Lesson 10
Binary Search Trees

Wholeness of the Lesson

Binary Search Trees arose as a natural solution to the need for incorporating efficient insertion and deletion capabilities of linked lists with the support provided for fast sorting and binary search of sorted elements available in arrays and array lists. Expansion from a linear structure to a two-dimensional structure makes a solution of this kind possible. Likewise, any problem becomes easier to solve if one can transcend the boundaries of the problem.

A List Wish-List

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

   B. If we are using a LinkedList, it is easy to maintain sorted order since insertions are efficient, but searches are not very efficient.

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 (binary search is highly efficient in an array list when elements are sorted)

A Solution: Binary Search Trees

1. A *binary tree* is a generalization of a linked list. It has a *root node* (analogous to the zeroth 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:

   <u>BST Rule:</u>
   A*t 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.*

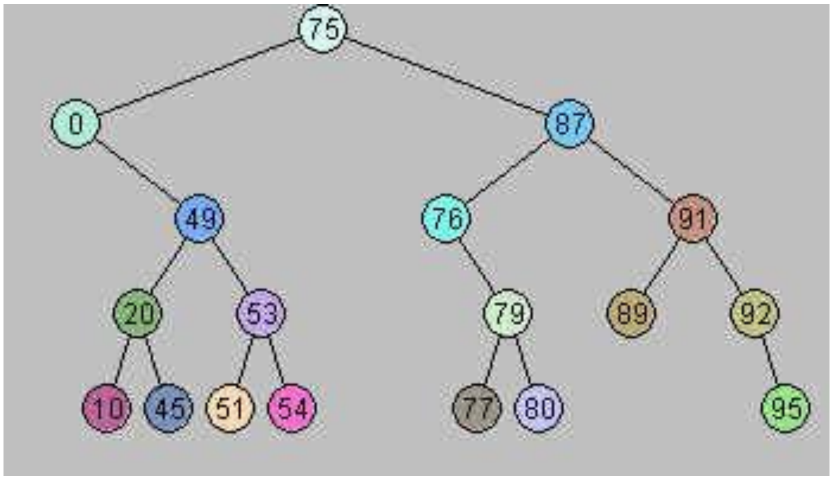
   For the moment, we assume all values are of type Integer.

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



Recursive 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. Otherwise, if x is less than the value in the root, insert x into the left subtree; if x is bigger than the value in the root, insert x into the right subtree

Recursive algorithm for finding an Integer x

The recursive find operation for BSTs is in reality the binary search algorithm in the context of BSTs.

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

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

```
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 BinaryNode(x, null, null);
      return;
   }
   BinaryNode n = root;
   boolean inserted = false;
   while(!inserted){
      if(x.compareTo(n.element)<0) {
         //available spot found on the left
         if(n.left == null){
            n.left = new BinaryNode(x,null,null);
            inserted = true;
         }
         //keep looking
         else {
            n = n.left;
         }
      }
      else if(x.compareTo(n.element)>0){
         //available spot 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
  }
}
```

Handling Duplicates in a BST

1. The algorithms and implementations above assume no duplicate values are being inserted into the BST.

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

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 BinaryNode. 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. How good is this new sorting algorithm? In fact, it is faster than MinSort for lists of 100 elements or more

BSTs in the Java Libraries

1. Java uses a special kind of BST (a "balanced" tree) as a background data structure for several of their data structures – namely `TreeSet` and `TreeMap`.

2. If a BST becomes unbalanced, its performance degrades dramatically; techniques have been developed to keep a tree from slipping into an unbalanced condition – the most popular such technique produces *red-black trees*.

MAIN POINT

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

> 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*.

BSTs provide efficient search, insert, and remove operations on orderable data. A binary search tree is an example of the principle of Diving: Because the structure is right, the basic operations are accomplished with maximum efficiency.

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Fundamental patterns of consciousness in the realm of binary trees

1. Binary search trees support *insert*, *remove,* and *find* operations with efficient performance, as well as efficient support for accessing elements in order, such as *findMin*, *findMax*, and finding elements in a specified range.
2. The structure of complete binary trees involves an expansion from 1 to 2 to 4 to 8, and eventually to 64 (mirroring to a large extent the significant numbers that mark the progress of unfoldment within the Ved, from A to AK to the fourfold Rishi, Devata, Chhandas, Samhita, to 8-fold prakriti through the first Richa to the 64 fundamental impulses that structure the first 192 syllables of Rk Ved).

3. **Transcendental Consciousness**: TC is the home of all the impulses of natural law, of creative intelligence.
4. **Wholeness moving within Itself**: In Unity Consciousness, the emergence the structure of pure knowledge as appreciated as a self-referral activity of consciousness interacting with itself.