

Advanced Data Structures

Implementation of Trees and comparison of running times

COP 5536 - Fall 2012

Name: Shravan Pentamsetty

UF ID: 9862-5845

Email: shravansetty@ufl.edu

TABLE OF CONTENTS

| | |
|--|-----------|
| 1. FUNCTION PROTOTYPES | 2 |
| 2. EXPECTATION BEFORE RUNNING PROGRAM..... | 7 |
| 3. RESULTS..... | 8 |
| 3.1 AVL TREES..... | 8 |
| 3.2 RED BLACK TREES..... | 8 |
| 3.3 B TREES | 9 |
| 3.4 OBSERVATIONS | 9 |
| 4. RESULT COMPARISION..... | 10 |
| 4.1 INSERT..... | 10 |
| 4.2 SEARCH | 10 |
| 4.3 GRAPHICAL REPRESENTATION OF COMPARISION | 11 |
| 5. RECOMMENDATIONS BASED ON MY EXPERIMENTS..... | 11 |
| 6. COMPILING INSTRUCTIONS | 12 |
| 6.1 RANDOM MODE | 12 |
| 6.2 USER INPUT MODE | 13 |
| 6.3 OUPUT FORMAT | 13 |

1. FUNCTION PROTOTYPES

1.1 DICTIONARY

```
dictionary.java
import java.io.*;

// Author: Shrayan Pentamsetty
// UF Id: 9862-5845
// ADS Project Fall 2012.
/* The purpose of this project is to compare the relative performance of
 * AVL, red-black, and B-trees as well as their counterparts with a hash table front end.
 * We shall focus only on the search and insert operations and require all keys to be distinct.
 * */
// This File contains main method. Entry point when the program is executed
public class dictionary {
    // Creating variables required to read the file
    static ArrayList<Integer> KeyFile = new ArrayList<Integer>();
    static ArrayList<Integer> ElementFile = new ArrayList<Integer>();
    static File f1;
    static File f2;
    static File f3;
    static File f4;
    static File f5;
    static File f6;
    static BufferedWriter bw;

    // Read the Keys and Elements in a file and store the values in an Array
    // List
    private static void readfile(String filename) throws NumberFormatException, {}

    // Create the necessary files when User Input mode is selected
    private static void createfiles() {}

    // Main Program
    public static void main(String[] args) throws NumberFormatException, {}
}
```

Dictionary.java is the class file, which contains the main method. This program is divided into two parts, based on the input arguments. The first part is the random mode, in which the hash table size and tree order are passed as arguments. The trees of each of the six types are created; inserts and search operations are performed for every tree using a random permutation generator. The corresponding running times of each operation are given as output in the console. The second part is user input mode, in which the input file name is given as an argument. AVL, B trees are created. The key element pairs are read from the input file and are inserted into the trees created. The output files are created using the method “Create files” and the corresponding values are written to files.

1.2 PERMUTATION GENERATOR

```
PG.java
import java.util.ArrayList;

//Permutation Generator Class
public class PG {
    int size;
    // Array
    ArrayList<Integer> A = new ArrayList<Integer>();
    // Random Generated Array
    ArrayList<Integer> RGA = new ArrayList<Integer>();

    public PG(int n) {}
}
```

Permutation Generator Class takes the input value as size of the random generated array(RGA) required.

1.3 AVL TREE - NODE

```
Node.java
public class Node {
    int balfac;
    Node left;
    Node right;
    int value;
    int element;

    public Node(int key, int element1) {}

    public Node() {}
}
```

Node Class is used to create nodes in AVL Trees. This takes the input as the Key and Element values. Balance factor, Left node and Right node are other instance variables in the class

1.4 AVL TREE

```
*AVLTree.java
import java.io.BufferedWriter;
public class AVLTree {
    Node root;
    // Default Constructor
    public AVLTree() {}
    // Method to Insert a Node into the AVL Tree
    public void insert(int key, int element1) {}
    // Method to find A,B,C Nodes when the imbalance is found
    public void find_fix(Node temp, Node nn, Node Parent) {}

    // Method to determine the type of imbalance and call the appropriate rotations
    public void rotate_main(Node A, Node B, Node C, Node Parent) {}

    // LL Rotate Method
    public void LL(Node A, Node B, Node C, Node Parent) {}

    // RR Rotate Method
    public void RR(Node A, Node B, Node C, Node Parent) {}

    // LR Rotate Method
    public void LR(Node A, Node B, Node C, Node Parent) {}

    // RL Rotate Method
    public void RL(Node A, Node B, Node C, Node Parent) {}

    // Methods to print the In-Order walk
    public void inorder_walk(Node root) {}
    public void inorder_walk(Node root, BufferedWriter bw) {}

    // Methods to print the Post-Order walk
    public void postorder_walk(Node root) {}
    public void postorder_walk(Node root, BufferedWriter bw) {}

    // Method to search a key in the AVL Tree
    public void search(int i) {}
    public void search(int key, Node a) {}
}
```

AVLTree Class contains methods to insert the key - element pairs into the tree, find the imbalance type, and fix the imbalance by using rotations. It also has the standard tree methods, namely inorder walk, postorder walk and searching a key.

