Design Patterns for Self-Balancing Trees

Dung ("Zung") Nguyen and Stephen B. Wong
Dept. of Computer Science
Rice University
Houston, TX 77005
dxnguyen@rice.edu, swong@rice.edu

1 Introduction

Lists and trees are standard topics in a computer science curriculum. In many applications, they are used to implement containers whose main behaviors consist of storage, retrieval and removal of data objects. Various forms of self-balancing trees (SBTs) such as B-trees guarantee a O(logN) efficiency for these computations. Current textbooks on this subject (see for example [2]) discuss them in terms of complicated, low-level pseudo-code. The abstract nature of the data structures and the algorithms that manipulate them is lost in a sea of details. The problem lies in the lack of delineation between the intrinsic structural operations of a tree and the extrinsic, order-driven calculations needed to maintain its balance.

However, the current tree framework proves to be inadequate to model self-balancing trees due to the inherent limitation of the visitor design pattern with regards to dynamically changing numbers of hosts. In this paper, we present enhancements to the current framework that overcomes the original limitations and produces an object-oriented SBT implementation that closely matches the abstract view of the structure. Our paper serves a second purpose of exemplifying how good OO design enables one to re-focus on the fundamental nature of the problem and create solutions that are both simple and powerful.

Section 2 specifies and implements a minimal and complete set of behaviors that are intrinsic to the tree structure. Each node in the tree can hold an arbitrary number of data elements. The size of the root node is used to represent the current state of the tree. We design such a tree as a composite structure, which behaves as a finite state machine whose number of states can vary dynamically at run-time.

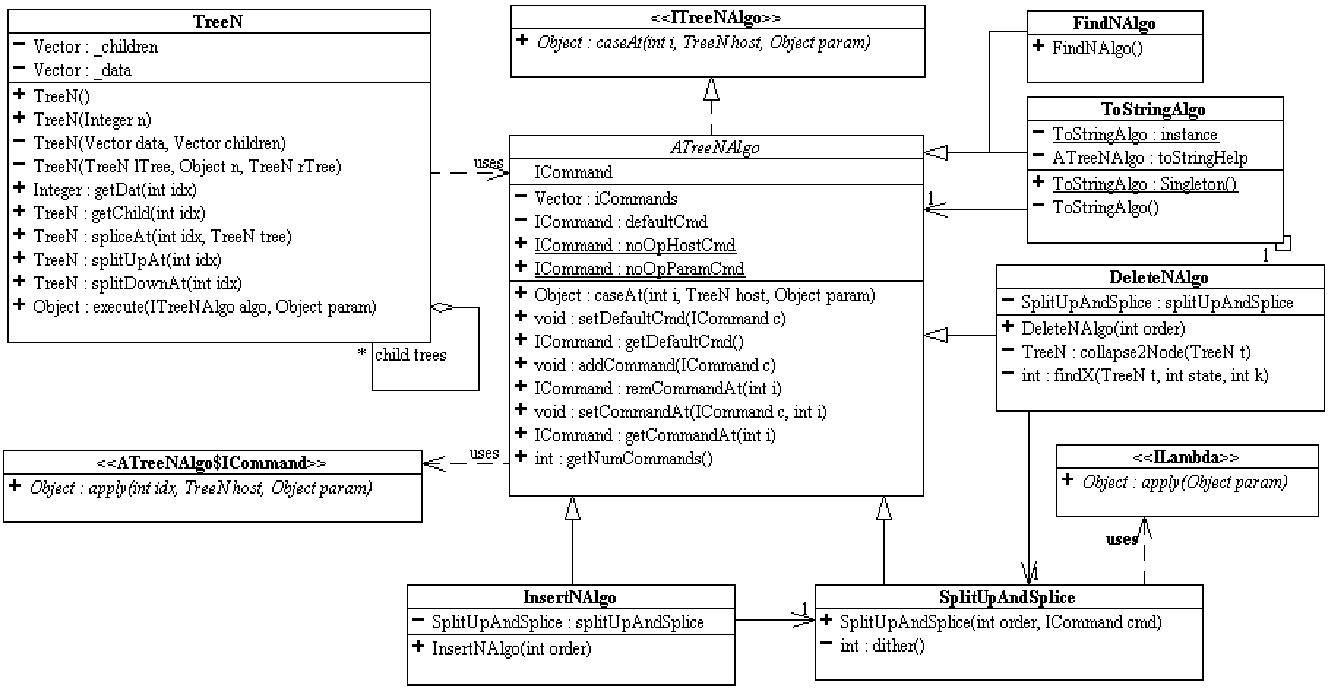

Figure 1: UML class diagram for the tree and algorithms as visitors.

Section 3 describes how we generalize the visitor pattern to decouple the extrinsic algorithms that operate on a tree from its intrinsic structural behaviors. In our formulation, the extrinsic algorithms act as visitors to the host tree. They are capable of not only reconfiguring the tree states and their transitions at run-time but also reconfiguring themselves to possess an appropriate number of visiting methods to match the tree states. The tree structure and its visitors thus form a framework with dynamically re-configurable components.

Section 4 defines the notion of a height-balanced tree and discusses the constraints on any operations that can modify the tree structure while maintaining its balance. In order to implement insertion and deletion in SBTs, it is essential to be able to move data vertically in the tree without changing the tree's height.

Section 5 describes our SBT insertion algorithm and its Java implementation. The algorithm's proof-of-correctness and complexity analysis will be shown to be straightforward, simple and intuitive.

Section 6 describes our SBT deletion algorithm and its Java implementation. As with the insertion algorithm, the deletion algorithm's proof-of-correctness and complexity analysis is straightforward, simple and intuitive.

2 The Tree Structure

We consider trees, called TreeN, that can hold multiple data elements in each node and where each node can have multiple child trees. Without loss of generality, we limit the data elements to be of Integer type. A TreeN can be either empty or non-empty. A non-empty TreeN holds an arbitrary, non-zero, number of data elements, n, and n+1 TreeN objects called "child trees". An empty TreeN holds no data and has no child trees. This recursive definition for the tree is well represented by the composite design pattern [1]. Since the operations on a tree often depend on the number of data elements in the nodes, we can model the tree as having different "states". The state of the tree is defined by the number of data elements in the root node of the tree. We can thus label each state with an integer value. For instance, an empty tree has state = 0, while a tree with one data element and two child trees (commonly referred to as a "2-node tree") is in state = 1. Operations on the tree may cause the tree to transition from one state to another as data elements and associated child trees are added or removed. The tree thus behaves as a finite state machine.

