An Extensive Examination of Data Structures Using C# 2.0

Visual Studio 2005

Scott Mitchell
4GuysFromRolla.com

Update January 2005

Summary: This article, the fourth in the series, begins with a quick examination of AVL trees and red-black trees, which are two different self-balancing binary search tree data structures. The remainder of the article examines the skip list data structure. Skip lists are an ingenious data structure that turns a linked list into a data structure that offers the same running time as the more complex self-balancing tree data structures. As we'll see, the skip list works its magic by giving each element in the linked list a random "height." In the latter part of the article, we'll be building a skip list class in C#, which can be downloaded. (28 printed pages)

Download the DataStructures20.msi sample file.

> **Editor's note** This six-part article series originally appeared on MSDN Online starting in November 2003. In January 2005 it was updated to take advantage of the new data structures and features available with the .NET Framework version 2.0, and C# 2.0. The original articles are still available at http://msdn.microsoft.com/vcsharp/default.aspx?pull=/library/en-us/dv_vstechart/html/datastructures_guide.asp.

> **Note** This article assumes the reader is familiar with C#.

Contents

Introduction

In Part 3 of this article series we looked at the general *tree* data structure. A tree is a data structure that consists of nodes, where each node has some value and an arbitrary number of children nodes. Trees are common data structures because many real-world problems exhibit tree-like behavior. For example, any sort of hierarchical relationship among people, things, or objects can be modeled as a tree.

A *binary tree* is a special kind of tree, one that limits each node to no more than two children. A *binary search tree*, or BST, is a binary tree whose nodes are arranged such that for every node *n*, all of the nodes in *n*'s left subtree have a value less than *n*, and all nodes in *n*'s right subtree have a value greater than *n*. As we discussed, in the average case BSTs offer $\log_2 n$ asymptotic time for inserts, deletes, and searches. ($\log_2 n$ is often referred to as *sublinear* because it outperforms linear asymptotic times.)

The disadvantage of BSTs is that in the worst-case their asymptotic running time is reduced to linear time. This happens if the items inserted into the BST are inserted in order or in near-order. In such a case, a BST performs no better than an array. As we discussed at the end of Part 3, there exist self-balancing binary search trees, ones that ensure that, regardless of the order of the data inserted, the tree maintains a $\log_2 n$ running time. In this article, we'll briefly discuss two self-balancing binary search trees: AVL trees and red-black trees. Following that, we'll take an in-depth look at skip lists. Skip lists are a really neat data structure that is much easier to implement than AVL trees or red-black trees, yet still guarantees a running time of $\log_2 n$.

> **Note** This article assumes you have read Part 3 of this article series. If you have not read Part 3, I strongly encourage you to read it thoroughly before continuing on with this fourth installment.

Self-Balancing Binary Search Trees

Recall that new nodes are inserted into a binary search tree at the leaves. That is, adding a node to a binary search tree involves tracing down a path of the binary search tree, taking left's and right's based on the comparison of the value of the current node and the node being inserted, until the path reaches a dead end. At this point, the newly inserted node is plugged into the tree at this reached dead end. Figure 1 illustrates the process of inserting a new node into a BST.

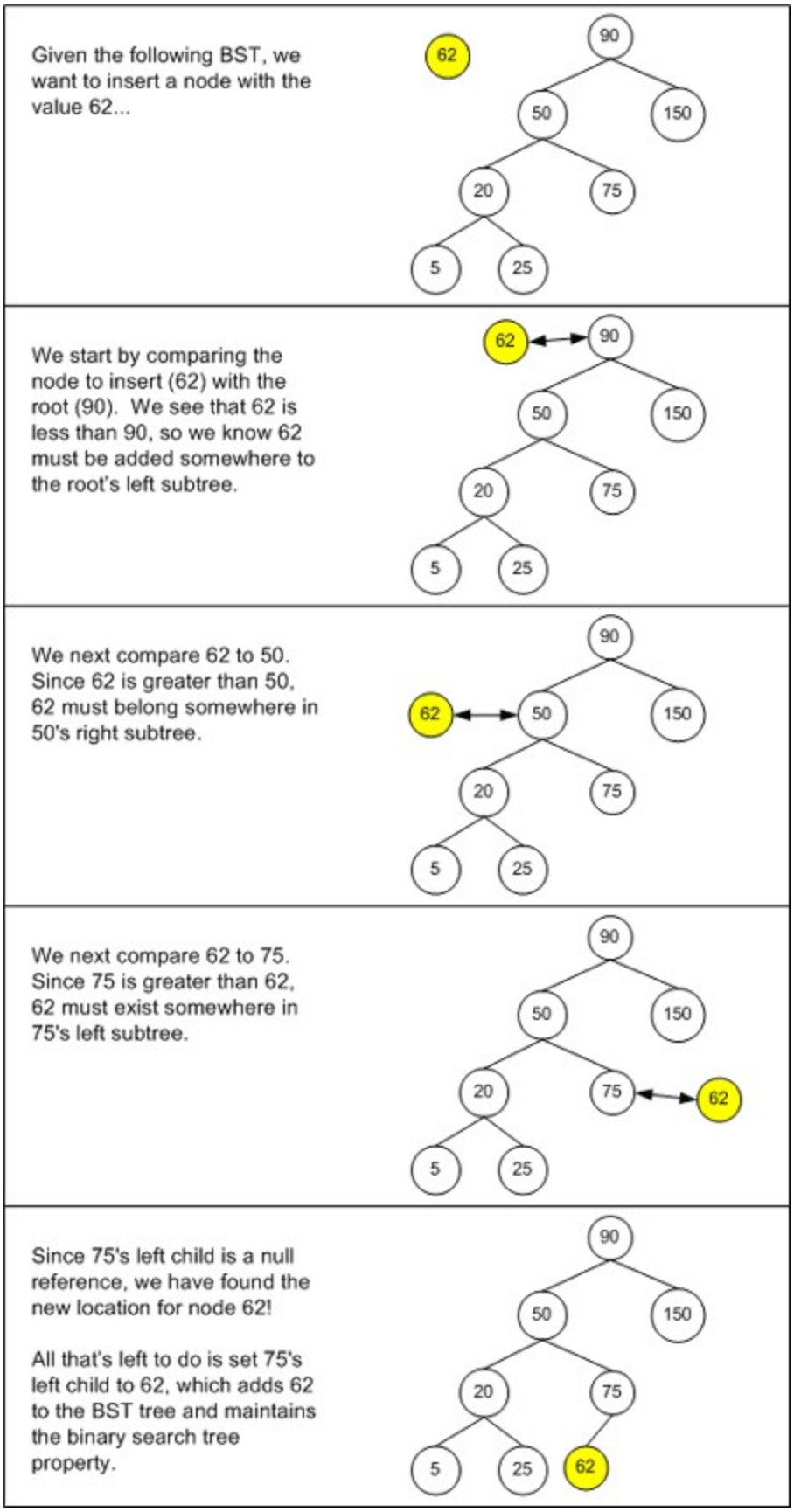

Figure 1. Inserting a new node into a BST

As Figure 1 shows, when making the comparison at the current node, the node to be inserted travels down the left path if its value is less than the current node, and down the right if its value is greater than the current's. Therefore, the structure of the BST is relative to the order with which the nodes are inserted. Figure 2 depicts a BST after nodes with values 20, 50, 90, 150, 175, and 200 have been added. Specifically, these nodes have been added in ascending order. The result is a BST with no breadth. That is, its topology consists of a single line of nodes rather than having the nodes fanned out.

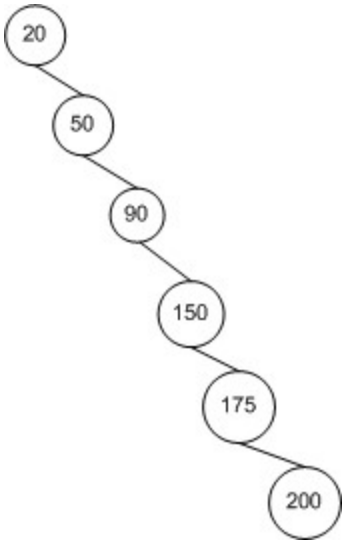

Figure 2. A BST after nodes with values of 20, 50, 90, 150, 175, and 200 have been added

BSTs—which offer sublinear running time for insertions, deletions, and searches—perform optimally when their nodes are arranged in a fanned out manner. This is because when searching for a node in a BST, each single step down the tree reduces the number of nodes that need to be potentially checked by one half. However, when a BST has a topology similar to the one in Figure 2, the running time for the BST's operations are much closer to linear time because each step down the tree only reduces the number of nodes that need to be searched by one. To see why, consider what must happen when searching for a particular value, such as 175. Starting at the root, 20, we must navigate down through each right child until we hit 175. That is, there is no savings in nodes that need to be checked at each step. Searching a BST like the one in Figure 2 is identical to searching an array—each element must be checked one at a time. Therefore, such a structured BST will exhibit a linear search time.

