Topics for Tuesday - 10/3/2000

1. Tree Algorithms - Ordered Storage
2. Implementations of Collections, Set and List Interfaces
3. WordCount example of use of HashSset.
4. The Map Interface
5. Implementations of the Map interface.
6. WordFrequency example use of HashMap
6. One implementation of a tree store/search algorithm.
7. Performance Comparisons for Trees and HashTables

Binary Tree -A Structure for Store and Search.

Problem Statement:

1. Accept and store many objects one at a time.
2. Each object to be stored and fetched has a unique ID or key and the keys have a natural order.
3. Store the objects such that a given object can be stored and retrieved in log(n) time where n is he number of items stored.
4. Store the objects in such a manner that the list of set of objects can be printed in key order.

Problem Context

1. Online systems where customers interact with sales clerks or fill out forms online.
2. Symbol table management for a compiler.

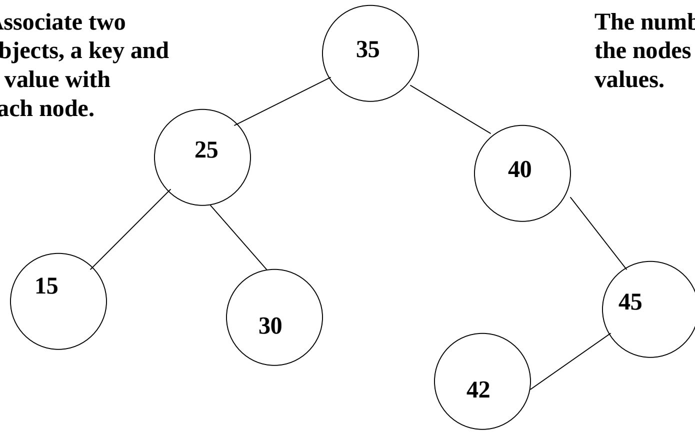
Hint - Remember the binary search algorithm for a sorted array of values.

Binary Tree -A Structure for Store and Search.

Consider a binary tree:

1. The root of the tree has two branches.
2. Each branch has two branches, etc.

Associate two objects, a key and a value with each node.



The numbers at the nodes are key values.

Binary Tree -A Structure for Store and Search.

Notation :

Root node - the root of the tree
subtree root - the root of a subtree
parent - a node with successor nodes
child - a node with a parent node
leaf node - a node with no successor nodes.

1. We can think of a sorted array, list or set list as forming a binary tree.
2.The root of the tree is the median value.
3. All of the values larger than the median are stored in subtrees to the right of the median .
4. All of the values less than the median are stored in subtrees to the left of the median.
4. The tree ends with leaf nodes which have no successors.

Binary Tree -A Structure for Store and Search.

What is the structure of a node in a binary tree?

Name of left node	**Name of object stored at node**	**Name of right node**

Therefore a TreeNode is an object where each instance of the object contains two other instances of the object.

Leaf nodes differ from other nodes by not having instances of left node and right node or by having these instances have the value null.

The "value" stored at a node is usually a unique id for an object instance. These unique id's are often referred to as "keys." In such a case the structure of the node is as follows.

Name of left node	**key**	**Object name**	**Name of right node**

Binary Tree -A Structure for Store and Search.

Search a binary tree for a given value.

1. **Examine the value at the root.**
2. **If (the search value is larger than the value at the root node)**
   continue the search in the node to the right of the root
   else continue the search in the node to the right of the root.
3. **Regard the node being searched as the root of a subtree and execute steps 2 and 3 for each subtree root.**

Cost of search - is $O(\log_2(n))$ compares if tree is balanced.

Balanced Tree - **The tree is symmetric on each side of the root.**
The left and right subtrees are of same size and depth.
Then the depth (number of levels) of the tree is $\log_2(n)$.
But if tree is not balanced then cost rises towards $O(n)$.

A tree will be balanced if the keys of the objects are uniformly distributed.

Uniformly distributed - There is equal probability of any key value occurring.

Binary Tree -A Structure for Store and Search.

Insert a value into a binary tree in conformance to the binary search rule.

