---------------------------------------------Trees-----------------------------------

The running time for trees is O(log N) on average.

Binary search tree:

A data structure consisting of the implementation of two library collections classes:

1. TreeSet

2. TreeMap

A tree can be defined in several ways.

One way is to define it recursively.

----------------------------Strcuture and Terminology of Trees-----------------------------

A tree is a collection of nodes.

The initial one at the top is called the ROOT.

Below it are zero or more nonempty SUBTREES

Each of their roots are connected by a directed EDGE from the root

The root of each subtree is called the CHILD of the root.

The root is the PARENT of each subtree root

A tree therefore, consists of N nodes, one of which is a ROOT, and N - 1 EDGES

N-1 edges because each edge connects some node to its parent, and every node except

the root has one parent

Nodes with NO CHILDREN are called LEAVES

Nodes with the SAME PARENT are called SIBLINGS.

GRANDPARENT and GRANDCHILD can be defined in the same way.

---------

A PATH from node1 to nodeK is defined as a sequence of nodes from parent to child from n1, n2, ..... nk.

The LENGTH of the path is the number of edges in this path.

There is a path of length 0 from every node to itself.

The DEPTH of nodeI is the length of the unique path FROM THE ROOT to nodeI.

The ROOT is at depth 0.

The HEIGHT of nodeI is the length of the **longest** path from nodeI to a LEAF.

If there'a path from nI and n2, then node1 is an ANCESTOR of n2.

n2 is a DESCENDANT of n1

If n1 is a PROPER ANCESTOR of n2, then n2 is a PROPER DESCENDANT of n1.

----------------------------Implemenation of Trees------------------------------

One way to implement a tree is to have in each node, besides its data, a link

to each child of the node.

Keep the children of each node in a linked list of tree nodes

This avoids wasting space since the number of children can greatly vary and is not always known in advance

This means that a node will have a link to its next sibling AND its FIRST CHID (the first child will have links to the rest of the siblings)

- LinkedList can still only be built one node at a time

---Applicationn----

A directory in the unix file system is just a file with a list of al its children.

A path name "usr/mark/book/ch1.r" are simply directories with parents and their children.

-------------------Preorder Traversal-----------------

In a PREORDER TRAVERSAL, work at a node is performed BEFORE its children are processed.

Ex: when something is execeuted for a node without first proceeding down to process the node's children

If there are N file names to be output, then the running time is O(N).

---------------PostOrder Traversal------------------

In a POSTORDER traversal, the work at a node is performed AFTER its children are evaluated

-------------------------DIFFERENCE Between Preorder and Postorder Traversal-------------------

Preorders start at root, and move their way downwards and execute actions on the node they're on as they traverse

Postorders start executing their actions at the very lowest child node, and move their way upwards as they go.

-----------------------------------Inorder Traversal----------------------------------------

Listing all the items in sorted order

Strategy:

Process the LEFT subtree first

Process CURRENT node

Process the RIGHT subtree

The total running time is O(N)

Constant work is being performed at every node in the tree.

Each node is visited once

We test each node is tested to see if it's null or not

set up two method calls

(in the example, these are each a recursive call either

to the left or right subtree)

and doing a println

Ex:

//Print tree contents in sorted order

public void printTree()

{

if( isEmpty() )

System.out.println( "Empty tree" );

else

printTree( root );

}

/\*\* INTERNAL method to print subtree in sorted order

\* @param t the node that roots the subtree

\*/

private void printTree( BinaryNode<AnyType> t )

{

if( t != null )

{

printTree( t.left );

System.out.println( t.element );

printTree( t.right );

}

}

--------------------------------------Postorder Traversal------------------------------------

Process both subtrees firt before we process a node

For example:

To compute the height of a node, we computethe height of the subtrees first

This routine declares the height of a leaf to be zero

- A null node is -1, a full node has height 1.

So a leaf (a node with no children), is -1 + 1 = 0.

Total running time is O(N), constant work is also being performed at each node here

Ex:

./\*\*

\* Internal method to compute height of a subtree

\* @param t the node that roots the subtree

\*/

private int height( BinaryNode<AnyType> t )

{

if( t == null )

return -1;

else

return 1 + Math.max( height( t.left ), height( t.right ) );

}

-------------------------------Preorder Traversal--------------------------------------------------------

The node is processed BEFORE its children

This could be useful for labeling each node with its depth.

-------------------------------Commonalities------------------------------------------------

Handle "null" case first, then the rest.

