COP5536 Fall 2012
Programming Project Report

Soham Das
UFID-36481082
email-soham@ufl.edu

October 27, 2012

1 Compilation Instructions

The entire project in coded in C++ and can be compiled using the g++ compiler.

1. Please extract the compressed folder *das_ soham.tar*.
2. There are two sub-folders for the two modes- *ModeRandom* and *ModeUser*.
3. In *ModeRandom*, the running time of insertion and searching is studied for the 6 different data-structures. The folder has a *makefile* in it to run all the programs. It takes a number n as input and generates a random permutation of n integers- 1 to n. Each of these integers serves as a key (value associated with key k is $2k$) and the time required to insert them in the dictionaries and then search each key in the order of the permutation is measured. The experiment is repeated for 10 different permutations and the running-times are recorded. Moreover, experiments were performed to figure out the optimal Btree order and how running times changed for different values of S. Some interesting observations from these experiments will be reported in the sections to follow.
4. In *ModeUser*, there is a *makefile* to run the programs for AVL, AVLHash, BTree and BTreeHash. Each program takes the input-filename from the user (the file has the number of keys, mentioned in its first line as specified) and displays the output dictionaries. The following output files are generated- AVL_ inorder.out, AVL_ postorder.out, AVLHash_ inorder.out, AVLHash_ postorder.out, BTree_ sorted.out, BTree_ level.out, BTreeHash_ sorted.out and BTreeHash_ level.out. Please note that $S = 3$ and $BTreeOrder = 3$ in this mode as mentioned.
5. Please note that the Btree order and the value of S is defined in the file header.h.

2 Program Structure

The same set of files is replicated (with minor changes for outputs) for the 2 modes to execute them conveniently. The important files for the three data-structures are listed below:-

1. AVL:- lib_ AVL.h (implemented), main_ AVL_ insert_ search.cpp and main_ AVL_ insert_ search_ hash.cpp for AVL and AVLHash respectively
2. RedBlack:- map from the STL library(in-built), main_ RB_ insert_ search.cpp and main_ RB_ insert_ search_ hash.cpp for RedBlack and RedBlackHash respectively
3. BTree:- lib_ BTree_ binsearch.h (implemented), main_ Btree_ insert_ search.cpp and main_ Btree_ insert_ search_ hash.cpp for BTree and BTreeHash respectively

We discuss each of these in details in the sub-sections to follow.

2.1 AVL

This library lib_ AVL.h contains the avltree class with all other functions. It has the *root* of the entire tree as the data-member and the following member-functions:-

- avltree():- It is a constructor which starts with making the root NULL.
- void create_ root(AVLNode *avl):- Creates the root of the tree with the AVLNode pointer passed to it.
- AVLNode* searchtree(AVLNode *root,int key):- This function takes in the root of the tree and a key and searches the key in the tree. It returns the pointer to the node with the key if the search is successful, else returns a NULL pointer. The stucture is as follows:-
  1. if the root is NULL, the key is *Not Found*.
  2. if (search key = root's key), the root is returned.
  3. if (search key < root's key), move to left child and continue recursively.
  4. if (search key > root's key), move to rt child and continue recursively.
- AVLNode* insert(AVLNode *root,int key,int value,char *bf_ update_ reqd):- This function inserts an element with (key,value) pair into the AVLTree pointed by the root ptr *root*. A character ptr bf_ update_ reqd is also taken as an arguement, which decides whether balance factor of the nodes are still to be updated in the recursive calls. The four different types of balancing- LL, LR, RR and RL rotations are implemented in this function to keep the tree balanced after insertion. The stucture is as follows:-
  1. if(root is NULL), we have reached the leaf and we insert the new (key,value) pair, creating a new node here.
     We set the bf_ update_ reqd '1', because this insertion may require changes in the balance factors of nodes, up the tree.
  2. if (key < root's key), we move to left and continue recursively.
     After returning from the recursive call, we check if further updates of balance factors is required or not. If so, we decide which strategy is to be undertaken.
     - if(root's balance factor=1 and root's left_ child's balance factor=1) the new node is inserted in the left subtree of the root's left child. So we perform an LL rotation.
     - if(root's balance factor=1 and root's left_ child's balance factor!=1), the new node is inserted in the rt subtree of the root's left child. So we perform LR rotation
     - the rotation balances the tree and we set bf_ update_ reqd='0'.
     - if(root's balance factor=0), we update it to 1, since the insertion took place at it's left child.