Figure 1 depicts the UML class diagram of TreeN together with algorithms that act as visitors. The visiting algorithms will be discussed in Section 3.

Code	Comment

`public class TreeN {`
`  private Vector _children = new Vector();`
`  private Vector _data = new Vector();`
	The data elements and child trees are held in Vectors. The state is the size of the _data vector. The _children.size() = _data.size()+1.

`  public TreeN() { }` `  public TreeN(Integer n) {  this(new TreeN(), n, new TreeN()); }`	Empty and 2-node tree constructors.

`  private TreeN(Vector data, Vector children) {`
`  _data = data; _children = children;`
`  }`
`  private TreeN(TreeN lTree, Object n, TreeN rTree) {`
`  _data.add(n); _children.add(lTree); _children.add(rTree);`
`  }`
	Private constructors for internal use.

`  public Integer getDat(int idx) { return (Integer) _data.get(idx); }` `  public TreeN getChild(int idx) { return (TreeN) _children.get(idx); }`	Parameterized accessor methods for the root data and child trees.

`  public TreeN spliceAt(int idx, TreeN tree) {`
`  int i=tree._data.size();`
`  if (i>0) {`
`  if (_data.size() > 0)  _children.set(idx, tree.getChild(i--));`
`  else  _children.add(idx,tree.getChild(i--));`
`  for (; i>=0;i--) {`
`  _data.add(idx,tree.getDat(i));`
`  _children.add(idx, tree.getChild(i));`
`  }`
`  }`
`  return this;`
`  }`
	After checking for empty trees, the data and child trees are copied from the source tree into this tree at the specified index.

`  public TreeN splitUpAt(int idx) {`
`  if (_data.size()>1) {`
`  TreeN lTree, rTree;`
`  Vector newData = new Vector(), newChildren = new Vector();`
`  Object rootDat = _data.remove(idx);`
`  for (int i = 0;i<idx;i++) {`
`  newData.add(_data.remove(0));`
`  newChildren.add(_children.remove(0));`
`  }`
`  newChildren.add(_children.remove(0));`
`  if (newData.size()>0)  lTree = new TreeN(newData,newChildren);`
`  else  lTree = (TreeN)  newChildren.firstElement();`
`  if (_data.size()>0)  rTree = new TreeN(_data, _children);`
`  else  rTree = (TreeN) _children.firstElement();`
`  (_data = new Vector()).add(rootDat);`
`  (_children = new Vector()).add(lTree);`
`  _children.add(rTree);`
`  }`
`  return this;`
`  }`
	After checking for the empty tree case, the new root data, as referenced by the supplied idx, is removed. The left side data and child trees are moved to new vectors and a new left child tree is constructed. Note the special case if there is no left side data. The original remaining data and child trees become the new right child tree, noting the special case of no data again. The removed root data becomes the new data for this tree and the new left and right child trees are added.

`  public TreeN splitDownAt(int idx) {`
`  if (_data.size()>1){`
`  TreeN newChild = new TreeN(getChild(idx),_data.remove(idx),getChild(idx+1));`
`  _children.remove(idx);`
`  _children.set(idx, newChild);`
`  }`
`  else  {`
`  _data.clear();`
`  _children.clear();`
`  }`
`  return this;`
`  }`
	A new 2-node tree is made with the removed data at idx and its original left and right child trees. The original left child reference is removed and the original right child reference is replaced with a reference to the new 2-node tree. However, if this tree is a 2-node to start, then the tree is simply cleared back to an empty tree.

`  public Object execute(ITreeNAlgo algo, Object param) {`
`  return algo.caseAt(_data.size(), this, param);`
`  }`
`}`
	Host hook method for visitors. The call is delegated to the visitor's case corresponding to this tree's state.

Listing 1: TreeN implementation

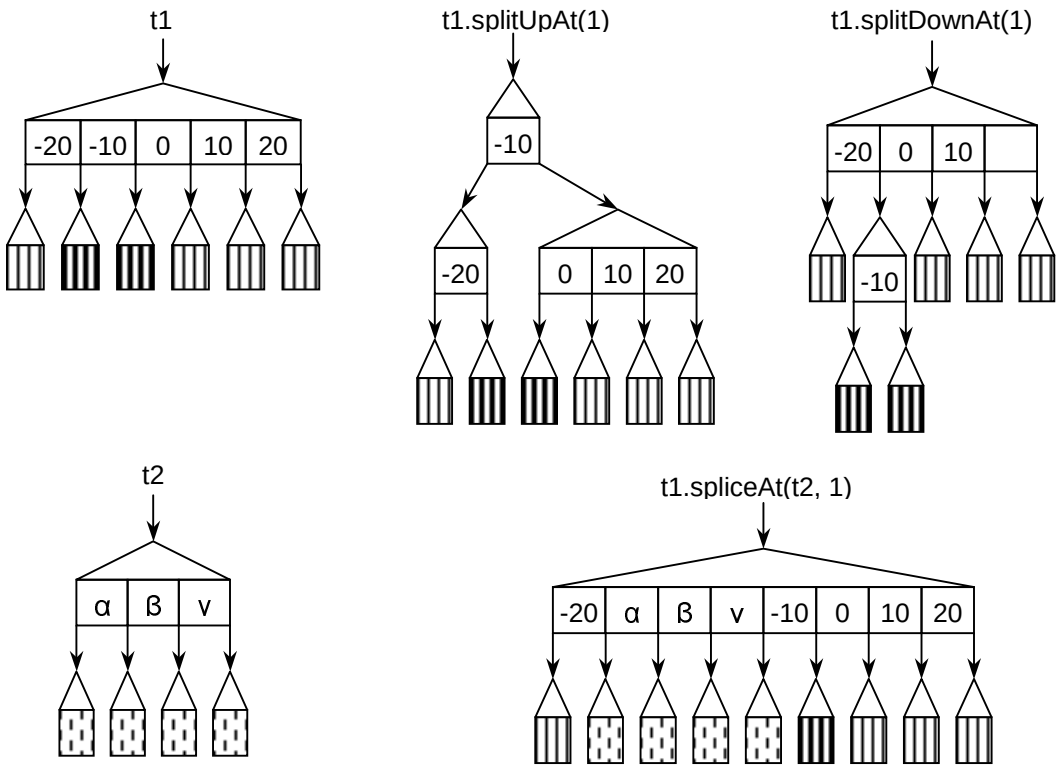

Figure 2: Intrinsic structural operations on the tree.

