Lesson 9
Binary Search Trees:
Solving Problems by Engaging the Field of Solutions

Wholeness of the Lesson

Binary search trees make it possible to store data in memory while preserving a specified order in a way that cannot be achieved as efficiently using any kind of a list. However, their worst case performance, which can potentially occur more often than desirable, reduces their efficiency to that of a linked list. One example of reducing or eliminating this worst case performance is through the introduction of a *balance condition* to ensure the tree never becomes skewed (for example, AVL trees and Red-Black trees.

Maharishi's Science of Consciousness: Balance is the expression of the *invincible* quality of pure creative intelligence, which preserves the integrity of unboundedness even as it is expressed within boundaries.

Binary Search Trees

1. For many purposes, keeping data stored in memory in a sorted order is a way to optimize a variety of searches on the data.

   A typical problem one might try to solve when it is possible to keep data sorted is:

   Find all Employees having a salary between $50,000 and $75,000.

2. If Employees have been maintained in sorted order – sorted by salary -- in some kind of list, then to solve the problem, we find the first Employee in the list with salary no less than 50,000 and also find the first Employee with salary bigger than 75,000. In this way we can specify the desired range of Employees.

3 How hard is it to implement this strategy: *maintain sorted order to optimize searches*?

   A. If we are using an ArrayList, then maintaining the list in sorted order is expensive because each time we add a new element, it must be inserted into the correct spot, and this requires array copy routines
   Exercise: What is the average-case running time to insert a new element?

   B. If we are using a LinkedList, it is easy to maintain sorted order since insertions are efficient, but searches are not very efficient.
   Exercise: What is the worst-case running time to carry out BinarySearch in a LinkedList?

4. *The Need.* We need a data structure that performs insertions efficiently in order to maintain sorted order (like a linked list) but that also performs finds efficiently (as in an array list where binary search is highly efficient)

A Solution: Binary Search Trees

1. A *binary tree* is a generalization of a linked list. It has a *root node* (analogous to the top node or header node in a linked list). And each node has a reference to a left and right child node (though some references may be null).

2. A *binary search tree* (BST) is a binary tree in which the BST Rule is satisfied:

   BST Rule:
   At each node N, every value in the left subtree of N is less than the value at N, and every value in the right subtree of N is greater than the value at N.

   For the moment, we assume all values are of type Integer and that there are no duplicates.

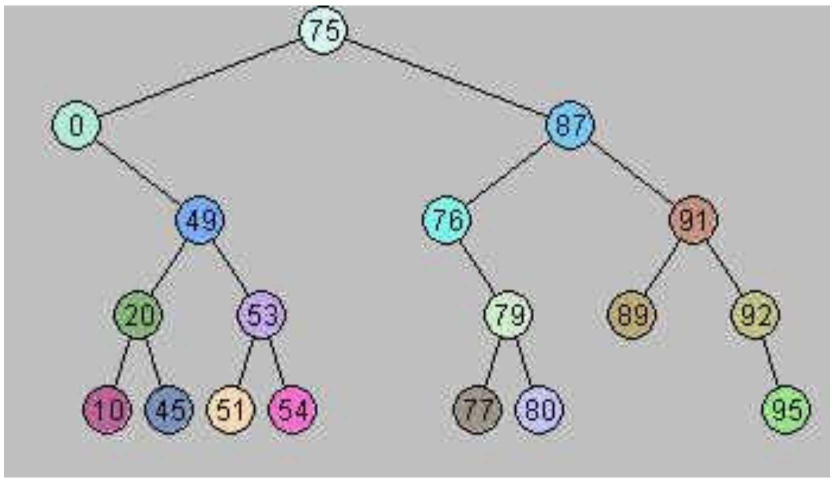
3. The fundamental operations on a BST are:

   ```
   public boolean find(Integer val)
   public void insert(Integer val)
   public boolean remove(Integer val)
   public void print()
   ```

4. When implemented properly, BSTs perform insertions and deletions faster than can be done on Linked Lists and performs any `find` with as much efficiency as the binary search on a sorted array.

5. In addition, because of the BST Rule, the BST keeps all data in sorted order, and the algorithm for displaying all data in its sorted order is very efficient.

A Binary Search Tree



Algorithm for insertion of Integer x [an iterative implementation is given below]

1. If the root is null, create a new root having value x

2. If x is less than the value in the root, examine the left subtree of the root; if bigger, examine the right subtree

3. If a subtree we are examining is null, create a new node having value x and attach it. Otherwise, if x is less than value stored in the root of the subtree, examine its left subtree; otherwise examine its right subtree

Exercise: Insert the following into an initially empty BST: 1,8,2,3,9,5,4

Recursive algorithm for finding an Integer x

Finds and insertions can be done recursively. The recursive find operation for BST's is in reality the binary search algorithm in the context of BST's.

