Project 4: Self Organizing Binary Search Trees

Due: 04/06/2017

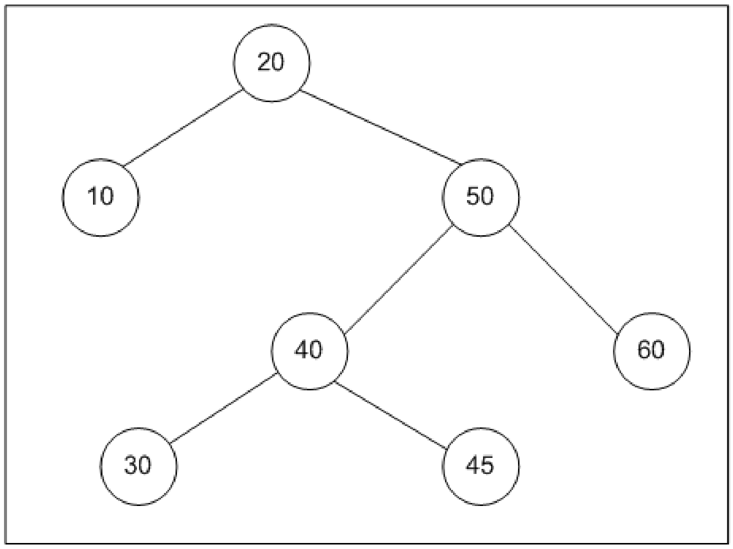
Educational Objectives: Understanding and developing self organizing binary search trees, understanding and developing a number of tree traversal algorithms, mastering the development of recursive algorithms. Analysis of algorithm complexity.

Statement of Work: Implement a generic self organizing binary search tree, which supports element insertion, deletion, search (and self-restructuring), and in-order and level-order traversals. Analyze the time complexity of one of the member functions of the binary search tree.

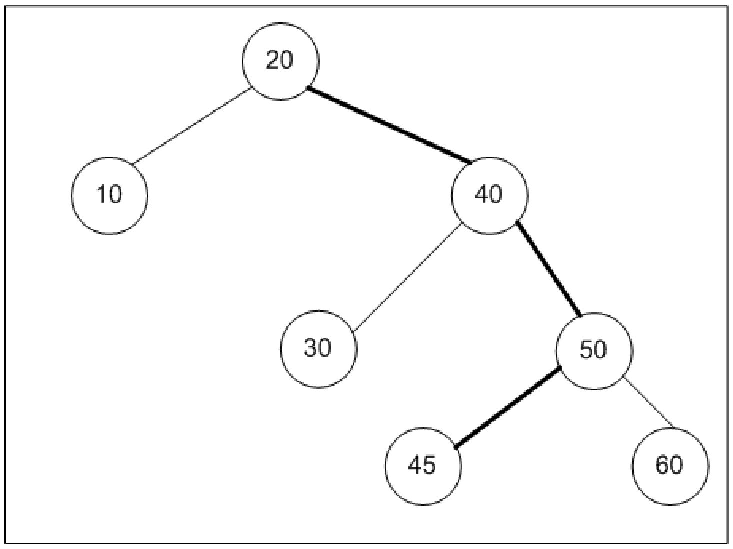
Project Requirements:

In this project you are asked to develop a generic self organizing binary search tree. A self organizing binary search tree can restructure itself depending on the search frequency of the elements in the tree. In particular, a threshold value is maintained by the tree and a search count is maintained by each node in the tree. The search count is increased by 1 each time there is a search of the value stored in the corresponding node. When the search count reaches the threshold, the corresponding node will move up in the tree by one level, which is achieved by a single rotation: rotate the position of the current node with that of the corresponding parent node. Let t denote the node whose search count is equal to the threshold, and p the parent node. First, let us assume that t is the left child of the parent node p. Then the rotation occurs as follows: 1) node t will occupy the position of p (that is, the parent of p will point to t instead of p); 2) the right child of node t becomes the left child of p; and 3) node p becomes the right child of node t. The following figure shows the single rotation. The case that t is the right child of the parent node p is analogous to the one we just discussed: 1) node t will occupy the position of p (that is, the parent of p will point to t instead of p); 2) the left child of node t becomes the right child of p; and 3) node p becomes the left child of node t.

After the single rotation, search count of node t is re-set to 0. When a node is already at the root (top) of the tree, no rotation should be performed even if the search count equals the specified threshold. For this case, you also need to re-set the search count to 0.



(a) Before rotation



(b) After rotation (assuming 40 is the one whose search count equals threshold)

Name your self organizing binary search tree class template as "BST". Your BST must have a nested structure named "BSTNode" to contain the node related information (including, e.g., element and pointers to children nodes). In addition, BST must at least support the following interfaces (in the following T is the abstract type of the elements stored in a node in BST).

Public interfaces

- **BST(int th=default_threshold_value)**: Constructor. Parameter th specifies the value for the single-rotation threshold. The default value default_threshold_value should be 1.

