Anthony Natale

Summer 2020

Data Structures & Algorithms

# Programming Assignment 2

# Approach

Some citations included in this project have been added to the function description where applicable and to this document as well.

At a high level, we are taking a Binary Tree implementation and turning it into a Double (Left/Right) Threaded Binary Tree Implementation.

What does this mean? It means that we have a Binary Tree, or a K-ary tree where K = 2, and we want to be able to use pointers from each node to access the inorder predecessor and successor (in addition to using the same pointers to access the left and/or right children and/or parent node).

A standard pointer-based Binary Tree implementation uses two pointers that we will be reusing. These two pointers typically reference the left child and right child of the current node. We will be using these pointers in two capacities-as “regular” and “threaded”.

Each of our “threaded” pointers (those that point to an inorder predecessor or successor) must be marked as such using a boolean (which indicates the context of each pointer, so “true” means that the pointer is a thread to an inorder predecessor or successor and “false” means it is a regular pointer to a child node).

# Process

1. The first thing I am going to do is put all the existing code into a Visual Studio C++ console application project.
2. Next, I will create main.cpp and add it as a source file in Visual Studio
3. In main.cpp, I will inherit the appropriate file-“BST.h”-and write my main() function that will handle creation and manipulation of the BST object through the BST implementation.
   1. To get a hang of how data moves through the objects, I’ll create a BST object, add a few couts in various methods to see which are called and when, and compile/run the program.
4. Determining changes
   1. We’ll need to make changes to the following files
      1. BSTNode.h
      2. BST.h
      3. Main.cpp
   2. Let’s break this down
5. Changes to BSTNode.h
   1. Since BST.h implements the Binary Nodes from BinNodes.h, we will need to make sure there is support for threaded pointers and associated Booleans.
   2. This will NOT be done by adding more pointers. We will merely reuse the pointers for the left and right children that are already there, but label them as either “threaded” or “regular” pointers using a private Boolean (“true” if “threaded”, “false” if “regular”).
   3. By adding more pointers, we defeat the purpose of the exercise. The goal is to optimize the binary tree structure in which there are left and right child pointers, allowing for more efficient access to other parts of the tree (the inorder predecessors and successors) where possible when no left or right child node exists.
   4. So, create context Booleans indicating the type of pointer of the lc and rc pointers.
   5. Create getter and setter methods for these Booleans.
   6. Integrate these new variables into the constructor w/ parameters method.
   7. Amend isLeaf() to incorporate a check for the context variable when deciding if the node has no children (because now, all of them will have pointers that point to another node, either a threaded or regular one).
      1. If it has no children, it’s a leaf
      2. If all of its children are threaded, it’s a leaf
         1. Have to count number of children and get the “true” thread context variables for them
         2. If they’re equal, it’s a leaf
   8. Now, we must ensure that the methods in BST.h that call methods in our BSTNode class do so with the appropriate parameters (the booleans needed for thread/regular context).
6. Changes to BST.h
   1. Add private variables
      1. lowestKey
      2. highestKey
         1. These track the lowest inserted key value and highest inserted key value for some comparison operations.
   2. Add/change private functions
      1. simpleThread()
         1. Handle the first node-it will have slightly different threading requirements than the others.
      2. standardThread()
         1. Handle applying threads to all non-root nodes.
         2. This is the bread-and-butter function
            1. It divides work between most of the other functions and they return lots of pointers to it so that it can assign the threads as needed.
         3. Implementation
            1. Logic:

Get a reference to the newly inserted node

Handle special cases where the new node is the left-most or right-most node.

Use null pointers on the outer side of such a node

Otherwise, assign left threads to the inorder predecessor

Assign right threads to the inorder successor

Adjust the parent accordingly

Adjust the boolean context variables

* + 1. InsertHelp()
       1. Checks the status of the “root” object being passed in.
          1. If the object is empty, create a new node (empty tree)
          2. If the object is not empty
          3. If the key of the object to be inserted is less than the root’s key

Take a left turn and run the function again, resetting root to be the “current” node.

* + - * 1. If the key of the object to be inserted is greater than the root’s key

Take a right turn and run the function again, resetting root to be the “current” node.

* + - 1. Eventually, the final turn will be accomplished and the new node will be inserted in the appropriate spot.
    1. findPredecessor()
       1. Finds the inorder predecessor of a newly inserted node and returns a pointer to it
       2. Citation-used this for logic flow
          1. Initially used this in a joint function to return and set both the inorder predecessor and successor, but decided to split them up.
          2. <https://www.geeksforgeeks.org/inorder-predecessor-successor-given-key-bst/>
    2. findSuccessor
       1. Finds the inorder successor of a newly inserted node and returns a pointer to it
       2. Citation--used this for logic flow
          1. Initially used this in a joint function to return and set both the inorder predecessor and successor, but decided to split them up.
          2. <https://www.geeksforgeeks.org/inorder-predecessor-successor-given-key-bst/>
    3. getNode()
       1. Looks for a node by key value in the tree and returns a pointer to it
    4. getParent()
       1. Looks for a node’s parent and returns a pointer to it
    5. getLowestParent()
       1. Function used to handle a special case-where attempting to get the parent of the lowest key value node
    6. getHighestParent()
       1. Function used to handle a special case-where attempting to get the parent of the highest key value node
    7. printInorderHelp()
       1. Implements the printInorder() function
    8. printReverseHelp()
       1. Implements the printReverse() function
    9. leftMostChildInRightSubtree()
       1. Gets a pointer to the node.
    10. rightMostChildInLeftSubtree()
        1. Gets a pointer to the node.

1. Changes to Main.cpp
   1. Main.cpp must be able to implement all the functions in BST.h and print the appropriate information to the user.
      1. Standard key printing with printHelp()
      2. Print Inorder of element
      3. Print Reverse of element

# Integrity Statements

1. I have not shared the source code in my program with anyone other than my instructor’s approved human sources.
2. I have not used source code obtained from another student, or any other unauthorized source, either modified or unmodified.
3. If any source code or documentation used in my program was obtained from another source, such as a text book or course notes, that has been clearly noted with a proper citation in the comments of my program.
4. I have not knowingly designed this program in such a way as to defeat or interfere with the normal operation of any machine it is graded on or to produce apparently correct results when in fact it does not.