Trees, Part 1: Unbalanced Trees

The first part of this chapter takes a look at trees in general and unbalanced binary trees. The second part looks at various schemes to balance trees and/or make them more efficient as search structures.

1 Tree Definitions

If you already know what a binary tree is, but not a general tree, then pay close attention, because binary trees are not a special case of general trees with degree two. There are various ways of defining trees; this one is consistent with the one defined in *Data Structures and Algorithms*, by Mark Allen Weiss.

Definition 1. A ***tree*** T consists of a possible empty set of nodes. If T is not empty, it consists of a distinguished node r called the ***root*** of T and zero or more *non-empty subtrees* $T_1, T_2, \ldots, T_k$ such that there is a directed edge from r to each of the roots of $T_1, T_2, \ldots, T_k$.

Note. The preceding definition allows a tree to be empty, but it does not allow it to have empty subtrees. It would make no sense for it to allow empty subtrees, because if it did, then the answer to the question, "how many subtrees does T have" would be undefined. It could have any number of empty subtrees.

Definition 2. A ***forest*** is a collection of non-empty trees.

Note. You can always create a tree from a forest by creating a new root node and making it the parent of the roots of all of the trees in the forest. Conversely, if you lop off the root of a tree, what is left is a forest.

These notes assume that you are familiar with the terminology of binary trees, e.g., parents, children, siblings, ancestors, descendants, grandparents, leaf nodes, internal nodes, external nodes, and so on, so their definitions are not repeated here. Because the definitions of height and depth may vary from one book to another, their definitions are included here, using the ones from the textbook.

Definition 3. A ***path*** from node n_1 to node n_k is a sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_{i+1} for $1 \leq i < k$. The ***length*** of a path is the number of edges in the path, not the number of nodes[1].

Because the edges in a tree are directed, all paths are "downward", i.e., towards leaves and away from the root. The ***height of a node*** is the length of the longest path from the node to any of its descendants. Naturally the longest path must be to a leaf node. The ***depth of a node*** is the length of the path from the root to the node. The root has depth 0. All leaf nodes have height 0.

The ***height of a tree*** is the height of its root. The ***degree of a node*** is the number of children of the node. The ***degree of a tree*** is the maximum degree of the degrees of its nodes. The tree in Figure 1 has height 3 and degree 6. The children of each node happen to be drawn in sorted order from left to right.

[1] Some authors define the length of a path to be the number of nodes, which will always be one greater than the number of edges.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

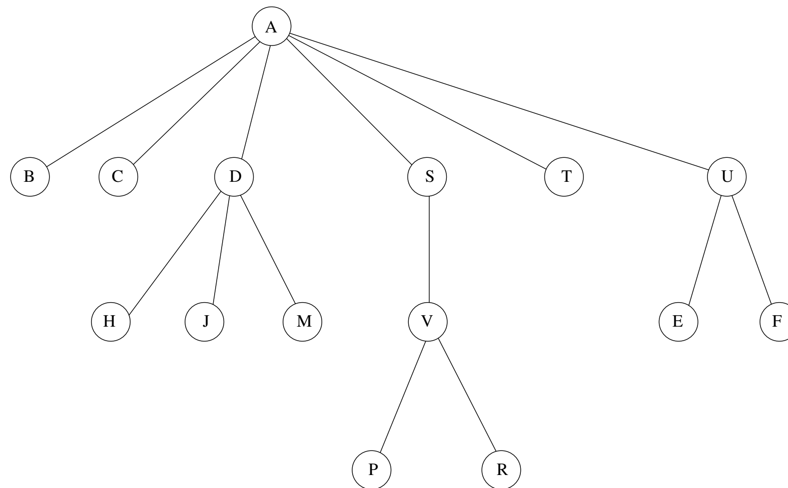

Figure 1: A general tree.

Problem 4. It is not hard to see that a tree with N nodes must have $N - 1$ edges because every node except the root has exactly one incoming edge. How many edges are in a forest with N nodes and K trees?

2 Applications of General Trees

A general tree is useful for representing hierarchies in which the number of children varies. There are many applications for which this is true.

In a modern file system, for example, a node represents a file, and if the file is a directory, then it is an internal node whose children are the files contained in the directory. Some file systems do not restrict the number of files per folder, implying that the number of children per node is varying and unbounded. Some file systems also allow hard links (as UNIX does) and the directory hierarchy is not in fact a tree because there can be edges from a node to one of its ancestors. In this case the structure is called a ***cyclic directed graph***.