1.5 AVL TREE WITH HASH FRONT-END

```
AVLHash.java
import java.io.IOException;

public class AVLHash {
    int sizeofhash;
    AVLTree[] a;

    public AVLHash(int s) {}
    // Method to insert key element pairs in the buckets
    public void insert(int i, int j) throws IOException {}
    //Method to search a key in the buckets
    public void search(int i) {}
}
```

AVLHash class takes the input as hash size. It has the methods Insert and Search to insert key element pairs and search a key respectively in a tree.

1.6 RED BLACK TREE

Insert into and Search from Red Black Tree are done using the standard TreeMap class available in java.util package.

1.7 RED BLACK TREE WITH HASH FRONT-END

```
RedBlackHash.java
import java.util.TreeMap;

public class RedBlackHash {
    int sizeofhash;
    TreeMap[] a;

    public RedBlackHash(int s) {}
    // Method to insert key element pairs in the buckets
    public void insert(int i, int j) {}
    //Method to search a key in the buckets
    public void search(int i) {}
}
```

RedBlackHash class is similar to AVL Hash except it stores the reference to a Red Black Tree instead of AVL Trees.

1.8 KEY ELEMENT PAIRS

```
1 KEP.java ✕
2
3 public class KEP {
4     int key;
5     int element;
6     //BTree Pointer;
7
8     public KEP(int key1, int element1) {
9         key = key1;
10        element = element1;
11    }
12 }
```

KEP Class takes key element pairs as input and creates an object of type KEP.

1.9 B-TREE NODE

```
1 BTreeNode.java ✕
2
3 import java.util.ArrayList;
4
5 public class BTreeNode {
6     static int btree_order;
7     BTreeNode[] pointers;
8     ArrayList<KEP> pairs;
9     boolean leaf;
10    int n;
11
12    // Constructor that takes input as keyelement pair and order
13    public BTreeNode(KEP k1, int order) {}
14    // Constructor that takes only order as input
15    public BTreeNode(int order) {}
16    // Method to create the arrays to store the pointers and KEP Objects
17    public void initArrays() {}
18 }
```

BTreeNode class is used to create a node in a Btree. Constructors in this class are overloaded to create a node by taking either order alone as input or both order and KEP object as input. A variable of type Boolean is created to identify if a node is leaf. BTree Node contains pointers and pairs according to the definition.

1.10 B-TREE

```
BTree.java
import java.io.BufferedWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Stack;

public class BTree {
    static int btree_order;
    static int node_count;
    static int level;
    BTreeNode root;

    // Constructor with no arguments. The order is hardcoded to 3 when default
    // constructor is called
    public BTree() {}

    // Constructor which takes order as input
    public BTree(int order) {}

    // Method to insert Key Value pair into
    public void insert(int key, int element) {}

    // Method to Split the node
    private void split(BTreeNode a, Stack<BTreeNode> SS) {}

    // Method to search the index of the nodes.
    private int get_index(BTreeNode x, KEP k) {}

    // Method to search a key in the tree
    public void Search(BTreeNode p, int key) {}

    // Method to print Sorted output
    public void walk_through(BTreeNode x, BufferedWriter bw) throws IOException {}

    // Method to print the Level Out Values
    public void level_order(BTreeNode root, BufferedWriter bw) {}
}
```

BTree class is used to create a Btree. It has methods to insert key element pairs, split the node when the node is full, search the index - where to insert a key and where to split, search a key in the tree.

1.11 B-TREE WITH HASH FRONT-END

```
BTreeHash.java
import java.io.IOException;

public class BTreeHash {

    int sizeofhash;
    BTree[] a;

    public BTreeHash(int s) {}
    // Method to insert the key element pairs into appropriate buckets
    public void insert(int i, int j) throws IOException {}
    // Method to search a key in appropriate bucket.
    public void search(int i) {}
}
```

BTreeHash class is similar to AVL Hash except it stores the reference to a B Tree instead of AVL Trees.

