**ASSIGNMENT**

Bank Accounts

**TOPICS**

Binary Search Trees

Insertions, Deletions and Updates

**DESCRIPTION**

**General**

Write a program that will maintain bank accounts using a binary search tree. It will also maintain a log file (Log.txt). Initially, the program will input bank accounts from an unordered master file (Master.txt) and will create the binary search tree representing the accounts. This needn’t be a balanced tree.

*After building the tree, the program will output the contents of the binary search tree to the log file (Log.txt).*

After the tree is built, it will read transactions from a transaction file (Transaction.txt) one at a time and perform the transaction on the binary search tree.

*For each transaction, it will log the following to the log file: It will output a line containing the contents of the transaction to be performed. After the transaction is completed, it will output the contents of the whole binary search tree to the log file.*

After all transactions are done, it will store the updated tree into an ordered master file (OrderedMaster.txt) in order of account numbers and will delete the tree.

Then it will read the accounts from the ordered master file (OrderedMaster.txt) and create a balance binary search tree.

*After the balanced tree is built, it will log the contents of the balanced binary search tree to the log file.*

**Files**

The program will make use of the following files in performing its tasks. Each of the file and the way it is used is described below:

**Master File**

The master (Master.txt) file will contain the account data in no particular order. For each account, it will contain the account id, the account name and the current balance.

The program will read data from the master (Master.txt) file and build a binary search tree. This tree need not be a balanced tree.

**Transaction File**

The transaction file (Transaction.txt) will contain a set of transactions. Each transaction will consist of a one letter transaction type, an account id, the account name and a plus or minus amount. The transaction types will include the following:

* I will imply an insert transaction.
* U will imply an update transaction.
* D will imply a delete transaction

After building the binary search tree as described above, the program will input transactions from the transaction file one at a time and will update the accounts accordingly.

For an Insert (I) transaction, the program will add a new account (node) in the binary search tree.

For a Delete (D) transaction, the program will delete an account (node) from the binary search tree.

For an Update (U) transaction, the program will update the balance in the corresponding account (node) in the binary search tree. For a plus amount, it will add the given amount to the existing balance. For a minus amount, it will subtract the given amount from the existing balance.

**Ordered Master File**

Initially, the ordered master (OrderedMaster.txt) file will not exist.

When all transactions are completed, the program will output the contents of the binary search tree to the ordered master (OrderedMaster.txt) file using In-Order traversal (see sample code below). This will result in the file OrderedMaster.txt containing the account data in order by account id. For each account, it will contain the account id, the account name and the current balance.

**Log File**

***Log File Contents***

The program will log information to a log (Log.txt) file as described below.

1. Before starting any updates, the program will output the following to the log (Log.txt) file:

* It will output the contents of the whole binary search tree.

1. Then for each update, it will output the following to the log file:

* Before the update, it will output a line containing the contents of the update to be performed.
* After the update, it will output the contents of the whole binary search tree.

After all updates are completed, it will store the contents of the tree to the ordered master (OrderedMaster.txt) file using In-Order traversal and will delete the existing tree.

Then it will input the contents of the tree from the ordered master (OrderedMaster.txt) file and build a new balanced binary search tree.

1. After building the balanced binary search tree, it will output the following to the log file.

* the contents of the whole balanced binary search tree.

***Log File Format***

For logging the contents of an update, it will output one line of text containing transaction type, account id, account name and the amount value to the log (Log.txt) file.

For logging the contents of the whole binary search tree, it will output the contents of the binary search tree to the log (Log.txt) file in Reverse In-Order traversal using levels   It will output only the id and the balance for each account. For each account, it will output two lines: one line for account id and the second for account balance. It will output the tree making using of the level information so that the output will appear as a tree lying sideways (see sample code below).

**TEST DATA**

Use the data below for a test run.

