Linköping Institute of Technology 28/04/2015
Department of Science and Technology/ITN
Aida Nordman

Data Structures
TND004
Lab 3

Course goals

- To implement the data structures and algorithms for a (simplified) container similar to **`std::map`**. More concretely, **threaded binary search tree** is the data structures in focus in this exercise.
- To implement bidirectional iterators over container objects of the class above.

Preparation

- Review the basic concepts about trees, binary search trees, and tree traversals introduced in Fö 5 and 6. Moreover, sections 4.1 to 4.3 of the course book are relevant for the exercises in this lab.
- During the lesson on week 17, implementation details related to threaded binary search trees were discussed. Review this lesson notes.
- You are also requested to study the documentation of the given classes (**`Node`**, **`BiIterator`**, and **`BST_threaded`**) before the lab session. This documentation is available from the course web page.

If you have any specific question about the exercises, then drop me an e-mail. Be short and concrete, otherwise you won't get a quick answer. You can write your e-mail in Swedish. Add the course code to the e-mail's subject, i.e. "**TND004:** …".

Presenting solutions

You should demonstrate your solutions orally during your fourth lab session on week 20. You also need to deliver a written paper with the answers to specific questions indicated in this lab description. This paper should be provided during the oral demonstration of your code. Hand written answers that we cannot understand are simply ignored. **Do not forget to put the name and LiU login of the group members in the delivered paper**.

Use of global variables is not allowed, but global constants are accepted.

Deadline

Since you can only present one lab in each lab session and week 20 is the deadline to present lab 1 and 2, if you want to present lab 3 on week 20 then you must have presented the first two labs before week 20.

Exercise 1: threaded binary search trees

The aim of this exercise is that you implement a container similar to **`std::map`**, with some simplifications. The data structure supporting the container is a threaded binary search tree (BST). The container class, named **`BST_threaded`**, stores pairs (**`<string, int>`**), where the first component (a **`string`**) is the key by which the BST is sorted.

It should also be possible to iterate through the elements stored in a container (an object of class **`BST_threaded`**), in increasing or decreasing order of the keys. To this end, you are also requested to implement a class representing bidirectional iterators, named **`BiIterator`**.

Finally, you need to write a program that reads a text file and displays a frequency table with all words occurring in the text sorted alphabetically. This task is similar to exercise 1.1 of lab 2 in this course. In this previous lab, you have used a hash table. You should now instead a threaded binary search tree to implement a frequency table.

The three classes you need to implement are briefly described below. Those member functions that you need to implement are marked with "**`//IMPLEMENT`**".

- Class **`BST_Threaded`** represents a container, similar to **`sdt::map`**, built upon a threaded binary search tree. File **`BST_Threaded.h`** contains the class definition, including data members and the member functions such as **`insert`**, **`remove`**, **`size`**, **`find`**, and a subscript **`operator[]`** similar to the one offered by **`std::map`**. As expected the BST consists of a set of linked nodes. As discussed in the lesson, an empty tree contains a dummy node. Thus, every node of the (real) tree has a parent node and this aspect simplifies the implementation code[1].
- Class **`Node`** represents a node of a (threaded) BST. Each node stores a pair **`<string, int>`**, named **`ELEMENT`**. The first component of the pair is a word and the second component is the number of occurrences of the word (e.g. in some text). Words are then used as search keys. Remember that in this exercise the trees are threaded. Thus, two extra boolean flags (**`l_thread`** and **`r_thread`**) are stored in each node, to indicate whether the left or right link are being used as threads.

  Class **`Node`** offers all implementation functions needed to be able to implement the member functions of **`BST_Threaded`**. This means that all implementation details of threaded binary search trees are in the class **`Node`**, in the form of private member functions. Consequently, several member functions of class **`BST_Threaded`** call member functions of class **`Node`**. See for instance **`BST_threaded::insert`** function.

  Note that when removing a node from a threaded BST, it may be needed to transform the link connecting the parent node to the node to be removed into a thread (see case **`1c`** depicted in slide 27 of the lesson). In this case, one of the thread flags in the parent node has then to be set to **`true`**. Consequently, when removing a node, it is needed to have access to the parent node. This is the reason why member functions **`Node::remove`** and **`Node::removeMe`** have an extra argument, named **`parent`**, of type **`Node*`**.

[1] Recall that dummy nodes were also used in the implementation of doubly and singly linked lists for similar reasons.

- Class **`BiIterator`** represents bidirectional iterators to allow for be possibility to iterate through the elements stored in a binary search tree, either in increasing or decreasing order of the keys (i.e. inorder traversal). Internally, an instance of this class stores a pointer to a node. Then, increment and decrement operations, **`BiIterator::operator++`** and **`BiIterator::operator--`**, make possible to advance the iterator to the next or previous node in the inorder enumeration of all keys, respectively.

  To be able to implement iterators efficiently motivates the need of having threads in the binary search tree.

After having studied the documentation of the given code, you should then follow the steps indicated below to proceed to the implementation of the classes **`Node`**, **`BST_Threaded`**, and **`BiIterator`**.

When implementing the class **`Node`**, the member functions **`Node::remove`** and **`Node::removeMe`** should be the last ones to be implemented, since removing a node from a threaded BST is the most challenging operation to implement.

