/\* Assignment: Tree Work Out

\* Author: Ryan Wood

\* Date Created: 4/14/2017

\* Last Modified: 4/1/2018

\* Requirements: Create a binary tree class that includes the functions

\* specified in Tree Work Out program specification

\*/

/\*Binary Tree Main file\*/

#include <iostream>

#include <fstream>

#include "BinaryTree.h"

using namespace std;

ifstream inFile("TreeRandNbrs.dat");

ofstream outFile("TreeOutput.out");

int values[150];

/\* Function: buildTree(BinaryTree&)

\* Purpose: builds the binary tree using the values

\* stored in the input file

\* Parameters: a reference to a binary tree

\*/

void buildTree(BinaryTree &bt)

{

int num = 0;

int count = 0;

while(inFile >> num)

{

values[count] = num;

bt.insert(num);

count++;

}

}

/\* Function: sumOfDigits(int)

Purpose: recursive function that takes each digit of the

given number and adds them up, returning the

resulting sum of the digits

Parameters: the number

Return: the sum of the digits of the number

\*/

int sumOfDigits(int num)

{

if(num <= 0)

return 0;

else

return (num % 10) + sumOfDigits(num/10);

}

/\* Function: testBinaryTreeFunctions(BinaryTree&)

\* Purpose: tests all the binary tree functions besides

\* insert. Satisfies the requirements of the

\* requirements specification

\* Parameters: the Binary Tree

\*/

void testBinaryTreeFunctions(BinaryTree &bt)

{

int count = 0;

int val = 0;

/\*Satisfies requirement a.\*/

outFile << "printInOrder" << endl;

bt.printInOrder(outFile);

outFile << endl;

/\*Satisfies requirement b.\*/

outFile << "printPreOrder" << endl;

bt.printPreOrder(outFile);

outFile << endl;

/\*Satisfies requirement c.\*/

outFile << "printPostOrder" << endl;

bt.printPostOrder(outFile);

outFile << endl;

/\*Satisfies requirement d.\*/

outFile << "numNodes" << endl;

outFile << bt.getNumNodes() << endl << endl;

/\*Satisfies requirement e.\*/

outFile << "sumNodes" << endl;

outFile << bt.sumValues();

outFile << endl;

/\*Satisfies requirement f.\*/

outFile << "numLeaves" << endl;

outFile << bt.numLeaves();

outFile << endl;

/\*satisfies requirement g.\*/

outFile << "deepest level value" << endl;

outFile << bt.getDeepestTreeValue();

outFile << endl;

/\*Satisfies requirement h.\*/

outFile << "Number of parents with only one child" << endl;

outFile << bt.numSingleChildren();

outFile << endl;

/\*Satisfies requirement i.\*/

outFile << "SubTree of node value 299 " << endl;

bt.printInOrder(bt.search(299), outFile);

outFile << endl;

/\*Satisfies requirement j.\*/

outFile << "Right subtree of node with value of 253 in postOrder." << endl;

TreeNode \*node = bt.search(253);

if(NULL != node)

{

bt.printPostOrder(node->right, outFile);

}

outFile << endl;

/\*Satisfies requirement k\*/

outFile << "NumNodesLeft ";

outFile << bt.numValsLeft();

outFile << endl;

outFile << "numNodesRight ";

outFile << bt.numValsRight();

outFile << endl;

outFile << "Removing values: ";

/\*Satisfy requirements creep for deltions\*/

for(count = 0; count < 150; count++)

{

val = values[count];

if(sumOfDigits(val) < 10)

{

bt.remove(val);

outFile << val << " ";

}

}

outFile << "After deleting all values with a digit sum less than ten "

<< "there are " << bt.getNumNodes() << " remaining." << endl;

bt.printInOrder(outFile);

outFile << endl;

}

int main()

{

BinaryTree bt;

if(!inFile)

{

cout << "No input file!" << endl;

outFile.close();

return -1;

}

buildTree(bt);

testBinaryTreeFunctions(bt);

inFile.close();

outFile.close();

return 0;

