" 'That's a great deal to make one word mean,' Alice said in a thoughtful tone.
" 'When I make a word do a lot of work like that,' said Humpty Dumpty, 'I always pay it extra.' "
Lewis Carroll, Through the Looking Glass, 1872

Peer design review **due**: see Canvas for due date; Code **Due**: see Canvas for due date.
So everyone benefits, you must ask all questions about this assignment during class, not in email.

Making Pointers do Extra Work

General requirements: You must follow the grading rubric and style checklist.

Note: Any sample or starter code provided by this website or the textbook is neither guaranteed nor ideal. Sample or starter code is intended to help you get started building your own code.

As with all work, you must ensure that it meets quality/efficiency/modularity/readability standards and requirements provided on the class website & instructions before you submit it.

You must analyze all instructions to ensure that you understand and meet all explicit and implicit requirements in the complete set of instructions, including any requirements that are enumerated or highlighted for you.

If you need clarification, you must ask questions, in a timely fashion, in class so everyone can hear the answers. If you email, wait until the weekend or the day before work is due, you will not get an answer.

You must deliver all the code needed to meet the requirements and solve the assigned problem, including any supporting files.
The grader must be able to simply run the g++ compiler on the main .cpp file you provide and have it compile and run. If what you deliver does not work under those circumstances, you have not met the essential requirements for acceptable work and should not expect to earn points for it.

Since a plain binary search tree with N nodes has N + 1 NULL pointers, half the space allocated in a binary search tree for pointer information is wasted. Suppose that if a node has a NULL left child, we make its left child pointer link to its inorder predecessor, and if a node has a NULL right child, we make its right child pointer link to its inorder successor. We call this a threaded binary search tree (tbst) and the extra links (that replace the NULL links) are called threads.

Building a Threaded Binary Search Tree

Start with the BST code from the Carrano textbook and augment it so that it correctly implements a modular, well designed and doxygen style documented ThreadedBST. You should do this based on the existing starter code for BinaryNode and BinarySearchTree classes in Carrano. You must properly design and correctly implement all of the public and private methods of the ThreadedBST (TBST) in a file called **tbst.cpp** (that you #include after the signatures in your **tbst.h**). You may use a KeyType of int during your draft, initial development process; but, you must implement a proper nodeData class (with operator<) for use in the tree in a file called **nodeData.cpp** (that you #include after the signatures in your **nodeData.h**) to complete the assignment. You must build a file named **lab5test.cpp** that fully tests each method in your **tbst.hb class and has no leaks when run with valgrind.**

Your code must use the thread links to make non-recursive (iterative) in-order traversals (that does not use internal or external stacks or arrays or any other form of tracking visited nodes). Your in-order traversal must use only the thread links and associated pointers. You also must create and test code to validate recursive pre-order, in-order and post-order traversals.

Partial notes on work you must do:

Design solution and plan tests: You must first use doxygen style comments to specify purpose and interfaces and non-c++ pseudo code to show your design for all classes and major modules. As part of your design documentation, you must specify unit tests for each module, with sample data and expected results. For example, how can you test to show that your default constructor or remove method for the tree work as expected? You can also use diagrams or drawings to clarify your design.

You **must submit a copy of your design online** on time and bring a printed copy of all design documentation to exchange with another pair for peer review. The design must cover all work required for this assignment and reflect clear, readable, modularity, cohesion and maintainability.

You must **submit a copy of your clear, specific feedback on the design you received online** and return the design with feedback to the authors by the end of the week you receive it.

Implement your initial TBST and Inorder methods in the TBST class as described above. Your test program must accept as the first command line argument: an integer, n (<= 52), specifying the number of nodes to create in your tree. If there is no integer command line argument, use a default value of 13. Your test program must then create a TBST and insert, in random order, pairs of numbers 1,2,…,n with the

corresponding letters A, B, ..., Z, …a, b, …, z into the TBST. (You should test with other sequences to verify proper behavior by your code.)

Your test code must then make a copy of the first tree and delete all the even numbers from the copy. Finally, it should use a non-recursive iterative inorder traversal of each tree, displaying (up to seven nodes of data per line) each node's contents with cout (just put a single space between each node's contents).

You may use and/or modify any code from the Carrano textbook for this assignment; but you must make it modular and ensure that it conforms to requirements and grading rubric/checklist.

Design and build a "token frequency" application using your TBST:

You must use your TBST in a file named **lab5main.cpp** to record, and display token frequency counts in any plain text document such as your program file or a copy of *Hamlet*. A token could be a word or punctuation or arithmetic operator. (Include apostrophe as part of a word, not as punctuation so that, for example, "OPHELIA " and " OPHELIA 's" both would get stored separately in your TBST. Do not store white space characters such as blank, tab or newline.)

