Computer Architecture
Homework 4

In this homework, we use the following structure type in order to encode binary trees of integers:

```
struct node{
  int key;
  struct node *left;
  struct node *right;
};
```

Exercise 1.

§1-1. Write a function which, given an integer and two binary trees, constructs a new node:

```
struct node *mknode(int x, struct node *left, struct node *right);
```

§1-2. Write a function

```
struct node *full_tree(int h)
```

which constructs a complete binary tree of height h whose nodes have all keys equal to 1.

§1-3. Write a function

```
void print_infix(struct node *tree)
```

which prints the node of the tree `tree` in the infix order. Write a program which prints the content of the complete binary tree of height 4. Check that it contains $1+2+4+8=15$ nodes.

§1-4. Write a function

```
void free_tree(struct node *tree)
```

which frees the memory space allocated for the tree of root `tree`. Modify your previous program in order to use the `free_tree` function and check with valgrind (or any other similar debugger) that all the memory has been deallocated.

Exercise 2.

A binary search tree (BST) is a binary tree whose nodes satisfy the following constraint: the key of every node in the tree is larger or equal to all the keys appearing in its left subtree, and smaller or equal to all the keys appearing in the right subtree.

§2-1. Write a function

```
int find(int val, struct node *bst)
```

which returns true if and only the integer `val` appears as a key somewhere in the binary search tree. You are welcome to write the function in an imperative or in a recursive style.

§2-2. Write a function

```
struct node *insert(int val, struct node *bst)
```

which inserts a node with key `val` in the tree, and does nothing if the key `val` appears already in the tree.

§2-3. Write a program which reads an integer k, reads k integers $a_1, \ldots, a_k$, stores them in a binary search tree ; then reads an integer *val* and indicates if this integer *val* appears as a key in the binary search tree.

Exercise 3.

In this last exercise, we are interested in the problem of deciding whether a given binary tree is in fact a binary search tree.

§3-1. Write a function

```
int max_tree(struct node *tree)
```

which returns the largest key appearing in a binary tree `tree`. Write a similar function

```
int min_tree(struct node *tree)
```

which returns the smallest key appearing in a binary tree `tree`. From this, deduce a function

```
int is_BST(struct node *tree)
```

which returns true if and only if the binary tree `tree` is a binary search tree.

§3-2. We would like to understand better the complexity of deciding whether a binary tree is in fact a binary search tree. In order to visualize the underlying algorithm, write the instruction

```
printf ("*");
```

in the first line of the functions `max_tree`, `min_tree` and `is_BST`. Each printed star `*` at run-time will then correspond to the call of one of these three functions.

§3-3. In order to decide faster whether a binary tree is a binary search tree, we would like to explore each node of the tree a *bounded* number of times. To that purpose, we propagate an interval which describes the possible values of the key during the exploration of the tree. Typically, suppose given a binary tree which satisfies the interval constraint that all its keys are in the interval $[a, b]$ and whose root node n has key val. In that case, the binary tree is a binary search tree if and only:

- the integer val is an element of the interval $[a, b]$,
- the left tree is a binary search tree which satisfies the interval constraint $[a, val]$,
- the right tree is a binary search tree which satisfies the interval constraint $[val, b]$.

Write a function

```
int is_BST_helper(struct node *tree, int a, int b)
```

which checks in linear time whether the tree `tree` is a binary search tree with all keys in the interval $[a, b]$. Deduce a function

```
int fast_is_BST(struct node *tree)
```

which decides in linear time whether the binary tree `tree` is a BST (you are allowed to use the functions `min_tree` and `max_tree` encoded before).

§3-4. Evaluate the complexity of the function by printing stars `*` as done in exercise §3-2.

References and acknowledgments: among all the practice exercises in C which I found in the literature, I most enjoyed these exercises designed by Juliusz Chroboczek and Yan Jurski, which I thus decided to translate and to adapt from French.