Congratulations! You passed!

TO PASS 80% or higher

Keep Learning

GRADE
100%

Week 3 Challenge

LATEST SUBMISSION GRADE
100%

1. **Implement a function that computes the height of each node in a binary tree and stores it in each node of the tree. Recall that the height of a node is the number of edges in its longest chain of descendants.**

   Consider a tree with root node A that has two child nodes B1 and B2. Let node B1 have no children. Let node B2 have one child C. Then the height of A would be two, because its longest chain of descendants (A -> B2 -> C) has two edges (A -> B2 and B2 -> C).

   The starter code below defines a class called "Node" that has two child pointers ("left" , "right") and an integer "height" member variable. There is also a constructor Node() that initializes the children to nullptr and the height to -1.

   Your job is to implement the procedure "computeHeight(Node *n)" that computes the height of the node n as well as the height of its children (if any).

   There is also a helper function "printTree(const Node *n)" that prints the current heights showing the tree as embedded parentheses. If a child is nullptr, then it will appear as an empty pair of parentheses: "()". The constructor initializes the height to -1 even though a node with no children should have a height of zero. If you see any -1 entries after you've run your computeHeight() procedure, you may have missed one or more nodes.

   The helper function printTreeVertical(const Node *n) is also available to you (although its complex definition is not shown). It displays a verbose, vertical printout of your tree, where the root is shown at the top, and left children are shown on higher rows than right children.

   5 / 5 points

```
/*
The height of a node is the number of edges in
its longest chain of descendants.

Implement computeHeight to compute the height
of the subtree rooted at the node n. Note that
this function does not return a value. You should
store the calculated height in that node's own
height member variable. Your function should also
do the same for EVERY node in the subtree rooted
at the current node. (This naturally lends itself
to a recursive solution!)

Assume that the following includes have already been
provided. You should not need any other includes
than these.

#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <string>

You have also the following class Node already defined.
You cannot change this class definition, so it is
shown here in a comment for your reference only:

class Node {
public:
  int height; // to be set by computeHeight()
  Node *left, *right;
  Node() { height = -1; left = right = nullptr; }
  ~Node() {
    delete left;
    left = nullptr;
    delete right;
    right = nullptr;
  }
};
*/

void computeHeight(Node *n) {
  if( nullptr == n )
  {
    //std::cout << "null, empty" << std::endl;

    return;
  }
  else
  {

    //std::cout << "calcing left sub tree" << std::endl;

    Node *leftNode = n->left;
    computeHeight( leftNode );

    //std::cout << "calcing right sub tree" << std::endl;

    Node *rightNode = n->right;
    computeHeight( rightNode );



    if( nullptr == leftNode && nullptr == rightNode )
    {
      // current node is leave node, height = 0
      n->height = 0;
    }
    else if( nullptr == leftNode && nullptr != rightNode )
    {
      //current node adds right node's height
      n->height = 1 + rightNode->height;
    }
    else if( nullptr == rightNode && nullptr != leftNode )
    {
      //current node adds left node's height
      n->height = 1 + leftNode->height;
```

```
    }
    else
    {
      //current node has two sub-tree
      //adds the one with longer height

      if( leftNode->height >= rightNode->height )
      {
        n->height = 1 + leftNode->height;
      }
      else
      {
        n->height = 1 + rightNode->height;
      }
    }


    return;
  }
  //end of if( nullptr == n ) ... else ...

}
//end of function computeHeight

// This function prints the tree in a nested linear format.
void printTree(const Node *n) {
  if (!n) return;
  std::cout << n->height << "(";
  printTree(n->left);
  std::cout << ")(";
  printTree(n->right);
  std::cout << ")";
}

// The printTreeVertical function gives you a verbose,
// vertical printout of the tree, where the leftmost nodes
// are displayed highest. This function has already been
// defined in some hidden code.
// It has this function prototype: void printTreeVertical(const Node* n);

// This main() function is for your personal testing with
// the Run button. When you're ready, click Submit to have
// your work tested and graded.
int main() {
  Node *n = new Node();
  n->left = new Node();
  n->right = new Node();
  n->right->left = new Node();
  n->right->right = new Node();
  n->right->right->right = new Node();

  computeHeight(n);

  printTree(n);
  std::cout << std::endl << std::endl;
  printTreeVertical(n);

  // The Node destructor will recursively
  // delete its children nodes.
  delete n;
  n = nullptr;

  return 0;
}
```

Run

Reset

Correct

[[Note] As a temporary fix for properly reporting the unit test's tree in the server output here, each newline will begin with a "#" symbol, which you can replace with a line break in your text editor for viewing. Padding spaces are shown as "." instead. Thanks for your patience! We'll improve this output soon.]

```
##Height: 5
#|
#|_ Height: 4
#|..|
#|..|_ [null]
#|..|
#|..|_ Height: 3
#|.....|
#|.....|_ Height: 2
#|.....|..|
#|.....|..|_ Height: 1
#|.....|..|..|
#|.....|..|..|_ Height: 0
#|.....|..|..|..|
#|.....|..|..|..|_ [null]
#|.....|..|..|..|
#|.....|..|..|..|_ [null]
#|.....|..|..|
#|.....|..|..|_ [null]
#|.....|..|
#|.....|..|_ [null]
#|.....|
#|.....|_ Height: 2
#|.....|
#|.....|_ Height: 1
#|.....|..|
#|.....|..|_ [null]
#|.....|..|
#|.....|..|_ Height: 0
#|.....|.....|
#|.....|.....|_ [null]
#|.....|.....|
#|.....|.....|_ [null]
#|.....|
#|.....|_ [null]
#|
#|_ Height: 1
#...|
#  |_ Height: 0
```

```
#...|..|
#...|..|_ [null]
#...|..|
#...|..|_ [null]
#...|
#...|_ [null]
```