cs124 Chapter 8 Trees Assignment

put your name here

April 11, 2012

Contents

Use AlDraTex to draw the tree diagrams.

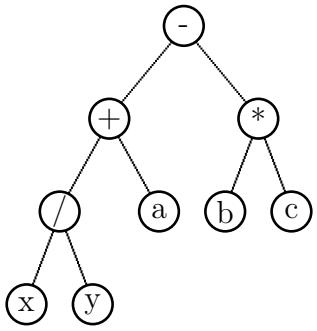
If your Latex system does not automatically load that package, get it here:
DraTex and put the files in the same directory with your tex file.

1 8.1 Tree Applications

1.1 Self-Check exercise 1

1. Draw binary expression trees for the following infix expressions. Your trees should enforce the C++ rules for operator evaluation (higher-precedence operators before lower-precedence operators and left associativity).
   a. `x / y + a - b * c`
   b. `(x * a) - y / b * (c + d)`

1.1.1 a.



1.1.2 b.

1.2 Self-Check exercise 2

2. Using the Huffman tree in Figure 8.5,
   a. Write the binary string for the message "scissors cuts paper".
   b. Decode the following binary string:
   110001000101000100101110110001111111000110101011110110100 1

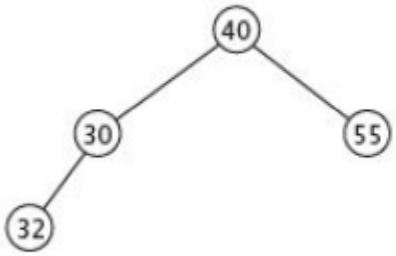
1.2.1 a.

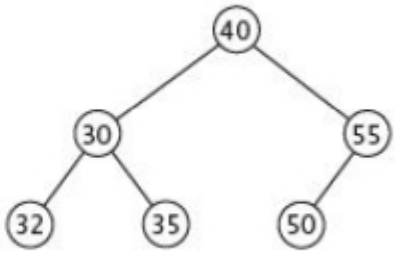
1.2.2 b.

2 8.2 Tree Traversals

2.1 Self-Check exercise 1

1. For the following trees:





If visiting a node displays the integer value stored, show the inorder, preorder, and postorder traversal of each tree.

2.2 Self-Check exercise 2

2. Draw an expression tree corresponding to each of the following:
   a. Inorder traversal is `x / y + 3 * b / c` (Your tree should represent the C++ meaning of the expression.)
   b. Postorder traversal is `x y z + a b - c * / -`
   c. Preorder traversal is `* + a - x y / c d`

2.2.1 a.

2.2.2 b.

2.2.3 c.

3 8.3 Binary Tree

3.1 Self-Check exercise 2

2. Show the tree that would be built by the following data lines:

```
30
15
4
NULL
NULL
20
18
NULL
19
NULL
NULL
NULL
35
32
NULL
NULL
38
NULL
NULL
```