In the version control software `git`, the state of the source code is represented by a general tree. There are two kinds of nodes, ordinary nodes, called ***blobs***, and ***tree*** nodes. Tree nodes can have any number of children.

In computational linguistics, as sentences are parsed, the parser creates a representation of the sentence as a tree whose nodes represent grammatical elements such as predicates, subjects, prepositional phrases, and so on. Some elements such as subject elements are always internal nodes because they are made up of simpler elements such as nouns and articles. Others are always leaf nodes, such as nouns. The number of children of the internal nodes is unbounded and varying.

In genealogical software, the tree of descendants of a given person is a general tree because the number of children of a given person is not fixed.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

3 Tree Implementations

Because one does not know the maximum degree that a tree may have, and because it is inefficient to create a structure with a very large fixed number of child entries, the most extensible implementation of a general tree uses a linked list to store the children of a node. It is not exactly what you might imagine immediately; it is a bit more subtle. A tree node contains an element and two pointers: one to the leftmost-child of the node and another to the sibling that is to the immediate right of a node:

```
struct TreeNode
{
    Object element;
    TreeNode * firstChild;
    TreeNode * nextSibling;
};
```

Figure 2 illustrates how this structure is used to represent the tree in Figure 1. The advantage of this representation is that this same node can be used to represent all nodes in the tree.

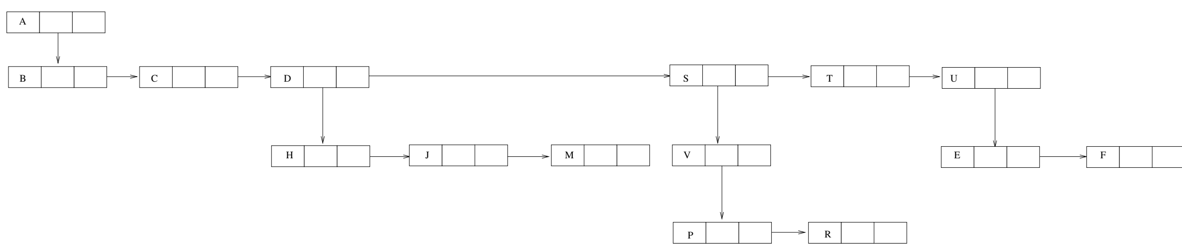

Figure 2: Implementation of tree from Figure 1.

3.1 General Tree Traversal

There are a few different ways to traverse a tree, some more efficient than others depending on the tree implementation. If a tree represents a directory hierarchy, then a ***pre-order traversal*** could be used to print the files and folders in the hierarchy in a natural way, much the way one sees them listed in a file browser window. The following pseudo-code description of such a function does this. It is written as if it were a member function of some class that represents a file (which also includes directories, which are really files.) Assume that `printname(int depth)` is a function that prints the name of the current file indented in proportion to its integer parameter, `depth`.

```
1 void file::listAll( int depth = 0) const
2 {
3     printname(depth);
4     if (isDirectory() )
5         foreach child c in this directory
6             c.listAll(depth + 1);
7 }
```

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

The algorithm is a pre-order traversal because it visits the root of every subtree (which is a directory) prior to visiting any of the children. The pseudo-code of the algorithm does not specify the order that the foreach loop uses to visit all of the children in a directory, but for the sake of precision, let us assume that they are visited in a "left-to-right" order. Bear in mind that a general tree has no notion of left and right. The easiest implementation will just descend the `firstChild` pointer of the directory node and then travel along the `nextSibling` pointers until it reaches a node that has no `nextSibling` (i.e., its `nextSibling` pointer is null.) If `printname()` prints a word with `depth` many tab characters preceding it, then this will print an indented listing of the directory tree, with files at depth d having d tabs to their left. For the tree in Figure 1, the output of this algorithm would be

```
A
    B
    C
    D
        H
        J
        M
    S
        V
            P
            R
    T
    U
        E
        F
```

Notice that the children are listed in dictionary order, because the children of each node were stored that way in the tree structure implementation.

