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

Abstract

We describe how we lead students through the process of specifying and implementing a design of mutable tree data structures as an object-oriented framework. Our design entails generalizing the visitor pattern in which the tree structure serves as host with a varying number of states and the algorithms operating on the tree act as visitors.

We demonstrate the capabilities of our tree framework with an object-oriented insertion algorithm and its matching deletion algorithm, which maintain the host tree's height balance while constrained to a maximum number of elements per node. We implement our algorithms in Java and make extensive use of anonymous inner classes. The key design elements are commands manufactured on the fly as anonymous inner objects. Their closures provide the appropriate context for them to operate with little parameter passing and thus promote a declarative style of programming with minimal flow control, reducing code complexity.

Our framework serves to illustrate how proper abstraction liberates us from thinking of low-level procedural details and leads us to implementations that closely reflect the essence of the system behavior. Our formulation is also an example of how object-oriented design principles overlap if not encompass those of functional and declarative programming.

Keywords

algorithm, B-Tree, closure, component software, data structure, declarative programming, design pattern, finite state machine, framework, functional programming, Java, inner class, lambda, object-oriented programming, self-balancing tree

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 $O(\log N)$ 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-dependent calculations needed to maintain its balance. The resulting morass of data manipulations hides the underlying concepts and hampers the students' learning.

We seek to alleviate the difficulties faced by students by offering an object-oriented (OO) formulation of SBTs, which is much easier to express and implement. We cover SBTs towards the end of our second semester (CS2) course. Our CS2 course introduces students to OO program design and the fundamental data structures and algorithms. It emphasizes proper formulation and abstraction of the problem domain in the programming process in order to build programs that are robust, flexible, and extensible. It teaches how design patterns help formulate and implement abstractions in effective and elegant ways. By the end the course, when the SBT material is presented, the students are already grounded in such principles as data and behavioral abstraction and the separation of variant (extrinsic) from invariant (intrinsic) behaviors. They are also familiar with common design patterns such as composite, state, visitor and command. An important lesson they learn from designing the SBT is how to abstractly decompose a problem by asking fundamental questions about the system and focusing on its intrinsic requirements. Hence, a major focus of this paper will be the thought progression involved with the design process. This advanced topic serves to hone and coalesce the concepts and skills practiced throughout the semester.

Our work is based on the framework proposed in 1999 by Nguyen and Wong [3]. Their framework decouples algorithms and data structures using a combination of composite, state and visitor design patterns. Later they illustrated its extensibility and flexibility by transparently adding lazy evaluation capabilities [4]. However, their simple 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 previous Nguyen/Wong 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. Effective use of polymorphism streamlines the code, facilitates

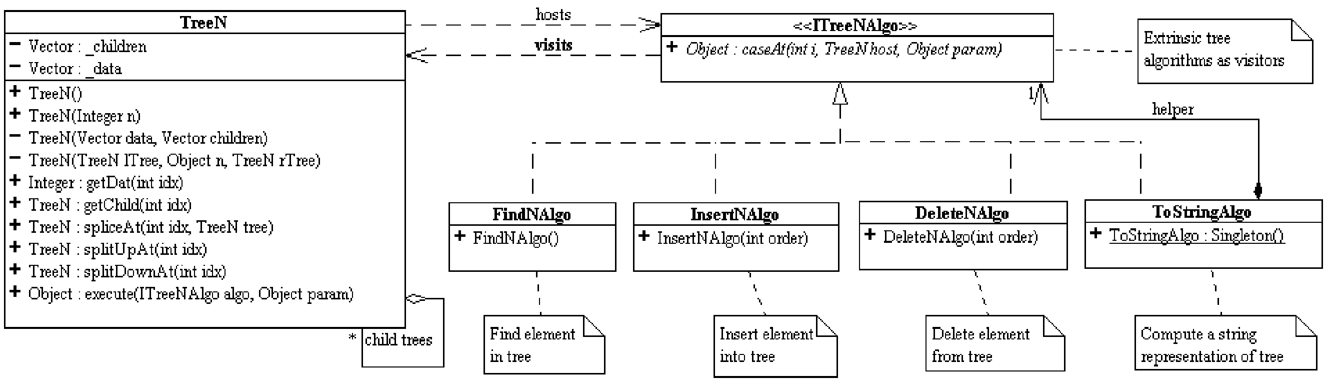
TreeN
− Vector : _children
− Vector : _data
+ TreeN()
+ TreeN(Integer n)
− TreeN(Vector data, Vector children)
− TreeN(TreeN lTree, Object n, TreeN rTree)
+ Integer : getDat(int idx)
+ TreeN : getChild(int idx)
+ TreeN : spliceAt(int idx, TreeN tree)
+ TreeN : splitUpAt(int idx)
+ TreeN : splitDownAt(int idx)
+ Object : execute(ITreeNAlgo algo, Object param)
* child trees
hosts
visits
<<ITreeNAlgo>>
+ Object : caseAt(int i, TreeN host, Object param)
Extrinsic tree algorithms as visitors
1
helper
FindNAlgo
+ FindNAlgo()
InsertNAlgo
+ InsertNAlgo(int order)
DeleteNAlgo
+ DeleteNAlgo(int order)
ToStringAlgo
+ ToStringAlgo : Singleton()
Find element in tree
Insert element into tree
Delete element from tree
Compute a string representation of tree

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

straightforward proofs of correctness and trivializes the complexity analysis.