We only pass the "reference" to the node that roots the subtree

No declaration of extra variables are made.

A more compact code = Less probability of bugs

-------------------------------Level-Order Traversal-------------------------------------------

All nodes at depth "d" are processed before any node at depth "d + 1"

This DIFFERS from the other traversals in that it's NOT DONE RECURSIVELY

A QUEUE is used, instead of the implied STACK of RECURSION

------------------------Remove------------------------------------------------------

As is common with many data structures, DELETION is the hardest operation.

There are several possibilities to consider:

If the node is a leaf, it can be deleted immediately.

If the node has one child, the node can be deleted after its parent adjusts a link to bypass the node

What if there's a node with two children?

1. We replace the data of the chosen node with the SMALLEST node of the RIGHT subtree

2. Recursively delete the node whose data was just used to replace the deleted node

Because the smallest node in the right subtree can't have a left subtree,

this "remove" is an easy one.

This second "remove" of this smallest node is the same as deleting a node that only has one child

The code below performs this deletion, but is inefficient: it makes TWO passes down the tree to find and delete the

smallest node in the right subtree when it's appropriate

This efficiency can be quickly removed by writing a special "removeMin" method

Ex:

\*we return the new root of the subtree\*

\* "t" is the node that rots the subtree \*

\* "x" is the item to remove \*

private BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )

{

if ( t == null )

return t; // item is not found; do nothing

int compareResult = x.compareTo(element);

if( compareResult < 0 )

t.left = remove( x, t.left );

else if ( compareResult > 0 )

t.right = remove( x, t.right );

else if ( t.left != null && t.right != null ) // two children

{

t.element = findMin( t.right ).element; // setting the node that is to be "removed" to the smallest node on the right subtree

t.right = remove ( t.element, t.right ); // removing the smallest node on the right subtree (that we just assigned to the replaced, or "removed" node)

}

else

t = ( t.left != null ) ? t.left : t.right;

return t;

}

----------------------------------------------

lazy deletion:

When an element is to be deleted, it is left in the tree and merely MARKED as being deleted.

This could be as simple as "valid = false".

This means we must change the methods accordingly to accommodate this marker.

This is popular especially if duplicate items are present, because then the field that keeps count of frequency

of appearance can be decremented

If the number of real nodes in the tree is the same as the number of "deleted" nodes, then the depth of the tree

is only expected to go up by a small constant

Therefore, there is a very small time penalty for lazy deletion.

Also, if a deleted item is REINSERTED, the big structure of allocating a new cell is avoided

------------------------Insert-------------------------------------------------------

1. To insert X intro tree T, proceed down the tree as you would with a "contains"

2. If X is found, do nothing (or "update" something)

3. Otherewise, insert x at the last spot on the path traversed

As seen in the example code:

We traverse the tree as though a "contains" were occurring

At the node with item 4, we need to go right, but there is no right subtree

Therefore, 5 is not in the tree.

We insert 5 at this spot, in the place of the empty right subtree

Since "t" references the root of the tree, and the root CHANGES on the first insertion,

"insert" is written as a method that returns a reference to the ROOT of the new tree

------------------------------------------------------------------------------

private BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t )

{

if (t == null )

return new BinaryNode<>(x, null, null );

int compareResult = x.compareTo( t.element );

if (compareResult < 0 )

t.left = insert( x, t.left );

else if ( compareResult > 0 )

t.right = insert( x, t.right );

else

; // Duplicate; do nothing

return t;

}

------------------------------------------------------------------------------

Duplicates are handled by keeping an extra field in the node record that indicates the

FREQUENCY OF OCCURENCE

This adds extra space to the entire tree but is better than putting duplicates in the tree

NOTICE:

This strategy does not work if the key that guides the "compareTo" method is only part of a larger structure

In THAT case, we keep all of the structures that have the same key in an AUXILIARY DATA STRUCTURE,

such as a list or another search tree

----------------------------------findMin & findMax-----------------------------------

These PRIVATE routines return a REFERENCE to the node containing the SMALLEST

and LARGEST elements in the tree, respectively (with the name of the functions)

findMin:

1. Start at the root

2. Go left as long as there is a left child

3. The stopping point is the smallest element

findMax has the same procedure as findMin except the branching is towards the right child

This tends to be so easy that most do not bother with recursion

We will code the routines both ways, doing "findMin" recursively and "findMax" nonrecursively

--------------------------------------------------------------------------------

findMin:

// recursive

private BinaryNode<AnyType> findMin( BinaryNode<AnyType> t) {

if (t == null ) {

return null;

} else if ( t.left == null ) {

return t;

}

return findMin( t.left );

}

findMax:

// nonrecursive

private BinaryNode<AnyType> findMax ( BinaryNode<AnyType> t ) {

if (t != null )

while ( t.right != null )

t = t.right;

return t; }

Notice how carefully we handle the degenerate case of an empty tree.