2. EXPECTATION BEFORE RUNNING PROGRAM

Time complexity of Insert and Search operations in balanced trees is $O(\text{height})$. Height of three trees are given below:

AVL Trees: $\log_2(n+1) \leq \text{height} \leq 1.44 \log_2(n+2)$

Red Black Trees: $\log_2(n+1) \leq \text{height} \leq 2\log_2(n+1)$

B Trees: $\text{height} \leq \log_{\text{ceil}(m/2)}[(n+1)/2] + 1$

Where 'n' is the number of internal nodes and 'm' is the order of B Tree.

Note:

- 1) Redblack height > avl height > btree height
- 2) With the hash table as the front-end, the number of trees would be less than or equal to the number of buckets.
- 3) Hash function is applied whenever a key element pair is inserted and stored in corresponding bucket. Hence the size of the tree is reduced unless all the key element pairs fall into the same bucket, which never happens. Hence, the performance of Insert is increased.
- 4) To search a key, it is easier to apply the hash function; go to the bucket and find the key in the tree, which has less height. Hence, the performance of Search is increased.
- 5) For a fixed number of nodes, increasing the order of B-Tree will decrease the height of the tree, which increases the performance of both inserts and searches.

With the help of the above notes, we can formulate the following expected results.

INSERT

Running time of the trees should be as follows:

$RT_{\text{redblack_insert}} > RT_{\text{avl_insert}} > RT_{\text{btree_insert}}$

SEARCH

For Search operation, running time of the trees should be as follows:

$RT_{\text{redblack_search}} > RT_{\text{avl_search}} > RT_{\text{btree_search}}$

INSERTS AND SEARCH WITH HASH TABLE AS FRONT-END

Running time of the trees should be as follows:

| | |
|---|---|
| $RT_{\text{redblack_insert}} > RT_{\text{redblackhash_insert}}$ | $RT_{\text{redblack_search}} > RT_{\text{redblackhash_search}}$ |
| $RT_{\text{avl_insert}} > RT_{\text{avlhash_insert}}$ | $RT_{\text{avl_search}} > RT_{\text{avlhash_search}}$ |
| $RT_{\text{btree_insert}} > RT_{\text{btreehash_insert}}$ | $RT_{\text{btree_insert}} > RT_{\text{btreehash_insert}}$ |

3. RESULTS

A random permutation generator is used to generate an array of numbers, which takes the input size as an argument. Each number is inserted into the trees using the respective insert methods in the respective tree class. Time for Insertion is calculated. Once they are inserted into the tree, Search operation is performed for each key in the inserted order. Time for Search is calculated.

Following results are provided for 'n' = 1,00,000. The times shown are the average times of 10 runs.

3.1 AVL TREES

Average time taken for Insertion: 1157 milliseconds

Average time taken for Search: 622 milliseconds

AVL TREES WITH HASH

| S (Size of Hash table) | Avg. Time taken for Insert (milliseconds) | Avg. Time taken for Search (milliseconds) |
|------------------------|---|---|
| 3 | 1156 | 633 |
| 11 | 1134 | 614 |
| 101 | 1053 | 549 |

3.2 RED BLACK TREES

Average time taken for Insertion: 1734 milliseconds

Average time taken for Search: 928 milliseconds

RED BLACK WITH HASH

| S (Size of Hash table) | Avg. Time taken for Insert (milliseconds) | Avg. Time taken for Search (milliseconds) |
|------------------------|---|---|
| 3 | 1724 | 1821 |
| 11 | 1903 | 1661 |
| 101 | 1531 | 2148 |

3.3 B TREES

| B-Tree Order | Avg. Time taken for Insert (milliseconds) | Avg. Time taken for Search (milliseconds) |
|--------------|---|---|
| 35 | 3655 | 1678 |
| 45 | 3346 | 1669 |
| 50 | 3107 | 1779 |
| 55 | 3045 | 1554 |
| 58 | 3567 | 2007 |

Optimal BTree Order is 55.

B TREES WITH HASH

| Order of the Tree | S (Size of Hash table) | Avg. Time taken for Insert (milliseconds) | Avg. Time taken for Search (milliseconds) |
|-------------------|------------------------|---|---|
| 55 | 3 | 11376 | 3442 |
| | 11 | 7427 | 3264 |
| | 101 | 3776 | 3120 |

3.4 OBSERVATIONS

- ✚ If there are more lookups to be performed, it is advisable to use AVL trees, as the height is less.
- ✚ If more Insert operations are to be performed, it is advisable to use RedBlack trees, as the numbers of rotations are less.
- ✚ Hash table front end provides an improvement in performance.
- ✚ Increasing the hash table size gives better performance.
- ✚ B-Trees are efficient when the search trees are stored on disk. When the trees are stored in memory as in this project, AVL trees and red black trees will perform better than b-trees.
- ✚ The Optimal order found is not the appropriate one, because the input order is different for every run, as the keys and elements are inserted from the Random Generator Array.