The methods of the tree should represent the intrinsic behaviors of the tree. For maximal decoupling and flexibility, the methods should form a complete and minimal set of operations from which all other possible operations on the tree can be constructed. The intrinsic structural behaviors of the tree serve exactly two purposes:

- To provide access to the tree's data and structural subcomponents, and
- To perform constructive and destructive modifications of the tree's internal structure, thus enabling the tree to transition from one state to another.

The intrinsic structural behaviors of the tree are *invariant*, that is they remain fixed in all applications, and enable us to build trees that can hold an arbitrary number of data elements per node.

A problem that one encounters immediately upon attempting to find the complete and minimal set of operations on a tree is that its constructive and destructive processes are not well defined. For instance, inserting a data element into the root node of a tree is not well defined because of the inability to unambiguously insert the required additional child tree. Such an insertion can only be clearly defined on leaf trees. The source of the problem is that such operations suffer from a mixing of data manipulations and structural modifications. To identify the intrinsic operations of the tree, one must separate the operations that manipulate data elements from those that modify the tree's structure. Structural modification should involve trees as atomic units and have well defined behavior for any tree. Data operations are relegated to the construction of new trees and to simple gettor methods. The intrinsic behaviors of a tree can thus be classified as constructors, structural modifiers and gettors. Listing 1 shows the Java implementation of TreeN and Figure 2 illustrates the intrinsic structural operations of the tree.

The purpose of a constructor is to initialize the instantiated object to a well-defined state. Since there are two clearly distinct states of a tree, empty and non-empty, each has an associated constructor. The empty tree constructor, TreeN(), creates a empty (state = 0) tree. The non-empty constructor, TreeN(Integer n), takes a single data element and constructs a 2-node (state = 1) leaf tree. This can be viewed as providing the base case and inductive case construction for the system. There is no need for construction of higher states as they can be created through structural modifications of 2-node leaf trees. The set of constructors is thus complete and minimal.

Structural modifiers are methods with side effects that work strictly on trees and not on data. They are also well defined for all trees in all possible states. To span the space of all possible structural modifications, one must fundamentally be able to modify the tree, a 2-dimensional entity, in both its width and height directions. In addition

to constructive processes in the two directions, a destructive process must also be provided. This only implies that the complete and minimal set of structural modifies must consist of three methods, none of which can be constructed from the other two. A full proof that only 3 methods constitute a complete and minimal set is beyond the scope of this paper.

spliceAt(int idx, TreeN tree) joins the supplied source tree to the target tree at index idx: The idx^{th} child of the target tree is deleted and the source tree is inserted between the idx^{th} and $idx+1^{th}$ elements of the target tree. The children of the source tree remain in their respective places with regards to the original elements of the source tree. Splicing an empty source tree into a non-empty tree is a no-operation. Splicing a non-empty source tree into an empty tree will mutate the empty target tree into a shallow copy of the source tree. This operation is mainly constructive in the horizontal direction. However, if the source tree being spliced into the target tree is a child of the target tree, then this method is also destructive in the vertical direction.

splitUpAt(int idx) mutates the tree, in state s, into a 2-node tree (state = 1), where the idx^{th} element becomes the root data and the left child is a state = idx tree with the 0 through idx-1 elements of the original root and the right tree is a state = s - idx-1 tree with the idx+1 through s elements of the original root. Split up on an empty tree is a no-operation. This method is constructive in the vertical direction because the height increases as well as destructive in the horizontal direction because the new child trees have fewer root data elements than the original root.

splitDownAt(int idx) removes the idx^{th} element from the root of the tree including its left and right child trees. The resultant new child tree is a 2-node tree where its root data is the original idx^{th} element and where its left and right children are the original idx^{th} element's left and right children respectively. Splitting down a 2-node tree will result in an empty tree. Splitting down an empty tree is a no-operation. Like splitUpAt(), this method is constructive in the vertical direction and destructive in the horizontal direction.

The standard "settors" that set a child tree to a new tree at index idx, and that set a data element at index idx, can be easily replicated using a combination of the above methods.

getDat(int idx) and getChild(int idx) are the standard "gettors" that provide access to data and child trees without side-effect. The root node's data elements can be accessed via an index idx, where $0 \leq idx <$ state (= node size). The root node's child trees can be accessed similarly but where $0 \leq idx \leq$ state. Since all data elements and child trees can be accessed through these methods and only through these methods, the set of gettors is thus minimal and complete.

execute(*ITreeNAlgo* algo, Object param) is the "accept" method for a host in the visitor design pattern [1]. It provides a "hook" for all algorithms defined externally to the tree to operate properly on the tree without knowing the state of the tree. The abstraction for all such extrinsic operations is encapsulated in an interface called *ITreeNAlgo*, which acts as a visitor to the tree host.

We do not consider operations such as specific insertion and deletion algorithms that maintain the balance of a tree as intrinsic to the tree's behavior. The tree is simply a structure and has no inherent knowledge of the properties of the data it contains or the heights of its child trees. These operations are extrinsic to the tree structure, and as Nguyen and Wong advocated in [3], they must be decoupled from the intrinsic structural behaviors of the tree. The visitor pattern was used to achieve this decoupling.

3 The Visitors

Algorithms on a host tree often depend on its state, the size of its root node. The *ITreeNAlgo* visitor interface (See Figure 1) thus must provide a specific method for each of the host states. Since any tree node can hold an arbitrary number of data elements, an arbitrary number of visiting methods must be defined. That is, the visitor must have a varying number of visiting methods to match the host's states. Since standard visitors would match one method per host state, the system is hamstrung by physical limitation that only a fixed number of methods can be defined. This limitation can be overcome by replacing the multiple different methods of the visitor with a single "caseAt" method parameterized by an integer index. The individual hosts are now identified by an integer value, the state number, and they can now parametrically call their respective method in the visitor. Since the host structure provides a complete set of public primitive behaviors, the visiting algorithms are not limited in their capabilities.

The roles of TreeN as a host and of *ITreeNAlgo* as a visitor are summarized in the following.

Hosts (TreeN)

- Characterized by an integer s. For a finite state machine, s represents the state of the host.
- Host provides a complete set of public methods for all tree operations.
- Hook method of host, execute(v, param), guarantees that is will call the s'th case of the visitor (v), *i.e.* return v.caseAt(s, this, param) .