}

/\* Assignment: Tree Work Out

\* Author: Ryan Wood

\* Date Created: 4/14/2017

\* Last Modified: 4/1/2018

\* Requirements: Create a binary tree class that includes the functions

\* specified in Tree Work Out program specification

\*/

/\*Binary Tree implementation file\*/

#include "BinaryTree.h"

#include <iostream>

#include <fstream>

using namespace std;

//private:

//TreeNode \*root;

void BinaryTree::insert(TreeNode \*&nodePtr, TreeNode \*&newNode)

{

if(NULL == nodePtr)

{

nodePtr = newNode;//base case, insert the node

}

else if(newNode->val < nodePtr->val)//case left, call insert on left

{

insert(nodePtr->left, newNode);

}

else//otherwise call insert on node right

{

insert(nodePtr->right, newNode);

}

}

void BinaryTree::destroySubTree(TreeNode \*tn)

{

if(NULL != tn)

{

destroySubTree(tn->left);

destroySubTree(tn->right);

delete(tn);

}

}

//public:

BinaryTree::BinaryTree()

{

root = NULL;

}

TreeNode \*BinaryTree::newNode()

{

TreeNode \*tn = new TreeNode;

tn->left = NULL;

tn->right = NULL;

tn->val = 0;

return tn;

}

TreeNode \*BinaryTree::newNode(int val)

{

TreeNode \*tn = newNode();

tn->val = val;

return tn;

}

void BinaryTree::insert(int val)

{

TreeNode \*node = newNode(val);

insert(root, node);

}

TreeNode \*BinaryTree::remove(TreeNode \*root, int val)

{

TreeNode \*tmp = NULL;

if(root == NULL)

return tmp;

if(val < root->val)// data is in the left sub tree.

root->left = remove(root->left, val);

else if(val > root->val)//data is in the right sub tree.

root->right = remove(root->right, val);

else

{

//case 1: the node has no children

if(root->left == NULL && root->right == NULL)

{

delete root;

root = NULL;

}

//case 2: node has one right child

else if(root->left == NULL)

{

tmp = root;

root = root->right;

delete tmp;

}

//case 3: node has one left child

else if(root->right == NULL)

{

tmp = root;

root = root->left;

delete tmp;

}

//case 4: the node has two children

else

{

//find the maximum of the left subtree

tmp = getRightmostNode(root->left, NULL);

root->val = tmp->val;//duplicate the node

root->right = remove(root->right, tmp->val);//delete the duplicate node

}

}

return root;//parent node can update reference

}

TreeNode \*BinaryTree::remove(int val)

{

return remove(root, val);

}

TreeNode \*BinaryTree::search(TreeNode \*tn, int val)

{

if(NULL == tn)

return tn;//base case 1, never found

else if(tn->val == val)

return tn;//base case 2, found match

else if(val < tn->val)

return search(tn->left, val);

else //if(val > tn->val)

return search(tn->right, val);

}

TreeNode \*BinaryTree::search(int val)

{

return search(root, val);

}

void BinaryTree::printInOrder(TreeNode \*tn, ofstream &out)

{

if(NULL != tn)

{

printInOrder(tn->left, out);

out << tn->val << endl;

printInOrder(tn->right, out);

}

}

void BinaryTree::printPreOrder(TreeNode \*tn, ofstream &out)

{

if(NULL != tn)

{

out << tn->val << endl;

printPreOrder(tn->left, out);

printPreOrder(tn->right, out);

}

}

void BinaryTree::printPostOrder(TreeNode \*tn, ofstream &out)

{

if(NULL != tn)

{

printPostOrder(tn->left, out);

printPostOrder(tn->right, out);

out << tn->val << endl;

}

}

int BinaryTree::getNumNodes(TreeNode \*node)

{

if(NULL == node)

{

return 0;

}

else

{

return 1 + getNumNodes(node->left) + getNumNodes(node->right);

}

}

int BinaryTree::sumValues(TreeNode \*node)

{

if(NULL == node)

{

return 0;

}

else

{

return node->val + sumValues(node->left) + sumValues(node->right);

