Plan for the week

- **Review:**
  - Boggle
  - Coding standards and inheritance
- **Understand linked lists from the bottom up and top-down**
  - As clients of java.util.LinkedList
  - As POJOs
  - Using linked lists to leverage algorithmic improvements
- **Binary Trees**
  - Just a linked list with 2 pointers per node?
- **Self-referential structures and recursion**
  - Why recursion works well with linked-structures
- **Lab: Memoizing**
  - BoggleScore, SpreadingNews, and related APTs

Documentation

- **Standard identifiers**
  - i, j, k: integer loop counters
  - n, len, length: integer number of elements in collection
  - x, y: cartesian coordinates (integer or real)
  - head, current, last: references used to iterate over lists.
- **Variable name guidelines**
  - Use nouns that describe what value is being stored
  - Don’t reiterate the type involved
- **Comments for methods and classes**
  - Abstraction: What does it do?
  - Implementation: How does it do it?
- **Inline comments as needed**

What can an Object do (to itself)?

  - Look at java.lang.Object
  - What is this class? What is its purpose?

- **toString()**
  - Used to print (System.out.println) an object
  - overriding toString() useful in new classes
  - String concatenation: String s = "value "+ x;
  - Default is basically a pointer-value

What else can you do to an Object?

- **equals(Object o)**
  - Determines if guts of two objects are the same, must override, e.g., for using a.indexOf(o) in ArrayList a
  - Default is ==, pointer equality

- **hashCode()**
  - Hashes object (guts) to value for efficient lookup

- **If you're implementing a new class, to play nice with others you must**
  - Override equals and hashCode
  - Ensure that equal objects return same hashCode value
Objects and values

- Primitive variables are boxes
  - think memory location with value
- Object variables are labels that are put on boxes

  ```java
  String s = new String("genome");
  String t = new String("genome");
  if (s == t) { they label the same box }
  if (s.equals(t)) { contents of boxes the same }
  ``

What's in the boxes? "genome" is in the boxes

Objects, values, classes

- For primitive types: int, char, double, boolean
  - Variables have names and are themselves boxes (metaphorically)
  - Two int variables assigned 17 are equal with ==
- For object types: String, ArrayList, others
  - Variables have names and are labels for boxes
  - If no box assigned, created, then label applied to null
  - Can assign label to existing box (via another label)
  - Can create new box using built-in new
- Object types are references/pointers/labels to storage

Anatomy of a class