**Master File:**

**27183 Teresa Wong 1234.56**

**12345 Jeff Lee 211.22**

**31456 Jack Smith 1200.00**

**14142 James Bond 1500.00**

**31623 Norris Hunt 1500.00**

**10203 Mindy Ho 2000.00**

**20103 Ed Sullivan 3000.00**

**30102 Ray Baldwin 3824.36**

**30201 Susan Woo 9646.75**

**22345 Norma Patel 2496.24**

**Transaction File**

**U 31623 Norris Hunt -1500.00**

**I 20301 Joe Hammil +500.00**

**I 31416 Becky Wu +100.00**

**U 10203 Mindy Ho -2000.00**

**U 14142 James Bond +1500.00**

**U 27183 Teresa Wong -1234.56**

**I 10101 Judy Malik +1000.00**

**I 32123 John Doe +900.00**

**I 22222 Joanne Doe +2750.02**

**D 31623 Norris Hunt  0**

**D 10203 Mindy Ho  0**

**D 27183 Teresa Wong  0**

**SUBMIT**

* Submit the source code.
* Submit a copy of the log file.

**IMPLEMENTATION**

**Bianary Search Tree**

Use a binary search tree for keeping accounts.

**Class**

Encapsulate the binary search tree in a class that provides the methods needed.

**Account Id**

Use the data type long for storing account id.

**Algorithm For Storing And Restoring The Tree**

When all transactions are completed, the program will do the following:

* Determine the total number of nodes in the tree (i.e. node count).
* Save the node count in ordered master (OrderedMaster.txt) file.
* Following the count, save the contents of the updated binary search tree in ordered master (OrderedMaster.txt) file using In-order traversal by id. (Do not save the link values).
* Make the root of the tree equals NULL. This will de-link the whole tree from the root.
* Input the node count from the ordered master (OrderedMaster.txt) file.
* Using the node count and data from the ordered master (OrderedMaster.txt) file, create a new balanced binary search tree.
* Log the contents of the binary search tree in Reverse In-order Traversal. Use a different amount of indentation at each level of the tree so that data will appear in the file as a tree lying sideways. For each node, log only the id and the balance.

**SAMPLE CODE**

#include <iostream>

#include <fstream>

#include <iomanip>

#include <string>

using namespace std;

struct RecType

{

int id;

string firstName;

string lastName;

double amount;

};

struct NodeType;

typedef NodeType \* NodePtr;

struct NodeType

{

  int id;

  string firstName;

  string lastName;

  double amount;

  NodePtr llink;

  NodePtr rlink;

};

class BinTree

{

private:

  NodePtr root;

  ifstream fin;

  ofstream fout;

  void rinsert (NodePtr & nodep, RecType rec);

  void rupdate (NodePtr nodep, RecType rec );

  void rremove (NodePtr & nodep, RecType rec );

  void rdisplay (NodePtr nodep, int level);

  void rfdisplay (NodePtr nodep, int level, ofstream & fout);

  void rstore (NodePtr nodep);

  void rrestore (NodePtr & nodep, int count);

  int rcountNodes (NodePtr nodep);

public:

  BinTree ( );

  void insert ( RecType rec );

  void update ( RecType rec );

  void remove ( RecType rec );

  void display ();

  void fdisplay (ofstream & fout);

  void store ( );

  void restore ();

  int countNodes ();

};

//The constructor

BinTree ::BinTree ( )

{

          root = NULL;

}

 //Inserting a node in the tree

void BinTree ::insert (RecType rec )

{

          rinsert (root, rec);

}

void BinTree ::rinsert (NodePtr & nodeptr, RecType rec)

{

  if (nodeptr == NULL) // reached not found

  {

  // create the node and link it

    nodeptr = new NodeType;

    nodeptr->id = rec.id;

          nodeptr->firstName = rec.firstName;

          nodeptr->lastName = rec.lastName;

          nodeptr->amount = rec.amount;

    nodeptr->llink = NULL;

    nodeptr->rlink= NULL;

  }

  else if  (rec.id < nodeptr->id)

    rinsert(nodeptr->llink, rec);

  else

    rinsert (nodeptr->rlink, rec);

}

//Updating a node in the tree

void BinTree ::update (RecType rec )

{

          rupdate (root, rec);

}

void BinTree ::rupdate (NodePtr nodep, RecType rec)