We place the test for emptiness first

This is especialy important in recursive programs

Also notice that it is safe to change "t" in "findMax", since we are only working with a

copy of the reference

Think variable scopes.

HOWEVER: a statement like "t.right = t.right.right" will actually make changes

- Seems like it'll be an infinite loop since after being assigned "t.right.right",

"t.right" never changes

--------------------------------Binary Search Trees------------------------------

If the binary search tree is balanced, the running time is O(log N)

If the binary search tree is NOT balanced, WHICH IS THE WORST-CASE SCENARIO, the running time is O(N)

- Like a linked list, which is a degenerate binary tree that only goes down one side

ON A TEST, YOU MUST ASSUME WORST-CASE SCENARIO - THIS MEANS ASSUME THE BINARY SEARCH TREE IS UNBALANCED

------------------------------------------------------------------------------------------

BInary Search Tree:

For every node X, in the tree, the values of all the items in its LEFT subtree are SMALLER than the item in X,

and the values of all the items in its RIGHT subtree are LARGER than the item in X.

--------------------------Operations-------------------------------

Because of the recursive definition of trees, it's common to write these routines RECURSIVELY

Since the average depth of a binary tree is O(log N), we usually do not need to worry about running out of stack space.

The binary search tree REQUIRES that all the items can be ordered.

In order to write this class generically, we need to use the interface COMPARABLE (from Chapter 1)

- It allows us to compare any two items in the tree using the "compareTo" method

Instead of the "equals" method, we determine if the items are equal if and only if the

"compareTo" method RETURNS 0

Alternatively, we can instead also use a function object to implement the binary search tree.

Similar to the LinkedList class, the "BinaryNode" class is a nested class.

private static class BinaryNode <AnyType> {

// Constructors

BinaryNode( AnyType theElement) {

this ( theElement, null, null );

}

BinaryNode ( AnyType theElement, BinaryNode<AnyType> lt, BinaryNode<AnyType> rt) {

element = theElement;

left = lt;

right = rt;

}

AnyType element; // the data in the node

BinaryNode<AnyType> left; // Left child

BinaryNode<AnyType> right; // Right child

}

----------------------Contains---------------------------

Returns true if there is a node in tree T that has item X

- false if there is no such node

If T is empty, we can just return false.

If the item stored at T is X, we can return true.

Otherewise, we make a recursive call on a subtree of T

- either left or right, corresponding to the "compareTo" result being < or > 0

// x is the item to search for

// t is the node that roots the subtree

private boolean contains(AnyType x, BinaryNode<AnyType> t) {

if (t == null)

return false;

int compareResult = x.compareTo( t. element );

if (compareResult < 0)

return contains( x, t.left );

else if ( compareResult > 0 )

return contains( x, t.right );

else

return true; // A match has been found

}

NOTICE THE ORDER OF THE TESTS:

We test for an empty tree FIRST

- Otherwise, we generate a "NullPointerException" attempting to access a data

field through a "null" reference

The remaining tests are also arranged with the LEAST LIKELY case LAST

Also notice that tail recursion is used here, which means they can easily be removed

with a while loop

HOWEVER: the simplicity of the algorithmic expression here and the amount of stack space available from

O(log N) allows us to use tail recursion without much consequence

--------------------------Binary Trees and Expression Trees--------------------------

A binary tree is a tree in which NO node can have more than TWO children

The depth of an average binary tree is considerably smaller than N.

The average depth is O(sqrt(N)).

For a BINARY SEARCH TREE, the average value of the depth is O(log N).

HOWEVER, the depth can be as large as N-1

-------------------------------Implementation-------------------------------------

Because a binary tree node has AT MOST TWO children, we can keep DIRECT LINKS to them.