1. If the root is null, return false

2. If the value in the root equals x, return true

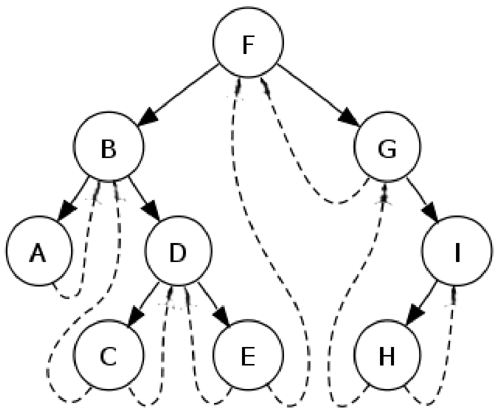
4. If x is less than the value in the root, return the result of searching the left subtree

5. If x is greater than the value in the root, return the result of seraching the right subtree

Recursive print algorithm – outputs values in every node in sorted order

1. If the root is null, return
2. Print the left subtree of the root, in sorted order
3. Print the root
4. Print the right subtree of the root, in sorted order

The path in which nodes are visited for this print algorithm is called an *in-order traversal*. The sequence of visited nodes starts at the far left and follows the only path possible so that nodes with smaller values are always visited before nodes with larger values.



Algorithm to remove Integer x

```
Case I: Node to remove is a leaf node
    Find it and set to null

Case II: Node to remove has one child
    Create new link from parent to child, and set node to be removed to null

Case III: Node to remove has two children
    Find smallest node in right subtree – say it stores an Integer y. This node has at
        most one child
    Replace x with y in node to be removed
    Delete node that used to store y – this is done as in Case II
```

Sample Code

```
/** Assumption: Values inserted into the tree are distinct. For insertion
  * sequences that contain duplicates, a different implementation of BST must
  *  be used
  */
public class MyBST {
   /** The tree root. */
   private BinaryNode root;

   public MyBST() {
     root = null;
   }
   /**
    * Prints the values in the nodes of the tree
    * in sorted order.
    */
    public void printTree( ) {
        if( root == null )
            System.out.println( "Empty tree" );
        else
            printTree( root );
    }
    private void printTree( BinaryNode t ){
        if( t != null ){
            printTree( t.left );
            System.out.println( t.element );
            printTree( t.right );
        }
    }
```

```
 public void insert(Integer x) {
   if (root == null) {
      root = new Node(x, null, null);
      return;
   }
   Node n = root;
   boolean inserted = false;
   while(!inserted){
      if(x.compareTo(n.element)<0) {
         //space found on the left
         if(n.left == null){
            n.left = new Node(x,null,null);
            inserted = true;
         }
         //keep looking
         else {
            n = n.left;
         }
      }
      else if(x.compareTo(n.element)>0){
         //space found on the right
         if(n.right==null){
            n.right = new Node(x,null,null);
            inserted = true;
         }
         //keep looking
         else {
            n = n.right;
         }
      }
   }

}
```

```
  private class BinaryNode {

        // Constructors
     BinaryNode( Integer theElement ){
        this( theElement, null, null );
     }

     BinaryNode( Integer element,
                 BinaryNode left,
                 BinaryNode right ){
        this.element = element;
        this.left = left;
        this.right = right;
     }
     private Integer element;      // The data in the node
     private BinaryNode left;      // Left child
     private BinaryNode right;     // Right child
  }
}
```

Analysis of Performance of BSTs

1. The running time of insertion, deletion, and search are all proportional to the time it takes to arrive at the node of interest. If *d* is the depth of this node, it therefore follows that performance of these operations is always O(*d*).

2. The question remains: How big is *d*? In the worst case, d is Θ(n), where n is the number of nodes in the tree. In good cases, when the tree is sufficiently balanced, d is Θ(log n). In fact, average-case analysis shows that d is Θ(log n). On the next slide, we discuss one of the ways this result is obtained

3. To improve the worst-case running time for BST operations, much research has been done on how to ensure that a binary tree, while it is growing, remains balanced, so that even in the worst case, the operations run in O(log n). We introduce several variations at the end of this lesson. The next lesson focuses on the most widely used variant: red-black trees.

Average Case Analysis of BSTs

1. What does "average-case" mean in the case of BSTs? There is more than one interpretation, though in each case the result is the same. We consider one approach here.

2. The main result is: The average depth of a node in a *randomly built* BST having *n* nodes is O(log *n*).

3. A BST with *n* nodes is said to be *randomly built* if *n* distinct integers are randomly chosen and inserted successively into an initially empty BST. (An alternative analysis uses the concept of a *randomly chosen* BST: What is meant in this case is that all BSTs having *n* nodes and storing as data the distinct integers 0, 1,…, n - 1 are equally likely to occur.)