{

}

//Removing a node from the tree

void BinTree ::remove (RecType rec )

{

          rremove (root, rec);

}

void BinTree ::rremove (NodePtr & nodep, RecType rec)

{

}

//Displaying the tree in reverse in order traversal with levels

void BinTree::display ()

{

          rdisplay (root, 1);

}

void BinTree::rdisplay (NodePtr nodep, int level)

{

          if (nodep != NULL)

          {

                   rdisplay (nodep->rlink, level + 1);

                   cout <<  setw (7 \* level) << nodep->id <<endl;

                   rdisplay (nodep->llink, level + 1);

          }

}

 //Logging the tree in reverse In-order traversal with levels

void BinTree::fdisplay (ofstream & fout)

{

          rfdisplay (root, 1, fout);

}

void BinTree::rfdisplay (NodePtr nodep, int level, ofstream & fout)

{

          if (nodep != NULL)

          {

                   rfdisplay (nodep->rlink, level + 1,fout);

                   fout <<  setw (7 \* level) << nodep->id <<endl;

                   rfdisplay (nodep->llink, level + 1,fout);

          }

}

//Storing the tree to a file OrderedMaster.txt

void BinTree::store ()

{

  fout.open("OrderedMaster.txt",ios::out );

  //save the node count as the first number in the file

  int count = countNodes();

  fout << count << endl;

  //save the tree

  rstore (root);

  fout.close();

}

void BinTree::rstore (NodePtr nodep)

{

  if (nodep != NULL)

  {

    rstore (nodep->llink);

          fout << nodep->id << " "

                    << nodep->firstName << " "

                    <<  nodep->lastName <<  " "

                    << nodep->amount << endl;

    rstore (nodep->rlink);

  }

}

//Restoring the tree from a file OrderedMaster.txt

void BinTree::restore ()

{

  fin.open("OrderedMaster.txt",ios::in );

  //input node count. This was stored as the first number in file.

  int count;

  fin >> count;

  //restore the tree

  rrestore (root,count);

  fin.close();

}

void BinTree::rrestore (NodePtr & nodep, int count)

{

  if (count > 0)

  {

          //create a node

    nodep = new NodeType;

          nodep->llink = NULL;

          nodep->rlink = NULL;

          //create and fill the left subtree

    if ( (count % 2) == 0)

      rrestore (nodep->llink, ((count-1)/2) + 1);

    else

      rrestore (nodep->llink, count/2);

          //fill in the node

    fin >> nodep->id >> nodep->firstName >> nodep->lastName >> nodep->amount;

          //create and fill the right subtree

    if ( (count % 2) == 0)

      rrestore (nodep->rlink, ((count-1)/2) );

    else

      rrestore (nodep->rlink, count/2);

  }

}

//Counting nodes

int BinTree::rcountNodes (NodePtr nodep )

{

  if (nodep != NULL)

  {

    return (rcountNodes (nodep->llink) + rcountNodes(nodep->rlink) + 1);

  }

  else

    return 0;

}

int BinTree::countNodes ()

{

  return rcountNodes (root);

}

void main ( )

{

  BinTree tree;

  ofstream fout (c:\\Log.txt”);

  ifstream finMaster (c:\\Master.txt”);

  ifstream finTransaction (c:\\Transaction.txt”);

  ifstream finOrderedMaster (c:\\OrderedMaster.txt”);

  RecType r;

          finMaster >> r.id;

          while (!finMaster.eof( ) )

          {

                   finMaster >> r.firstName;

                   finMaster >> r.lastName;

                   finMaster >> r.amount;

                   tree.insert(r);

                   finMaster >> r.id;

          }

  //display the tree

  tree.display();

  tree.fdisplay(fout);

  //store the tree

  tree.store();

  //restore the tree

  tree.restore();

  //display the restored tree.

  tree.display();

  tree.fdisplay(fout);

}

**DISCUSSION**

Towards the end of the Discussion section, some code is given.  This code is provided for understanding general concepts relating to binary search tree and is not directly usable in this assignment.

General Trees:

* At each node none, one or more children.