There is no single notion of in-order traversal because of the fact that the number of subtrees varies from one node to the next and the root may be visited in many positions relative to its children. However, one can define ***post-order traversals*** of general trees. One use of post-order is in computing disk block usage for each directory. For example, the UNIX `du` command will display the amount of disk space used by every file, and cumulatively, for every directory in its command-line argument. In order to do this, it must obtain the usage of the child nodes before the parent node. The general algorithm would be of the form

```
1  int file::disk_usage() const
2  {
3
4      int size = usage(); // some function that counts disk blocks
           of the current file
5
6      if (isDirectory() )
7          foreach child c in this directory
8              size = size + c.disk_usage();
9      return size;
10 }
```

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

The `usage()` function is a member function that returns the number of disk blocks used by the current file. Line 8 contains a recursive call of the `disk_usage()` function on child `c`. There will be no infinite recursion if the directory structure has no links back to ancestors.

4 Binary Trees

A binary tree is not a general tree because binary trees distinguish between the left and right subtrees. A binary tree is either empty or it has a root node and a left and right subtree, each of which are binary trees, and whose roots are children of the root of the tree. Structurally, the following typifies the definition of a binary tree node:

```
typedef binary_tree_node* node_ptr;
struct binary_tree_node
{
    Object   element;
    node_ptr left;
    node_ptr right;
};
```

Notice that its structure looks like that of a node in a doubly-linked list - it has a data item and two pointers. The difference of course is that these pointers point to children of the node, not its left and right neighbors!

The most important applications of binary trees are in compiler design and in search structures. The ***binary search tree*** is an important data structure. These notes assume that you are familiar with them from a previous course. The objective here is to look at how they can be implemented using C++ and how we can overcome the inefficiencies of unbalanced binary trees.

5 Binary Search Trees

A binary search tree (BST) is a binary tree that stores comparable elements in such a way that insertions, deletions, and find operations never require more than $O(h)$ operations, where h is the height of the tree. To be clear, a BST is defined as follows.

Definition 5. A binary relation $\leq$ on a set S is a ***total ordering*** of S if, for any elements $a, b, c \in S$

- $a \leq a$ (reflexivity)
- if $a \leq b$ and $b \leq a$ then $a = b$ (antisymmetry)
- if $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity)
- either $a \leq b$ or $b \leq a$ (completeness)

Definition 6. $(S, \leq)$ is a ***totally ordered set*** if S is a set and $\leq$ is a total ordering on S.

Definition 7. If x is a node, $element(x)$ is the element contained in node x.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

Definition 8. A binary search tree for the totally ordered set $(S, \leq)$ is a binary tree in which

- for each node x, $element(x) \in S$, and
- for every node z in the left subtree of x, element(z) $\leq$ element(x), and for every node z in the right subtree of x,$element(x) \leq element(z)$
- for each element $a \in S$, there exists a unique node x such that $element(x) = a$.

It is important to realize that binary search trees use an implicit ordering relation to organize their data, and that a comparison operation must always be defined for this data.

A binary search tree must support insertion, deletion, find, a test for emptiness, and a find-minimum operation. It should also support a list-all operation that lists all elements in sorted order. Since a binary search tree is a container (an object that contains other objects), it needs to provide methods to create an empty binary search tree, and to make a new tree as a copy of an existing tree, as well as methods to destroy instances of trees. Following is a partial interface for a BST template class, containing the essential methods. Note that this code has a nested class definition: the `bst_node` struct definition is contained within the BST class definition. Earlier versions of C++ did not support this. Because the struct definition is contained in the BST definition, the references to `bst_node` do not need to be written using the template parameter in implementations of the BST member functions.

```
template <typename Comparable>
class BST
{
public:
    BST  ( );                      // default
    BST  ( const BST & tree);      // copy constructor
    ~BST ( );                      // destructor

    // Search methods:
    const Comparable& find    ( const Comparable& x) const;
    const Comparable& findMin () const;
    const Comparable& findMax () const;

    // Displaying the tree contents:
    void   print   ( ostream& out ) const;

    // Tree modifiers:
    void   clear();                             // empty the tree
    void   insert( const Comparable& x);   // insert element x
    void   remove( const Comparable& x);   // remove element x

private:
    // The node definition is inaccessible to clients of the tree
         class
    struct bst_node
    {
        Comparable  element;
        bst_node*    left;
```