1. If the tree is empty
   insert an item in the root node
2. Else if the current node's key matches the new item's key
   do not insert since this item is aduplicate
3. Else if the new items key is smaller than the current node's key
   insert the new item in the left subtree of the current node.
4. Else if the new item is larger than the current node's key
   insert the new item in the right subtree of the current node.

This algorithm will produce a balanced tree only if the keys are uniformly distributed. There are algorithms which will rebalance a tree and algorithms which improve balance

- AVL trees
- Red-Black trees,
- etc.

but these are beyond our scope for this course.

Binary Tree -A Structure for Store and Search.

Remove an item from a binary tree.

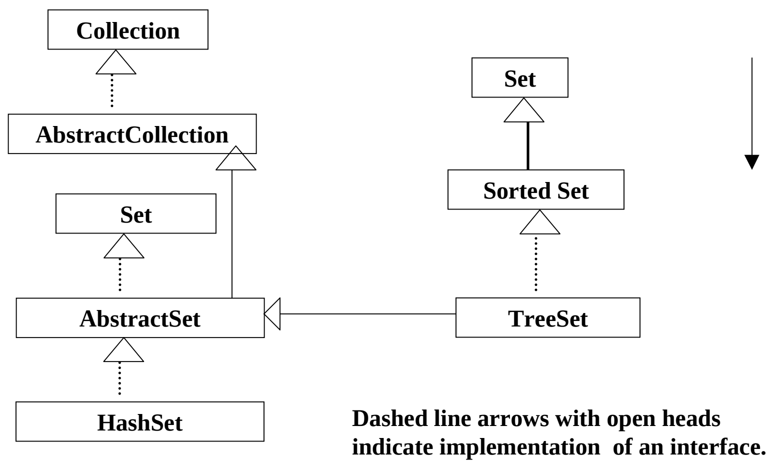
1. Find the item in the tree.
2. If the node containing the item is a leaf node
   delete that node and remove the reference to that node in its parent.
3. If the node containing the item is a parent node
   promote the left child to the parent position
      in the tree and
   connect the parent of the deleted item to its new
      child
   connect the promoted node to any children.
4. If the node containing the item to be removed is the root node
   promote the left child and connect it to the right subtree of the former root.

Cost is still $O(\log_2(n)$ but with a larger constant than insertion.

(Note - Removal is slightly more complex since there are a number of special cases but this is sufficient for our purposes.)

Collections Framework - Part 2

java.util Implementations of Collection and Set Interfaces

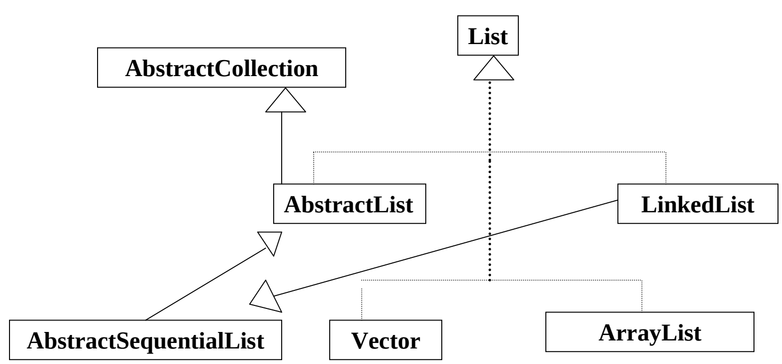


Dashed line arrows with open heads indicate implementation of an interface. Notice that Vector implements the List interface.

10/3/2000 Elements of Software - Fall 2000 9

Collections Framework - Part 2

Implementations of the List Interface



10/3/2000 Elements of Software - Fall 2000 10

public abstract class AbstractCollection extends Object implements Collection

This class provides a skeletal implementation of the Collection interface, to minimize the effort required to implement this interface.

To implement an unmodifiable collection, the programmer needs only to extend this class and provide implementations for the iterator and size methods. (The iterator returned by the iterator method must implement hasNext and next.)

To implement a modifiable collection, the programmer must additionally override this class's add method (which otherwise throws an UnsupportedOperationException), and the iterator returned by the iterator method must additionally implement its remove method.

