CPS222 - DATA STRUCTURES AND ALGORITHMS

Project #3 - Due Wednesday, March 23, at the start of class.

Purposes: To give you experience working with binary search trees
To give you experience working with threaded binary trees.
To give you experience working with C++ pointers
To give you experience with implementing iterators

Note: This project must be done in teams of 2. It is desirable that students of similar C++ background work together

Introduction

For this project, you will be implementing a binary search tree class that stores in each node a key (a character string) and a value (an integer). Such a class might be used to support a telephone directory, where the key is a person's name, and the value the person's phone number.

Your tree will support the usual binary search tree operations insert, empty, lookup and erase, plus Iterators that allow visiting all the nodes in the tree in inorder, reverse inorder, preorder, reverse preorder, postorder, or reverse postorder. To support this, the binary search tree you build will be fully in-threaded, with a header node.

Requirements

1. Create a new server directory: ~/cps222/project3. Copy these files from `/home/cps222/project3` into it:

   `project3.h` `project3.cc` `project3tester.o` `testdemo.in` `Makefile`

2. All of your coding (with one possible exception noted below) will be done in project3.cc. There are 12 methods that currently are unimplemented stubs (so indicated by comments). Your task is to implement some or all of these methods. (There are also many fully-implemented methods you should leave alone)

   a) The completion of all of the following is required, and will be worth 35 points out of the maximum of 70 possible for correct operation:

   Methods of class ThreadedBinarySearchTree:

`~ThreadedBinarySearchTree()`	- destructor - must delete <u>all</u> nodes in the tree.
`void insert(string key, int value)`	- insert a new node
`bool empty() const`	- return true iff the tree is empty
`Iterator lookup(string key) const`	- returns an iterator to node containing key, or `end()` if not found

   Methods of nested class ThreadedBinarySearchTree::Iterator:

`Iterator & insucc()`	- Advances to next node in inorder traversal (see implementation notes below)
`Iterator & inpred()`	- Advances to next node in reverse inorder traversal

b) The implementation of the following will be worth 5 additional points each (maximum 25 for all five). (All are methods of the nested Iterator class).

`Iterator parent() const`	- returns a new iterator to the parent of the current node
`Iterator & presucc()`	- Advances to next node in preorder traversal
`Iterator & prepred()`	-Advances to next node in reverse preorder traversal
`Iterator & postsucc()`	- Advances to next node in postorder traversal
`Iterator & postpred()`	- Advances to next node in reverse postorder traversal

Note: `parent()` is needed by `prepred()` and `postsucc()`, and so should be working correctly before these are attempted.

c) The implementation of the following will be worth 10 additional points, with partial credit possible if some cases work correctly and others do not. It is recommended you not attempt it until all of the other methods are working. (In fact, you must have `parent()` working to make it work). It is a method of class ThreadedBinarySearchTree.

`void erase(Iterator iter)`	- removes the node pointed to by iter.

3. You can compile your project3.cc file, link it, and run it with a file of test data using the following commands:

```
make
./project3 < testfile
```

4. Create one or more files of data to thoroughly exercise your program - see discussion below on how to do this. Turn in your file(s), together with the output produced by your program. If your test run output shows any errors in your program that you were unable to fix, indicate them on the output you turn in, so I will know that you spotted them. The thoroughness of your test data will account for 10 points toward your project grade. Turning in output with errors you didn't catch, or failing to include test cases that catch errors you should have caught, will result in a significant reduction in the grade for this item. (I.e. don't deal with errors in your program by eliminating the test cases that identify them - unless, of course, the result is to crash the program and prevent further testing. In this case, hand in a separate explanation of what test data causes the crash.)

5. As always, make good use of internal comments and whitespace to document your program and make it readable. Methodology and documentation will account for 10 points toward your project grade.

Summary of Grading:	Required methods	35
	pre/post pred/succ, parent (5 each)	25
	erase	10
	Testing	10
	Methodology/documentation	10
	Quiz	10
		100

Implementation Notes:

1. As an example of what the code you write for the various ___succ() and ____pred() operations should look like, the following is the code for `insucc()`.

```
Iterator & Iterator::insucc()
{
    // If the node has a thread for a right child, then the thread points to
    // the inorder successor; else, we go left as far as possible in the right
    // subtree

    if (isThread(_ptr -> _rchild))
        _ptr = makePointer(_ptr -> _rchild);
    else
    {
        _ptr = _ptr -> _rchild;
        while (! isThread(_ptr -> _lchild))
            _ptr = _ptr -> _lchild;
    }
    return * this;
}
```

2. Three static methods are declared in class ThreadedBinarySearchTree to help you work with threads. The implementation code for these is supplied as part of project3.cc. (But you should not modify it in any way!)

`Node * makeThread(Node * ptr) -`	Takes an ordinary pointer as an argument, and returns the same pointer marked as a thread (i.e. with the high order bit set to 1. (The argument is not altered - a new pointer is returned.)
`bool isThread(Node * ptr) -`	Takes a pointer or a thread as an argument, and returns true iff its argument is a thread (i.e. its high order bit is set to 1)
`Node * makePointer(Node * ptr) -`	Takes a pointer marked as a thread as an argument, and returns as its value the same pointer no longer marked as a thread (i.e. with the high order bit set to 0). (The argument is not altered - a new pointer is returned.)

