COP 5536 FALL 2012

Programming Project Report

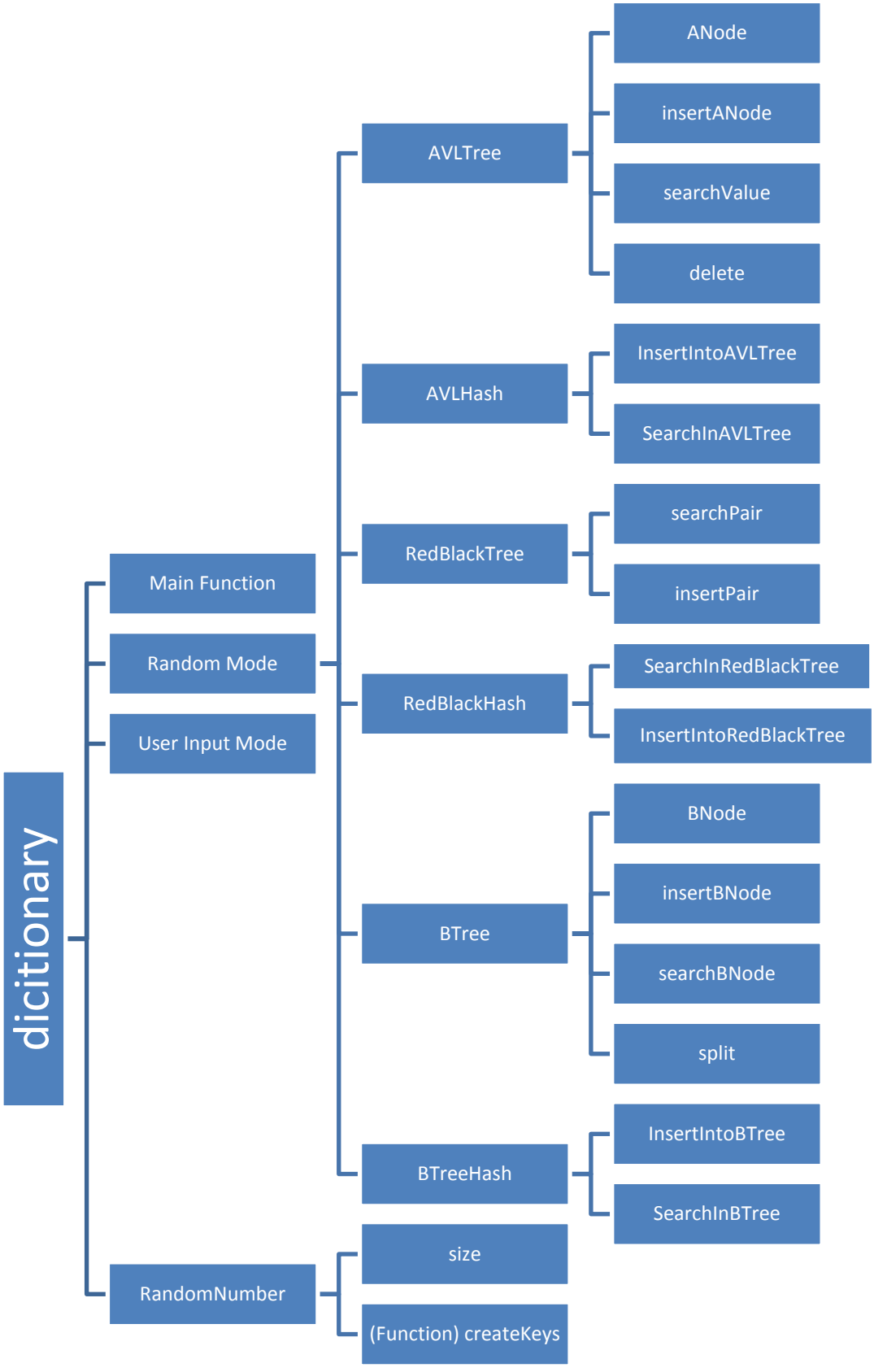
Name: Dawei Jia

UFID: 91365931

UF Email account: jiadawei@ufl.edu

The structure of the program

`



The statement of complier

I write this program using java in windows. And I write my code and compile this program in MyEcilpse. So when user uses this program, he should import this project into MyEcilpse and debug and run application in it. The input file should be in the project file. I set it as “123.txt”. Or user must input the path of input file .The format has been given in the instruction in the program. And output files are also in the project file. After user run the user input mode, he can refresh the project. And user can check output file in MyEclipse directly with new data.
User can change the data of 123.txt in MyEclipse and after run the application, user can see the accurate output file by refreshing the project. I strongly recommend user use this method to test the user input mode. It is very easy in this way.

The structure of program

6 data structures are described in 6 java files. The dicitionary.java is the main java file. User can choose random mode and user input mode in the beginning. When user choose random mode, it will show the results of inserting and searching one million pairs into 6 structures. When user chooses user input mode, he should input the accurate file name. It is better to put the file in the project file. Otherwise user should input the absolute path of file.
The main methods and attributes of every data structure are shown on the above picture.
Every method has detailed comments in the program.

Function prototypes

The attribute getter, setter methods and the construction method for the class are not shown in the following functions prototypes.

```
Class Anode<T extends Comparable, V>{
public int leftHeight()
public int rightHieght()
public int calculateHeight();
public int calculateHeight(ANode<T,V> tree)
}

Class AVLTree<T extends Comparable, V>{
public void insertANode(T key,V value)
public ANode<T,V> insertANode(ANode<T,V> parent, T key,V value)
public void searchValue(T key)
public void delete(T key)
public ANode<T,V> delete(ANode<T,V> parent, T key)
public ANode<T,V> LRotation(ANode<T,V> parent)
public ANode<T,V> RRotation(ANode<T,V> parent)