3.2 Self-Check exercise 4

4. Write the strings that would be displayed for the two binary trees in Figure 8.6.

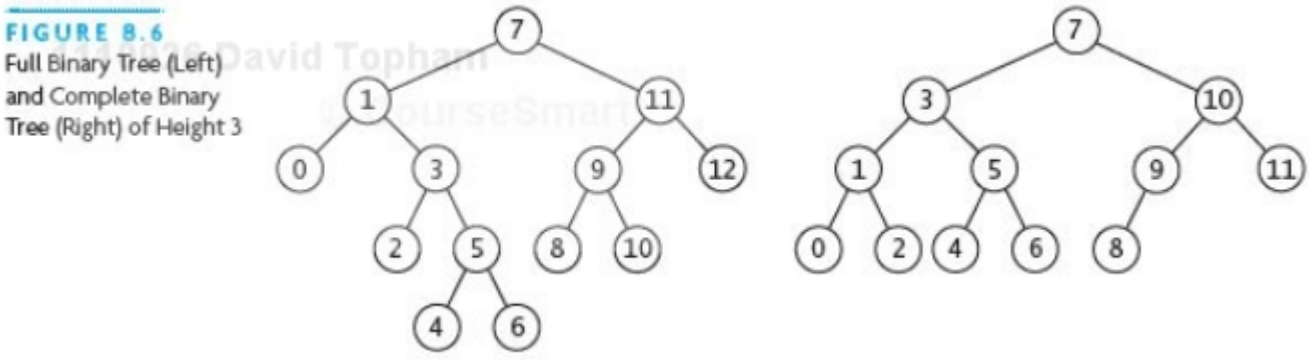


FIGURE 8.6
Full Binary Tree (Left) and Complete Binary Tree (Right) of Height 3

PROGRAMMING

1. Write a function for the `Binary_Tree` class that returns the preorder traversal of a binary tree as a sequence of strings each separated by a space.
2. Write a function to display the postorder traversal of a binary tree in the same form as Programming Exercise 1.
3. Write a recursive member function to find the height of a `Binary_Tree`.
4. Write a recursive member function to find the number of nodes in a `Binary_Tree`.

Binary Tree A Binary Tree Node has data and 2 child node pointers. the `to_string` method simplifies displaying the data and overloading the insertion operator makes it easy to use.

BTNode.h

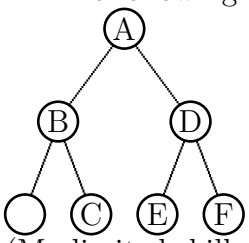
```
#pragma once
#include <sstream>
template<typename Item_Type>
struct BTNode
{
  Item_Type data;
  BTNode<Item_Type>* left;
  BTNode<Item_Type>* right;
  BTNode(const Item_Type& the_data,
         BTNode<Item_Type>* left_val = NULL,
         BTNode<Item_Type>* right_val = NULL) :
    data(the_data), left(left_val),  right(right_val) {}
  virtual ~BTNode() {}
  virtual std::string to_string() const {
    std::ostringstream os;
    os << data;
    return os.str();
  }
};
template<typename Item_Type>
std::ostream& operator<<(std::ostream& out,
                         const BTNode<Item_Type>& node) {
  return out << node.to_string();
}
```

Program to test the binary tree. The program requires input in a specified format to describe the tree. It them calls each of the traversal methods.
Once the assignment is complete each of those commented lines can be activated.

Test_Binary_Tree.cpp

```
#include <string>
#include <iostream>
#include <fstream>
#include "Binary_Tree.h"
#include "pre_order_traversal.h"
#include "post_order_traversal.h"
#include "in_order_traversal.h"
using namespace std;
int main(int argc, char* argv[]) {
  if (argc < 2) {
    cerr << "Usage Test_Binary_Tree <input file>\n";
    return 1;
  }
  ifstream in(argv[1]);
  if (!in) {
    cerr << "Unable to open " << argv[1] << " for input\n";
    return 1;
  }
  Binary_Tree<string> the_tree;
  in >> the_tree;
  cout << "Pre-order traversal\n";
  pre_order_traversal(the_tree, cout, 0);
  //  cout << the_tree.pre_order() << endl;
  cout << "Post-order traversal\n";
  post_order_traversal(the_tree, cout, 0);
  //  cout << the_tree.post_order() << endl;
  cout << "In-order traversal\n";
  in_order_traversal(the_tree, cout, 0);
  //  cout << the_tree.in_order() << endl;
  cout << "String representation\n";
  cout << the_tree << endl;
  //  cout << "The height of the tree is " << the_tree.height() << endl;
  //  cout << "The tree has " << the_tree.number_of_nodes() << " nodes\n";
  return 0;
}
```

The following is a sample input file. The data corresponds to this binary tree:

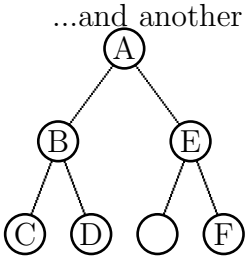


(My limited skills using DraTex caused me to draw an empty left child for B even though it really should just be a NULL with no line or circle!)

