Sample Final, Version 1.5

This sample final contains the kinds of questions that will be on the actual final. It is much longer than the final will be. At least one question on the final will be all but identical to one here. The final is comprehensive: some questions may come from first half of the course. See the sample midterm for examples of such questions.

We will not post solutions to the sample final problems because, in our experience, people will be tempted to look at the solution before completely doing the problem. If anyone wishes to check their answers, they may give us a paper copy for review.

1 Definitions and Motivation

- What is a binary tree? If we process a binary tree in "pre order", which node is processed first? Last?
- What property makes a binary tree a binary search tree?
- Insert the following numbers into a binary search tree (in this order, without rebalancing):
  58, 37, 25, 89, 78, 42, 59, 38, 90, 55, 88, 36
- Give the pre-order, in-order, and post-order traversals of the resulting binary search tree from the previous question.
- Why is it hard to implement an iterator (internal or external) for a binary search tree, even before we consider remove, and why is it even harder once we consider remove? Explain by suggesting possible implementations.
- How does a hashtable get close to constant-time access?
- Compare and contrast linear probing and chained hashing. Which one has “graceful degradation” and why? (and what is it?) (Here we assume that we don’t don’t resize the array.)
- What about double hashing? How does it improve over linear probing? What extra requirements are there? Why?
- Why does a hash table with open-address hashing need a “no object” (zombie) value or a parallel array of booleans? Give an example of the problem that can happen if we have neither of these techniques? Which of the two techniques is more space efficient? Why?
- When writing an iterator for hash tables with chaining, many people find it awkward to implement remove since we didn’t have dummy nodes. Why do you think we don’t use dummy nodes for the buckets?
- When can the remove() method of an iterator be called? What element does it remove? Why have the restriction?
- Whenever a caller or client modifies a library collection class, all iterators are made invalid (with the exception of the iterator through which the change was made, in the case of remove). Give a reason for this rule in the case of an array list (using a dynamically sized array, in which an iterator has a pointer to the array) and in the case of list (using linked list nodes, in which case the iterator points to one or more list nodes).
- When is a linked list a better choice than a dynamic array to implement a container class?
- What is the adjacency matrix technique for implementing graphs? The edge-list technique?
- What is a “sparse” graph? What implementation technique for graphs is inefficient for sparse graphs? Explain.
- What is depth-first search? What is breadth-first search?
• When is it good to use `@SuppressWarnings("unchecked")` and when not?

• Describe the `mergesort` algorithm. What algorithmic technique does it use? When is this algorithm to be preferred over insertion sort?

• Repeat the same description for `quicksort`.

• What is XML? Why is it used? What is the difference between an element and an attribute?

• This question is not relevant to Spr ’18 Why is there a method called `addAttr` in `AbstractShape` which simply throws an exception? What design pattern is it a part of? What role does it play?

2 Reading Code

Read the solutions to Homeworks 1-13 and answer the following questions:

1. Homework #1
   (a) What does it mean for a class to be immutable?
   (b) Why should an immutable class override `equals`?
   (c) Why does `Particle#move()` assign position. Why not just
       `velocity.move(position)`;
   (d) What does it mean for the constructor to begin `this(toMidi(pitch),duration)`? What must be true about `toMidi`?

2. Homework #2
   (a) What is the connection (if any) between `_data.length` and `_manyItems`?
   (b) Why does `ParticleSeq.insert` call `ensureCapacity` every time it is called? Isn’t that inefficient?
   (c) Why does `removeCurrent` require a for-loop?
   (d) Why does `ManyParticles#move` clone the sequence of particles?

3. Homework #3
   (a) This Homework’s `ParticleCollection` class has the same data structure (basically) as that of Homework #2, but the interface is different. Describe the major change.
   (b) Why do we have versions on both the collection and on the iterator? Why does the iterator version not change when the iterator is changed (say in `next`)?
   (c) In `MyIterator.remove` there is the lines:
       `myVersion++;
       _version++;
   If both versions are the same, why do we need to keep separate copies and increment them in parallel. More concretely, what if one of these lines were omitted but not the other? What if both were omitted?