public ANode<T,V> leftRotate(ANode<T,V> parent, ANode<T,V> child)
public ANode<T,V> rightRotate(ANode<T,V> parent, ANode<T,V> child)
```

```
public void printInOrder()
public void printInOrder(ANode<T,V> root)
public void AVL_inorder()
public void printPostOrder()
public void printPostOrder(ANode<T,V> root)
public void AVL_postorder()
}

Class AVLHash<T extends Comparable, V>{
public int hashTable(T key)
public void InsertIntoAVLTree(T key,V value)
public void SearchInAVLTree(T key)
public void printInOrder()
public void printPostOrder()
}
Class Node<T extends Comparabe, V>{

}

Class BNode<T extends Comarable, V>{
public ArrayList<Node> sortPairs()

}

Class BTree<T extends Comparable, V>{
public void insertBNode(Node pair)
public void insertToLeaf(BNode bn,Node pair)
public void checkAndSplit(BNode bn)
public void split(BNode bn)
public void searchValue(T key)
public void printSortedOrder()
public void printSortedOrder(BNode<T,V> root)
public void printOutFile()
public void printLevelOrder()
public void printLevelOrder(BNode<T,V> root)
public void BTreeHashLevel()
public void BTreeHashLevel(BNode<T,V> root)

}

Class BTreeHash<T extends Comparable, V>{
public int hashTable(T key)
public void InsertIntoBtree(T key,V value)
public void searchInBtree(T key)
```

```
public void searchInBtree(T key)
public void printLevelOrder()
}
Class RedBlackTree<T extends Comparable,V> extends TreeMap{
public void searchPair(T key)
public void insertPair(T key,V value)
}

Class RedBlackHash<T extends Comparable,V>{
public int hashTable(T key)
public void InsertIntoRedBlackTree(T key,V value)
public void SearchInRedBlackTree(T key)
public void printInOrder()
public void printPostOrder()
}

class randomNumber{
public int[] createKeys()
}
class randomMode{
public randomMode()
public void execute()
public void executeBTree()
}
class userInputMode{
public userInputMode()
public void execute()
}

class dicitionary<T extends Comparable,V>{
public static void main(String args[])
}
```

Random Mode

1. Run your program (all codes) in random mode for n = 1000000 and tabulate the reported average insert and search times for each of the 6 structures (one table for inserts, another for searches). Show these average times also using bar charts. For the BTree codes, use the optimal BTree order determined in 4. and the s value determined in 5.

