DSInk:
A Software Library for Developing Data Structures Applications for the Tablet PC

by

Robert S. Culler

Submitted to the graduate faculty of the Department of Computer Science
in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science.

March 29, 2005

Clemson University
Clemson, SC 29634 – 0974

Abstract

As technology has evolved in the classroom, more and more computer science instructors are using computers projected onto a screen as the standard teaching device in their lectures. However, there is no good way to draw shapes and structures on a standard computer, and the instructor typically uses a whiteboard to draw these structures. The Tablet PC provides an excellent alternative to this combination approach, which often disrupts the flow of the lecture. Numerous instructional tools have been developed for the Tablet PC. Most of these, however, use digital ink only for displaying notes or annotating documents. Thus, the digital ink can be thought of as “static”. We have developed the Data Structures Ink (DSInk) library to provide a way of giving meaning to ink drawn by the user. This library provides a set of tools that allows for the rapid and efficient development of data structure applications. Powerful applications can be (and have been) built that effectively illustrate data structures and that dynamically perform algorithms on these structures. The library is straightforward to use in developing applications, and is easily extensible. Also, the flow of the classroom lecture will be improved by using DSInk applications, and, by making these applications publicly available, students can reinforce what they learned in class on their own time. This paper describes the motivation for DSInk, related work in the area of Tablet PC instructional applications, and a general overview of the Tablet PC platform. The approach to the design of DSInk is discussed, and the class structure of the library is presented. Applications that have been developed - including binary search tree, red-black tree, and AVL tree applications – are also described. Finally, the paper presents conclusions and future work that will be conducted.

Acknowledgements

I would like to thank Dr. Roy Pargas for his guidance throughout the development of DSInk and for his help in choosing Tablet PC's as my area of research. I would also like to thank Dr. Brian Malloy and Dr. Robert Geist for serving on my committee. I would like to express my gratitude to Dr. Wayne Madison and the Clemson University Computer Science Department for letting me borrow a Tablet PC on which to develop and test the DSink library.

I would also like to thank Shawn A. Van Ness at the Leszynski Group for providing me with a private copy of the Leszynski InShape™ Developer Preview Edition, and Stefan Wick for answering many of my Tablet PC development questions.

Contents

List of Figures

1. Introduction
2. Related Work
3. Problem Statement
4. Overview of Tablet PC Platform
    - 4.1 Ink Collection
    - 4.2 Ink Data Management
    - 4.3 Ink Recognition
    - 4.4 Development on the Tablet PC
5. Approach
    - 5.1 The *Node* Class
    - 5.2 The *TreeNode* Class
    - 5.3 The *AVLnode* Class
    - 5.4 The *RedBlackNode* Class
    - 5.5 The *Edge* Class
    - 5.6 The *Tree* Class
    - 5.7 The *BSTree* Class
    - 5.8 The *AVLTree* Class
    - 5.9 The *RedBlackTree* Class
6. Results
    - 6.1 Shape Recognition
    - 6.2 Modes
        - 6.2.1 Binary Search Tree Mode
        - 6.2.2 AVL Tree Mode
        - 6.2.3 Red Black Tree Mode
    - 6.3 Selecting and Moving Nodes
    - 6.4 Traversals
7. Conclusions and Future Work
8. References

Appendix A – DSInk Library Reference

List of Figures

Figure 1 Relationship between *Ink, Stroke,* and *Strokes* objects
Figure 2 DSInk Class Diagram
Figure 3 Pseudocode for *getSubtree* method
Figure 4 Pseudocode for *insert* method in *BSTree* class
Figure 5 Pseudocode for *remove* method in *BSTree* class
Figure 6 Pseudocode for *insert* method in *AVLTree* class
Figure 7 Pseudocode for *rotateWithLeftChild* method
Figure 8 Pseudocode for *doubleWithLeftChild* method
Figure 9 Pseudocode for *insert* method in *RedBlackTree* class
Figure 10 Recognizing a line as an edge between two nodes
Figure 11 Binary search tree before and after removal of value 8
Figure 12 AVL Tree During Insert
Figure 13 AVL Tree After Rotation
Figure 14 Red-black tree before and after insertion of value 26
Figure 15 Tree before selection
Figure 16 Code for redrawing an edge between two nodes after a node has moved
Figure 17 Tree after node 6 has been selected and moved
Figure 18 Red-black tree after post-order traversal

1. Introduction

As technology has evolved in the classroom, more and more computer science instructors are conducting lectures using a computer connected to the Internet and projected onto a screen. While this method of teaching offers certain advantages, such as the ability to write and execute code, convenient access to the Internet, the ability to give PowerPoint presentations, etc., it unfortunately lacks some of the beneficial characteristics of its whiteboard and pen counterpart. Most notable is the lack of ability to easily draw shapes and structures. Hence, many instructors use a combination of the two approaches – using the computer and projector for slides and code, and using the whiteboard to draw shapes and structures. Unfortunately, this combination approach also has some drawbacks. First of all, the projector screen, in many cases, covers up a portion of the whiteboard. The instructor then has two choices: he can raise the screen each time he wants to use the whiteboard, which can be time-consuming and awkward depending on how many times he is going back and forth, or, he can leave the screen where it is and use the portion of the whiteboard not covered by the screen. This usable portion of the board may be very small. In fact, [16] suggests that only 20% of the board may be available in smaller classrooms. Also, the usable portion is often off to either side of the projector, which makes it difficult for some students to get a good view of the board. Another problem that can arise in switching between these two modes of lecturing is the constant practice of having to turn off and turn back on lights at the front of the classroom. If the lights are dimmed at the front of the classroom to allow the students to see the projection screen more clearly, these lights will have to be turned back on while

notes are being made on the whiteboard. The flow of the lecture will, undoubtedly, be disrupted.

By providing a pen, the Tablet PC provides an excellent alternative to the regular desktop or laptop computer. The instructor can not only make use of all of the capabilities of a regular computer, but he also has the ability to draw anything he wants, much like using a whiteboard. Additionally, he doesn't have to walk back and forth between the computer and the board or raise and lower a projection screen. The instructor can also make use of the advantages that digital ink has over the whiteboard. He can easily change the color of ink without having to use multiple pens. He can erase ink easily simply by inverting the pen or by clicking on an "Erase Mode" button in an application. He can move and resize ink that has already been drawn. And perhaps most important, he can save digital ink. Therefore, a student who misses class or who wishes to review a lecture can access any notes that the instructor has written in class and posted on his website. For example, if the instructor lectures on the topic of linked lists and uses the pen to describe how inserts and deletes are performed, this digital ink can be saved, and a student who misses the lecture can view it at a later time. Another advantage of using a Tablet PC is that the instructor can maintain eye contact with the students while lecturing and never impedes a student's view of the lecture notes. With a wireless connection between the Tablet PC and the projector, the instructor can even walk around the classroom and interact with the students. This provides for an environment which supports active learning. Willis and Miertschin conducted a one-year study of the Tablet PC as an instructional tool and concluded that the Tablet PC-based lecture enhanced the areas of note-taking, presentation creation, document markup, and information

management [20]. Also, studies performed by [1], [3], [13], [14] have all tested Tablet PC-based instruction with favorable reaction from students and instructors.

In this work, we describe the motivation and design of Data Structures Ink (DSInk), a software library which allows for the development of powerful data structures applications for the Tablet PC. Instructional applications developed with the DSInk library not only provide for all of the advantages of a Tablet PC-based lecture, but we feel they also improve over existing Tablet PC instructional applications by giving meaning to the user-drawn ink.

2. Related Work

A number of instructional tools for the Tablet PC have been developed. The most popular of these is Classroom Presenter, developed at the University of Washington [1], [2]. Presenter is a slide-based presentation system that allows an instructor to annotate pre-prepared slides. The instructor runs Presenter on a Tablet PC and communicates to a second machine, also running Presenter, which drives the projector. The instructor's view includes navigation views and numerous ink options, while the projector view shows only the current slide. Presenter also has the ability to broadcast the presentation, including real-time inking, to students running Presenter on Tablet PC's. Students with Tablet PC's can submit real-time feedback to the instructor as well. Since the spring of 2002, Presenter has been used in 25 different computer science courses with positive feedback from both students and instructors. Students indicate that they pay more attention to the lecture, the material is more easily understood, and they encourage the use of Presenter in the future.

Ubiquitous Presenter (UP) [19] is an extension to Presenter. UP provides the ability for non-Tablet PC users in the audience of a lecture to receive the Presenter slides via web browsers. Also, students can view the presentation in class either synchronously or asynchronously, which is not possible using Presenter. Therefore, a student can choose to look at previous slides while the instructor continues the presentation. Finally, using UP, students who do not have Tablet PC's can also submit annotated slides to the instructor over Internet browsers by overlaying typed text on the slides.

The DyKnow system [3] is similar to UP. Using this system, students sit at pen-enabled video tablets while the instructor lectures on an electronic whiteboard or Tablet PC. The instructor can draw, type, or import information onto his teaching device and this information will show up immediately on the students' video tablet. Additionally, students can annotate the information on their screens and submit this information to the instructor. Also, the instructor can give a student control, and the student's screen will be displayed on the electronic whiteboard at the front of the classroom. This system was tested at DePauw University with very positive results. Students rated the system as enjoyable, and stated that it enhanced their understanding of course material, provided them with a better set of notes, and kept their attention in class. They also indicated that they would be more likely to take another class if DyKnow were used in the class.

Golub developed the Tablet Mylar Slides (TMS) application [6] at the University of Maryland. TMS is a digital version of an overhead projector that merges the best aspects of the two types of traditional transparencies: decks and rolls. TMS allows the instructor to have slides of any length and also to insert and delete material anywhere he would like.

Other related work includes Lecturer's Assistant [4], a very early system that integrated slides, writing, and an overhead projector, and ActiveClass [7], which is a PDA-based system.

3. Problem Statement

Although many instructional applications have been developed on the Tablet PC, most of them do not fully take advantage of the capabilities provided by the Tablet PC platform – most notably, the ability for digital ink to have meaning. These applications simply treat the pen as a note-taking device. However, when we use ink only to mark up a slide or to display hand-written notes on a blank page, we are missing out on a powerful set of tools provided to us by the Tablet PC SDK. It would be nice to be able to draw some shape or structure on the screen and have that ink recognized and treated as the structure drawn.

