Esame di Laboratorio del 13/09/2022

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, per un massimo di 3 compilazioni per esercizio. Il numero di sottomissioni, invece, non è sottoposto a vincoli temporali o quantitativi.

Esercizio 1

Il paradigma "divide et impera" o "divide and conquer" in inglese è stato utilizzato in un'ampia varietà di problemi. In sostanza, l'obiettivo è quello di suddividere un problema in due sottoproblemi di uguale dimensione, risolvili in modo ricorsivo, e quindi usare le soluzioni a questi sottoproblemi più semplici per risolvere il problema originale. L'algoritmo di ordinamento *mergesort* è un esempio di algoritmo che utilizza questo paradigma. Nello specifico, il numero di confronti utilizzati dal *mergesort* è dato dalla seguente funzione ricorsiva

$$
\begin{aligned}
C_0 &= 0 \\
C_1 &= 1 \\
C_N &= C_{\lfloor N/2 \rfloor} + C_{\lceil N/2 \rceil} + N
\end{aligned}
$$

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

```
dividenconquer <n>
```

Il programma prende in input un numero intero positivo, `n`, che rappresenta la lunghezza di un vettore da ordinare. Utilizzando la funzione ricorsiva sopra riportata, il programma deve calcolare e stampare a video quanti sono i confronti effettuati dall'algoritmo *mergesort* per ordinare una sequenza di lunghezza `n`.

Se n < 0 o se il numero di parametri passati al programma è sbagliato, questo termina con codice `1` senza stampare nulla, in tutti gli altri casi termina con codice `0` dopo aver stampato su `stdout`.

Seguono alcuni esempi:

```
cmd: dividenconquer 0
stdout: 0

cmd: dividenconquer 1
stdout: 1

cmd: dividenconquer 2
stdout: 4

cmd: dividenconquer 3
stdout: 8

cmd: dividenconquer 4
```

```
stdout: 12

cmd: dividenconquer 5
stdout: 17
```

Non saranno considerate valide soluzioni che non fanno uso della ricorsione per determinare C_N.

Esercizio 2

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`, e implementa un algoritmo di backtracking che stampa 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 seguenti 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` la procedura deve stampare (non necessariamente in quest'ordine):

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

Se invece `n = 4` la procedura 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,
(1, 0, 2, 0), (1, 0, 2, 1), (1, 2, 0, 1), (1, 2, 0, 2), (1, 2, 1, 2), (2, 0, 1,
(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 il carattere `'('` e terminare con `')'`. Gli elementi di una sequenza e le sequenze stesse devono essere separate dai caratteri `','` e `' '`. Se `n < 3` la funzione non stampa nulla.

Esercizio 3

Nei file `reverse.h` e `reverse.c` si implementi la definizione della seguente funzione:

```
extern Item* Reverse(Item *list, int right);
```

La funzione prende in input una lista di `n` ElemType di tipo `int`, `list`, e un numero intero `right`. La funzione inverte i nodi della lista fino a quello di indice `right` escluso e restituisce la lista risultante. Si consideri il primo `item` in posizione 0.

Se `right >= len(list)`, ovvero se `right` è maggiore o uguale all'ultimo indice nella lista, la funzione deve invertire tutti gli elementi della lista.

Se `right <= 1` la funzione ritorna la lista di input, senza modificarla.

La funzione deve modificare il campo `next` degli `Item` della lista di input, non saranno considerate valide soluzione che creano una nuova lista con gli `item` "invertiti" o che modificano il campo `value`.