For doubly linked lists, a node consists of the element information and TWO REFERENCES (left and right) to other nodes.

class BinaryNode {

// Friendly data; accessible by other package routines

Object element; // the data in the node

BinaryNode left; // left child

BinaryNode right; // right child

}

---------------Example: Expression Trees---------------

The leaves of an expression tree are OPERANDS (constants or variable names)

The other nodes contain OPERATORS

This tree is binary because all operators are binary

Refer to pg. 109, Figure 4.14, to see what an expression looks like.

----Using Inorder Traversal-----

Using an expression tree, we can produce an overly parenthesized infix expression

by recursively producing a parenthesized left expression, printing out the operator at the root,

and finally recursively producing a parenthesized right expression

This strategy of (left, node, right) is known as an INORDER INTRAVERSAL

Ex:

left subtree: a + (b\*c)

right subtree: ((d \* e) + f) \* g

Total: (a + (b \* c)) + ((d \* e) + f) \* g)

--Using Postorder Traversal-----

Recursively print out the left subtree, the right subtree, and THEN the operator (left, right, Node)

More like, moving from left to right, starting witht the operand and moving to the operator of the two operands afterwards,

THEN proceeding to next operands, etc...

Ex:

a b c \* + d e \* f + g \* +

--Using Preorder Traversal------

Print out operator first and then recursively print out the left and right subtrees (node, left, right)

Ex:

+ + a \* b c \* + \* d e f g

-------------------------Constructing an Expression Tree------------------------------

Let's give an algorithm to convert a postfix expression into an expression tree

We already have an algorithm to convert infix to postfix, so we can generate expression trees from the two common types of input

This method will strongly resemble the postfix evaluation algorithm from before with the Stacks chapter (and we will use stacks here too!)

-----Example-----

Input:

a b + c d e + \* \*

1. The first two symbols are operands, so we create one-node trees and PUSH these trees onto a stack

2. "+" is read, so two trees are POPPED, a NEW TREE is formed after merging these three elements, and this NEW TREE is PUSHED onto the stack

3. "c", "d", and "e" are read. For each of the operands, a one-node tree is created and the corresponding tree is PUSHED onto the stack

4. "+" is read, so two trees are POPPED (the tree with the "+" is now out) and merged with "+" as root, then PUSHED

5. "\*" is read, so two trees are POPPED and form a new tree with "\*" as root, (a new tree is PUSHED onto the stack?)

6. Last symbol, "\*" is read, two trees are merged, and the final tree is left on the stack.

-----In-Class Example------

class ExprNode {

char operator;

int operand;

ExprNode left;

ExprNode right;

}

void postFix (ExprNode t) {

if (t == null)

return;

postFix( t.left ); - if there is no left or right, then we return null, as stated in the base case

postFix( t.right);

if (t.left != null)

System.out.println( t.operator );

else

System.out.println( t. operand );

}

---------Creating an Expression Tree class

public class ExprTree {

private ExprNode root;

public ExprTree(String postfix) {

/\* run through the stack based algorithm to

\* buld the expr tree.

\*

\* Remember, you're pushing ExprNode on to the stack.

\* When done, POP the stack and make that the root

\*/

}

public int eval() {

// public version of the function

return eval(root); // allows us to hide more of the tree structure and keep things clean and organized

}

private int eval(ExprNode t) {

// do your traversal to evaluate the tree rooted at t and return it

}

static private class ExprNode {

int operand;

char operator;

ExprNode left;

ExprNode right;

}

}

----------------------------------AVL Trees------------------------------------------

An AVL (Adelson-Velskii and Landis) tree is a binary search tree with a BALANCE CONDITION.

The balance condition must be easy to maintain, and it ensures that the depth of the tree is O(log N)

The simplest idea for a balance condition:

Require that the left and right subtrees have the same height.

Another idea:

Every NODE must have left and subtrees of the same height.

If the height of an empty subtree is defined to be -1 (as is usual), then only perfectly balanced trees of 2^k - 1 nodes would

satisfy this criterion.

Thus, although this guarantees trees of small depth, the balance condition is too rigid to be useful and needs to be relaxed.

An AVL tree is identical to a binary search tree, except that for EVERY NODE in the tree, the height of the left and right subtrees

can differ by at most 1. (No matter how big the subtree is for each node - even if we're talking about a root node and two immense branches)

(The height of an empty subtree is defined to be -1).

REMEMBER:

The HEIGHT of a node is the length of the longest path from the node to a LEAF (nodes with no children - effectively the end of a tree)

The HEIGHT of a TREE is the length of the longest path from the root to the FARTHEST LEAF (the very very bottom of the tree).

