Lesson 7
A Review of Data Structures:
Fully Developing the Container of Knowledge

Wholeness of the Lesson

An analysis of the average-case and worst-case running times of many familiar data structures (for instance, array lists, linked lists, stacks, queues, hashtables, binary search trees) highlights their strengths and potential weaknesses; clarifies which data structures should be used for different purposes; and points to aspects of their performance that could potentially be improved. Likewise, finer levels of intelligence are more expanded but at the same time more discriminating. For this reason, action that arises from a higher level of consciousness spontaneously computes the best path for success and fulfillment.

Abstract Data Types

1. Definition: A set of objects together with a set of operations

2. Example: the Integer data type consists of a range of integers together with the operations of addition, multiplication, and many others

The LIST ADT

1. The set of objects is any ordered list $A_1, A_2, \ldots, A_n$

2. Typical operations:

find(Object o)	returns position of first occurrence
findKth(int pos)	returns element based on index
insert(Object o, int pos)	inserts object into specified position
remove(Object o)	removes first occurrence of object
printList()	outputs all elements
makeEmpty()	empties the List

3. Other specialized operations are often considered, like *findMin*, *findMax*, *contains*.

4. Can be implemented in more than one way.

Array Implementation of LIST

1. In this implementation, an array is maintained internally in a List class and LIST operations are performed by internally carrying out array operations.

2. ***Advantage:*** Provides fastest O(1) implementation of *findKth*

3. ***Disadvantages:***

    - *insert* and *remove:* the average cost of inserting an element in an array list in which background array is not full is n/2. (running time: O(*n*))

    - periodic array resizing may be necessary

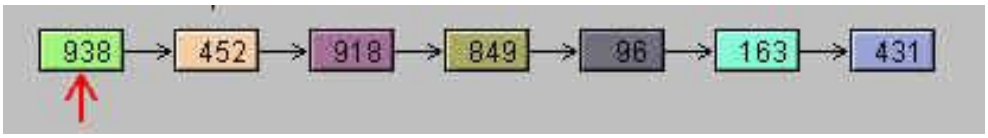
4. Other running times:

    - *makeEmpty, printList, find* are all O(*n*)

5. Examples in Java: `ArrayList` and `Vector` are array-based implementations of LIST. Both implement the `RandomAccess` interface

LinkedList Implementation of LIST

1. The Need: Improve performance of *insert, remove* and avoid the cost of resizing incurred by the array implementation.



2. A LinkedList consists of Nodes. In addition to storing an Object, a Node contains a link to the next Node (which may be Null). For doubly linked lists, a Node also contains a link to the previous Node.

3. ***Operations***

   - *find* requires traversing the Nodes via links, starting at the first Node.
   - *insert* requires traversing the Nodes to locate position and adjusting links
   - *remove* requires doing a *find*, and when the object is found, the *previous* object has to be located so that it can be linked to the *next* object

Running Times for LinkedList

1. *find* and *findKth* are O(*n*)

2. *insert* and *remove* are O(*n*), but are faster than in the array implementation because re-copying elements has been avoided

3. *printList* and *makeEmpty* are O(*n*)

When to Use What

1. When the *findKth* operation is used more often than *remove* and *insert*, an array-based implementation is a good idea because it makes use of the random-access feature of the array.

2. If the main search operation requires iteration through the entire List, and insertions and/or deletions are frequent, a LinkedList is preferable since insertions/deletions are faster and the *find* operation is essentially the same for both implementations.