BinaryTree1.txt

```
A
B
NULL
C
NULL
NULL
D
E
NULL
NULL
F
NULL
NULL
```

...and another one:



BinaryTree2.txt

```
A
B
C
NULL
NULL
D
NULL
NULL
E
NULL
F
NULL
NULL
```

The binary tree class uses the BTNode to organize the tree structure with its left and right children.

Binary_Tree.h

```
#pragma once
#include <cstddef>
#include <sstream>
#include <stdexcept>
#include <string>
#include <algorithm>
#include "BTNode.h"

template<typename Item_Type>
 class Binary_Tree
{
public:
 Binary_Tree() : root(NULL) {}
 Binary_Tree(const Item_Type& the_data,
        const Binary_Tree<Item_Type>& left_child  = Binary_Tree(),
        const Binary_Tree<Item_Type>& right_child  = Binary_Tree())
  :  root(new BTNode<Item_Type>(the_data, left_child.root, right_child.root))
   {  /* empty body */ }
  virtual ~Binary_Tree() {  /* empty body */ }
  Binary_Tree<Item_Type> get_left_subtree() const;
  Binary_Tree<Item_Type> get_right_subtree() const;
  const Item_Type& get_data() const;
  bool is_null() const;
  bool is_leaf() const;
  virtual std::string to_string() const;
  static Binary_Tree<Item_Type> read_binary_tree(std::istream& in);
  std::string root_to_string() const { return root->to_string(); }
protected:
  Binary_Tree(BTNode<Item_Type>* new_root) : root(new_root) {}
  BTNode<Item_Type>* root;
};
Overloading the stream operators
Implementation of member functions
```

The stream operators facilitatate reading and writing tree structures.

Overloading the stream operators

```
template<typename Item_Type>
  std::ostream& operator<<(std::ostream& out,
                           const Binary_Tree<Item_Type>& tree)
{
  return out << tree.to_string();
}

// Overloading the istream extraction operator
template<typename Item_Type>
  std::istream& operator>>(std::istream& in,
                           Binary_Tree<Item_Type>& tree)
{
  tree = Binary_Tree<Item_Type>::read_binary_tree(in);
  return in;
}
```

Implementation of member functions

```
template<typename Item_Type>
Binary_Tree<Item_Type>  Binary_Tree<Item_Type>::get_left_subtree() const
{
  if (root == NULL) {
    throw std::invalid_argument("get_left_subtree on empty tree");
  }
  return Binary_Tree<Item_Type>(root->left);
}

template<typename Item_Type>
Binary_Tree<Item_Type> Binary_Tree<Item_Type>::get_right_subtree() const
{
  if (root == NULL) {
    throw std::invalid_argument("get_right_subtree on null tree");
  }
  return Binary_Tree<Item_Type>(root->right);
}

template<typename Item_Type>
const Item_Type& Binary_Tree<Item_Type>::get_data() const {
  if (root == NULL) {
    throw std::invalid_argument("get_data on null tree");
  }
  return root->data;
}

template<typename Item_Type>
bool Binary_Tree<Item_Type>::is_null() const {
  return root == NULL;
}

template<typename Item_Type>
  bool Binary_Tree<Item_Type>::is_leaf() const {
  if (root != NULL) {
    return root->left == NULL && root->right == NULL;
  } else
    return true;
}
```

convert binary tree to a string

read binary tree from input

This code makes it easy to display the tree by returning it formatted as a string.

convert binary tree to a string

```
template<typename Item_Type>
std::string Binary_Tree<Item_Type>::to_string() const {
  std::ostringstream os;
  if (is_null())
    os << "NULL\n";
  else {
    os << *root << '\n';
    os << get_left_subtree().to_string();
    os << get_right_subtree().to_string();
  }
  return os.str();
}
```

This code reads the input data file to store the tree info in the object.