Visitors (implement *ITreeNAlgo*)

- Provides a single caseAt(int index, TreeN host, Object param) method that returns an Object.
- Guarantees that there is a behavior for all values of index. This includes the possibility of throwing an exception or a no-operation.

Code	Comment

`public abstract class ATreeNAlgo implements ITreeNAlgo {`
`  public static interface ICommand {`
`  public Object apply(int idx, TreeN host, Object param);`
`  }`
	The command interface used is a nested interface.

`  private Vector iCommands = new Vector();` `  private ICommand defaultCmd = noOpHostCmd;`	Commands stored in a vector plus one default command.

`  public Object caseAt(int i, TreeN host, Object param) {`
`  if (0<=i && i<iCommands.size()) return ((ICommand)iCommands.get(i)).apply(i, host, param);`
`  else return defaultCmd.apply(i, host, param);`
`  }`
	To call a case is to call the corresponding command or the default if it is not in the vector.

`  public void setDefaultCmd(ICommand c) { defaultCmd = c; }`
`  public ICommand getDefaultCmd() { return defaultCmd; }`
`  public void addCommand(ICommand c) { iCommands.add(c); }`
`  public ICommand remCommandAt(int i) { return (ICommand)iCommands.remove(i); }`
`  public void setCommandAt(ICommand c, int i) { iCommands.set(i, c); }`
`  public ICommand getCommandAt(int i) { return (ICommand) iCommands.get(i); }`
`  public int getNumCommands() { return iCommands.size(); }`
	Command management methods.

`  public static final ICommand noOpHostCmd = new ICommand() {`
`  public Object apply(int idx, TreeN host, Object param) { return host; }`
`  };`
`  public static final ICommand noOpParamCmd = new ICommand() {`
`  public Object apply(int idx, TreeN host, Object param) { return param; }`
`  };`
`}`
	No-operation commands for utility use.

Listing 2: *ATreeNAlgo* visitor code

The effect of the strict contract specified by the last bullet items for the host and visitor above is that conditionals concerning the state of the tree are eliminated. Operations can thus be performed on the tree *without worrying about the state of the tree* as the tree and visitor combination is guaranteed to take the correct action.

ATreeNAlgo is an implementation of *ITreeNAlgo* (see Figure 1 and Listing 2). Our solution while not unique does embody the most salient features of any implementation. Since the number of states for which the visitor must provide behaviors is arbitrary, the visitor must have a way of installing as many behaviors as needed for a particular application.

The command design pattern [1] provides an elegant solution to this problem. First, let us examine the methods of a visitor. The accepting of a visitor by a host has a particular semantic, however, since the host type/state is unknown at run time, that semantic cannot depend on which concrete host is in use. Thus each method that the visitor supplies for each different host must have identical semantics with respect to the visitor. Conversely, since the host has no idea which concrete visitor is being used, *the visitor is without semantics to the host*. Compare this to the semantics of a command object. Since the invoker of the command does not know what concrete command it is using nor what that command does, *the command has no semantic to the invoker*. The command only has a semantic to the client that supplied it. A visitor can thus be considered to be a collection of semantically related commands.

Thus we can replace each method of the visitor with a corresponding command object. The simplest solution here is simply to hold a vector of command objects, *ICommand,* within the visitor *ATreeNAlgo*. The caseAt(i,…) method simply delegates to the i^{th} command in the vector. If the requested command is not found in the vector (*i.e.* i is out of bounds), the call is instead delegated to a specific default command, thus enforcing the visitor's contract to always supply a behavior for every index. For further safety, the default command is initialized to a no-operation command upon the instantiation of the visitor.

To manage the installed commands, *ATreeNAlgo* provides a gettor and a settor for commands at a specific index, add and remove command methods, a gettor and a settor for the default behavior command, and a method to get the number of installed non-default commands. With these management methods, *ATreeNAlgo*'s have the ability to be dynamically reconfigured, including the default case. This reconfiguring can be done by an outside entity or by the visitor itself and can be performed at any time, including during the visitor's execution by the host. Since the visitors determine what transitions take place and to what state the transitions are taken (within the limits of the state-transition methods provided by the host), it is possible for a visitor to change the number of states available to the system as well as change the change the transitions that take place. Thus the visitor can completely reprogram the entire finite state machine at run time. It should be noted however, that since the host executes an *ITreeNAlgo*, not an *ATreeNAlgo*, the host is unaware of any dynamic reconfiguration capabilities of the visitor.

The ability to self-(re)configure means that a visitor's constructor can install whatever and as many cases it needs to solve a particular problem. The parameters needed to specify a particular problem are given to the constructor as input values and the constructor then proceeds to build the visitor based on those parameters. For instance, the SplitUpAndSplice visitor (discussed further in Section 5 and listed in Listing 3) will split up the host tree only if the state of the host is greater than a predefined order. The value of order is given to the visitor's constructor, which installs order+1 no-op commands (supplied) into itself plus a single default command that splits up the host. The result is that the visitor can distinguish all states > order and split up them without any need for conditionals.

As shown by the preceding example, the default case for the *ATreeNAlgo* is more than a safety valve for out-of-bounds case indices. Fundamentally, it provides a behavior for an unbounded set of indices. For instance, it can be used to define behavior for sets of states such as "non-empty trees" or "trees with state > 7". This also gives the visitor the ability to handle changing numbers of states without reconfiguring. For instance, ToStringAlgo in Figure 1 can print any tree of arbitrary complexity by only using one command for the empty tree and one default command that handles the non-empty tree.

While *ATreeNAlgo* is simple and versatile solution, one can easily imagine other possible implementations, particularly ones that are not subject to the vector's limitations. For instance, a hash table of commands would allow for arbitrary case definitions or a scheme of "breakpoints" to classify indices would enable single commands can handle a range of indices.

4 Self-Balancing Trees

Fundamentally, a tree is not constrained to any sort of ordering or balancing—this is the bailiwick of particular insertion/deletion algorithms. That is, the ordering and balance of a tree is an extrinsic, variant behavior. A SBT is one whose insertion and deletion algorithms maintain the tree's height balance. SBTs are usually considered for trees whose elements are totally ordered. We will thus consider trees whose root data elements, x_i are in strict ascending order Also all data elements in the i^{th} child tree are less than x_i and all elements in the $i+1^{th}$ child tree are greater than x_i. The need for non-trivial balancing only arises when there is an imposed maximum on the number of data elements per node. We call this maximum number the "order" of the tree, and we will consider only trees with order > 1. For example, the well-known "2-3-4 tree" is of order 3.