All the tree operations of an AVL tree can thus be performed in O(log N) time, except possibly insertion (we'll assume lazy deletion).

-----------------------Insertion------------------

When we do an insertion, we need to update all the balancing information for the nodes on the path BACK to the root.

This means, however, that an insertion could VIOLATE the AVL Tree property

(It could cause a greater difference in height that exceeds 1 between the subtrees of a node)

If this is the case, then this property has to be restored BEFORE the insertion step is CONSIDERED OVER.

This can always be done with a simple modification to the tree, known as: ROTATION

After an insertion, only nodes that are on the PATH from THE INSERTION POINT to the ROOT might have their

balance altered (whose new balance violates the AVL property)

Because, only those nodes have their subtrees altered.

Since any node has at most TWO children, and a height imbalance requries that the node's two subtrees' height differe by TWO,

we see four possible cases for why a violation might occur:

1. Insertion into the LEFT subtree of the LEFT child of the node

2. Insertion into the RIGHT subtree of the LEFT child of the node

3. Insertion into the LEFT subtree of the RIGHT child of the node

4. Insertion into the RIGHT subtree of the RIGHT child of the node

The first case, and fourth case, where the insertion occurs on the "outside" (left-left, or right-right) is fixed by:

a SINGLE ROTATION of the tree

The second, and third case, in which the insertion occurs on the "inside" (left-right, right-left), is handled by:

a DOUBLE ROTATION

These are fundamental operations that will be used several times in balanced-tree algorithms.

--------------------------------------------Single Rotation-----------------------------------------------------

Let's go through an example (Figure 4.31):

Scenario:

NodeK2 has a right subtree with a single-level subtree Z.

NodeK2 has a left subtree of NodeK1.

Node K1 has a two-level left subtree X, and a single-level right subtree Y.

NodeK2 violates the AVL balance property because its LEFT subtree is two levels deeper than its right subtree

(Height of subtree X + NodeK1 = Three levels)

Its left-left subtree X has grown another level from an insertion, causing it to be two levels deeper than NodeK2's right subtree Z.

To IDEALLY rebalance the tree, we need to move X up a level and Z down a level.

TECHNICALLY this COMPLETELY balances the height, and is more than the AVL property requires

Solution:

Make NodeK1 the new root.

NodeK2 becomes the right child of NodeK1

X remains as left child of K1

Z remains as right child of K2

Y, can then be placed as K2's LEFT CHILD

Ordering requirements remain satisfied AND the tree is balanced now.

This work only requires a few link changes.

X moves up on level

Y stays at the same level

Z moves down a level

K1 and K2 now have subtrees of exactly the same height

FURTHERMORE:

The new height of the ENTIRE subtree is EXACTLY THE SAME as the height of the original subtree PRIOR to the insertion into X.

THUS, no further updating of heights on the path to the root is needed,

And consequently, NO FURTHER ROTATIONS are needed.

Think of it as an ACTUAL rotation (either clockwise OR counter-clockwise rotation)

-----------------Using a Single Rotation----------------

The left-left and right-right holds true LITERALLY for one PAIR of nodes.

Don't look at it in terms of the ENTIRE tree, if it's COMPLETELY ON THE OUTSKIRTS of the LARGER TREE.

Left-left could still seem to look like it's INSIDE the LARGER TREE, but in terms of the PARENT node and its CHILD node....

if it's on the left of the CHILD's left subtree, THEN IT'S CASE 1 - ON THE OUTSIDE

------------------------------------------Double Rotation-----------------------------------------------------------

Let's look closer into the Y subtree from before (from Figure 4.31), and let it be a node with its own two subtrees.

Basically, the node of Y goes up to the very top with the NodeK1 on its left subtree, and NodeK2 on its right subtree.

The original element on the left subtree of Y goes to the RIGHT subtree of NodeK1 (it's still on the left subtree of NodeY)

The original element on the right subtree of Y goes to the LEFT subtree of NodeK2 (it's still on the right subtree of NodeY)

Keep in mind that in diagram 4.35, NodeK1, NodeK2, and NodeK3 are labeled differently than how I described it here.

There, NodeK2 is NodeY.

My descriptions are based on Figure 4.31

Think of it as NodeY just going up to the top of both nodes

Then giving its LEFT subtree to the original root's LEFT child's RIGHT subtree

Then giving its RIGHT subtree to the original root's RIGHT child's LEFT subtree

These child nodes are the nodes that were right above NodeY.

Right child is the above right.