This is the goal of the DSInk library. This library provides a set of tools that allows for the development of data structure applications. Powerful applications can be (and have been) built that give meaning to the ink that is drawn and effectively illustrate data structures and many algorithms that can be performed on these structures. For example, suppose an instructor is lecturing on the topic of AVL trees and wishes to illustrate an insertion into a tree. Using an application which gives no meaning to ink, the instructor would need to draw the nodes, edges, and values of the tree and then describe the insertion process by pointing to nodes that are visited and drawing the new node at the point where it is inserted. If the resulting AVL tree is unbalanced and a rotation needs to be made, the instructor must erase strokes that he has already drawn and draw new strokes to reflect new relationships between nodes in the tree. Thus, the screen can

become quite chaotic and the insert algorithm may not be clearly reflected to the students. The DSInk library seeks to solve this problem by giving the strokes on the screen meaning. Thus, the semantics of the strokes can be used to determine relationships to other strokes. Using the library, the instructor can draw two circles on the screen, one below and to the right of the other, and draw a line between them. These circles would be treated as nodes in the tree and the line would be treated as an edge between the two nodes. Also, the parent-child relationship between the nodes would be kept. Using this relationship information, an integer could be drawn in a specified region of the application, an "Insert" button could be pressed, and the integer would be inserted into the correct position in the tree, highlighting nodes as they are visited. Also, if the tree becomes unbalanced after the insertion, the appropriate rotation can be performed dynamically, and the resulting balanced tree will be displayed to the user. This eliminates the task of the instructor having to modify existing strokes on the screen himself. Additionally, the application can be made available to students, so they can interact with the data structures and algorithms themselves. Thus, by giving meaning to the strokes drawn by the user, the DSInk library provides for applications which can dynamically illustrate data structures and algorithms.

4. Overview of Tablet PC Platform

We suggest consulting [9] for a detailed discussion of developing Tablet PC applications. A general overview of the Tablet PC Platform follows.

The Tablet PC is a laptop computer with a digitizer integrated into the screen running Microsoft's Windows XP Tablet PC Edition, which is a superset of Windows XP. The screen can rotate and fold back onto itself. Tablet PC's also come in slate form

with no keyboard. The digitizer is a device that provides pointer (stylus, mouse, etc.) information reflecting user input to a computer to be processed. The digitizer detects *x*- and *y*- coordinates, as well as whether the pointer is active (i.e. the stylus is touching the screen) or not. In order to satisfy Microsoft guidelines, the digitizer must sample the location of the stylus at least 100 times per second, and must support a resolution of at least 600 points per inch [9]. Because of the amount of data sampled and the frequency of sampling, using the stylus on the screen remarkably resembles using a pen and paper. Even information such as pressure and angle of the stylus can be collected and used to display the digital ink. The digitizer communicates this information by constructing *packets*, which contain the *x*- and *y*- coordinate information, pressure, angle, rotation, and other information.

The Tablet PC platform is composed of three logical pieces: Ink Collection, Ink Data Management, and Ink Recognition. Each of these three pieces has its own API. Ink Collection provides the ability to receive stylus input from the digitizer. This input can be used to display the ink input by the user or to perform commands based on this input. The Ink Data Management API provides the ability to save and transform ink that is drawn by the user. These transformations include changing the color of ink, resizing ink, moving ink, selecting ink, and deleting ink, among others. Finally, the Ink Recognition API allows for the recognition of ink as text and other objects, such as shapes and gestures.

4.1 Ink Collection

The Ink Collection API has two important objects: *InkCollector* objects and *InkOverlay* objects. The *InkCollector*'s main purpose is to provide for real-time inking

into an application. It captures any input from the stylus in its *host window*, which is a rectangular region that can be attached to any Windows handle, such as a form or a panel. Unfortunately, real-time inking is the only functionality provided by the *InkCollector*. The *InkOverlay* object, on the other hand, is a superset of the *InkCollector*. The *InkOverlay* not only provides the ability for real-time inking, but it also allows the ink that has been drawn to be selected, manipulated, and deleted. The *InkOverlay* contains three editing modes that indicate the type of input behavior that is active. The first editing mode, *Ink*, simply indicates that ink will be displayed on the screen as it is drawn by the user. The *Select* editing mode provides the ability to select ink already drawn on the screen by drawing a lasso around the ink. Once the ink is selected, it can be moved or resized. The final editing mode is *Delete*. In this mode, whenever the stylus is touching the screen and touches ink on the screen, the ink will be deleted, either at the stroke level or at the point level. These objects also provide a set of events that can be used to introduce new functionality and alter the behavior of the objects. These events can be divided into the following categories: stroke and gesture events, pen movement events, mouse trigger events, tablet hardware events, rendering events, and ink editing events.

4.2 Ink Data Management

The Ink Data Management API manages and manipulates ink once it is input by the user. Three important classes are provided by the Ink Data Management API: *Ink*, *Stroke*, and *Strokes*. An *Ink* object is a container for *Stroke* objects and is automatically created whenever an *InkCollector* or *InkOverlay* object is created. It can be referenced by the *Ink* property of the *InkCollector* or *InkOverlay* objects. An *Ink* object can contain zero or more *Stroke* objects, but a *Stroke* object can be contained by exactly one *Ink*

object. A snapshot of all of the *Stroke* objects owned by a particular *Ink* object can referred to by the *Strokes* property of the *Ink* object. Also, the *Ink* class provides a number of functions that provide for the manipulation of *Strokes* collections and *Stroke* objects, such as creation, deletion, and transformation.

A *Stroke* is created whenever the stylus touches the screen, moves around, and is moved off the screen. It consists of a series of packets. Each *Stroke* object is given a specific ID within the context of its owner, an *Ink* object. This ID lasts throughout the lifetime of the *Stroke* object, even if it is moved or resized. After the *Stroke* object has been created, its color, width, transparency, etc. can be changed using its *DrawingAttributes* property.

A *Strokes* object is a collection of references to *Stroke* objects. A *Strokes* collection is created by an *Ink* object and can only contain *Stroke* objects owned by that *Ink* object. It is a convenient way to refer to a group of *Stroke* objects as a collection. However, inclusion of a *Stroke* in a *Strokes* collection has no effect on the *Stroke*'s lifetime – the lifetime of a *Stroke* is determined solely by the *Ink* object which owns it. Therefore, it is possible for a *Strokes* object to refer to a *Stroke* that has been deleted, and checks must be performed when using *Strokes* collections so unexpected behavior will not occur.

For an example of the relationship between *Ink, Stroke,* and *Strokes* objects, consider the sample of code in Figure 1.

```
1       Strokes RedStrokes = myInkOverlay.Ink.CreateStrokes();
2       for (int i = 0; i < 5; i++)
        {
3             Stroke s = myInkOverlay.Ink.Strokes[i];
4             s.DrawingAttributes.Color = Color.Red;
5             RedStrokes.Add(s);
        }
6       myInkOverlay.Ink.DeleteStroke(myInkOverlay.Ink.Strokes[0]);
```

Figure 1: Relationship between *Ink, Stroke,* and *Strokes* objects

In this example, a new *Strokes* collection, `RedStrokes`, is created (line 1). The first five *Stroke* objects that the user draws are colored red and added to `RedStrokes`. `RedStrokes` conveniently refers to all of the red strokes, while the entire set of *Stroke* objects on the screen can still be referred to by using `myInkOverly.Ink.Strokes`. Finally, the first *Stroke* is deleted (line 6), but is still referenced by the `RedStrokes` collection. Therefore, an exception would occur if `RedStroke[0]` were used in any way.

4.3 Ink Recognition

The final piece of the Tablet PC platform is the Ink Recognition API. This API provides the ability for the recognition of not only handwriting, but shapes, symbols, and gestures as well. The Tablet PC platform provides for ink recognition with software libraries called *recognizers*. These recognizers compute the textual or object representation of ink strokes for a language, which is a set of predefined words or objects that can be represented in writing. Recognizers reside in DLL files, and any number of recognizers can be installed on a system. Therefore, a single application, for example, is able to recognize English, Chinese, and shapes if the proper recognizers are installed and referenced. Microsoft ships a set of recognizers with the Tablet PC operating system, and the extensible architecture of the Tablet PC also allows for third parties to build their own

recognizers. For example, in the DSInk applications that we have developed, we use inShape™, a shape recognition library developed by the Leszynski group [10].

Recognition can be performed synchronously or asynchronously. In synchronous recognition, also known as foreground recognition, the thread that requests the recognition blocks until the recognition is completed. In asynchronous recognition, or background recognition, the thread that requests the recognition is allowed to continue to do work and is notified when the recognition is complete. Foreground recognition is easier to code and is appropriate for most ink-enabled applications. However, if an application wishes to recognize a large amount of ink input by the user, using foreground recognition may give the appearance of a hung program since the main thread blocks until results are received from the recognizer. Using background recognition in this case does not result in the appearance of a hung program because the main thread continues to do work and is notified when the recognizer provides it results.

The simplest form of text recognition takes place by calling the *ToString* method of a *Strokes* object. Calling this method sends the *Strokes* object to the default recognizer, which calculates the recognition results and returns the highest probability result as a string. This is the method we use to recognize integers in DSInk applications described in section 6. Note that more complicated recognition can be performed using *RecognizerContext* and *RecognitionResult* objects, but because we do not make use of such objects in our applications, they are beyond the realm of our discussion.

4.4 Development on the Tablet PC

Tablet PC applications can be developed on any computer running Windows XP – a Tablet PC is not needed. For systems running Windows XP or Windows XP Tablet PC

Edition, the only requirements are the Tablet PC SDK [12] and Microsoft Visual Studio 6 Service Pack 5 or later. All Tablet PC's are shipped with handwriting recognition libraries. However, users of non-Tablet PC's must download a set of recognizers [11] in order to test handwriting recognition in their applications. Although a Tablet PC is not needed for development, testing on a Tablet PC is much more realistic than on a non-Tablet PC. A mouse simply cannot provide the same user experience as using a stylus to draw on the screen. If some sort of recognition is involved in the application, testing by using a mouse may not accurately reflect this recognition capability because the mouse does not provide the same quality or quantity of data needed for ink recognition as does the stylus [5]. However, if developing on a Tablet PC is not an option, the next best alternative is the external digitizer pad, such as those manufactured by Wacom [17].

5. Approach

The DSInk library was designed with the goal of providing a useful set of tools that allows for the rapid and efficient development of applications to be used to illustrate data structures and their algorithms. Therefore, we want to be able to model nodes in any structure, edges, trees, graphs, lists, stacks, queues, and structures within computer science outside the realm of data structures, such as finite state machines and UML structures. The library was also designed with the goal of being easily modifiable and extensible. Figure 2 shows the structure of the DSInk library.