Warning: attempting to use a thread as a pointer without running it through makePointer() will cause unpredictable results and may crash your program - e.g. if p is a threaded pointer, then

```
r = p -> _rchild;                    // is incorrect, and may crash
```

... But:

```
t = makePointer(p);
r = t -> _rchild;                    // is correct
```

... Or:

```
r = makePointer(p) -> _rchild;       // is also correct
```

3. The code supplied for you (including the constructor for the tree) assumes that you will build a tree having a header node that corresponds to the following conventions:

   a) Its _lchild will point to the actual root of the tree. If the tree is empty, then the header's _lchild will be a **thread** to the header itself. (This initial state is created by the constructor.)

   b) Its _rchild will point to the header itself (actual pointer, **not a thread**). (This initial state is created by the constructor.)

4. The first node in the tree (in inorder) should have a thread to the header as its _lchild. The last node in the tree (in inorder) should have a thread to the header as its _rchild.

5. The various __succ and __pred methods should work correctly when applied to an Iterator that points to the header. The result should be the first/last node in the tree in the specified order - respectively.

6. The `end()` method returns an Iterator that points to the header. The various __succ and _pred methods should make their Iterator point to the header when they go past the last node in the tree in their specified order, `parent()` should return this value if applied to the root of the tree, and `lookup()` should return this value if no node containing the key occurs in the tree. (In many cases, this will happen without special effort if the tree is threaded correctly and the methods are implemented correctly.) The value NULL should **never, never, never** occur!

7. For implementing the `insert()` method, you may decide to do so either with a loop or by means of recursion. If you choose the latter approach, you will need to create a private recursive auxiliary procedure. The prototype for this may be added to the private section of the header file. You may not alter anything in the public section, including the declaration of the main insert method (which the test driver calls.) Adding a private recursive auxiliary `insert()` method is the only change you may make to the class declaration.

8 When implementing the `~ThreadedBinarySearchTree()` method, use an iterator to systematically visit and delete all the nodes in the tree. Be sure to advance the iterator before deleting the node! Note that you should not use the `erase()` method here - it is unnecessary, since you have no need to preserve the structure of the tree. (And besides, you will not have implemented it when you first implement the destructor.)

Testing procedure:

1. The tester will repeatedly prompt you for a command by printing the prompt "Command?". A command consists of a single character, as shown below - in some cases followed by one or two parameters. (Command letters may be either upper or lower case; keys are case sensitive.)

2. | | |
|---|---|
| `?` | Call the isempty() method and report the result |
| `I key value` | Insert a new key/value pair in the tree |
| `T order` | Do a forward traversal - order can be 1 to specify inorder, 2 to specify preorder, 4 to specify postorder, or an orring together of two or more values - e.g. 7 would do all three orders. (The appropriate `__succ()` method is called repeatedly to do the traversal.) |
| `R order` | Do a reverse traversal - same order values as above. (The appropriate `__pred()` method is called repeatedly to do the traversal.) |
| `L key` | Lookup the key - report if found and information in the node |
| `P key` | Lookup the key, and then invoke the `parent()` method on the result and report information found. (If lookup fails to find the node, `parent()` is invoked on the header.) |
| `E key` | Lookup the key, and then erase it from the tree. (Can't do if not found!) (After erasing keys, the tree should be traversed to ensure that the node is really gone and that the tree structure has been preserved.) |
| `H` | Print a help message - list of valid commands |
| `Q` | Quit the tester |
| `# anything` | Comment - the line is echoed to the output, but otherwise ignored |

   (Note: output from the tester will be neater if all keys are kept to 14 characters or less.)

3. The traversal, lookup, and parent commands result in printing out nodes in the tree, using a format like the following:

```
0x6eb70    aardvark       - value:   1 - children: thr header         0x6eb90
0x6eb90    bee            - value:   2 - children: thr 0x6eb70    thr 0x6eb50
...
```

   This says that the node at memory address 6eb70 (hex) contains the key aardvark and the value 1. Its left child is a thread to the header, and its right child is the node at 6eb90. The node at 6eb90 holds the key bee and the value 2. Its left child is a thread to the node at 6eb70 (i.e.the node containing aardvark) and its right child is a thread to the node at 6eb50 - not shown.

4. To save you having to type lots and lots of input for every test, you can construct a test data file consisting of a series of commands to be executed. (A simple test file is available to you as an example in the same directory you copied the files from under the name testdemo.in - use this as an example of how to set up a file, not as a comprehensive test of your program).

   To use a file as input, rather than the keyboard, redirect standard input when running your program - e.g. to read from a file called testdemo.dat, you would run your program as follows:

```
project3 < testdemo.dat
```

5. You can also send output to a file, rather than the screen, redirecting standard output - e.g. to read input from testdemo.in and send output to testdemo.out, you would run your program as follows:

```
project3 < testdemo.in > testdemo.out
```

6. When the test driver terminates, it will print a final message reporting on your allocation and deallocation of nodes - e.g. one of the following:

```
You created a total of: 4 nodes
You deleted a total of: 4 nodes
You are a good steward of nodes!
```

Or:

```
You created a total of: 4 nodes
You deleted a total of: 0 nodes
LITTERBUG!
```

Or:

```
You created a total of: 4 nodes
You deleted a total of: 5 nodes
WARNING: YOU DELETED SOME NODE(S) MORE THAN ONCE!
```

Obviously, any message other than one like the first indicates a problem in your program that needs to be fixed!

7. **It is essential that you test your program thoroughly**, as noted under project requirements. Do **not** naively assume that a program that successfully processes the example data in testdemo.in is correct!