You must complete, at least, the following tasks in **lab5main.cpp**:

1. Design and implement a node data class that can hold a token and an integer frequency pair
2. Print a brief introduction when your program starts listing authors and purpose of the application.
3. Display seven of the token-frequency pairs per line, **in increasing frequency order**, up to the 21 **most frequently** occurring tokens, followed by their frequency. On the next line, display seven of these token-frequency pairs per line, in increasing frequency order, up to the 21 **least frequently** occurring tokens, with their frequency
4. Display a one line message listing the file name and time required to process each file
5. Accept up to 50 file names on the command line to process. If there are no such file names, use "hamlet.txt" as a default file to process. Print a message to cerr if there are any file open errors then proceed to the next file if there are any others.
6. Unless the last argument on the command line [argc -1] is DEBUG, your code must not print anything but required output (to avoid this, use, for example, NOBUG as the last argument)
7. If the next to last argument on the command line [argc -2] is FILE, your code also must print (to a file named **lab5out.txt**) **all tokens, in alphabetical order**, followed (on the same line) by their frequency of occurrence in the document and must do so with seven of these token-frequency pairs per line. **Your tree must be sorted in alphabetical order with As to the left of Zs.**

You must answer the following questions in your preliminary written, design documentation (in other words, think about and research these things before you start on your design or implementation):

1. How can we distinguish threads from real child pointers?
2. What are the potential advantages of using threaded trees?
3. Does your implementation use any more memory than the "plain" BST implementation? If yes, can you think of any (obscure?) way to eliminate this extra memory requirement?
4. What is your implementation/test plan?
5. What steps must you go through to change the vanilla BST implementation to a Threaded BST?
6. Which BST methods need to be modified/overridden and which will work as is?
7. What are advantages and disadvantages of modifying the Carrano BST versus sub-classing from it?

You need not provide complete doxygen comments for each test function; but for **all other submitted work, use full file/class and function doxygen style comments.** See Appendix I, Listing 1-1 and http://www.doxygen.org for examples. At the top of each file submitted, **you must list the author(s) and describe its purpose**. You must precede each function/method implementation in the .cpp file by a complete doxygen style comment. Clearly state any assumptions you make in the appropriate comment block. For example, if you assume data is to be of a given format, document this as a pre-condition in functions that input data. You must use informative variable names in all signatures/prototypes and put signatures in .h files or above main(…).

What to turn in see Canvas for deadlines**:**

1. **Paper** copies of all **designs**, **code** with line and page numbersand **typescripts (see below)**
2. Online submitted copies of all code, test data and associated files
3. Online submitted Separate typescripts for each of lab5test.cpp and lab5main.cpp from Linux showing your compilation command and results and then results of valgrind to demonstrate complete execution (without DEBUG output) of each program and the absence of memory leaks.
4. The details of this assignment are subject to revision until Sunday after your designs are due.

In this order, for up to 10% extra credit: Implement and test a method to balance the TBST (see 16.4)

2nd, for up to 10% extra credit: Implement and test an iterative, in-order TBST **iterator** (see interlude 6).

---------- see answers to common questions below ----------

Answers to common questions:

You will find that you can base some (perhaps much) of your high-level and your module level design on material in the textbooks. After you start coding and running your tests, you may find you need considerable time to revise some (or many) of your designs, algorithms and code. Your design grade will depend on the completeness of the design and test cases as evidence of significant thinking about the problem, not on how perfect the design is. (Review Ch. 1, Appendix B and C.) To borrow a concept from D. Knuth, "The real goal is to formulate our programs [and design] in such a way that they are easily understood."

You may want to build a "draft implementation" ThreadedBST using just integer data to simplify your intial program. After that works, you can change to using the nodeData class. The integer version would be an intermediate step that will help you build your understanding. After you get that to work correctly, you must build a revised version using a more complex node to keep track of "tokens" (words, punctuation, etc) and counts. The revised version is the one to use for the final application. You should not submit your draft version.

You must develop your own code for this assignment. Use the text book pseudo code, .h and .cpp files as starting points. See a version of binarynode.cpp on the sample code page of our CSS342 website. See Ch. 16 student source code for a start on a BinaryNodeTree.cpp.

Design of a ThreadedBST retrieval function return depends on whether you are thinking of a public or private function. For the public version, you should determine the return type from your analysis of what the application needs. For the private version, you should determine the return type from your analysis of what you need to implement to support your design of the TBST as a tool.

The "KeyType" in this assignment refers to the data used to sort the tree. The examples in Ch. 16 refer to ItemType since that is the only data in the node. For the application problem, you have more than one piece of data in the node (the token is the key and the count), you must distinguish the two different types by using different <class *typeNames*> if you use templates.

You may produce an iterative in-order traversal without building a separate class iterator. A better design, for extra credit, entails adding an iterative traversal in an iterator as a separate method (or class). Interlude 6 discusses iterators.
Note: listing 16-3 and 16-4 refer to a parameter named "visit" in the public traversal sections. "visit" is the function to apply to each item as you traverse through the tree. Be sure to see the Programming tip on p. 467 about a reference parameter.