- if(root's balance factor=-1), we update it to 0 and set bf_ update_ reqd='0'

3. if(key > root's key), we move to rt and continue recursively.
   After returning from the recursive call, we check if further updates of balance factors is required or not. If so, we decide which strategy is to be undertaken.
   - if (root's balance factor=1), no rotation is required, since key was inserted in its rt subtree and as a result it becomes 0.
   - if (root's balance factor is 0), no rotation is required, and it becomes -1.
   - if(root's balance factor =-1 and root's rt child's balance factor=-1), balancing is required as the root's new balance factor becomes -2. So we perform a RR rotation to remove the imbalance.
   - if(root's balance factor=-1 and root's rt child's balance factor!=-1), the new node is inserted in the left subtree of the root's rt child. So we perform RL rotation.
   - Now we can safely make the root's balance factor 0.
   - And we no more require to update the balance factors of the nodes higher up.

- void display_ inorder(AVLNode *root):- This function when called prints out the inorder traversal of the tree pointed by *root*. A simple recursive structure is followed:-
  if(root != NULL)
  - move to the left child
  - print the value at node
  - move to the right child

- void display_ postorder(AVLNode *root):- This function when called prints out the postorder traversal of the tree pointed by *root*. A simple recursive structure is followed:-
  if(root != NULL)
  - move to the left child
  - move to the right child
  - print the value at node

Apart from the library, 2 main() functions- main_ AVL_ insert_ search.cpp and main_ AVL_ insert_ search_ hash.cpp, are used to implement AVL and AVLHash, which uses the functions defined in the above library.

2.2 RedBlack

The map library (C++) is used for implementing the RedBlack tree. The *insert* and *find* functions, defined in this library were called to insert and search keys into the Red-Black trees. Apart from the library, 2 main() functions- main_ RB_ insert_ search.cpp and main_ RB_ insert_ search_ hash.cpp, are used to implement RedBlack and RedBlackHash.

2.3 BTree

This library lib_ BTree_ binsearch.h contains the Btree class with all other functions. Binary search is carried out at each node to search or insert and so the name. It has the *root* of the entire tree as the data-member and the following member-functions:-

- Btree():- It is a constructor which starts with making the root NULL.
- void create_ root(BtreeNode *btree):- Creates the root of the tree with the BtreeNode pointer passed to it.
- int search(BtreeNode *root,int key):- This function takes in the root of the tree and a key and searches the key in the tree. It returns the corresponding value if the key if found, else -1 is returned. Since we are only dealing with positive integers, this is not an issue. The structure is shown below:
  - if(root==NULL) return -1.
  - Perform a binary search on the node with the search_ key, the search ends at index 'pos'.
  - if(root's keys[pos]=key), return root's value at index 'pos'.
  - if((root's keys[pos] > key) OR (root's keys[pos] is vacant)) we search root's child_ ptrs[pos]
- static BtreeNode* insert(BtreeNode *root,int key,int value):- This function inserts an element with (key,value) pair into the BTree pointed by the root ptr *root*, if the capacity of a node is exceeded, the node is split into two and the splitting moves up the tree if required. The structure is shown below:-
  1. if(root is not leaf), follow child_ ptrs corresponding to the key just > search_ key.
  2. Insert the key into the root.
  3. If the capacity of the root is exceeded, we split it into 2 new nodes left_ new and right_ new now, copy the keys and child_ ptrs from the root to these new nodes. Update the sibling_ ptrs as well. The middle key is moved up and inserted into the parent of the root.
  4. if this insertion again exceeds the capacity we iteratively do the same thing.

5. This splitting of node can move up to the root, which eventually creates a new node and this case is handled separately.
6. everytime a split is made, care is taken to update the parent pointers precisely.