It is important to realize that the running time of a BST's operations is related to the BST's *height*. The height of a tree is defined as the length of the longest path starting at the root. The height of a tree can be defined recursively as follows:

- The height of a node with no children is 0.
- The height of a node with one child is the height of that child plus one.
- The height of a node with two children is one plus the greater height of the two children.

To compute the height of a tree, start at its leaf nodes and assign them a height of 0. Then move up the tree using the three rules outlined to compute the height of each leaf nodes' parent. Continue in this manner until every node of the tree has been labeled. The height of the tree, then, is the height of the root node. Figure 3 shows a number of binary trees with their height computed at each node. For practice, take a second to compute the heights of the trees yourself to make sure your numbers match up with the numbers presented in the figure below.

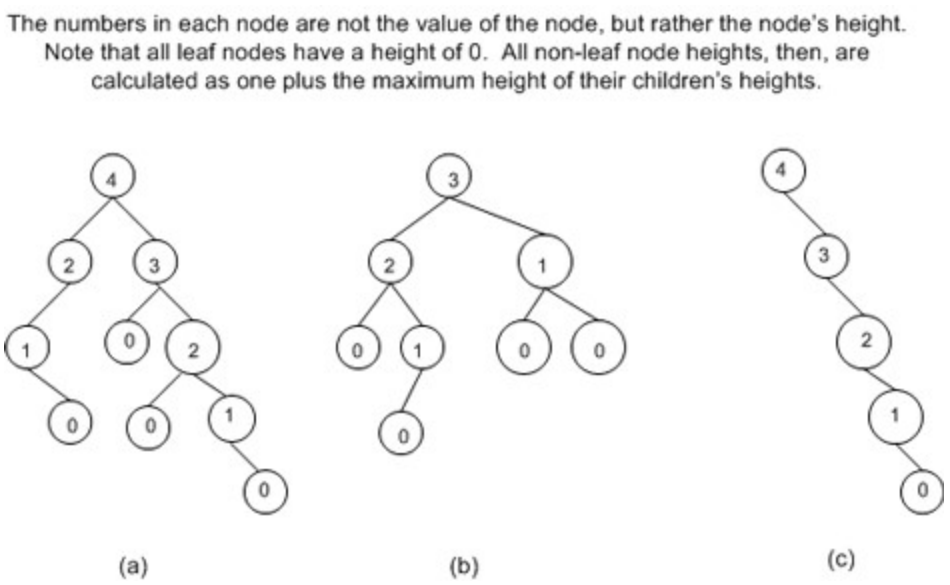

Figure 3. Example binary trees with their height computed at each node

A BST exhibits $\log_2 n$ running times when its height, when defined in terms of the number of nodes, *n*, in the tree, is near the floor of $\log_2 n$. (The floor of a number *x* is the greatest integer less than *x*. So, the floor of 5.38 would be 5 and the floor of 3.14159 would be 3. For positive numbers *x*, the floor of *x* can be found by simply truncating the decimal part of *x*, if any.) Of the three trees in Figure 3, tree (b) has the best height to number of nodes ratio, as the height is 3 and the number of nodes present in the tree is 8. As we discussed in Part 1 of this article series, $\log_a b = y$ is another way of writing $a^y = b$. $\log_2 8$, then, equals 3, because $2^3 = 8$. Tree (a) has 10 nodes and a height of 4. $\log_2 10$ equals 3.3219 and change, the floor of that being 3. So, 4 is not the ideal height. Notice that by rearranging the topology of tree (a)—by moving the far-bottom right node to the child of one of the non-leaf nodes with only one child—we could reduce the tree's height by one, thereby giving the tree an optimal height to node ratio. Finally, tree (c) has the worst height to node ratio. With its 5 nodes it could have an optimal height of 2, but due to its linear topology is has a height of 4.

The challenge we are faced with, then, is ensuring that the topology of the resulting BST exhibits an optimal ratio of height to the number of nodes. Because the topology of a BST is based upon the order in which the nodes are inserted, intuitively you might opt to solve this problem by ensuring that the data that's added to a BST is not added in near-sorted order. While this is possible if you know the data that will be added to the BST beforehand, it might not be practical. If you are not aware of the data that will be added—like if it's added based on user input, or is added as it's read from a sensor—then there is no hope of guaranteeing the data is not inserted in near-sorted order. The solution, then, is not to try to dictate the order with which the data is inserted, but to ensure that after each insertion the BST remains *balanced*. Data structures that are designed to maintain balance are referred to as *self-balancing binary search trees*.

A *balanced tree* is a tree that maintains some predefined ratio between its height and breadth. Different data structures define their own ratios for balance, but all have it close to $\log_2 n$. A self-balancing BST, then, exhibits $\log_2 n$ asymptotic running time. There are numerous self-balancing BST data structures in existence, such as AVL trees, red-black trees, 2-3 trees, 2-3-4 trees, splay trees, B-trees, and others. In the next two sections, we'll take a brief look at two of these self-balancing trees—AVL trees and red-black trees.

Examining AVL Trees

In 1962 Russian mathematicians G. M. **A**ndel'son-**V**el-skii and E. M. **L**andis invented the first self-balancing BST, called an AVL tree. AVL trees must maintain the following balance property—for every node *n*, the height of *n*'s left and right subtrees can differ by at most 1. The height of a node's left or right subtree is the height computed for its left or right node using the technique discussed in the previous section. If a node has only one child, then the height of childless subtree is defined to be -1.

Figure 4 shows, conceptually, the height-relationship each node in an AVL tree must maintain. Figure 5 provides three examples of BSTs. The numbers in the nodes represent the nodes' values; the numbers to the right and left of each node represent the height of the nodes' left and right subtrees. In Figure 5, trees (a) and (b) are valid AVL trees, but trees (c) and (d) are not, since not all nodes adhere to the AVL balance property.

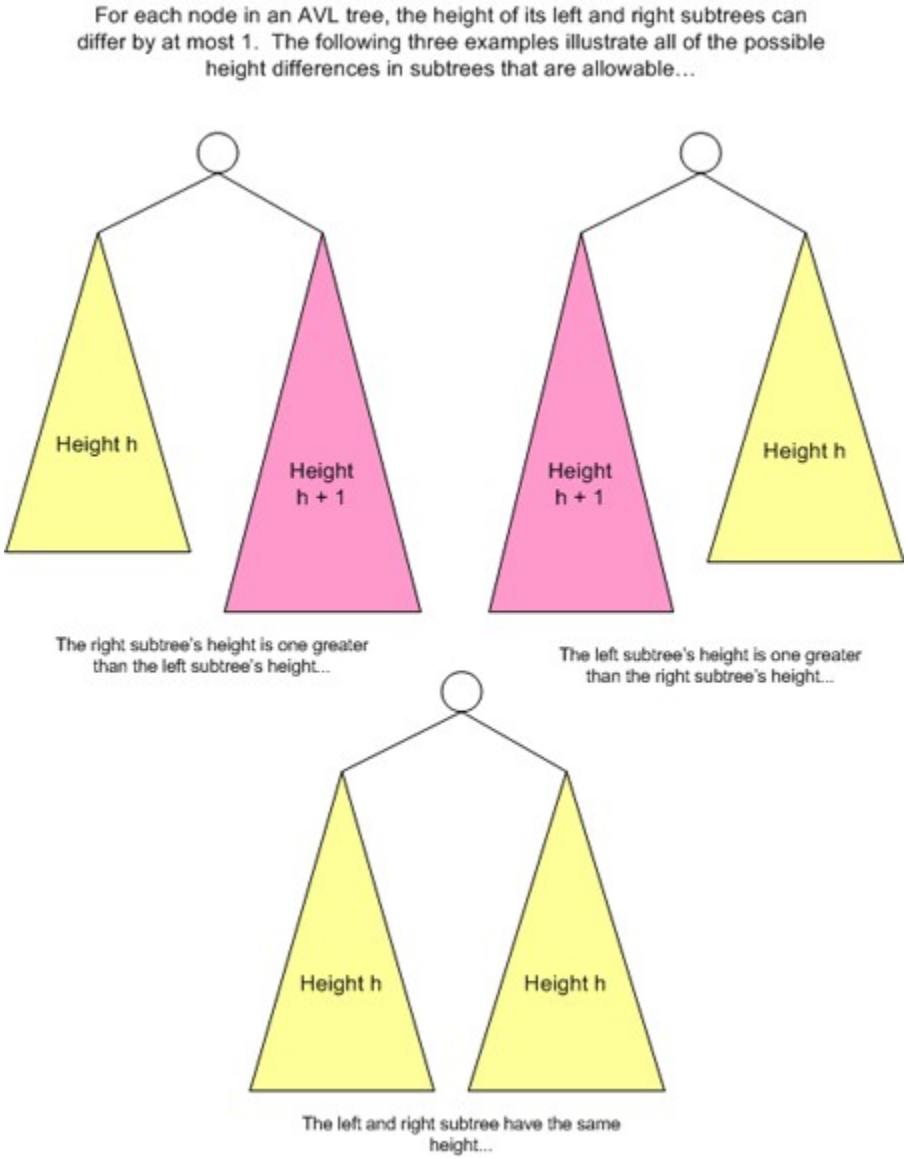

Figure 4. The height of left and right subtrees in an AVL tree cannot differ by more than one.

