Esame di Laboratorio del 28/07/2021

Note importanti:

- È considerato errore qualsiasi output non richiesto dagli esercizi.
- È consentito utilizzare funzioni ausiliarie per risolvere gli esercizi (in alcuni casi è caldamente consigliato o indispensabile!).
- Quando caricate il codice sul sistema assicuratevi che siano presenti tutte le direttive di include necessarie, comprese quelle per l'utilizzo delle primitive. Non dovete caricare l'implementazione delle primitive.
- È importante sviluppare il codice in Visual Studio (o altri IDE) prima del caricamento sul sistema, così da poter effettuare il debug delle funzioni realizzate!
- Su OLJ non sarà possibile eseguire più di una compilazione/test ogni 3 minuti.

Esercizio 1

Scrivere un programma a linea di comando con la seguente sintassi:

```
radicenumerica <n>
```

Il programma `radicenumerica` prende in input un numero intero positivo `n` e deve stamparne a video (`stdout`) la radice numerica, ovvero la somma delle cifre reiterata fino ad ottenere un valore monocifra, quindi compreso tra 0 e 9. Il programma deve calcolare la radice numerica tramite una funzione **ricorsiva**.

Ad esempio, dato `n=193`, il programma stamperà `4`:

```
193 -> 1 + 9 + 3 = 13
13 -> 1 + 3 = 4
```

Dato invece `n=193465`, il programma stamperà `1`:

```
193465 -> 1 + 9 + 3 + 4 + 6 + 5 = 28
28 -> 2 + 8 = 10
10 -> 1 + 0 = 1
```

Se il numero dei parametri passati al programma non è corretto o se `n < 0`, questo termina con codice `1`, senza stampare nulla, altrimenti termina con codice di uscita `0` dopo aver completato la stampa.

Non saranno considerate valide soluzione che non fanno uso della ricorsione per calcolare la radice numerica di `n`.

Esercizio 2

Avete un insieme di biscotti di diverse dimensioni, che volete distribuire ad un gruppo di bambini in modo da accontentarne il più possibile.

Nel file `biscotti.c` definire la funzione:

```
extern int AssegnaBiscotti(const int* bam, size_t bam_size,
                           const int* bis, size_t bis_size);
```

I parametri di input della funzione sono:

- Un vettore di bambini, `bam`, di dimensione `bam_size`, contenente il valore di "golosità" (in grammi) di ciascun bambino;
- Un vettore di biscotti, `bis`, di dimensione `bis_size`, contenete la massa in grammi di ciascun biscotto.

La funzione, utilizzando un algoritmo di backtracking, deve determinare il modo più efficiente di distribuire i biscotti, con l'obiettivo di massimizzare il numero di bambini soddisfatti. Ogni bambino può ricevere 0, 1 o anche più biscotti, ed è considerato soddisfatto se riceve biscotti per una massa totale maggiore o uguale del suo valore di golosità. I biscotti non sono divisibili.

La funzione deve ritornare il numero massimo di bambini che possono essere accontentati, che sarà compreso nell'intervallo `[0, bam_size]`.

Se uno dei vettori di input è `NULL` o ha dimensione 0, la funzione ritorna 0.

Di seguito vengono riportati alcuni esempi:

Esempio 1:

```
input:
bam = [10, 10, 10]
bis = [10, 12, 14]

output:
ret = 3
```

Esempio 2:

```
input:
bam = [10, 20, 30]
bis = [10, 6, 7, 8]

output:
ret = 2
```

In questo secondo caso non c'è modo di accontentare tutti e tre i bambini, ma si possono accontentare i primi due, dando il primo biscotto al primo bambino e tutti gli altri al secondo bambino.

Esempio 3:

```
input:
bam = [10, 20, 30]
bis = [10, 50]

output:
ret = 2
```

In questo caso abbiamo solo due biscotti in totale, quindi è impossibile accontentare 3 bambini, ma è possibile arrivare a 2 poiché le masse dei biscotti lo consentono.

Esercizio 3

Nel file `reverse.c` definire la funzione corrispondente alla seguente dichiarazione:

```
extern Item *Reverse(Item *i);
```

La funzione prende in input una lista di interi, `i`, e la **modifica** ribaltandone il contenuto.

Ad esempio, data `i -> 1 -> 3 -> 4 -> 7 -> 8 -> NULL` la funzione ritorna la lista `i' -> 8 -> 7 -> 4 -> 3 -> 1 -> NULL`. Data `i -> 1 -> 2 -> NULL` la funzione deve ritornare `i' -> 2 -> 1 -> NULL`.

La funzione ritorna il puntatore alla testa della lista modificata. La funzione deve modificare il campo `next` degli `Item` esistenti, senza allocare nuova memoria. Non saranno considerate valide soluzioni che producono una *nuova* lista con gli elementi ribaltati.

Per la risoluzione di questo esercizio avete a disposizione le seguenti definizioni:

```
typedef int ElemType;

struct Item {
    ElemType value;
    struct Item *next;
};
typedef struct Item Item;
```

e le seguenti funzioni primitive e non:

```
ElemType ElemCopy(const ElemType *e);
void ElemSwap (ElemType *e1, ElemType *e2)
void ElemDelete(ElemType *e);
void ElemWrite(const ElemType *e, FILE *f);
void ElemWriteStdout(const ElemType *e);

Item *ListCreateEmpty(void);
Item *ListInsertHead(const ElemType *e, Item* i);
bool ListIsEmpty(const Item *i);
```

```
const ElemType *ListGetHeadValue(const Item *i);
Item *ListGetTail(const Item *i);
Item *ListInsertBack(Item *i, const ElemType *e);
void ListDelete(Item *item);
void ListWrite(const Item *i, FILE *f);
void ListWriteStdout(const Item *i);
```

Trovate le definizioni, le dichiarazioni e le rispettive implementazioni nei file `elemtype.h`, `elemtype.c`, `list.h` e `list.c` scaricabili da OLJ, così come la loro documentazione.