Section 2 explores the fundamental nature of trees with multiple data elements per node. The result is a specification and implementation of a minimal and complete set of behaviors that are intrinsic to the tree structure. The number of elements in 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.

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. The standard visitor pattern is extended to handle the arbitrary numbers of hosts encountered in an SBT system. 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 four basic operations that transport data vertically by one level and modify the tree structure while maintaining its balance. These operations serve as building blocks for the insertion and deletion algorithms.

Section 5 describes our SBT insertion algorithm and its Java implementation. The algorithm's intuitive heuristics will lead to a rigorous proof of correctness. The 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 heuristics are intuitive and lead to a rigorous proof-of-correctness. Since both insertion and deletion rely on vertical transportation of data, their complexity analyses are identical.

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. The first step with the students is to lead them to a concise and precise definition of the problem, that is, what *exactly* is the data structure under consideration? Recursive data definitions are fundamental not only to good OO design but to computing in general.

2.1 Data Definition

A **TreeN** *can be either empty or non-empty.*

- *An empty* **TreeN** *contains nothing.*
- *A non-empty* **TreeN** *holds an arbitrary, positive, number of data elements, n, and n+1* **TreeN** *objects called "child trees". For* 0 <= **i** < *n, the* **i**th *and the* **i+1**th *child trees are called the left child tree and the right child tree of the* **i**th *data element respectively.*

The above inductive 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" which determine the tree's behavior at any given moment. The state of the tree is defined by the number of data elements in the root node of the tree. We can thus identify 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.

2.2 Intrinsic *vs.* Extrinsic Behavior

The next step with the students is to identify the *intrinsic* behaviors of the system and declare them as the public methods 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 are those that serve exactly two purposes:

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

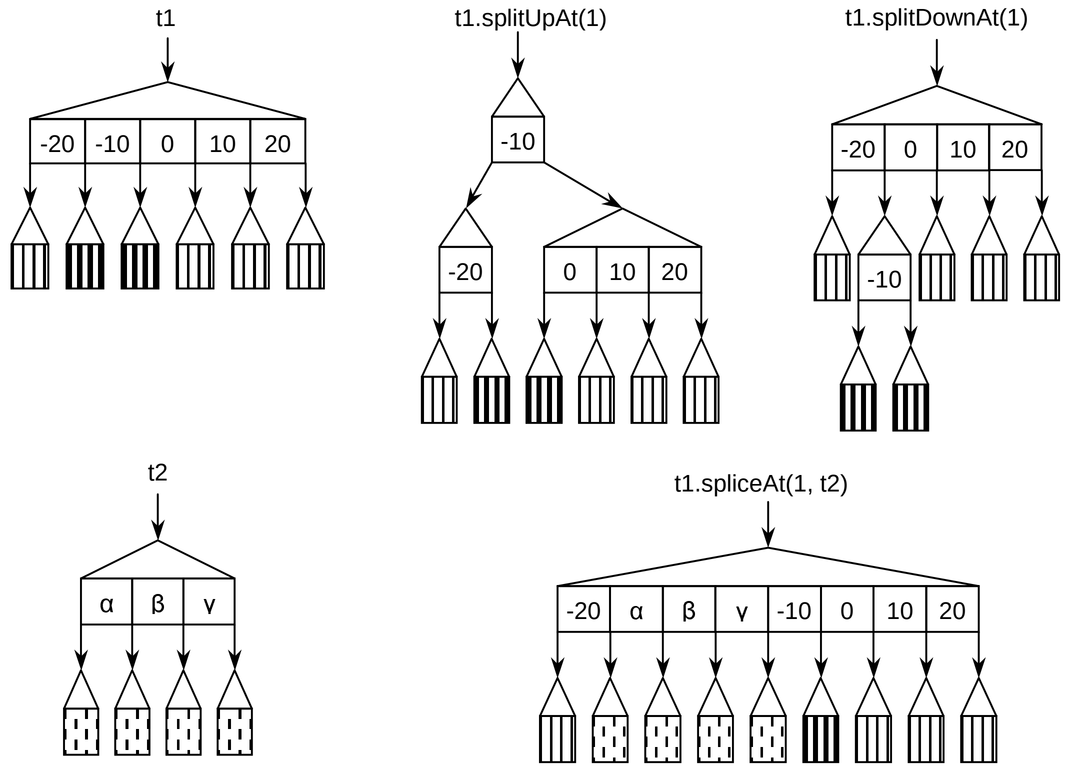

Figure 2: Intrinsic structural operations on the tree.