However, before one can create an algorithm to create or maintain a balanced tree, one must first study what key issues affect the tree's balance. Here is a recursive definition of a balanced tree:

- An empty tree is balanced.
- A non-empty tree is balanced if and only if all its child trees are balanced and all have the same height.

From the definition of a balanced tree, we can see that in order to preserve its balance, any operation on the tree must affect the depth of all the children equally. The only place in the tree that an operation could have this effect is at the root. The only operations on a tree that can do this are a splitting up the root node and splicing all the children into the root ("collapsing"). It turns out that the latter operation can be restricted to 2-node roots without loss of generality.

On the other hand, insertion of a data element must take place at a leaf node because only by traversing the tree all the way down to a leaf can one make the determination that the data does not already exist in the tree.

Deletion of a data element is well defined only when the tree is a leaf. Any other situation leads to ambiguous choices on the disposal of one or more of the child trees. Thus, once again, the deletion of a data element must take place at the leaf level of a balanced tree.

Therefore, what we see here is that changes to the height of the tree must take place at the root level, but insertion and deletion must take place at the leaf level. Insertion may cause "excess" data, that is, the number of data elements exceeds the order of the tree. We call this situation a "*virtual state*" because it is disallowed by our BST restrictions, but is still a valid operational state of the tree. This excess data must therefore be moved to the top of the tree where height changes won't affect the overall tree balance. The transportation of the data to the root must not effect any height increase in the child tree, as this would lead to a net imbalance of the whole tree.

Since the data element to be deleted must be ultimately located at the leaf level, it must be moved from its original location down to the leaf level for deletion. There is a non-zero possibility however, that the data to be deleted is located in a 2-node, and thus to guarantee no net change in height of the child tree, one must move a candidate data element down from the root node. When the data to be deleted is encountered, it will become the candidate element and continue to be pushed down to the leaves. After deletion, there may exist excess data due to the downward transportation process. Thus, just as the insertion case, this excess data must be transported upwards to the root.

We thus see that vertical transportation of data in a tree, without disturbing the height, is a key operation in insertion and deletion into/from balanced trees. This process must

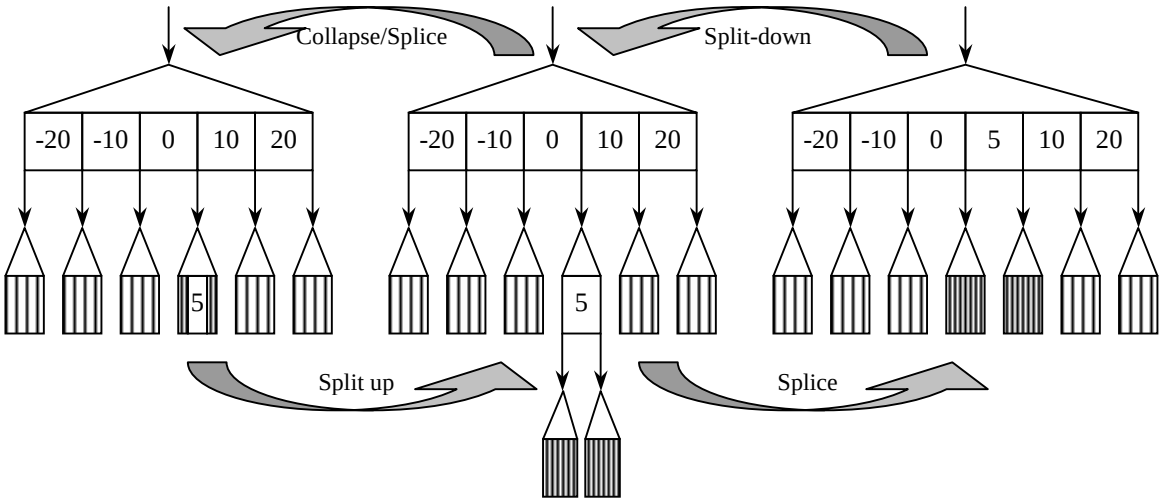

Figure 3: Height-preserving vertical data movement in the tree.

be understood before an insertion or deletion algorithm can be written. It makes sense that the tree should intrinsically support the structural operations needed to perform this type of data movement. We will now examine the operations needed to move a single data element up or down one level without a net change in tree height (see Figure 3).

Consider first moving a data element up one level. In this case, we wish to move the element "5", which is located in the set of data elements in the root of a child tree. This child tree is the parent's #3 index child tree (See the left-side of Figure 3). The desired destination for the "5" is in the #3 data element location in the parent tree. Because of the strict ordering of the data elements and children trees, the starting child index and ending element index will always be the same. This is a very important programmatic point to remember.

First we must isolate the "5" element. We do so by "splitting up" the child tree (as opposed to "splitting down", which we will discuss later) at the "5" element. This involves taking all the elements to the left of the "5" and creating a separate tree with those elements and then attaching that new tree as the left child of the "5". The same is done with the data elements to the right of the "5" and attaching them to the right. The "5" thus becomes the only root data element in a state = 1 node (2-node state). The reconfigured child tree is now spliced into the parent, and the "5"'s children trees now return to being direct children of the parent. There has been no net depth increase. Note also that if the "5" was chosen because it was the center root data element of the child tree, the splitting up results in a tree that is optimally balanced.

Moving data downwards is very similar to the upward movement process. Starting from the right side of Figure 3, we isolate the "5" by "splitting down" the parent. The means that a new 2-node child tree is formed with the "5" as the sole root node data, and the children trees to the left and the right of the "5" remain its children. The "5" is then "collapsed" with its two children, which is simply to splice the 2-node with the "5", first with its right child tree and then with its left child tree. Once again, there is no net depth change.

The root node offers a couple of special cases. When data is moving upwards and the root node needs to split up, perhaps due to exceeding the order of the tree, the split up operation is not paired with a splice operation because the root has no parent. Thus, the root node ends up as a 2-node and the total depth of the tree has increased by one. When data is moving downwards and the root node is a 2-node, the collapse operation is unpaired with a split down operation. The total depth of the tree thus decreases by one. Because changes at the root node affect the depth of all the children equally, there is no change in the tree's depth balance by operations at the root.

We can now focus on the essence of insertion and deletion into balanced trees:

An insertion algorithm must
1. Search for the leaf location to insert the key.
2. Propagate any "excess" data upwards to the root *without any net height change of the child trees.*
3. Restrict any net height increase to the root, which will happen naturally because its split up will be unpaired with a splice.