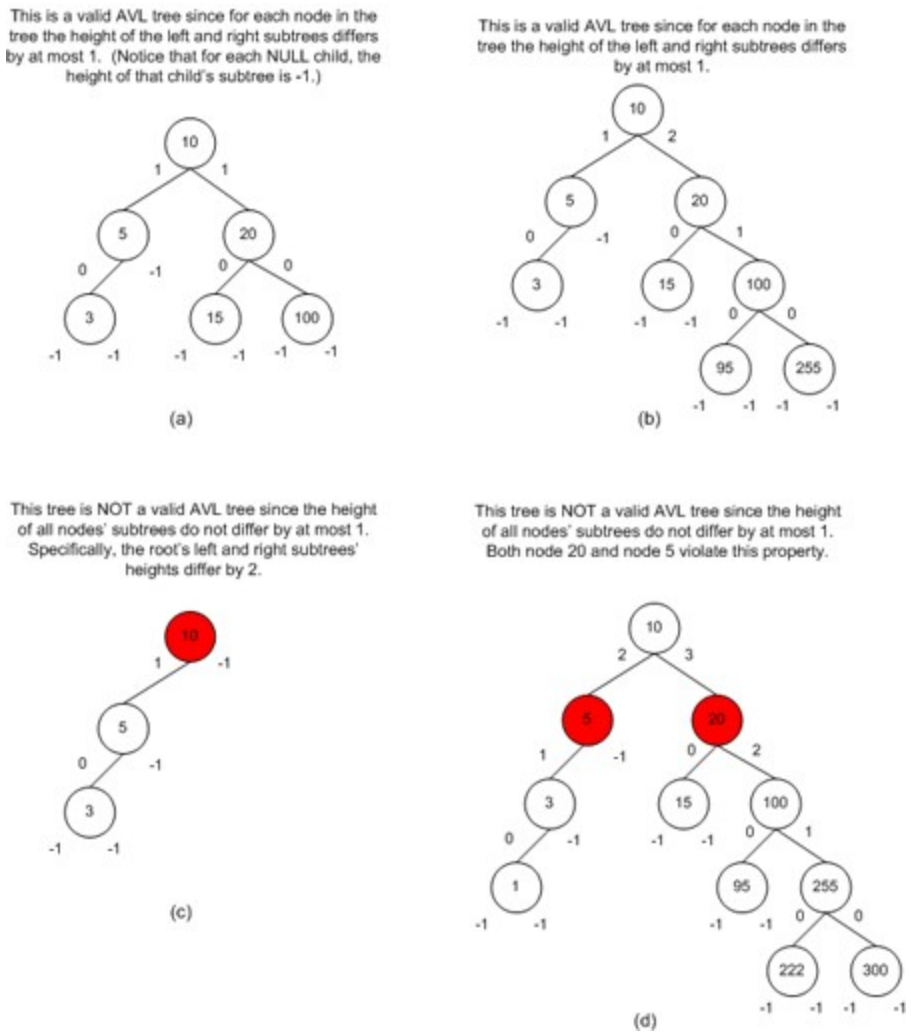

Figure 5. Example trees, where (a) and (b) are valid AVL trees, but (c) and d are not.

> **Note** Realize that AVL trees are binary search trees, so in addition to maintaining a balance property, an AVL tree must also maintain the binary search tree property.

When creating an AVL tree data structure, the challenge is to ensure that the AVL balance remains regardless of the operations performed on the tree. That is, as nodes are added or deleted, it is vital that the balance property remains. AVL trees maintain the balance through *rotations*. A rotation slightly reshapes the tree's topology such that the AVL balance property is restored and, just as importantly, the binary search tree property is maintained.

Inserting a new node into an AVL tree is a two-stage process. First, the node is inserted into the tree using the same algorithm for adding a new node to a BST. That is, the new node is added as a leaf node in the appropriate location to maintain the BST property. After adding a new node, it might be the case that adding this new node caused the AVL balance property to be violated at some node along the path traveled down from the root to where the newly inserted node was added. To fix any violations, stage two involves traversing back up the access path, checking the height of the left and right subtrees for each node along this return path. If the heights of the subtrees differs by more than 1, a rotation is performed to fix the anomaly.

Figure 6 illustrates the steps for a rotation on node 3. Notice that after stage 1 of the insertion routine, the AVL tree property was violated at node 5, because node 5's left subtree's height was two greater than its right subtree's height. To remedy this, a rotation was performed on node 3, the root of node 5's left subtree. This rotation fixed the balance inconsistency and also maintained the BST property.

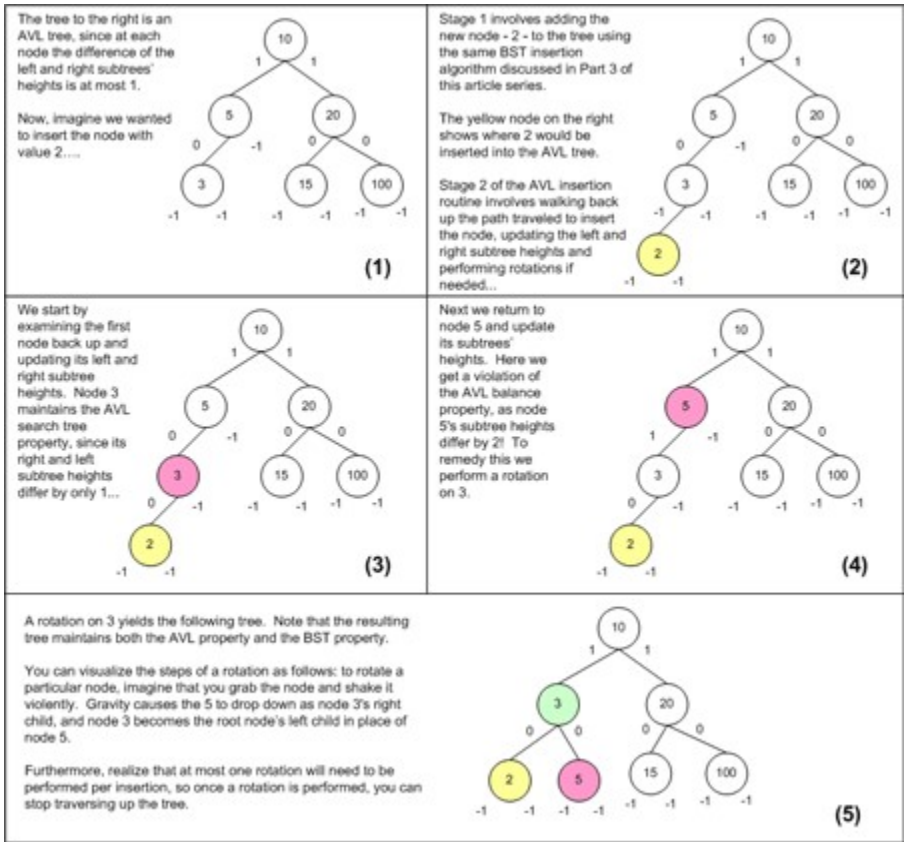

Figure 6. AVL trees stay balanced through rotations

In addition to the simple, single rotation shown in Figure 6, there are more involved rotations than are sometimes required. A thorough discussion of the set of rotations potentially needed by an AVL tree is beyond the scope of this article. What is important to realize is that both insertions and deletions can disturb the balance property to which that AVL trees must adhere. To fix any perturbations, rotations are used.

> **Note** To familiarize yourself with insertions, deletions, and rotations from an AVL tree, check out the AVL tree applet at http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm. This Java applet illustrates how the topology of an AVL tree changes with additions and deletions.

By ensuring that all nodes' subtrees' heights differ by 1 at most, AVL trees guarantee that insertions, deletions, and searches will always have an asymptotic running time of $\log_2 n$, regardless of the order of insertions into the tree.

A Look at Red-Black Trees

The red-black tree data structure was invented in 1972 by Rudolf Bayer, a computer science professor at the Technical University of Munich. In addition to its data and left and right children, the nodes of a red-black tree contain an extra bit of information—a color, which can be either one of two colors, red or black. Red-black trees are complicated further by the concept of a specialized class of node referred to as NIL nodes. NIL nodes are pseudo-nodes that exist as the leaves of the red-black tree. That is, all regular nodes—those with some data associated with them—are internal nodes. Rather than having a NULL pointer for a childless regular node, the node is assumed to have a NIL node in place of that NULL value. This concept can be understandably confusing. Hopefully the diagram in Figure 7 will clear up any confusion.