The intrinsic structural behaviors of the tree are *invariant*, that is they remain fixed in all applications, and enable us to build trees of *any* shape that can hold an *arbitrary* number of data elements in *any* node. Extrinsic behaviors are those that are dependent on a particular application of the tree and are thus *variant*. The variant behaviors will be addressed in Section 3. Distinguishing and separating intrinsic from extrinsic object behaviors is a central theme in our approach to OO pedagogy throughout the course.

To identify the intrinsic operations of the tree, it is crucial that we 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. Delineating data manipulators from structural modifiers eliminates the usual problem of insertions and deletions that can only be unambiguously applied to a limited number of tree configurations.

Figure 1 depicts the UML class diagram of **TreeN** together with algorithms that act as visitors (discussed in Section 3). **Figure 2** illustrates the intrinsic structural operations of the tree (discussed below) and **Listing 1** shows the Java implementation of **TreeN**.

2.3 Constructors

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 states > 1 as they can be created through structural modifications of 2-node leaf trees. The set of constructors is thus complete and minimal.

2.4 Structural Modifiers

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 three methods constitute a complete and minimal set is beyond the scope of this paper. An examination of the following three methods in **Figure 2** reveals that they create constructive and destructive behavior in *both* the horizontal and vertical directions.

splitUpAt(int i) mutates the receiver, in state **s**, into a 2-node tree (state = **1**), where the **i**th element becomes the root data and the left child's root contains the **0** through **i-1** elements of the original root and the right child's root contains the **i+1** through **s** elements of the original root. Splitting up on an empty tree is a no-operation.

splitDownAt(int i) removes the **i**th element from the root of the receiver including its corresponding left and right child trees. The resultant new child tree is a 2-node tree where its root data is the original **i**th element and where its left and right children are the original **i**th element's left and right children respectively. Splitting down a 2-node tree results in an empty tree and is equivalent to a deletion of a single data element. Splitting down an empty tree is a no-operation.

spliceAt(int i, TreeN t) joins the supplied source tree **t** to the receiver at index **i**: the **i**th child of the receiver is deleted and the root node of **t** is "spliced" between the **i**th and **i+1**th elements of the receiver. The children of **t** remain in their respective places with regards to the original elements of **t**. 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 receiver tree into a shallow copy of the source tree.

2.5 Data Accessors

getDat(int i) and **getChild(int i)** 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 **i**, where $0 \leq$ **i** $<$ state (= node size). The root node's child trees can be accessed similarly but where $0 \leq$ **i** $\leq$ 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.

The standard "settors" that set a child tree to a new tree at index **i**, and that set a data element at index **i**, can be easily replicated using a combination of the above methods. This is a simple and elucidating exercise for the students.

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 should be decoupled from the *intrinsic* structural behaviors of the tree. The visitor pattern, with the extrinsic algorithms as visitors and the tree structure as the host, was used to achieve this decoupling. The *ability* of the tree structure to perform *all* possible extrinsic operations is an *intrinsic* behavior of the tree and can be expressed as a "hook" method.

2.6 Extensibility Hook

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.

3 The Visitors

With the intrinsic behaviors aside, the students can now concentrate on the extrinsic, variant behaviors of the system. The students are lead to focus on the following two key characteristics of the system.

```
public class TreeN {
   private Vector _children = new Vector();
   private Vector _data = new Vector();

   public TreeN() { }
   public TreeN(Integer n) {  this(new TreeN(), n, new TreeN()); }

   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);
   }
   public Integer getDat(int i) { return (Integer)_data.get(i); }
   public TreeN getChild(int i) { return (TreeN)_children.get(i); }

   public TreeN spliceAt(int i, TreeN tree) {
      int k =tree.data.size();
      if (k > 0) {
         if (_data.size() > 0)   _children.set(i, tree.getChild(k--));
         else   _children.add(i, tree.getChild(k--));
         for (; k >= 0, k--) {
            _data.add(i, tree.getDat(k));
            _children.add(i, tree.getChild(k));
         }
      }
      return this;
   }
   public TreeN splitUpAt(int i) {
      if (_data.size() > 1) {
         TreeN lTree, rTree;
         Vector newData = new Vector(), newChildren = new Vector();
         Object rootDat = _data.remove(i);
         for (int k = 0; k < i; k++) {
            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;
   }
   public TreeN splitDownAt(int i) {
      if (_data.size() > 1) {
         TreeN newChild =
         new TreeN(getChild(i),_data.remove(i),getChild(i+1));
         _children.remove(i);
         _children.set(i, newChild);
      }
      else {
         _data.clear();
         _children.clear();
      }
      return this;
   }
   public Object execute(ITreeNAlgo algo, Object param) {
      return algo.caseAt(_data.size(), this, param);
   }
}
```