The programmer should generally provide a void (no argument) and Collection constructor, as per the recommendation in the Collection interface specification.

The documentation for each non-abstract methods in this class describes its implementation in detail. Each of these methods may be overridden if the collection being implemented admits a more efficient implementation.

Method Summary for Abstract Collection

boolean add(Object o)
Ensures that this collection contains the specified element (optional operation).

boolean addAll(Collection c)
Adds all of the elements in the specified collection to this collection (optional operation).

void clear()
Removes all of the elements from this collection (optional operation).

boolean contains(Object o)
Returns true if this collection contains the specified element.

boolean containsAll(Collection c)
Returns true if this collection contains all of the elements in the specified collection.

Method Summary for Abstract Collection - Continued

boolean isEmpty()
Returns true if this collection contains no elements.
abstract Iterator iterator()
Returns an iterator over the elements contained in this collection.

boolean remove(Object o)
Removes a single instance of the specified element from this collection, if it is present (optional operation). boolean removeAll(Collection c)
Removes from this collection all of its elements that are contained in the specified collection (optional operation).
boolean retainAll(Collection c)
Retains only the elements in this collection that are contained in the specified collection (optional operation).
abstract int size()
Returns the number of elements in this collection.

Collections Framework - Part 2

Method Summary for Abstract Collection - Continued

Object[] toArray()
Returns an array containing all of the elements in this collection.

Object[] toArray(Object[] a)
Returns an array with a runtime type is that of the specified array and that contains all of the elements in this collection.

String toString()
Returns a string representation of this collection.

public abstract class AbstractSet
extends <u>AbstractCollection</u>
implements <u>Set</u>

This class provides a skeletal implementation of the `Set` interface to minimize the effort required to implement this interface.

The process of implementing a set by extending this class is identical to that of implementing a Collection by extending AbstractCollection, except that all of the methods and constructors in subclasses of this class must obey the additional constraints imposed by the `Set` interface (for instance, the add method must not permit addition of multiple intances of an object to a set).

Note that this class does not override any of the implementations from the `AbstractCollection` class. It merely adds implementations for equals and hashCode.

Method Summary for AbstractSet

`boolean equals(Object o)`
Compares the specified object with this set for equality.

`int hashCode()`
Returns the hash code value for this set.

public abstract class AbstractList extends AbstractCollection implements List

This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array). For sequential access data (such as a linked list), AbstractSequentialList should be used in preference to this class.

To implement an unmodifiable list, the programmer needs only to extend this class and provide implementations for the get(int index) and size() methods.

To implement a modifiable list, the programmer must additionally override the set(int index, Object element) method (which otherwise throws an UnsupportedOperationException. If the list is variable-size the programmer must additionally override the add(int index, Object element) and remove(int index) methods.

The programmer should generally provide a void (no argument) and collection constructor, as per the recommendation in the Collection interface specification.

Unlike the other abstract collection implementations, the programmer does not have to provide an iterator implementation; the iterator and list iterator are implemented by this class, on top the "random access" methods: get(int index), set(int index, Object element), set(int index, Object element), add(int index, Object element) and remove(int index).

Method Summary for AbstractList

```
void add(int index, Object element)
Inserts the specified element at the specified position in
this list (optional operation).

boolean add(Object o)
Appends the specified element to the end of this List
(optional operation).

boolean addAll(int index, Collection c)
Inserts all of the elements in the specified collection into
this list at the specified position (optional operation).

void clear()
Removes all of the elements from this collection (optional
operation).

boolean equals(Object o)
Compares the specified object with this list for equality.

abstract Object get(int index)
Returns the element at the specified position in this list.
```

Method Summary for AbstractList

int hashCode()
Returns the hash code value for this list. int indexOf(Object o)
Returns the index in this list of the first occurence of the specified element, or -1 if the list does not contain this element.

Iterator iterator()
Returns an iterator over the elements in this list in proper sequence.

int lastIndexOf(Object o)
Returns the index in this list of the last occurence of the specified element, or -1 if the list does not contain this element.

ListIterator listIterator()
Returns an iterator of the elements in this list (in proper sequence).