}

}

int BinaryTree::sumValues()

{

return sumValues(root);

}

int BinaryTree::getNumNodes()

{

return getNumNodes(root);

}

double BinaryTree::getAverage()

{

int sum = sumValues(root);

int count = getNumNodes();

if(count == 0)

{

return 0;

}

else

{

return (double) sum / count;

}

}

TreeNode \*BinaryTree::getLeftmostNode(TreeNode \*node, TreeNode \*prev)

{

if(NULL == node)

return prev;

else

{

prev = node;

return getLeftmostNode(node->left, prev);

}

}

TreeNode \*BinaryTree::getRightmostNode(TreeNode \*node, TreeNode \*prev)

{

if(NULL == node)

return prev;

else

{

prev = node;

return getRightmostNode(node->right, prev);

}

}

int BinaryTree::getDeepestTreeValueLeft(TreeNode \*node, TreeNode \*prev, int &level)

{

if(NULL == node)

{

if(NULL == prev)

{

return 0;

}

else

{

return prev->val;

}

}

else

{

level++;

prev = node;

return getDeepestTreeValueLeft(node->left, prev, level);

}

}

int BinaryTree::getDeepestTreeValueRight(TreeNode \*node, TreeNode \*prev, int &level)

{

if(NULL == node)

{

if(NULL == prev)

{

return 0;

}

else

{

return prev->val;

}

}

else

{

level++;

prev = node;

return getDeepestTreeValueRight(node->right, prev, level);

}

}

int BinaryTree::getDeepestTreeValue()

{

int levelLeft = 0;

int levelRight = 0;

int valLeft = 0;

int valRight = 0;

valLeft = getDeepestTreeValueLeft(root, NULL, levelLeft);

valRight = getDeepestTreeValueRight(root, NULL, levelRight);

if(levelLeft > levelRight)

{

return valLeft;

}

else if(levelRight > levelLeft)

{

return valRight;

}

else

{

return valLeft;//if levels are equal, return one of the values

}

}

int BinaryTree::numValsLeft(TreeNode \*tn)

{

if(NULL != tn && NULL != tn->left)

{

return getNumNodes(tn->left);

}

else

{

return 0;

}

}

int BinaryTree::numValsRight(TreeNode \*tn)

{

if(NULL != tn && NULL != tn->left)

{

return getNumNodes(tn->right);

}

else

{

return 0;

}

}

int BinaryTree::numValsLeft()

{

return numValsLeft(root);

}

int BinaryTree::numValsRight()

{

return numValsRight(root);

}

int BinaryTree::numLeaves(TreeNode \*node)

{

if(NULL == node)

{

return 0;

}

if(node->left == NULL && node->right == NULL)

{

return 1;

}

else

{

return numLeaves(node->left) + numLeaves(node->right);

}

}

int BinaryTree::numLeaves()

{

return numLeaves(root);

}

int BinaryTree::numSingleChildren(TreeNode \*node)

{

if(NULL == node)

{

return 0;

}

else if(NULL == node->left && NULL != node->right)

{

return 1 + numSingleChildren(node->right);

}

else if(NULL == node->right && NULL != node->left)

{

return 1 + numSingleChildren(node->left);

}

else

{

return numSingleChildren(node->left) + numSingleChildren(node->right);

}

}

int BinaryTree::numSingleChildren()

{

return numSingleChildren(root);

}

/\* Assignment: Tree Work Out

\* Author: Ryan Wood

\* Date Created: 4/14/2017

\* Last Modified: 4/1/2018

\* Requirements: Create a binary tree class that includes the functions

\* specified in Tree Work Out program specification

\*/

/\*Binary Tree Header file\*/

#include <iostream>

#include <fstream>

using namespace std;

/\*TreeNode structure. One value. One left leaf and on right leaf\*/

struct TreeNode

{

int val;

TreeNode \*left;

TreeNode \*right;

};

class BinaryTree

{

private:

TreeNode \*root;

/\* Function: insert(TreeNode\*&, TreeNode \*&)

\* Purpos: inserts the given new node to the left

\* or to the right of the given node.

