ADVANCED DATA STRUCTURES

COP 5536

PROGRAMMING PROJECT FALL 2012

Submitted By:

Nitin Agrahara Ravikumar

73714398

email: arnitin@ufl.edu

Introduction

This report serves the address the structure of the program and analysis of the data obtained through execution of the program with various parameters. The project has been written in C++ and has been compiled using the g++ compiler. A makefile has been provided with the source code. “make clean” will delete all the binary and .out files. “make” will compile the project. The compilation will generate a binary called ‘**dictionary’** which can be run.

Program Structure

The Program can be divided into three parts. One each for the AVL trees, Red Black trees and BTrees. As specified by the requirements, Read Black trees have been implemented using STL maps and do not have their own implementation in this project.

For the implementation of **AVL Trees** we have a class as shown below:

```
class node {
public:
        node * left;
        node * right;
        int     heightVal;
        int     key;
        int     val;

        node () {
                heightVal = 0;
                left = NULL;
                right = NULL;
        }

};
```

Each object has a left and right pointer, pointing to the node’s two children. The heightVal is used to store the height at the location. As the names states, the key and val are used to store the key and the value. The constructor makes sure that the variables are properly initialized.

An insert into a specified AVL Tree is achieved using the **insertAVL (node * &In, int key)** function. The structure of this function is as described below:

Check if **In** node is Null. If yes, allocate memory for it and store the key and val. If not, check if the key in is lesser than or greater than that the current node's key value. If less, then recursively call the same function on the left child else on the right child. In both the cases, as soon as the insert is done, the height difference between the two children is calculated and if it not 1 , rotations are done. Since the maximum height difference according to this logic is 2, we check for that value. The Rotations are managed by the method:

Void BalanceAVL (node * &In, int key, int type)

The type parameters tell us if the node was interested in the left or the right child. Based on this information, we decide which sub tree is heavier and appropriately perform a LR LL RR OR RL rotation to balance. The rotation functions are just two viz.

Void LeftRot (node *&In)

Void RightRot (node *&In)

The rotations are as stated. The pivot for the rotation is provided as the input and a right or left rotation is done on it. The pointers are swapped and the result is reflected in the main tree because we are passing pointer address. LL and RR are performed using a combination of the above two functions.

After balancing the height of that node is recalculated. Since we are using recursion, this will happen for all nodes in the path. The Search method is a simple recursive binary search implemented in the following function:

int searchAVL(int key, node* Tree)

Which return true if the search key was found and 0 if not. Other method involved with the AVL trees are the following two functions.

void inorder (node *&In, ofstream &file)

void postorder(node *&In, ofstream &file)

Which are also recursive in nature and write the values to the file provided as parameter. The AVLHash is just an array of AVL node pointers on whom the search and insert method employ the specified 'key mod s' calculation.

For the RB Trees, as mentioned an STL map is created and the map.insert() and map.find(key) methods are employed. The hash methodology is the same as described for AVL Hash but the map functions are used instead.

For the implementation of **BTree** we have a class as shown below:

```
class BTreeNode{
        public :
                int             currKeyCount;
                int *           keys;
                int *           vals;
                BTreeNode ** ptrs;
                BTreeNode ();
                BTreeNode (int val);

};

BTreeNode :: BTreeNode(int val){
        keys = new int[val - 1 ];
        vals = new int[val - 1 ];
        ptrs = new BTreeNode*[val];
}
```

Each object has a currKeyCount telling us how many keys it is currently holding. An int array for the keys and one more for the vals. A variable called ptrs which is an array of pointers to hold the addresses of all its children.

