Binary Search Tree Visualization Algorithm

Vadym Borovskiy[1], Jüller Müller[2], Matthieu-Patrick Schapranow[3], Alexander Zeier[4]
Hasso Plattner Institute for Software Systems Engineering
[1](vadym.borovskiy@hpi.uni-potsdam.de)
[2](juergen.mueller@hpi.uni-potsdam.de)
[3](matthieu.schapranow@hpi.uni-potsdam.de)
[4](alexander.zeier@hpi.uni-potsdam.de)

Abstract - **Binary search tree is a very common data structure in computer programming. Working with large BSTs can become complicated and inefficient unless a programmer can visualize them. This article contributes with two BST visualization algorithms that draw a tree in time linearly proportional to the number of nodes in a tree.**

Keywords - **Binary Search Tree, Tree Visualization**

I. INTRODUCTION

Many tasks require a programmer to organize data in collections and perform different operations on these collections. Moreover, the collections and the operations must often be designed in a way that guarantees certain parameters of program execution, for example speed and memory consumption. Because working with data collections or sets is so frequently encountered exercise, a number of attempts has been made to standardize these exercises and, thus, reduce the time and effort of their implementation. For this reasons many standard data structures and algorithms appeared. Using these well-defined data structures and algorithms[1] programmers can quickly and efficiently solve various tasks.

One of the standard data structures that has been widely used in programming is the *tree* data structure. Tree structure means a ”branching” relationship between nodes (Knuth, 1973) and imposes a hierarchical structure on the collection of items. There are many types of trees: binary trees, balanced trees, 2-3trees, B-trees, red-black trees, Fibonacci trees, AVL trees to name just a few. Each type of the tree data structure has been designed to support a specific set of properties essential in a given situation.

Tree structures are the most important nonlinear structures that arise in computer algorithms (Knuth, 1973). Trees have numerous applications. They are used to analyze electrical circuits, to represent the structure of mathematical formulas, to organize information in database systems, to present the syntactic structure of source programs in compilers and many others (Aho et al., 1983).

In our last project we required to perform a complicated analysis of data supplied by experts. In the core of analysis algorithm were Search, Minimum, Maximum, Insert and Delete operations on a collection of integers of size up to 5000 items. The efficiency of the main analysis algorithm greatly depended on the efficiency of the aforementioned operations. We also had a requirement of implementing this analysis algorithm in a flexible way that will minimize the effort of migrating it to other platforms (originally we used .NET, but the execution environment could change to Win32 C++ or Java). The last requirement greatly limited the choice of implementation tools. In particular, we could not use any platform-specific data types and design techniques. Therefore, we decided to implement our own data type that will guarantee the required efficiency of the above operations. The implementation was allowed to use only basic programming constructs (like *for*-loops, 4-byte integers, UNICODE strings and so on), which could easily be ported to other environments.

After reviewing classical data structures and algorithms we decided to use the *binary* searc*h* tre*e* data structure as a prototype of the collection that our analysis algorithm would operate on. A binary search tree (BST) is a tree with keys which are always stored in a way that satisfies the binary-search-tree property (Cormen et al., 2001): If y is a node in the left subtree of node x, then the key of y is less than or equal to the key of x. If y is in the right subtree of x, then the key of y is greater than or equal to the key of x.

Having implemented the collection and the basic operations on the collection we started developing our analysis algorithm on top of those. Very soon we experienced that working with a big BST can become cumbersome unless a developer has a visual representation of the tree. In fact, when a tree became too big (more than 3000 nodes) we could not debug the analysis algorithm efficiently because we simply lost track of where we were in a tree. Hence, we arrived at the challenge of visualizing BSTs on which the analysis algorithm worked. The desired algorithm had to be simple, easy to port to other execution environments and efficient for large trees.

In this article we present two methods of drawing a binary search tree. The article is structured as follows. The section 2 names other approaches to tree/graph visualization. The section 3 lists all the assumptions we made about BSTs to draw and contributes with two ways of drawing a BST.

[1]or their modifications developed to handle specific requirements.

978-1-4244-3672-9/09/$25.00 ©2009 IEEE

II. RELATED WORK

A tree is an acyclic graph. Therefore, graph visualization tools can be used to draw a BST.

Graphviz[2] - is open source graph visualization software. It has several main graph layout programs. It also has web and interactive graphical interfaces, and auxiliary tools, libraries, and language bindings.

Microsoft Automatic Graph Layout[3] - formerly known as GLEE, is a .NET tool for laying out and visualizing directed graphs. MSAGL can be used to represent complex directed graphs as well as trees. MSAGL includes three integrated components: (i) *An automatic layout engine* is used to position and connect graphs' vertices. (ii) *A drawing layer* enables the modification of the attributes (e.g. color, style) of the graphical components. (iii) *A Windows-based viewer* that uses the engine and drawing layer to create interactive graphs.