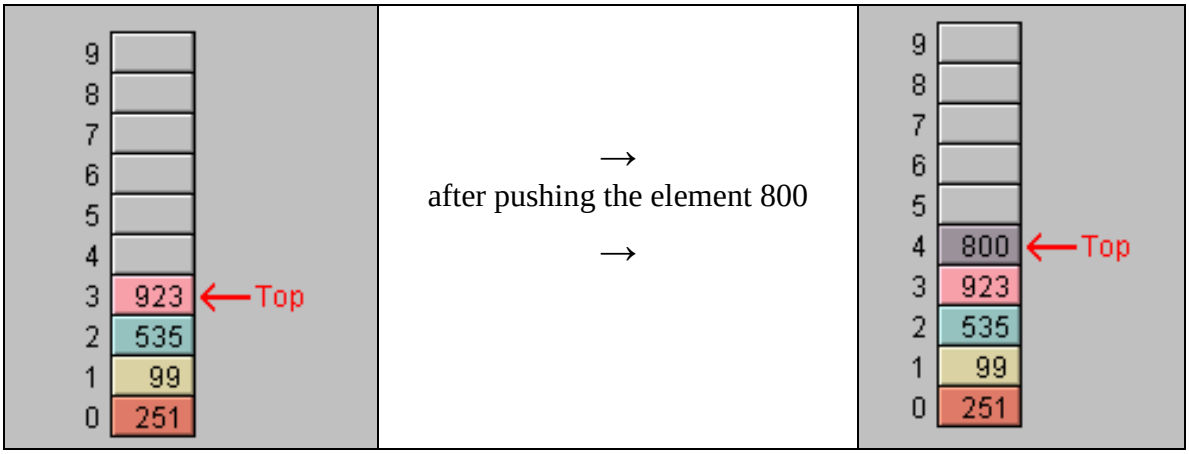
The STACK ADT

1. **Definition**: A STACK is a LIST in which insertions and deletions can occur relative to just one designated position (called the *top of the stack*).

2. **Operations**:

pop	remove top of the stack (usually also returns this object)
push	insert object as new top of stack
peek (or top)	view object at top of the stack without removing it

3. **Model:**



4. **Running Times**

   All operations, under a reasonable implementation, run in constant time – that is, O(1).

5. **Implementation**

   Stacks can be implemented using arrays (rightmost element is top) or linked lists.

Implementation of STACK in Java Libraries

1. The standard Java distribution comes with a `Stack` class, which is a subclass of `Vector`.

2. `Vector` is an array-based implementation of LIST. Therefore, for implementations that require many more pushes than pops, a stack based on a Linked List should be used.

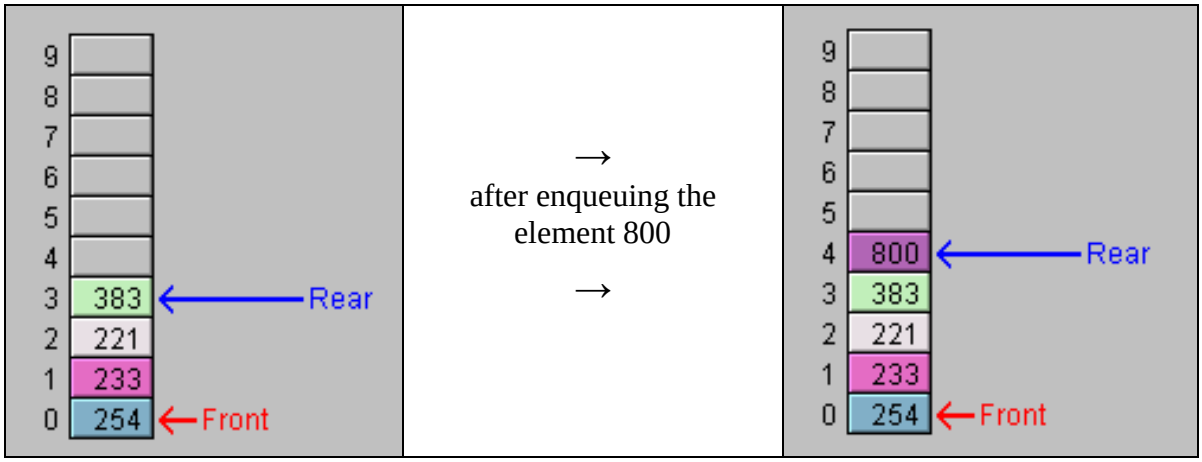
The QUEUE ADT

1. **Definition.** Like a STACK, a QUEUE is a specialized LIST in which insertions may occur only at a designated position (the *back*) and deletions may occur only at a designated position (the *front*).

2. **Operations**:

dequeue	remove the element at the front and return it
enqueue	insert object at the back
peek	view object at front of queue without removing it

3. **Model:**



4. **Running Times**

   All operations, under a reasonable implementation, run in constant time.

5. **Implementation of Queues**

   Queues can be implemented using an array or linked list. To avoid wasted space, in the array implementation, you can (using pointers) link last array slot to the first to provide a "circular array".

Java's Implementation

As of jdk1.5, Java provides a Queue interface that is implemented by LinkedList and several other collections classes.

Main Point

Array Lists provide O(1) performance for lookup by index because of random access provided by the background array, but perform insertions and deletions in $\Theta(n)$ time, with extra overhead because of the need to break the underlying array into pieces. Linked lists improve the performance of insertion and deletion steps to O(1), though locating the insertion point still requires $\Theta(n)$ time. On the other hand, linked lists, lacking random access, perform reads of all kinds in $\Theta(n)$ time. Stacks and queues achieve O(1) performance of their main operations, which involve either reading / removing the top element or inserting a new element either at the top or the end. Stacks and queues achieve their high level of efficiency by concentrating on a single point of input (top of stack or end of queue) and a single point of output (top of stack or front of queue).