Listing 1: TreeN implementation

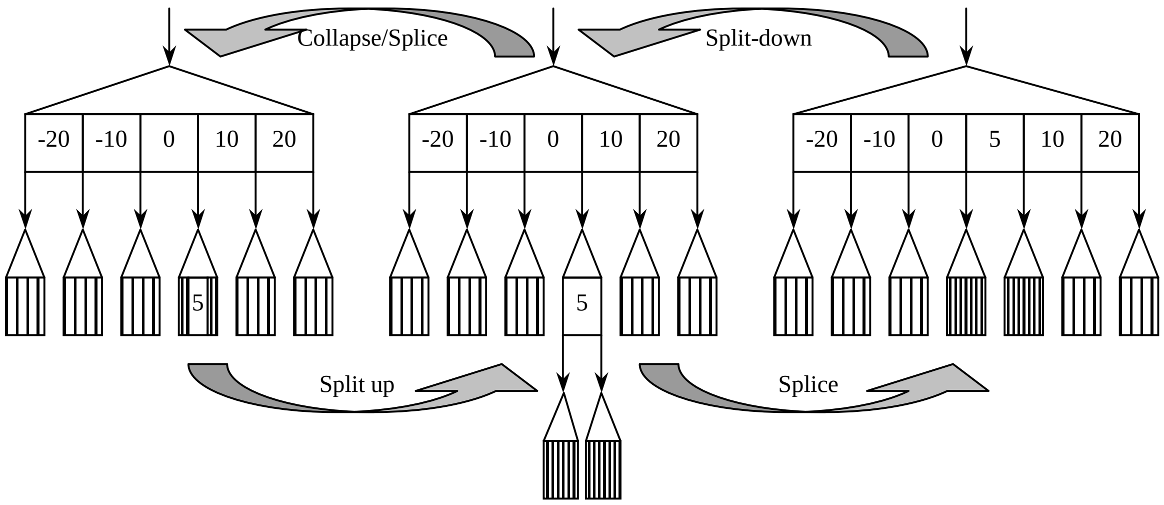

Figure 3: Height-preserving vertical data movement in the tree.

1. All extrinsic operations can be constructed solely from the intrinsic behaviors.
2. Extrinsic behaviors can depend on the state of the tree, which is dynamic and arbitrary.

The visitor design pattern has been proven useful for implementing extrinsic operations [3] but is inadequate for the problem at hand. The students' prior grounding in the visitor pattern enables them to easily understand its limitations and the extensions required to overcome them.

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, *all* other algorithms on the tree can be expressed in terms of these primitive behaviors and encapsulated as visitors.

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

Visitor (implements ***ITreeNAlgo***) must

- provide a "visiting" method, namely **Object caseAt(int s, TreeN h, Object p)**, to operate on a host tree **h** that is in state **s** with a given input **p**;
- guarantee that this visiting method has a well-defined behavior for all values of **s**. This includes the possibility of throwing an exception or of performing a no-operation.

Host (**TreeN**) must

- be in some state characterized by an integer **s**;
- provide
    - o a complete set of public methods for all *intrinsic* structural and data access operations, and
    - o a "hook" method, namely **Object execute(ITreeNAlgo v, Object p)**, to perform any *extrinsic* tree algorithm **v** with a given input **p. v.caseAt(…)** is guaranteed to be called with the host's current state, the host and **p** as arguments. The result will be returned.

The visitor design pattern is a small-scale example of a component-framework system. One of the benefits of the inverted control structure of component-framework systems is that the component (the visitor) is guaranteed to have the correct behavior called for any given state of the framework (the host).

4 Self-Balancing Trees

As before, students are lead to 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.

Fundamentally, a tree is not constrained to any sort of ordering or balancing—this is the bailiwick of particular insertion/deletion algorithms. 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 with the following "*search tree property*" (STP):

1. The root data elements, $\mathbf{x_i}$ are in *strict* ascending order.
2. All data elements in the **i**th child tree, if it exists, are less than $\mathbf{x_i}$.

3. All elements in the **i+1**th child tree, if it exists, are greater than **x_i**.
4. All subtrees satisfy the STP.

As a result, there are no duplicate elements.

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.

To help students craft algorithms for creating and maintaining a balanced tree of a given order, we first discuss four simple operations that, in effect, move data elements vertically by one level, mutating the tree structure and yet preserving the tree's balance and order. Students will discover that vertical transportation of data in a tree, without disturbing the height balance, is crucial in insertion and deletion into/from balanced trees. Understanding this process will enable them to design and implement a relatively simple yet efficient insertion or deletion algorithm for SBTs.

4.1 Splitting up the root node

For a tree whose root node has more than one data element, it is evident that growing the tree upward at the root will increase the overall tree height by one and still maintain the tree's balance. This can be done by a simple call to **splitUpAt(…)** as illustrated in **Figure 2**.

4.2 Collapsing the root node

Splicing all the children into the root ("collapsing") will decrease the overall tree height by one and still maintain the tree's balance. This can be done by repeated calls to **spliceAt(…)** as illustrated in **Figure 2** (consider **t2** as a child tree of **t1**).