\* Recursive function, calls itself until

\* it finds a null node left or right

\* Parameters: the node after which the new node should be placed,

\* the new node with a value and null left and right nodes

\*/

void insert(TreeNode \*&node, TreeNode \*&newNode);

/\* Function: getLeftmostNode(TreeNode\*, TreeNode\*)

Purpose: For the given node, this function determines which

node of the subtree has no left link, the final

left value in the subtree

Parameters: the node, the parent of the node if it is to be considered

or a NULL TreeeNode

Return: the leftmost node of the subtree

\*/

TreeNode \*getLeftmostNode(TreeNode \*tn, TreeNode \*prev);

/\* Function: getRightmostNode(TreeNode\*, TreeNode\*)

Purpose: for the given subtree, this function determines

what is the node with the largest value, in

other words the rightmost node in the subtree

Parameters: the node, the parent if it is to be considered

or a NULL TreeNode

Return: The rightmost node of the subtree

\*/

TreeNode \*getRightmostNode(TreeNode \*tn, TreeNode \*prev);

/\* Function: remove(TreeNode\*, int val)

Purpose: for the given node (subtree), searches the tree for the given

value and removes that node from the tree

Parameters: the node, the value

Return: a copy of the node that is replaced

\*/

TreeNode \*remove(TreeNode\* tn, int val);

/\* Function: destroySubTree(TreeNode\*)

\* Purpose: deletes all members of the tree from the given node

\* through all of its children. Recursive

\* Parameters: the sub tree root node

\*/

void destroySubTree(TreeNode \*tn);

/\* Function: numValsLeft(TreeNode\*)

\* Purpose: calculates the number of values to the left

\* of the given subtree node. Recursive

\* Parameters: the subtree root node

\* Return: the number of values in the tree's left subtree

\*/

int numValsLeft(TreeNode \*tn);

/\* Function: numValsRight(TreeNode\*)

\* Purpose: calculates the number of values to the right of

\* the given subree node. Recursive

\* Parameters: the subtree root node

\* Return: the number of values in the tree's right subtree

\*/

int numValsRight(TreeNode \*tn);

/\* Function: getNumNodes(TreeNode\*)

\* Purpose: calculates the number of nodes in the

\* given subtree. Recursive

\* Parameters: the subtree root node

\* Return: The number of nodes of the subtree

\*/

int getNumNodes(TreeNode \*tn);

/\* Function: sumValues(TreeNode\*)

\* Purpose: Adds up all the values in the

\* given subtree. Recursive

\* Parameters: the root node of the sub tree

\* Return: the sum of the values in the sub tree

\*/

int sumValues(TreeNode \*tn);

/\* Function: getAverage(TreeNode\*)

\* Purpose: calculates the average of all the values

\* in the given subtree

\* Parameters: the subtree root node

\* Return: the average value in the subtree

\*/

double getAverage(TreeNode \*tn);

/\* Function: \*search(TreeNode\*, int)TreeNode

\* Purpose: searches the given subtree for the

\* given value recursively. Gives back

\* the node having the value if it is found.

\* Parameters: the root node of the sub tree

\* Return: the node having the value or NULL if not found

\*/

TreeNode \*search(TreeNode \*tn, int val);

/\* Function: getDeepestTreeValueLeft(TreeNode\*, TreeNode\*, int&)

\* Purpose: Determines the value on the given subtree's left

\* subtree that is at the deepest level of the subtree.

\* Sets the given level value to the level where it finds

\* this value. Recursive

\* Parameters: the root node of the subtree,

\*. the previous root node (or NULL),

\* the level value, starting at zero

\* Return: the value at the deepest level of the subtree

\*/

int getDeepestTreeValueLeft(TreeNode \*tn, TreeNode \*prev, int &level);

/\* Function: getDeepestTreeValueRight(TreeNode\*, TreeNode\*, int&)

\* Purpose: Determines the value on the given subtree's right

\* subtree that is at the deepest level of the subtree.