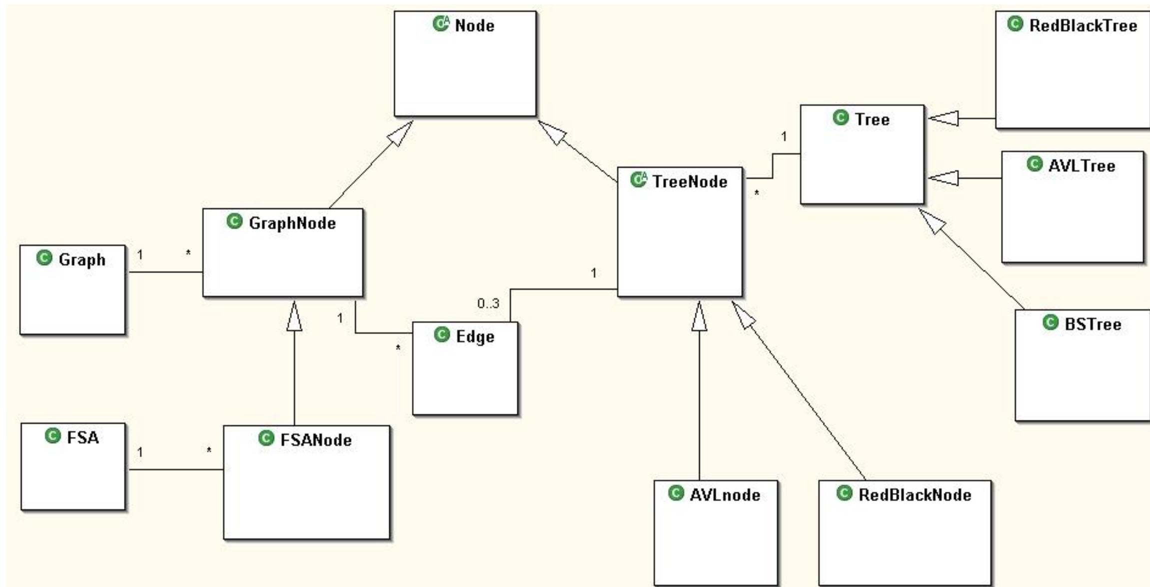

Figure 2: DSInk Class Diagram

The interested reader can view a complete reference to the DSInk library in Appendix A. Also note that the Tablet PC API reference [15] was used in the development of DSInk.

5.1 The Node Class

The *Node* class is an abstract class that can be used to illustrate any data structures which contain nodes. Linked lists, trees, heaps, and graphs can all make use of this class. Each *Node* object has three attributes which are common to all nodes. The first is a *type* string, which specifies the type of the *Node*. The next attribute is a *Stroke* object called *nodeStroke*. The *nodeStroke* is the actual *Stroke* that represents the structure of the node – a circle for a tree node, a rectangle for a linked-list node, etc. The final attribute of a *Node* object is a *Strokes* collection called *label*. This *Strokes* collection is the value associated with the node. For example, if the integer “15” is drawn inside of a tree node, then all of the *Stroke* objects which make up “15” will be contained in the *label* collection and will be associated with that node. The *Node* class provides an abstract method, *drawNode*. Each type of node will define its own *drawNode* method, which can be used

to automatically draw the node to the screen (by clicking on a button, for example) or to draw a "perfect" version of the node if a less-than-perfect representation of the node drawn by the user is recognized.

5.2 The TreeNode Class

The *TreeNode* class is a subclass of the *Node* class. A *TreeNode* object represents a binary tree node. In addition to the attributes provided by its superclass, a *TreeNode* object also has *Edge* attributes, *leftEdge, rightEdge,* and *parentEdge*. These attributes refer to the node's left child, right child, and parent, respectively. Methods are provided to *get* and *set* the edges, the two children, and the parent of the node. This class was made a superclass of other tree node classes in order to reduce the amount of duplicate code that has to be written. Thus, any type of tree node (AVL, red-black, splay, etc.) can inherit from the *TreeNode* class and use its methods. Additionally, these nodes can define attributes and methods specific to that type of node, such as height in an AVL node and color in a red-black node.

5.3 The AVLnode Class

The subclass of the *TreeNode* class is the *AVLnode* class, which represents a node in an AVL tree. In addition to the attributes provided by the *TreeNode* class, this class also provides a *height* attribute, used during insertions to detect when the AVL tree is out of balance and rotations need to be performed. The class also provides methods to *get* and *set* the height of a node.

5.4 The RedBlackNode Class

The *RedBlackNode* class also inherits from the *TreeNode* class. This class provides a *color* field which serves two purposes: to alter the color of the node and to determine

when the red-black tree is out of balance (after an insert), requiring a rotation. Methods to *get* and *set* the color are provided.

5.5 The Edge Class

An *Edge* object represents an edge between two nodes in a binary tree and has the following attributes: *edgeStroke, parentNode, childNode, startPtID,* and *endPtID.* The attribute *edgeStroke* is a *Stroke* object that represents the actual line on the screen between the two nodes. The attributes *parentNode* and *childNode* are *TreeNode* objects that refer to the parent and child node, respectively, connected by the *Edge*. Finally, *startPtID* is the ID of the point within the *nodeStroke* of *parentNode* where the *Edge* meets the *nodeStroke*, and *endPtID* is the similar point on *childNode*. Methods are provided to get and set all of the attributes listed above.

5.6 The Tree Class

The *Tree* Class is an abstract class which is used to model any type of binary tree. A *Tree* object maintains a set of *TreeNode* objects in an array list which can grow and shrink dynamically. Storing the *TreeNode* objects in an array list allows for convenient access to all of the nodes in the tree without having to perform a traversal. This class provides a number of useful methods that are common to all binary trees. First, methods are provided that add and remove nodes from the arraylist. This class also provides methods, *drawNewLeftChild* and *drawNewRightChild*, that automatically draw left and right children, respectively, given a particular *TreeNode* object. These two methods are important in the case where a new node needs to be added to the tree and displayed on the screen without the user explicitly drawing it, such as in an insert. Another important method that is useful when any self-balancing tree becomes out of balance is the

getSubtree method. Given a particular node, this method collects into one *Strokes* collection all of the *Stroke* objects associated with that node as well as all of the *Stroke* objects associated with the subtree rooted at that node. The pseudocode for the *getSubtree* algorithm is shown in Figure 3.

```
1     getSubtree(TreeNode t, Strokes s)
      {
2           if t is null
3                 return
4           add t's nodestroke to s
5           add t's label to s
6           if t has a left child
7                 add t's left edge to s
8           call getSubtree with t's left child
9           if t has a right child
10                add t's right edge to s
11          call getSubtree with t's right child
      }
```

Figure 3: Pseudocode for *getSubtree* method

The importance of the *getSubTree* method will be seen more clearly when we discuss AVL and red-black trees. The *Tree* class also provides a number of traversal methods (*preOrder, postOrder, inOrder,* and *reversePostOrder*), which are common to all types of binary trees. Other traversal methods can easily be added to the *Tree* class. Another important method provided by the *Tree* class is the *findMin* method. Given a node in a binary search tree, this method returns the minimum value of the subtree rooted at the node. This method is used in the removal of a node with two children from the tree, as will be discussed in section 5.7. The *Tree* class defines two abstract methods, *insert* and *remove*, which must be implemented by each of its subclasses. The *insert* method inserts a new value into the existing tree, and the *remove* method removes a value from the tree.

These operations are different for each type of tree, and, therefore, each subclass of the *Tree* class must implements its own *insert* and *remove* methods.

5.7 The BSTree Class

The *BSTree* class is a subclass of the *Tree* class that represents a binary search tree. A binary search tree has the following properties:

- Given a node *n*, the value of the left child of *n* is smaller than the value of *n*.
- Given a node *n*, the value of the right child of *n* is greater than the value of *n*.

A *BSTree* object contains zero or more *TreeNode* objects. This class, in addition to the methods provided by its super class, provides methods to insert and remove *TreeNode* objects to/from the tree. The *insert* method is based on Weiss's algorithm [18] and is described in the Figure 4 below.

```
1      insert(TreeNode t, Strokes s)
       {
2            if t is null
             {
3                  create a new node
4                  add node to the tree
5                  display new node, edge, and label on the screen
             }
6            else if value of s is less than value of t
             {
7                  highlight node t
8                  insert(t.left, s)
             }
9            else if value of s is greater than value of t
             {
10                 highlight node t
11                 insert(t.right, s)
             }
12           else
13                 display message that value is a duplicate
       }
```

Figure 4: Pseudocode for *insert* method in *BSTree* class

Note that the *Strokes* collection *s* represents the integer that the user has drawn. The collection *s* is parsed as an integer by the following statement:

```
int num = int.Parse(s.ToString());
```

Calling *s.ToString()* simply uses the built-in handwriting recognizers to recognize and return a String representation of the collection *s*. Also, notice that the nodes in the tree are highlighted as they are visited on the insert (line 7, 10). The *remove* method is also based on Weiss's remove algorithm [18].

```
1       remove(TreeNode t, Strokes s)
        {
2             if t is null
3                   value not in tree - return
4             if value of s is less than value of t
              {
5                   highlight node t
6                   remove(t.left, s)
              }
7             else if value of s is greater than value of t
              {
8                   highlight node t
9                   remove(t.right, s)
              }
10            else if t has two children
              {
11                  highlight node t
12                  min → findMin(t.right)
13                  remove(t.right, min)
              }
14            else if t has a right child
              {
15                  highlight node t
16                  remove t from tree
17                  draw edge from t's parent to t's right child
              }
18            else if t has a left child
              {
19                  highlight node t
20                  remove t from tree
21                  draw edge from t's parent to t's left child
              }
22            else
              {
23                  highlight node t
24                  remove t from tree
              }

        }
```

Figure 5: Pseudocode for *remove* method in *BSTree* class

The *Strokes* collection *s* is treated just the same as in the *insert* method. Also, the nodes are, again, highlighted as they are visited to better illustrate the algorithm (line 5, 8, 11, 15, 19, 23).