```java
public class Foo {
    private int mySize;
    private String myName;
    public Foo(){
        // what's needed?
    }
    public int getSize(){
        return mySize;
    }
    public double getArea(){
        double x;
        x = Math.sqrt(mySize);
        return x;
    }
}
```

What values for vars (variables) and ivars (instance variables)?

Tomato and Tomato, how to code

- `java.util.Collection and java.util.Collections`
  - one is an interface
    - add(), addAll(), remove(), removeAll(), clear()
    - toArray(), size(), iterator()
  - one is a collection of static methods
    - sort(), shuffle(), reverse(), max(), min()
    - frequency(), binarySearch(), indexOfSubList()

- `java.util.Arrays`
  - Also a collection of static methods
    - sort(), fill(), copyOf(), asList()
Methods, Interfaces, Inheritance

- A method by any other name would smell as sweet
  - Method in OO languages, functions, procedures in others
  - Parameters and return value: communication
    - Do objects or methods communicate?: OO v procedural

- Static: Math.sqrt, Character.toUpperCase, ...
  - Don’t belong to an object, invoked via class (clue above?)
  - Java API helpful here

- Interface: implement class with required, related methods
  - Map: HashMap, TreeMap
  - List: ArrayList, LinkedList, Stack, Vector

Interfaces continued

- In the beginning
  - Make code work, don’t worry about generalizing
  - But, if you write code using Map rather than TreeMap
    - Can swap in a new implementation, coded generally!

- Don’t know how to optimize: space or time
  - Facilitate change: use interface rather than concrete class
  - My DVD connects to my TV, regardless of brand, why?
  - How do you turn on a Nokia cell phone? Motorola? But!

- Interfaces facilitate code refactoring
  - Don’t add functionality, change speed or memory or ...

What does Object-Oriented mean?

- Very common method of organizing code
  - Design classes, which encapsulate state and behavior
  - Some classes can be similar to, but different from their parent class: inheritance
    - Super class, subclass
  - Inherit behavior, use as is or modify and use or both

- Complex to design a hierarchy of classes, but important
  - More of this in Compsci 108 or on-the-job training
  - We’re solving simple problems, not designing re-usable libraries

- Simple does not mean straight-forward!

Inheritance and Interfaces

- Interfaces provide method names and parameters
  - The method signature we can expect and thus use!
  - What can we do to an ArrayList? To a LinkedList?
  - What can we do to a Map or Set or PriorityQueue?
  - IAutoPlayer is an interface

- Abstract classes can implement core, duplicated code
  - If we can add one object to a [set,map,list], can we add an entire list of objects? java.util.AbstractCollection
  - If we can iterate can we remove? Convert to array? Obtain size?
  - AbstractAutoPlayer is an interface
Miscellany

- Inheritance and overriding
  - Inheritance is the process by which a Class assumes the properties of its Superclasses
  - An object checks its own methods before consulting the Superclass thus overriding the Superclass methods
  - Polymorphism: many classes respond to some common message.

- Access Control
  - public: accessible anywhere
  - private: accessible only within methods of this class
  - protected: accessible to this class or subclasses
  - No modifier: accessible within class and package

Contrast LinkedList and ArrayList

- See ISimpleList, SimpleLinkedList, SimpleArrayList
  - Meant to illustrate concepts, not industrial-strength
  - Very similar to industrial-strength, however

- ArrayList --- why is access \(O(1)\) or constant time?
  - Storage in memory is contiguous, all elements same size
  - Where is the 1\(^{st}\) element? 40\(^{th}\)? 360\(^{th}\)?
  - Doesn’t matter what’s in the ArrayList, everything is a pointer or a reference (what about null?)

What about LinkedList?

- Why is access of N\(^{th}\) element linear time?
- Why is adding to front constant-time \(O(1)\)?

Linked list applications continued

- If programming in C, there are no “growable-arrays”, so typically linked lists used when # elements in a collection varies, isn’t known, can’t be fixed at compile time
  - Could grow array, potentially expensive/wasteful especially if # elements is small.
  - Also need # elements in array, requires extra parameter
  - With linked list, one pointer used to access all the elements in a collection

- Simulation/modeling of DNA gene-splicing
  - Given list of millions of CGTA… for DNA strand, find locations where new DNA/gene can be spliced in
    - Remove target sequence, insert new sequence
Linked lists, CDT and ADT

- As an ADT
  - A list is empty, or contains an element and a list
  - ( ) or (x, (y, ( ) ) )

- As a picture

- As a CDT (concrete data type) pojo: plain old Java object
  ```java
  public class Node {
    String value;
    Node next;
    Node(String s, Node link) {
      value = s;
      next = link;
    }
  }
  // … declarations here
  Node list = null;
  while (scanner.hasNext()) {
    list = new Node(scanner.next(), list);
  }
  ```

Building linked lists

- Add words to the front of a list (draw a picture)
  - Create new node with next pointing to list, reset start of list

  ```java
  public class Node {
    String value;
    Node next;
    Node(String s, Node link) {
      value = s;
      next = link;
    }
  }
  // … declarations here
  Node list = null;
  while (scanner.hasNext()) {
    list = new Node(scanner.next(), list);
  }
  ```

Dissection of add-to-front

- List initially empty
- First node has first word
- Each new word causes new node to be created
  - New node added to front
- RHS of operator = completely evaluated before assignment

Standard list processing (iterative)

- Visit all nodes once, e.g., count them or process them

  ```java
  public int size(Node list) {
    int count = 0;
    while (list != null) {
      count++;
      list = list.next;
    }
    return count;
  }
  ```

- What changes in code if we generalize what process means?
  - Print nodes?
  - Append “s” to all strings in list?
Standard list processing (recursive)

- Visit all nodes once, e.g., count them
  
  ```java
  public int recsize(Node list) {
    if (list == null) return 0;
    return 1 + recsize(list.next);
  }
  ```

- Base case is almost always empty list: null pointer
  - Must return correct value, perform correct action
  - Recursive calls use this value/state to anchor recursion
  - Sometimes one node list also used, two "base" cases

- Recursive calls make progress towards base case
  - Almost always using list.next as argument

Recursion with pictures

- Counting recursively
  
  ```java
  int recsize(Node list) {
    if (list == null) return 0;
    return 1 + recsize(list.next);
  }
  ```

Recursion and linked lists

- Print nodes in reverse order
  - Print all but first node and...
    - Print first node before or after other printing?