The size for all the above mentioned arrays is initialized with a constructor which specifies the degree of the BTree.

void BTreeInsert(int key, BTreeNode *&BTreeIn)

This function handles the insert into a BTree. The structure is as follows:

If input Tree is null, allocate memory for it and populate the values. It becomes the root. If not, then recursively go to the location where the key is to be inserted. This is done by using the help of the following function:

int location(int key, int * keys, int keyCount)

This function will return the location where a key has to be inserted. This can be easiy be found out because the keys are always in a non-decreasing order. A simple traversal gives us the location. Once that location is obtained, repeat the process to traverse as deep in as possible. When three, check if there aren't any keys. If not the make this also a key. If other keys exist, then check if a new key can be accommodated. If it is possible, the insert into the proper index with the help of the location function. Now, if it cannot accommodate, then we need to split the node. Now the split pos is calculated and all left to it and right to it are made children and the split key is moved up. Since recursion is being used,

again, the code will check if that can be accommodated or not. And follow the same procedure as described earlier.

All the above mentioned functionality happens in this method:

int whatAction(BTreeNode ** insertNode, int key, int * splitKey, BTreeNode * &curRoot)

It is initially called from the BTree insert method and the recursion ensues.

int BTreeSearch(int key, BTreeNode * Tree)

Is the method used to do a search in the BTree It is a non-recursive method and it traverses through the list of keys to find the proper location where a key should ideally be. If the location is not the key then moves a level deeper into the node pointed to by hat location and repeat.

Other methods involved with the BTree are

void BTreeLevel(BTreeNode * BTree, ofstream & file)

This method basically does a BFS on the BTree and prints the vals level wise into the file specified. BFS is achieved using a STL Queue.

void BTreeInorder(BTreeNode * In, ofstream & file)

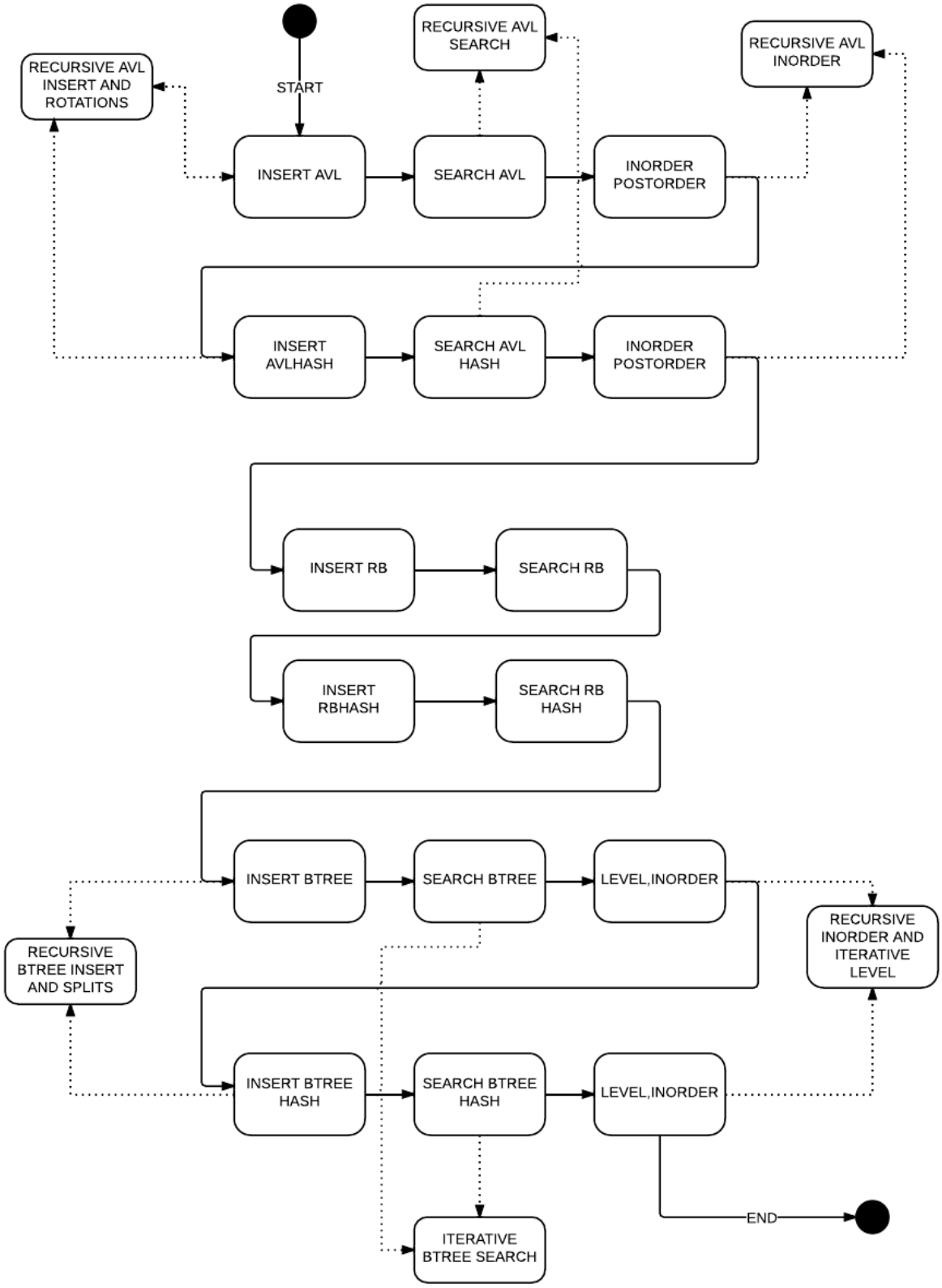
Is for inorder printing of vals into the file. This method does the equivalent of a DFS on the BTree to get the key inorder and write them to the file.