4. Homework #4
   (a) Does the `ParticleSeq` class have the same interface as either of the last homeworks? Data Structure?
   (b) Why does `_wellFormed` count up the nodes rather than just calling `size()`?
   (c) The invariant checker checks whether `precursor` is null or in the list. What would it mean to be non-null and not be in the list? How could that happen?
   (d) What does it mean if the precursor is the same as the cursor?
(e) `addBefore` has a condition (if). What aspects of the two branches give us more hope that the code is either correct or completely wrong?
(f) And why does `addAll` call `clone`?
(g) What if the “hard work” of clone is missing? What sort of test would notice the problem?

5. Homework #5

(a) How can `Key` objects return a number if it’s only an interface?
(b) Why don’t we simplify the code by getting rid of the “keys” and just ask a `key()` method to `Triangle`?
(c) Why is the `Group` class nested in the `Triangle` class?
(d) The invariant of the group says it is supposed to check that the tail is in the same list as the head. What does that mean? How is it checked?
(e) How can sorting be linear time? Even if one or two triangles are out of order?
(f) Why can’t we clear a group by simply setting `first` and `last` pointers to null?

6. Homework #6

(a) Why can’t we initialize the dummy to point to itself using:
   ```java
   _dummy = new Node((E)_dummy,_dummy,_dummy);
   ```
(b) `addAll` calls `addAfter` which calls `addBefore`. Is this mutual recursion? How do we know it doesn’t go on forever?
(c) `addAll` special cases the empty addend. Would the rest of the code work properly for this case? Explain!
(d) The dummy node is supposed to make the coding easier. How would `remove` have to be more complex, if we didn’t have a dummy node?
(e) In `clone()` why is it necessary to change `result._cursor`?

7. Homework #7

(a) The nested class is called `class Node<T>` and yet in the main class, we only ever say `Node<E>`. Why?
(b) In previous assignments we didn’t require javadoc on methods such as `isEmpty` in class `Stack`, but this time it was required. Why?
(c) Where in the PostScript Engine are methods such as `copy` called?
(d) Why does `index` pop lots of things off the stack? I though it was just supposed to pop of a number and push on a value?

8. Homework #8

(a) Why does `checkTree` return an integer?
(b) In `doPut` if `r` is null it assigns `r` to a new node. Trick question: how does that get the new node into the tree? What are the three places that a node can be assigned into the tree.
(c) Why does `doCopy` use pre-order recursion? Why does `doValues` use in-order recursion? What happened if they swapped their traversal choices?

9. Homework #9

(a) How does `doPut` handle the fact that we have a new field in our nodes? (the “parent”)
(b) Why does the `BSTSet` class have a comparator? Why not just use `compareTo` (or even `<`)?
(c) Explain the very complex “while” loop in method `next()`.
(d) When we need to promote a substitute when removing a node with both children set in doRemove, why do we get the immediate predecessor, not the immediate successor?

(e) On the other hand, explain the special case at the heart of the iterator’s remove method: why does it have to do the “unfortunate” move of the right subtree? What bad thing would happen if we got rid of the whole special case?

(f) What is _current used for in the iterator?

10. Homework #10

(a) There are two methods called get here. Why? Which one get called? If we have a variable n of type Name, how can we call each get method on n?

(b) Why does containsKey take an Object parameter? This class is a map from names to objects, so all keys will be names.

(c) Why do we have an if in clear? What would go wrong if this code was unconditional, always did the same thing?

(d) What does Node<?,?> mean? Why do we use this type?

(e) Both Dictionary.remove and EntrySet.remove take an object. Why is their code different?

(f) Why does the doRemove at one point increase the size field?

(g) In the iterator, what does the stack of _pending nodes represent?

(h) What does the iterator’s remove() set current to be null?

11. Homework #11

(a) How does the invariant avoid getting stuck in an infinite loop?

(b) Why doesn’t size() return the size of the array?