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

```
        bst_node*    right;

        // node constructor:
        bst_node( const Comparable& item, bst_node* lt, bst_node*
            rt):
                element(item), left(lt), right(rt) {}
    };
    // The pointer to the root of tree is the only data item
    bst_node *root;

    // Recursive methods called by public methods:
    void      insert     ( const Comparable& x, bst_node* & t );
    void      remove     ( const Comparable& x, bst_node* & t );
    bst_node* find       ( const Comparable& x, bst_node* t )
       const;
    bst_node* findMin    ( bst_node* t) const;
    bst_node* findMax    ( bst_node* t) const;
    void      make_empty ( bst_node* t );
    void      print      ( ostream& out,  bst_node* t) const;
    bst_node* copy       (bst_node* t) const;
};
```

5.1 Algorithms

Most tree algorithms are expressed easily using recursion. On the other hand, to do so requires that the parameter of the algorithm is a pointer to the current node, as in this pseudo-code recursive function:

```
void some_tree_function ( bst_node* p )
{
    if ( null != p )
        some_tree_function ( p->left )
}
```

However, a function like this should not be in the public part of a class definition because one does not want to expose the node pointers to an object's clients. Nodes should be hidden from the object's clients. Put another way, the clients should not know nor should they care about how the data is structured inside the tree; in fact, they should not even know that it is a tree! The binary search tree should be a black box with hooks to the methods it makes available to its clients, and no more.

While this may seem like a conundrum, it is not. An elegant solution is to create a "wrapper" public method that the client calls that wraps a call to a recursive function. This is exactly how many of the methods are implemented. Below are a few implementations. The rest are similar, but you should not skip them; you have to make sure you understand how this class implementation works!

5.1.1 The `find` Algorithm

The `find` algorithm recursively descends the tree's pointers. In short, if the current node is empty, it returns an indication that the key being sought is not in the tree, otherwise if the key is smaller

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

than the current node's element, it descends and searches in the left subtree, or if it is larger, it searches in the right subtree. Otherwise the key is equal to the element in the current node and has been found. The recursive implementation is thus

```
template <class Comparable>
bst_node* BST<Comparable>::find ( const Comparable& x, bst_node*
   t ) const
{
    if ( NULL == t )
        return NULL;
    else if ( x < t->element )
        return find( x, t->left );
    else if ( t->element < x )
        return find( x, t->right );
    else
        return t; // found it
}
```

and the public wrapper would be

```
template <class Comparable>
const Comparable& find    ( const Comparable& x) const
{
    bst_node* t;
    t = find(x, root);
    if ( NULL != t )
        return t->element;
    else
        // return some indication that it was not found
}
```

5.1.2 The insert algorithm

Insertion is also recursive and similar to the `find` algorithm. It is like searching for the element to be inserted, except that if it is not in the tree, we insert it and if it is in the tree we do nothing. In short, if the current node is empty, we reached the point in the tree at which the element is supposed to be but it is not there, so we insert a new node there containing the element. Otherwise, if the element to be inserted is smaller than the one in the current node, we call `insert()` with the left subtree pointer, if it is larger, we call `insert()` with the right subtree pointer, and if it is equal to the current node's element, if the tree does not allow duplicates, we do nothing. (A variation of this tree would support duplicates, in which case the node would have a counter and possibly a pointer to a list of elements with the given key.)

```
template <class Comparable>
void BST<Comparable>::insert( const Comparable & x, bst_node* & t
    )
{
```

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

```
    if ( NULL == t )
        t = new bst_node( x, NULL, NULL );
    else if ( x < t->element )
        insert( x, t->left );
    else if ( t->element < x )
        insert( x, t->right );
    else
        ;  // Duplicate; do nothing
}
```

and the public wrapper is

```
template <class Comparable>
void BST<Comparable>::insert( const Comparable & x )
{
    insert( x, root );
}
```

5.1.3 The remove Algorithm

The `remove`, or `delete`, algorithm is the most complex. It depends on how many children the node to be deleted has. If it has no children, it is easy: it just deletes the node. If it has one child, it deletes the node and makes the only child the child of the node's parent. If it has two children, then it finds the smallest node in the node's right subtree and copies it into the node, overwriting the element that was in the node and therefore deleting it, and then deleting the node that contained the smallest node in the right subtree. That node cannot possibly have two children because if it did, one would have to be smaller than the node, contradicting the assumption that it was the smallest node in the right subtree.

