Tutorial: Drawing a Binary Tree

In Chapter 10, we include the function displayTree() in the library "d_tnode.h". The console-based function outputs the nodes of a tree vertically. The tnode<T> argument is the root of the binary tree and the argument maxCharacters is the length of the longest string that represents a value of a node.

```
displayTree():
    // provides a vertical console based display of a binary tree; output
    // of a node value requires no more than maxCharacters
    template <typename T>
    void displayTree(tnode<T> *t, int maxCharacters);
```

As a supplement to the Ford-Topp text, "Data Structures with C++", this tutorial introduces the function drawTree() that outputs the nodes of a tree in a graphics window. Nodes are displayed as shaded circles with the value in the center and lines representing the edges. The function has the same argument list as displayTree() and is found in "d_tdraw.h".

```
drawTree()
    // displays a tree in a graphical window
    template <typename T>
    void drawTree(tnode<T> t, int maxCharacters);
```

In the stree class, we use a modified version of displayTree() as a member function. This tutorial introduces an extended version of the stree class that includes the member functions drawTree() and drawTrees().

Let us begin by introducing algorithm for the the design and implementation of functions that draw a tree vertically. The same algorithm is used by both displayTree() and drawTree(). Since the latter is more interesting, we use graphical drawing as the basis for the tutorial.

The Display Algorithm

A typical representation of a binary tree uses circles and lines to denote the nodes and connecting paths between the nodes. In this tutorial, we design an algorithm for the function drawTree() that displays trees programmatically on a graphics window. The discussion includes a recursive function the builds a shadow tree which is a clone of the original tree but with nodes that contain additional information. To illustrate drawTree(), we develop an application that displays the character trees, Tree 0, Tree 1, and Tree 2. As we develop the algorithms for drawTree(), we includes a listing of the code. The listing contains format details that allow us to label the nodes with numeric, character, or string data.

The algorithm for drawTree() uses the graphics library to draw nodes and connecting lines for the tree in a window. View the drawing surface as a square grid with a cell denoted by the pair (level, col). The level is a row in the grid corresponding to the level of the node relative to the root. Figure 1 is the representation of Tree 0 in the grid.
The algorithm for drawTree is a two step process. The first step uses the recursive function buildShadowTree() to make an inorder scan of the tree which creates a second tree, called a shadow tree. The nodes of the shadow tree store the value of the node in the original tree formatted as a string and the (level, col) position of the shadow tree node in the window. The second step uses a level order of shadow tree to physically draw the nodes on the drawing surface. In this step, we convert the (level, col) pair for a cell to (x, y) coordinates on the screen. Using the (x, y) coordinates, we draw the circle (node), the text (label), and the line (edge) connecting a node to one of its children.

**Building the Shadow Tree**

The recursive function buildShadowTree() uses an inorder scan (LNR) of the original tree to build the shadow tree. This insures that when we visit the nodes from left to right, we move from one grid column to another. For instance, with Tree 0, the order of visit is B D A E C. Note in Figure 1 that this is column-order for the nodes in the tree.

A node in the shadow tree has the basic tnode structure, with additional data members level and column that specify the coordinates for a cell in the grid. The data member, value, is a string that contains ASCII characters representing the node data. For instance, if a tree node has integer value 135, then value in the corresponding shadow tree node is the string "135". The nodes are declared as stnodeShadow objects.

<table>
<thead>
<tr>
<th>left</th>
<th>value</th>
<th>level</th>
<th>column</th>
<th>right</th>
</tr>
</thead>
</table>

**DECLARATION: stnodeShadow CLASS**

```cpp
// objects hold a formatted label string and the level, column
// coordinates for a shadow tree node
class stnodeShadow
{
  public:
    string nodeValueStr; // formatted node value
    int level, column;
    stnodeShadow* left, *right;

    stnodeShadow()
    {}       
};
```

The algorithm for the function buildShadowTree() resembles copyTree() with the exception that it makes an inorder scan of the original tree rather than a postorder scan. Each recursive step creates a node in the shadow tree by first allocating a stnodeShadow node, assigning it the string that corresponds to the value of the node in the original tree, and then attaching the node to its left and right subtrees. For arguments, the function takes a tnode pointer, t, which provides access to the value of the tree node, the integer level that denotes the level in the tree, and
an integer reference argument col. The return value is a stnodeShadow pointer that references the root of the shadow tree.