4. RESULT COMPARISION

4.1 INSERT

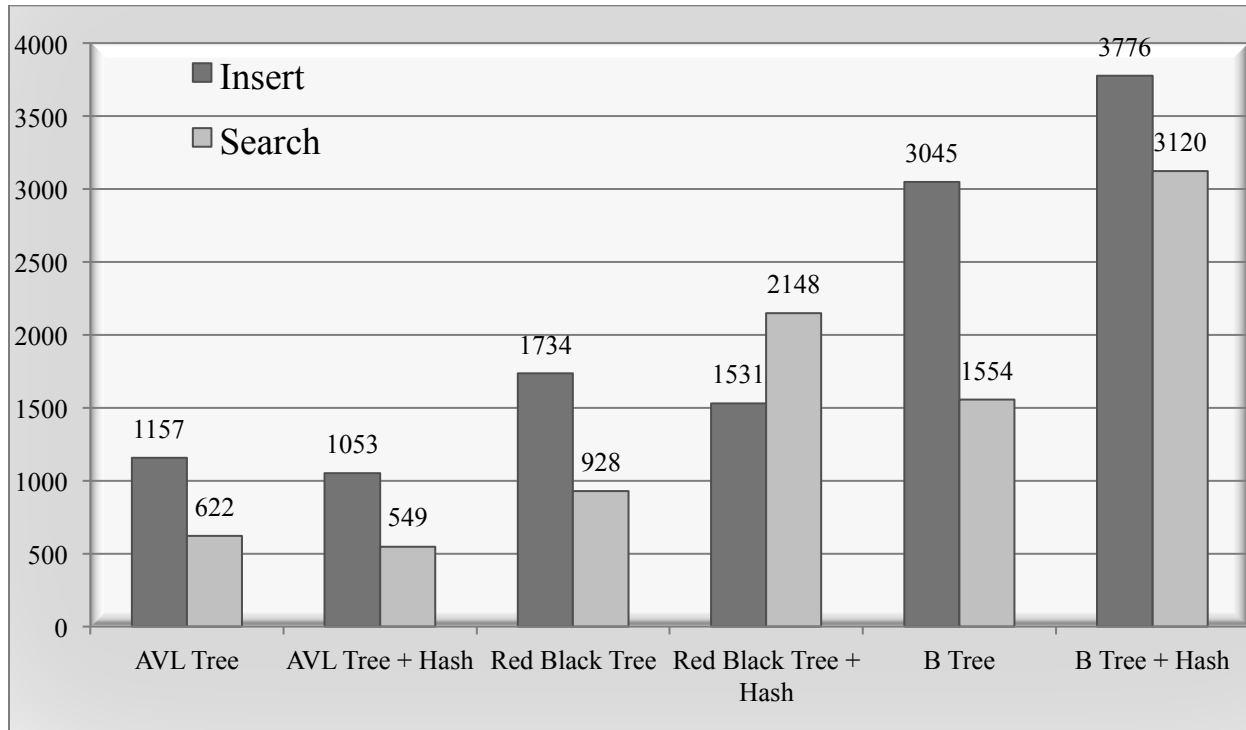
| TREE STRUCTURE | Time taken for the Insertion (milliseconds) |
|------------------------------------|---|
| AVL Tree | 1157 |
| AVL Tree with Hash front-end | 1053 |
| Red Black Tree | 1734 |
| Red Black Tree with Hash front-end | 1531 |
| B Tree | 3045 |
| B Tree with Hash front-end | 3776 |

4.2 SEARCH

| TREE STRUCTURE | Time taken for the Insertion (milliseconds) |
|------------------------------------|---|
| AVL Tree | 622 |
| AVL Tree with Hash front-end | 549 |
| Red Black Tree | 928 |
| Red Black Tree with Hash front-end | 2148 |
| B Tree | 1554 |
| B Tree with Hash front-end | 3120 |

4.3 GRAPHICAL REPRESENTATION OF COMPARISON

Given below is the chart that compares the executing times of Inserts and Deletes of 10,00,000 numbers.