4.3 Lifting data upward

To move a data element up one level in a tree, without disturbing the height balance or ordering, can be accomplished by combining a split-up operation, **splitUpAt(…)**, with a splice operation, **spliceAt(…)**. This process is illustrated in **Figure 3,** reading left to right. Here 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 at the parent's #3 index. The desired destination for the "5" is in the #3 data element location in the parent tree. A split-up operation followed by a splice operation neatly moves the 5 up one level without disturbing the height balance or ordering.

4.4 Pushing data downward

Moving data down one level in the tree without disturbing its height balance and ordering can be accomplished by combining a split-down operation, **splitDownAt(…)**, with a collapse operation (4.2). Once again, consider the example in **Figure 3**, reading from right to left this time, where we will move the 5 back to its original position. This is easily accomplished by a split-down operation followed by a collapse operation. Clearly, height balance and ordering is preserved.

The above four operations serve as basic building blocks for maintaining the invariants of the tree's balance and ordering. The students are now ready for the discussion of the insertion and deletion algorithms.

5 Self-Balancing Tree Insertion

Self-balancing tree insertion of a key **k** into a **TreeN T** entails applying the **spliceAt(…)** method to an appropriate subtree of **T** while maintaining three invariants: the STP (search tree property), the height balance constraint, and the order constraint.

5.1 Heuristics

It is easy to convince students that, when the tree **T** is not empty, insertion of the key **k** must take place at a leaf because only by traversing **T** all the way down to a leaf can one make the determination whether or not **k** already exists in the tree. Once we are at the appropriate leaf **L**, the call **L.spliceAt(x, new TreeN(k))** will insert **k** into **L** at an appropriate index **x** and maintain the STP of **T**. However, such splicing may cause the number of data element in the leaf node to exceed the prescribed order by one. In this case, **L** is said to be in a "*virtual state*" because such a state violates the order constraint, but is still a valid operational state of the tree.

The discussion in the preceding section (4) suggests transporting the excess data upward towards the root in order to re-attain the normal state while maintaining the tree's height balance and STP. This can be accomplished by repeated application of the split-up and splice combination described in subsection 4.3, before reaching the root. When the data transport reaches the root, only a split-up (4.1) is required if the root in a virtual state. This is equivalent to a split-up followed by a no-op. We focus the students on this vertical data transport, as it is the key to self-balancing.

We abstract the splice and no-op into an abstract function, or in the parlance of design patterns, a "command," ***ILambda*** (see **Listing 2**). This interface performs an arbitrary task on its parameter when its **apply(…)** method is called. Consider for example the following command:

```
ILambda cmd = new ILambda() {
                    public Object apply(Object tree) {
                       return T.spliceAt(x, (TreeN)tree);
                       }
                  };
```

cmd.apply(S) will splice any **TreeN S** to the given tree **T** at the given index **x**. On the other hand, the anonymous ***ILambda*** object

```
new ILambda() {
   public Object apply(Object tree) {
      return T;
   }
}
```

can serve as a no-op command.

In the above two commands, **T** and **x** are *free* and must be bound in some non-local environment in order for the commands to make sense. In Java, such bindings are made possible using inner classes. The commands are created anonymously on-the-fly and

Code	Comment
`public interface ILambda {`	Function abstraction representing a command. A function knows how to perform an operation on an input Object p and return the result as an Object.
`   public Object apply(Object p);`	
`}`	
`public class SplitUpAndApply  implements ITreeNAlgo {`	
`      private int _order;`	Constructor takes the order of the tree.
`      public SplitUpAndApply(int order) { _order = order; }`	
`      public Object caseAt(int s, TreeN host, Object cmd) {`	**cmd** is an ***ILambda*** operating on **TreeN**.
`            if (s <= _order)  return host;`	Case **s** = state <= order: do nothing (no-op).
`            host.splitUpAt(s/2);`	Default (state>order) case: split up **host** at its midpoint and then apply the ***ILambda*** parameter to **host**.
`            return ((ILambda) cmd).apply(host);`	
`      }`	
`}`	

Listing 2 *ILambda* command and SplitUpAndApply visitor

passed as input parameters to the process of transporting data upward encapsulated in a ***ITreeNAlgo*** visitor called **SplitUpAndApply** (see **Listing 2**) which can be expressed as follows.

- Host **state** $\leq$ **order**: do nothing.
- Host **state** > **order**: Split up the host and apply the supplied abstract ***ILambda*** command to the host which will either perform a no-op or a splice.

Listing 3 shows the code for our insertion algorithm that makes use of the above **SplitUpAndApply** visitor and ***ILambda*** commands. Our implementation is an adaptation of a common technique in functional programming (for example, see [5]). More specifically, in the case of non-empty host trees, the insertion algorithm simply sets up a "helper" algorithm and passes a no-op command to it since at the top level call the host is the root.

As shown in **Listing 3**, the helper algorithm and the splice commands are created on the fly as an anonymous inner objects, which are analogous to lambda expressions in functional programming. They allow us to express computations in terms of data in their closures. The use of anonymous inner classes greatly simplifies the code by minimizing parameter passing and control structures. This often results in program code that is declarative, clear and concise.