- **BST(const string input, int th=default_threshold_value)**: Constructor. The first parameter is a string containing the values (separated by spaces) to be inserted into the BST. An example could be string "1 23 34 30" for an integer-valued BST, which indicates to insert integers 1, 23, 34, and 30 into the tree in that order. The second parameter specifies the value for the single-rotation threshold. The default value default_threshold_value should be 1.
- **BST(const BST&)**: copy constructor. You need to copy both the element and the corresponding search count in each node.
- **BST(BST&&)**: move constructor.
- **~BST()**: destructor.
- **void buildFromInputString(const string input)**: parameter "input" is string containing the values to be inserted to the tree (separated by spaces), as we have discussed above. The tree should be built based on the input string. If the tree contains nodes before the function is called, you need to first delete all the existing nodes.
- **const BST & operator= (const BST &)**: copy assignment operator. You need to copy both the element and the corresponding search count in each node.
- **const BST & operator=(BST &&)**: move assignment operator.
- **bool empty()**: return true if the tree is empty. Return false otherwise.

  The following public interfaces will call the corresponding private versions of the functions to perform certain tasks:

- **void printInOrder() const**: print out the elements in the tree in the in-order traversal.
- **void printLevelOrder() const**: print out the elements in the tree in the level-order traversal.
- **int numOfNodes() const**: return the number of nodes in the tree.
- **int height() const**: return the height of the tree.
- **void makeEmpty()**: delete all nodes from the tree (make the tree empty)
- **void insert(const T& v)**: insert v into the tree.
- **void insert(T &&v)**: insert v into the tree (move instead of copy)
- **void remove(const T& v)**: delete value v from the tree.
- **bool contains(const T& v)**: search to determine if v is the tree.
-

  Private interfaces. **1) All the required private member functions must be implemented recursively except the function printLevelOrder(BSTNode *t). 2) No static variables or global variables can be used in implementing these recursive functions (zero point for the corresponding function if it uses static or global variables). 3) For some of the private member functions, we intentionally leave out the parameters. You need to decide if you need to have additional parameters for each such private member functions.**

- **void printInOrder(BSTNode *t) const**: print the elements in the subtree rooted at t in the in-order traversal.
- **void printLevelOrder(BSTNode *t) const**: print the elements in the subtree rooted at t in the level-order traversal.
- **void makeEmpty(BSTNode* &t)**: delete all nodes in the subtree rooted at t. Called by functions such as the destructor.
- **void insert(const T& v, BSTNode *&t)**: insert v into the subtree rooted at t. No duplicated elements are allowed. If value v is already in the subtree, this function does nothing. Initialize the search count to 0 for the newly added node.
- **void insert(T&& v, BSTNode *&t)**: insert v into the subtree rooted at t (move instead of copy). No duplicated elements are allowed. If value v is already in the subtree, this function does nothing. Initialize the search count to 0 for the newly added node.
- **void remove(const T& v, BSTNode *&t)**: remove value v from the subtree rooted at t (if it is in the subtree). If the node x containing v has two child nodes, replace the value of node x with the smallest value in the right subtree of the node x.
- **bool contains(const T& v, BSTNode *&t, ...other parameters if necessary...):** return true if v is in the subtree rooted at t; otherwise, return false. Note that the search count of the corresponding node containing v should be increased by 1. If search count reaches the threshold, perform the single rotation discussed above. Reset search count to 0. If the node containing the value v is already the root of the tree, do not rotate (but you do need to reset the search count to 0). Note that you can add other parameters if necessary. You may or may need to add additional parameters depending on your design and implementation of BST.
- **int numOfNodes(BSTNode *t) const**: return the number of nodes in the subtree rooted at t.
- **int height(BSTNode *t) const**: return the height of the subtree rooted at t. Recall that the height of a tree is the number of the links along the longest path from the root to any of the leaf nodes.
- **BSTNode * clone(BSTNode *t) const**: clone all nodes in the subtree rooted at t. Called by functions such as the assignment operator=. Note that you also need to copy the search count in the original node to the corresponding cloned node.

Other Requirements:

- Analyze the worst-case time complexity of the private member function **height(BSTNode *t)** of the binary search tree. Give the complexity in the form of Big-O. Your analysis can be informal; however, it must be clearly understandable by others. Name the file containing the complexity analysis as "analysis.txt".
- You can use any C++/STL containers and algorithms
- If you need to use any containers, you must use the ones provided in C++/STL. You cannot use the ones you developed in the previous projects.

Provided Code

The tar file contains the following files:

1. proj4_driver.cpp: the driver program to test your implementation of the self organizing binary search tree.
2. proj4.x: executable code (compiled on linprog.cs.fsu.edu)
3. proj4.input: sample input file. You can redirect the standard input of proj4.x to this file, or you can just type line by line when the program asks for input.

Deliverable Requirements

Turn in the tar file containing all c++ header files, source files, makefile, and analysis.txt via the blackboard system.

Notes:

- Note: The first person to find a programming error in our provided program will get a bonus point! There is no known error in the provided program.
- You should thoroughly test your program in addition to the simple sample input file proj4.input. We will test your program using other test files in addition to proj4.input file.