Manu Sethi
UF-ID # 1031-7465
UF email: sethi@ufl.edu

How to compile:

$ g++ -o dictionary dictionary.cpp

How to run:

For random mode:
$ ./dictionary -r s b_tree_order

For user mode:
$ ./dictionary -u file_name

Function prototypes and program structure:

The **main()** function is located inside **dictionary.cpp.**
This above file includes the following files written for this project :

#include "AVL.h"
#include "AVLHash.h"
#include "RBHash.h"
#include "BTree.h"
#include "BTreeHash.h"

The above header files contain the implementation of each class and methods and not just their prototypes. For Red black trees we use STL **map** included as:
#include <map>

The class **AVL** defined in the file **AVL.h** has the following prototype:

```
class AVL
{
public :
	AVL() {root = NULL ; }
	pair<int , int>* find(int theKey)  ; // returning int* so as to get the iterator
	void insert(const pair<int , int>& thePair) ; // I have made the element const int
since no duplicate keys are there
	void inOrder(ostream&) const ;
	void postOrder(ostream&) ;

private :
	AVLTreeNode* root ;
	void adjustBFA2N(AVLTreeNode* someAncestor , AVLTreeNode* newNode) ;
	void inOrderInside(ostream& out , AVLTreeNode* t) const ;
	void postOrderInside(ostream& out , AVLTreeNode* t) ;

} ;
```

In the above class the insert function is written as an iterative method and all the rotations are handled inside that function itself. The function **adjustBF2AN()** stands for adjusting Balance factors from Ancestor to new Node. This is required when a new node is added but only the balance factors are required to be adjusted up to some ancestor node. This method is called in several instances for example, when a new node is added but there does not exist any Anode. In that case only the balance factors need to be adjusted up to the root of the tree and so the function is called as below with the following parameters:

adjustBFA2N(AVLTreeNode* root , AVLTreeNode* newNode) ;

This above function is also called before handling the rotations when the balance factors change.

The prototype for the AVLHash class is:

```
class AVLHash
{
public:
	// constructor
	AVLHash(int theTableSize)
	{
		s = theTableSize ;
		for (int i = 0 ; i < theTableSize ; i++)
			myHashTable.push_back(new AVL()) ;
	}
	pair<int , int>* search(int theKey) const ;
	void insert(const pair<int , int>& thePair) ;
	void inOrder(ostream&) const ;
private:
	int s ;
	vector<AVL*> myHashTable ;

} ;
```

The prototype for the Btree class is

```
class BTree
{
	public:
		BTree() {root = NULL ; }
		BTree(int m) {root = NULL ; mWay = m ;} // check this
		void insert(const pair<int,int>& thePair) ;
		void levelOrder(ostream&) ;
		void sortedOrder(ostream&) ;
		pair<int,int>* search(int theKey) const ;
	private:
		BTreeNode* root ;
		int mWay ; // what to do with this in the node field??
		void insertInParent(const pair<int,int>& thePair , BTreeNode* splitChild2 ,
stack<BTreeNode*>& ppstack) ;
		void sortedOrderInside(ostream& out , BTreeNode* t) ;
} ;
```

This code for inserting in the BTree is a recursive code which starts inserting from the leaf and propagates upward. The code inserts at each level and then checks to see if the node is overfull or not. If the node is overfull it splits it at d = ceil(mWay/2) where mWay is the order of the BTree. Then it propagates upward with the split value and tries to insert in the parent. If no parent exists then it inserts the new split value in a new node and makes that new node the root of the tree.

The prototype for the **BTreeHash** class is:

```
class BTreeHash
{
public:
	// constructor
	BTreeHash(int BTreeOrder , int theTableSize)
	{
		s = theTableSize ;
		for (int i = 0 ; i < theTableSize ; i++)
			myHashTable.push_back(new BTree(BTreeOrder)) ;
	}
	pair<int , int>* search(int theKey) const ;
	void insert(const pair<int , int>& thePair) ;
	void levelOrder(ostream&) ;
	void sortedOrder(ostream&) ;
private:
	int s ;
	vector<BTree*> myHashTable ;

} ;
```

The prototype for the class **RBHash** is:

```
class RBHash
{
public:
	// constructor
	RBHash(int theTableSize)
	{
		s = theTableSize ;
		for (int i = 0 ; i < theTableSize ; i++)
			myHashTable.push_back(new map<int,int>()) ;
	}
	map<int,int>::iterator find(int theKey) const ;
	void insert(const pair<int , int>& thePair) ;
private:class RBHash
{
public:
	// constructor
	RBHash(int theTableSize)
	{
		s = theTableSize ;
		for (int i = 0 ; i < theTableSize ; i++)
			myHashTable.push_back(new map()) ;
	}
	map::iterator find(int theKey) const ;
	void insert(const pair& thePair) ;
private:
	int s ;
	vector* > myHashTable ;

} ;
	int s ;
	vector<map<int,int>* > myHashTable ;

} ;
```

Experiments

The optimal value of BTree order was obtained by experimenting in random mode with n = 1000000 positive integers as keys. The file used for experiment is **BTreeOptimalOrderTests.cpp**

To compile and run this file use:

g++ -o btreetests BTreeOptimalOrderTests.cpp

The results obtained showed that the optimal BTree order is 11.