\* Sets the given level value to the level where it finds

\* this value. Recursive

\* Parameters: the root node of the subtree,

\* the previous root node (or NULL),

\* the level value, starting at zero

\* Return: the value at the deepest level of the subtree

\*/

int getDeepestTreeValueRight(TreeNode \*tn, TreeNode \*prev, int &level);

/\* Function: numLeaves(TreeNode\*)

\* Purpose: gets the number of leaves of the subtree. Recursive

\* Parameters: the root node of the subtree

\* Return: the number of nodes in the subtree

\*/

int numLeaves(TreeNode \*node);

/\* Function: numSingleChildren(TreeNode\*)

\* Purpose: determines the number of parents in the tree that

\* have only one child. In other words, determines the

\* number of nodes in the sub tree having no siblings

\* Recursive function

\* Parameters: the root node of the sub tree

\* Return: the number of children with no siblings

\*/

int numSingleChildren(TreeNode \*node);

public:

/\* Function: BinaryTree()

\* Purpose: default constructor of the BinaryTree class.

\* Sets the root of the tree to NULL

\*/

BinaryTree();

/\* Function: ~BinaryTree()

\* Purpose: destructor for the BinaryTree class.

\* Simply calls destroySubTree on the tree root

\*/

~BinaryTree()

{destroySubTree(root);}

/\* Function: \*newNode()

\* Purpose: creates a new node with NULL left and right

\* leaves and a zero value.

\* Return: an empty TreeNode pointer

\*/

TreeNode \*newNode();

/\* Function: \*newNode(int)

\* Purpose: creates a new node with NULL left and right

\* leaves and the given integer for the value.

\* Parameters: the value

\* Return: a TreeNode pointer having the given value

\*/

TreeNode \*newNode(int val);

/\* Function: insert(int)

\* Purpose: creates a new node in the tree with the given

\* value and inserts it into the appropriate location

\* of the tree.

\* Parameters: the value

\*/

void insert(int val);

/\* Function: remove(int)

Purpose: finds the given value in the tree and removes that

node, shifting values as necessary

Parameters: the value

Return: a copy of the replaced node

\*/

TreeNode \*remove(int val);

/\* Function: \*search(int)

\* Purpose: finds the node in the tree having the given value

\* and returns the node if the value is found.

\* Parameters: the value

\* Return: The node having the value or NULL if not found

\*/

TreeNode \*search(int val);

/\* Function: printInOrder(TreeNode\*, oftream&)

\* Purpose: prints the given subtree using the given

\* reference to an output stream in order

\* from smallest to largest value

\* Parameters: the root node of the sub tree,

\* a reference to an output stream

\*/

void printInOrder(TreeNode \*tn, ofstream &out);

/\* Function: printInOrder(oftream&)

\* Purpose: prints the tree using the given

\* reference to an output stream in order

\* from smallest to largest value

\* Parameters: a reference to an output stream

\*/

void printInOrder(ofstream &out)

{printInOrder(root, out);}

/\* Function: printPreOrder(TreeNode\*, oftream&)

\* Purpose: prints the given subtree using the given

\* reference to an output stream

\* in preOrder form

\* Parameters: the root node of the sub tree,

\* a reference to an output stream

\*/

void printPreOrder(TreeNode \*tn, ofstream &out);

/\* Function: printPreOrder(oftream&)

\* Purpose: prints the tree using the given

\* reference to an output stream

\* in PreOrder form

\* Parameters: a reference to an output stream

\*/

void printPreOrder(ofstream &out)

{printPreOrder(root, out);}

/\* Function: printPostOrder(TreeNode\*, oftream&)

\* Purpose: prints the given subtree using the given

\* reference to an output stream

\* in postOrder form

\* Parameters: the root node of the sub tree,

\* a reference to an output stream

\*/

void printPostOrder(TreeNode \*tn, ofstream &out);

/\* Function: printPostOrder(oftream&)

\* Purpose: prints the tree using the given

\* reference to an output stream

\* in PostOrder form

\* Parameters: a reference to an output stream

\*/

void printPostOrder(ofstream &out)

{printPostOrder(root, out);}

/\* Function: getNumNdes()

