

# Simulazione di Esame di Laboratorio del 31/05/2022

---

## 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, così da poter effettuare il debug delle funzioni realizzate!

## Esercizio 1

Dato un numero  $n$  positivo, la sequenza di *Hailstone*, definita dal matematico tedesco Lothar Collatz, si ottiene a partire da  $n$  procedendo come segue:

1. Se  $n$  è pari il prossimo elemento della sequenza si ottiene facendo  $n/2$ ;
2. Se  $n$  è dispari l'elemento successivo è dato da  $3*n + 1$ .

Il procedimento si ripete per il nuovo  $n$ . Ad esempio, la sequenza ottenuta a partire da  $n = 5$  è la seguente: 5, 16, 8, 4, 2, 1, 4, 2, 1, .... O ancora, se  $n = 11$  abbiamo: 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, .... Come si può notare, una volta arrivati ad 1 la sequenza si ripete all'infinito.

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

```
hailstone <s>
```

Il programma prende in input un numero  $n$  e deve stampare a video (stdout) la sequenza di *Hailstone* costruita **ricorsivamente**, ovvero facendo uso di una funzione ricorsiva, a partire da  $n$ . La ricorsione e quindi il programma deve interrompersi quando viene incontrato il primo 1 della sequenza.

Il formato dell'output deve essere il seguente: ogni elemento fatta eccezione per l'ultimo, deve essere seguito dai caratteri <virgola> e <spazio>.

Se il valore  $n$  passato al programma è minore o uguale di zero, questo termina senza stampare nulla.

Se il numero dei parametri passati al programma non è corretto, questo termina con codice di errore -1 senza stampare nulla. In tutti gli altri casi il programma termina con codice di uscita 0.

## Esercizio 2

Creare i file `prezzo.h` e `prezzo.c` che consentano di definire la seguente struttura:

```
struct Articolo {  
    char nome[11];  
    int prezzo;  
};
```

e la procedura:

```
extern void TrovaArticoli(const struct Articolo *a, size_t a_size, int sum);
```

Dati in input un vettore di articoli, `a`, la sua dimensione, `a_size`, e un numero intero, `sum`, la procedura implementa un algoritmo di backtracking che individua e stampa su `stdout` tutti i gruppi di articoli il cui valore totale è `sum`. L'output dovrà avere il seguente formato:

```
nome_articolo_1, nome_articolo_2, ↓
nome_articolo_2, nome_articolo_3, ↓
nome_articolo_4 ↓
```

Ogni riga contiene una soluzione, ovvero i nomi degli articoli la cui somma dei prezzi vale `sum`. I nomi degli articoli sono separati dai caratteri " , " .

Se il problema non ammette soluzioni o se il vettore di articoli è vuoto la funzione non stampa nulla. In output non devono comparire soluzioni ripetute. Un output del tipo:

```
nome_articolo_1, nome_articolo_2, ↓
nome_articolo_2, nome_articolo_1, ↓
```

è da considerarsi sbagliato in quanto la stessa soluzione compare due volte. L'ordine con cui vengono stampate le soluzioni non è significativo.

### Esercizio 3

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

```
extern Item *ConcatenaN(Item **v, size_t v_size);
```

La funzione prende in input un vettore di puntatori `Item*`, `v`, e la sua dimensione `v_size`. Ogni elemento del vettore `v` contiene l'indirizzo della testa di una lista o `NULL`. La funzione deve concatenare tutte le liste contenute in `v`, a partire dalla prima, e ritornare la lista risultante. La funzione deve concatenare le liste esistenti e non crearne una nuova!

Se tutti i puntatori in `v` sono `NULL` o se `v_size` è 0, la funzione deve ritornare una lista vuota.

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:

```
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);

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 `read.c` definire la funzione corrispondente alla seguente dichiarazione:

```
extern Node *TreeRead(const char *filename);
```

La funzione prende in input il nome di un file di testo, `filename`, che contiene i dati di un albero binario di caratteri. I dati all'interno del file sono così strutturati:

- Ogni carattere alfanumerico identifica un nodo dell'albero;
- Le foglie sono precedute dal carattere ' . ';
- Ogni nodo è seguito dal figlio sinistro (che a sua volta può avere figli) e poi da quello destro (che a sua volta può avere figli);
- Ogni nodo, ad eccezione delle foglie, ha sempre entrambi i figli;
- I *whitespace* ( ' ', '\t', '\r', '\n', '\v', '\f') all'interno del file non hanno alcun significato.

Dato ad esempio il file:

```

a
  .x
  b
    d
      .s
      .u
    .c

```

la funzione deve produrre l'albero:

```

      a
     /\
    x  b
     /\
    d  c
   /\
  s  u

```

Dal momento che i *whitespace* non hanno significato, il file che segue corrisponde anch'esso all'albero appena mostrato:

```
a.xbd.s.u.c
```

La funzione deve aprire il file in modalità lettura tradotta, leggerne il contenuto e costruire l'albero corrispondente che deve quindi essere ritornato. Si assuma che il file di input sia sempre correttamente

formato, ovvero che rispetti sempre la sintassi precedentemente definita. Se non è possibile aprire il file o se il file è vuoto la funzione deve ritornare un albero vuoto (NULL).

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 `isheap.c` definire la funzione corrispondente alla seguente dichiarazione:

```
extern bool IsHeap(const Heap *h);
```

La struct `Heap` di input, `h`, memorizza nel vettore puntato da `data` un albero binario di elementi *di qualunque tipo*. La dimensione del vettore è riportata nel campo `size` della struct. L'albero di input è quasi completo, ovvero tutti i livelli, ad eccezione eventualmente dell'ultimo, sono completi; nell'ultimo livello possono mancare alcune foglie consecutive a partire dall'ultima foglia a destra.

La funzione deve scorrere l'albero di input e ritornare `true` se questo rappresenta un min-heap o è vuoto, `false` altrimenti. Se `h` è un puntatore a NULL la funzione ritorna `false`.

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

```
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 *h);
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 OIJ, così come la loro documentazione.