Science of Consciousness: Wholeness contains within it diverse – even contradictory – values. These opposite values that are integrated within wholeness make wholeness a field of all possibilities. Experience of this wholeness value brings into our lives the ability to handle and make good use of opposite values.

Stacks and queues make use of the principle from Maharishi Vedic Science that the dynamism of creation arises in the concentration of dyamic intelligence to a point value ("collapse of infinity to a point”).

The HASHTABLE ADT

1. A **Hashtable** is a generalization of an array in which *any* object can be used as a key, not just just integers. A Hashtable has two main components - *keys* and *values*, and keys are used to look up corresponding values.

2. **Operations:**

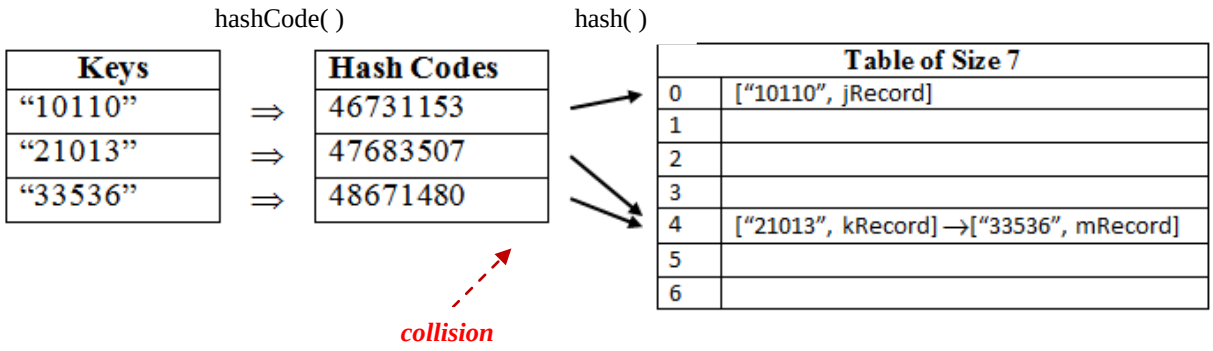
get	returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
put	maps the specified key to the specified value in this hashtable.
remove	removes the key (and its corresponding value) from this hashtable.

Hashing Strategy

1. In a Hashtable, values lying in a large range are compressed into a (typically) much smaller range consisting of array indices.

2. This transformation is accomplished in two Steps:
   a. create a *hashcode* for each key – typically, this is a big integer
   b. create a *hash value* for each hashcode – the hash value is an index in the underlying table for the Hashtable.

User's View of Hashtable	
Key (= Employee ID)	Value (= Record)
"10110"	jRecord
"21013"	kRecord
"33536"	mRecord



$$\text{hashcode(key)} = key.hashCode() \qquad \text{hash(hashcode)} = \text{hashcode} \,\%\, \text{tablesize}$$

Running Times for HashMap – The Worst Case

The worst case for a hashtable occurs when all keys are mapped to the same hash value – in that case, all values are stacked up (typically in a list) in a single slot. This results in $\Theta(n)$ performance for *find*, *insert*, and *remove* operations.

Example: Here, for typical hashtable implementations, all instances of `WeakHashCode` that are used as keys in a hashtable will be stacked up in slot #7 of the underlying hashtable table.

```
public class WeakHashCode {
    private String x = "" + Math.random();
    public String getX() {
        return x;
    }
    @Override
    public int hashCode() {
        return 7;
    }
}
```

Running Times for HashMap

There are several factors that contribute to the running time of hashtable operations:

1. Cost of computing hashCode().
2. Cost of computing hash values.
3. The degree to which these two computations minimize collisions

Cost of Computing hashCode()

1. The hashCode method that is used for classes in the Java libraries, when the choice of keys is reasonable, runs in O(1). Here are some examples of "reasonable" keys and how hashCode is computed for them:

    a. If `f` is Boolean, compute `(fvalue ? 1 : 0)` (where fvalue is `f.booleanValue()`)
    b. If `f` is a Byte, Character, Short, or Integer, compute `(int) fvalue.`
    c. If `f` is a Long, compute `(int) (fvalue ^ (fvalue >>> 32))`
    d. If `f` is a Float, compute `Float.floatToIntBits(fvalue)`
    e. If `f` is a Double, compute `Double.doubleToLongBits(fvalue)` which produces a long `f1`, then return `(int) (f1 ^ (f1 >>> 32))`