- void display_ sorted(BtreeNode *root):- This function when called prints out the inorder(sorted) traversal of the tree pointed by *root*. The structure is given below:-
  1. if(root's child_ ptrs[0] is NULL), we are in a leaf and we print all keys
  2. for each child_ ptr in root,
     (a) if(the child_ ptr is not NULL) we call the function recursively on this child_ ptr.
     (b) if(root's corresponding key !=-1), print it out.
- void display_ level(BtreeNode *root):- This function when called prints out the nodes of the tree, pointed by the *root*, one level after the other. We make use of the sibling_ ptrs here. The structure is as follows:-
  1. while(root!=NULL)
  2. we follow rt_ sibling_ ptrs and print nodes at each level.
  3. we move down to the next level following the leftmost child_ ptr.

Apart from the library, 2 main() functions- main_ Btree_ insert_ search.cpp and main_ Btree_ insert_ search_ hash.cpp, are used to implement Btree and BtreeHash, which uses the functions defined in the above library.

3 Result Comparison

3.1 Expected Results

To compare the performance of the different data structures, let us take a general case, where we have N (key,value) pairs to be inserted and then searched into the dictionaries. The hashed versions have S buckets. The order of the Btree is say, k. First, lets discuss non-hashed schemes.

- An AVL tree with i records has a height ht_{AVL}, where

$$log(i+1) < ht_{AVL} < 1.44log(i+2) \tag{1}$$

- A Red-Black tree with i records has a height ht_{RB}, where

$$log(i+1) < ht_{RB} < 2log(i+1) \tag{2}$$

- A Btree of order k with i records has a height h_{BT} where

$$h_{BT} \leq log_{ceil(k/2)}([(n+1)/2]+1) \tag{3}$$

From the above equations, it is evident that Btree with a order > 4 approximately is bound to be faster in insertions and searches, because the height of a Btree is lower that that of the corresponding AVL or RB trees. Moreover, if we increase the order, we expect the running times to go down till we reach a point when the nodes become so large that even scanning them using binary search takes considerable time. From that point onwards, the time starts to rise again. So there lies an optimal order of Btrees for which the sequence of insertion and searching takes place in minimum time. Between AVL and RB trees, AVL is expected to give a better performance as it has a tighter bound on the maxmimum height.

Now, lets move on to the hashed schemes. If we are using a fair hash function, that distributes the keys almost uniformly into each of the S buckets, we expect the size of each bucket after i insertions to be i/S. Thus the running time to insertion or searching at this point gets reduced to $(log(i/s)) = log(i) - log(S)$. Thus if we increase the S value by a small amount, we surely expect the running times to go down but not by much. For example, if we use say, $S = 3$, $log3$ hardly makes any difference. However for larger S values we expect things to be better.

3.2 4. Optimal Btree Order

The running times of insertion and searching in Btree is measured in the random mode with $n = 1000000$. The results for insertion times and search times are shown in the Figures 1 and 2 respectively and the time-values are listed in the table below. The running times are measured for orders ranging from 3 to 50. A few of them are shown in the graphs. From this experiment, it can be seen that the optimal Btree order is around 15 to 20. If we take an order less than that, the height of the tree increases and makes the operations expensive. On the other hand, as we increase the order to higher values, the searching in a node takes time, which in turn increases the cost of operations. Thus insertion and searching both takes minimum time when Btree order is around 15 to 20.

Order	Insertion Times	Search Times
3	1.50	1.03
5	1.20	0.72
10	1.11	0.55
15	1.07	0.53
20	1.09	0.55
25	1.2	0.54
30	1.22	0.58
35	1.24	0.61
40	1.31	0.63
45	1.28	0.66
50	1.37	0.63

Table 1: Table showing Running Times in secs for different BTree Orders

A few points worth mentioning over here is that in this Btree implementation, we have kept the keys and pointers of a node in arrays. Inserting keys in these arrays takes considerable time for shifting. A linked-list implementation may come handy here and this can affect the optimal order of the tree.