Main Method:

The main method sequentially calls all the methods. Initially, it checks the mode in which the program is being run. Checks if all the parameters have been provided and initializes the files pointers. Next, it either generates random numbers or reads them from a file. The numbers are stored in an array. This array is used for the insert and search methods.

The AVL insert method is called first and immediately, its search for all the elements inserted. The same is done for RB and BTrees.

The following diagram represents the gist of the program


START
RECURSIVE AVL INSERT AND ROTATIONS
RECURSIVE AVL SEARCH
RECURSIVE AVL INORDER
INSERT AVL
SEARCH AVL
INORDER POSTORDER
INSERT AVLHASH
SEARCH AVL HASH
INORDER POSTORDER
INSERT RB
SEARCH RB
INSERT RBHASH
SEARCH RB HASH
INSERT BTREE
SEARCH BTREE
LEVEL,INORDER
RECURSIVE BTREE INSERT AND SPLITS
RECURSIVE INORDER AND ITERATIVE LEVEL
INSERT BTREE HASH
SEARCH BTREE HASH
LEVEL,INORDER
END
ITERATIVE BTREE SEARCH

Observations and Results

Expected Results:

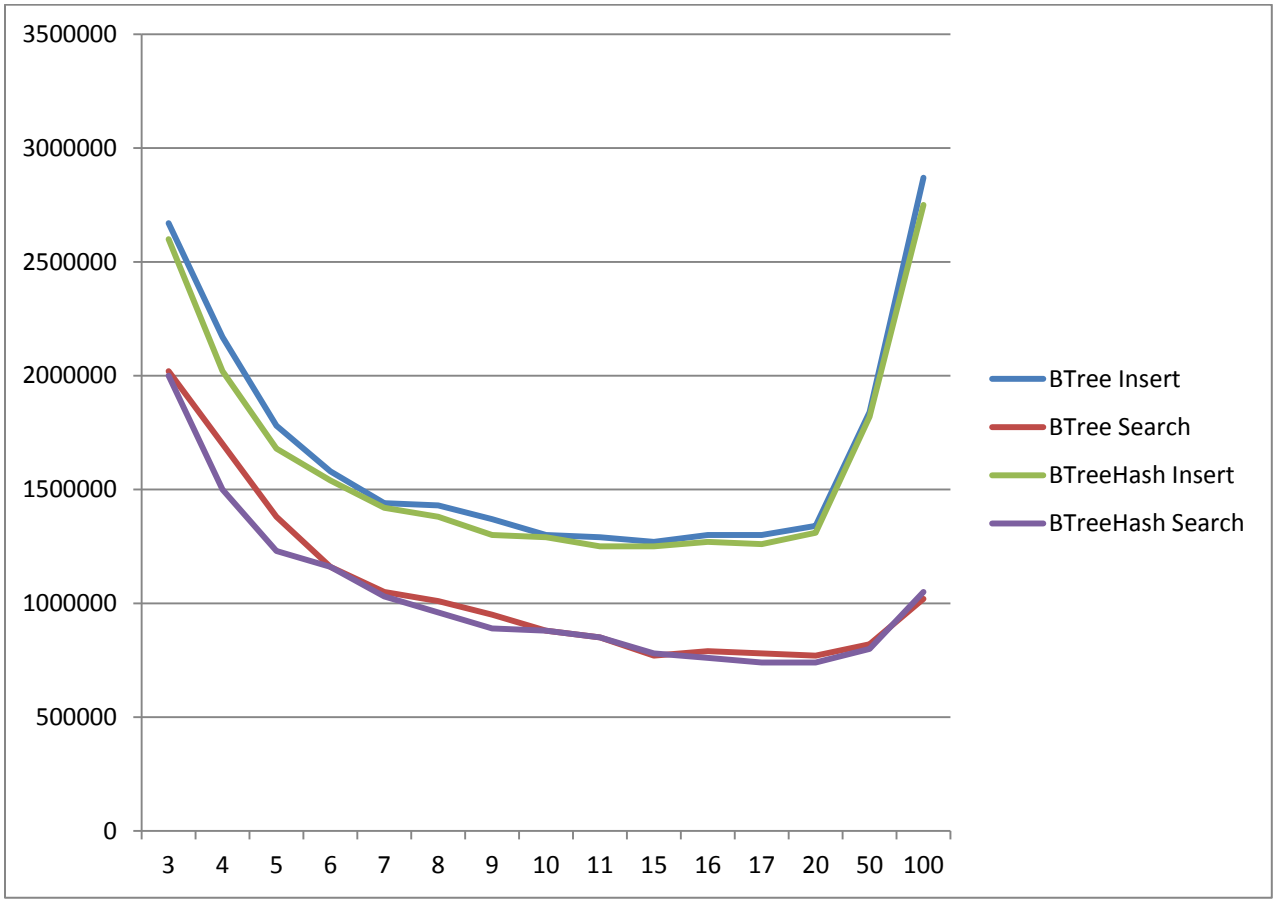
It is expected that the insert and search time should decrease as we progress to different schemes. That is search and insertion times for AVL trees is expected to be greater than that of RB Trees which themselves are greater than that of BTrees. The reason being that balancing in AVL takes more time than in RB Trees.

Note : All results shown are values obtained by running the program on **Thunder** and the time is in **microseconds**

Experiment with BTrees:

The BTree was run with several degrees to find the optimal value. According to the graph and table shown below, the optimal value of degree was calculated to be 16.