  ```java
  public void print(Node list) {
    if (list != null) {
      System.out.println(list.info);
      print(list.next);
    }
  }
  ```

Binary Trees

- Linked lists: efficient insertion/deletion, inefficient search
  - ArrayList: search can be efficient, insertion/deletion not

- Binary trees: efficient insertion, deletion, and search
  - Trees used in many contexts, not just for searching, e.g., expression trees
  - Search in O(log n) like sorted array
  - Insertion/deletion O(1) like list, once location found!
  - Binary trees are inherently recursive, difficult to process trees non-recursively, but possible
    - Recursion never required, often makes coding simpler
From doubly-linked lists to binary trees

- Instead of using prev and next to point to a linear arrangement, use them to divide the universe in half
  - Similar to binary search, everything less goes left, everything greater goes right
- How do we search?
- How do we insert?

Basic tree definitions

- Binary tree is a structure:
  - empty
  - root node with left and right subtrees
- terminology: parent, children, leaf node, internal node, depth, height, path
  - link from node N to M then N is parent of M
  - M is child of N
  - leaf node has no children
  - internal node has 1 or 2 children
  - path is sequence of nodes, N₁, N₂, ... Nₖ
    - Nₖ is parent of Nₖ₊₁
    - sometimes edge instead of node
  - depth (level) of node: length of root-to-node path
    - level of root is 1 (measured in nodes)
  - height of node: length of longest node-to-leaf path
    - height of tree is height of root
- Trees can have many shapes: short/bushy, long/stringy
  - If height is h, how many nodes in tree?

A TreeNode by any other name...

- What does this look like?
  - What does the picture look like?

```java
public class TreeNode
{
    TreeNode left;
    TreeNode right;
    String info;
    TreeNode(String s,
             TreeNode llink, TreeNode rlink)
    {
        info = s;
        left = llink;
        right = rlink;
    }
}
```

Printing a search tree in order

- When is root printed?
  - After left subtree, before right subtree.

```java
void visit(TreeNode t)
{
    if (t != null) {
        visit(t.left);
        System.out.println(t.info);
        visit(t.right);
    }
}
```

- Inorder traversal
- Big-Oh?
Tree traversals

- Different traversals useful in different contexts
  - Inorder prints search tree in order
    - Visit left-subtree, process root, visit right-subtree
  - Preorder useful for reading/writing trees
    - Process root, visit left-subtree, visit right-subtree
  - Postorder useful for destroying trees
    - Visit left-subtree, visit right-subtree, process root

Tree functions

- Compute height of a tree, what is complexity?
  ```java
  int height(Tree root)
  {
    if (root == null) return 0;
    else {
      return 1 + Math.max(height(root.left), height(root.right));
    }
  }
  ```
- Modify function to compute number of nodes in a tree, does complexity change?
- What about computing number of leaf nodes?

Balanced Trees and Complexity

- A tree is height-balanced if
  - Left and right subtrees are height-balanced
  - Left and right heights differ by at most one

```java
boolean isBalanced(Tree root)
{
  if (root == null) return true;
  return isBalanced(root.left) && isBalanced(root.right) &&
         Math.abs(height(root.left) - height(root.right)) <= 1;
}
```

What is complexity?

- Assume trees are “balanced” in analyzing complexity
  - Roughly half the nodes in each subtree
  - Leads to easier analysis
- How to develop recurrence relation?
  - What is T(n)?
  - What other work is done?
- How to solve recurrence relation
  - Plug, expand, plug, expand, find pattern
  - A real proof requires induction to verify correctness