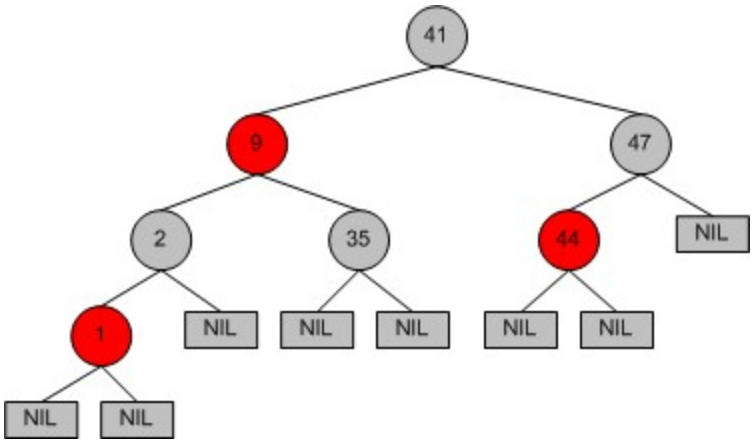

Figure 7. Red-black trees add the concept of a NIL node.

Red-black trees are trees that have the following four properties:

1. Every node is colored either red or black.
2. Every NIL node is black.
3. If a node is red, then both of its children are black.
4. Every path from a node to a descendant leaf contains the same number of black nodes.

The first three properties are pretty self-explanatory. The fourth property, which is the most important of the four, simply states that starting from any node in the tree, the number of black nodes from that node to any leaf (NIL), must be the same. In Figure 7, take the root node as an example. Starting from 41 and going to any NIL, you will encounter the same number of black nodes—3. For example, taking a path from 41 to the left-most NIL node, we start on 41, a black node. We then travel down to node 9, then node 2, which is also black, then node 1, and finally the left-most NIL node. In this journey we encountered three black nodes—41, 2, and the final NIL node. In fact, if we travel from 41 to *any* NIL node, we'll always encounter precisely three black nodes.

Like the AVL tree, red-black trees are another form of self-balancing binary search tree. Whereas the balance property of an AVL tree was explicitly stated as a relationship between the heights of each node's left and right subtrees, red-black trees guarantee their balance in a more conspicuous manner. It can be shown that a tree that implements the four red-black tree properties has a height that is always less than $2 * \log_2(n+1)$, where n is the total number of nodes in the tree. For this reason, red-black trees ensure that all operations can be performed within an asymptotic running time of $\log_2 n$.

Like AVL trees, any time a red-black tree has nodes inserted or deleted, it is important to verify that the red-black tree properties have not been violated. With AVL trees, the balance property was restored using rotations. With red-black trees, the red-black tree properties are restored through recoloring and rotations. Red-black trees are notoriously complex in their recoloring and rotation rules, requiring the nodes along the access path to make decisions based upon their color in contrast to the color of their parents and uncles. (An uncle of a node *n* is the node that is *n*'s parent's sibling node.) A thorough discussion of recoloring and rotation rules is far beyond the scope of this article.

To view the recoloring and rotations of a red-black tree as nodes are added and deleted, check out the red-black tree applet, which can also be accessed at http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm.

A Quick Primer on Linked Lists

One common data structure we've yet to discuss is the *linked list*. Because the skip list data structure we'll be examining next is the mutation of a linked list into a data structure with self-balanced binary tree running times, it is important that before diving into the specifics of skip lists we take a moment to discuss linked lists first.

Recall that with a binary tree, each node in the tree contains some bit of data and a reference to its left and right children. A linked list can be thought of as a unary tree. That is, each element in a linked list has some data associated with it, and a *single* reference to its neighbor. As Figure 8 illustrates, each element in a linked list forms a link in the chain. Each link is tied to its neighboring node, the node on its right.

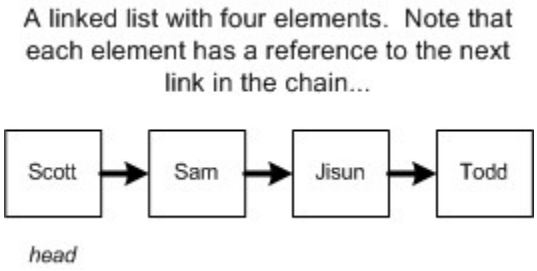

Figure 8. A four-element linked list

When we created a binary tree data structure in Part 3, the binary tree data structure only needed to contain a reference to the root of the tree. The root itself contained references to its children, and those children contained references to their children, and so on. Similarly, with the linked list data structure, when implementing a structure we only need to keep a reference to the *head* of the list because each element in the list maintains a reference to the next item in the list.

Linked lists have the same linear running time for searches as arrays. That is, to find if the element Sam is in the linked list in Figure 8, we have to start at the head and check each element one by one. There are no shortcuts as with binary trees or hashtables. Similarly, deleting from a linked list takes linear time because the linked list must first be searched for the item to be deleted. Once the item is found, removing it from the linked list involves reassigning the deleted item's left neightbor's neighbor reference to the deleted item's neighbor. Figure 9 illustrates the pointer reassignment that must occur when deleting an item from a linked list.

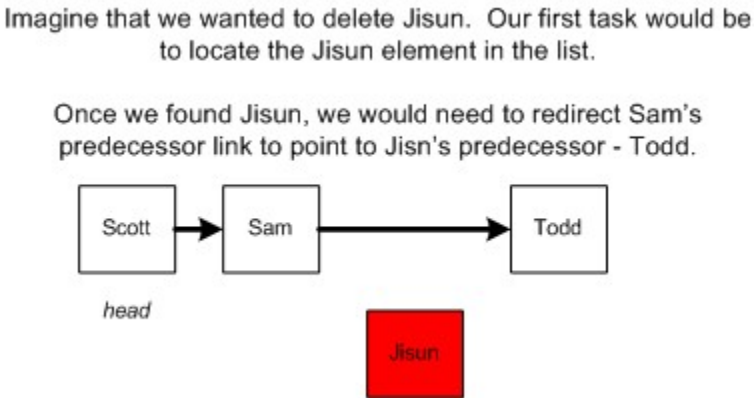

Figure 9. Deleting an element from a linked list

The asymptotic time required to insert a new element into a linked list depends on whether or not the linked list is a sorted list. If the list's elements need not be sorted, insertion can occur in constant time because we can add the

element to the front of the list. This involves creating a new element, having its neighbor reference point to the current linked list head, and, finally, reassigning the linked list's head to the newly inserted element.

If the linked list elements need to be maintained in sorted order, then when adding a new element the first step is to locate where in the list it belongs. This is accomplished by exhaustively iterating from the beginning of the list to the element until the spot where the new element belongs. Let *e* be the element immediately before the location where the new element will be added. To insert the new element *e*'s processor reference must now point to the newly inserted element, and the new element's neighbor reference needs to be assigned to *e*'s old neighbor. Figure 10 illustrates this concept graphically.

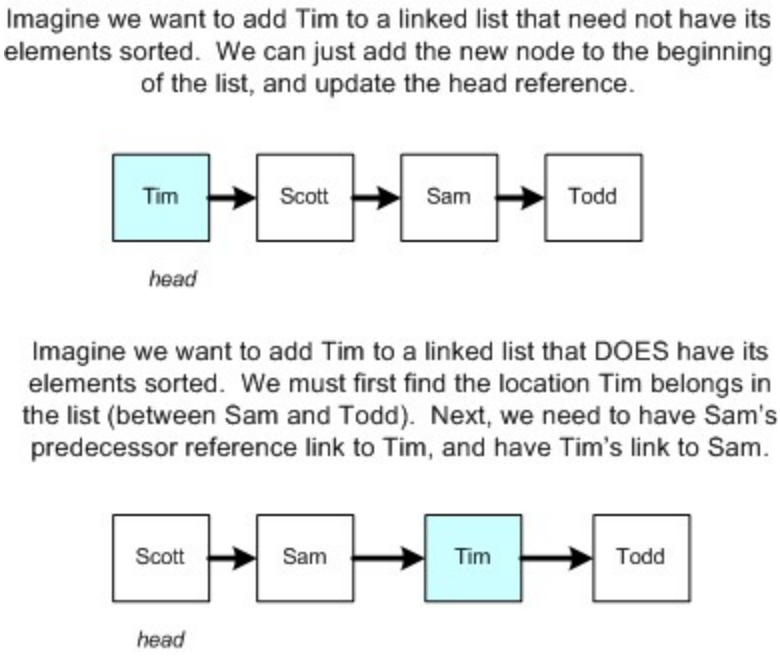

Figure 10. Inserting elements into a sorted linked list

Notice that linked lists do not provide direct access, like an array. That is, if you want to access the i^{th} element of a linked list, you have to start at the front of the list and walk through i links. With an array, though, you can jump straight to the i^{th} element. Given this, along with the fact that linked lists do not offer better search running times than arrays, you might wonder why anyone would want to use a linked list.