For insert operation

n=1000000
BTree order=4

Hash table size=5

	1	2	3	4	5	6	7	8	9	10
AVLTree	2839	2232	2169	2208	2470	2644	2071	2308	2025	2214
AVLHash	1976	1833	1642	1564	1525	1521	1846	1866	1742	1612
RBTree	1938	1818	1441	1635	1599	1584	1589	1542	1694	1632
RBHash	1568	1322	1274	1264	1561	1842	1446	1166	1315	1322
BTree	8660	7408	6454	6461	7510	6557	7388	6461	6403	7295
BTreeHash	6085	9011	5185	7161	5922	5964	5710	5798	6009	6049

	AVLTree	AVLHash	RBTree	RBHash	BTree	BTreeHash
Average Time	2318	1712	1493	1410	6404	6289

For search operation

n=1000000

BTree order=4

Hash table size=5

	1	2	3	4	5	6	7	8	9	10
AVLTree	1743	1284	1389	1397	1436	1470	1507	1480	1463	1528
AVLHash	1383	1282	1338	1320	1369	1517	1416	1374	1217	1292
RBTree	1075	1036	1178	1162	1254	1210	1104	1258	1243	1239
RBHash	1116	1308	1079	1175	1160	1103	1155	1187	1059	1193
BTree	3633	4045	3838	4703	4156	5638	4167	4068	4010	4238
BTreeHash	3468	3853	3824	3898	3838	3789	3721	3594	3660	3970

	AVLTree	AVLHash	RBTree	RBHash	BTree	BTreeHash
Average Time	1469	1350	1175	1153	4249	3761

2. Experiment with your BTree codes (d and e) in random mode with n = 1000000 and determine the optimal BTree order. Your report should include the insert and search times for a few of the Btree orders experimented with.

Insert operation(BTree)

Order/time	1	2	3	4	5
6	7056	5281	5332	5440	4839
7	6147	5242	4633	4586	4640
8	6035	4584	4564	5125	4608
9	6159	5416	4564	4442	4681

10	7721	5524	4797	4850	5461
15	9327	6905	5470	5389	5404
20	10346	7579	7174	7173	7869

	6	7	8	9	10	15	20
Average time	5589	5049	4153	5052	5670	6499	8028

Since the average time for order 4 is 6000. I think the optimal order is 8 for insert operation.

Search Operation(BTree)

Order/time	1	2	3	4	5
6	3323	3291	3222	3555	3678
7	3437	3334	3315	3305	3321
8	3346	3526	3877	3409	3508
9	3617	3459	4451	3596	4302
10	3390	3195	3213	3568	3441
15	4329	3781	3716	3872	3723
20	4329	6791	5395	4160	3631

	6	7	8	9	10	15	20
Average time	3413	3342	3553	3885	3361	3884	4861

Since the average time for order 4 is 4000. The optimal order is 7 for search operation.

Insert operation(BTreeHash) hashSize=11

Order/time	1	2	3	4	5
6	3891	3948	3885	3887	3940
7	4124	3787	3554	4053	3643
8	3861	3530	5104	4000	3654
9	3414	3677	5232	3376	3611
10	3505	3491	3457	3507	3516
15	6566	4368	4130	3923	4059
20	5642	5044	4977	5356	5082

	6	7	8	9	10	15	20
Average time	3910	3832	4029	3862	3495	4609	5220

Since the average time for order 4 is 6000. The optimal order is 10 for insert operation.