\* Purpose: determines the number of nodes in the tree

\* Return: the number of nodes

\*/

int getNumNodes();

/\* Function: sumValues()

\* Purpose: adds up the values of the tree

\* Return: the sum of all values in the tree

\*/

int sumValues();

/\* Function: numLeaves()

\* Purpose: determines the number of nodes in the tree

\* that have no children (leaves)

\* Return: the number of nodes that are leaves

\*/

int numLeaves();

/\* Function: getAverage()

\* Purpose: determines the average of the values in the tree

\* Return: The average value of the tree

\*/

double getAverage();

/\* Function: numValsLeft()

\* Purpose: Determines the number of values that are on the left

\* side of the tree

\* Return: The number of values on the left side of the tree

\*/

int numValsLeft();

/\* Function: numValsRight()

\* Purpose: determines the number of values that are to the right

\* side of the tree

\* Return: The number of nodes on the right

\*/

int numValsRight();

/\* Function: getDeepestTreeValue()

\* Purpose: determines the value that is at the deepest

\* level of the tree. If there are multiple values

\* at the deepest level, the one on the left is returned

\*/

int getDeepestTreeValue();

/\* Function: numSingleChildren()

\* Purpose: determines the number of nodes in the tree that

\* have no siblings or the number of parents

\* that have only one child

\* Return: the number of nodes with no sibling

\*/

int numSingleChildren();

};

OUTPUTS CONVERTED TO MS WORD TABLES

printInOrder

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 100 | 102 | 107 | 116 | 117 | 123 | 127 | 130 | 133 | 134 | 136 | 138 | 139 | 140 | 141 |
| 142 | 147 | 150 | 151 | 153 | 156 | 157 | 159 | 161 | 165 | 169 | 170 | 171 | 173 | 179 |
| 187 | 193 | 195 | 196 | 208 | 212 | 218 | 222 | 223 | 224 | 226 | 236 | 237 | 238 | 239 |
| 240 | 245 | 247 | 248 | 253 | 254 | 256 | 257 | 261 | 264 | 266 | 271 | 272 | 273 | 274 |
| 276 | 277 | 281 | 283 | 284 | 286 | 291 | 294 | 296 | 299 | 304 | 308 | 313 | 320 | 327 |
| 328 | 332 | 334 | 336 | 337 | 340 | 342 | 345 | 349 | 350 | 355 | 359 | 362 | 366 | 369 |
| 372 | 373 | 374 | 379 | 380 | 382 | 387 | 388 | 391 | 392 | 394 | 395 | 402 | 403 | 405 |
| 406 | 408 | 410 | 412 | 418 | 421 | 426 | 427 | 428 | 431 | 433 | 435 | 436 | 441 | 447 |
| 448 | 450 | 452 | 453 | 457 | 459 | 464 | 468 | 471 | 474 | 485 | 488 | 491 | 492 | 495 |
| 498 | 500 | 505 | 506 | 508 | 511 | 512 | 520 | 524 | 529 | 530 | 531 | 541 | 544 | 546 |

printPreOrder

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 141 | 117 | 116 | 107 | 100 | 102 | 134 | 123 | 130 | 127 | 133 | 136 | 140 | 139 | 138 |
| 500 | 369 | 328 | 208 | 161 | 147 | 142 | 150 | 151 | 156 | 153 | 157 | 159 | 195 | 169 |
| 165 | 179 | 170 | 173 | 171 | 193 | 187 | 196 | 264 | 253 | 226 | 212 | 223 | 218 | 222 |
| 224 | 240 | 237 | 236 | 239 | 238 | 248 | 247 | 245 | 256 | 254 | 261 | 257 | 277 | 273 |
| 266 | 272 | 271 | 276 | 274 | 304 | 299 | 283 | 281 | 294 | 286 | 284 | 291 | 296 | 308 |
| 327 | 313 | 320 | 345 | 342 | 332 | 337 | 334 | 336 | 340 | 366 | 362 | 359 | 355 | 350 |
| 349 | 405 | 395 | 392 | 388 | 372 | 379 | 373 | 374 | 382 | 380 | 387 | 391 | 394 | 402 |
| 403 | 431 | 427 | 421 | 406 | 412 | 408 | 410 | 418 | 426 | 428 | 468 | 453 | 447 | 435 |
| 433 | 441 | 436 | 452 | 450 | 448 | 457 | 464 | 459 | 471 | 485 | 474 | 492 | 488 | 491 |
| 498 | 495 | 524 | 512 | 511 | 506 | 505 | 508 | 520 | 541 | 529 | 531 | 530 | 544 | 546 |