    // recursive inorder scan used to build the shadow tree
    template <typename T>
    stnodeShadow *buildShadowTree(tnode<T> *t, int level, int& column);

Using a reference argument for column is key to the algorithm. This makes the variable global relative to the recursive process. Each recursive call in the inorder scan creates a node in the next column. Each recursive call simply increments column. Other than that, buildShadowTree() uses the copyTree() design. Each recursive call creates a child node and returns a pointer to the node. The return values become the values for the left and right fields of the stnodeShadow object.

    buildShadowTree()
    // recursive inorder scan used to build the shadow tree
    template <typename T>
    stnodeShadow *buildShadowTree(tnode<T> *t, int level, int& column)
    {
        // pointer to new shadow tree node
        stnodeShadow *newNode = NULL;
        // text and ostr used to perform format conversion
        char text[80];
        ostrstream ostr(text, 80);
        if (t != NULL)
        {
            // create the new shadow tree node
            newNode = new stnodeShadow;

            // allocate node for left child at next level in tree; attach node
            stnodeShadow *newLeft = buildShadowTree(t->left, level+1, column);
            newNode->left = newLeft;

            // initialize data members of the new node
            ostr << t->nodeValue << ends; // format conversion
            newNode->nodeValueStr = text;
            newNode->level = level;
            newNode->column = column;

            // update column to next cell in the table
            column++;

            // allocate node for right child at next level in tree; attach node
            stnodeShadow *newRight = buildShadowTree(t->right, level+1, column);
            newNode->right = newRight;
        }
        return newNode;
    }

Displaying the Shadow Tree

The function drawTree() calls buildShadowTree() to create the shadow tree and then displays the tree using a level order scan of the shadow tree nodes. Each shadow tree node provides the label and the (level,col) coordinate for the drawing of a circle on the screen. The left and right pointer fields indicate whether the node has children. If so, we
need to draw a line (edge) connecting the parent node (circle) with the child node (circle). The function drawTree() takes a tNode pointer that designates the root of the original tree and the integer maxCharacters that specifies the string length for the longest label in the tree.

```
template <typename T>
nvoid drawTree(tnode<T> *t, int maxCharacters);
```

The design of drawTree() focuses on the level order scan of the shadow tree. The scan uses a queue to store and access the nodes. The rest of the function is a fairly detailed conversion of the cell pair (level, column) to the screen pair (x, y) which is the upper-left-hand corner of the square in the grid that will hold a tree node and its label (Figure 2). To determine the dimensions of the square, we use the value maxCharacters which is the length of the longest label. Since each character in the label is approximately \(\text{UNITS\_PER\_CHAR} = 0.15\) units wide, the length of the size of the square is \(\text{cellSide} = \text{UNITS\_PER\_CHAR} \times \text{maxCharacters} + 0.2\) units. The value 0.2 is selected to allow room outside the label for the drawing of the circle (node).

```
// approximate width of a character drawn by textShape
const double UNITS_PER_CHAR = .15;

// length of side for square in the screen grid
double cellSide = maxCharacters*UNITS_PER_CHAR + 0.2;

// circle with radius cellSide/2.0 represents the node
circleShape node(0,0,cellSide/2.0, lightblue);

// text draws the formatted value in a node
```

![Figure 2](image2.png)

**FIGURE 2**

Cell in the drawing surface with point \((x,y)\) defining the coordinates for the upper-left-hand corner of the square.

The node is a light-blue circle with radius \(\text{cellSide}/2.0\). To draw the circle, we must locate its center which is at coordinates \((x + \text{cellSide}/2.0, y + \text{cellSide}/2.0)\). This is also the starting point for drawing a line (edge) to a child. The end point for the line is the center of the child node. When we descend to the next level and draw the children, an edge is clipped at the boundary of a node (Figure 3).