Steps to follow: implementation and testing

1. Download the zipped folder with the required source files from the course website.
2. Create a project in Code::Blocks with the files **`node.h`**, **`node.cpp`**, **`threaded_BST.h`**, **`threaded_BST.cpp`**, **`BiIterator.h`**, **`BiIterator.cpp`**, and **`test1.cpp`**. You should be able to compile and run the program. Although not all functions are really implemented, it is provided a minimum (dummy) code for the member functions such that the program can be executed.
3. Implement the functions the **`BST_threaded`** constructor, destructor, **`BST_threaded::empty`**, and **`BST_threaded::size`**. Do not forget to compile the file **`threaded_BST.cpp`** (use **`Ctrl-Shift-F9`**) to check for any compilation errors before you proceed.
4. Fix any errors and run it. Compare the output with the contents of the file **`test1_out.txt`**. Note that only **`PHASE 0`** tests are active.
5. Implement the function **`Node::insert.`** Note that this function is called by **`BST_threaded::insert`** that is already implemented (see its code). Do not forget to compile the file **`node.cpp`** (use **`Ctrl-Shift-F9`**) to check for any compilation errors before you proceed.
6. Uncomment **`PHASE 1`** tests of **`test1.cpp`**. Compile your program, fix any errors, and run it. Compare the output with the contents of the file **`test1_out.txt`**.
7. Implement the functions **`BST_threaded::find`**, **`BST_threaded::operator[]`**, and **`Node::find`**. Do not forget to compile separately the files **`node.cpp`** and **`threaded_BST.cpp`** (use **`Ctrl-Shift-F9`**) to check for any compilation errors before you proceed.
8. Uncomment **`PHASE 2`** tests of **`test1.cpp`**. Compile your program, fix any errors, and run it. Compare the output with the contents of the file **`test1_out.txt`**.
9. Implement the functions **`Node::findMin`** and **`Node::findMax`**. Do not forget to compile the file **`node.cpp`** (use **`Ctrl-Shift-F9`**) to check for any compilation errors before you proceed.
10. Implement the function **`BST_threaded::begin`**[2] and all member functions of class **`BiIterator`**. Do not forget to compile separately the files **`BiIterator.cpp`** and **`threaded_BST.cpp`** (use **`Ctrl-Shift-F9`**) to check for any compilation errors before you proceed.
11. Uncomment **`PHASE 3`** tests of **`test1.cpp`**. Compile your program, fix any errors, and run it. Compare the output with the contents of the file **`test1_out.txt`**.
12. Implement the function **`BST_threaded::remove`**. Do not forget to compile the file **`threaded_BST.cpp`** (use **`Ctrl-Shift-F9`**) to check for any compilation errors before you proceed.
13. Implement the functions **`Node::remove`** and **`Node::removeMe.`** Function **`removeMe`** is called by function **`remove`**, once the program has found the node to be

[2] Member function **`BST_threaded::end()`** is implemented.

deleted. Do not forget to compile the file **node.cpp** (use **Ctrl-Shift-F9**) to check for any compilation errors before you proceed.

14. Uncomment **PHASE 4** tests of **test1.cpp**. Compile your program, fix any errors, and run it. Compare the output with the contents of the file **test1_out.txt**.
15. You have now implemented all needed member functions of classes **BST_threaded**, **Node**, and **BiIterator**. As you noticed, you have also performed some simple tests to check that your code is likely to be correct.
16. Replace the file **test1.cpp** by the file **main.cpp**, in the project.
17. Add a program to the file **main.cpp** to display a frequency table with words occurring in a given text file. Your program should use the **BST_Threaded** class and the **BiIterator** class and have the functionally described below.
18. Compile your program and fix any errors. You can test it with the text files **mesg.txt** and **mesg1.txt**. The expected output is in the files **mesg_out.txt** and **mesg1_out.txt**, respectively.

Frequency table

After implementing and testing the classes above, you need to write a program that uses the **BST_Threaded** class and the **BiIterator** class to display a frequency table with words occurring in a given text file. More concretely, your program should perform the following steps by the order indicated below.

- Load the words in a given text file into a frequency table. This table is represented as a threaded binary search tree. Notice that the text file may contain words consisting of lower-case and upper-case letters, as well as numbers. Numbers are also considered words and should be stored in the table. It is also possible that punctuation symbols (e.g. '**?**', '**!**', '**,**') occur in the text, as usual. Obviously, punctuation symbols are not part of a word.
- Display the number of words occurring in the text and the number of different words.
- Display the frequency table of all words in the text, sorted alphabetically.
- Remove all words occurring only once in the text from the frequency table.
- Display again the number of words in the table and the frequency table.
- Request two words, **w1** and **w2**, to the user and then display all words in the interval **[w1,w2]**, in decreasing alphabetical order (only words occurring several times in the text). Note that the user given words may not exist in the text.

Questions to answer

Finally, write the answer to the following questions and deliver your written answers to the lab assistant, when you present this lab.

Indicate and motivate the time complexity of the operations **BiIterator::operator++** and **BiIterator::operator--**, in the best, average, and worst cases.

Lycka till!