The results obtained by running the above file on my Linux x86_64 Ubuntu i7core are copied below. The times are in microseconds.

```
manu@SuperManu:~/Dropbox/ADSProject_Fall2012$ g++ -o btreetests
BTreeOptimalOrderTests.cpp
manu@SuperManu:~/Dropbox/ADSProject_Fall2012$ ./btreetests
order = 3      btTimeInsert = 3680000
order = 7      btTimeInsert = 2820000
order = 11     btTimeInsert = 2720000
order = 15     btTimeInsert = 2780000
order = 17     btTimeInsert = 2880000
order = 40     btTimeInsert = 4720000
order = 100    btTimeInsert = 13820000
manu@SuperManu:~/Dropbox/ADSProject_Fall2012$ g++ -o btreetests
BTreeOptimalOrderTests.cpp
manu@SuperManu:~/Dropbox/ADSProject_Fall2012$ ./btreetests
order = 7      btTimeInsert = 2840000
order = 8      btTimeInsert = 2810000
order = 9      btTimeInsert = 2770000
order = 10     btTimeInsert = 2750000
order = 11     btTimeInsert = 2730000
order = 12     btTimeInsert = 2750000
order = 13     btTimeInsert = 2740000
order = 14     btTimeInsert = 2750000
order = 15     btTimeInsert = 2750000
manu@SuperManu:~/Dropbox/ADSProject_Fall2012$
```

Average times obtained by running all the structures are tabulated below. For the BTree the optimal order 11 as determined above was used. All the hashed structures were experimented for s=3, 11, and 101. The file used is dictionary1.cpp The results in microseconds are copied below:

=====reporting average Insert times=====
BTree order=11 2.808e+06
BTreeHash order=11 s=3 2.716e+06
BTreeHash order=11 s=7 2.657e+06
BTreeHash order=11 s=101 2.54e+06
AVLTree 621000
AVLHash s=3 548000
AVLHash s=7 540000
AVLHash s=101 514000
RBTree 820000
RBHash s=3 801000
RBHash s=7 758000
RBHash s=101 697000

=====reporting average Search times=====
BTree order=11 701000
BTreeHash order=11 s=3 705000
BTreeHash order=11 s=7 673000
BTreeHash order=11 s=101 620000
AVLTree 506000
AVLHash s=3 452000
AVLHash s=7 439000
AVLHash s=101 417000
RBTree 740000
RBHash s=3 734000
RBHash s=7 697000
RBHash s=101 647000

Summary of Result Comparison

Optimal BTree order = 11

Other results and expected values are tabulated below:

Data Structure	Insert (from code) (seconds)	Insert (expected) (seconds)	Search (from code) (seconds)	Search (expected) (seconds)
AVL	**0.621**	**0.6**	**0.506**	**0.6**
AVLHash(3)	**0.548**		**0.452**	
AVLHash(11)	**0.540**		**0.439**	
AVLHash(101)	**0.514**		**0.417**	
RB	**0.820**	**0.6**	**0.820**	**0.6**
RBHash(3)	**0.801**		**0.801**	
RBHash(11)	**0.758**		**0.758**	
RBHash(101)	**0.697**		**0.697**	
BTree	**2.808**	**0.6**	**0.701**	**0.6**
BTreeHash(3)	**2.716**		**0.705**	
BTreeHash(11)	**2.657**		**0.673**	
BTreeHash(101)	**2.540**		**0.620**	

All the expected values are based on log(n) per insert. So, even though the asymptotic complexities are same, practically BTrees take more time for in memory operations. This is because at each node a vector of values is maintained which has to be sorted everytime a new node is inserted. Similarly a search is required at each node. This is the overhead for at each node in case of a BTree. However, if the data is stored in a disk and inserted from a disk then BTree is much faster because time to sort at each node is more than compensated by the disk I/O operations which are much more expensive.

Hence for in-memory operations RB or AVL is better.
For data on disk BTree is better.
Hash tables definitely perform better for exact searches because the height if tree is reduced and in a hash table the bucket can be located in O(1) expected time and then the key can be further searched in O(h) time. In a worst case scenario also the hash table with Balanced search trees will perform better rather than a hash table with a list.
For cases with nearest search matches only Balanced search trees will perform better by augmenting them with leftSize field at each node. Hash tables cannot be used in this case because we cannot find a bucket for a nearest match searches.