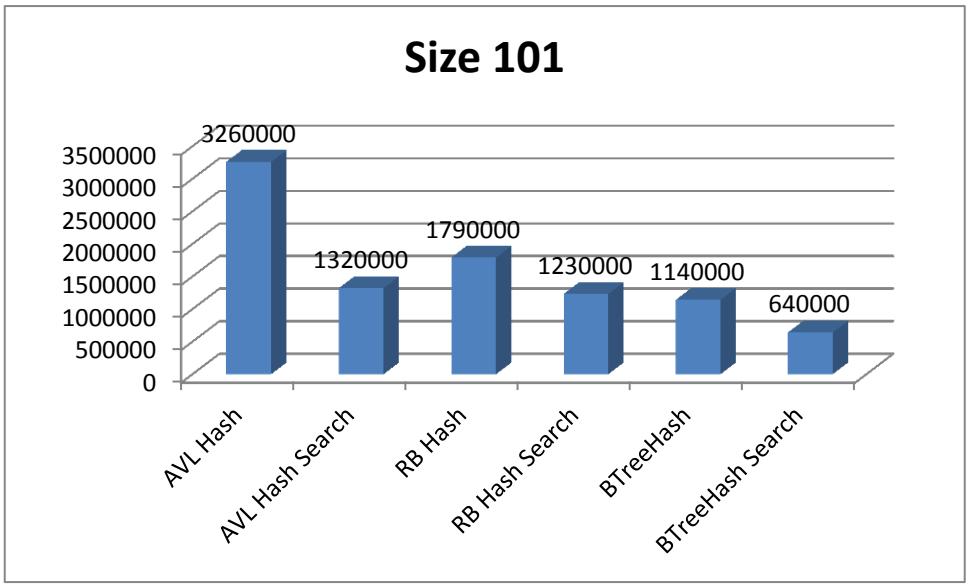
Degree	BTree Insert	BTree Search	BTreeHash Insert	BTreeHash Search
2	11500000	9710000	10960000	9320000
3	2670000	2020000	2600000	2000000
4	2170000	1700000	2020000	1500000
5	1780000	1380000	1680000	1230000
6	1580000	1160000	1540000	1160000
7	1440000	1050000	1420000	1030000
8	1430000	1010000	1380000	960000
9	1370000	950000	1300000	890000
10	1300000	880000	1290000	880000
11	1290000	850000	1250000	850000
15	1270000	770000	1250000	780000
16	**1300000**	**790000**	**1270000**	**760000**
17	**1300000**	**780000**	**1260000**	**740000**
20	1340000	770000	1310000	740000
50	1840000	820000	1820000	800000
100	2870000	1020000	2750000	1050000