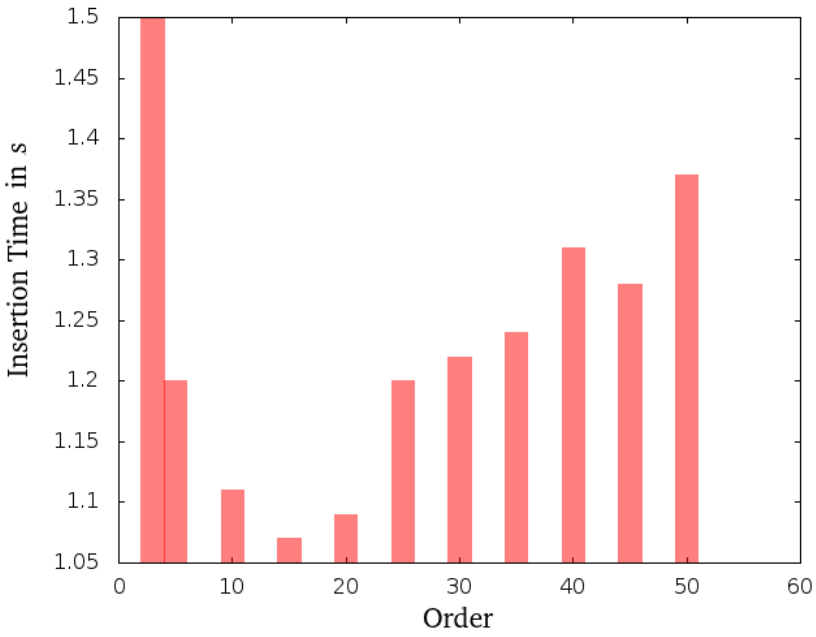

Figure 1: Insertion Times in secs with Btree Order

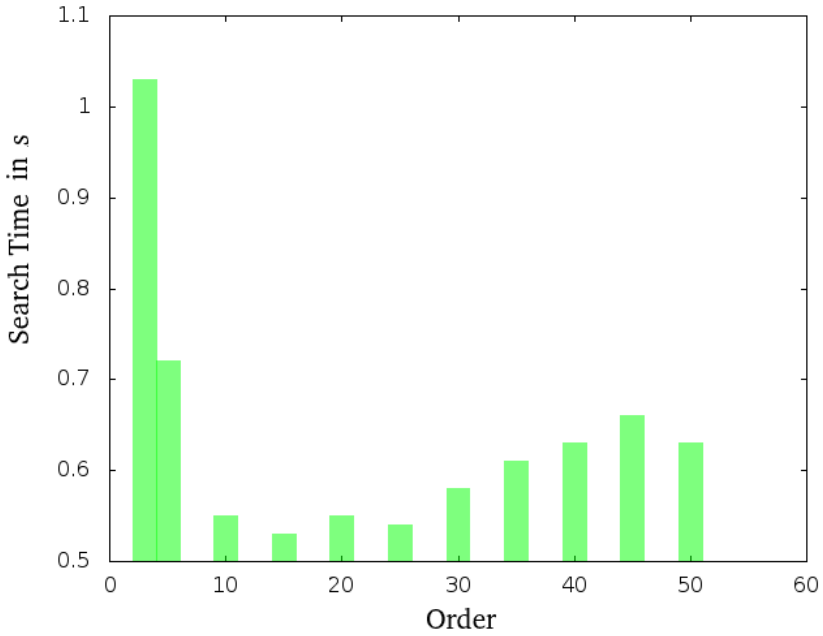

Figure 2: Search Times in secs with Btree Order

3.3 5. Different S Values

The running times of insertion and searches are measured for the 3 hashed data structures and S is varied to see whether there is any improvement. Like the previous experiment, $n = 1000000$ and the optimal Btree order 15 is used expecting best performance. 3 values of S are tried out- 3, 11 and 101 and the results are displayed in Figure 3 and 4.