5. RECOMMENDATIONS BASED ON MY EXPERIMENTS

WORST CASE PERFORMANCE IS IMPORTANT

BTree would be a good way to implement, in an environment, where worst-case performance is important.

EXPECTED PERFORMANCE IS IMPORTANT

AVL Trees provided optimum performance in most of the cases. And I would recommend using AVL Trees (both with or without hash), Red Black Trees if expected performance is of interest.

NEAREST MATCH SEARCHES

Hash based structures cannot address nearest match queries, hence the trees should be used and hash table front-end is not advisable. It would a good idea to use AVL trees as the insert and search times are better compared to others.

6. COMPILING INSTRUCTIONS

The class dictionary.java is the file, which has Main method, is the starting point in the program. I have used the JDK Version “1.6.0_35” to create class files.

```
JohnCena-2:bin shravansetty$ java -version
java version "1.6.0_35"
Java(TM) SE Runtime Environment (build 1.6.0_35-b10-428-11M3811)
Java HotSpot(TM) 64-Bit Server VM (build 20.10-b01-428, mixed mode)
JohnCena-2:bin shravansetty$
```

To compile the program, place all java files in a folder and run this cmd: **\$ javac dictionary.java**. The program can be executed in following two modes:

1. Random Mode
2. User Input Mode

6.1 RANDOM MODE

To execute the program in random mode, the arguments should be in the following format:

\$ java dictionary -r <hashtablesize> <btreeorder>

-r: indicates that the program is being executed in random mode.

<hashtablesize>: the size of the hashtable that you want to create.

<btreeorder>: order of btree

When run, program asks to input the number of elements you would like to insert. Enter the number and hit “Enter” key. The output is printed on to the stdout screen.

Example:

```
Console [X]
<terminated> dictionary [Java Application] /System/Library/Java/JavaV
Random Mode
*****
Enter the number of elements: 12345
*****
Insertion into AVL Tree took 50 milliseconds
Search from AVL Tree took 6 milliseconds
*****
Insertion into AVLHash took 30 milliseconds
Search from AVL hash took 34 milliseconds
*****
Insertion into RedBlack tree took 18 milliseconds
Search from RedBlack Tree took 7 milliseconds
*****
Insertion into RedBlack Hash took 38 milliseconds
Search from RedBlack Hash took 44 milliseconds
*****
Insertion into BTree took 47 milliseconds
Search from B-Tree took 33 milliseconds
*****
Insertion into BTreeHash took 84 milliseconds
Search from BTreeHash took 9 milliseconds
```

6.2 USER INPUT MODE

To execute the program in random mode, the arguments should be in the following format:

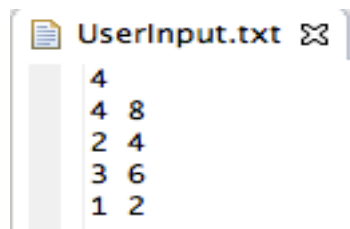
\$ java dictionary -u <inputfilename>

-u: indicates that the program is being executed in user input mode.

<inputfilename>: the name of input file which contains the key element pairs.

Input file should be in the following format: Number of lines followed by the key element pairs separated by a space.

Sample User Input: First line indicates the number of keys present in the file. Second line contains the Key(1st number) and Element(2nd number) separated by a space.



```
UserInput.txt
4
4 8
2 4
3 6
1 2
```

Example:

```
JohnCena-2:bin shravansetty$ java dictionary -u ~/Documents/Eclipse/ADS/UserInput.txt
User Input Mode: File Name:: /Users/shravansetty/Documents/Eclipse/ADS/UserInput.txt
Program Executed and the files are created in the current directory
JohnCena-2:bin shravansetty$
```

After the program is executed, the output files are created in the current directory and the output is written to corresponding files.

6.3 OUPUT FORMAT

Note: Element values are written to the respective files. Key values are not written for the file.

- 1) **“AVL_inorder.out”** - Inorder of the AVL Tree is printed with a space between each element value.
- 2) **“AVL_postorder.out”** - Postorder of the AVL Tree is printed with a space between each element value.
- 3) **“AVLHash_inorder.out”** - Inorder of the AVL Trees are printed with a space between each element value. Two new line characters separate each tree.
- 4) **“BTree_sorted.out”** – Element values in the tree are written in sorted output.
- 5) **“BTree_level.out”** - Element values are written from level 0 – level n. Within level, the values are separated by a space. A new line character separates each Level.
- 6) **“BTreeHash_level.out”** - Element values are written from level 0 – level n. Within level, the values are separated by a space. A new line character separates each Level. Two new line characters separate each tree.