A deletion algorithm must
1. Push a deletion candidate downwards to a leaf position *without any net height change in the child trees*, replacing the candidate with the actual data when it is encountered.

Code	Comment
`public class SplitUpAndSplice extends ATreeNAlgo {` `  public SplitUpAndSplice (int order) {`	Constructor takes the order of the tree and a function to execute for states<order.
`    for (int i = 0; i <= order; i++) addCommand(noOpHostcmd);`	Load the supplied command in for all cases ≤ order.

`    setDefaultCmd(new ICommand() {`
`      public Object apply(int idx, TreeN host, Object lambda) {`
`        host.splitUpAt((idx - dither()) / 2);`
`        return ((ILambda)lambda).apply(host);`
`      }`
`    });`
`  }`
	Load the default (state>order) case command. In this case, split up the host at its midpoint and then run the supplied function with the host as a parameter.

`  private final int dither() {`
`    return (int)(2 * Math.random());`
`  }`
`}`
	Randomizes choice of two midpoints when order+1 is even. Helps distribute data more evenly.

Listing 3: SplitUpAndSplice algorithm code

2. Propagate any "excess" data upwards to the root *without any net height change of the child trees.*
3. Restrict a net height change to the root, which will happen naturally because its collapse is unpaired with a split down and its split up is unpaired with a splice.

The deletion algorithm is essentially the same as the insertion algorithm except that it transports data from the root to the leaves as well as from the leaves to the root.

Since both insertion and deletion both propagate excess data upwards to the root, we encapsulate that behavior in a visitor algorithm called SplitUpAndSplice (see Listing 3). SplitUpAndSplice splits up virtual states and uses a supplied command to splice the resultant 2-node tree into the host's parent. To handle arbitrary tree orders that change dynamically, the algorithm self-configures upon construction. SplitUpAndSplice's constructor loads supplied case commands (no-ops generally) into the cases for states ≤ order and sets the default case (states > order) is to split up the tree and then apply the splice command. The splice command is will do the splicing of the host to its parent.

5 Self-Balancing Tree Insertion Algorithm

We will now discuss the insertion algorithm for SBTs. Listing 4 shows the complete Java implementation.

To insert a key into a host tree, we first express a "helper algorithm" that will insert a key into a host tree and re-establish the tree order and height balance using a supplied command.

Helper algorithm: Insert key into host tree using a splice command to maintain height balance and tree order. The supplied splice command has access to the host's parent.

- **Empty host case:** Insert the key here by
  1. Instantiating a 2-node tree with the key
  2. Executing the splice command to splice the new tree into the host's parent.
- **Non-empty host case:**
  1. Search for the child tree whose key range includes the supplied key.
  2. Recursively insert the key into this child tree, passing a command that can properly splice the child to the host.
  3. Execute SplitUpAndSplice on the host passing it the splice command to splice the resultant 2-node host into its parent at the host's location in the parent.

The insertion algorithm simply sets up the above helper algorithm and passes to it a no-op splice command since the root of the tree has no parent.

Insert key into host tree:

- **Empty host case:** Insert key into host by
  1. Instantiating a 2-node tree.
  2. Splicing the new tree into the host.
- **Non-empty host case:**
  1. Delegate to helper algorithm with a no-op splice command.

As shown in Listing 4, the helper algorithm and the splice commands are created on the fly as an anonymous inner objects. The use of anonymous inner classes greatly simplifies the code, minimizing parameter passing and control structures. The anonymous inner classes are effectively lambda functions and thus enable us to harness the power of functional programming in an OO paradigm. For instance, the helper algorithm's closure includes the main algorithm, thus it can directly access the key parameter, even though it may have recurred many levels away from the root. The splice commands, which are really just simple lambda functions are created by the parent tree and thus retain the value of the child index in their closure. So when the child tree, via the SplitUpAndSplice command, attempts to perform the

Code	Comment

`public class InsertNAlgo extends ATreeNAlgo {`
`  private SplitUpAndSplice  splitUpAndSplice;`
 	
`  public InsertNAlgo (int order) {`
`    splitUpAndSplice = new SplitUpAndSplice(order, noOpHostCmd);`
	The constructor initializes the SplitUpAndSplice algorithm to split up only those trees whose state > order.The key to insert is the parameter of the host's execute() method.

`    addCommand(new ICommand() {`
`      public Object apply(int idx, TreeN host, Object n) {`
`        return // student to write code here`
`      }`
`    });`
	The empty case command is loaded. The empty case simply splices a new tree into the host to mutate it into a 2-node tree.

`    setDefaultCmd(new ICommand() {`
`      public Object apply(int idx, final TreeN host, final Object n) {`
`        host.execute(new ATreeNAlgo() {`
`          {`
`            final ATreeNAlgo helper = this;`
	The default (non-empty) case is loaded. The initial call simply sets up a non-recursive call to a helper algorithm, passing a no-op splice command.
	The last line is needed to be able to recur on the anonymous inner object.

`            addCommand(new ICommand() {`
`              public Object apply(int idx, TreeN h, Object cmd) {`
`                // student to write code here`
`              }`
`            });`
	The helper's empty case command is loaded. The host definitely has a parent here. The empty case means that we are at a leaf and that the key was not found. Thus, a new tree is instantiated and spliced into the parent.

`            setDefaultCmd(new ICommand() {` `              public Object apply(int idx, final TreeN h, final Object cmd) {`	The helper's default (non-empty) case command is loaded.

`                final int[] x={0};`
`                for(; x[0] < idx; x[0]++) {`
`                  int d = h.getDat(x[0]).intValue();`
`                  if (d >= ((Integer)n).intValue()) {`
`                    if (d == ((Integer)n).intValue())  return h;`
`                    else break;`
`                  }`
`                }`
	Linear search for the index of the child tree that will hold the key (the insertion point). Could use a binary search.

`                h.getChild(x[0]).execute(helper, new ILambda() {`
`                  public Object apply(Object child) {`
`                    // student to write code here`
`                  }`
`                });`
	Recur, passing it the command (*ILambda*) to splice at the computed insertion point. The splice command's closure is effectively a memento that holds the previous insertion point.

`                return // student to write code here`
`              }`
`            });`
`          }`
	Split this host if necessary and splice the excess data into the parent using the splice command supplied by this host's parent.

`        }, new ILambda() {`
`            public Object apply(Object child){`
`              return host;`
`            }`
`        });`
	The no-op splice command passed to the first call of the helper on the root node.