4. Given a randomly build BST *T*, P(*T*) denotes the total path length of *T*, which is the sum of the depths of all the nodes in *T*. If we let P(*n*) denote the average path length for all BSTs having *n* nodes (where data consists of distinct values from the range 0… *n* - 1), then one shows that P(*n*) is O(*n*log *n*). So the average depth of a single node is O(log n). One shows that P(*n*) satisfies the recurrence

   $$P(n) = P(k) + P(n - k - 1) + n - 1,$$

   where $0 \leq k \leq n$ (path length of left subtree + path length of right subtree + adding 1 to every path length relative to the root of main tree), and the solution is indeed O(nlog n)

Running Time of In-Order Traversal (“Print Tree”)

Pseudo-Code

Algorithm: In-OrderTraversal(*node*)

```
Input: A node node of a binary tree B with n nodes and with left child left and right
child right
Output: Every node in the subtree of B at node is marked/printed

if node $\neq$ null then
    In-OrderTraversal(left)
    print(node)
    In-OrderTraversal(right)
```

Running Time – Guessing Method

Let c be the time to evaluate “*node* ≠ null**”** and let d be the time spent within each self-call, other than other self-calls.

Then T(0) = c (in this case the root is null). If B has a non-null root and left subtree has k elements then right subtree has n – k – 1 elements, then running time is given by:

$$T(n) = T(k) + T(n - k - 1) + d$$

We try values to guess a formula for T(n):

$T(0) = c$
$T(1) = T(0) + T(0) + d = c + d + c$
$T(2) = T(1) + T(0) + d = (c+d+c) + c + d = (c + d) * 2 + c$
$T(n) = (c+d) * n + c$

Verification

Let f(n) = (c + d)n + c. Clearly f(0) = c. We must show f satisfies the recurrence. Assume that it does for all m < n. Then for any k for which $0 \leq k \leq n$,

$$\begin{aligned} f(n) &= (c + d)n + c \\ &= [(c + d)n + c] - (c + d) + c + d \\ &= [(c + d)k + c] + [(c + d)(n - k - 1) + c] + d \\ &= f(k) + f(n - k - 1) + d \end{aligned}$$

Handling Duplicates in A BST

1. Running times of find, insert, and delete given above, and their implementations, make the assumption that there are no duplicate values.

2. To handle duplicate values, store in each Node a List (instead of a value); then when a value is added to a Node, it is instead added to the List.

3. Example: Suppose we are storing Employees in a BST, ordered by Name. Our BST will be created so that the value in each node is a List<Employee> instance. Then, after insertions, all Employees with the same name will be found in a single List located in a single Node.

4. Example: Suppose we want to have a data structure that provides us with a count of how many Employees have a particular salary. To begin, we could order Employees by salary (using a SalaryComparator), insert into a BST using Lists as values in each Node. After all Employees have been inserted, we could answer the query "How many have salary 50000?" by searching for the 50,000 node and then returning the size of the list stored at that node.

Using BSTs for Sorting

1. Consider the following procedure for sorting a list of Integers:
    - Insert them into a BST
    - Print the results (or modify "print" so that it puts values in a list)

2. *Exercise:* How good is this new sorting algorithm?

3. Here is an alternative approach to sorting a list of Integers using a BST:
    - Insert them into a BST
    - Repeatedly read off and remove the min value in the tree

Introduction to Balanced BSTs: AVL Trees

1. In order to avoid worst case scenarios for BSTs, several kinds of *balanced* BSTs have been devised. Typically, these work by formulating a *balance condition* that ensures that, as long as the condition is satisfied, the BST (having n nodes) must have height O(log n). The balance condition is forced to be true by performing balancing steps after every insertion and deletion.

2. One of the first kinds of balanced BSTs was the *AVL Tree* (A.V. L. are the first letters of the last names of the three guys who invented it).

3. *The AVL Balance Condition:* For any node x, the difference between the height of the left subtree at x and the right subtree at x is at most 1.

   (We adopt the convention that the height of a tree having 0 nodes– i.e. a null root -- is –1)

4. **AVL Tree Height Theorem.** Every AVL tree having n nodes has height O(log n). Therefore, insertion, deletions, and searches all run in O(log n), even in the worst case.

   The proof is based entirely on the AVL Balance Condition. Using this condition, one shows by induction on height that if an AVL tree has height h and has n nodes, then $n \geq F_{h+1}$, where F is the Fibonacci sequence. Since F grows exponentially, it follows from this inequality that h is O(log n).

Optional: Proof of the AVL Tree Height Theorem

Notice first that for all k,

(*) $$2F_k > F_{k+1}$$

This follows because $2F_k = F_k + F_k > F_k + F_{k-1} = F_{k+1}$.

