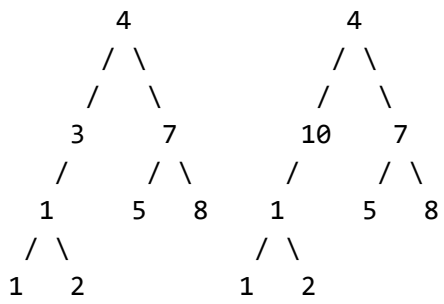
Esercitazione di Laboratorio del 21/05/2021: Alberi

Esercizio 1

Un albero binario di ricerca (dall'inglese Binary Search Tree - BST) deve soddisfare le seguenti tre proprietà:

1. Il sottoalbero sinistro di un nodo contiene soltanto chiavi minori (o al più uguali) alla chiave del nodo stesso;
2. Il sottoalbero destro di un nodo contiene soltanto chiavi maggiori della chiave del nodo;

L'albero di sinistra, ad esempio, è un albero BST perché rispetta le proprietà appena elencate. Quello di destra, invece, è un albero binario **non** BST in quanto la chiave `10` viola le proprietà.



Nel file `insert.c` implementare la definizione delle seguenti funzioni:

```
extern Node *BstInsert(const ElemType *e, Node *n)
extern Node *BstInsertRec(const ElemType *e, Node *n)
```

Dato un albero BST eventualmente vuoto, `n`, e una chiave, `e`, la funzione `BstInsert()` crea ed aggiunge al BST un nuovo nodo di chiave `e`. La funzione garantisce che siano rispettate le proprietà BST dopo l'inserimento del nodo e ritorna l'albero risultante, ovvero il puntatore al nodo radice. Se l'albero di input è vuoto, la funzione deve ritornare un nuovo albero costituito dalla sola radice di chiave `e`.

La funzione `BstInsertRec()` si comporta come `BstInsert()`, ma **utilizza un approccio ricorsivo**.

Si scriva un opportuno `main()` di prova per testare le funzioni.

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;
};
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.

Su OLJ dovete sottomettere solamente il file `insert.c`

Esercizio 2

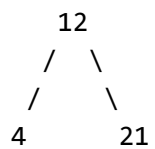
Un nodo di un albero è detto *dominante* se non è una foglia e se contiene un valore maggiore della somma dei valori contenuti nei suoi due figli (destro e sinistro).

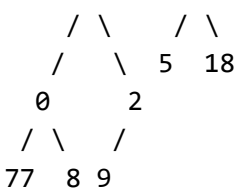
Nel file `dominant.c` implementare la definizione della funzione:

```
extern int CountDominant(const Node *t);
```

La funzione prende in input un albero binario e deve restituire il numero di nodi dominanti in esso contenuti. Se l'albero è vuoto o non contiene nodi dominanti la funzione deve ritornare 0. Si scriva un opportuno `main()` di prova per testare la funzione.

Ad esempio, l'albero che segue contiene un solo nodo dominante, quello di valore 4:





Quanti sono i nodi dominanti in un albero BST?

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.

Su OLJ dovete sottomettere solamente il file `dominant.c`

Esercizio 3

Nel file `treemax.c` implementare la definizione delle seguenti funzioni:

```
extern const ElemType *BstTreeMax(const Node *n);
extern const ElemType *TreeMax(const Node *n);
```

La funzione `BstTreeMax()` prende in input un BST e ritorna l'indirizzo dell'elemento di valore massimo. La funzione deve cercare l'elemento sfruttando le proprietà BST, minimizzando il numero di accessi ai nodi. Se l'albero è vuoto la funzione ritorna `NULL`.

La funzione `TreeMax()` prende in input un albero binario, non necessariamente BST e ritorna l'indirizzo dell'elemento di valore massimo. Se l'albero è vuoto la funzione ritorna `NULL`.

In entrambi i casi, se nell'albero sono presenti più nodi aventi chiave massima la funzione deve ritornare l'indirizzo di quella corrispondente al nodo più **vicino** alla radice. Se due chiavi di valore massimo si trovano allo stesso livello (questo può accadere solo se siamo in presenza di un **non** BST) la funzione deve ritornare l'indirizzo dell'elemento più a **sinistra**.