`        return host;`
`      }`
`    });`
`  }`
`}`
	Return the host to allow chaining.

Listing 4: Self-balancing tree insertion algorithm

splice using the parent supplied splice command, the child index can be directly accessed without need to recalculate it. The splice command also has direct access to a reference to the parent tree in its closure, so the target of the splice does not need to be passed as a parameter. One way of understanding the role of the inner classes is to compare them to the memento design pattern, which is used to store state information for later use [1].

As is often true, good OO design leads to declarative code that follows the proof of correctness exactly. We can see this effect in the proof that the insertion point can be found:

1. An empty tree does not contain the key.
2. If a non-empty tree does not contain the key, then it is obvious that there exists an index `x` such if the `x-1`'th data element exists, it is strictly less than the key and if the `x`'th data element exists, it is strictly greater than the key. If `x` does not exist, then the key is in the tree already.
3. If `x` exists, then step 2 can be recursively applied to the `x`'th child until the child is an empty tree.
4. If the `x`'th child is an empty tree, then the key does not exist in the tree and `x` is the insertion point for the key.

The proof that the insertion maintains the height balance is simple, straightforward and intuitive:

1. The initial call to the root node involves a split up with a no-op splice of the root node, which has no effect on the tree's balance.
2. The recursive call involves a split up and splice pair, which does not affect the height of the tree.
3. The insertion at the leaf involves replacing an empty tree with a 2-node tree that is spliced into its parent. This does not change the height of the tree.
4. Thus, there is no net effect on balance of the tree.

The complexity analysis for the insertion is trivial:

1. All operations at a node are worst case O(order).
2. All the algorithm does is to recur once down to the bottom of the tree and then return.
3. Therefore the overall complexity of the algorithm is O(log N) where N is the number of elements in the tree since the tree is balanced.

6 Self-Balancing Tree Deletion Algorithm

As with the insertion algorithm, the deletion algorithm closely follows the abstract description of what deletion must do. Listing 5 shows the complete Java implementation. The deletion code is essentially the same as the insertion code except that it identifies the state = 1 (2-node state) as a special case and it pushes data downward as well as upward.

We first describe a helper algorithm to delete a key from a host and re-establish the tree order and height balance using a supplied command.

Helper algorithm: delete key from host tree using a splice command to maintain height balance and tree order. The supplied splice command has access to the host's parent.

- **Empty host case:**
  1. Do nothing.
- **State = 1 (2-node) host case:** Note that a 2-node host must be a leaf.
  1. Key == data: Split down host tree (deletes the root element from the tree) and return the data element
  2. Key != data: Call supplied splice command to splice this tree back into its parent.
- **State > 1 (default) host cases:**
  1. Calculate the candidate key in the node, who as a tree with its left and right children trees, is guaranteed to hold the supplied key if it exists in the whole tree.
  2. Split down the host at the candidate key.
  3. Collapse the new 2-node child tree with its two children.
  4. Recursively delete the key from that child, passing a command that can properly splice the child to the host.
  5. Execute SplitUpAndSplice on the host passing it the splice command to splice the resultant 2-node host into its parent at the host's location in the parent.

The deletion algorithm simply sets up the above helper algorithm and passes to it a no-op splice command since the root of the tree has no parent.

Delete key from host tree:.

- **Empty host case**:
  1. Do nothing.
- **State = 1 (2-node) host case**:
  1. Collapse the tree with the two children.
  2. Delegate to the default case command.
- **State > 1 (default) host cases:**
  1. Delegate to helper algorithm with a no-op splice command

The 2-node case is singled out for two reasons: The first is that when the root is a 2-node, data cannot be pushed downward from it, so it needs to be collapsed before the split down process begins. This is what one expects because the deletion process will cause the tree to shorten after enough data elements have been removed. Essentially that point is reached when the root runs out of data to push downward. Having a 2-node root does not guarantee that the tree will shorten on the next deletion however, due to the excess data being pushed upwards to the root. The second reason for singling out the 2-node case is that when a data element is split down from a leaf, it forms a 2-node *below* the leaf level. This then serves as an indication that the leaf level has been reached. It also conveniently and automatically isolates the key to be deleted from the rest of the tree. This situation only occurs during the execution of the helper algorithm, so it is also clearly separated from previously described root node situation.

Pushing the candidate data downward is accomplished by pairing a split down operation with a "collapse" operation as described in Section 4. The collapsing process may create a virtual state if the two children's states sum to the order of the tree (+1 due to the pushed down data). Once again, this is easily handled by the system, as it is still an operational state of the tree. Since one of the data elements of the virtual state is pushed down to the next level, when the excess data is spliced back in during the recursion's return, the splitting up process will split the virtual state in two. Since the virtual state is split in the middle, the resultant children are guaranteed to have states less than or equal to the order.

Code	Comment
`public class DeleteNAlgo extends ATreeNAlgo {` `  private SplitUpAndSplice  splitUpAndSplice;`	The constructor initializes the SplitUpAndSplice algorithm to split up only those trees whose state > order.
`  public DeleteNAlgo (int order) {` `    splitUpAndSplice = new SplitUpAndSplice(order, noOpHostCmd);`	The key to delete is the parameter of the host's execute() method.
`    addCommand(noOpParamCmd);`	The root level no-op empty case command is loaded. The key is returned.

`    addCommand(new ICommand() {`
`      public Object apply(int idx, TreeN host, Object key) {`
`        return getDefaultCmd().apply(idx, collapse2Node(host), key);`
`      }`
`    });`
	The root level 2-node (state=1) case is loaded. This case collapses the 2-node and then delegates to the default command.

`    setDefaultCmd(new ICommand() {`
`      public Object apply(int idx, final TreeN host, final Object n) {`
`        return host.execute(new ATreeNAlgo() {`
`          {`
`            final ATreeNAlgo me = this;`
	The default (state>1) case is loaded. The initial call simply sets up a non-recursive call to a helper algorithm, passing a no-op splice command.
	The last line is needed to be able to recur on the anonymous inner object.

`            addCommand(noOpParamCmd);`	The helper's no-op empty case command is loaded. The param is returned.

`            addCommand(new ICommand() {`
`              public Object apply(int idx, TreeN h, Object cmd) {`
`                if (h.getDat(0).equals(key)) {`
`                  Object d = h.getDat(0);`
`                  h.splitDownAt(0);`
`                  return d;`
`                }`
`                else {`
`                  ((ILambda)cmd).apply(h);`
`                  return h.getDat(0);`
`                }`
`              }`
`            });`
	Load the 2-node (state=1) case command. The host definitely has a parent here. This case encountered only if the data has been pushed down through a leaf.
	If the key is found, then delete it from the 2-node using a split down.
	If key is not found, splice the candidate data back into the parent.