Both products are technology specific and far too complex for our task.

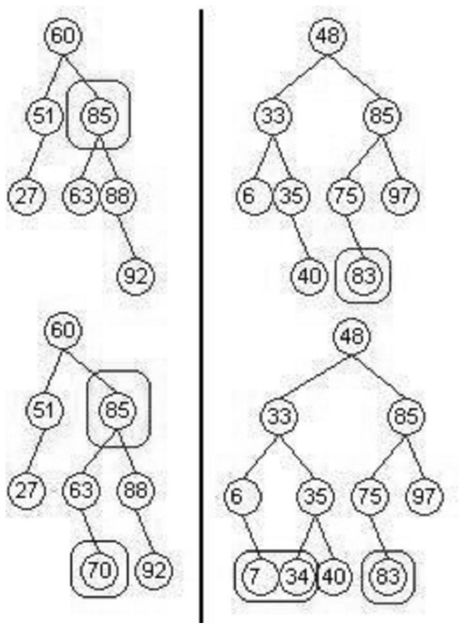

Fig.1: Position of a node

III. DRAWING ALGORITHM

When building a BST the coordinates of every node depend on the node's relative position to other nodes in a tree. In other words, the knowledge of a tree's structure is required. In particular, the number of a node's children, siblings, ancestors and descendants and their position influence the position of a currently considered node. On the Figure 1 the nodes 85 and 83 have different horizontal coordinates depending on the number of children and siblings respectively.

[2]www.graphviz.org

[3]www.research.microsoft.com

The BST data structure does not offer this information and the only way to get it is to traverse a tree or at least some part of it. Of course, before drawing a node we could pre-process the required part of the tree, but this would ensue a performance hit and drawing a large tree would become slow.

In this section we present two approaches of drawing a BST. Each of them draws a tree in θ(N) time, where N is the number of nodes in a tree. Memory consumption depends linearly on the height of a tree - O(h)[4]. Both algorithms draw a BST while they traverse it. No node is visited twice. Both algorithms require the height of a tree to be known and the second algorithms needs the number of nodes in a tree.

The algorithms below assume that a BST is presented as a number of interlinked instances of a *Node* structure. The structure has the following elements: (i) *Key* -an integer denoting the key of a node; (ii) *Rl* -a right child, a link/pointer to another instance of *Node* structure; (iii) *Ll* -a left child, a link/pointer to another instance of *Node* structure. If a node does not have any of the children a corresponding pointer is set to *null*. In addition no tree contains duplicated nodes[5].

A. Drawing BST with Fixed Coordinates

The algorithm presented in this subsection calculates the coordinates of a node from the node's depth and the position of the node at its level. The Figure 2 helps to understand how the algorithm works. The figure shows a logical grid in the cells of which the nodes of a BST are drawn. All cells of the logical grid are indexed as on the Figure 3.

Using the following rules the algorithm defines a cell in which a node must be drawn: (i) The root is always drawn in the cell (0,0); (ii) The row index *i* of any child equals to that of its parent plus 1; (iii) The column index *j* of a left child equals to doubled column index of its parent, and for the right child *j* equals to the index of the left child plus 1.

The algorithm traverses a tree in postorder. Therefore, children of any node are drawn first. Every level of the tree gets equal space on a canvas. This space is divided into equal compartments and in the middle of each compartment a node is drawn. The number of compartments equals the maximum number of nodes at a given level -2^i, where *i* is the vertical distance from a node to the root of the tree, called the *depth* of a node. The depth of the root is 0.

The algorithm requires a canvas of the size

$$\begin{aligned} \textit{Canvas}\ \text{width} &= \text{wc} * 2^{\textit{TreeHeight}} \\ \textit{Canvas height} &= \textit{hc} * \textit{TreeHeight} \end{aligned} \qquad (1)$$

[4]For a randomly built tree the height is limited by $\log_2 n$

[5]This is not a critical assumption and does not change the algorithms. This assumption simplifies drawing lines connecting nodes.

where *wc*,*hc* are the width and height of a compartment at the bottom level of the grid. These are constants defined by a programmer. The width of com partments at other levels is W_i= wc*$2^{TreeHeight-I}$ as shown on the Figure 2.

The following listing demonstrates the algorithm implemented in C#.