Left child is the above left.

-----------Deciding if we need to do a Double Rotation----

2. Insertion into the RIGHT subtree of the LEFT child of the node

3. Insertion into the LEFT subtree of the RIGHT child of the node

NO MATTER HOW BIG THE TREE IS, look only in terms of ONE PARENT NODE, and its TWO CHILD NODES.

-----------------------Step-By-Step----------------------------------

A valid, unbalanced binary search tree: 3-2-4

1. Rot: Rotate 3 and 2. (so that they swap): 2-3-4

2. Rot: rotate 3 and 4

3

2 4

Now tree is balanced.

Another example:

k1

k2 c

A k3

b1 b2

We want k3 to be the root

Rotation 1: Rotate k3 and k2

k1

k3 c

k2 b2

A b1

Rotatation 2: Rotate k3 and k1

k3

k2 k1

A b1 b2 c

Now it's balanced

-----------------------------------------------------------Algorithm for Rotations---------------------------------------------------------

1. Recursively insert X into the appropriate subtree of T

2. If the heght of the subtree does not change, we're done.

3. If a height imbalance appears in T:

We choose to either do a single or double rotation based on X and its RELATION to the rest of the items in T

4. Update the heights - checking up on the path upwards

5. DONE

Because one rotation always suffices (either single or double),

a carefully coded NON-RECURSIVE version generally turns out to be FASTER than a RECURSIVE version

However, it's difficult to code a non-recursive one, whereas a recursive implementation is easily readable

STORAGE OF HEIGHT INFORMATION:

Because the difference in height is guaranteed to be small, we could just use TWO BITS (+1, 0 -1)

HOWEVER: this results in some loss of clarity

The slight speed advantage obtained by storing balance factors is hardly worth the loss of clarity and relative simplicty of the code

Furthermore, most machines will align to at least an 8-bit BYTE anyway, allowing us to store heights up to 127.

Because the tree is balanced, this amount of space is plenty sufficient.

-----------------------------Example Code------------------------------------

First, we need to write the AvlNode class.

private static class AvlNode<AnyType>

{

// Constructors

AvlNode( AnyType theElement )

{

this( theElement, null, null );

}

AvlNode( AnyType theElement, AvlNode<AnyType> lt, AvlNode<AnyType> rt )

{

element = theElement;

left = lt;

right = rt;

height = 0;

}

AnyType element; // the data in the node

AvlNode<AnyType> left; // left child

AvlNode<AnyType> right; // right child

int height; // height

We also need a quick method to return the height of a node.

We need it to handle the case of a "null" reference.

/\*\*

\* Return the height of node t, or -1, if null.

\*/

private int height( AvlNode<AnyType> t )

{

return t == null ? -1 : t.height; // if true (that t = null), then return "-1"

// OTHERWISE, return "t.height"

}

Now we write the basic insertion routine, compared to the other method,

we also add a line that invokes the "balance" method

The "balance" routine applies a SINGLE or DOUBLE ROTATION if NEEDED,

UPDATES the HEIGHT,

and RETURNS the RESULTING TREE

/\*\*

\* "x" is the item to insert

\* "t" is the node that ROOTS the subtree

\* RETURN the NEW ROOT of the subtree

\*/

private AvlNode<AnyType> insert( .......... [continue later after written homework or if necessary for homework]

------------------------------Average-Case Analysis------------------------------

So far, all the operations of "contains', "remove", "findMin", "findMax", and "insert", have O(log N) time.

This is because in constant time we descend a level of a tree, but then when we operate,

we operate on a tree that is now roughly half as large

Indeed, the running time of all operations is O(d), where "d" is the depth of the node containing the accessed item

(for "remove": this might be the node that is replaced)

The AVERAGE DEPTH over ALL NODES in a tree is O(log N) on the assumption that all insertion sequences

are EQUALLY LIKELY.

In the ABSENCE OF DELETIONS, or when LAZY DELETION is used, we can conclude that the average running times

of the operation above are O(log N).

NOTICE:

If an input comes intro a tree PRE-SORTED, then a series of "inserts" will take QUADRATIC time and give

a very expensive implementation of a linked list.

This is because the tree will consist only of nodes with no left children.

Internal path length:

The SUM of the depths of all nodes in a tree

For a BINARY SEARCH TREE, we cannot guarantee an O(log N) bound on any single operation,

BUT we can show that any sequence of M operations takes total time O(M logN) in the worst case scenario.