The algorithm makes a call to the private, recursive `findMin()` function, which finds the smallest element in the tree whose pointer it is given:

```
bst_node* findMin    ( bst_node* t) const
{
    if ( NULL == t )
        return NULL;
    if (NULL == t->left )
        return t;
    return findMin ( t->left );
}
```

(The recursion in this algorithm is easily removed.) The `remove` algorithm is implemented with the following private recursive function and public wrapper.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

```
template <class Comparable>
void BST<Comparable>::remove ( const Comparable& x, bst_node* & t
    )
{
    if ( NULL == t )
        return;   // Item not found; do nothing
    if ( x < t->element )
        remove( x, t->left );
    else if ( t->element < x )
        remove( x, t->right );
    else if ( t->left != NULL && t->right != NULL ) { // Two
        children
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else {
        bst_node *oldNode = t;
        t = ( t->left != NULL ) ? t->left : t->right;
        delete oldNode;
    }
}

template <class Comparable>
void BST<Comparable>::remove( const Comparable & x )
{
    remove( x, root );
}
```

Notes

- The call to `findMin()` followed by the recursive call to `remove()` could be replaced by a call to a single function that finds the minimum and deletes it at the same time, thereby avoiding the second traversal down the right subtree when `remove()` is called.
- If the BST is used in an application in which elements are not deleted very often, ***lazy deletion*** can be used to reduce the running time of deleting at the expense of increased storage. In lazy deletion, the nodes contain an additional member, either a count or a boolean flag. When an element is inserted, the count is set to 1. When it is deleted, it is set to 0. The node itself is not deleted. The running time to delete a node is greatly reduced - it is the same as a find operation - but the number of nodes in the tree is larger than the number of elements. Another benefit is that if the element is inserted again after a deletion, a new node does not have to be allocated; the node's count is reset to 1. This change in the node structure also supports having duplicate nodes in the tree.

5.1.4 Other Algorithms

The remaining methods are fairly easy to figure out, except possibly the `copy()` method, which is used by the BST class's copy constructor. The `copy()` method can be implemented recursively as follows:

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

```
template <class Comparable>
bst_node* BST<Comparable>::copy (bst_node* t) const
{
    if ( NULL == t )
        return NULL;
    else
       return new bst_node( t->element, copy(t->left), copy(t->right) );
}
```

Notice the two recursive calls in the `return` statement. For large trees this is very inefficient and is better implemented non-recursively!

The `print()` method is just an in-order traversal of the tree, and the `make_empty()` function is a post-order traversal in which the operation at each node is a call to delete the node.

5.2 Performance Analysis

The insertion, deletion, and search algorithms each take a number of steps that, in the wost case, is proportional to the height of the tree. Deletion may involve a larger constant than insertion and search because of the extra steps involved when the element to be deleted is in a node with two subtrees, but it still does not visit more nodes than the length of the longest path in the tree. The running time of these algorithms is therefore dependent on the height of the tree. Hence the question, what is the expected, or average, height of a binary search tree?

The order of insertions determines the shape and therefore the height of the tree. If the values 24, 36, 16, 10, 17, 8, 27, 31, 20, 6, 22 are inserted into an initially empty binary search tree, it will result in the tree in Figure 3, whose height is 4. On the other hand, if the values are inserted in the order 36, 31, 27, 24, 22, 20, 17, 16, 10, 8, 6, the tree will be linear, with height 10.

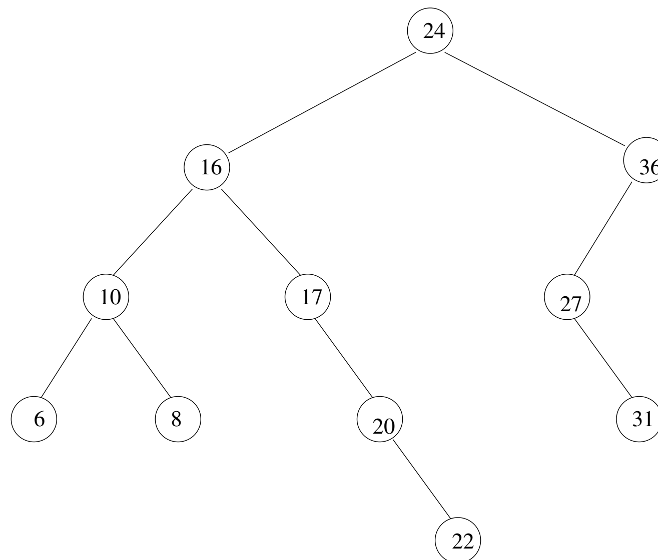

Figure 3: A binary search tree

The preceding argument shows that the order of insertions affects tree height and that in the worst case the height is $O(N)$. But what about the average height? To clarify the problem, we can assume