Binary Tree:

* At each node none, one or two children
* Children named: left, right.

Binary Search Tree: Has the following properties.

A Binary Tree.

* Every entryin left subtree   is < node entry
* Every entryin right subtree is > node entry

Linked list:

* It has a head.
* Head points to a series of nodes in a linked list.
* One node points to at most one node that follows it.

Tree:

* Tree has a head which is called a root.
* The rot points to a single first node.
* A node may point to multiple nodes that follow it.
* The node is a parent node.
* The nodes that follow it are children nodes.
* A node may have zero, one or more children.

Tree vs. Linked Lists:

* In a linked list each node has at most one child.
* In a tree a node may have multiple children.

Binary Tree:

* In a binary tree each node has at most two children.
* A node that has no children is called a leaf node.

A Leaf node:

* A node that has no children is called a leaf node.

A Subtree:

* A node in a binary tree has a left subtree and a right subtree.

Left Subtree:

* All the nodes that are descendants of the left branch of a node constitute the left subtree.

Right Subtree:

* All the descendant nodes in the right branch of a node make up the right subtree.

Left Child:

* A node in the left branch of the node.

Right Child:

* The very first node attached to the right branch is called  the right child.

A Descendant:

* Any node that is attached to the left of the right branch of a node is called the descendant of the node.
* A descendant node is a child or a sub-child of the node.

Search Algorithms:

Linear Search:

* Start with the first node and search all node to exhaust the search.
* Only after searching al the nodes, we can be sure that item is missing.

Hash Search:

* Nodes are divided into multiple hash lists or queues.
* We only search one list.
* We keep the size of each list very short, by increasing the number of hash lists.
* Increasing the number of hash list has a very small overhead for each extra list, the overhead is the head pointer variable.
* Typically , attempt is made to keep the size of hash list to 2 or 3.

Drawback:

* The nodes in a hash list are not  in a chronological (alphabetical) sorted order.

Binary Search:

Requirements.

Nodes should be sorted.

When you reach the leaf node you know it doesn’t exist.

Recursion

Self referencing

//Display a linked list Using recursion.

//Code below displays a linked list in forward order and reverse order.

void display()

{

rdisplay(head};

}

void rdisplay(Nodeptr nodeptr)

{

          if (nodeptr != NULL)

 {

          //Display in forward direction

          cout<< nodeptr->item << endl;

          //recursive call

          rdisplay (nodeptr->link);

          //display in reverse order

cout << nodeptr->item<<endl; /

}

}

**Sample Code For A Sample Binary Search Tree**

(This sample tree has a very simple node containing only one item).

struct NodeType;

typedef NodeType \* NodePtr;

struct NodeType

{

int item;

NodePtr llink;

NodePtr rlink;

};

class BinTree

{

private:

NodePtr root;

void rsearch(NodePtr nodeptr, int item, int & found);

void rinsert (NodePtr & nodeptr, int item);

void rremove(NodePtr & nodeptr, int item);

void rdisplay(NodePtr nodeptr);

public:

BinTree();

void search(int item, int & found);

void insert (int item);

void remove(int item);

void display();

}

//Search Code

void BinTree::seach (int item, int& found)

{

rsearch(root, item found);

}

void BinTree::rsearch (NodePtr nodeptr, int item , int & found)

{

if (nodeptr = = 0)

          found = 0;

else if ( item = = nodeptr-> item)

found =1;

else if  (item < nodeptr->item)

rsearch( nodeptr->llink, item, fonud);

else

 rsearch(nodeptr->rlink, item, found);

}

//Insert Code

void BinTree::insert (int item)

{

rinsert (root, item);

}

void BinTree::rinsert (NodePtr & nodeptr, int item)

{

if (nodeptr = =NULL) // reached not found

{

// create the node and link it

          nodeptr = new NodeType;

         nodeptr->item – >item;

         nodeptr->llink = NULL;

         nodeptr->rlink= NULL;

}

          else if  (item < nodeptr-> item)

                             rinsert(nodeptr->llink, item);

         else

rinsert (nodeptr->rlink, item);