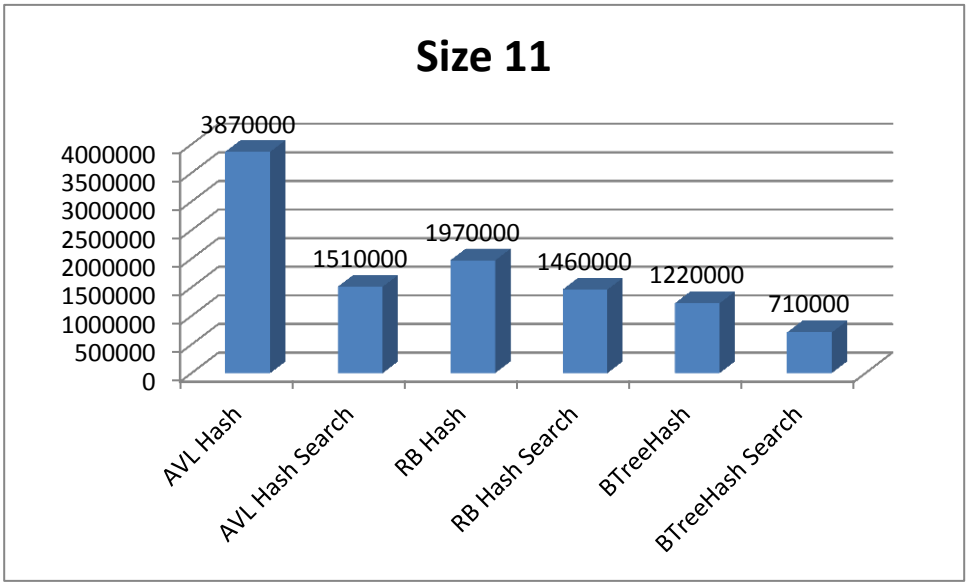


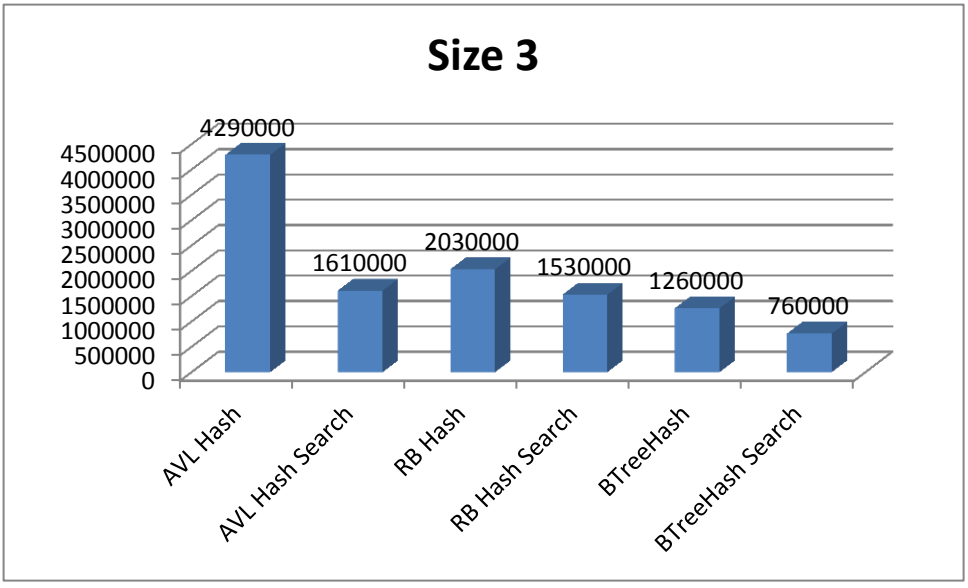
Hash Tests:

Degree	AVL Hash	AVL Hash Search	RB Hash	RB Hash Search	BTreeHash	BTreeHash Search
101	**3260000**	**1320000**	**1790000**	**1360000**	**1140000**	**640000**
11	3870000	1510000	1970000	1460000	1220000	710000
3	4290000	1610000	2030000	1530000	1260000	760000

The hash tests show that the hash size of 101 is the optimal choice because least time among the three was observed. The graphical representation follows. The input size is $10^{\wedge 6.}$


Size 101
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
3260000
1320000
1790000
1230000
1140000
640000
AVL Hash
AVL Hash Search
RB Hash
RB Hash Search
BTreeHash
BTreeHash Search


Size 11
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
3870000
1510000
1970000
1460000
1220000
710000
AVL Hash
AVL Hash Search
RB Hash
RB Hash Search
BTreeHash
BTreeHash Search