5.2 Correctness

To help the students gain a deeper understanding of the algorithm, we need to lead them through a more rigorous proof of correctness.

Case 0 of **InsertNAlgo** is trivial. The correctness of the default case hinges on the fact that the recursive call inside the helper algorithm does indeed insert the key into the host tree and maintains the host's STP and height balance. We label the anonymous helper algorithm instance **insertHelper** and claim the following.

Lemma 1: Let **T** be a balanced *non-empty* **TreeN** of order > 1 that satisfies the STP, and suppose **key** is *not* in the root node of **T**. Let **x** be the index at the root of **T** such that if the **x-1**th data element exists, it is strictly less than **key** and if the **x**th data element exists, it is strictly greater than **key**. Let **S** be the **x**th child tree of **T**. Then, the following post-conditions hold for the call

```
S.execute(insertHelper, new ILambda() {
   public Object apply(Object tree) {
      return T.spliceAt(x, (TreeN)tree);
   }
});
```

- **T** contains **key** and preserves the STP and its height.
- All subtrees of **T** satisfy the order constraint.

Proof of lemma 1: We shall prove by induction on the height of **T**.

- Case height = 1: **S** is empty. Thus case 0 of **insertHelper** is called and evaluates to
  T.spliceAt(x, new TreeN(key)).
  As a result, **T** contains only **key** at the root, clearly satisfies the STP and preserves its height. All subtrees of **T** are empty and trivially satisfy the order constraint.
- Suppose the lemma holds for all **TreeN T** of order > 1 and of heights h that satisfy the STP and such that $1 \leq h \leq n$. We will prove the lemma holds for the case of any **TreeN T** of order > 1 and of height **n+1** that satisfies the STP.
  a) Here, **S** is not empty and thus the default case of **insertHelper** is called with the parameter **h** bound to **S** and the parameter **cmd** bound to the anonymous object
  ```
  new ILambda() {
     public Object  apply(Object tree) {
        return T.spliceAt(x, (TreeN)tree);
     }
  }.
  ```
  The body of this case consists of the following three computations in sequence:
     - a **for** loop;
     - a call to execute recursively **insertHelper**;
     - a call to execute the **splitUpAndSplice** algorithm.

  We now examine the effects of each of the above computations.
  b) If **key** is in the root node of **S**, the **for** loop will find a match for **key** and return. Nothing affects **T**, and so the lemma holds.
  c) If **key** is *not* in the root of **S**, by the STP, the for loop finds the index **x[0]** of the child tree of **S** where **key**

Code	Comment
`public class InsertNAlgo implements ITreeNAlgo {` `private SplitUpAndApply  splitUpAndSplice;` `public InsertNAlgo (int order) { splitUpAndSplice = new SplitUpAndApply(order);}` `public Object caseAt(int s, final TreeN host, final Object key) {` `switch(s) {`	Initialize the **splitUpAndSplice** algorithm to split up only those trees whose state > **order**. The key to insert is the third parameter
`case 0: {  return host.spliceAt(0, new TreeN((Integer) key)); }`	The empty case simply splices a new tree into the host to mutate it into a 2-node tree.
`default: {` `host.execute(new ITreeNAlgo() {` `public Object caseAt(int s_help, final TreeN h, final Object cmd) {` `switch(s_help) {`	The default (state > 0) case simply sets up a call to a helper algorithm, passing a no-op command.
`case 0: {` `return ((ILambda)cmd).apply(new TreeN((Integer)key));` `}`	The helper's empty case: The parent tree has height 1 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.
`default: {` `final int[] x = {0};` `for (; x[0] < s_help; x[0]++) {` `int d = h.getDat(x[0]).intValue();` `if (d >= ((Integer)key).intValue()) {` `if (d == ((Integer)key).intValue()) return h;` `else break;` `}` `}`	The helper's default (state > 0) case: The tree has height at least 1 here. Linear search for the index of the child tree that will hold the key (the insertion point) and preserve the STP of the tree. Could alternatively use a binary search. If the key is in the host tree already, do nothing, else **x[0]** is the index of the child tree where the key should be inserted, so break out of the loop.
`h.getChild(x[0]).execute(this, new ILambda() {` `public Object apply(Object tree) {` `return h.spliceAt(x[0], (TreeN)tree);` `}` `});`	Recur on the helper, passing it an ***ILambda*** command to splice at the computed insertion point. The splice command's closure is effectively a memento that holds the previous insertion point.
`return h.execute(splitUpAndSplice, cmd);` `}` `};  // end switch(s_help)` `}`	If necessary, split this host and splice the excess data into the parent using the supplied ***ILambda*** command.
`}, new ILambda() {` `public Object apply(Object tree){` `return host;` `}` `});`	The no-op ***ILambda*** command is passed to the first call of the helper on the root node.
`return host;` `}` `};  // end switch(s)` `}` `}`	Return the host to allow chaining.