The primary benefit of linked lists is that adding or removing items does not involve messy and time-consuming resizing. Recall that array's have fixed size and therefore if an array needs to have more elements added to it than it has capacity, the array must be resized. Granted, the List class hides the code complexity of this, but resizing still carries with it a performance penalty. Furthermore, linked lists are ideal for interactively adding items in sorted order. With an array, inserting an item in the middle of the array requires that all remaining elements be shifted down one spot, and the array resized. In short, an array is usually a better choice if you have an idea on the upper bound of the amount of data that needs to be stored. If you have no conceivable notion as to how many elements will need to be stored, then a link list might be a better choice.

In closing, linked lists are fairly simple to implement. The main challenge comes with the threading or rethreading of the neighbor links with insertions or deletions, but the complexity of adding or removing an element from a linked list pales in comparison to the complexity of balancing an AVL or red-black tree.

With version 2.0 of the .NET Framework, a linked list class has been added to the Base Class Library—`System.Collections.Generic.LinkedList`. This class implements a *doubly-linked list*, which is a linked list whose nodes have a reference to both their next neighbor and their previous neighbor. A `LinkedList` is composed of a variable number of `LinkedListNode` instances, which, in addition to the next and previous references, contains a `Value` property whose type can be specified using Generics.

Skip Lists: A Linked List with Self-Balancing BST-Like Properties

Back in 1989 William Pugh, a computer science professor at the University of Maryland, was looking at sorted linked lists one day thinking about their running time. Clearly a sorted linked list takes linear time to search since potentially each element must be visited, one right after the other. Pugh thought to himself that if half the elements in a sorted linked list had *two* neighbor references—one pointing to its immediate neighbor, and another pointing to the neighbor two elements ahead—while the other half just had one, then searching a sorted linked list could be done in half the time. Figure 11 illustrates a two-reference sorted linked list.

Figure 11. A skip list

The way such a linked list saves searching time is due in part to the fact that the elements are sorted, as well as the varying heights. To search for, say, Dave, we'd start at the *head element*, which is a dummy element whose height is the same height as the maximum element height in the list. The head element does not contain any data, it merely serves as a place to start searching.

We start at the highest link because it lets us skip over lower elements. We begin by following the head element's top link to Bob. At this point we can ask ourselves, does Bob come before or after Dave? If it comes before Dave, then we know Dave, if he's in the list, must exist somewhere to the right of Bob. If Bob comes before Dave, then Bob must exist somewhere between where we're currently positioned and Bob. In this case, Dave comes after Bob alphabetically, so we can repeat our search again from the Bob element. Notice that by moving onto Bob, we are skipping over Alice. At Bob, we repeat the search at the same level. Following the top-most pointer we reach Dave, bypassing Cal. Because we have found what we are looking for, we can stop searching.

Now, imagine that we wanted to search for Cal. We'd begin by starting at the head element, and then moving onto Bob. At Bob, we'd start by following the top-most reference to Dave. Because Dave comes *after* Cal, we know that Cal must exist somewhere between Bob and Dave. Therefore, we move down to the next lower reference level and continue our comparison.

The efficiency of such a linked list arises because we are able to move two elements over every time instead of just one. This makes the running time on the order of $n/2$, which, while better than a regular sorted link list, is still an asymptotically linear running time. Realizing this, Pugh wondered what would happen if rather than limiting the height of an element to 2, it was instead allowed to go up to $\log_2 n$ for n elements. That is, if there were 8 elements in the linked list, there would be elements with height up to 3; if there were 16 elements, there would be elements with height up to 4. As Figure 12 shows, by intelligently choosing the heights of each of the elements, the search time would be reduced to $\log_2 n$.

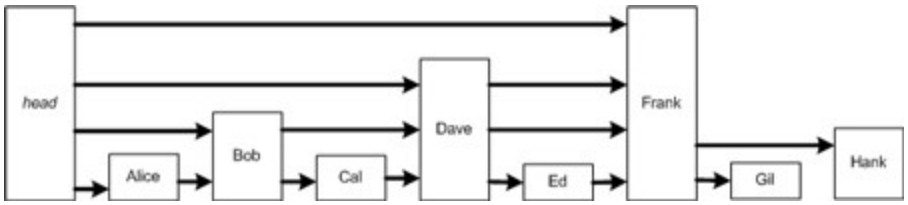

Figure 12. By increasing the height of each node in a skip list, better searching performance can be gained.

Notice that the nodes in Figure 12 every 2^{ith} node has references 2^i elements ahead. That is, the 2^0 element, Alice, has a reference to 2^0 elements ahead—Bob. The 2^{11} element, Bob, has a reference to a node 2^1 elements ahead—Dave. Dave, the 2^2 element, has a reference 2^2 elements ahead—Frank. Had there been more elements, Frank—the 2^3 element—would have a reference to the element 2^3 elements ahead.

The disadvantage of the approach illustrated in Figure 12 is that adding new elements or removing existing ones can wreck havoc on the precise structure. That is, if Dave is deleted, now Ed becomes the 2^2 element, and Gil the 2^3 element, and so on. This means *all* of the elements to the right of the deleted element will need to have their height and references readjusted. The same problem crops up with inserts. This redistribution of heights and references would not only complicate the code for this data structure, but would also reduce the insertion and deletion running times to linear.

Pugh noticed that this pattern created 50% of the elements at height 1, 25% at height 2, 12.5% at height 3, and so on. That is, $1/2^i$ percent of the elements were at height *i*. Rather than trying to ensure the correct heights for each element with respect to its ordinal index in the list, Pugh decided to just randomly pick a height using the ideal distribution—50% at height 1, 25% at height 2, and so on. What Pugh discovered was that such a randomized linked list was not only very easy to create in code, but that it also exhibited $\log_2 n$ running time for insertions, deletions, and lookups. Pugh named his randomized lists *skip lists* since iterating through the list skips over lower-height elements.

In the remaining sections, we'll examine the insertion, deletion, and lookup functions of the skip list, and implement them in a C# class. We'll finish off with an empirical look at the skip list's performance and discuss the tradeoffs between skip lists and self-balancing BSTs.

Creating the SkipListNode and SkipListNodeList Classes

A skip list, like a binary tree, is made up of a collection of elements. Each element in a skip list has some data associated with it, a height, and a collection of element references. For example, in Figure 12 the Bob element has the data Bob, a height of 2, and two element references: one to Dave and one to Cal. Before creating a skip list class, we first need to create a class that represents an element in the skip list. I created a class that extended the base `Node` class that we examined in Part 3 of this article series, and named this extended class `SkipListNode`. The germane code for `SkipListNode` is shown below. (The complete skip list code is available in this article as a code download.) Note that the `SkipListNode` is implemented using Generics, thereby allowing the developer creating a skip list to specify the type of data the skip list will contain at develop-time.

```
public class SkipListNode<T> : Node<T>
{
    private SkipListNode() {}   // no default constructor available, must supply height
    public SkipListNode(int height)
```

```
    {
        base.Neighbors = new SkipListNodeList<T>(height);
    }

    public SkipListNode(T value, int height) : base(value)
    {
        base.Neighbors = new SkipListNodeList<T>(height);
    }

    public int Height
    {
        get { return base.Neighbors.Count; }
    }

    public SkipListNode<T> this[int index]
    {
        get { return (SkipListNode<T>) base.Neighbors[index]; }
        set { base.Neighbors[index] = value; }
    }
}
```

The `SkipListNode` class uses a `SkipListNodeList` class to store its collection of `SkipListNode` references; these `SkipListNode` references are the `SkipListNode`'s neighbors. (You can see that in the constructor the base class's `Neighbors` property is assigned to a new `SkipListNodeList<T>` instance with the specified height.) The `SkipListNodeList` class extends the base NodeList class by adding two methods—`IncrementHeight()` and `DecrementHeight()`. As we'll see in the "Inserting Into a SkipList" section, the height of a SkipList might need to be incremented or decremented, depending on the height of the added or removed item.

```
public class SkipListNodeList<T> : NodeList<T>
{
    public SkipListNodeList(int height) : base(height) { }

    internal void IncrementHeight()
    {
        // add a dummy entry
        base.Items.Add(default(Node<T>));
    }

    internal void DecrementHeight()
    {
        // delete the last entry
        base.Items.RemoveAt(base.Items.Count - 1);
    }
}
```

The `SkipListNodeList` constructor accepts a height input parameter that indicates the number of neighbor references that the node will need. It allocates the specified number of elements in the base class's `Neighbors` property by calling the base class's constructor, passing in the height. The `IncrementHeight()` and `DecrementHeight()` methods add a new node to the collection and remove the top-most node, respectively. In a bit we'll see why these two helper methods are needed.

With the `SkipListNode` and `SkipListNodeList` classes created, we're ready to move on to creating the `SkipList` class. The `SkipList` class, as we'll see, contains a single reference to the head element. It also provides methods for searching the list, enumerating through the list's elements, adding elements to the list, and removing elements from the list.