5.8 The AVLTree Class

The *AVLTree* class is also a subclass of the *Tree* class. An *AVLTree* object represents an AVL tree, which is a binary search tree with the following additional property: for an arbitrary node *n* in the tree, the height of the left subtree of *n* and the height of the right subtree of *n* can differ by at most 1 [18]. Note that the height of an empty subtree is defined to be –1 [18]. So, height information must be kept in an AVL tree, and this height information is kept in each *AVLnode* object which is contained in the *AVLTree*. The *AVLTree* class extends the *Tree* class by provided a number of methods used in the insertion of values into an AVL tree. The *insert* method is based on an algorithm proposed by Weiss [18] and is shown in Figure 6.

```
1    insert(TreeNode t, Strokes s)
     {
2          if t is null
           {
3                create/display new node
                 make it the appropriate child of its parent
           }
4          else if value of s is less than value of t
           {
5                highlight node t
6                insert(t.left, s)
7                if height(t.left) – height(t.right) = 2
                 {
8                     if value of s is less than value of t.left
9                            rotateWithLeftChild(t)
                 }
10               else
11                    doubleWithLeftChild(t)
           }
12         else if value of s is greater than value of t
           {
13               highlight node t
14               insert(t.right, s)
15               if height(t.right) – height(t.left) = 2
                 {
16                     if value of s is greater than value of t.right
17                     rotateWithRightChild(t)
                 }
18               else
19                     doubleWithRightChild(t)
           }
20         else
21               display message indicating duplicate value in tree
22         t.height → max (height(t.left), height(t.right)) + 1
     }
```

Figure 6: Pseudocode for *insert* method in *AVLTree* class

Notice that after a value is inserted into the tree, that node, *n,* is a leaf in the tree, and, hence, the height of *n* will be 0 (because the height of both of its children is –1). Next, *n*'s parent checks the height of both of its children (line 7, 15). If these heights differ by two, the tree is out of balance and a rotation needs to be made at *n*'s parent. Thus, one of the four rotation methods, *rotateWithLeftChild, doubleWithLeftChild, rotateWithRightChild,* or *doubleWithRightChild* will be called. We will discuss the first two of these, as the last two are similar. The *rotateWithLeftChild* method will be called

when, given a node *n*, an insertion into the left subtree of the left child of *n* causes *n* to become unbalanced. This method is based loosely on Weiss's algorithm [18] and is described in Figure 7.

```
1      rotateWithLeftChild(AVLnode n)
       {
2          Strokes rightsubtree → n and all strokes in n's right
             subtree
3          Strokes leftsubtree → n.left and all strokes in
             n.left's left subtree
4          AVLnode x → n.left
5          n.left → x.right
6          x.right → n
7          move rightsubtree and leftsubtree to proper positions
             to show this rotation
8          n.height → max (height(n.left), height(n.right)) + 1
9          x.height → max (height(x.left), height(n)) + 1
       }
```

Figure 7: Pseudocode for *rotateWithLeftChild* method

Basically, this method performs a single rotation with node *n* and its left child. The method *doubleWithLeftChild* is called when, given a node *n,* an insertion into the right subtree of the left child of *n* causes *n* to become unbalanced. The method simply performs two subroutine calls:

```
1      doubleWithLeftChild(AVLnode n)
       {
2              rotateWithRightChild(n.left)
3              rotateWithLeftChild(n)
       }
```

Figure 8: Pseudocode for *doubleWithLeftChild* method

As mentioned previously, the methods *rotateWithRightChild* and *doubleWithRightChild*, perform similar functions as the prior two methods with a node *n* and its right child.

5.9 The RedBlackTree Class

The *RedBlackTree* class is also a subclass of the *Tree* class. A *RedBlackTree* object represents a red-black tree, which is a binary search tree with the following additional properties [18]:

- Every node is colored either red or black
- The root is black.
- If a node is red, its children must be black.
- Every path from a node to a *null* reference must contain the same number of black nodes.

Therefore, after *RedBlackNode* is inserted into a *RedBlackTree* object, a rotation may be required, and colors of nodes may have to change in order to satisfy the properties above. Thus, a number of methods, which are used in order to perform inserts, are provided by the *RedBlackNode* class. The class implements the abstract *insert* method of the *Tree* class using the algorithm, adapted from [8], described in Figure 9.

```
1   insert(TreeNode t, Strokes s)
    {
2         n → binarysearchtreeinsert(t, s)
3         n.color → red
4         while n != root and n.parent.color = red
5               if n.uncle != null and n.uncle.color = red
6                   n.parent.color → black
7                   n.uncle.color → black
8                   n.grandparent.color → red
9                   n → n.grandparent
                }
10              else
11                  if isLeftChild(n.parent) and !isLeftChild(n)
                    {
12                          n → n.parent
13                          rotateWithRightChild(n)
                    }
14                  else if !isLeftChild(n.parent) and isLeftChild(n)
15                          n → n.parent
16                          rotateWithLeftChild(n)
                    }
17                  n.parent.color → black
18                  n.grandparent.color → red
19                  if isLeftChild(n.parent)
20                          rotateWithLeftChild(n.grandparent)
21                  else
22                          rotateWithRightChild(n.grandparent)
                }
          }
23        root.color → black
    }
```

Figure 9: Pseudocode for *insert* method in *RedBlackTree* class

Note that the rotate methods are similar to the rotate methods in the *AVLTree* class, except that they do not update any height information for a node. The *isLeftChild* method returns *true* if the *RedBlackNode* passed in is a left child of its parent or *false* otherwise.

6. Results

We have designed and implemented an application to test the functionality and ease-of-use of the DSInk library. The purpose of the application is to illustrate the algorithms that can be performed on binary search trees, AVL trees, and red-black trees. These algorithms include inserting a value into the tree, removing a value from the tree,

and performing a traversal. The application is a Windows Form. It uses an *InkOverlay* object to capture ink, and a *Panel* is set as the *InkOverlay*'s host window.

6.1 Shape Recognition

As previously mentioned, the application uses the Leszynski inShape™ [10] library to recognize shapes drawn by the user. This library provides shape recognition capabilities for a number of different shapes. In order to utilize this library in the application, an event handler is called each time a new *Stroke* is drawn by the user. Inside this event handler, a *Strokes* collection, *recoStrokes,* is created and the last *Stroke* drawn by the user is added to this collection:

```
recoStrokes.Add
   (myInkOverlay.Ink.Strokes[myInkOverlay.Ink.Strokes.Count-1]);
```

Next, an attempt is made to recognize the *Strokes* collection with the statement:

```
RecoResult result =
      reco.Recognize(recoStrokes, this.simpleShapes);
```

If the result is a circle, then a method that handles circle recognition is called. Likewise, if the result is a line, then a method that handles line recognition is called.

6.2 Modes

The application makes three different modes available to the user: 1) binary search tree mode, 2) AVL tree mode, and 3) red-black tree mode. These modes can be changed by accessing the "Tree Mode" menu item at the top of the window. The behavior of the application varies slightly depending on which mode the user selects.

6.2.1 Binary Search Tree Mode

If the application is in "Binary Search Tree" mode, then a new *TreeNode* object is created when a stroke is recognized as a circle. This new *TreeNode* object calls its *drawNode* method which draws a "perfect" circle in place of the skewed circle that the

user has drawn. Also, this *TreeNode* is added to a global *BSTree* object, *bstree*. The user can add nodes to the tree in two different ways.

First, the user can draw nodes, edges, and labels. For example, suppose the user draws two circles – one at the top, center of the window and another slightly below and to the left of the first circle. Both of these circles are recognized, and two *TreeNode* objects are created and added to *bstree*. However, there is no relationship between the two nodes. Now, suppose the user draws a line from the left side of the upper node to the top of the lower node, as can be seen in Figure 10. First, the *Stroke* is recognized as a line. Next, the *Stroke* objects nearest to the head and tail of the line are found. If both of these objects are *nodeStroke*'s within *bstree*, meaning they are circles that the user has drawn, a new *Edge* object is created with the line as the parameter. Next, the skewed line is redrawn as a straight line and the ends of the line "snap" to the *nodeStroke*'s of the two *TreeNode* objects.

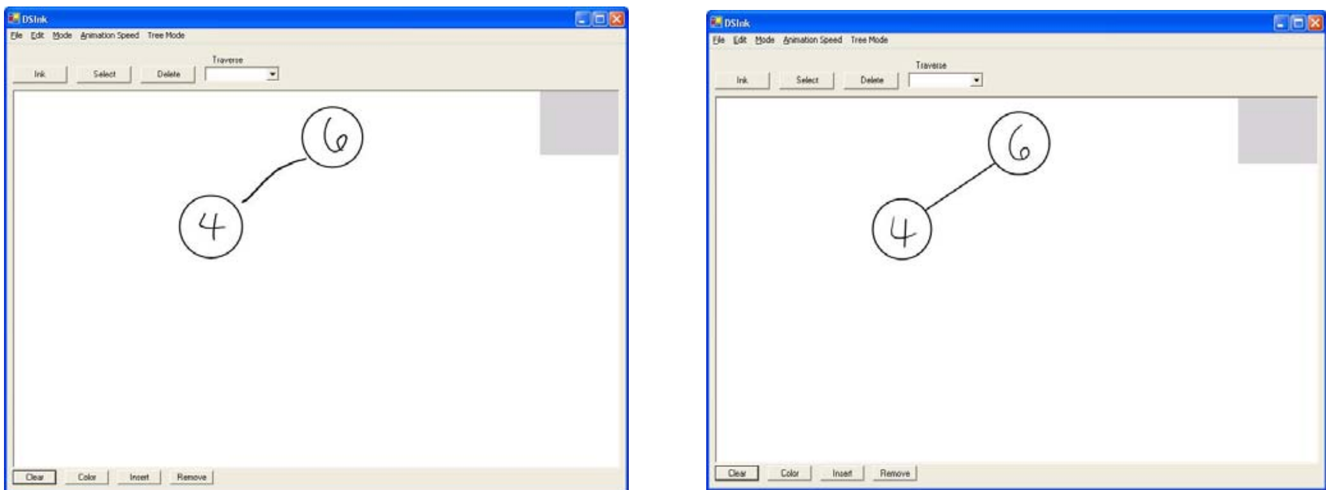

Figure 10: Recognizing a line as an edge between two nodes

Also, since there is now a relationship between the *TreeNode* objects, this information must be updated. First, the *parent* of the *Edge* is set to the *TreeNode* object that is higher

in the window than the other object, and the *child* of the *Edge* is set to the lower *TreeNode*. These positions are determined by looking at the *y*-values of the endpoints of the line. Next, the *startPointID* field of the *Edge* is set. This corresponds to the point ID on the *nodeStroke* of the *parent* where the *Edge* and *nodeStroke* meet. The *endPointID* is similarly set using the *child* of the *Edge.* These ID's are needed in case the user selects and moves a *nodeStroke*, which will be discussed shortly. Next, the *parentEdge* field of the lower *TreeNode* is set to the *Edge* object. Finally, the *x*-value of the endpoint of the line with the lower *y*-value (i.e. top point of the line) is compared with the *x*-value of the center of the top circle. Since, the *x*-value of the line is less than the *x*-value of the center, the *leftEdge* of the upper *TreeNode* is set to the *Edge* object. If the top of the line had been connected to the right side of the circle, then the *Edge* would have been a *rightEdge* to the upper *TreeNode*. Note that integers can be drawn inside the nodes and will be set as the *TreeNode* object's *label*.

The second way to add a node to the binary search tree is by pressing the "Insert" button at the bottom of the window. The user must draw an integer in the gray box at the upper right corner of the window. When the "Insert" button is pressed, this integer will be inserted into the tree in binary search order. First, the *insert* method for *bstree* is called with the *Strokes* collection in the gray box as a parameter. The *Strokes* collection is parsed as an integer and is inserted correctly into the tree as described in Section 5.7. The nodes are highlighted as they are visited to better illustrate the insert algorithm. Note that the appropriate fields are set for the *Edge* and two *TreeNode* objects where the insertion takes place.

