Project report

Shian Huang UFID: 16656957, email: shianhuanguf@gamil.com

I am using Eclipse to compile all of my codes.

Summary of result comparison:

Before running my programs, when n=1000000, btree should be the best tree to handle this large of data, since btree can define the capacity of a node, which can highly decrease the height of tree, therefore, reducing the insertion and searching time. Red Black Tree is the slowest tree, because compared to btree, one node of RBTree can just hold one value; compared to AVLTree, RBTree's node contains one more attribute: color, so when building RBTree, we need to not only consider the rotation, but also consider the color flip, which takes more time than AVLTree. According to above analysis, AVLTree has the average performance.

Structure of my program:

BTree:

```
Class BtreeTest{

        public void insertValue(T key, T value);

        private void split(Node<T> node);

        public static int[] getRandomPermutation (int length);

        public boolean searchKey(T key);

        Class Node{

                private T getValue(int index);

                private void addValue(T value);

                private T getKey(int index);

                private void addKey(T key);

                private Node<T> getChild(int index);

                private void addChild(Node<T> child);

                private boolean removeChild(Node<T> child)

                }

                main function{

                        }

        }
```

AVLTree:

```
public class AVLTreeTest{
        class Node{
                private void setLeftNode(Node node);
                private void setRightNode(Node node);
                private void setRoot();
                private void exchangeNode(Node node1, Node node2);
                private void update();
                private int[] childHeight();
        }
        public boolean searchValue(int key);
        public void insertValue(int key, int value);
        private Node getRoot();
        private void up(Node node);
        private Node Rotation(Node node);
        private Node rightRotation(Node node);
        private Node rotateLeft(Node node);
        private Node rotateLeftDouble(Node node);
        private Node rotateRightDouble(Node node);
        private ArrayList inOrder(Node node);
        public ArrayList postOrder(Node node);
        public static void main(String[] args){
        }
}
```

Red Black Tree

```
class TreeMapDemo{

        public static int[] getRandomPermutation (int length);

        public static void main(String args[]){

                calling the put function and containKey function

        }

}
```

AVLTreeHash:

```
public class AVLTreeHash{

        public static int[] getRandomPermutation (int length);

        public int keyModeS(int number, int hashSize);

        public static void main(String args[]){

        calling getRandomPermutation();

        initialize the AVLTree array;

        according to k mode s, adding value to corresponding trees;

        after inserting values, do the search operation.

        Output the insertion time and searching time

        }

}
```

The BTreeHash and Red Black Tree Hash are similar to the AVLTree Hash.

Determine the best order for BTree:

Btree	order = 30	Btree	order = 50	Btree	order = 5
insert	search	insert	search	insert	search
14297	3404	18085	3766	26935	3543
14197	3282	16193	4443	26496	3045
15619	4446	14927	4125	25317	3064
16497	4296	15330	4211	25573	3583
14721	3287	15718	4273	29335	2946
13191	3237	14699	4206	24027	3042
17327	4213	16694	4380	25540	3647
13586	3110	17770	3821	24272	3072
16289	4367	17557	4534	24426	2981
14835	3204	16022	4480	24939	3001
15055.9	3684.6	16299.5	4223.9	25686	3192.4

We will pick order = 30

Performance for different trees' hash:

TreeMapHash	size=3	size=11		size = 101	
insert	search	insert	search	insert	search
6918	1359	7085	1424	6676	986
6126	1436	6693	1427	6597	1446
6328	1551	6482	1428	7413	1162
6776	1463	5708	1037	6910	1372
6604	1400	6705	1065	7496	1378
6216	1448	7231	1424	6825	1376
7677	1414	5992	1041	6820	1377
6462	1453	6064	1043	7066	1374
6714	1455	5728	1045	6212	1377
5765	1185	7148	1419	6353	1382
6558.6	1416.4	6483.6	1235.3	6836.8	1323

AVLTreeHash	size=3	size=11		size = 101	
insert	search	insert	search	insert	search
1822	1047	1822	990	1749	962
1890	1060	1822	1040	1780	956
1995	1063	1821	990	1615	837
1924	1068	1766	976	1785	961
1793	1062	1763	980	1780	961
1919	1074	1852	1017	1843	993
1928	1060	1790	984	1788	983
1902	1077	1786	997	1827	972
1849	1063	1789	997	1774	987
1917	1054	1809	996	1809	978
1893.9	1062.8	1802	996.7	1775	959

BTreeHash	size=3	size=11		size=101	
insert	search	insert	search	insert	search
12560	3562	14277	3871	12681	3624
15095	3291	12550	3492	12201	3542
14748	3280	13180	3421	12338	3678
12689	3255	11483	3657	12994	3210
12540	3532	12247	3336	15276	3323
11158	3321	12068	3325	12093	3410
11112	3492	11227	3416	14444	3720
12738	3508	12216	3245	13083	3211
12080	3821	12168	3216	12192	3023
12205	3664	12065	3268	11398	3680
12692.5	3472.6	12348.1	3424.7	12870	3442.1

