Esame di Laboratorio di Fondamenti di Informatica II e Lab. del 08/01/2020

Note importanti:

- È considerato errore qualsiasi output non richiesto dagli esercizi.
- È consentito utilizzare funzioni ausiliarie per risolvere gli esercizi.
- 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, prima del caricamento sul sistema, per poter effettuare il debug delle funzioni realizzate!

Esercizio 1 (4 punti)

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

```
extern int Quoziente(size_t x, size_t y);
```

La funzione prende in input due interi senza segno `x` e `y` e deve calcolare ricorsivamente il quoziente della divisione intera tra `x` e `y`.

Si noti che, fino a quando $x \geq y$:

$$\frac{x}{y} = \frac{x-y+y}{y} = 1 + \frac{x-y}{y}$$

Ad esempio, il quoziente di `x = 8` diviso `y = 3` può essere calcolato ricorsivamente come:

$$\frac{8}{3} = 1 + \frac{8-3}{3} = 1 + \frac{5}{3} = 1 + 1 + \frac{5-3}{3} = 1 + 1 + \frac{2}{3} = 1 + 1 + 0 = 2$$

Il quoziente di `2` diviso `3` corrisponde a `0` in quanto `2 < 3`. Quindi in questo esempio la funzione deve ritornare `2`.

Se `y` vale zero la divisione è impossibile e la funzione ricorsiva deve ritornare `-1`.

Non saranno considerate valide soluzioni che non fanno uso della ricorsione per il calcolo del quoziente.

Esercizio 2 (8 punti)

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

```
extern void GolaCresta(size_t n);
```

La procedura prende in input un numero intero senza segno (`n`) maggiore o uguale a `3` e utilizzando un algoritmo di backtracking deve stampare su `stdout` tutte le sequenze di lunghezza n con elementi in `{0, 1, 2}` in cui per qualsiasi terna x_i, x_{i+1}, x_{i+2}, dove x_i corrisponde al numero in prima posizione, x_{i+1} corrisponde al numero in seconda posizione e x_{i+2} al numero in terza posizione, siano soddisfatte le seguente condizioni $x_i < x_{i+1}$ e $x_{i+1} > x_{i+2}$, oppure $x_i > x_{i+1}$ e $x_{i+1} < x_{i+2}$.

Ad esempio se `n = 3` allora il programma deve stampare (non necessariamente in quest'ordine):

```
(0, 1, 0), (0, 2, 0), (0, 2, 1), (1, 0, 1), (1, 0, 2) e (1, 2, 0), (1, 2, 1), (2, 0, 1),
(2, 0, 2), (2, 1, 2),
```

Se invece `n = 4` allora il programma deve stampare (non necessariamente in quest'ordine):

```
(0, 1, 0, 1), (0, 1, 0, 2), (0, 2, 0, 1), (0, 2, 0, 2), (0, 2, 1, 2), (1, 0, 1, 0),
(1, 0, 2, 0), (1, 0, 2, 1), (1, 2, 0, 1), (1, 2, 0, 2), (1, 2, 1, 2), (2, 0, 1, 0),
(2, 0, 2, 0), (2, 0, 2, 1), (2, 1, 2, 0), (2, 1, 2, 1),
```

Il formato dell'output deve corrispondere a quello degli esempi, ovvero ogni sequenza deve iniziare con `<aperta tonda>` e terminare con `<chiusa tonda>`, gli elementi di una sequenza e le sequenze stesse devono essere separate dai caratteri `<virgola>` e `<spazio>`.

Esercizio 3 (7 punti)

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

```
extern list OrderedMerge(list la, list lb);
```

La funzione prende in input due liste di interi `la` e `lb` ordinate in senso crescente. La funzione deve fondere le due liste preservando l'ordinamento degli elementi: la fusione deve essere fatta aggiornando i puntatori degli `item` esistenti, non è consentito creare nuovi `item` (allocare memoria per nuovi `item`) o ordinare la lista dopo averla creata.

La funzione deve quindi ritornare la lista risultante, ovvero l'indirizzo della sua testa. Se una delle due liste di input è `NULL`, la funzione deve ritornare l'altra. Se entrambe le liste `la` e `lb` sono `NULL` la funzione deve ritornare `NULL`.

Se `la` e `lb` fossero ad esempio le due liste seguenti:

la funzione deve allora produrre in output la lista:

Si noti che gli indirizzi dei nodi non sono cambiati!

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

```
typedef int element;

typedef struct list_element {
  element value;
  struct list_element *next;
} item;

typedef item* list;
```

e le seguenti primitive:

```
list EmptyList(void);
list Cons(const element *e, list l);
bool IsEmpty(list l);
element Head(list l);
list Tail(list l);
element Copy(const element *e);
void FreeList(list l);
```

trovate le definizioni, le dichiarazioni e le rispettive implementazioni nei file list_int.h e list_int.c. La documentazione delle primitive sopraelencate è disponibile al link:

http://imagelab.ing.unimore.it/olj2/esami/materiale/20200801_esame/list_int/list__int_8h.html

Esercizio 4 (7 punti)

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

```
extern void ExpressionTree(tree t);
```

La procedura prende in input un albero binario `t` che rappresenta un'espressione aritmetica binaria. La procedura deve quindi scorrere l'albero, leggerne il contenuto e scrivere l'espressione corrispondente su `stdout` aggiungendo opportunamente parentesi tonde.