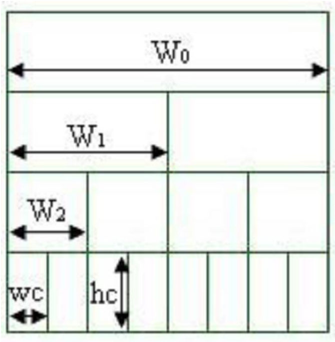

Fig.2.Logical grid

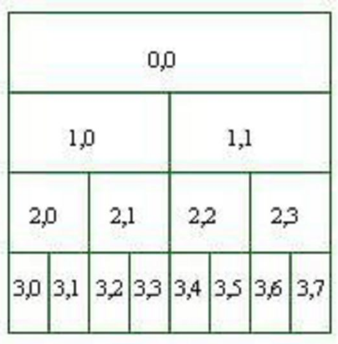

Fig.3.Cell indices

```
void DisplayTree(Node root) {
    if (root == null)
    return;
    //Start the recursion from
    //the root's children
    if (root.Ll != null)
    DrawTree(root.Ll, root, 0, 0);
    if (root.Rl != null)
    DrawTree(root.Rl, root, 0, 0);
    //Calculate the root's X-coordinate
    int x = wc*Math.Pow(2,H-1);
    Ellipse(x-d/2, (hc-d)/2, d, d);
    DrawString(root.ToString(),
    x-d/2, hc/2-d/3);
}
//c - a node to draw
//p - the parent of the c-node
//i,j - indices of the parent's cell
void DrawTree(Node c, Node p,
                int i, int j) {
    //Indices of the c-node's cell
    int ic, jc;
    ic = i + 1;
    jc = 2 * j;
    if (p.Key < c.Key)
      //c-node is the right child
      jc++;
    if (c.Ll != null)
     DrawTree(c.Ll, c, ic, jc);
    if (c.Rl != null)
     DrawTree(c.Rl, c, ic, jc);
    //Calculate the c,p-node's coordinates
    //from their cell indices
    int x, y, xc, yc;
    x=(2*j+1)*lc*Math.Pow(2,H-i-1);
    y = (2*i+1)*hc/2;
    xc=(2*jc+1)*lc*Math.Pow(2,H-ic-1);
    yc = (2*ic+1)*hc/2;
    //Draw a line connecting c and p nodes
    Line(x, y+d/2, xc, yc-d/2);
    //Draw the c-node
    Ellipse(xc-d/2, yc-d/2, d, d);
    DrawString(c.ToString(),
                  xc-d/2, yc-d/3);
}
```

The Figure 4 shows a BST drawn by this algorithm.

From computational perspective this method is quite good, but it has a serious disadvantage. It uses a canvas in the least efficient way. If a tree is close to balanced[6] or even complete[7] this deficiency becomes less obvious. However, if a tree has its leaves distributed across all levels, the algorithm will require a large canvas. This can be seen on an extreme case when a BST degenerates into a

[6]A balanced BST is a BST in which the height of the left subtree of every node never differs by more than ±1 from the height of its right subtree (Knuth, 1973).

[7]A binary tree is said to be complete if for some integer *k* every node of the depth less than *k* has both a left and a right

child and every node of the depth *k* is a leaf (Aho et al., 1974).
linked list as on the right part of the Figure 5. Another example is shown on the left part of the Figure 5. This is a perfectly drawn BST. However, if we add only one node to the tree, the algorithm will require a canvas two times as wide as the current one. Only because of a single node at the level 4 the whole tree will require a considerably wider canvas. This happens because the algorithm reserves the maximum possible space for any level of the tree, no matter how many nodes are on this level. The bigger a BST the more obvious this deficiency becomes.

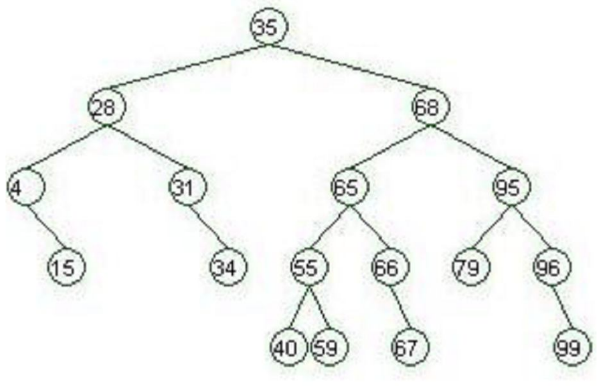

Fig.4. BST with fixed coordinates

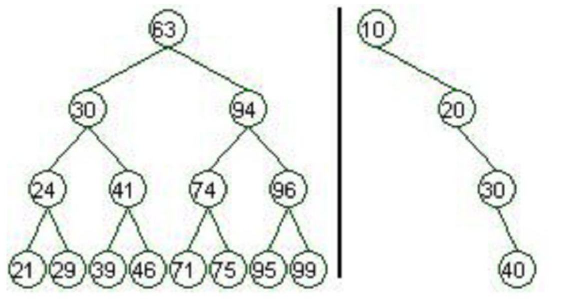

Fig.5: The deficiency of the algorithm

B. Drawing BST with Floating Coordinates

In this subsection we present an algorithm that draws a node horizontally as close as possible to a previously drawn node. Therefore, no canvas space is wasted. In fact, the algorithm produces the most compact drawing possible for a given BST.