Method Summary for AbstractList - Continued

ListIterator listIterator(int index)
Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position in the list.

Object remove(int index)
Removes the element at the specified position in this list (optional operation).

protected void removeRange(int fromIndex, int toIndex)
Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.

Object set(int index, Object element)
Replaces the element at the specified position in this list with the specified element (optional operation).

List subList(int fromIndex, int toIndex)
Returns a view of the portion of this list between fromIndex, inclusive, and toIndex, exclusive.

public class HashSet
extends <u>AbstractSet</u>
implements <u>Set</u>, <u>Cloneable</u>, <u>Serializable</u>

This class implements the `Set` interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element.

This class offers constant time performance for the basic operations (`add`, remove, contains and size), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of the backing HashMap instance (the number of buckets). Thus, it's very important not to set the intial capacity too high (or the load factor too low) if iteration performance is important.

Note that this implementation is not synchronized. If multiple threads access a set concurrently, and at least one of the threads modifies the set, it *must* be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be "wrapped" using the `Collections.synchronizedSet` method. This is best done at creation time, to prevent accidental unsynchronized access to the HashSet instance:

```
     Set s = Collections.synchronizedSet(new
HashSet(...));
```

The iterators returned by this class's `iterator` method are *fail-fast*: if the set is modified at any time after the iterator is created, in any way except through the iterator's own remove method, the Iterator throws a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Method Summary for HashSet

```
boolean      add(Object o)
Adds the specified element to this set if it is not
already present.
void   clear()
Removes all of the elements from this set.
Object clone()
Returns a shallow copy of this HashSet instance: the
elements themselves are not cloned.
boolean      contains(Object o)
Returns true if this set contains the specified element.
boolean      isEmpty()
Returns true if this set contains no elements.
Iterator     iterator()
Returns an iterator over the elements in this set.
boolean      remove(Object o)
Removes the given element from this set if it is present.
int    size()
Returns the number of elements in this set (its
cardinality).
```



```
import java.io.*;
public class CountWords {
 static public void main(String[] args) {
  HashSet words = new HashSet();
  int count = 0;
  try  {
  BufferedReader in = new BufferedReader(new FileReader(args[0]));
  String delim = " \t\n.,:;?!-/()[]\"\’";
  String line;
   while ((line = in.readLine()) != null) {
    StringTokenizer st = new StringTokenizer(line, delim);
    while (st.hasMoreTokens()) {
     count++;
     words.add(st.nextToken().toLowerCase());
    }
   }
  } catch (FileNotFoundException e1) {}
   catch (IOException e2) {}
  System.out.println("Total number of words: " + count);
  System.out.println("Number of different words: " + words.size());
 }
}
```

What is a Map?

Map - A "set" of (key,value) pairs.

A key is an instance of a class and duplicate values of keys are not allowed.

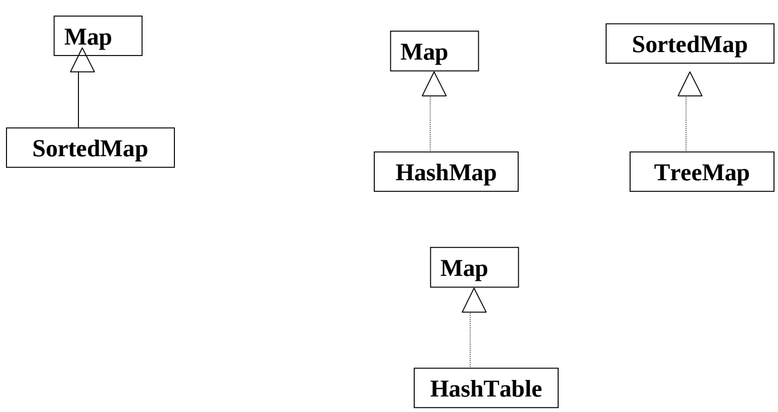
A value is an instance of a class. A given value may be associated with multiple keys.

The values are accessed through the keys.

Maps are frequently used where the value objects are large but have a subset of their attributes which generate a unique key.