2. *Hashing Objects.* Good hashCode functions will create a hashCode based on the instance variables of the object (using the same data as is used to override equals). Here is an example of a scheme (described by J. Bloch) for combining the hashCodes of the instance variables; using schemes like these is standard practice:

    Suppose your class has instance variables u, v, w and corresponding hashCodes hash_u, hash_v, hash_w (if u, v or w is a primitive hash_u, hash_v, hash_w would be obtained as described above; otherwise if, say, u is an object of some kind, hash_u = u.hashCode()). Then:

```
@Override
public int hashCode() {
   int result = 17;
   result += 31 * result + hash_u;
   result += 31 * result + hash_v;
   result += 31 * result + hash_w;
   return result;
}
```

    If computing each of the hashCodes required only O(1) time, then running time to execute hashCode() for an instance of the object that has u, v, w as its instance variables is also O(1).

3. *Hashing Strings*. Here is how `hashCode` is overridden in the Java `String` class: (in the `String` class, the variable `value` stores `this.toCharArray()`).

```
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char val[] = value;

        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```

If all strings used as keys in a hashtable have length < m (for some fixed integer m), then the cost of computing each hashCode is O(m). This should be viewed as a fixed constant running time, therefore O(1).

4. *Cases in which hashCode is expensive.* Whenever keys for a hashtable have variable length without some clear upper bound, the cost of performing hashCode may end up being proportional to (or grow even faster than) the table size itself. In such cases, we cannot claim hashCode runs in O(1) time.

   Example: You could use Java Lists as keys in a hashtable. If you know for sure that the Lists that will be used as keys always have size less than a fixed number m, then hashCodes of these lists can still be computed in O(1) time. But if the Lists may end up containing arbitrarily many elements, then this is no longer the case.

Cost of Computing Hash Values

1. *Traditional computation.* Suppose you have an underlying table of size n, so that keys for your hashtable need to be mapped into one of the slots 0, 1, 2, . . ., n – 1. A generally effective method to compute hash values from hash codes is:

   ```
   hashCode  mod  n
   ```

2. *Java's approach.* For HashMap In Java, the underlying tablesize is set to be the smallest power of 2 that is greater than starting value (set by user or given by default). Hash values are computed in two steps: First a *mixing* step, followed by something like modding by n as in the traditional approach. These are also obviously O(1) operations.

   a. Mixing step

   ```
   static final int hash(Object key) {
       int h;
       return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
   }
   ```

   Here, the usual hashCode is mixed (using XOR) with its high-level bits. (This manages to destroy certain patterns that could skew the table.)

   b. Locating table index:

   ```
   index  = (n - 1) & hash
   ```

Cost of Handling Collisions

1. When the hash value function *h* maps two *hashCodes* to the same slot, we have a *collision.* When there are too many collisions, the hash table is said to be *skewed.* In the worst case, all *hashCodes* are mapped to the same slot. In order for hashtable operations to be O(1), collisions have to be handled well.

2. *Separate Chaining.* Separate chaining provides a way to handle collisions by placing in a list all key/value pairs for which the hash values associated with the key are equal: All key/value pairs belonging to the same slot are lined up in a list that lives in that slot.

3. *Need for Mixing*. To prevent the table from becoming skewed, keys need to be mapped to table slots so that each table slot stores approximately the same number of values as any other. This is achieved by "mixing up" data in the keys so that the possible array indices are all (approximately) equally likely to occur.

   Mixing may be introduced in two places

   A. In `hashCode` (recall Bloch's method; also, recall Java's `hashCode()` method for `String`)

   B. In computing the hash value (recall Java's technique does more than simply `hashCode % tableSize`)

4. *Load Factor*. Suppose the underlying table of a hashtable has *n* slots and we insert *m* elements. Then the *load factor* for the hashtable is given by

$$\alpha = m/n.$$

   A. Assuming input data has no patterns that would cause the table to be skewed, and assuming hashing distributes keys reasonably evenly, then

   The expected size of the list in each slot in the table is α.

   B. Under these assumptions, the running time of `put, get, remove`, and `containsKey` is always O(α).