	AVLHash INS	AVLHash SRCH	RBHash INS	RBHash SRCH	BtreeHash INS	Btreehash SRCH
3	1.18	0.98	1.61	1.36	1.03	0.54
11	1.14	0.94	1.58	1.32	1.04	0.49
101	1.07	0.88	1.49	1.23	0.95	0.44

Table 2: Table showing Running Times in secs for different S values

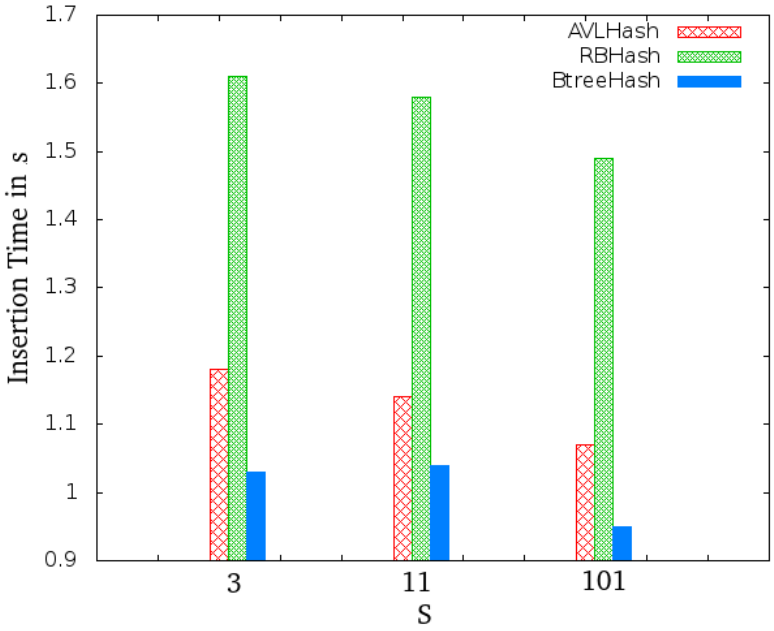

Figure 3: Insertion Times in secs with S

The insertion times indeed improve as we increase the S value as evident from Figure 3. In figure 4 we have plotted the search times and we find improvements for higher values of S. Increasing the values of S creates multiple dictionaries with smaller number of keys, so we may have a reduction in the height. As a result the operations become faster.

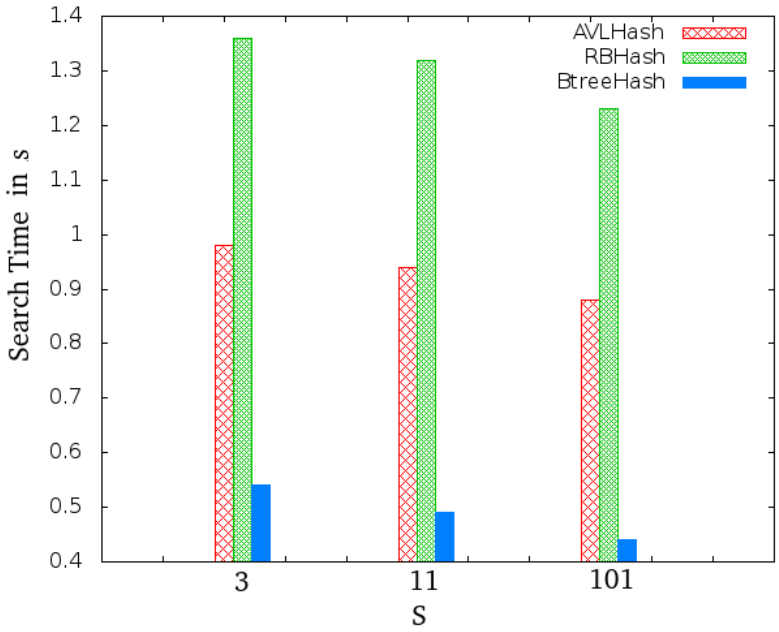

Figure 4: Search Times in secs with S

3.4 6. Different S Values

The programs are run in the random mode with $n = 1000000$, $S = 101$ and Btree Order 15. Experiments are run 10 times and the run-times for insertion and searches are listed in the tables below:

INS Times	1	2	3	4	5	6	7	8	9	10	AVG
AVL	1.19	1.20	1.19	1.19	1.20	1.19	1.19	1.20	1.19	1.18	1.192
AVLHash	1.08	1.09	1.07	1.09	1.07	1.06	1.27	1.07	1.07	1.08	1.095
RB	1.68	1.68	1.67	1.68	1.67	1.66	1.69	1.68	1.68	1.73	1.682
RBHash	1.49	1.51	1.49	1.50	1.50	1.50	1.77	1.49	1.50	1.56	1.531
BTree	1.07	1.07	1.06	1.08	1.10	1.05	1.12	1.08	1.07	1.09	1.079
BTreeHash	0.94	0.93	0.93	0.94	0.95	0.92	0.92	0.93	0.93	0.94	0.933