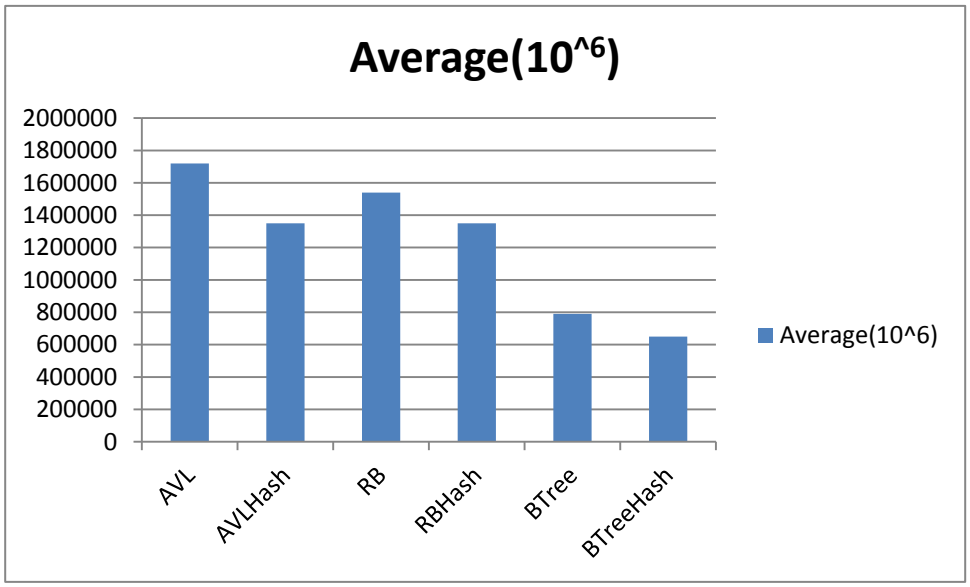

Size 3
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
4290000
1610000
2030000
1530000
1260000
760000
AVL Hash
AVL Hash Search
RB Hash
RB Hash Search
BTreeHash
BTreeHash Search

Run Tests:

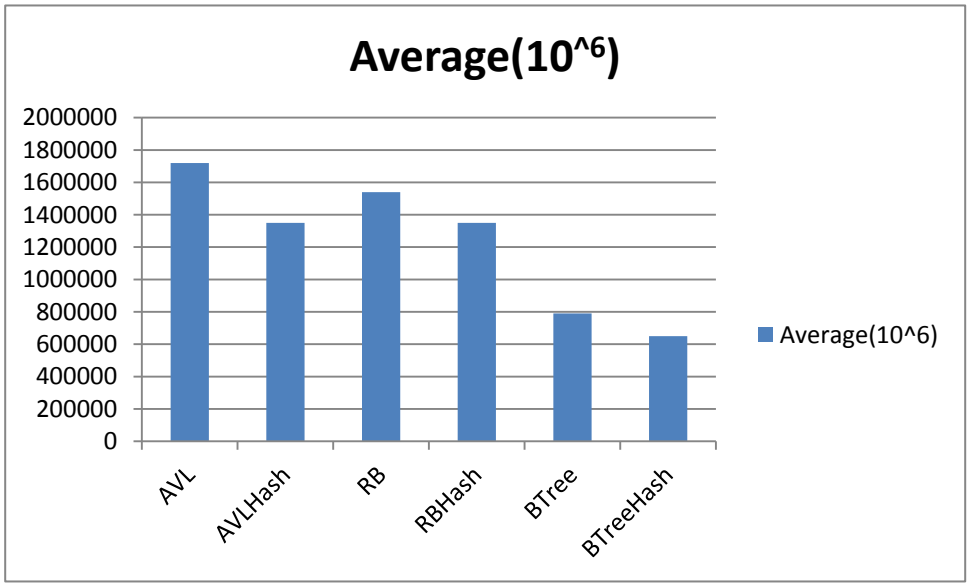
The following are run cases for a large input. The Hash size used is 101 because of the results observed above and the BTree Degree is 16 as determined by the tests done above. Best performance seen for BTrees. Average is calculated over 10 runs.

Average for input size **1000000 ($10^{\wedge 6}$)**

Insert	AVL	AVLHash	RB	RBHash	BTree	BTreeHash
Average(10^6)	4740000	254000	2040000	1800000	1310000	1150000