While traversing a BST the algorithm internally keeps track of space usage by accumulating the horizontal and vertical offsets of already drawn nodes. The drawing of a tree is done recursively. Inside every recursion step the offsets are updated. When drawing the next node its coordinates are calculated from these values. The horizontal offset of a node *c* depends on the sum of the width of its left subtree and the horizontal offset of the last drawn node. The vertical offset depends on that of the node's parent.

```
void DisplayTree(Node root) {
    if (root == null)
        return;
    //Horizontal and vertical offsets
    int oX = 0;
    int oY = (hc-d)/2;
    //The width of tree and X-coordinate
    //of its root
    int w, rX;
    //Start the recursion from the root
    DrawTree(root, oX, oY, out rX, out w);
}
//c - a node to draw
//oX, oY - initial offsets
void DrawTree(Node c, int oX, int oY,
                  out int rX, out int w) {
    w = 0;
    //The width of a subtree
    int stW = 0;
    //X-coordinate of a subtree's root
    int stRX;
    if (c.IsLeaf) {
      w = d;
      rX = oX;
    }
    else {
      //Draw left subtree
      if (c.Ll != null) {
          DrawTree(c.Ll, oX, oY+hc,
                        out stRX, out stW);
      w = stW;
      //Draw a line connecting the left
      //subtree with the c-node
      rX = oX+stW-d/2;
      Line(rX+d/2,oY+d,stRX+d/2,oY+hc);
    }
      else {
      //Handling the case if there is
      //no left child
      w = d/2;
```

```
        rX = oX;
    }
    //Draw right subtree
    if (c.Rl != null) {
        DrawTree(c.Rl, oX+w, oY+hc,
                      out stRX, out stW);
        w += stW;
        //Draw a line connecting the right
        //subtree with the c-node
        Line(rX+d/2,oY+d,stRX+d/2,oY+hc);
    }
    else
        //Handling the case if there is
        //no right child
        w += d/2;
    }
    //Draw the c-node
    DrawString(c.ToString(), rX, oY);
    Ellipse(rX, oY, d, d);
}
```

The Figure 6 shows how the same tree from the previous subsection is drawn in a more compact way.

The algorithm requires a canvas of the size

$$Canvas\_width \le \frac{wc * (N+1)}{2} \qquad (2)$$

$$Canvas\ height = hc * TreeHeight$$

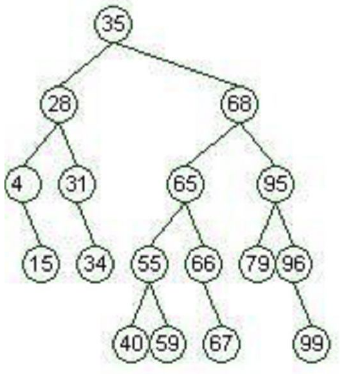

Fig.6.BST with floating coordinates

where N is the number of nodes in a tree. As it can be seen the width of a canvas cannot be calculated precisely beforehand. The reason is the unknown structure of a tree. Only after a tree has been traversed the exact value of *Canvas width* can be obtained. The minimal width has a BST degenerated into a linked list. In this case the algorithm will position the nodes of the tree one below another with minimal horizontal offset. The maximal width has a complete tree, as the opposite extreme to the linked list. In this case the width of the tree is the width of its bottom level. If a tree is complete and has N nodes, the bottom level will always have $\frac{N+1}{2}$ nodes each wc wide.

CONCLUSION

The current work addresses the issue of drawing a binary search tree and contributes with two algorithms of BST visualization. Each of them draws a tree in $\Theta(n)$ time. Memory consumption is proportional to the height of a tree -$O(h)$. Both algorithms draw a BST while they traverse it and require the height of a tree to be known. The the second algorithm needs the number of nodes in a tree as well.

In terms of computational performance the algorithms behave the same, since both traverse the tree in the same way. However, the second algorithm is more preferred, because it uses a canvas more efficiently.

REFERENCES

[1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1974). The design and analysis of computer algorithms. Addison-Wesley.

[2] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1983). Data structures and algorithms. Addison-Wesley.

[3] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2001). Introduction to Algorithms. The MIT Press.

[4] Knuth, D. (1973). The art of computer programming. Addison-Wesley.