Table 3: Running Times in secs for Insertions in 6 data-structures (10 runs)

SRCH Times	1	2	3	4	5	6	7	8	9	10	AVG
AVL	0.99	0.99	0.99	0.99	0.99	0.98	1.01	0.99	1.00	0.99	0.992
AVLHash	0.89	0.89	0.89	0.88	0.89	0.89	0.94	0.90	0.90	0.91	0.898
RB	1.38	1.38	1.38	1.38	1.41	1.39	1.48	1.39	1.39	1.46	1.404
RBHash	1.23	1.22	1.22	1.22	1.24	1.23	1.32	1.24	1.23	1.33	1.248
BTree	0.53	0.53	0.53	0.52	0.53	0.53	0.53	0.53	0.53	0.53	0.529
BTreeHash	0.43	0.44	0.44	0.44	0.44	0.43	0.44	0.44	0.43	0.45	0.438

Table 4: Running Times in secs for Searches in 6 data-structures (10 runs)

The results, averaged over 10 runs, are dispayed in a bar chart as shown in Figure 5, which shows a distinct comparison of the 6 data-structures.

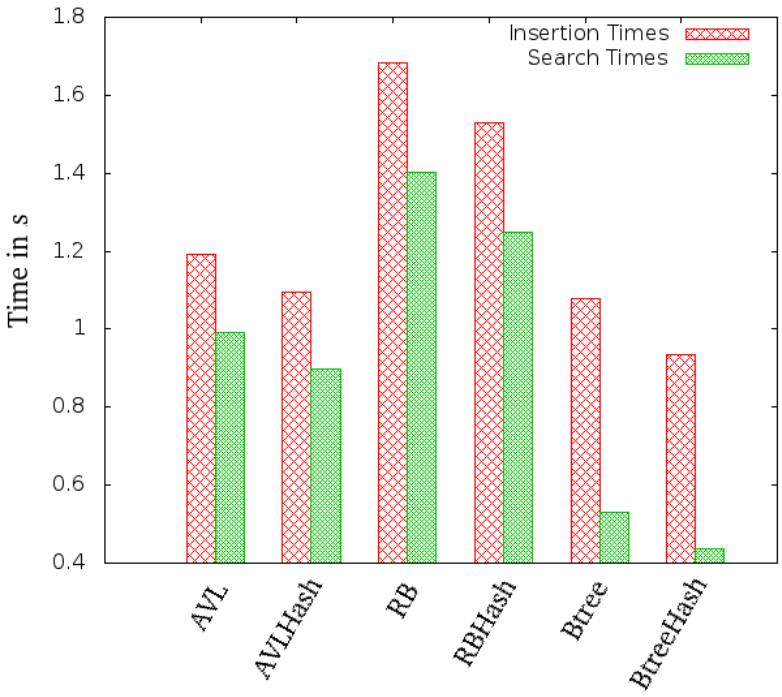

Figure 5: Run Times in secs of the 6 data-structures

3.5 6. Recommendations

Based on these experiments, a few recommendations regarding implementing dictionaries can be as follows:-

- B-tree should be used when we have a large number of keys on disk or any other slow storage medium. A hash front-end can also help since it performs better compared to a normal B-tree as evident from Figure 5.
- A Red-Black tree can be used when we have smaller number of keys and we don't have large number of insertions and searches on the tree.
- AVL tree can be used when the number of insertions is very small compared to the number of searches i.e. for static dictionaries.
- For nearest match searches we can use a Btree because it takes $O(ht)$ where $ht = log(n)/log(m)$, m being the order of the tree.
- In terms of expected performance again, Btree is the best data structure because it has a tighter bound on the height, which is the deciding factor for insertion and searching.
- In terms of the worst case performance, the AVL Tree is worst in terms of insertion in the dictionary and the red-black tree is the worst in case of searching. We have compared the performances of these data-structures with a Btree of optimal order. So this naturally gives Btree an edge.