Qual è il costo computazionale delle due funzioni?

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;
};
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.

Su OLJ dovete sottomettere solamente il file `treemax.c`

Esercizio 4

Nel file `treemin.c` implementare la definizione delle seguenti funzioni:

```
extern const ElemType *BstTreeMin(const Node *n);
extern const ElemType *TreeMin(const Node *n);
```

La funzione `BstTreeMin()` prende in input un BST e ritorna l'indirizzo dell'elemento di valore minimo. La funzione deve cercare l'elemento sfruttando le proprietà BST, minimizzando il numero di accessi ai nodi. Se l'albero è vuoto la funzione ritorna `NULL`.

La funzione `TreeMin()` prende in input un albero binario, non necessariamente BST e ritorna l'indirizzo dell'elemento di valore minimo. Se l'albero è vuoto la funzione ritorna `NULL`.

In entrambi i casi, se nell'albero sono presenti più nodi aventi chiave minima la funzione deve ritornare l'indirizzo di quella corrispondente al nodo più **lontano** dalla radice. Se due chiavi di valore minimo si trovano allo stesso livello (questo può accadere solo se siamo in presenza di un **non** BST) la funzione deve ritornare l'indirizzo dell'elemento più a **destra**.

Qual è il costo computazionale delle due funzioni?

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;
};
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.

Su OLJ dovete sottomettere solamente il file `treemin.c`

Esercizio 5

Nel file `delete.c` implementare la definizione della seguente funzione:

```
extern Node *DeleteBstNode(Node *n, const ElemType *key);
```

La funzione prende in input un BST, n, e una chiave, key. La funzione deve eliminare dell'albero il nodo avente la chiave specificata (se presente), assicurando che le proprietà BST siano rispettate. La funzione ritorna quindi l'albero risultante.

Nel caso siano presenti più nodi di chiave key la funzione elimina quello più a sinistra, ovvero quello che si trova a profondità maggiore.

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;
};
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.

Su OLJ dovete sottomettere solamente il file `delete.c`

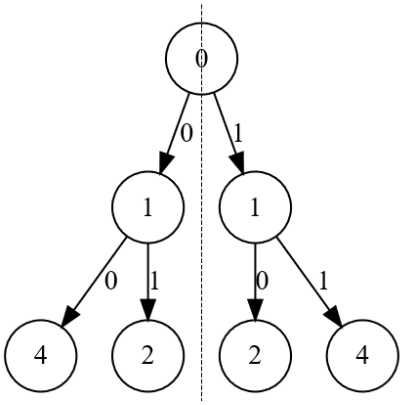
Esercizio 6

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

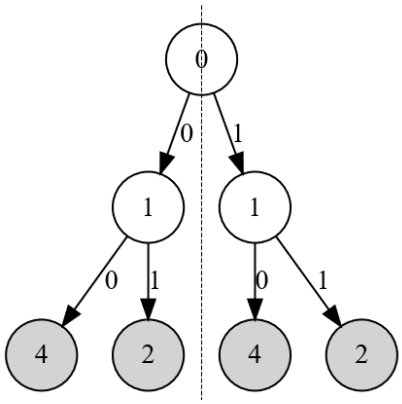
```c
extern bool TreeIsMirror(const Node *t);
```

La funzione prende in input un albero binario di `int` e deve ritornare `true` se questo è specchio di se stesso, ovvero se è simmetrico rispetto all'asse centrale, `false` altrimenti. Alberi vuoti o contenenti solamente la radice sono da considerarsi simmetrici.

Ad esempio, l'albero:



è specchio di sé stesso e quindi la funzione deve ritornare `true`, mentre l'albero:



non lo è (in grigio i nodi che non rispettano la proprietà di simmetria) e quindi la funzione deve ritornare `false`.

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;
};
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.

Su OLJ dovete sottomettere solamente il file `mirror.c`