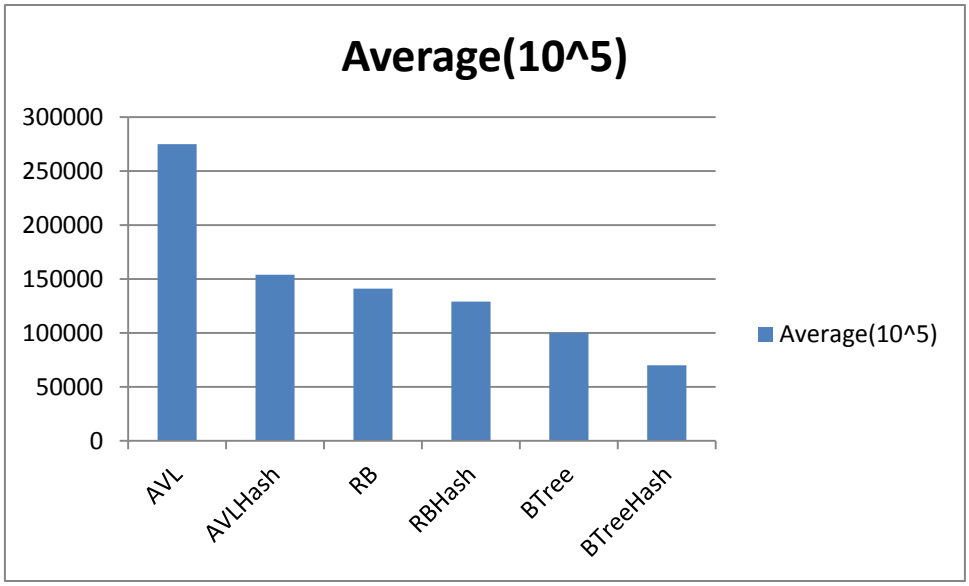


Search	AVL	AVLHash	RB	RBHash	BTree	BTreeHash
Average(10^6)	1720000	1350000	1540000	1350000	790000	650000

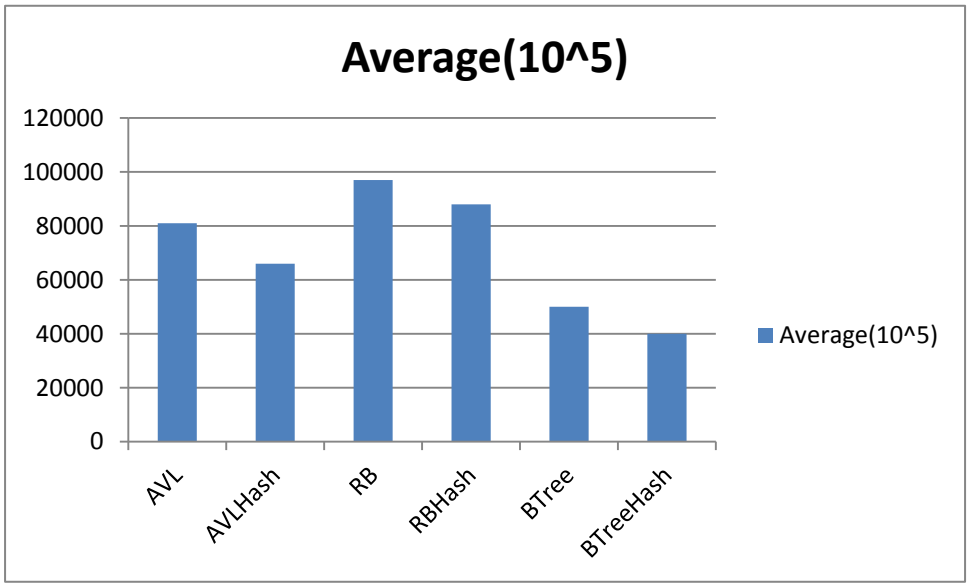


Average for input size **100000 ($10^{\wedge 5}$) –** Best performance observed by BTrees. Interesting observation is that AVLHash performs better.

Insert	AVL	AVLHash	RB	RBHash	BTree	BTreeHash
Average(10^5)	275000	154000	141000	129000	100000	70000



Search	AVL	AVLHash	RB	RBHash	BTree	BTreeHash
Average(10^5)	81000	66000	97000	88000	50000	40000



Conclusion

It can be concluded from the experiments done above that as the value goes larger, BTree are the better way to implement a dictionary. It is seen that for lower input value BTrees do not fare well as compared with others but as the input value increases, is gets quite evident the BTrees are the best choice. If nearest match is to implemented, then still BTrees would be better