`            setDefaultCmd(new ICommand() {` `              public Object apply(int idx, final TreeN h, final Object cmd) {`	The helper's default (state>1) case command is loaded.

`                final int x = findX(h, idx, ((Integer)key).intValue());`
`                h.splitDownAt(x);`
`                TreeN newChild = collapse2Node(h.getChild(x));`
	Find the candidate key, split the host down there and collapse the new child tree.

`                Object result = newChild.execute(me, new ILambda() {`
`                  public Object apply(Object child) {`
`                    return h.spliceAt(x, (TreeN)child);`
`                  }`
`                });`
	Recur, passing it the command (*ILambda*) to splice at the computed insertion point. The splice command's closure is effectively a memento that holds the child insertion point.

`                h.execute(splitUpAndSplice, cmd);` `                return result;`	Split this host if necessary and splice the excess data into the parent using the splice command supplied by this host's parent.

`              }`
`            });`
`          }`	

`        }, new ILambda() {`
`            public Object apply(Object child){`
`              return host;`
`            }`
`        });`
`    }`
`  });`
`}`
	The no-op splice command passed to the first call of the helper on the root node.

`  private final TreeN collapse2Node(TreeN t) {`
`    t.spliceAt(1,t.getChild(1));`
`    return t.spliceAt(0,t.getChild(0));`
`  }`
	Utility method to collapse a 2-node tree with its children.

`  private final int findX(TreeN t, int state, int k) {`
`    for(int i = 0;i< state; i++) if(t.getDat(i).intValue()>=k) return i;`
`    return state-1;`
`  }`
`}`
	Utility method for linear search for the candidate data element. Candidate may actually be the key. Could use a binary search.

Listing 5: Self-balancing tree deletion algorithm

Conspicuously absent in the above algorithm are the traditional rotation operations. Rotations occur when locally, there aren't enough data elements to maintain the tree height. The above algorithm ensures the proper amount of data by always pushing down data from the root. In addition, the collapsing and splitting up of the nodes promotes tree fullness better than the single element transfer in a rotation operation does.

As is with the insertion code, the deletion code is declarative and follows the proof of correctness exactly in the following proof that the key to be deleted can be found:

1. An empty tree does not contain the key.
2. If a tree contains the key, then it is obvious that there exists an index x such that the 2-node tree formed by the x'th element with its original left and right child trees contains that key.
3. It is obvious that if a 2-node tree contains the key, then the collapsed tree contains the key.
4. Steps 2 and 3 can be recursively applied to the x'th child formed by splitting down the tree at x.
5. If the result after Step 2 is a 2-node tree, then we are at a leaf level. The key exists in the tree if and only if the data element is equal to the key.

The proof that the deletion maintains the tree's balance is simple, straightforward and intuitive:

1. The initial call to the root node may involve collapse without a split down, which has no effect on the tree's balance.
2. The recursive call involves a split down paired with a collapse, which has no net effect on the tree's height.
3. At the leaf level, the split down is paired with either a splice with the parent or a split down of a 2-node tree, neither of which has any net effect on the tree height.
4. The return from the recursive call involves a split up paired with a splice, which has no net effect on the tree height.
5. At the root level, the split up of the child is paired with a no-op splice of the root node, which has no effect on the tree's balance.
6. Thus, there is no net effect on balance of the tree.

The complexity analysis once again is trivial:

1. All operations at a node are worst case O(order)
2. All the algorithm does is to recur once down to the bottom of the tree and then return
3. Therefore the overall complexity of the algorithm is O(log N) where N is the number of elements in the tree.

7 Conclusion

We have presented our tree framework and exhibited its complete Java implementation. As Listing 4 and Listing 5 show, the code of SBT insertion and deletion are each simple enough to easily fit on one page but yet be almost a word for word match with its proof of correctness.

In our framework, the tree structure serves as the re-usable invariant component with a complete and minimal set of intrinsic behaviors. The behaviors are partitioned into constructors, structural modifiers and data access. The extrinsic algorithms that operate on the tree act as visitors and add an open-ended number of variant behaviors to the tree.

We generalize the visitor pattern by modeling visitors as collections of semantically equivalent commands which can be dynamically loaded. Individual visiting methods are replaced with a single parameterized method. Default behavior capability in our implementation allows an additional level of program abstraction, greatly simplifying the algorithms. The generalized visitor can handle a dynamically changing number of hosts, or in this case, a single host with dynamically changing numbers of states, each of which may require a different behavior from the visitor.

The insertion and deletion process on a SBT relies on vertical data movement that preserves the height balance of the tree. The intrinsic structural operations of the tree were shown to easily support this process. The insertion and deletion algorithms were then expressed in terms of leaf manipulations, vertical data movement and root manipulations. Their implementations closely matched their abstract descriptions and their proofs of correctness and complexity analysis were simple, straightforward and intuitive. These algorithms, when plugged into the tree framework, transform the tree structure into a SBT. This demonstrates the framework's flexibility and extensibility. The algorithms can be easily modified to support other self-balancing tree structures such as B-trees.

It should be noted that students must be prepared in advance before encountering this material. Students must be versed and proficient in abstract behaviors, frameworks systems and design patterns, especially the visitor and command patterns.

OO and design patterns promote a drive towards proper abstraction. The students can focus on the fundamental principles involved with the system without the distractions of low-level manipulation code. Abstract concepts such as closures, lambda functions, itemized case analysis and other abstract behaviors are well represented in our formulation. Functional programming and declarative

programming come in naturally without the traditional topical boundaries that hinder students' learning.

8 Acknowledgement

The authors would like to thank Caleb Hyde for his hard work in helping make sense of the system during its development.

References

[1] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[2] Cormen, T., Leiserson, C., Rivest, N., and Stein, C. *Introduction to Algorithms, 2nd ed*., MIT Press, 2001.

[3] Nguyen, D. and Wong, S. *Design Patterns for Decoupling Data Structures and Algorithms*, SIGCSE Bulletin, 31, 1, March 1999, 87-91.

[4] Nguyen, D. and Wong, S. *Design Patterns for Lazy Evaluation*, SIGCSE Bulletin, 32, 1, March 2000, 21-25.