Esame di Laboratorio del 14/06/2021

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 (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:

```
printbin <n>
```

Il programma prende in input un numero intero `n` e deve stampare a video (`stdout`) la rappresentazione binaria del numero, ottenuta mediante funzione **ricorsiva**.

Ad esempio, dato `n = 4` il programma deve produrre il seguente output:

```
100
```

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 la stampa della rappresentazione binaria di `n`.

Esercizio 2

Creare i file `domino.h` e `domino.c` che consentano di definire le seguenti strutture:

```c
typedef struct {
    uint8_t val1;
    uint8_t val2;
} Tessera;

typedef struct {
    uint32_t indice;
    bool flipped;
} Placing;
```

e la procedura:

```
extern Placing *Domino(const Tessera *t, size_t t_size, size_t *ret_size);
```

Una `Tessera` del domino è rappresentata da due numeri `val1` e `val2` (compresi tra 0 e 6, estremi inclusi) e dalla sua posizione (*indice*) all'interno del vettore di `Tessera`, `t`. La struttura `Placing` viene utilizzata per rappresentare il posizionamento delle tessere, nello specifico, `indice` è l'indice della tessera nel vettore, mentre `flipped` indica il verso con cui la tessera è stata posizionata. Se `flipped == false`, la tessera è posizionata in modo tale che `val1` sia rivolto a "sinistra" e `val2` a "destra". Viceversa, se `flipped == true`, la tessere è posizionata in modo tale che `val2` sia rivolto a "sinistra" e `val1` a "destra".

Data, ad esempio, la tessera `Tessera t_example = { .val1 = 5, .val2 = 3 }`, supponendo che questa sia la tessera di indice `i = 2` questa può essere posizionata in due modi:

`Placing p_example1 = { .indice = 2, .flipped = false }`, ovvero:

```
+---+---+
| 5 | 3 |
+---+---+
```

Oppure `Placing p_example2 = { .indice = 2, .flipped = true }`, ovvero:

```
+---+---+
| 3 | 5 |
+---+---+
```

Dati in input un vettore di tessere, `t`, e la sua dimensione, `t_size`, la procedura implementa un algoritmo di backtracking che individua il "serpente" più lungo che è possibile costruire con le tessere a disposizione, rispettando le regole del gioco: due tessere possono essere posizionate una di seguito all'altra solo se il numero corrispondente ai lati adiacenti è uguale. Supponiamo che il "serpente" soluzione corrente sia attualmente il seguente:

```
+---+---+ +---+---+
| 3 | 5 | | 5 | 6 |
+---+---+ +---+---+
```

Una nuova tessera può essere posizionata in coda al "serpente" se e solo se almeno una delle due sezioni contiene il valore 6. Inoltre, il posizionamento dovrà essere tale per cui la sezione di valore 6 sia rivolta verso la coda corrente del serpente. Quindi se abbiamo a disposizione la tessera:

```
+---+---+
| 2 | 6 |
+---+---+
```

il posizionamento che segue **non** è corretto:

```
+---+---+ +---+---+ +---+---+
| 3 | 5 | | 5 | 6 | | 2 | 6 |
+---+---+ +---+---+ +---+---+
```

Mentre sarebbe corretto posizionare la nuova tessera come segue:

```
+---+---+ +---+---+ +---+---+
| 3 | 5 | | 5 | 6 | | 6 | 2 |
+---+---+ +---+---+ +---+---+
```

La prima tessera posizionata, la testa del "serpente", può essere posizionata in qualunque modo.

La procedura ritorna un vettore di `Placing`, allocato dinamicamente, che rappresenta la soluzione ottima individuata, ovvero il "serpente" più lungo (N.B. potrebbero esistere soluzioni equivalentemente ottime, la procedura deve individuarne una). Il vettore indica, nell'ordine, le tessere scelte e il modo in cui sono state posizionate. Al termine della procedura `ret_size` dovrà contenere la dimensione in numero di elementi del vettore ritornato.

Una tessera può essere utilizzata una e una sola volta nella costruzione del "serpente".

Suggerimento: si esplori lo spazio delle soluzioni facendo crescere il "serpente" sempre solo da un lato. Questo semplifica l'implementazione, garantendo l'individuazione di tutte le disposizioni di tessere possibili.

Esempio completo:

Supponiamo di disporre di 4 diverse tessere:

```
t = { { .val1 = 5, .val2 = 3 }, { .val1 = 5, .val2 = 6 },
      { .val1 = 5, .val2 = 4 }, { .val1 = 6, .val2 = 2 } }
```

Una delle soluzioni ottime (di lunghezza 3) è la seguente:

```
+---+---+ +---+---+ +---+---+
| 3 | 5 | | 5 | 6 | | 6 | 2 |
+---+---+ +---+---+ +---+---+
```

Rappresentata con il vettore di Placing:

```
p = { {.indice = 0, .flipped = true}, {.indice = 1, .flipped = false},
      {.indice = 3, .flipped = false} }
```

Esercizio 3

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

```
extern Item *CreaDaInterni(const Item *i, double r);
```

La funzione prende in input una lista di punti sul piano cartesiano, rappresentati come coppia di coordinate intere (vedi definizione sotto), ed un raggio `r`. La funzione deve creare e ritornare una **nuova** lista, contenente tutti e soli gli elementi di `i` interni al cerchio di raggio `r` centrato nell'origine degli assi cartesiani. La lista ritornata deve contenere gli elementi nello stesso ordine con cui essi compaiono in `i`.

Si ricordi che la formula per calcolare la distanza tra due punti $A(x_A, y_A)$, $B(x_B, y_B)$ è la seguente:

$$d = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}$$

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

```
typedef struct {
    int x;
    int y;
} Point2D;

typedef Point2D 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.

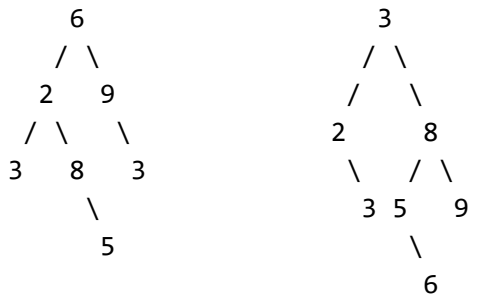
Esercizio 4

Nel file `tree2bst.c` definire la funzione corrispondente alla seguente dichiarazione:

```
extern Node *Tree2Bst(const Node *t);
```

Dato un albero binario di `int`, `t`, la funzione deve attraversarlo utilizzando una visita in *in-ordine* e costruire con i suoi elementi un **nuovo** BST; ogni elemento deve essere aggiunto al BST nell'ordine di esplorazione di `t`, con il seguente criterio: i figli di sinistra di un nodo devono essere sempre minori o uguali al padre, i figli di destra devono essere maggiori del padre. Se l'albero `t` è vuoto la funzione deve ritornare un BST vuoto.

Dato ad esempio l'albero a sinistra, la funzione deve ritornare il BST a destra.



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

```
typedef char ElemType;

struct Node {
    ElemType value;
    struct Node *left;
    struct Node *right;
};
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 `push.c` definire la procedura corrispondente alla seguente dichiarazione:

```
extern void Push(Heap *h, const ElemType *e);
```

Dato un min-heap di elementi di *qualunque tipo*, `h`, e un valore, `e`, la procedura aggiunge l'elemento di valore e all'heap, garantendo che le proprietà min-heap siano rispettate. La procedura deve avere complessità computazione $O(\log_2 n)$.

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 `Push()` 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.