![Figure 3](image3.png)

**FIGURE 3**

Drawing nodes and edges

Let us identify some of the key variables, objects, and constants used by drawTree().
There are two main steps to the drawTree() algorithm. The first step begins by declaring integer variables for the level and column of a node and using them to call buildShadowTree(). The return value, root, is a pointer to the root of the shadow tree.

```cpp
int level = 0, column = 0;
// build the shadow tree
stnodeShadow *root = buildShadowTree(t, level, column);
```

The second step is a level-order scan of the shadow tree. The scan uses a queue that stores the nodes; actually pointers to the nodes. Before beginning the scan, a call to openWindow() creates the drawing surface.

```cpp
// store siblings of each stnodeShadow object in a queue so that
// they are visited in order at the next level of the tree
queue<stnodeShadow *> q;
```

As nodes are popped from the queue, drawTree() carries out the task of drawing the node, its label, and edges to any children. We noted some of the calculations above. A more complete explanation is given by the documentation comments in the code listing. At the end of the scan the graphics library function, viewWindow(), maintains the tree drawing until the user presses a key. This allows the user to view, copy, or print the tree. The function concludes by calling closeWindow() to shut-down the graphics system.

```cpp
drawTree()
template <typename T>
void drawTree(tnode<T> *t, int maxCharacters)
{
    // approximate width of a character drawn by textShape
    const double UNITS_PER_CHAR = .15;
    // estimated width of a formatted node value.
    // add .2 to allow space outside the label
    double cellSide = maxCharacters*UNITS_PER_CHAR + 0.2;

    string label;
    int level = 0, column = 0;

    // build the shadow tree
    stnodeShadow *root = buildShadowTreeD(t, level, column);

    // use during the level order scan of the shadow tree
    stnodeShadow *currNode;

    // node draws the circle shape that represents a node.
    // the radius is (cellSide + .20)/2.0
    circleShape node(0,0,cellSide/2.0, lightblue);

    // text draws the formatted value in a node
    textShape text(0,0,"",darkgray);
```
// edge draws edges in the tree
lineShape edge(0,0,0,0,black);

// x, y coordinates of a circle center on the screen
double x, childx, y, childy;

// open the drawing window
openWindow();

// store siblings of each stnodeShadow object in a queue so that
// they are visited in order at the next level of the tree
queue<stnodeShadow *> q;

// insert the root in the queue
q.push(root);

// continue the iterative process until the queue is empty
while(!q.empty())
{
    // delete front node from queue and make it the current node
    currNode = q.front();
    q.pop();

    // assign formatted node label to string label
    label = currNode->nodeValueStr;

    // convert each (level, column) coordinate into screen
    // coordinates for the center of the node. add .1 so
    // we stay away from the edges of the screen
    x = currNode->column * cellSide + cellSide/2.0 + 0.1;
    y = currNode->level * cellSide + cellSide/2.0 + 0.1;

    // if a left child exists, draw an edge from the current
    // node center to the child node center. insert the child
    // in the queue
    if(currNode->left != NULL)
    {
        edge.move(x, y);
        // compute the center of the left child node
        childx = currNode->left->column * cellSide + cellSide/2.0 + 0.1;
        childy = currNode->left->level * cellSide + cellSide/2.0 + 0.1;
        edge.setEndPoint(childx, childy);
        edge.draw();
        q.push(currNode->left);
    }

    // if a right child exists, draw an edge from the current
    // node center to the child node center. insert the child
    // in the queue
    if(currNode->right != NULL)
    {
        edge.move(x, y);
        // compute the center of the right child node
        childx = currNode->right->column * cellSide + cellSide/2.0 + 0.1;
        childy = currNode->right->level * cellSide + cellSide/2.0 + 0.1;
        edge.setEndPoint(childx, childy);
Application: Binary Tree Draw

A program illustrates drawTree() for the three characters trees, Tree1, Tree2, and Tree 3 and for a six-node tree with string labels. The program is listed as "bintreeDraw.cpp" in Ch 10 of the software supplement.