Values (and nodes) can also be removed from the binary search tree in a manner similar to the insertion process above. First, the user draws an integer in the gray box at the top right corner. Then, the user presses the “Remove” button at the bottom of the window. When this happens, the *remove* method for *bstree* is called, and the *TreeNode* containing the integer the user has drawn is removed from the tree according to the algorithm described in section 5.7.

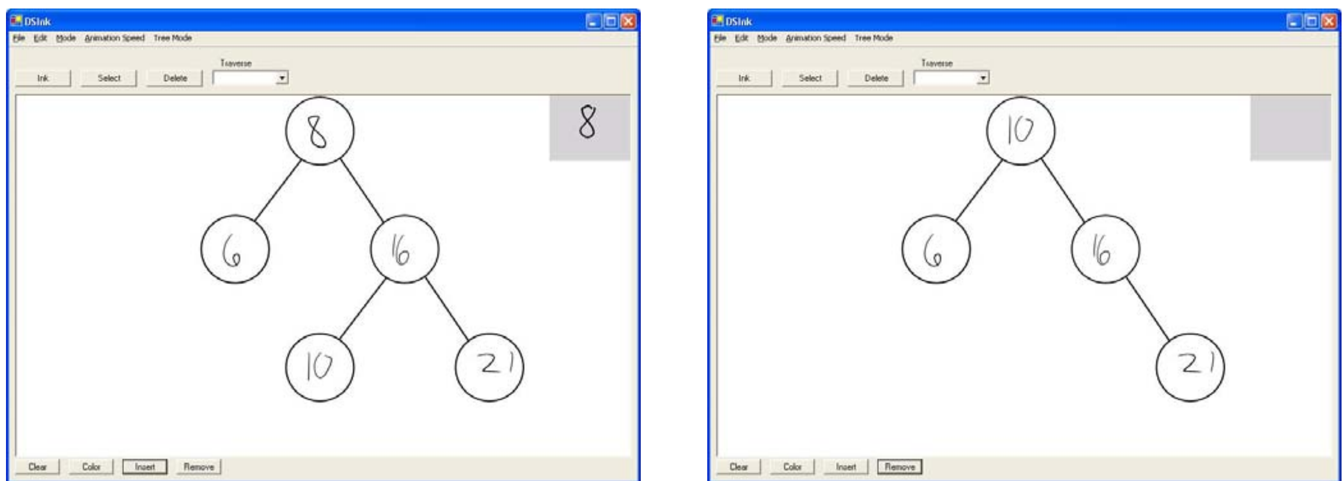

Figure 11: Binary search tree before and after removal of value 8

6.2.2 AVL Tree Mode

The user can also select “AVL Tree” mode from the “Tree Mode” menu. If this mode is chosen, the application behaves slightly differently. First, if the user draws circles, they are recognized and new *AVLnode* objects are created and added to an *AVLTree* object, *avltree.* The user can draw edges between nodes just as described before. The next difference found in the “AVL Tree” mode is the way in which the “Insert” button is handled. As in the previous mode, the user draws the integer to insert in the gray box at the top right corner of the window. However, when the “Insert” button

is pushed, the *insert* method for *avltree* is called instead of *bstree.insert().* This method inserts a value into *avltree* as described in Section 5.8. Again, the nodes are highlighted as they are visited to illustrate the insertion algorithm. If an insertion into the tree causes a node to become unbalanced, a message is printed to the screen saying which type of rotation is about to occur, and that type of rotation is performed. Figure 12 shows an AVL tree just before a rotation is to be performed.

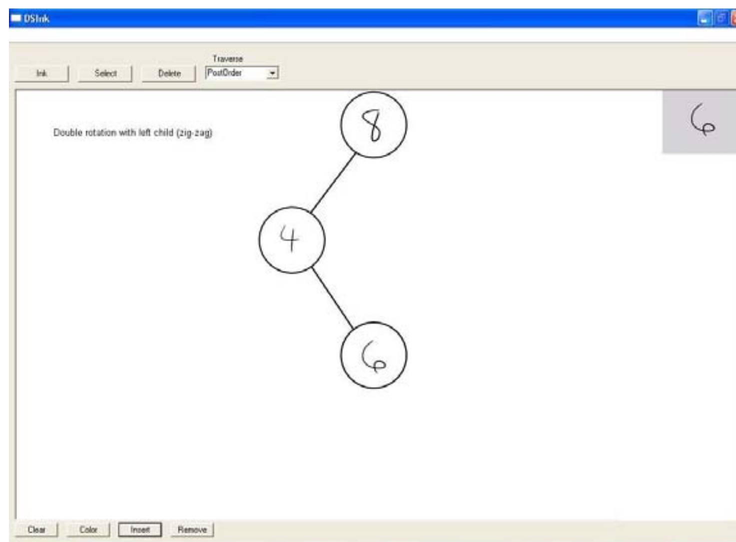

Figure 12: AVL Tree During Insert

Note that the text at the upper left corner of the window (clearly visible when using the application) states, “Double Rotation with left child”. Also, after the insertion, the height of each node is printed just outside the node. Figure 13 shows the AVL tree after the value “6” is inserted and a rotation has been performed.

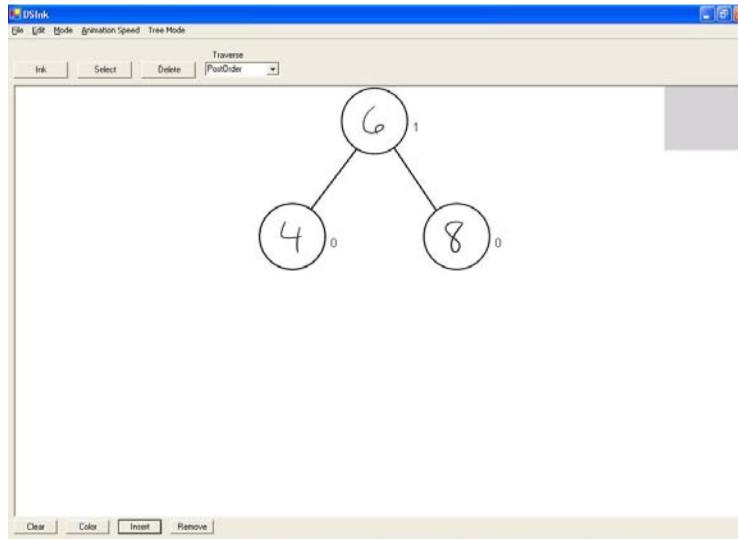

Figure 13: AVL Tree After Rotation

6.2.3 Red-Black Tree Mode

A third tree mode that the user can select is "RedBlack Tree" mode. After circles drawn by the user are recognized, *RedBlackNode* objects are created an inserted into a *RedBlackTree* object, *rbtree.* Again, the treatment of edges is similar to the previous two modes. If the "Insert" button is pressed, then the *insert* method for *rbtree* will be called. The integer drawn by the user will be inserted into *rbtree* in the manner described in Section 5.9. As in the two previous *insert* methods, the nodes in the tree are highlighted as they are visited. After an insertion, any nodes that need to change color do so, and any rotations that need to be made because of unbalanced nodes are performed. Figure 14 shows an insert into a red-black tree and a subsequent rotation.

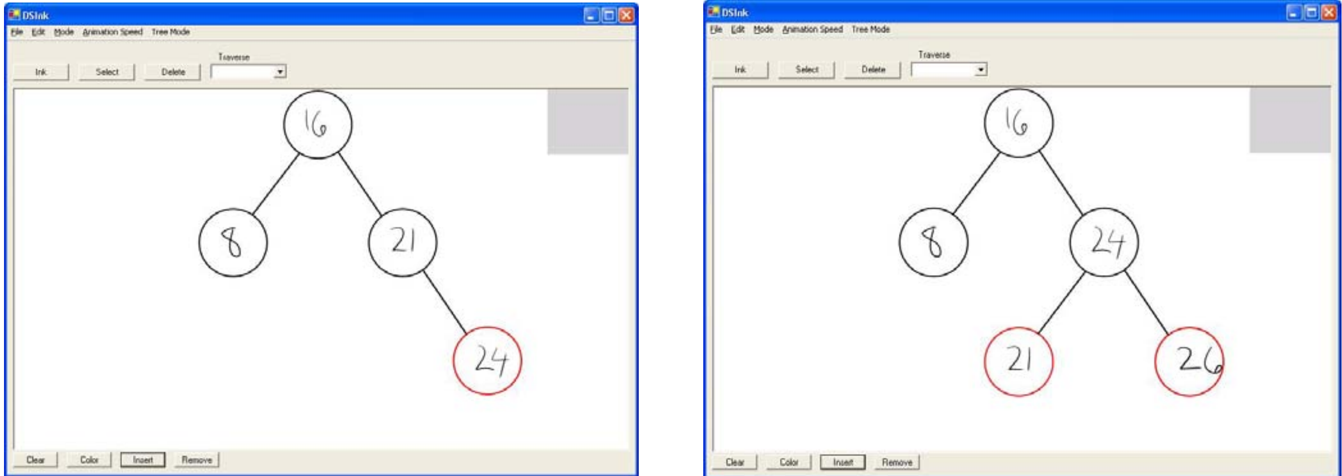

Figure 14: Red-black tree before and after insertion of value 26

6.3 Selecting and Moving Nodes

A node can be selected and moved around in the window, and its edges will be redrawn in the correct locations. The keys to this process are the *startPointID* and *endPointID* fields of the *Edge* object. A *SelectionMoved* method handles this case. First, the individual *Stroke* objects in the *myInkOverlay.Selection* collection are compared against the *nodeStroke* objects in the tree to determine if a *Stroke* that has been selected is indeed a *nodeStroke*. If so, then the edges of that node can simply be redrawn. For example, consider the node containing value "6" in Figure 15 below.