Collections Framework - Part 2

Map Interfaces **Map Interface Implementations**



Collections Framework - Part 2

public interface Map

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

This interface takes the place of the `Dictionary` class, which was a totally abstract class rather than an interface.

The `Map` interface provides three *collection views*, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the `TreeMap` class, make specific guarantees as to their order; others, like the `HashMap` class, do not.

Collections Framework - Part 2

Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the equals and hashCode methods are no longer well defined on a such a map.

All general-purpose map implementation classes should provide two "standard" constructors: a void (no arguments) constructor which creates an empty map, and a constructor with a single argument of type `Map`, which creates a new map with the same key-value mappings as its argument. In effect, the latter constructor allows the user to copy any map, producing an equivalent map of the desired class. There is no way to enforce this recommendation (as interfaces cannot contain constructors) but all of the general-purpose map implementations in the JDK comply.

Method Summary for Map Interface - Continued

void clear()
Removes all mappings from this map (optional operatio
boolean containsKey(Object key)
Returns true if this map contains a mapping for the specified key.
Boolean containsValue(Object value)
Returns true if this map maps one or more keys to the specified value.
Set entrySet()
Returns a set view of the mappings contained in this map.
boolean equals(Object o)
Compares the specified object with this map for equality.
Object get(Object key)
Returns the value to which this map maps the specified key.
Int hashCode()
Returns the hash code value for this map.

Boolean isEmpty()
Returns true if this map contains no key-value mappings
Set keySet()
Returns a set view of the keys contained in this map.
Object put(Object key, Object value)
Associates the specified value with the specified key in this map (optional operation).
Void putAll(Map t)
Copies all of the mappings from the specified map to this map (optional operation).
Object remove(Object key)
Removes the mapping for this key from this map if present (optional operation).
int size()
Returns the number of key-value mappings in this map.
Collection values()
Returns a collection view of the values contained in this map.

Collections Framework - Part 2

public class HashMap extends AbstractMap implements Map, Cloneable, Serializable

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the intial capacity too high (or the load factor too low) if iteration performance is important.

Collections Framework - Part 2

An instance of HashMap has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the capacity is roughly doubled by calling the rehash method.

As a general rule, te default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.

If many mappings are to be stored in a `HashMap` instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table.

Note that this implementation is not synchronized. If multiple threads access this map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the `Collections.synchronizedMap` method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

```
Map m = Collections.synchronizedMap(new HashMap(...));
```


The iterators returned by all of this class's "collection view methods" are *fail-fast*: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Method Summary for HashMap

```
void clear()
Removes all mappings from this map.

Object clone()
Returns a shallow copy of this HashMap instance: the
keys and values themselves are not cloned.

boolean containsKey(Object key)
Returns true if this map contains a mapping for the
specified key.

boolean containsValue(Object value)
Returns true if this map maps one or more keys to the
specified value.

Set entrySet()
Returns a collection view of the mappings contained in
this map.
```



```
Object get(Object key)
Returns the value to which this map maps the specified key.
boolean isEmpty()
Returns true if this map contains no key-value mappings.
Set keySet()
Returns a set view of the keys contained in this map. Object
put(Object key, Object value)
Associates the specified value with the specified key in
this map.
void putAll(Map t)
Copies all of the mappings from the specified map to this
one.
Object remove(Object key)
Removes the mapping for this key from this map if present.
int size()
Returns the number of key-value mappings in this map.
Collection values()
Returns a collection view of the values contained in this
map.
```

Collections Framework - Part 2

```
import java.util.*;
import java.io.*;
public class WordFrequency {
  static public void main(String[] args) {
    HashMap words = new HashMap();
    Count count;
    String delim = " \t\n.,:;?!-/()[]\"\’";
    try  {
   BufferedReader in = new BufferedReader(new FileReader(args[0]));
    String line, word;
      while ((line = in.readLine()) != null) {
      StringTokenizer st = new StringTokenizer(line, delim);
      while (st.hasMoreTokens()) {
        word = st.nextToken().toLowerCase();
        count = (Count) words.get(word);
        if (count == null) {
          words.put(word, new Count(word, 1));
        } else { count.i++; }
     }
    }
   }
```