   - `put`: O(1) to locate the slot (via hashing), O(α) to find the next available position in the list
   - `get`: O(1) to locate the slot (via hashing), O(α) to find the requested key in the list
   - `remove`: O(1) to locate the slot, O(α) to find the requested key in the list, O(1) to remove from the list (recall the list is a *linked* list)
   - `containsKey`: O(1) to locate the slot (via hashing), O(α) to find the requested key in the list

   C. When *m* is O(*n*) (that is, number of elements is not "too much" larger than `tableSize`), then α is O(1).

   D. *Rehashing.* If the number *m* of elements becomes significantly larger than the table size *n*, then the hashtable is no longer efficient. Java handles this by rehashing the hashtable whenever the load factor gets too big (the threshhold can be set by the user but defaults to 0.75).

Summary on Hashtables

1. It is accurate to assert that the hashtable operations `get, put, remove, containsKey` perform in O(1) time on average, under the following assumptions:
    a. Both `hashCode` and `hash` functions perform in O(1). This will be the case if keys are typical (numeric objects, Strings of bounded length, objects whose instance variables are not too complex)
    b. The number *m* of elements in the table is O(*n*), where *n* is the table size

    These assumptions are realistic as long as:
    A. The developer overrides `hashCode`( ) in a way that mixes up values adequately (e.g. using Bloch's method)
    B. The keys used in a hashtable are not too complex (avoiding using potentially long collection classes or strings whose lengths are not uniformly bounded)

Implementation in Java Libraries

In the Java Collections API, the classes `HashMap` and `Hashtable` are provided. The main differences between them are:

a. `HashMap` accepts `null` values while `Hashtable` does not
b. Methods in `Hashtable` are synchronized, but not in `HashMap`

MAIN POINT

Hashtables are a generalization of the concept of an array. They support (nearly) random access of table elements by looking up with a (possibly) non-integer key, and therefore their main operations have an average-case running time of O(1). Hashtables illustrate the principle of *Do less and accomplish more* by providing extremely fast implementation of the main List operations.

Binary Search Trees

1. Question: What is the best data structure to keep data in sorted order in memory?

2. 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 running time to carry out BinarySearch in a LinkedList?

3. *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

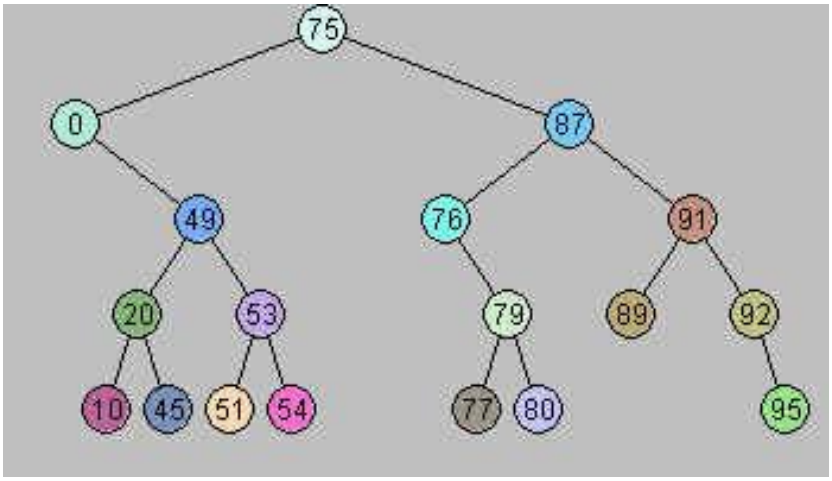
1. A *binary tree* is a generalization of a linked list. It has a *root node*. And each node has a reference to a left and right child node (though some references may be null).

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

   BST Rule:
   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 and that there are no duplicates.

A Binary Search Tree



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 BinarySearch 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.

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

1. If the root (of the tree being examined) is null, create a new root having value x

2. Else:
   - if x is less than the value in the root, examine the left subtree of the root
   - if x is bigger than the value in the root, examine the right subtree

3. Repeat steps 1, 2 until the condition in (1) is true and a new Node is added that stores x.

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 BSTs is in reality the binary search algorithm in the context of BSTs.

1. If the root (of the tree being examined) 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 searching 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 I or Case II

Sample Code

```
/** Assumption: Values inserted into the tree are distinct. For insertion
  * sequences that contain duplicates, a different implementation of BST is
  * usually 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.

Optional: 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. *Optional Outline of Proof:* 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)

Optional: Running Time of In-Order Traversal ("Print Tree")

Pseudo-Code

Algorithm: InOrderTraversal(*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* ≠ null **then**
InOrderTraversal(*left*)
print(*node*)
InOrderTraversal(*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. We note that
(c+d)n = (c+d)k + (c+d)(n-k-1) + (c+d).

It follows that for any k for which $0 \leq k \leq n$,

f(n) = (c + d)n + c
= [(c + d)k + c] + [(c + d)(n – k – 1) + c] + d
= f(k) + f(n – k – 1) + d

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.

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.

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.