Lemma. If an AVL tree has height *h* and has *n* nodes, then $n \geq F_{h+1}$, where *F* is the Fibonacci sequence

We prove the Lemma by induction on height. When h = 0, n = 1 and the result is obvious. Assume the result holds for heights less than h.

Let T be AVL with height h and n nodes. Let T_L and T_R be the left and right subtrees of T; let n_L be the number of nodes in T_L and n_R the number of nodes in T_R; let h_L be the height of T_L and h_R the height of T_R. Assume $h_L \geq h_R$. Then $h = h_L + 1$

Case 1: $h_L = h_R$

$$\begin{aligned} n &= n_L + n_R + 1 \\ &\geq F_{hL+1} + F_{hR+1} + 1 \\ &\geq 2F_{hL+1} + 1 \\ &\geq F_{hL+2} + 1 \\ &= F_{h+1} + 1 \\ &\geq F_{h+1} \end{aligned}$$

Case 2: $h_L = h_R + 1$

$$\begin{aligned} n &= n_L + n_R + 1 \\ &\geq F_{hL+1} + F_{hR+1} + 1 \\ &= F_h + F_{h-1} + 1 \\ &= F_{h+1} + 1 \\ &\geq F_{h+1} \end{aligned}$$

Main Point

AVL trees are binary search trees in which remain balanced after insertions and deletions by preserving the AVL *balance condition.* The balance condition is: *For every node in the tree, the height of the left and right subtrees can differ by at most 1*. The balance condition is maintained, after insertions and deletions, by strategic use of *single* and *double rotations*. Worst-case running time for *insert, remove, find* is O(log *n*). The balance condition illustrates the principle that boundaries can serve to *give expression to* boundless intelligence rather than simply *limiting* that intelligence.

三十幅共一轂: 當其無有車之用。 埏埴以為器, 當其無有器之用。 鑿戶牖以為室, 當其無有室之用。 故有之以為利, 無之以為用。	Thirty spokes share the wheel’s hub; It is the center hole that makes it useful. Shape clay into a vessel; It is the space within that makes it useful Cut doors and windows for a room; It is the holes which make it useful. Therefore, profit comes from what is there; Usefulness comes from what is not there. *Daodejing 11*

Red-Black Trees

1. Red-black trees are a more recent (and more efficient) alternative to AVL trees, though the balance condition is slightly more complicated.

2. A BST is *red-black* if it has the following 4 properties:

   A. Every node is colored either red or black
   B. The root is colored black.
   C. If a node is red, its children are black.
   D. For each node *n*, every path from *n* to a NULL reference has the same
      number of black nodes.

Theorem. Every red-black tree having *n* nodes has height O(log *n*). Therefore, insertion, deletions, and searches all run in O(log *n*), even in the worst case.

CONNECTING THE PARTS OF KNOWLEDGE
TO THE WHOLENESS OF KNOWLEDGE

1. A Binary Search Tree can be used to maintain data in sorted order more efficiently than is possible using any kind of list. Average case running time for insertions and searchers is $O(\log n)$.

2. In a Binary Search Tree that does not incorporate procedures to maintain balance, insertions, deletions and searches all have a worst-case running time of $\Omega(n)$. By incorporating balance conditions, the worst case can be improved to $O(\log n)$.

3. *Transcendental Consciousness* is the field of perfect balance. All differences have Transcendental Consciousness as their common source.

4. *Impulses Within The Transcendental Field.* The sequential unfoldment that occurs within pure consciousness and that lies at the basis of creation proceeds in such a way that each new expression remains fully connected to its source. In this way, the balance between the competing emerging forces is maintained.

5. *Wholeness Moving Within Itself.* In Unity Consciousness, balance between inner and outer has reached such a state of completion that the two are recognized as alternative viewpoints of a single unified wholeness.

CONNECTING THE PARTS OF KNOWLEDGE
TO THE WHOLENESS OF KNOWLEDGE

1. A Binary Search Tree can be used to maintain data in sorted order more efficiently than is possible using any kind of list. Average case running time for insertions and searchers is O(log n).

6. In a Binary Search Tree that does not incorporate procedures to maintain balance, insertions, deletions and searches all have a worst-case running time of $\Omega(n)$. By incorporating balance conditions, the worst case can be improved to O(log n).

7. *Transcendental Consciousness* is the field of perfect balance. All differences have Transcendental Consciousness as their common source.

8. *Impulses Within The Transcendental Field.* The sequential unfoldment that occurs within pure consciousness and that lies at the basis of creation proceeds in such a way that each new expression remains fully connected to its source. In this way, the balance between the competing emerging forces is maintained.

9. *Wholeness Moving Within Itself.* In Unity Consciousness, balance between inner and outer has reached such a state of completion that the two are recognized as alternative viewpoints of a single unified wholeness.