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

that keys are positive integers. Since the actual numeric difference between successive keys does not affect the shape of the tree, we might as well assume that a tree with N nodes consists of the integers 1 through N. In other words, we get the same tree with the sequence $1, 2, 3$ as we do with $1, 20, 400$ or $2, 23, 800$, so we assume that the sequence is $1, 2, 3, \ldots, N$.

Suppose that the first key to be inserted into the initially empty tree is the number i, where $1 \leq i \leq N$. This implies that the left subtree must have $(i-1)$ nodes and the right subtree must have $(N-i)$ nodes, because there are $(i-1)$ numbers less than i and $(N-i)$ numbers greater than i. This statement is true of every subtree of the tree, namely that the value of the root determines the sizes of its left and right subtrees.

Definition 9. The ***internal path length*** of a tree is the sum of the depths of all nodes in the tree.

Let $D(N)$ denote the average internal path length of an arbitrary tree T with N nodes. Suppose that the root is the number i. Then the left subtree has $(i-1)$ nodes and it has average internal path $D(i-1)$. To get from the root to any node in the left subtree requires traversing one extra edge, so the path to each of the $(i-1)$ nodes in the left subtree is one edge longer and the average internal path length of the left subtree starting at the root is $D(i-1)+(i-1)$. For analogous reasons the right subtree has an average internal path length of $D(N-i)+(N-i)$. This leads to the recurrence relation:

$$\begin{aligned} D(1) &= 0 \\ D(N) &= D(i-1) + D(N-i) + (i-1) + (N-i) \\ &= D(i-1) + D(N-i) + (N-1) \end{aligned} \tag{1}$$

If all sequences $1, 2, \ldots, N$ are equally likely then there is a uniform $1/N$ probability that the first number will be i. In other words, the root may be any of $1, 2, \ldots, N$ with equal probability, and hence, the average internal path length is the sum of the right-hand sides of Eq. 1 with i taking on the values $1, 2, \ldots, N$ divided by N:

$$\begin{aligned} D(N) &= \frac{1}{N}\sum_{i=1}^{N}\left(D(i-1) + D(N-i) + N - 1\right) \\ &= \frac{1}{N}\sum_{i=1}^{N} D(i-1) + \frac{1}{N}\sum_{i=1}^{N} D(N-i) + \frac{1}{N}(N \cdot (N-1)) \\ &= \frac{1}{N}\sum_{i=0}^{N-1} D(i) + \frac{1}{N}\sum_{i=0}^{N-1} D(i) + (N-1) \\ &= \frac{2}{N}\sum_{i=0}^{N-1} D(i) + (N-1) \end{aligned}$$

This recurrence relation will be solved in a later chapter. It will be shown that $D(N)$ is $O(N \log N)$. Therefore the average binary search tree has a total path length of $O(N \log N)$. Since this is the total of all path lengths of a tree with N nodes, the average path is length $O(\log N)$.

This shows that on average, insertions, deletions, and finds are $O(logN)$ operations if the trees are constructed randomly from N keys. But if trees are subjected to deletions, then their shapes are

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

much harder to analyze. This is because the set of all possible binary search trees will not have a uniform distribution, and the problem cannot be modeled as we just did. Some studies have shown that, when insertions and deletions alternate, the expected depth will be $\Theta(\sqrt{N})$. Furthermore, the deletion algorithm favors one side or the other when it must delete an element from a node with two children. To avoid this, the deletion algorithm could alternate between choosing the smallest element in the right subtree and the largest in the left subtree. The effect of this change has not been established definitively.

A more interesting solution to the problem was invented by Tarjan in the early 1980s and the technique was subsequently generalized to other problems. He invented self-adjusting trees, which after each operation, not just insertions and deletions, restructured the tree to make future operations more efficient.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.