read binary tree from input

```
template<typename Item_Type>
  Binary_Tree<Item_Type> Binary_Tree<Item_Type>::
    read_binary_tree(std::istream& in) {
  std::string next_line;
  getline(in, next_line);
  if (next_line == "NULL") {
    return Binary_Tree<Item_Type>();
  } else {
    Item_Type the_data;
    std::istringstream ins(next_line);
    ins >> the_data;
    Binary_Tree<Item_Type> left = read_binary_tree(in);
    Binary_Tree<Item_Type> right = read_binary_tree(in);
    return Binary_Tree<Item_Type>(the_data, left, right);
  }
}
```

pre-order traversal means to process the parent node before processing its children which are then done in left to right order.

pre_order_traversal.h

```
#ifndef PRE_ORDER_TRAVERSAL_H
#define PRE_ORDER_TRAVERSAL_H

#include "Binary_Tree.h"
#include <ostream>

template<typename Item_Type>
void pre_order_traversal(const Binary_Tree<Item_Type>& the_tree,
       std::ostream& out, int level)
{
  if (the_tree.is_null()) {
    for (int i = 0; i < level; i++)
      out << "  ";
    out << "null\n";
  }
  else {
    for (int i = 0; i < level; i++)
      out << "  ";
    out << the_tree.get_data() << std::endl;
    pre_order_traversal(the_tree.get_left_subtree(), out, level + 1);
    pre_order_traversal(the_tree.get_right_subtree(), out, level + 1);
  }
}

#endif
```

For post-order traversal, the parent (or root) node is processed after the children.

post_order_traversal.h

```
#ifndef POST_ORDER_TRAVERSAL_H
#define POST_ORDER_TRAVERSAL_H

#include "Binary_Tree.h"
#include <ostream>

template<typename Item_Type>
void post_order_traversal(const Binary_Tree<Item_Type>& the_tree,
       std::ostream& out, int level)
{
  if (the_tree.is_null()) {
    for (int i = 0; i < level; i++)
      out << "  ";
    out << "null\n";
  }
  else {
    post_order_traversal(the_tree.get_left_subtree(), out, level + 1);
    post_order_traversal(the_tree.get_right_subtree(), out, level + 1);
    for (int i = 0; i < level; i++)
      out << "  ";
    out << the_tree.get_data() << std::endl;
  }
}

#endif
```

An in-order traversal processes left child, then parent, then right child.

in_order_traversal.h

```
#ifndef IN_ORDER_TRAVERSAL_H
#define IN_ORDER_TRAVERSAL_H

#include "Binary_Tree.h"
#include <ostream>

template<typename Item_Type>
void in_order_traversal(const Binary_Tree<Item_Type>& the_tree,
       std::ostream& out, int level)
{
  if (the_tree.is_null()) {
    for (int i = 0; i < level; i++)
      out << "  ";
    out << "null\n";
  }
  else {
    in_order_traversal(the_tree.get_left_subtree(), out, level + 1);
    for (int i = 0; i < level; i++)
      out << "  ";
    out << the_tree.get_data() << std::endl;
    in_order_traversal(the_tree.get_right_subtree(), out, level + 1);
  }
}

#endif
```

3.3 Programming exercise 1

e.g. BinaryTree1.txt would output this string: A B C D E F

preorder

```
template<typename Item_Type>
void pre_order(const Binary_Tree<Item_Type>& the_tree,
       std::ostream& out, int level)
{
  if (the_tree.is_null()) {
    for (int i = 0; i < level; i++)
      out << "  ";
    out << "null\n";
  }
  else {
    pre_order(the_tree.get_left_subtree(), out, level + 1);
    for (int i = 0; i < level; i++)
      out << "  ";
    out << the_tree.get_data() << std::endl;
    pre_order(the_tree.get_left_subtree(), out, level + 1);
    pre_order(the_tree.get_right_subtree(), out, level + 1);
  }
}
```

3.4 Programming exercise 2

e.g. BinaryTree1.txt would output this string: C B E F D A

3.5 Programming exercise 3

e.g. BinaryTree1.txt would output this: The height of the tree is 3

3.6 Programming exercise 4

e.g. BinaryTree1.txt would output this: The tree has 6 nodes