Collections Framework - Part 2

```
    catch (FileNOtFoundException e1) {}
    catch (IOException e2) {}
   Set set = words.entrySet();
   Iterator iter = set.iterator();
   while (iter.hasNext()) {
     Map.Entry entry = (Map.Entry) iter.next();
     word = (String) entry.getKey();
     count = (Count) entry.getValue();
     System.out.println(word +
                                            (word.length() < 8 ? "\t\t" : "\t") +
                                            count.i);
   }
 }
```

```
static class Count {
  Count(String word, int i) {
   this.word = word;
   this.i = i;
  }

  String word;
  int i;
 }
}
```



```
public class TreeNode
{
        // Data Fields
        Object info;
        TreeNode left;
        TreeNode right;
        // Methods
        // Constructors
        public TreeNode() {};
        public TreeNode(Object i)
                {
                info = i;
                left = null;
                right = null;
                }
```

```
        public boolean insertNode(Object item)
        {  if (item.toString().compareTo(info.toString()) ==0)
                return false;
           else if  (item.toString().compareTo(info.toString())<0)
                // target key < root key - check left subtree
                if (left == null)
                        { left = new TreeNode(item);
                        // Insert item.
                        return true;
                        }
                else  // Insert in left subtree.
                        return left.insertNode(item);
                // target key > root key - check right subtree
                else
                if (right == null)
                        { right = new TreeNode(item);  // Insert item.
                        return true; }
                else  // Insert in right subtree.
                        return right.insertNode(item);
        }
}
```



```
public class SearchTree
{
// Data fields
        private TreeNode root;
//Methods
        public boolean insert(Object item)
        /* Inserts item into a binary search tree.
        precondition : none
        postcondition: Returns true if the insertion is performed.
        Returns false if there is a node with the same key as item. */
        { if (isEmpty())
        {       // empty search tree
                root = new TreeNode(item); // Insert item in root.
                return true;
        }
        else  // Find insertion point and insert item.
                return root.insertNode(item);
        }
        public boolean isEmpty()
        { return (root == null);
        }
```

Collections Framework - Part 2

```
public boolean search(Object target)
/* Searches for the tree item that matches target
precondition : none.
postcondition: Returns true if target's key is found otherwise, returns false.*/
        {return search(root, target);}
private boolean search(TreeNode root, Object target)
{       if (isEmpty()) return false; // Tree is empty.
        else if (target.toString().compareTo(root.info.toString())== 0)
                return true;  // Target is found.
        else if (target.toString().compareTo(root.info.toString())< 0)
        // target key < root key - check left subtree
        if (root.left == null)
                return false;  // Left subtree empty.
        else  // Search left.
                return search(root.left, target);
        // target key > root key - check right subtree
        else if  (root.right == null)
                return false; //Right subtree empty.
        else  // Search right.
                return search(root.right, target);       }
```


Collections Framework - Part 2

```
        /* Displays a binary search tree in key order.
        precondition : none
        postcondition: Displays each node in key order.            */
        public void display()
        { display(root);     }
        private void display(TreeNode root)
        {if (!isEmpty())
        {
                // Display left subtree.
                if (root.left != null)
                display(root.left);
                // Display root key.
                displayResult((root.info).toString());
                // Display right subtree.
                if (root.right != null)
                display(root.right);}
        }
}
```

```
public class TreeTest
{ public static void main (String[] args)
        { SearchTree t = new SearchTree();
        // Insert 12 names and nicknames in the tree
        t.insert("Richard");
        t.insert("Debbie");
        t.insert("Robin");
        t.insert("Chris");
        t.insert("Jacquie");
        t.insert("Jeff");
        t.insert("Dustin");
        t.insert("Richie");
        t.insert("Deborah");
        t.insert("Jeffrey");
        t.insert("Jacqueline");
        t.insert("Christopher");
        // Display the tree contents.
        t.display();
        }
}
```



```
import java.util.*;
public class TreePerformanceTest
{       public static void main (String[] args)
        {
        TreeMap t = new TreeMap();
        int n = 1;
        int i;
        Object present;
        while (n > 0 )
        {System.out.println(" Enter Number of items to be stored");
        n = SavitchIn.readLineInt();
        RandomStringGenerator RandomKey = new RandomStringGenerator(8);
         // fill the hashtable with random values
        long startTime = System.currentTimeMillis();
        for (i = 0; i < n; i++)
        {t.insert(RandomKey.generateString()); }
        long endTime = System.currentTimeMillis();
        long runTime = endTime - startTime;
        System.out.println("Store time in milliseconds = " + runTime);
        float runTimePerItem = (float)runTime/(float)n;
        System.out.println(" Store time per item = " + runTimePerItem);
```