Listing 3: Self-balancing tree insertion algorithm

must be inserted in order to maintain the STP of **S**. Consequently, the conditions of the lemma holds with **S**, **S.getChild(x[0])** and **x[0]** in place of **T**, **S** and **x**, respectively.

d) By the induction hypothesis, after the recursive call

```
S.getChild(x[0]).execute(this, new ILambda() {
    public Object apply(Object tree) {
        return S.spliceAt(x[0], (TreeN)tree);
    }
});
```

- **S** contains **key** and preserves the STP and height.
- All subtrees of **S** satisfy the order constraint.

e) As a result, the only place where **S** can break the order constraint is at its root node. After the insertion of key, the size of the root node of **S** can only exceed the order by at most one. Thus, splitting up **S** will re-satisfy the order constraint. The call

```
S.excecute(splitUpAndSplice, cmd)
```

splits **S** up at the middle and splices it to **T** at the index **x**, if **S** break the order constraint. From the discussion in subsection 4.3, such a combination of operations does not affect the height of **T** and the order of the subtrees of **T**. Moreover, from the very definition of **x**, the STP of **T** is preserved.

f) Thus, the lemma holds for **T** of height **n+1.**

- By induction, lemma 1 holds for all balanced **TreeN T** or order > 1 and of heights ≥ 1 that satisfy the STP.

We are now ready to prove correctness of the insertion algorithm **InsertNAlgo**.

Theorem 1: The algorithm **InsertNAlgo** inserts without duplication an Integer **key** into a balanced **host TreeN** and maintains **host**'s STP and height balance.

Proof of theorem 1: We shall prove the theorem by considering the two cases: **host** is empty and **host** is not empty.

- When **host** is empty, case 0 is called: **host** simply mutates to a 2-node tree containing **key**. Obviously, **host** maintains the STP, the height balance and the order constraint.
- When **host** is not empty, the default case is invoked, which calls for **host** to execute the insert helper algorithm with parameter **h** bound to **host** and the parameter **cmd** bound to the no-op anonymous object

```
new ILambda() {
   public Object apply(Object tree) {
      return host;
   }
}
```

  a) As in the proof of lemma 1, if **key** is in the root node of **host**, the **for** loop will find a match for **key** and return from the call. Nothing affects **host**, and so the theorem holds.
  b) If **key** is not in the root node, then the recursive call to the helper is made. It follows from lemma 1 that **host** then contains **key**, preserves its STP and height and all subtrees of **host** satisfy the order constraint.
  c) The size of the root node of **host** can only exceed the prescribed order by at most one. If this happens, the application of the **splitAndSplice** algorithm on **host** will split **host** up at the middle, re-establishing the order constraint and maintaining the height balance of **host**. There will be no splicing to a parent tree however, since the supplied command is a no-op. QED

The above proof illustrates many important techniques used in more theoretically oriented upper division courses.

5.3 Complexity Analysis

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

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. If the key to be deleted is at a leaf node, a simple call to **splitDownAt(…)** will remove the key from the tree. Thus, similar to the insertion case, self-balancing tree deletion *must* take place at *the leaf level* while maintaining three invariants: the STP (search tree property), the height balance constraint, and the order constraint. Once again, the students are lead to focus on the vertical transport of data.

6.1 Heuristics

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. We must therefore find a method of transporting data from the root to the leaves without affecting the height of the child trees. The problem is that there is a possibility that the data to be deleted is located in a 2-node, and data cannot be pushed down out of a 2-node without changing the tree's height. To solve this problem, consider the following facts:

1. Only trees with order > 1 can push data downwards without changing their height because data will be left in the node.
2. From fact 1, if one data element is added to all nodes in the tree from which data is being pushed downwards, then height preservation is guaranteed.
3. Any element x_i in a node, when combined with its left and right child trees, can be considered as the root data element of a 2-node tree.
4. From the STP of the tree, we can always identify a "*candidate*" element in the root node whose associated 2–node tree is guaranteed to hold the desired data element to be deleted, should it exist in the tree at all.

Recursively pushing a candidate element down from the root of the tree will effectively add one element to all child nodes along the path to from the root to the leaves that contains the element to be deleted. When the data to be deleted is encountered, it will *automatically* become the candidate element and continue to be pushed down to the leaves. Thus, except at the root and the leaf, height preservation is guaranteed during the data transport process. At the root, a height change is possible, but that will not affect the balance of the tree. At the leaf, the candidate element is either the element to be deleted, upon which it will be removed, or if it is not, just as the insertion case, this excess data will be transported back upwards to the root. In either case, no height change takes place.

The deletion algorithm is thus analogous to the insertion algorithm except that it transports data from the root to the leaves as well as from the leaves to the root. **Listing 4** 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, plus, it pushes data downward as well as upward.

The 2-node case is singled out because 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 from the leaves.

