COP 5536

ADVANCED DATA STRUCTURES

**Spring 2016**

PROGRAMMING PROJECT REPORT

By:

Arvindh Seshadhri

UFID: 49930009

Email: aseshadhri@ufl.edu

TABLE OF CONTENTS

|  |  |
| --- | --- |
| Files, Classes and Methods | 1 |
| Structure of the program | 3 |
| Compiler and Environment details | 4 |
| Results and Observations | 5 |
| Conclusion | 6 |
| References | 6 |

FILES, CLASSES AND METHODS

File – bbst.java

Class – bbst.class

* Class – RBTree: The class RBTree is used to create the tree nodes initially in the balanced binary search tree, which is then converted to red black tree. Every node will have a parent node, a left child and a right child, which are initialized as null when a node is created. The node color is given a Boolean value where red is true and black is false. Every node also holds the id and count values as required. The constructor RBTree (int, int) assigns these values to a node.
* newbst (int [], int [], int, int, int): The method Newbst builds the required red black tree by first building a balanced binary search tree recursively. The level i.e. height of the tree is used to color the last level of the tree to red.
* TreeMinimum(RBTree): TreeMinimumtakes the root node of a tree or subtree as an argument and searches for the node with the minimum id value and returns it.
* TreeMaximum (RBTree): TreeMaximum takes the root node of a tree or subtree as an argument and searches for the node with the maximum id value and returns it.
* Previous\_key (RBTree): This function returns the predecessor node of a given node and returns it.
* previous (int): This function takes id value of a node. It searches for the node initially. If the node is found, it returns the node, else it returns the node with the id value closest to the given id.
* search (RBTree, int): The search functions takes root node and id value to be searched as arguments. It returns the node closest in id value to the id provided. If the root node is null, then null value is returned.
* Next\_key (RBTree): The function returns the successor node of a given node and returns it.
* next (int): This function takes id value of a node and searches for the node using search(RBTree, int) function. If the node is found, it returns the node, else it returns the node with id value closest to the given id.
* inrange (RBTree, int, int): The inrange function implements the requirement of returning total count of nodes, which has value lying between two nodes inclusive. The node passed to the function as argument is the root node.
* count (int): The count function implements the required functionality where, when given an id value of a node, its count needs to be returned.
* Rotate\_left (RBTree): The Rotate\_left function takes care of the left rotations required to be performed after insertions or deletions are done on a red black tree.
* Rotate\_right (RBTree): The Rotate\_right function takes care of the right rotations to be performed after insertions or deletions are done on a red black tree.
* Transplant (RBTree, RBTree): This function implements the changes necessary to maintain the red black tree properties after deletion of node.
* deleteNode (RBTree): When the count value of a node after reduction goes to less than or equal to 0, the node is deleted. deleteNode function takes care of this functionality. The initial deleting of node is done by this function and then the changes required to maintain the red and black tree properties are then done by deleteFixer function.
* deletefix (RBTree): The deleteFixer function takes care of the necessary changes to the tree to maintain the red black properties after deletion.
* reduce (int, int): The reduce function takes care of the required reduce count value of a node functionality. If the count value after reduction is a positive integer, the count value is returned. Else, the node is deleted and count value is returned as 0. deleteNode function is called when the count value goes to 0 or less after reduction.
* insertNode (int, int): When the id of a node whose count is to be increased is not found, a node with the given id and count are inserted. insertNode function takes care of this functionality. The initial inserting of node is done by this function and then the changes required to maintain the red black tree properties are then done by Insert\_Fix function.
* Insertfix (RBTree): The Insert\_Fix function takes care of the necessary changes to the tree to maintain the red black tree properties after insertion.
* increase (int, int): The increase function takes care of the required increase count value functionality. If the node whose count is to be increase is not found, the node is inserted with its id as the search value given and its count as the increase amount given. insertNode function is called when the id value of node to be increased is not found.
* main(): The main function is where we take the input from the file, commands from the second file and write output to a third file.

STRUCTURE OF PROGRAM

Once the makefile is invoked, and the java file gets compiled, while implementing the code, filename is taken as an argument which contains all the node ids and values. After that, a file with commands is needed and a destination file to save the output to.

Every line of the file is stored in a string and the string is then split into the two numbers, which are stored in the arrays. As the input of a file is finished, function to create the Balanced Binary Search Tree is called.

The function creates a balanced binary search tree recursively. This tree is then converted to a red black tree by coloring all the nodes at height 0 as red

Once the red black tree is created, the commands from commands.txt file or manual commands are extracted one by one and the respective functions are called based on the first word in the command.

COMPILER AND ENVIRONMENT DETAILS

The project has been compiled and tested under the following platforms:

Platform/Operating System Compiler Test Result

Macintosh javac Pass

Linux javac Pass

Steps to execute the project:

1. Put the java and makefile in a single folder
2. Go to the directory and invoke the makefile using ‘make’
3. The program can then be run using: java bbst test\_file\_name < commands\_file> output\_file

RESULTS AND OBSERVATIONS

After compiling the code and implementing it using the test files, the program showed desired results when compared with the output files provided. The running time of algorithm did not increase exponentially with exponential increase in the number of nodes. Test files with 10^6 nodes could run in less than 3 seconds and files with 10^7 nodes took less than 12 seconds. The program also runs for the other given files with 10^2 and 10^8 nodes. The file with 10^8 number of nodes required increase in heap size with command as –Xmx8000m etc. to run without giving memory errors in Java. The file with 10^8 nodes took about 100 seconds to run in the local machine.

CONCLUSION

A more efficient way of creating the Red Black Tree is by first making a balanced binary search tree and then making it red black.

The construction of balanced binary search tree is done in O (n) time. The functions using search algorithms to find the nodes and increase, decrease counts, finding the next and previous nodes of given values, all take O (log n) time to run. The inrange function which gives total of count from one id value to another takes O (log n + s) time as required.

REFERENCES

1. Cormen, Thomas H., Charles Eric. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. Third ed. Cambridge, MA: MIT Press, 2009.