Binary Search Tree

This **project** aims to implement a binary search tree (BST) in C to store and manage a **dictionary**. The BST will store information about keywords, including a description for each keyword. You will implement the essential BST operations. **It is recommended that you use only commands that are taught in the course material.**

The idea of a binary search tree is that it is built of **nodes**. Nodes are organized so that for each node, the values stored in the left subtree are less than the values stored in the node. Similarly, the values stored in the right subtree are greater than the values stored in the node. This organization guarantees that searching and inserting a value is efficient: we only need to travel one path from the root to one leaf. If the search tree is well-balanced, then its height is approximately lg(*n*), that is, the base-2 logarithm of the size *n* of the tree.

In your project, you need to use the following definition of nodes:

```
typedef struct node {
    char key[30];
    char description[200];
    struct node *left;
    struct node *right;
} Node;
```

The interpretations of the fields are:

- `key`: A string (up to 29 characters + null terminator) representing the keyword that identifies the node. The key consists only of one word. Does not contain any newline characters.
- `description`: A string (up to 199 characters + null terminator) containing information about the keyword. Does not contain any newline characters.
- `left`: A pointer to the root of the left subtree. All keys in the left subtree are lexicographically smaller than the key in the current node.
- `right`: A pointer to the root of the right subtree. All keys in the right subtree are lexicographically greater than the key in the current node.

Your task is to implement the following functions for **nodes**

Function **createNode** takes *key* and *description* as parameters and returns a new node. If *key* or *description* is `NULL`, it prompts the user to enter the string. The left and right child pointers of the new node are set to `NULL`.

The function **printNode** works so that if the node is not NULL, it prints the key and description of the node. If the node parameter is NULL, it prints the message: `The node is NULL`.

The BST consists of nodes. The binary tree is defined as a structure having just one field, which is a pointer to the root of the tree.

```
typedef struct {
    Node *root;
} Tree;
```

Your task is to implement the following **functions** related to tree. You can decide the prototype of the functions by yourself. Also the functions can be recursive or iterative.

The function **createTree** creates an empty binary tree. It initializes the root pointer of the newly created tree to `NULL` (indicating an empty tree), and **returns** a pointer to the new `Tree` structure.

The function **treeEmpty** checks whether a given binary tree is empty. It returns 1 (true) if the tree is empty and 0 (false) otherwise.

The function **insertNode** inserts a new **node** into a binary search tree (not values, but a node). If the tree is empty, the new node becomes the root. Otherwise, it inserts the new node into the correct subtree as a leaf. If a node with the same key already exists in the tree (a duplicate), the function prints a message stating this and frees the memory allocated for the new node.

The function **inorderPrint** performs an inorder traversal of a binary tree, printing at most `n` nodes. This number `n` is given as a parameter. A counter (`count`) tracks the number of printed nodes, stopping the traversal once `n` nodes have been printed. Note that this function should print the key in ***lexicographical*** order.

The function **search** searches the BST for a node with the given key. It returns a pointer to the node if found and returns NULL if the key is not found.

The function **delete** deletes the node with the given `key` from the BST. Consider the different cases of deletion (leaf node, node with one child, node with two children). If a node has two childs, we delete its ***inorder successor.*** The inorder successor is guaranteed to have at most one child (why?) and copy the values of the successor to the node which contained the value to be deleted. Free the memory occupied by the deleted node.

The function **loadTextFile** reads data from a text file with the given filename. Your function should be compatible with the file `dictionary.txt` that can be found in Moodle. The beginning of the file looks like this:

```
Jewel:A precious stone or gem.
Ink:A colored fluid used for writing or printing.
Earthquake:A sudden and violent shaking of the ground.
```

The `key` consists always of one word. The colon (:) separates the key from the description. Spaces are not allowed around the colon. The text ends with a dot (.) and lines are terminated by just “\n” (Unix/Linux/macOS). You should create a node from each line, and store the node in a tree which is given as a parameter. Blank lines are **not** allowed in the input file. The function should handle potential errors, such as: (i) file not found, (ii) invalid file format (e.g., missing colons, incorrect number of elements), and (iii) memory allocation failures.

The function **storeTextFile** stores your current BST to a file of the same format as `dictionary.txt`. This means that you can load your stored file with the above-defined function

loadTextFile. The data should be written to the file so that the stored tree is traversed using the https://www.geeksforgeeks.org/preorder-traversal-of-binary-tree/ This guarantees that if the data is loaded again from the file, *the tree will be more or less balanced*. Similarly, storeTextFile should handle errors like file opening failures.

A **freeTree** function prevents memory leaks. It frees all the nodes of the tree given as a parameter.

The **main program** should be a **menu-based** function which has a menu item for all the above-listed functions and for the exit.

Documentation

Your project documentation (document.pdf) must include the following:

AI declaration: A clear statement of which AI tools (if any) you used during the project and how they were used (e.g., code generation, debugging assistance, research). Be specific about the tools and their role. AI declaration **must be included**, otherwise the work will be **rejected** in this round.

Implementation: Describe in detail the functionality of each function, including the main program. Describe any important implementation details and design choices you made. If you decide to do "code division" and/or `makefile`, please document also these.

Testing: Give a "proof" that your functions are working. Describe how you have tested your software and include the following cases:

1. **Inserting nodes**: Include also what happens when you try to insert a duplicate.
2. **Searching for nodes**: Test for both existing and non-existing keys.
3. **Deleting Nodes**: Your test should include deleting leaf nodes, nodes with one child, nodes with two children, and the root node.
4. **Loading and storing files**: Check that these operations do not miss data and that everything works as should.
5. **Printing the tree**: Note that there is a parameter `n`, which tells to print only `n` nodes.
6. **Memory leak detection**: End your testing with **freeTree** function. Use Valgrind and include the Valgrind output to your documentation.

Grading

Correct implementation (0-60 points): The main criterion is that the functions work as instructed in all situations. If there are issues, points will be deducted.

Documentation (0-20 points): Clear and thorough documentation helps in understanding the code and maintaining it in the future.

Code Division: You may have an additional **10 points** if you divide your code into separate C source files together with header files. The division needs to be according to these instructions:

- Functions for node are in the file `Node.c`. In addition, you provide a header file `Node.h` that contains the definition of the Node and the declaration of node functions.
- Functions for the are in the file `Tree.c`. Furthermore, you need to write a header file `Tree.h` that contains the definition of the tree and the declarations of tree functions.
- The main program is in the file `main.c`.

The project must compile correctly using the command:
```
gcc -Wall main.c Tree.c Node.c -o project
```

The last step is to write a `makefile`, which first compiles C source codes to **object codes**, and then links the object codes to one **executable**, whose name is `project`. This will give you 10 **points** if done correctly. The **makefile** needs to be written according to the instructions given in the course materials.

To pass the project work, you need to gather ***at least 51 points***. The **grade** of the project work is determined by this table:

Points	51-60	61-70	71-80	81-90	91-100
Grade	1	2	3	4	5

Submission:

Submit your C source code files, a Makefile (if you have one), and your project documentation to CodeGrade by the deadline.