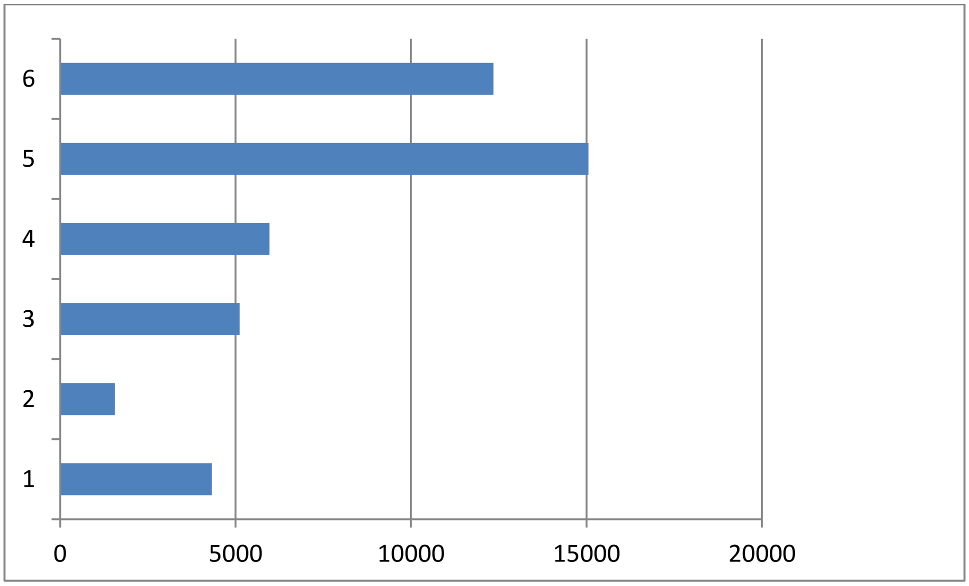
We will pick size = 11.

Insert time and search time for Six structures(the last row for every chart is the average for every column)

Data for six structures of insertion time:

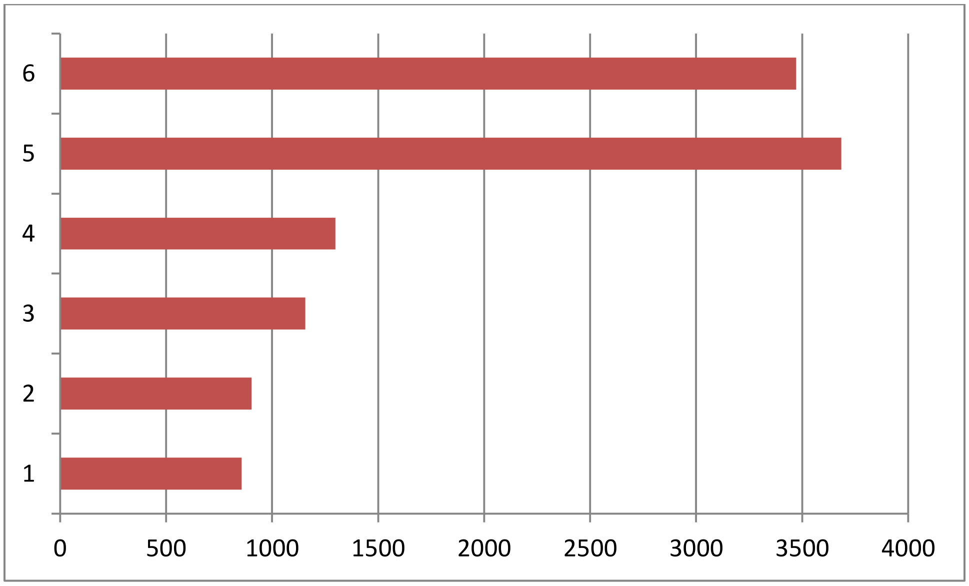
Insertion operation:

1 AVLTree	2 AVLTreeHash	3 RBTree	4 RBHash	5 Btree	6 BTreeHash
insert	insert	insert	insert	insert	insert
4576	1605	4924	5821	14297	14277
4541	1747	5294	6047	14197	12550
4749	1487	4967	6614	15619	13180
4110	1525	5138	6809	16497	11483
4207	1500	5512	5748	14721	12247
4254	1658	5231	5824	13191	12068
4187	1525	4794	5595	17327	11227
4202	1543	5388	5550	13586	12216
4224	1537	4935	5922	16289	12168
4234	1500	4981	5698	14835	12065
4328.4	1562.7	5116.4	5962.8	15055.9	12348.1



Search operation:

1 AVLTree	2 AVLTreeHash	3 RBTree	4 RBTreeHash	5 Btree	6 BTreeHash
search	search	search	search	search	search
839	877	1134	1291	3404	3562
847	1058	1130	1301	3282	3291
924	871	1150	1302	4446	3280
846	870	1145	1291	4296	3255
844	867	1139	1306	3287	3532
878	958	1123	1308	3237	3321
834	864	1140	1301	4213	3492
863	898	1126	1314	3110	3508
845	906	1298	1310	4367	3821
844	862	1177	1255	3204	3664
856.4	903.1	1156.2	1297.9	3684.6	3472.6



Conclusion:

The testing result is not matching my expectation, BTree is the slowest, maybe there is something wrong with my code, I will further study this problem. However, AVLTree is faster than Red Black tree is matching my analysis. According to my testing result, I will choose the AVLTree to implement a dictionary whose size is 1000000.