printPostOrder

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 102 | 100 | 107 | 116 | 127 | 133 | 130 | 123 | 138 | 139 | 140 | 136 | 134 | 117 | 142 |
| 153 | 159 | 157 | 156 | 151 | 150 | 147 | 165 | 171 | 173 | 170 | 187 | 193 | 179 | 169 |
| 196 | 195 | 161 | 222 | 218 | 224 | 223 | 212 | 236 | 238 | 239 | 237 | 245 | 247 | 248 |
| 240 | 226 | 254 | 257 | 261 | 256 | 253 | 271 | 272 | 266 | 274 | 276 | 273 | 281 | 284 |
| 291 | 286 | 296 | 294 | 283 | 299 | 320 | 313 | 327 | 308 | 304 | 277 | 264 | 208 | 336 |
| 334 | 340 | 337 | 332 | 342 | 349 | 350 | 355 | 359 | 362 | 366 | 345 | 328 | 374 | 373 |
| 380 | 387 | 382 | 379 | 372 | 391 | 388 | 394 | 392 | 403 | 402 | 395 | 410 | 408 | 418 |
| 412 | 406 | 426 | 421 | 428 | 427 | 433 | 436 | 441 | 435 | 448 | 450 | 452 | 447 | 459 |
| 464 | 457 | 453 | 474 | 491 | 488 | 495 | 498 | 492 | 485 | 471 | 468 | 431 | 405 | 369 |
| 505 | 508 | 506 | 511 | 520 | 512 | 530 | 531 | 529 | 546 | 544 | 541 | 524 | 500 | 141 |

numNodes

150

sumNodes

48112

numLeaves

50

deepest level value

546

Number of parents with only one child

51

SubTree of node value 299

281

283

284

286

291

294

296

299

Right subtree of node with value of 253 in postOrder.

254

257

261

256

NumNodesLeft 14

numNodesRight 135

Removing values: 141 117 134 500 512 405 431 161 342 304 402 332 421 212 511 223 240 142 140 170 150 151 123 116 531 441 130 171 520 412 340 261 450 350 224 107 403 133 222 530 313 100 410 102 320 153 After deleting all values with a digit sum less than ten there are 124 remaining.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 116 | 116 | 127 | 127 | 133 | 133 | 136 | 138 | 139 | 140 | 140 | 147 | 156 | 157 | 159 |
| 159 | 165 | 169 | 173 | 179 | 187 | 193 | 195 | 196 | 208 | 218 | 222 | 226 | 236 | 237 |
| 238 | 239 | 239 | 245 | 247 | 248 | 253 | 254 | 256 | 257 | 264 | 266 | 271 | 272 | 273 |
| 274 | 276 | 277 | 281 | 283 | 284 | 286 | 291 | 294 | 296 | 299 | 299 | 308 | 327 | 328 |
| 334 | 336 | 337 | 345 | 349 | 355 | 359 | 362 | 366 | 369 | 372 | 373 | 374 | 379 | 380 |
| 382 | 387 | 388 | 391 | 392 | 394 | 395 | 403 | 403 | 406 | 408 | 410 | 410 | 418 | 418 |
| 426 | 427 | 428 | 428 | 433 | 435 | 436 | 447 | 448 | 452 | 453 | 457 | 459 | 464 | 468 |
| 471 | 474 | 485 | 488 | 491 | 492 | 495 | 498 | 498 | 505 | 506 | 508 | 511 | 511 | 524 |
| 529 | 541 | 544 | 546 |  |  |  |  |  |  |  |  |  |  |  |