PROGRAM 1 VERTICAL TREE DISPLAY

In a loop, the program draws Trees 0 through 2. The function clearTree() deletes each tree before the loop draws the next tree. The string tree is built from scratch.

```cpp
#include "d_tnode.h"   // tnode library
#include "d_tdraw.h"

using namespace std;

int main()
{
    // declare a character tree
tnode<char> *root;
tnode<string> *strroot, *b, *c, *d, *e, *f;
    int i;

    for (i=0; i <= 2; i++)
    {
        root = buildTree(i);
        // each node requires one character
drawTree(root,1);

        // delete the tree
clearTree(root);
    }

    // build tree with string nodes
    f = new tnode<string> ("Dolan");
```
```
e = new tnode<string> ("Wren");
d = new tnode<string> ("Gates");
c = new tnode<string> ("Smith", e, f);
b = new tnode<string> ("Brand", (tnode<string> *)NULL, d);
strroot = new tnode<string> ("Daly", b, c);
drawTree(strroot,5);

return 0;
```

**Drawing Member Functions in the Stree Class**

In the text, the stree class implements the member function `displayTree()` that provides only a console display of the tree. An extended version of the class implements the graphical functions `drawTree()` and `drawTrees()`. The functions take an integer argument that specifies the maximum string length of an element. With `drawTree()`, the frame holds until you hit the Enter key and then the window closes. With `drawTrees()`, hitting the Enter key clears the screen, awaiting the next display which can be provided by another function call to `drawTrees()` or a terminating call to `drawTree()`. The function `drawTrees()` enables you to draw multiple trees in separate frames.

```
void drawTree(int maxCharacters) {
    // stree member function displays the stree collection in a graphical window
}

void drawTrees(TNode<T> t, int maxCharacters) {
    // stree member function displays a tree and leaves the window open
}
```

**Application: Stree Collection Draw with Strings and Characters**

A program illustrates `drawTree()` and `drawTrees()` for a search tree of characters from the string "global". With `drawTrees()` you can view the building of the tree, character by characters. A second tree is created from an array and then displayed. The program is listed as "streeDrawA.cpp" in Ch 10 of the software supplement.
In a loop, the program inserts successive elements from the string "global" and then displays the tree. The window stays open. The tree with strings from array list is created and then displayed using drawTree(). In this case, the window is closed. A sample of frames are displayed as output.

```cpp
#include <string>
#include "d_stree.h" // stree class
using namespace std;

int main()
{
    string str = "global";
    string list[] = {"Dolan", "Wren", "Gates", "Smith", "Brand"};

    // declare a character tree
    stree<char> charTree;
    int i;

    for (i=0; i < str.length(); i++)
    {
        charTree.insert(str[i]);

        // display tree after each insert
        charTree.drawTrees(1);
    }

    stree<string> strTree(list, list+5);
    strTree.drawTree(5);

    return 0;
}
```

**Application: Stree Collection Draw with Integers**

A program illustrates drawTree() and drawTrees() for a search tree of integers values from an array. In the run of the program you can view the stree growing insert by insert and then view the tree after deletion of elements 60, 30, and 50 respectively. The program is listed as "streeDrawB.cpp" in Ch 10 of the software supplement.
The output frames display the tree after all of the elements have been added and then again after the deletions.

```cpp
#include <iostream>
#include "d_stree.h"  // stree class
using namespace std;

int main()
{
    int list[] = {50, 30, 75, 55, 60, 25, 52, 37, 53, 62, 10, 85, 33};
    stree<int> tree;
    int i;
    for (i = 0; i < 13; i++)
    {
        tree.insert(list[i]);
        tree.drawTrees(2);
    }
    tree.erase(60);
    tree.drawTrees(2);
    tree.erase(30);
    tree.drawTrees(2);
    tree.erase(50);
    tree.drawTree(2);
    return 0;
}
```

After Inserting List

After Deleting 60, 30, 52