Seguono alcuni esempi:

```
INPUT:
list = [0, 1, 2, 3, 4, 5, 6, 7, 8], right = 3
OUTPUT
list = [2, 1, 0, 3, 4, 5, 6, 7, 8]


INPUT:
list = [0, 1, 2, 3, 4, 5, 6, 7, 8], right = 7
OUTPUT
list = [6, 5, 4, 3, 2, 1, 0, 7, 8]


INPUT:
list = [0, 1, 2, 3, 4, 5, 6, 7, 8], right = 12
OUTPUT
list = [8, 7, 6, 5, 4, 3, 2, 1, 0]


INPUT:
list = [0, 1, 2, 3, 4, 5, 6, 7, 8], right = 0
OUTPUT
list = [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

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

```c
typedef int ElemType;

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

e le seguenti funzioni primitive e non:

```c
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

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

```
extern void PopulatingNext(Node *t);
```

La funzione `PopulatingNext()` prende in input un albero binario `t`. I nodi dell'albero sono così definiti:

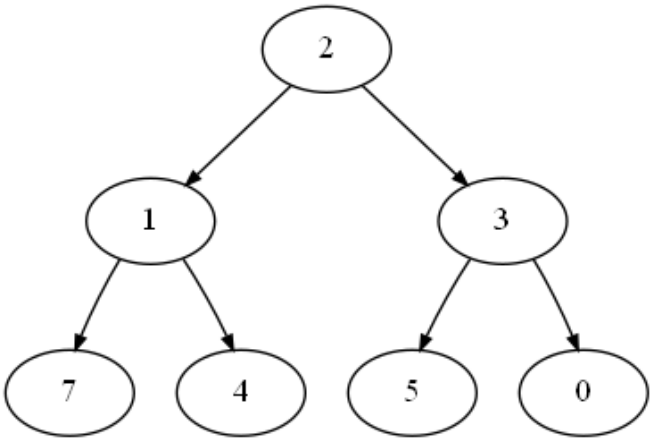
```
typedef int ElemType;

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

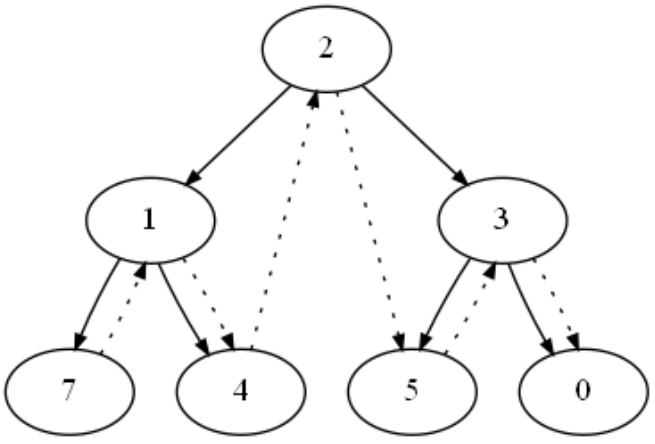
Nello specifico, oltre all'indirizzo dei figli sinistro e destro, ogni nodi memorizza l'indirizzo del nodo successivo, `next`, nell'esplorazione dell'albero in-ordine. Inizialmente, tutti i `next` sono `NULL`.

La funzione deve aggiornare opportunamente il campo `next` di tutti i nodi dell'albero.

Dato ad esempio l'albero:



La funzione deve impostare i puntatori a `next` come rappresentato dalle frecce tratteggiate:



Il `next` di `0`, l'ultimo nodo nella visita in-order, è l'unico a rimanere `NULL`. Seguendo i puntatori a `next` a partire dal primo nodo che si visita nell'esplorazione in-ordine dell'albero (7 in questo esempio), si ottiene la sequenza `7 -> 1 -> 4 -> 2 -> 5 -> 3 -> 0 -> NULL`.

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

```
typedef int ElemType;
```

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

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);

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 `binarysearch.c` definire la procedura corrispondente alla seguente dichiarazione:

```
extern int BinarySearch(const int *v, size_t v_size, int value);
```

La funzione prende in input un vettore di `int` ordinato in senso crescente, `v`, e la sua dimensione, `v_size`, e deve applicare l'algoritmo di ricerca binaria per trovare `value` dentro `v`. Il vettore non contiene duplicati. Se `value` non è presente la funzione ritorna `-1`, altrimenti ritorna la posizione nel vettore.