

- Chi preferisce le soluzioni ricorsive apprezzerà la versione seguente. Conviene introdurre una funzioncina ausiliaria, per propagare il conteggio dello "sbilanciamento dei maggiori" come parametro. Si noti che uso una sola variabile da tenere bilanciata a zero con incrementi e decrementi, per non propagare due contatori.

```
int equipotenti ( Lista L1, Lista L2 ) {
  return controlla(L1, L2, 0);
}

int controlla ( Lista L1, Lista L2, int bilancio ){
  if ( L1 == NULL || L2 == NULL )
    return ( L1 == L2 && bilancio == 0 );

  if ( L1->valore > L2->valore )
          bilancio++;
  else if ( L1->valore < L2->valore )
          bilancio--;

  return controlla( L1->next, > L2->next, bilancio );
}
```

Esercizio

- Supponendo date le seguenti definizioni:

```
typedef struct El {Tipo s;
                   struct El *next;} Elemento;
typedef Elemento *Lista;
```

- definire una funzione `foo` che, data una `Lista` `l` ed un intero `el` inserisce `el` dopo il terzo elemento di `l`. Se quest'ultima non contiene almeno tre elementi, viene lasciata inalterata.

Soluzione

- Definiamo dapprima una funzione `third` che restituisce il puntatore al terzo elemento della lista, o `NULL` se la lista ha meno di tre elementi.

```
Lista third(Lista lis) {
    int count = 1;
    while ((lis != NULL) && (count < 3))  {
        count ++;
        lis = lis -> next;
    }

    if (count = = 3)
        return lis;
    else
        return NULL;
}
```

- Definiamo ora la funzione `foo`.

```
void foo (Lista lis, Tipo el) {
    Lista tmp, new;
    tmp = third(lis);
    if (tmp != NULL){
        new = malloc(sizeof(Elemento));
        new -> next = tmp -> next;
        new -> s = el;
        tmp -> next = new;
    }
}
```

Esercizio

- Supponendo date le seguenti definizioni:

```
typedef struct El {int s;
                   struct El *next;}
  Elemento;
typedef Elemento *Lista;
```

- definire una funzione `foo` che, data una `Lista` `l` ed un intero `el` inserisce `el` dopo l'ultimo elemento di `l` maggiore di `el`. Se `l` non contiene alcun elemento maggiore di `el`, la procedura lascia la lista inalterata.

Soluzione

- Definiamo dapprima una funzione che restituisce il puntatore all'ultimo elemento di una lista maggiore di un elemento dato (`NULL` se non ci sono elementi maggiori dell'elemento dato).

```
Lista LastBigger(Lista lis, int el) {
    Lista tmp=NULL;
    while (lis != NULL) {
        if ((lis -> s) > el)
            tmp = lis;
        lis = lis -> next;
    }

    return tmp;
}
```

- Definiamo ora la funzione foo (va bene void e non serve restituisca lis perché la testa non può mai cambiare)

```
void foo(Lista lis, int el) {
    Lista temp, nuovo;
    temp = LastBigger(lis, el);
    if (temp != NULL) {
        nuovo = malloc(sizeof(Elemento));
        nuovo->s = el;
        nuovo->next = temp->next;
        temp->next =nuovo;
    }
}
```

Esercizio - variante

- Semplice variante del problema precedente: se ℓ non contiene alcun elemento maggiore di el, quest'ultimo viene inserito in testa alla lista

```
Lista foo(Lista lis, int el) {
    Lista temp, nuovo;
    temp = LastBigger(lis, el);
    nuovo = malloc(sizeof(Elemento));
    nuovo->s = el;
    if (temp != NULL) {
        nuovo -> next = temp -> next;
        temp -> next = nuovo;
    } else {
        nuovo -> next = lis;
        lis = nuovo;
    }
    return lis;
}
```

Esercizio

- Supponendo date le seguenti definizioni:

```
typedef struct El {int s;
                   struct El *next;} Elemento;
typedef Elemento *Lista;
```

- definire una funzione `Eliminaiel` che, data una `Lista` ℓ elimini i primi i elementi. Ad esempio, data la lista rappresentata dalla seguente figura

```
--> 2 --> 7 --> 7 --> 9 --> 9
```

e i uguale a 2 la procedura modifica la lista come segue:

```
--> 7 --> 9 --> 9
```

```
void Eliminaiel(Lista *lis, int i) {
   Lista aux, temp = *lis;
   int j=0;
   while ((temp!=NULL)&& (j<i)) {
        aux = temp;
        temp = temp->next;
        free(aux);
        j++;
   }
   *lis = temp;
}
```

```
void Eliminaiel(Lista *lis, int i) {
    Lista aux;
    if (i>0 && (*lis!=NULL)) {
        aux = *lis;
        *lis = *lis -> next;
        free(aux);
        Eliminaiel(lis,i-1);
    }
}
```

Esercizio

- Supponendo date le seguenti definizioni:

```
struct El
typedef struct El {int s;
                   struct El *next;}
  Elemento;
typedef Elemento *Lista;
```

- Definire una funzione `InserisciOrd` che, data una `Lista` ℓ **ordinata**, ed un elemento el, inserisce quest'ultimo al posto giusto.

- Definiamo prima la funzione `LastSmallerOrEqual` come segue. Sfruttiamo l'ipotesi che la lista è ordinata, non vuota e che il primo elemento è minore o uguale a `el`.

```
Lista LastSmallerOrEqual(Lista lis, int el) {
    ListaDiInteri tmp=lis;
    int trovato = 0;
    while (!trovato && (tmp-> next != NULL)) {
        if ((tmp -> next -> s) > el)
                trovato = 1;
        else
                tmp = tmp -> next;
    }
    return tmp;
}
```

```
Lista InserisciOrd (Lista lis, int el) {
    Lista nuovo, temp = lis;
    nuovo = malloc(sizeof(Elemento));
    nuovo->s = el;
    nuovo->next = NULL;
    if (temp = = NULL)
        return nuovo;
    else
        if (temp->s > el) {/*l'elemento va inserito in testa alla lista*/
            nuovo->next = temp;
            return nuovo;
         } else {
            temp = LastSmallerOrEqual(temp, el);
            nuovo->next = temp->next;
            temp->next = nuovo;
            return lis;
        }
}
```

Esercizio

- Definire una funzione `foo` che, data una `ListaDiInteri` ℓ ed un intero el, inserisce quest'ultimo tra i primi due elementi consecutivi che siano, rispettivamente, strettamente minore e strettamente maggiore di el. Se la lista ℓ non contiene due elementi consecutivi siffatti, la procedura inserisce el in testa.

Definiamo una funzione `FindPosition`

```
Lista FindPosition (Lista lis, int el) {
    if (lis==NULL)
        return NULL;
    else
        if (lis->next == NULL)
            return NULL;
        else
            if ((lis->s < el) && (lis->next->s > el))
                return lis;
            else
                return FindPosition(lis->next, el);
}
```

```
Lista foo(Lista lis, int el) {
    Lista temp, nuovo, pos;
    nuovo = malloc(sizeof(Elemento));
    nuovo->s = el;
    temp=FindPosition(lis, el);
    if (temp==NULL) {
        nuovo->next=lis;
        return nuovo;
    } else {
        nuovo->next = temp->next;
        temp->next=nuovo;
        return lis;
    }
}
```