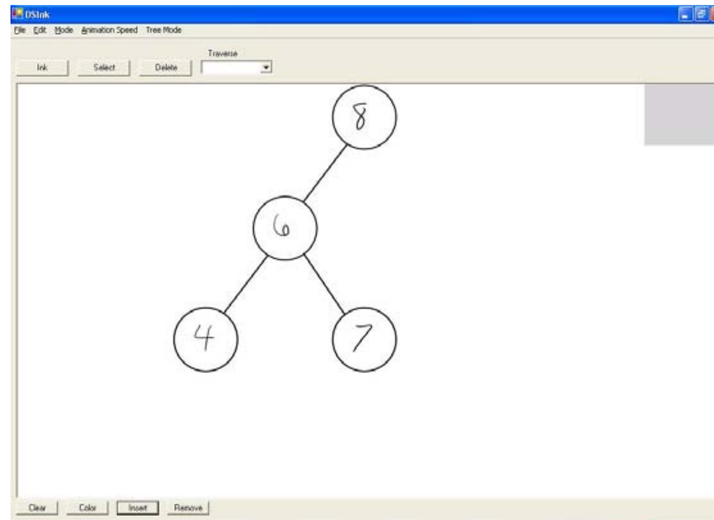

Figure 15: Tree before selection

It has three *Edge* objects associated with it: a *parentEdge,* a *leftEdge*, and a *rightEdge*. When the node is moved, the *edgeStroke* of the *parentEdge* is deleted and a new *Stroke* is drawn as follows:

```
1      startID = node.getParentEdge().getStartPointId();
2      endID = node.getParentEdge().getEndPointId();
3      pts[0]= node.getParent().getNodeStroke().GetPoint(startID);
4      pts[1] = node.getNodeStroke().GetPoint(endID);
5      edgeStroke = myInkOverlay.Ink.CreateStroke(pts);
6      node.getParentEdge().setEdgeStroke(edgeStroke);
```

Figure 16: Code for redrawing an edge between two nodes after a node has moved

This new *Stroke* is set as the *edgeStroke* of *parentEdge* (line 6). The other two edges are redrawn in the same manner. It is important to note that any or none of the three edges could have also been selected and the results would be the same. The resulting tree after the node has been moved is shown in Figure 17.

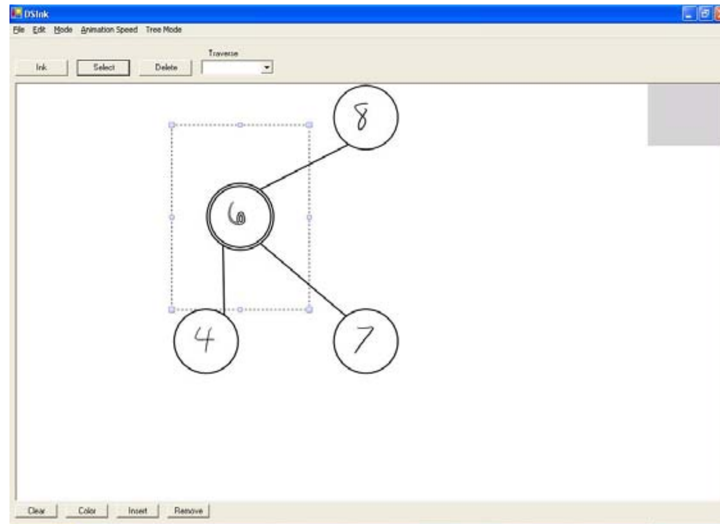

Figure 17: Tree after node 6 has been selected and moved

6.4 Traversals

The user has four different traversals to choose from that can be performed on a tree in any mode: 1) in-order, 2) pre-order, 3) post-order, and 4) reverse post-order. When the user selects one of these from the combo box at the top of the window, the *Traverse* method is called with the type of traversal to perform as input. This method then calls the appropriate traversal method inside the *Tree* class. As the nodes are visited when the traversal is being performed, they are highlighted red so the user can clearly see the order of the traversal. Also, the values of the nodes in the order they are visited are printed at the top left corner of the window after the traversal is finished. Figure 18 shows a red-black tree after a post-order traversal.

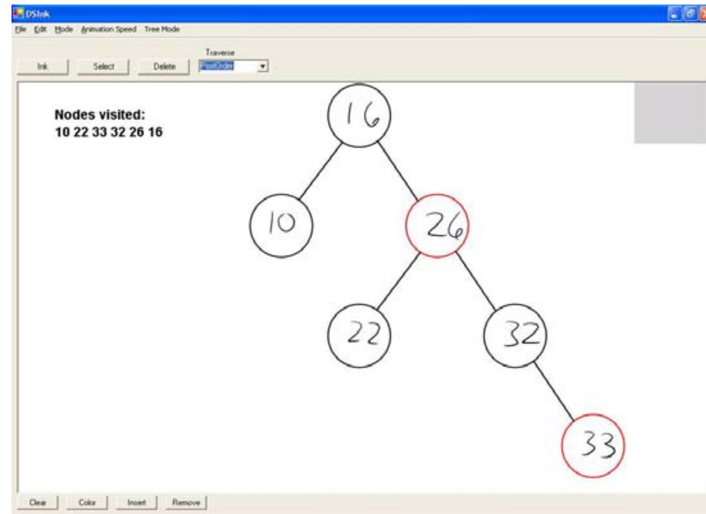

Figure 18: Red-black tree after post-order traversal

Note that the text at the upper left corner reads, "Nodes visited: 10 22 33 32 26 16". The speed at which the nodes are highlighted and at which the traversal is performed can be adjusted by selecting "Slow", "Medium", or "Fast" from the "Animation Speed" menu.

7. Conclusions and Future Work

There is great potential for the Tablet PC as an instructional tool in the classroom – we have only touched the surface in discovering its capabilities. Specifically, we have made the following accomplishments in our research:

1) We have developed a library of classes that model structures by giving meaning to user-drawn ink.

   i) The *Node* Class
   ii) The *TreeNode* Class
   iii) The *AVLnode* Class
   iv) The *RedBlackTree* Class
   v) The *Edge* Class
   vi) The *Tree* Class
   vii) The *BSTree* Class
   viii) The *AVLTree* Class
   ix) The *RedBlackTree* Class

2) We have developed a number of algorithms for dynamically showing operations performed on data structures.

   i) Insert, remove, and find for binary search trees.
   ii) Insert and find for AVL trees.
   iii) Rotations for AVL trees.
   iv) Insert and find for red-black trees.
   v) Rotations and re-coloring for red-black trees.
   vi) Traversals for any type of binary tree.

3) We have developed applications which make use of the DSInk library.

   i) Binary Search Tree
   ii) AVL Tree
   iii) Red-black Tree.
   iv) Finite State Machine (currently being developed)

4) We have provided several templates for development of additional structures in the future.

5) We have constructed a webpage consisting of numerous links regarding Tablet PCs and development on Tablet PCs.

The Tablet PC may very well turn out to be the standard teaching device in the classroom of the future. We have developed the DSInk library as an initial attempt to exploit the powerful capabilities provided by the Tablet PC. Overall, we feel that the DSInk library has proved to be an effective tool for rapidly and efficiently developing data structures applications for many reasons. First, the DSInk library improves upon many currently used instructional tools by giving user-drawn ink semantic properties. These semantic properties given to the ink on the screen are the keys to clearly illustrating the structures and dynamically performing algorithms on them. Second, the library proved to be straightforward and easy to use. Applications which make use of the library can be developed quickly and easily. Also, the library is easily modifiable. It can be extended to allow other structures and algorithms to be created. Other algorithms can

easily be added to existing classes as well. For example, an algorithm which illustrates depth first search could be trivially added to the *Tree* class.

The classroom lecture will also be improved when DSInk applications are introduced in the classroom. First, the flow of the lecture will be improved because the instructor now has a seamless way to illustrate structures and their algorithms. Also, students will pay more attention to the lecture because of the innovative technology being used. If the applications are made available to the students, then learning will be enhanced outside the classroom as well.

In the future, we plan to extend the DSInk library to include more data structures and algorithms and to include structures within computer science outside the realm of data structures, such as finite state machines. We also plan to test applications built with the DSInk library in computer science classes at Clemson University for student feedback. We will describe the general steps necessary to add the following structures to the DSInk library: leftist heaps, linked lists, graphs, and finite state machines.

A node in a leftist heap is identical to a node in a binary tree with an additional *null path length* property [18], which is defined as the length of the shortest path from a node, *n*, to a node without two children. So, we will define a *LeftistHeapNode* class which extends the *TreeNode* class. This class will include an integer attribute, *npl*, which holds the null path length of the node. We will also create a *LeftistHeap* class which extends the *Tree* class. A *LeftistHeap* object will contain zero or more *LeftistHeapNode* objects. The *LeftistHeap* class will contain methods to insert values, merge two heaps, and delete values from the leftist heap.

The *LinkedListNode* class will extend the *Node* class. This class will also contain an *Edge* attribute, *next*, which will represent a pointer to the next node in the list. The *LinkedList* class will represent a linked list. An object of this type will contain zero or more *LinkedListNode* objects. In the application which makes use of the *LinkedList* class, we will use the Leszynski shape recognition library to recognize rectangles drawn by the user. If a rectangle is recognized, a *LinkedListNode* will be created, a perfect representation of the rectangle will be drawn in place of the user's, and this *Stroke* will be set as the *nodeStroke* of the *LinkedListNode*. If a line is drawn from one node to another, the node on the left will be treated as the *parent* of the *Edge* object and the node on the right will be treated as the *child*. The *startPointID* and *endPointID* will be set in the same manner in which they are set in the tree examples. Finally, a straight arrow will replace the user's skewed line and will snap to the edges of the nodes. The *LinkedList* class will contain methods to insert values at the head of the list and at the tail of the list. Methods will also be provided to delete values from both the head and tail of the list. These methods can be animated in a similar manner to that provided by the tree application.

A graph can be represented with the classes *GraphNode* and *Graph*. The *GraphNode* class will extend the *Node* class by containing an additional attribute, *edge*. Since a node in a graph can have any number of incoming and outgoing edges, this *edge* attribute will be an *ArrayList* of *Edge* objects. Note that because labels are associated with edges in graphs, we will add a *Strokes* collection *label* to the *Edge* class. The *Graph* class will maintain an *ArrayList* of *GraphNode* objects called *graph*. Methods will be provided to display the adjacency matrix and adjacency list representation of the graph.

Also, methods to illustrate algorithms such as Dijkstra's algorithm, Prim's algorithm, and Kruskal's algorithm will be provided by the class. In a graph application, we will again rely on the Leszynski shape recognition library to recognize circles, lines, and arcs. We will provide two modes to the user: directed graph mode and undirected graph mode. If a *Stroke* is recognized as a circle, a new *GraphNode* will be created and added to *graph.* If a line or arc is drawn between two nodes, an *Edge* object will be created. The properties of the *Edge* object will be set according to the direction the line was drawn. If the line was drawn from left to right, then the node on the left will be set as the *parent* of the *Edge* and the node on the right will be set as the *child* of the *Edge.* Also, the line will snap to the edges of the nodes, and the line will be redrawn as an arrow if we are in "Directed Graph" mode. To associate a label with an edge, we will look at the bounding box of the edge, which is the smallest rectangle that completely encloses the edge. If part of a *Stroke* is within the bounding box of the edge, we will consider it the *label* of the edge. Note that measures will be taken to ensure that non-labels that are drawn within bounding box (another edge, for example) are not considered to be *label* of the edge. Given the nodes, edges, and labels, methods to illustrate the various graph algorithms will be straightforward to implement.