> **Note** For a graphical view of skip lists in action, be sure to check out the skip list applet at http://iamwww.unibe.ch/~wenger/DA/SkipList/. You can add and remove items from a skip list and visually see how the structure and height of the skip list is altered with each operation.

Creating the SkipList Class

The **SkipList** class provides an abstraction of a skip list. It contains public methods like:

- **Add(value):** adds a new item to the skip list.
- **Remove(value):** removes an existing item from the skip list.
- **Contains(value):** returns true if the item exists in the skip list, false otherwise.

And public properties such as:

- **Height:** the height of the tallest element in the skip list.
- **Count:** the total number of elements in the skip list.

The skeletal structure of the class is shown below. Over the next several sections we'll examine the skip list's operations and fill in the code for its methods.

```
public class SkipList<T> : IEnumerable<T>, ICollection<T>
{
   SkipListNode<T> _head;
   int _count;
   Random _rndNum;
   private IComparer<T> comparer = Comparer<T>.Default;

   protected readonly double _prob = 0.5;

   public int Height
   {
      get { return _head.Height; }
   }

   public int Count
   {
      get { return _count; }
   }

   public SkipList() : this(-1, null) {}
   public SkipList(int randomSeed) : this(randomSeed, null) {}
   public SkipList(IComparer<T> comparer) : this(-1, comparer) {}
   public SkipList(int randomSeed, IComparer<T> comparer)
   {
```

```
        _head = new SkipListNode<T>(1);
        _count = 0;
        if (randomSeed < 0)
            _rndNum = new Random();
        else
            _rndNum = new Random(randomSeed);

        if (comparer != null) this.comparer = comparer;
    }

    protected virtual int ChooseRandomHeight(int maxLevel)
    {
        ...
    }

    public bool Contains(T value)
    {
        ...
    }

    public void Add(T value)
    {
        ...
    }

    public bool Remove(T value)
    {
        ...
    }
}
```

We'll fill in the code for the methods in a bit, but for now pay close attention to the class's private member variables, public properties, and constructors. There are three relevant private member variables:

- `_head`, which is the list's head element. Remember that a skip list has a dummy head element (refer back to Figures 11 and 12 for a graphical depiction of the head element).
- `_count`, an integer value keeping track of how many elements are in the skip list.
- `_rndNum`, an instance of the `Random` class. Because we need to randomly determine the height when adding a new element to the list, we'll use this `Random` instance to generate the random numbers.

The **SkipList** class has two read-only public properties, `Height` and `Count`. `Height` returns the height of the tallest skip list element. Because the head is always equal to the tallest skip list element, we can simply return the head element's `Height` property. The `Count` property simply returns the current value of the private member variable `count`. (`count`, as we'll see, is incremented in the `Add()` method and decremented in the `Remove()` method.)

Notice there are four forms of the `SkipList` constructor, which provide all permutations for specifying a random seed and a custom comparer. The default constructor creates a skip list using the default comparer for the type `T`, allowing the `Random` class to choose the random seed. Regardless of what constructor form is used, a head for the skip list is created as a new `SkipListNode<T>` instance with height 1, and `count` is set equal to 0.

> **Note** Computer random number generators, such as the `Random` class in the .NET Framework, are referred to as pseudo-random number generators because they don't really pick random numbers but instead use a function to generate the random numbers. The random number generating function works by starting with some value, called the seed. Based on the seed, a sequence of random numbers are computed. Slight changes in the seed value lead to seemingly random changes in the series of numbers returned.

If you use the `Random` class's default constructor, the system clock is used to generate a seed. You can optionally specify a specific seed value, however. The benefit of specifying a seed is that if you use the same seed value, you'll get the same sequence of random numbers. Being able to get the same results is beneficial when testing the correctness and efficiency of a randomized algorithm like the skip list.

Searching a Skip List

The algorithm for searching a skip list for a particular value is fairly straightforward. Non-formally, the search process can be described as follows: we start with the head element's top-most reference. Let *e* be the element referenced by the head's top-most reference. We check to see if the *e*'s value is less than, greater than, or equal to the value for which we are searching. If it equals the value, then we have found the item we're looking for. If it's greater than the value we're looking for then if the value exists in the list, it must be to the left of *e*, meaning it must have a lesser height than *e*. Therefore, we move down to the second level head node reference and repeat this process.

If, on the other hand, the value of *e* is less than the value we're looking for then the value, if it exists in the list, must be on the right hand side of *e*. Therefore, we repeat these steps for the top-most reference of *e*. This process continues until we either find the value we're searching for, or exhaust all the "levels" without finding the value.

More formally, the algorithm can be spelled out with the following pseudocode:

```
SkipListNode current = head
for i = skipList.Height downto 1
  while current[i].Value < valueSearchingFor
    current = current[i]  // move to the next node

if current[i].Value == valueSearchingFor then
   return true
else
return false
```

Take a moment to trace the algorithm over the skip list shown in Figure 13. The red arrows show the path of checks when searching the skip lists. Skip list (a) shows the results when searching for Ed; skip list (b) shows the results when searching for Cal; skip list (c) shows the results when searching for Gus, which does not exist in the skip list. Notice that throughout the algorithm we are moving in a right, downward direction. The algorithm never moves to a node to the left of the current node, and never moves to a higher reference level.

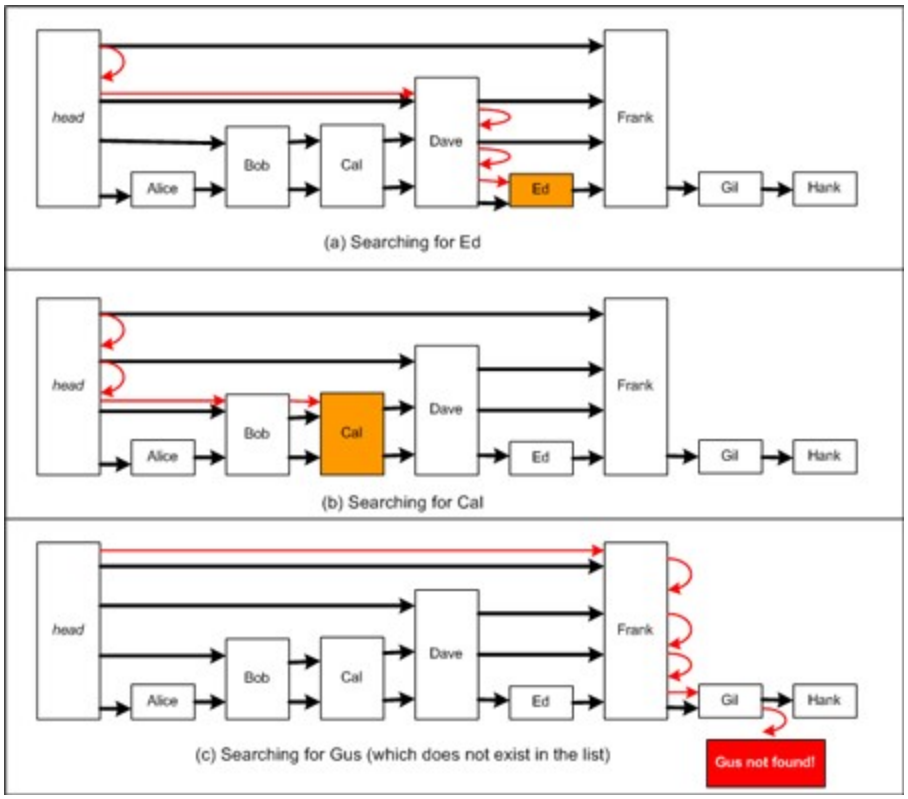

Figure 13. Searching over a skip list.

The code for the `Contains(value)` method is quite simple, involving just a `while` and a `for` loop. The `for` loop iterates down through the reference level layers; the `while` loop iterates across the skip list's elements.

```
public bool Contains(T value)
{
   SkipListNode<T> current = _head;

for (int i = _head.Height - 1; i >= 0; i--)
   {
      while (current[i] != null)
      {
         int results = comparer.Compare(current[i].Value, value);
         if (results == 0)
            return true;   // we found the element
         else if (results < 0)
            current = current[i];   // the element is to the left, so move down a leve;
         else // results > 0
            break;  // exit while loop, because the element is to the right of this node, at
(or lower than) the current level
      }
   }

   // if we reach here, we searched to the end of the list without finding the element
   return false;
}
```

Inserting into a Skip List

Inserting a new element into a skip list is akin to adding a new element in a sorted link list, and involves two steps. First, we must locate where in the skip list the new element belongs. This location is found by using the search algorithm to find the location that comes immediately before the spot the new element will be added. Second, we have to thread the new element into the list by updating the necessary references.