L'albero `t` è così definito:

```
typedef struct element {
    char op;
    int  vl;

    bool is_op;
}element;

typedef struct tree_element {
    element value;
    struct tree_element *left,*right;
} node;

typedef node* tree;
```

ogni nodo dell'albero può essere un `char` che rappresenta l'operatore o un `int` che rappresenta l'operando. Il campo `is_op` di tipo booleano serve a distinguere i due casi. Quando `is_op` è `true` l'`element` corrente è un operatore, ovvero solo il campo `op` è valido, mentre quando `is_op` è `false` significa che l'`element` corrente è un operando, quindi solo il campo `vl` è valido.

La struttura dell'albero `t` è così definita:

- ogni foglia dell'albero è un numero (`is_op = false`) che rappresenta un operando;
- ogni nodo non foglia dell'albero è un operatore (`is_op = true`);
- le operazioni a precedenza maggiore, ovvero quelle che devono essere esguite per prime, sono quelle più vicine alle foglie.

La costruzione dell'espressione sarà quindi una semplice visita in `in-ordine` dell'albero con qualche piccola accortezza in più: se il sottoalbero sinistro (o destro) è un operando posso proseguire l'esplorazione normalmente. Se il sottoalbero sinistro (o destro) è un operatore prima di procedere l'attraversamento dell'albero devo aggiungere una `(` che dovrò chiudere opportunamente al termine dell'esplorazione del sottoalbero stesso.

Dato ad esempio l'albero:

la procedura deve produrre l'output

```
2 * (3 - (5 + 2))
```

Se l'albero fosse invece:

la procedura dovrebbe produrre l'output

```
(7 - 12) * (3 - (5 + 2))
```

Se l'albero di `t` è vuoto la procedura deve stampare `no expression` su `stdout`.

Per la risoluzione di questo esercizio avete a disposizione, oltre che le definizioni sopra riportate, le seguenti primitive:

```
tree EmptyTree(void);
tree ConsTree(const element *e, tree l, tree r);
bool IsEmpty(tree t);
element* GetRoot(tree t);
tree Left(tree t);
tree Right(tree t);
bool IsLeaf(tree t);
```

trovate le definizioni, le dichiarazioni e le rispettive implementazioni nei file tree_expr.h e tree_expr.c. La documentazione delle primitive sopraelencate è disponibile al link:

http://imagelab.ing.unimore.it/olj2/esami/materiale/20200801_esame/tree_expr/tree__expr_8h.html

Esercizio 5 (7 punti)

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

```
extern tree ReadTree(FILE *f);
```

La funzione prende in input un file aperto in modalità lettura tradotta (testo). La funzione deve leggere il file in cui sono memorizzati i nodi di un albero di caratteri, costruire e ritornare l'albero letto. All'interno del file i nodi (caratteri) sono memorizzati in sequenza dall'alto verso il basso (dalla radice alle foglie) da sinistra verso destra senza spazi. L'albero memorizzato nel file è sempre completo.

Dato ad esempio il file:

```
abcdefghijklmno
```

la funzione deve costruire e ritornare l'albero

Si può assumere che il file sia corretamente formato. Se non è possibile leggere il file o se questo è vuoto, la funzione deve ritornare un albero vuoto.

Suggerimento: Si consiglia di utilizzare la funzione `fseek()` per spostarsi all'interno del file e semplificarne la lettura. Si noti che, per come è strutturato, il file contiene una rappresentazione dell'albero che assomiglia molto alla rappresentazione mediante vettore. Ovvero, numerando le chiavi da `1` a `n`

```
1 2 3 4 5 6 7 8 9 ...
a b c d e f g h i j k l m n o
```

la radice si trova in posizione `1`, il suo figlio sinistro in posizione `2` e quello destro in posizione `3`. In generale dato un nodo in posizione `i` il suo figlio sinistro si trova in posizione `2*i` mentre quello destro in posizione `2*i + 1`.

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

```
```

```
typedef char element;
typedef struct tree_element {
element value;
    struct tree_element *left, *right;
} node;
typedef node* tree;
```

e le seguenti primitive:

```
tree EmptyTree();
tree ConsTree(const element *e, tree l, tree r);
bool IsEmpty(tree t);
element *GetRoot(tree t);
tree Left(tree t);
tree Right(tree t);
bool IsLeaf(tree t);
tree InsertBinOrd(const element *e, tree t);
```

trovate le definizioni, le dichiarazioni e le rispettive implementazioni nei file tree_char.h e tree_char.c. La documentazione delle primitive sopraelencate è disponibile al link:

http://imagelab.ing.unimore.it/olj2/esami/materiale/20200801_esame/tree_char/tree__char_8h.html