A finite state machine will be constructed in a manner similar to the graph. An *FSANode* class will extend the *GraphNode* class by providing two additional Boolean attributes, *isStartState* and *isFinalState*. These attributes will reflect whether the *FSANode* is a start state or a final state, respectively. The *FSA* class will contain an *ArrayList* of *FSANodes*. It will also contain a method that takes in an input string from the user and determines whether the string is accepted or not. This method will clearly

illustrate the transitions made between states (via highlighting, etc.). In the application, edges between nodes will be treated in the same manner as in the graph application (in "Directed Graph" mode). Also, in order to designate a node as a start state or a final state, the user will select the node and press the appropriate "Start State" or "End State" button included in the application. These buttons will simply set the appropriate Boolean values of the *FSANode* object.

8. References

[1] Anderson, R., Anderson, R., Simon, B., Wolfman, S.A., VanDeGrift, T., and Yasuhara, K. Experiences with a Tablet PC Based Lecture Presentation System in Computer Science Courses. *35th SIGCSE*, March 2004.

[2] Anderson, R.J., Hoyer, C., Wolfman, S.A., and Anderson, R. A Study of Digital Ink in Lecture Presentation. *CHI 2004*, April 2004.

[3] Berque, D., Bonebright, T., and Whitesell, M. Using Pen-Based Computers Across the Computer Science Curriculum. *35th SIGCSE*, March 2004.

[4] Buckalew, C., and Porter, A. The Lecturer's Assistant. *25th SIGCSE,* pp. 193-197, 1994.

[5] Goldberg, Arin. Developing Tablet PC Software by Using the Windows XP Tablet PC Edition 2005 Recognition Pack http://msdn.microsoft.com/mobility/tabletpc/default.aspx?pull=/library/en-us/dntablet/html/tabpc05rcpk.asp

[6] Golub, E. Handwritten Slides on a TabletPC in a Discrete Mathematics Course. *35th SIGCSE*, March 2004.

[7] Griswold, W., Shanahan, P., Brown, S., Boyer, J., Ratto, M., Shapiro, R., and Truong, T. ActiveCampus – Experiments in Community-Oriented Ubiquitous Computing. *IEEE Computer*, pp. 73-81, October 2004.

[8] Cormen, T.H., Leiserson, C.E., and Rivest, R.L. Introduction to Algorithms. McGraw-Hill, 1990.

[9] Jarrett, R. and Su, P. Building Tablet PC Applications. Microsoft Press, 2003.

[10] Leszynski inShape™ Developer Preview Edition. http://www.leszynski.com/tabletpc/tpcinShape.htm.

[11] Microsoft Windows XP Tablet PC Edition 2005 Recognizer Pack. http://www.microsoft.com/downloads/details.aspx?FamilyId=080184DD-5E92-4464-B907-10762E9F918B&displaylang=en

[12] Microsoft Windows XP Tablet PC Edition Software Development Kit 1.7. http://www.microsoft.com/downloads/details.aspx?familyid=b46d4b83-a821-40bc-aa85-c9ee3d6e9699&displaylang=en

[13] Mock, Kenrick. Teaching with Tablet PC's. Journal of Computing Sciences in Colleges, vol. 20, issue 2, December 2004.

[14] Simon, B., Anderson, R., Hoyer, C., and Su, J. Preliminary Experiences with a Tablet PC Based System to Support Active Learning in Computer Science Courses. *ITICSE '04*, June 2004.

[15] Tablet PC API Reference. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tpcsdk10/lonestar/unmanaged_ref/tbidxmainreference.asp

[16] Timmins, S.J. Tablet PC: Blackboard to the Web. *SIGUCCS '04*, October 2004.

[17] Wacom. http://www.wacom.com

[18] Weiss, M.A. Data Structures & Algorithm Analysis in Java. Addison-Wesley, 1999.

[19] Wilkerson, M., Griswold, W. G., and Simon, B. Ubiquitous Presenter: Increasing Student Access and Control in a Digital Lecturing Environment. *36th SIGCSE*, February 2005.

[20] Willis, C.L. and Miertschin, S.L. Tablet PC's as Instructional Tools or the Pen is Mightier than the 'Board! *SIGITE '04*, October 2004.

Appendix A – DSInk Library Reference

This appendix is a complete reference to the DSInk library. The classes that make up the library, along with their instance variables and methods, are listed below.

The *Node* Class

The *Node* class is an abstract class that can be used to illustrate any data structures which contain nodes. The class contains the following instance variables:

```
string  type;
Stroke  nodeStroke;
Strokes label;
```

The following methods are included in the *Node* class:

drawNode

```
public abstract void drawNode
      (InkOverlay myInkOverlay, Panel inkPanel, Stroke inStroke)
```

This is an abstract method which each of its subclasses must implement to draw the *Stroke* that represents the shape of the *Node*.

getNodeStroke

```
public Stroke getNodeStroke()
```

This method returns the *nodeStroke* of the *Node* object.

getType

```
public string getType()
```

This method returns the *type* of the *Node* object.

getLabel

```
public Strokes getLabel()
```

This method returns the *Strokes* collection, *label*, associated with the *Node* object.

setStroke

```
public void setStroke(Stroke inType)
```

This method sets the *nodeStroke* of the *Node* object.

setType

```
public void setType(Stroke inStroke)
```

This method sets the *nodeStroke* of the *Node* object.

The *TreeNode* Class

The *TreeNode* class is a subclass of the *Node* class. A *TreeNode* object represents a binary tree node. In addition to the attributes provided by its superclass, a *TreeNode* object also has the following attributes:

```
private Edge   leftEdge,
               rightEdge,
               parentEdge;
private int    radius;
```

The following methods are included in the *TreeNode* class:

getLeftChild

```
public TreeNode getLeftChild()
```

This method returns the left child of the *TreeNode* object.

setLeftChild

```
public void setLeftChild(TreeNode t)
```

This method sets the left child of the *TreeNode* object to t.

getLeftEdge

```
public Edge getLeftEdge()
```

This method returns the left edge of the *TreeNode* object.

setLeftEdge

```
public void setLeftEdge(Edge e)
```

This method sets the left edge of the *TreeNode* object to e.

getRightChild

```
public TreeNode getRightChild()
```

This method returns the right child of the *TreeNode* object.

setRightChild

```
public void setRightChild(TreeNode t)
```

This method sets the right child of the *TreeNode* object to t.

getRightEdge

```
public Edge getRightEdge()
```

This method returns the right edge of the *TreeNode* object.

setRightEdge

```
public void setRightEdge(Edge e)
```

This method sets the right edge of the *TreeNode* object to e.

getParent

```
public TreeNode getParent()
```

This method returns the parent of the *TreeNode* object.

setParent

```
public void setParent(TreeNode t)
```

This method sets the parent of the *TreeNode* object to t.

getParentEdge

```
public Edge getParentEdge()
```

This method returns the parent edge of the *TreeNode* object.

setParentEdge

```
public void setParentEdge(Edge e)
```

This method sets the parent edge of the *TreeNode* object to e.

drawCircle

```
public Stroke drawCircle
      (int x, int y, int inRadius, bool select,
       InkOverlay myInkOverlay, Panel inkPanel)
```

This method draws a circle at the x,y coordinates of the *Panel* passed in. The radius of the circle is specified by *inRadius*. If *select* is true, then the circle will be selected when it is drawn in *inkPanel.*

drawNode

```
public override void drawNode
      (InkOverlay myInkOverlay, Panel inkPanel, Stroke inStroke)
```

This method takes in a skewed version of a circle, *inStroke*, drawn by the user and draws a "perfect" circle in its place. It calls *drawCircle* to draw the circle and set the newly drawn circle as the *nodeStroke* of the *TreeNode*.

getPointID

```
public int getPointID(Point pt)
```

This method returns the ID of the point in the *nodeStroke* of the *TreeNode* object closest to the *Point* passed in. This ID is used by the *Edge* objects associated with the *TreeNode*.

getRadius

```
public int getRadius()
```

This method returns the radius of the *TreeNode* object.

isInsideNode

```
public bool isInsideNode(Point startStroke, Point endStroke)
```

This method determines if a *Stroke* is entirely contained within a *TreeNode* object's *nodeStroke*. The points passed in are the two endpoints of the *Stroke*.

isLeftEdge

```
public bool isLeftEdge(Point startPt)
```

This method determines, given the start point of an *Edge*, if the *Edge* is a left edge of the *TreeNode* object. The method returns *true* if startPt.X is less than the x-value of the center of the node.

setLabel

```
public void setLabel(InkOverlay myInkOverlay)
```

This method sets the *Stroke* object(s) inside the *nodeStroke* of the *TreeNode* as the *label*.

The *AVLnode* Class

An *AVLnode* object represents a node in an AVL tree. This class is a subclass of the *TreeNode* class. It also provides the following instance variable:

```
private int height;
```

The following methods are included in the *AVLnode* class:

getHeight

```
public int getHeight()
```

This method returns the height of the *AVLnode*.

setHeight

```
public void setHeight(int x)
```

This method sets the height of the *AVLnode* to x.

The *RedBlackNode* Class

A *RedBlackNode* object represents a node in an red-black tree. This class is a subclass of the *TreeNode* class. It also provides the following instance variable:

```
private Color color;
```

The following methods are included in the *RedBlackNode* class:

getColor

```
public Color getColor()
```

This method returns the *color* of the *RedBlackNode*.

setColor

```
public void setColor(Color c)
```

This method sets the *color* of the *RedBlackNode* to *c*.

The *Edge* Class

An *Edge* object represents an edge between two nodes in a binary tree. An object of this class has the following attributes:

```
private Stroke   edgeStroke;
private TreeNode parentNode,
                 childNode;
private int      startPtID,
                 endPtID;
```

The following methods are included in the *Edge* class:

getChild

```
public TreeNode getChild()
```

This method returns the child *TreeNode* of the *Edge*.

setChild

```
public void setChild(TreeNode t)
```

This method sets the child of the *Edge* to the *TreeNode* t.

getEdge

```
public Stroke getEdge()
```

This method returns the *edgeStroke* of the *Edge* object.

setEdgeStroke

```
public void setEdgeStroke(Stroke stroke)
```

This method sets the *edgeStroke* of the *Edge* object to *Stroke* passed in.

getParent

```
public TreeNode getParent()
```

This method returns the parent *TreeNode* of the *Edge*.

setParent

```
public void setParent(TreeNode t)
```

This method sets the parent of the *Edge* to the *TreeNode* t.

getEndPointId

```
public int getEndPointId()
```

This method returns the *endPtID* of the *Edge* object.

setEndPointId

```
public void setEndPointID(int id)
```

This method sets the *endPtID* of the *Edge* object to the *int* passed in.

getStartPointId

```
public int getStartPointId()
```

This method returns the *startPtID* of the *Edge* object.

setStartPointId

```
public void setStartPointID(int id)
```

This method sets the *startPtID* of the *Edge* object to the *int* passed in.