Because skip list elements can has many levels and therefore many references, threading a new element into a skip list is not nearly as simple as threading in a new element into a simple linked list. Figure 14 shows a diagram of a skip list and the threading process that needs to be done to add the element Gus. For this example, imagine that the randomly determined height for the Gus element was 3. To successfully thread in the Gus element, we'd need to update Frank's level 3 and 2 references, as well as Gil's level 1 reference. Gus's level 1 reference would point to Hank. If there were additional nodes to the right of Hank, Gus's level 2 reference would point to the first element to the right of Hank with height 2 or greater, while Gus's level 3 reference would point to the first element right of Hank with height 3 or greater.

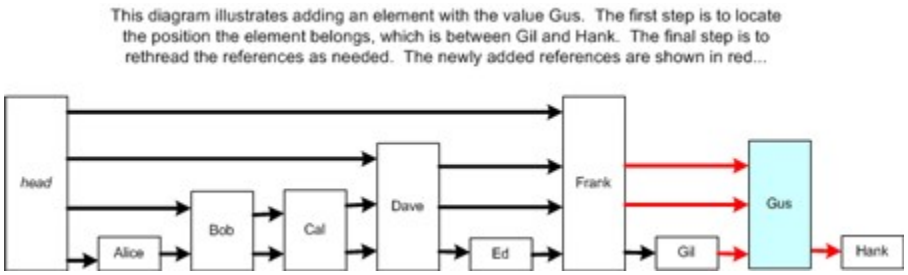

Figure 14. Inserting elements into a skip list

In order to properly rethread the skip list after inserting the new element, we need to keep track of the last element encountered for each height. In Figure 14, Frank was the last element encountered for references at levels 4, 3, and 2, while Gil was the last element encountered for reference level 1. In the insert algorithm below, this record of last elements for each level is maintained by the `updates` array. This array, which is populated as the search for the location for the new element is performed, is created in the `BuildUpdateTable()` method (also shown below).

> **Note** Given a skip list of height h, in order to thread in a new node, h references will have to be updated in existing skip list nodes. The `updates` array maintains references to the h nodes in the skip list whose references will need to be updated. That is, `updates[i]` contains the node whose reference at level i will need to be rethreaded to point to the newly inserted node (assuming the newly inserted node's height is greater than or equal to i.)

```
public void Add(T value)
{
   SkipListNode<T>[] updates = BuildUpdateTable(value);
   SkipListNode<T> current = updates[0];

   // see if a duplicate is being inserted
   if (current[0] != null && current[0].Value.CompareTo(value) == 0)
      // cannot enter a duplicate, handle this case by either just returning or by throwing a
n exception
      return;

   // create a new node
   SkipListNode<T> n = new SkipListNode<T>(value, ChooseRandomHeight(head.Height + 1));
```

```
   _count++;   // increment the count of elements in the skip list

   // if the node's level is greater than the head's level, increase the head's level
   if (n.Height > _head.Height)
   {
      _head.IncrementHeight();
      _head[_head.Height - 1] = n;
   }

   // splice the new node into the list
   for (int i = 0; i < n.Height; i++)
   {
      if (i < updates.Length)
      {
         n[i] = updates[i][i];
         updates[i][i] = n;
      }
   }
}

protected SkipListNode<T>[] BuildUpdateTable(T value)
{
    SkipListNode<T>[] updates = new SkipListNode<T>[_head.Height];
    SkipListNode<T> current = _head;

    // determine the nodes that need to be updated at each level
    for (int i = _head.Height - 1; i >= 0; i--)
    {
        while (current[i] != null && comparer.Compare(current[i].Value, value) < 0)
            current = current[i];

        updates[i] = current;
    }

    return updates;
}
```

There are a couple of key portions of the `Add(value)` and `BuildUpdateTable()` methods to pay close attention to. First, in the `BuildUpdateTable()` method, be certain to examine the `for` loop. In this loop, the `updates` array is fully populated. The `SkipListNode` immediately preceding where the new node will be inserted is represented by `updates[0]`.

After `BuildUpdateTable()`, a check is done to make sure that the data being entered is not a duplicate. I chose to implement my skip list such that duplicates are not allowed; however, skip lists can handle duplicate values just fine. If you want to allow for duplicates, simply remove this check.

Next, a new `SkipListNode` instance, *n*, is created. This represents the element to be added to the skip list. Note that the height of the newly created `SkipListNode` is determined by a call to the `ChooseRandomHeight()` method, passing in the current skip list height plus one. We'll examine this method shortly. Another thing to note is that after adding the `SkipListNode`, a check is made to see if the new `SkipListNode`'s height is greater than that

of the skip list's head element's height. If it is, then the head element's height needs to be incremented, since the head element height should have the same height as the tallest element in the skip list.

The final `for` loop rethreads the references. It does this by iterating through the `updates` array, having the newly inserted `SkipListNode`'s references point to the `SkipListNode`s previously pointed to by the `SkipListNode` in the `updates` array, and then having the `updates` array `SkipListNode` update its reference to the newly inserted `SkipListNode`. To help clarify things, try running through the `Add(value)` method code using the skip list in Figure 14, where the added `SkipListNode`'s height happens to be 3.

Randomly Determining the Newly Inserted SkipListNode's Height

When inserting a new element into the skip list, we need to randomly select a height for the newly added `SkipListNode`. Recall from our earlier discussions of skip lists that when Pugh first envisioned multi-level linked list elements, he imagined a linked list where each 2^{ith} element had a reference to an element 2^{i} elements away. In such a list, precisely 50% of the nodes would have height 1, 25% with height 2, and so on.

The `ChooseRandomHeight()` method uses a simple technique to compute heights so that the distribution of values matches Pugh's initial vision. This distribution can be achieved by flipping a coin and setting the height to one greater than however many heads in a row were achieved. That is, if upon the first flip you get a tails, then the height of the new element will be one. If you get one heads and then a tails, the height will be 2. Two heads followed by a tails indicates a height of three, and so on. Because there is a 50% probability that you will get a tails, a 25% probability that you will get a heads and then a tails, a 12.5% probability that you will get two heads and then a tails, and so on, the distribution works out to be the same as the desired distribution.

The code to compute the random height is given by the following simple code snippet:

```
const double _prob = 0.5;
protected virtual int ChooseRandomHeight()
{
   int level = 1;
   while (_rndNum.NextDouble() < _prob)
      level++;

   return level;
}
```

One concern with the above method is that the value returned might be extraordinarily large. That is, imagine that we have a skip list with, say, two elements, both with height 1. When adding our third element, we randomly choose the height to be 10. This is an unlikely event because there is only roughly a 0.1% chance of selecting such a height, but it could conceivable happen. The downside of this, now, is that our skip list has an element with height 10, meaning there will be a number of superfluous levels in our skip list. To put it more bluntly, the references at levels 2 up to 10 would be unutilized. Even as additional elements were added to the list, there's still only a 3% chance of getting a node over a height of 5, so we'd likely have many wasted levels.

Pugh suggests a couple of solutions to this problem. One is to simply ignore it. Having superfluous levels doesn't require any change in the code of the data structure, nor does it affect the asymptotic running time. Another solution proposed by Pugh calls for using "fixed dice" when choosing the random level, which is the approach I chose to use for the `SkipList` class. With the "fixed dice" approach, you restrict the height of the new element to be a height of at most one greater than the tallest element currently in the skip list. The actual implementation of the `ChooseRandomHeight()` method is shown below, which implements this "fixed dice" approach. Notice that a

`maxLevel` input parameter is passed in, and the `while` loop exits prematurely if level reaches this maximum. In the `Add(value)` method, note that the `maxLevel` value passed in is the height of the head element plus one. (Recall that the head element's height is the same as the height of the maximum element in the skip list.)

```
protected virtual int ChooseRandomHeight(int maxLevel)
{
   int level = 1;
   while (_rndNum.NextDouble() < _prob && level < maxLevel)
      level++;

   return level;
}
```

Because the head element should be the same height as the tallest element in the skip list, in the `Add(value)` method, if the newly added `SkipListNode`'s height is greater than the head element's height, I call the `IncrementHeight()` method:

```
/* - snippet from the Add() method... */
if (n.Height > _head.Height)
{
   _head.IncrementHeight();
   _head[_head.Height - 1] = n;
}
/*********************************/
```

The `IncrementHeight()` method simply adds a new `SkipListNode` to the head element. Because the height of the head node must be the height of the tallest element in the skip list, if we add an element taller than the head, we want to increment the head's height, which is what the code in the `if` statement above accomplishes.

> **Note** In his paper, "Skip Lists: A Probabilistic Alternative to Balanced Trees," Pugh examines the effects of changing the value of `_prob` from 0.5 to other values, such as 0.25, 0.125, and others. Lower values of `_prob` decrease the average number of references per element, but increase the likelihood of the search taking substantially longer than expected. For more details, be sure to read Pugh's paper, which is mentioned in the References section at the end of this article.

Deleting an Element from a Skip List

Like adding an element to a skip list, removing an element involves a two-step process. First, the element to be deleted must be found. After that, the element needs to be snipped from the list and the references need to be rethreaded. Figure 15 shows the rethreading that must occur when Dave is removed from the skip list.