Esercizio 4

Dato un array di `ElemType` senza duplicati, `v`, un *albero binario minimo* può essere costruito in maniera ricorsiva mediante il seguente algoritmo:

1. Si crea la radice dell'albero il cui valore, `m`, è il minimo in `v`;
2. Si costruisce ricorsivamente il sottoalbero sinistro utilizzando il sottovettore `vl` contenente gli elementi di `v` a sinistra di `m`;
3. Si costruisce ricorsivamente il sottoalbero destro utilizzando il sottovettore `vr` contenente gli elementi di `v` a destra di `m`;

Nel file `minbin.c` definire la funzione corrispondente alla seguente dichiarazione:

```
extern Node *CreateMinBinTree(const ElemType *v, size_t v_size);
```

La funzione prende in input un vettore di `ElemType`, costruisce e ritorna l'*albero binario minimo* secondo l'algoritmo sopra descritto.

Esempio 1:

```
input:
v = [3,2,1,6,8,5]

output:

     1
    / \
   /   \
  2     5
 /     /
3     6
       \
        8
```

Esempio 2:

```
input:
v = [3,2,1]
```

```
Output:
    1
   /
  2
 /
3
```

Per la risoluzione di questo esercizio avete a disposizione le seguenti definizioni:

```c
typedef int ElemType;

struct Node {
    ElemType value;
    struct Node *left;
    struct Node *right;
};
typedef struct Node Node;
```

e le seguenti funzioni primitive e non:

```c
int ElemCompare(const ElemType *e1, const ElemType *e2);
ElemType ElemCopy(const ElemType *e);
void ElemDelete(ElemType *e);
int ElemRead(FILE *f, ElemType *e);
int ElemReadStdin(ElemType *e);
void ElemWrite(const ElemType *e, FILE *f);
void ElemWriteStdout(const ElemType *e);

Node *TreeCreateEmpty(void);
Node *TreeCreateRoot(const ElemType *e, Node *l, Node *r);
bool TreeIsEmpty(const Node *n);
const ElemType *TreeGetRootValue(const Node *n);
Node *TreeLeft(const Node *n);
Node *TreeRight(const Node *n);
bool TreeIsLeaf(const Node *n);
void TreeDelete(Node *n);

void TreeWritePreOrder(const Node *n, FILE *f);
void TreeWriteStdoutPreOrder(const Node *n);
void TreeWriteInOrder(const Node *n, FILE *f);
void TreeWriteStdoutInOrder(const Node *n);
void TreeWritePostOrder(const Node *n, FILE *f);
void TreeWriteStdoutPostOrder(const Node *n);
```

Trovate le definizioni, le dichiarazioni e le rispettive implementazioni nei file `elemtype.h`, `elemtype.c`, `tree.h` e `tree.c` scaricabili da OLJ, così come la loro documentazione.

Esercizio 5

Nel file `pop.c` definire la procedura corrispondente alla seguente dichiarazione:

```
extern bool Pop(Heap *h, ElemType *e);
```

Dato un min-heap di elementi di *qualunque tipo*, h, la funzione rimuove dalla heap il valore minimo, e lo copia nella variabile puntata da e. La funzione deve garantire che le proprietà min-heap siano rispettate, e deve avere complessità computazionale non superiore a $O(\log_2 n)$.

Se h contiene almeno un elemento (prima della rimozione del minimo), la funzione ritorna `true`. Altrimenti, la funzione ritorna `false` e lascia invariato il valore puntato da e.

Per la risoluzione di questo esercizio avete a disposizione le seguenti definizioni. Si noti che le primitive fornite sono relative al tipo `int`, ma l'implementazione della funzione `Pop()` deve rimanere valida al variare della definizione dell'`ElemType`:

```
typedef int ElemType;

struct Heap {
    ElemType *data;
    size_t size;
};
typedef struct Heap Heap;
```

e le seguenti funzioni primitive e non:

```
int ElemCompare(const ElemType *e1, const ElemType *e2);
ElemType ElemCopy(const ElemType *e);
void ElemDelete(ElemType *e);
int ElemRead(FILE *f, ElemType *e);
int ElemReadStdin(ElemType *e);
void ElemWrite(const ElemType *e, FILE *f);
void ElemWriteStdout(const ElemType *e);

int HeapLeft(int i);
int HeapRight(int i);
int HeapParent(int i);
Heap *HeapCreateEmpty(void);
bool HeapIsEmpty(const Heap *h);
void HeapDelete(Heap *h);
void HeapWrite(const Heap *h, FILE *f);
void HeapWriteStdout(const Heap *i);
ElemType *HeapGetNodeValue(const Heap *h, int i);
void HeapMinInsertNode(Heap *h, const ElemType *e);
void HeapMinMoveUp(Heap *h, int i);
void HeapMinMoveDown(Heap *h, int i);
```

Trovate le definizioni, le dichiarazioni e le rispettive implementazioni nei file `elemtype.h`, `elemtype.c`, `minheap.h` e `minheap.c` scaricabili da OLJ, così come la loro documentazione.