Search operation(BTreeHash) hashSize=11

Order/time	1	2	3	4	5
6	3378	3216	3252	3249	3261
7	3102	3014	2923	3023	3212
8	3174	3227	3135	3164	3091
9	3144	3165	3470	3128	3199
10	3053	3072	3027	3094	3069
15	3568	4780	3365	3293	3385
20	4168	4335	4385	4420	4116

	6	7	8	9	10	15	20
Average time	3271	3154	3158	3221	3063	3678	4264

Since the average time for order 4 is 4000. The optimal order is 10 for search operation.

3. Experiment with your hashed structures with s values 3, 11 and 101. Your report should include the insert and search times for these s values experimented with.

For insert operation

n=1000000
BTree order=4
Hash table size=3

	1	2	3	4	5	6	7	8	9	10
AVLHash	1888	1152	1750	1647	1557	1722	1749	1786	1894	1592
RBHash	1615	1271	1231	1529	1241	1251	1234	1899	1638	1270
BTreeHash	5308	5993	5968	5167	5488	7728	5906	5262	5524	5551

	AVLHash	RBHash	BTreeHash
Average Time	1673	1412	5789

n=1000000
BTree order=4
Hash table size=11

	1	2	3	4	5	6	7	8	9	10
AVLHash	2164	2056	1690	1771	2215	1951	2133	1917	1863	2084
RBHash	1770	1441	1275	1241	1453	1751	1472	1446	1421	1275
BTreeHash	5669	5673	5775	5450	7561	6171	5963	5608	5755	7835

	AVLHash	RBHash	BTreeHash
Average Time	1984	1454	6146

n=1000000
BTree order=4
Hash table size=101

	1	2	3	4	5	6	7	8	9	10
AVLHash	2006	1945	1935	1786	1885	2010	1762	1694	2001	1805
RBHash	1899	1416	1397	1391	1543	1468	1413	1382	1276	1345
BTreeHash	5707	6104	6001	5658	6054	7891	5665	5766	8071	5701

	AVLHash	RBHash	BTreeHash
Average Time	1882	1453	6261

For search operation

n=1000000
BTree order=4
Hash table size=3

	1	2	3	4	5	6	7	8	9	10
AVLHash	1474	1627	1437	1400	1454	1349	1324	1266	1469	1689
RBHash	1167	1209	1161	1112	1120	1199	1165	1379	1221	1144
BTreeHash	4579	3775	3663	3887	3962	3911	4590	3799	4681	3807

	AVLHash	RBHash	BTreeHash
Average Time	1448	1187	4065

n=1000000
BTree order=4
Hash table size=11

	1	2	3	4	5	6	7	8	9	10
AVLHash	2322	1494	1609	1472	1634	1488	1830	1756	1532	1510
RBHash	1303	1497	1276	1184	1201	1270	1206	1194	1171	1209
BTreeHash	4388	4002	4267	5429	4194	3898	3960	4128	4202	4142

	AVLHash	RBHash	BTreeHash
Average Time	1664	1251	4261

n=1000000
BTree order=4
Hash table size=101

	1	2	3	4	5	6	7	8	9	10
AVLHash	1409	1206	1228	2138	1220	1170	1105	2186	1268	1239
RBHash	1109	1026	1062	1670	1024	1009	1017	1207	1078	1205
BTreeHash	3447	3384	3496	3425	3268	3539	3532	3716	6103	3496

	AVLHash	RBHash	BTreeHash
Average Time	1416	1140	3740

Summary of result comparison

I think RedBlack Hash will have the best performance. It has similar efficiency comparing to binary search tree. Since it extends TreeMap, which is a class that existing already, it can have better algorithm regarding to my algorithms for AVL tree and B tree. Although the height of B tree is less than binary search tree, the searching process in the node will have more expense which can offset the less height.
Based on the experiments, the best way to implement a dictionary with one million keys is to use red black hash. Considering the worst case, the time for red black will not become O(n).
For the relative performance of these structures, the expected performance of is O(log N). Therefore, there is little difference between the efficiencies of these 6 structures.
I think we should consider the average time of search and insert operation. Since we input a random permutation, we can get some evidence data after a large number of experiments. And about the dictionary, the main operations are search and insert. Therefore we should consider the overall condition of search and insert.