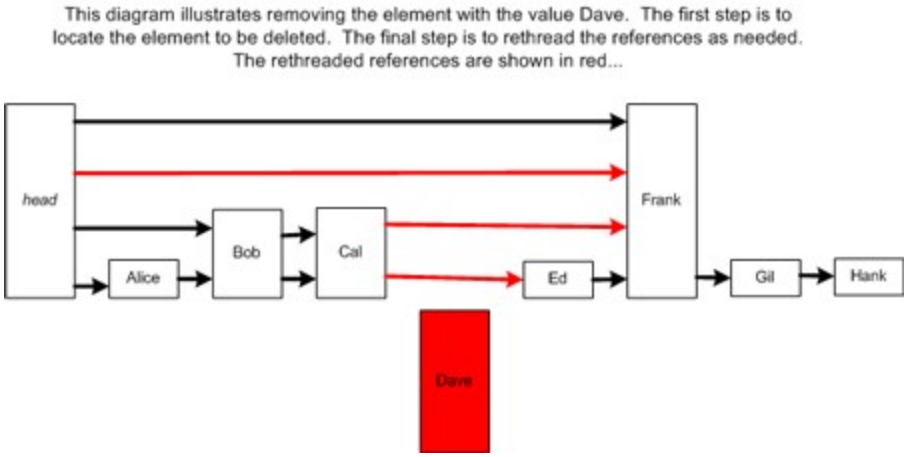

Figure 15. Deleting an element from a skip list

Like with the `Add(value)` method, `Remove(value)` maintains an `updates` array that keeps track of the elements at each level that appear immediately before the element to be deleted (as with `Add()`, this `updates` array is populated with a call to `BuildUpdateTable()`.). Once this updates array has been populated, the array is iterated through from the bottom up, and the elements in the array are rethreaded to point to the deleted element's references at the corresponding levels. The `Remove(value)` method code follows.

```
public virtual void Remove(IComparable value)
{
   SkipListNode<T>[] updates = BuildUpdateTable(value);
   SkipListNode<T> current = updates[0][0];

   if (current != null && comparer.Compare(current.Value, value) == 0)
   {
      _count--;

      // We found the data to delete
      for (int i = 0; i < _head.Height; i++)
      {
         if (updates[i][i] != current)
            break;
         else
            updates[i][i] = current[i];
      }

      // finally, see if we need to trim the height of the list
      if (_head[_head.Height - 1] == null)
         // we removed the single, tallest item... reduce the list height
         _head.DecrementHeight();

      return true;   // item removed, return true
   }
   else
      // the data to delete wasn't found – return false            return false;
}
```

The `Remove(value)` method starts like the `Add(value)` method, by populating the `updates` array with a call to `BuildUpdateTable()`. With the `updates` array populated, we next check to ensure that the element reached does indeed contain the value to be deleted. If not, the element to be deleted was not found in the skip list, so the `Remove()` method returns `false`. Assuming the element reached is the element to be deleted, the `_count` member variable is decremented and the references are rethreaded. Lastly, if we deleted the element with the greatest height, then we should decrement the height of the head element. This is accomplished via a call to the `DecrementHeight()` method of the `SkipListNode` class.

Analyzing the Skip List's Running Time

In "Skip Lists: A Probabilistic Alternative to Balanced Trees," Pugh provides a quick proof showing that the skip list's search, insertion, and deletion running times are asymptotically bounded by $\log_2 n$ in the average case. However, a skip list can exhibit linear time in the worst case, but the likelihood of the worst case happening is very, very, very, very slim.

Because the heights of the elements of a skip list are randomly chosen, there is a chance that all, or virtually all, elements in the skip list will end up with the same height. For example, imagine that we had a skip list with 100 elements, all that happened to have height 1 chosen for their randomly selected height. Such a skip list would be, essentially, a normal linked list, not unlike the one shown in Figure 8. As we discussed earlier, the running time for operations on a normal linked list is linear.

While such worst-case scenarios are possible, realize that they are highly improbable. To put things in perspective, the likelihood of having a skip list with 100 height 1 elements is the same likelihood of flipping a coin 100 times and having it come up tails all 100 times. The chances of this happening are precisely 1 in 1,267,650,600,228,229,401,496,703,205,376. Of course with more elements, the probability goes down even further. For more information be sure to read about Pugh's probabilistic analysis of skip lists in his paper.

Examining Some Empirical Results

Included in the article's download is the `SkipList` class along with a testing Windows Forms application. With this testing application, you can manually add, remove, and inspect the list, and can see the nodes of the list displayed. Also, this testing application includes a "stress tester," where you can indicate how many operations to perform and an optional random seed value. The stress tester then creates a skip list, adds at least half as many elements as operations requested, and then, with the remaining operations, does a mix of inserts, deletes, and queries. At the end you can see review a log of the operations performed and their result, along with the skip list height, the number of comparisons needed for the operation, and the number of elements in the list.

The graph in Figure 16 shows the average number of comparisons per operation for increasing skip list sizes. Note that as the skip list doubles in size, the average number of comparisons needed per operation only increases by a small amount (one or two more comparisons). To fully understand the utility of logarithmic growth, consider how the time for searching an array would fare on this graph. For a 256 element array, on average 128 comparisons would be needed to find an element. For a 512 element array, on average 256 comparisons would be needed. Compare that to the skip list, which for skip lists with 256 and 512 elements require only 9 and 10 comparisons on average.

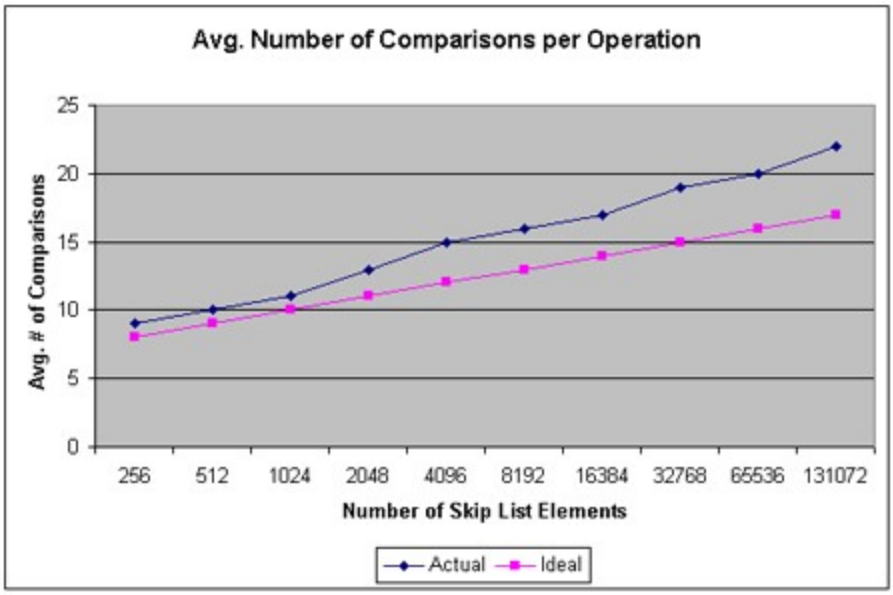

Figure 16. Viewing the logarithmic growth of comparisons required for an increasing number of skip list elements.

Conclusion

In Part 3 of this article series we looked at binary trees and binary search trees. BSTs provide an efficient $\log_2 n$ running time in the average case. However, the running time is sensitive to the topology of the tree, and a tree with an suboptimal ratio of breadth to height can reduce the running time of a BST's operations to linear time.

To remedy this worst-case running time of BSTs, which could happen quite easily since the topology of a BST is directly dependent on the order with which items are added, computer scientists have been inventing a myriad of self-balancing BSTs, starting with the AVL tree created in the 1960s. While data structures such as the AVL tree, the red-black tree, and numerous other specialized BSTs offer $\log_2 n$ running time in both the average and worst case, they require especially complex code that can be difficult to correctly create.

An alternative data structure that offers the same asymptotic running time as a self-balanced BST, is William Pugh's skip list. The skip list is a specialized, sorted link list, one whose elements have a height associated with them. In this article, we constructed a `SkipList` class and saw just how straightforward the skip list's operations were, and how easy it was to implement them in code.

This fourth part of the article series completes our discussion of trees. In the fifth installment, we'll look at *graphs*. A graph is a collection of vertexes with an arbitrary number of edges connecting each vertex to one another. As we'll see in Part 5, trees are a special form of graphs. Graphs have an extraordinary number of applications in real-world problems.

Happy Programming!

References

- Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. "Introduction to Algorithms." MIT Press. 1990.
- Pugh, William. "Skip Lists: A Probabilistic Alternative to Balanced Trees." Available online at ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.pdf

Scott Mitchell, author of six books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since January 1998. Scott works as an independent consultant, trainer, and writer, and holds a Masters degree in Computer Science from the University of California – San Diego. He can be reached at mitchell@4guysfromrolla.com, or via his blog at http://ScottOnWriting.NET.