The *Tree* Class

The *Tree* class is an abstract class which is used to model any type of binary tree. The class provides the following instance variables:

```
public ArrayList tree;
private string strOrder;
private int animationSpeed;
```

Note that the last two variables are used in the printing of values after traversals and in determining the speed of animation, respectively. The following methods are included in the *Tree* class:

AddNode

```
public void AddNode(TreeNode n)
```

This method adds a *TreeNode* object to the *tree* array list.

Clear

```
public void Clear()
```

This method clears all *TreeNode* objects from the *tree* array list.

Delay

```
public void Delay(int n)
```

This method contains a triply-nested loop which simply wastes time. The method is used to highlight nodes in traversals, insertions, and deletions. The larger the value of *n*, the longer the delay will be.

drawNewLeftChild

```
public void drawNewLeftChild
      (TreeNode t, TreeNode newNode, InkOverlay myInkOverlay,
      Panel inkPanel)
```

This method draws a new left child of *TreeNode t* in the *Panel* passed in. The *TreeNode, newNode,* will become the left child of *t.*

drawNewRightChild

```
public void drawNewRightChild
      (TreeNode t, TreeNode newNode, InkOverlay myInkOverlay,
      Panel inkPanel)
```

This method draws a new right child of *TreeNode t* in the *Panel* passed in. The *TreeNode, newNode,* will become the right child of *t.*

find

```
public void find
      (InkOverlay myInkOverlay, Strokes strokes, TreeNode node,
      Panel inkPanel);
```

This method finds a value in a binary tree. The nodes are highlighted as they are visited. If the value is in the tree, a “found” message is printed to the screen. If the value is not in the tree, a “not found” message appears.

findMin

```
public TreeNode findMin(TreeNode t)
```

This method finds the minimum value in the subtree rooted at *TreeNode t*. The *TreeNode* object which contains this value is returned.

findRoot

```
public TreeNode findRoot()
```

This method finds and returns the root of *tree*.

findTreeNode

```
public TreeNode findTreeNode(Stroke stroke)
```

Given a *Stroke* object, *stroke,* this method returns a *TreeNode* object if *stroke* is the *nodeStroke* of any *TreeNode* in the tree. It returns *null* if *stroke* is not equal to any *nodeStroke* in the tree.

getCenter

```
public Point getCenter(Stroke stroke)
```

This method returns the center point of a circle *Stroke* passed in. It is useful when moving a *label* inside a node.

getNumNodes

```
public int getNumNodes()
```

This method returns the number of *TreeNodes* in *tree.*

getSubtree

```
public void getSubtree(TreeNode t, Strokes strokes)
```

This method collects all of the *Stroke* objects associated with the subtree rooted at t into the collection *strokes*.

highlight

```
public void highLight(TreeNode t, Panel inkPanel, Color color)
```

This method highlights the *nodeStroke* of *t* the color that is passed in. The *nodeStroke* will change from its original color to the color passed in, delay, and then change back to its original color.

inOrder

```
private void inOrder
      (TreeNode t, InkOverlay myInkOverlay, Panel inkPanel)
```

This method performs the inorder traversal of *tree* by highlighting the *TreeNode* objects in the order they are visited.

insert

```
public abstract void insert
      (InkOverlay myInkOverlay, Strokes strokes, TreeNode node,
      Panel inkPanel);
```

This is an abstract method that each of the *Tree* class's subclasses must implement in order to insert a value into *tree.* Each of the *insert* methods will be unique to each of the subclasses.

moveStrokes

```
public void moveStrokes
      (Strokes strokes, TreeNode t, InkOverlay myInkOverlay)
```

This method moves the *Strokes* collection passed in inside the *nodeStroke* of *TreeNode t.*

postOrder

```
private void postOrder
      (TreeNode t, InkOverlay myInkOverlay, Panel inkPanel)
```

This method performs the postorder traversal of *tree* by highlighting the *TreeNode* objects in the order they are visited.

preOrder

```
private void preorder
      (TreeNode t, InkOverlay myInkOverlay, Panel inkPanel)
```

This method performs the pre-order traversal of *tree* by highlighting the *TreeNode* objects in the order they are visited.

reversePostOrder

```
private void reversePostOrder
      (TreeNode t, InkOverlay myInkOverlay, Panel inkPanel)
```

This method performs the reverse postorder traversal of *tree* by highlighting the *TreeNode* objects in the order they are visited.

RemoveNode

```
public void RemoveNode(TreeNode n)
```

This method removes *TreeNode n* from *tree.*

setAnimationSpeed

```
public void setAnimationSpeed(int n)
```

This method sets the instance variable *animationSpeed* to the value passed in. The *animationSpeed* variable is used as the parameter to the *Delay* method when it is called in the *highlight* method.

setLabels

```
public void setLabels(InkOverlay myInkOverlay)
```

This method is a wrapper that calls the *setLabel* method for each *TreeNode* object in *tree.*

Traverse

```
public void Traverse
      (InkOverlay myInkOverlay, Panel inkPanel, string traversal)
```

This method calls the appropriate traversal method (*inOrder, preOrder, postOrder, reversePostOrder*) given the string *traversal* passed in. It also prints the values of the *TreeNode* objects in the upper left corner of *inkPanel* in the order they are visited.

The *BSTree* Class

The *BSTree* class is a subclass of the *Tree* class that represents a binary search tree. A *BSTree* object contains zero or more *TreeNode* objects. The following methods are included in the *BSTree* class:

insert

```
public override void insert
      (InkOverlay myInkOverlay, Strokes strokes, TreeNode t,
      Panel inkPanel)
```

This method inserts an integer value into a *BSTree* object. The *Strokes* collection, *strokes,* represents the integer to be inserted that the user has drawn. The *TreeNode, t*, is the root node of the tree where the value will be inserted. As the *TreeNode* objects are visited in the insertion, they are highlighted. The speed at which they are highlighted can be changed by adjusting the *animationSpeed* variable.

remove

```
public TreeNode remove
      (InkOverlay myInkOverlay, Strokes strokes, TreeNode t,
      Panel inkPanel)
```

This method deletes an integer value from a *BSTree* object. The *Strokes* collection, *strokes,* represents the integer to be removed that the user has drawn. The *TreeNode, t*, is the root node of the tree where the value will be removed. As the *TreeNode* objects are visited in the removal process, they are highlighted. The

speed at which they are highlighted can be changed by adjusting the *animationSpeed* variable.

The *AVLTree* Class

The *AVLTree* class is also a subclass of the *Tree* class. An *AVLTree* object represents an AVL tree and contains zero or more *AVLnode* objects. The following methods are included in the *AVLTree* class:

doubleWithLeftChild

```
private void doubleWithLeftChild
      (AVLnode n, InkOverlay myInkOverlay, Panel inkPanel)
```

This method performs a double rotation of *AVLnode, n*, with its left child. This method is called when, given a node *n*, an insertion into the right subtree of the left child of *n* causes *n* to become unbalanced.

doubleWithRightChild

```
private void doubleWithRightChild
      (AVLnode n, InkOverlay myInkOverlay, Panel inkPanel)
```

This method performs a double rotation of *AVLnode, n*, with its right child. This method is called when, given a node *n*, an insertion into the left subtree of the right child of *n* causes *n* to become unbalanced.

drawHeight

```
public void drawHeight(InkOverlay myInkOverlay, Panel inkPanel)
```

This method displays the *height* of an *AVLnode* just outside its *nodeStroke*. This is useful in illustrating which node is out of balance on an insert.

height

```
public int height(AVLnode t)
```

This method returns the *height* of an *AVLnode* or –1 if the *AVLnode* is *null*.

insert

```
public override void insert
      (InkOverlay myInkOverlay, Strokes strokes, TreeNode node,
      Panel inkPanel)
```

This method inserts an integer value into a *AVLTree* object. The *Strokes* collection, *strokes,* represents the integer to be inserted that the user has drawn.

The *TreeNode, node*, is the root node of the tree where the value will be inserted. As the *AVLnode* objects are visited in the insertion, they are highlighted. The speed at which they are highlighted can be changed by adjusting the *animationSpeed* variable. If the tree becomes unbalanced on an insertion, the appropriate rotation method is called, and the tree rebalances itself.

rotateWithLeftChild

```
public TreeNode rotateWithLeftChild
      (AVLnode n, InkOverlay myInkOverlay, Panel inkPanel)
```

This method performs a single rotation of *AVLnode, n*, with its left child. This method is called when, given a node *n*, an insertion into the left subtree of the left child of *n* causes *n* to become unbalanced.

rotateWithRightChild

```
public TreeNode rotateWithRightChild
      (AVLnode n, InkOverlay myInkOverlay, Panel inkPanel)
```

This method performs a single rotation of *AVLnode, n*, with its right child. This method is called when, given a node *n*, an insertion into the right subtree of the right child of *n* causes *n* to become unbalanced.

The *RedBlackTree* Class

The *RedBlackTree* class is also a subclass of the *Tree* class. An *RedBlackTree* object represents an red-black tree and contains zero or more *RedBlackTree* objects. The following methods are included in the *RedBlackTree* class:

insert

```
public override void insert
      (InkOverlay myInkOverlay, Strokes strokes, TreeNode node,
      Panel inkPanel)
```

This method inserts an integer value into a *RedBlackTree* object. The *Strokes* collection, *strokes,* represents the integer to be inserted that the user has drawn. The *TreeNode, node*, is the root node of the tree where the value will be inserted. As the *RedBlackTree* objects are visited in the insertion, they are highlighted. The speed at which they are highlighted can be changed by adjusting the *animationSpeed* variable. If the tree becomes unbalanced on an insertion, the appropriate rotation method is called, and the tree rebalances itself.

isLeftChild

```
private bool isLeftChild(RedBlackNode n)
```

This method returns *true* if *n* is a left child of its parent or *false* otherwise.

rotateWithLeftChild

```
public TreeNode rotateWithLeftChild
      (RedBlackode n, InkOverlay myInkOverlay, Panel inkPanel)
```

This method performs a single rotation of *RedBlackNode, n*, with its left child.

rotateWithRightChild

```
public TreeNode rotateWithRightChild
      (RedBlackode n, InkOverlay myInkOverlay, Panel inkPanel)
```

This method performs a single rotation of *RedBlackNode, n*, with its right child.

uncle

```
public RedBlackNode uncle(RedBlackNode n)
```

This method returns the uncle of *n* or *null* if the uncle does not exist.