Collections Framework - Part 2

```
        startTime = System.currentTimeMillis();
        for (i = 0; i < n; i++)
        { present = t.search(t.root,(Object)RandomKey.generateString());}
        endTime = System.currentTimeMillis();
        runTime = endTime - startTime;
        System.out.println("Fetch time in milliseconds = " + runTime);
        runTimePerItem = (float)runTime/(float)n;
        System.out.println(" Fetch time per item = " + runTimePerItem);
        System.out.println(" Enter Number of items to be stored");
        t.clear();}
    }

  }
```


Collections Framework - Part 2

```
import java.util.*;
import corejava.*;
import structure.*;
public class HashPerformance
{   public static void main(String[] args)
  {        int n = 1;
           while (n > 0)
           {System.out.println(" Enter Number of items to be stored");
           n = SavitchIn.readLineInt();
           Hashtable a = new Hashtable();
           int i;
           Object Value;
           Object Key;
           RandomStringGenerator RandomKey = new RandomStringGenerator(6);
           RandomStringGenerator RandomValue = new RandomStringGenerator(20);
           // fill the hashtable with random values
```

Collections Framework - Part 2

```
long startTime = System.currentTimeMillis();
for (i = 0; i < n; i++)
{Value = a.put((Object)RandomKey.generateString(),
            (Object) RandomValue.generateString());
}
            long endTime = System.currentTimeMillis();
long runTime = endTime - startTime;
System.out.println("Store time in milliseconds = " + runTime);
float runTimePerItem = (float)runTime/(float)n;
System.out.println(" Store time per item = " + runTimePerItem);
startTime = System.currentTimeMillis();
for (i = 0; i < a.size(); i++)
{ Value = a.get((Object)RandomKey.generateString());
}
```


Collections Framework - Part 2

```
          endTime = System.currentTimeMillis();
          runTime = endTime - startTime;
          System.out.println("Fetch time in milliseconds = " + runTime);
          runTimePerItem = (float)runTime/(float)n;
          System.out.println(" Fetch time per item = " + runTimePerItem);
          a.clear();
          }
   }
 }
```

java.util class Stack
java.lang.Object
```
java.lang.Object
  |
  +--java.util.AbstractCollection
        |
        +--java.util.AbstractList
              |
              +--java.util.Vector
                    |
                    +--java.util.Stack
```

public class Stack extends Vector

The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends class Vector with five operations that allow a vector to be treated as a stack. The usual push and pop operations are provided, as well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top.

When a stack is first created, it contains no items.

boolean
empty()
Tests if this stack is empty.
Object
peek()
Looks at the object at the top of this stack without removing it from the stack.
Object
pop()
Removes the object at the top of this stack and returns that object as the value of this function.
Object
push(Object item)
Pushes an item onto the top of this stack.
int
search(Object o)
Returns the 1-based position where an object is on this stack.

Application of a Stack - Maze Traversal

Maze - Structure of partially connected cells with Start cell and End cell connected by at least one path. Each cell has k neighboring cells.

Maze Traversal -

1. **Enter Start cell**
2. **Choose neighboring cell to enter.**
3. **Enter neighboring cell.**
4. **Continue to enter neighboring cells until a cell with no exits is encountered or until the End cell is encountered.**
5. **If a cell with no exits is encountered, go back one cell and take a different exit until either all exits have been taken and the End cell not reached or the End cell is reached.**
6. **Repeat 5 until either you have returned to the Start cell and exhausted all exits or you have reached the End cell.**

Stack automates path reversal.