Just as in the insertion algorithm, for trees with states > 1, the deletion algorithm simply sets up an anonymous helper algorithm and passes to it an anonymous no-op splice command since the root of the tree has no parent. The helper algorithm is very similar to that in the insertion algorithm in that it uses an

Code	Comment

public class DeleteNAlgo **implements** ITreeNAlgo {
 	
private SplitUpAndApply splitUpAndSplice;
 	
public DeleteNAlgo (**int** order) { splitUpAndSplice = **new** SplitUpAndApply(order); }
 	
public Object caseAt(**int** s, **final** TreeN host, **final** Object key) {
switch(s) {
	Initialize the **splitUpAndSplice** algorithm to split up only those trees whose state > **order**. The key to insert is the third parameter.

case 0: { **return** null; }	Empty case: do nothing and return.
case 1: { collapse2Node(host); }	Case state = 1: collapse the 2-node and then fall through to the default case.

default: {
return host.execute(**new** ITreeNAlgo() {
public Object caseAt(**int** s_help,**final** TreeN h, Object cmd) {
switch(s_help) {
	Default (state>1) case: set up a call to a helper algorithm, passing a no-op splice command.

case 0: { **return** null; }	Helper's empty case: **key** is not in the tree; do nothing and return.

case 1: {
if (h.getDat(0).equals(key)) {
Object d = h.getDat(0);
h.splitDownAt(0);
return d;
}
else {
((ILambda)cmd).apply(h);
return null;
}
}
	Helper's state = 1 case: encountered only if the data has been pushed down through a leaf.
	
	If **key** is found, then delete it from the 2-node using a split down.
	
	If **key** is not found, splice the candidate key back into the parent.

default : {
final int x = findX(h, s_help, ((Integer)key).intValue());
TreeN newChild =
collapse2Node(h.splitDownAt(x).getChild(x));
	The helper's default (state>1): find the candidate key index, split **h** down at that point and collapse the resultant child tree. **h** still maintains its STP.

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

h.execute(splitUpAndSplice, cmd);
return result;
}
} // end switch(s_help)
}
	If necessary, split this host and splice the excess data into the parent using the supplied ***ILambda*** command.

}, **new** ILambda() {
public Object apply(Object child) { **return** host; }
});
}
} // end switch(s)
}
	The no-op ***ILambda*** command is passed to the first call of the helper on the root node because the root has no parent.

private 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 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 4: Self-balancing tree deletion algorithm

appropriate ***ILambda*** command to splice excess data into the parent tree.
In the helper algorithm, the 2-node case is singled out because 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.

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 tree in virtual state. 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. As in the insertion algorithm, the order constraint is maintained.

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 does the single element transfer in a rotation operation.

6.2 Correctness and Complexity

Let us label the anonymous helper algorithm **deleteHelper**. Analogous to the insertion algorithm, we prove correctness for **DeleteNAlgo** by first establishing the following lemma on **deleteHelper**:

Lemma 2: Let **T** be a balanced *non-empty* **TreeN** of order > 1 that satisfies the STP and **key** be an Integer object. Let **x** be the index at the root of **T** such that if the **x-1**th data element exists, it is strictly less than **key** and if the **x+1**th data element exists, it is strictly greater than **key**. Then, after executing

```
TreeN S = collapse2Node(T.splitDownAt(x).getChild(x));
S.execute(deleteHelper, new ILambda() {
    public Object apply(Object tree) {
        return T.spliceAt(x, (TreeN)tree);
    }
});
```

the following post-conditions hold:

- **T** does not contain **key** and preserves the STP and its height.
- All subtrees of **T** satisfy the order constraint.

The correctness of **DeleteNAlgo** then follows.

Theorem 2: The algorithm **DeleteNAlgo** removes an Integer **key** from a balanced **host TreeN** and maintains **host**'s STP and height balance.

We leave as exercises the proofs for lemma 2 and theorem 2 since they are essentially identical to those of lemma1 and theorem 1.

The complexity analysis once again is trivial and is identical to the analysis of the insertion algorithm.

7 Conclusion

We have presented our tree framework and exhibited its complete Java implementation. As **Listing 3** and **Listing 4** show, the code of SBT insertion and deletion are each simple enough to easily fit on one page without relying on non-intuitive manipulations.

The fundamental design principle is the separation of invariant behaviors from variant behaviors. 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 on the tree act as visitors and add an open-ended number of variant behaviors to the tree.

We generalize the visitor pattern by replacing individual visiting methods with a single parameterized method. 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 visiting behavior.

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 lead directly to rigorous proofs of correctness. The complexity analyses 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.

Studying OO SBTs reinforces students' understanding of abstract decomposition, OOP, design patterns, complexity analysis, and proof-of-correctness. The drive towards proper abstraction unifies all the above principles. 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 expressions, 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.

Acknowledgement

The authors would like to thank Caleb Hyde for his hard work in helping make sense of the system during its early 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, pp. 87-91.

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

[5] Felleisen, M., Findler, B., Flatt, M., Krishnamurthi, S. *How to Design Programs*, MIT Press, 2001.