CSC 218, Assignment 3

Due Thursday, 13 March 2003 at midnight

This is basically a civilized wrapper to the functions on binary trees you already implemented. Review the previous handout.

You can start this assignment from either your solution or my solution for the previous assignment.

1. Implement binary trees over strings as a *class* `btree`. Exactly all the following member functions should be public. No other data should be accessible outside the class, *but* the data *type* for nodes (call it `btree_node`) should be accessible.

   - Provide a void constructor, a copy constructor, and an assignment operator (i.e., `operator=`). *Contrary to what I said here earlier, you may want to use deep copy for everything*, including *the information held in nodes. If you do not do this, you will run into problems when implementing the command* `check` *in Question 3.*
   - `void btree::zap(btree_node* to_delete)` behaves just as the function `deltree` from Assignment 2 (except that it alters the member data instead of returning a new tree). Specifically, given an object `t` of type `btree`, after calling `t.zap(n)` (for some node `n`) `t` will contain the same tree as before, except that the subtree of `t` rooted at `n` is erased and replaced with an empty subtree. Of course, if `t` does not contain `n` at all, then `t` remains unchanged (but the memory allocated for the tree rooted at `n` should be still deallocated).
   - `void btree::print(void)` pretty prints the tree to an output stream. Print the tree so that you can identify the actual structure of the thing. An simple inorder traversal will *not* do. See the function `printtree` from my solution to Assignment 2 for an idea of how the output should look like (although printing the pointers is obviously not necessary).
   - Also implement a suitable operator << for printing trees to an output stream in the same form as the one printed by `btree::print`.
   - Provide a suitable destructor, which deallocates all the allocated data.

   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. Inherit publicly from class `btree` which you created in your answer to Question 1 to create the class `stree` of search trees. Exactly all the following member functions should be public.

   - Provide if necessary suitable constructors (including a copy constructor), assignment operator, and destructor.

- `btree_node* stree::member(char* info)` behaves just as `memtree` from Assignment 2 does. It must return 0 (i.e., NULL) if `info` is not found. Specifically, given an object `t` of type `stree` and a string `s`, calling `t.member(s)` returns the (pointer to) the node that contains `s` or 0/NULL if there is no such a node in `t`.
- `void stree::insert(char* info)` behaves just as `instree` from Assignment 2 does. That is, for some object `t` of type `stree` and some string `s`, after the call `t.insert(s)` `t` will contain the same tree as before, except that it will also contain `s`.
- `void stree::del(btree_node* node)` behaves just as `delnode` from Assignment 2 does. Specifically, given an object `t` of type `stree` and a node `n`, calling `t.del(n)` will result in `t` containing the same tree as before, sans the information held in node `n`.

  For example, you must be able to delete one occurrence of a string `s` from a tree `t` by doing `t.del(t.member(s))`.

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 class developed for Question 2, implement a program that operates interactively on search trees. Specifically, the program accept commands from the standard input. Commands are lines containing the name of the command, possibly followed by one argument (on the same line, separated by a blank from the command name). All the commands operate on one search tree (referred to henceforth as the "current tree"). The following commands should be accepted by your program (arguments are shown in italics).

   `make`
   : Creates anew an empty current tree. If a current tree exists, deletes all of its content and reinitializes it as an empty tree.

   `member` *str*
   : Prints `yes` or `no` depending on whether *str* is contained in the current tree. The argument *str* may contain blanks.

   `delete` *str*
   : Deletes *all* the occurrences of *str* from the current tree. Again, the argument may contain blanks.

   `insert` *str*
   : Inserts *str* into the current tree. Duplicates are allowed, so *str* is inserted again even if it already exists in the current tree. You do not have to ask, of course the argument may contain blanks.

   `print`
   : Pretty prints the current tree, so that not only its content but also its structure is visible.

   `list`
   : Lists the information from the current tree in inorder, i.e., sorted.

   `check`
   : Makes a backup of the current tree. Up to five most recent backups are kept. If there are five backups already, the oldest will be discarded and the new one added.

`restore`

Restores the current tree from the most recent backup and deletes this backup. The previous content of the current tree is discarded. If no backup exists, this command behaves just like `make`.

Commands may be also specified by an unambiguous prefix. For instance, all of the following commands should be accepted:

```
ma
i some string
in some other string
inse my string
```

However, the command `m` should not be accepted, because it is ambiguous (it may be either `make` or `member`).

If a command other than `make` is issued before the current tree has been initialized, the command should have no effect and an suitable error message should be printed (however, your program should not exit, but wait instead for the next command).

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 default target with the name `itree`. Feel free to include any other program that might tests parts of your implementation that is not tested by your answer to Question 3.

Include a description of your tests, as well as an argument that your tests are complete.

I *will* subtract marks for blatant memory leaks, so be careful when allocating memory dynamically.