}

//Remove Code

void BinTree::remove (int item)

{

  rremove(root, item);

}

void BinTree::rremove (Nodeptr & nodeptr, int item )

{

          //item missing

if (nodeptr  = = NULL)

return;

//item in left subtree make recursive call

else if ( item < nodeptr->item)

rremove ( nodeptr-> llink, item );

// item in right subtree make recursive call

else if ( item > nodeptr->item)

      rremove(nodeptr->rlink, item);

//item found remove node

else{

          //case: Node is a leaf

if( nodeptr->llink = = NULL &&  nodeptr->rlink = = NULL)

{

nodeptr = NULL;

//also free the space (not shown)

                   }

//case: left child missing

else if ( nodeptr-> llink = = NULL)

{

nodeptr = nodeptr->rlink;

//also free the space (not shown)

}

//case: right child missing

else if ( nodeptr->rlink = = NULL)

{

                   nodeptr = nodeptr->llink;

//also free the space (not shown)

}

//case both children present

else

{

//call rremove\_succerssor

int item.suc;

rremove\_suc(nodeptr->rlink, item\_suc);

nodeptr->item = item.suc;

}

          }

}

//This method find the smallest successor, removes it and returns its contents.

void BinTree::rremove\_suc(NodePtr & nodeptr, int & sitem)

{

//smallest successor found

// remove it

//return its item;

if (nodeptr->llink = = NULL)

{

          //return node contents

          sitem = nodeptr->item; //return its item

          //delink the node

nodeptr = nodeptr->rlink;

//free the space (not shown)

}

// look in left subtree

else

          rremove\_suc(nodeptr->llink, sitem);

}

}

void BinTree::rremnove\_suc(NodePtr &nodeptr, int & sitem)

{

//smallest successor found

//remove it and return its item

if (noeptr->llink = = NULL

{

sitem = nodptr> item;

nodeptr = nodeptr->rlink;

//Also delete the node (not shown).

}

//Not yet found. Look in left subtree recursively.

else

{

rremove\_suc(nodeptr->llink, sitem);

}

}

//Display Tree

**//Traversal Techniques**

//Inorder Traversal

void BinTree::display()

{

rdispay(root);

}

void BinTree::rdiaplay(NodePtr nodeptr)\_

{

if (nodeptr !=NULL}

{

//display left subtree (recursive call)

rdiapay(nodeptr->llink);

//display the item

cout<< nodeptr->item<<endl;

                   //dispay right subtree(recursive call)

                   rdisplay(nodeptr->rlink);

          }

}

//Preorder Traversal

void BinTree::display()

{

rdispay(root);

}

void BinTree::rdisplay (NodePtr nodep)

{

          if (nodep != NULL)

          {

                   //display the node

                   cout <<nodep->item << endl;

//recursive call

                   rdisplay (nodep->llink);

                   //recursive call

                   rdisplay (nodep->rlink);

          }

}

//Postorder Traversal

void BinTree::display()

{

rdispay(root);

}

void BinTree::rdisplay (NodePtr nodep)

{

          if (nodep != NULL)

          {

//recursive call

                   rdisplay (nodep->llink);

                   //recursive call

                   rdisplay (nodep->rlink);

                   //display the node

                   cout <<nodep->item << endl;

          }

}

//Reverse Inorder Traversal

void BinTree::display()

{

rdispay(root);

}

void BinTree::rdisplay (NodePtr nodep)

{

          if (nodep != NULL)

          {

          //recursive call

          rdisplay (nodep->rlink);

          //display the node

          cout <<nodep->item << endl;

          //recursive call

          rdisplay (nodep->llink);

          }

}

//Reverse Inorder Traversal With Varying Level Indentation

void BinTree::display ()

{

          rdisplay (root, 1);

}

void BinTree::rdisplay (NodePtr nodep, int level)

{

          if (nodep != NULL)

          {

                   rdisplay (nodep->rlink, level + 1);

                   cout <<  setw (7 \* level) << nodep->item<<endl;

                   rdisplay (nodep->llink, level + 1);

          }

}