CSC 218, Assignment 2

Due 20 February 2003 at midnight

As opposed to the linked list discussed in class (which is a linear data type), a *binary tree* is not linear. A node in such a tree contain some information (say, strings of characters) and links to two "children" (call them left and right child), which are themselves (possibly empty) binary trees. A node with both children empty is called a *leaf*. The (only) node of a tree without a parent is called the *root node* of that tree.

A *search tree* is a binary tree with the following additional property: Given any search tree t holding information i in its root node and with left and right children l and r, respectively, it holds that i is larger than any information stored in l, and strictly smaller than any information stored in r.

Of course, a search tree only holds data that over which a total order relation exists. In particular, strings do have a (natural) total order, namely the "lexicographic" order, as established by `strcmp`. We shall refer to this order in what follows.

In this assignment, you will implement and work with binary and search trees.

Note. The functions `deltree`, `instree`, and `delnode` described below receive a tree and returns the altered tree. However, the *tree content of the argument shall also be changed* (i.e., these functions are "surgical"). At a first look, the return type of these functions can thus be `void` instead of `btree`, i.e., the following two calls would work be in most cases similar:

```
some_tree = deltree(some_tree,a_node_in_some_tree);
deltree(some_tree,a_node_in_some_tree);
```

However, the return value is needed when you change the *root node*. Indeed, the following two calls are no longer equivalent, the second giving incorrect results:

```
some_tree = deltree(some_tree,some_tree);
deltree(some_tree,some_tree);
```

1. Implement a binary tree data type which holds strings (i.e., *pointers to characters*). That is,
   - Define a suitable C++ structure for nodes in a binary tree, and a type `btree` for binary trees (as some form of a pointer).
   - Implement the following functions:
     - `btree mknode(char* info, btree left, btree right)`
       creates and returns a node holding `info` and with `left` and `right` as children. Use shallow copy for `info`.

- `btree deltree(btree tree, btree to_delete)`
  deletes the *tree* `to_delete` from `tree` and returns the result, i.e., `tree` (which will be thus altered).
- `char* key(btree tree)`
  returns the information (i.e., the pointer to the string) held in the root node of `tree`.
- `btree lefttree(btree tree)` and `btree righttree(btree tree)`
  return the left and right child of `tree`, respectively.

Provide your implementation of binary trees so that it could be used in other programs, i.e., as a header and a C++ file.

2. Using the implementation of `btree` constructed for Question 1, implement the following operations over search trees.

   - `btree memtree(btree tree, char* info)`
     searches for `info` in the search tree `tree` and returns the (pointer to the) node that holds `info`; returns `NULL` whenever `info` is not in `tree`.

     Searching for information i in a search tree t is performed as follows: If the information in the root node of t is identical to i, return t. If on the other hand i is smaller than the information held in the root node of t, set t to be the left child of t and repeat from beginning. Finally, if i is larger than the information held in the root node of t, set t to be the right child of t and repeat from beginning.
   - `btree instree(btree tree, char* info)`
     inserts `info` in the search tree `tree` and returns the result, which must also be a search tree. The insertion process also alters `tree`. Use shallow copy for `info`.

     Inserting in a search tree is accomplished by reaching the appropriate leaf node and creating a new child of that leaf holding the new info.
   - `btree delnode(btree tree, btree node)`
     deletes the *node* `node` from `tree` and returns the result. Also changes `tree`.

     Deleting a node n in a search tree is accomplished by finding the maximal node lm of the left child of n, inserting into n the information contained in rr, and then deleting (recursively) lm. Deleting lm is quite simple, because it has at most one child (and thus you replace it with its sole child and then delete it physically).

   Provide your implementation of search trees so that it could be used in other programs, i.e., as a header and a C++ file.

3. Using the search tree implementation developed for Question 2, implement the function `void treesort(char** strings, int nstrings)` which receives an array of strings `strings` of length `nstrings`. Upon return of `treesort`, the array `strings` must be sorted in lexicographic order.

   Provide a function `main` that reads from standard input a number n, then n strings one per line (read at most 1024 characres from each line), and prints to the standard output the n strings sorted using `treesort`. I would prefer that your program does not print any prompt when reading n or the strings.

What to submit Make sure you review the submission guidelines on the course's Web page.

Submit your answers to Questions 1, 2, and 3, together with a suitable makefile. Your answer to Question 3 *must* be created by the makefile with the name `treesort`. Target `all` must create `treesort` and any other program you used to test your implementation. Tests must be provided and must be documented.