(c) Why does get check the hash code and the uri itself? Wouldn’t all the resources in the bucket have the same hash code anyway?

(d) Why do we have to use a while loop in the rehash? What would happen if we had for (; o != null; o = o.next) { ... }?

(e) Why is add removing things?

(f) Explain the iterator invariant. What are all the ways it finds a problem? Give a picture for each one and explain why it’s wrong.

(g) Why doesn’t remove need to use placeholders (“zombie” elements)?

12. Homework #12

(a) Trick Question: why do we use add in the constructor but offer in the crawl method when doing our breadth-first search?

(b) Semi-Trick Question: What does listener.accept(_) do?

(c) Why don’t we check if a URL is already visited before offering it to the queue?

(d) When can crawl return something other than zero?

(e) How is (11,12) -> 12.size() - 11.size() a comparator? What is it comparing?

(f) This code depends on the content hash being unique.
   i. What goes wrong if the content hash isn’t unique?
   ii. Why is it impossible for the context hash to be unique?
   iii. What is the difference between the content hash and the hash code used in the resource table?

The following questions still need to be updated for Spring 2018

13. Homework #13
(a) What does `out[lo++] = in[lo1++]` do?
(b) Why do we need two whiles loops even after the main while loop in `merge` is done?
(c) The output array is only supposed to be written in the range `[lo, hi)`. Why not start the output at 0?
(d) The `merge` method assumes that the two sequences of values to merge are next to each other in the array. How much this assumption is safe to make?
(e) Explain the mutual recursion. If `mergeSortKeep` and `mergeSortMove` were recursive in the normal way, what extra step would be required?
(f) The `difference` method requires that the input sequences are both sorted. What will happen if they are not sorted?
(g) What benefit is possible because they are sorted?
(h) Why doesn’t `unique` use `.equals(_)` to find out if two elements are equal?
(i) Why does SpellCheck.getWords do nothing for elements of type `script`?
(j) In SpellCheck what is `n` used for. Why wouldn’t it it be the number of words in the document being checked?

14. Homework #14 (will not be on final exam)

- In a binary search tree data structure (with parent) pointers, we implement an iterator:

```java
private class MyIterator implements Iterator<T> {

    private Node<T> previous, next;

    MyIterator() {
        next = root;
        if (next != null) {
            while (next.left != null) {
                next = next.left;
            }
        }
    }

    public boolean hasNext() {
        return next != null;
    }

    public T next() {
        if (next == null) throw new NoSuchElementException("at end of iteration");
        assert _wellFormed() : "invariant broken at beginning of next()";
        previous = next;
        if (next.right == null) {
            while (next.parent != null && next.parent.right == next) {
                next = next.parent;
            }
            next = next.parent;
        } else {
            next = next.right;
            while (next.left != null) {
                next = next.left;
            }
        }
    }
```
return previous.data;
}

Please answer the following questions about this code:

1. Why is the constructor doing lots of work, not just starting at the root?
2. What is the distinction between previous and next? Why do we return previous.data at the end of next(), not the more reasonable next.data?
3. What does the condition next.right == null signify about the iterator with respect to the tree rooted at “next”?
4. Why does the first branch go up one more time after the while loop is done?
5. If next.parent is null in the while loop of the first branch, what does that mean?
6. The right branch does something very similar to what elsewhere in this code? Why?

• Read the following code that finds a path by search through a directed graph:

```java
private Set<Node> visited = new HashSet<Node>();

/**
 * Initialize for a new search
 */
protected abstract void init();

/**
 * Add a search state to be considered later
 * @param s search state to add.
 */
protected abstract void add(SearchState s);

/**
 * Are there any more search states to consider?
 * @return whether there are any more search states
 */
protected abstract boolean hasNext();

/**
 * Return next search state to consider
 * @return
 */
protected abstract SearchState next();

public SearchState find(Node from, Node to) {
    visited.clear();
    init();
    add(new SearchState(from));
    while (hasNext()) {
        SearchState s = next();
        Node last = s.last();
        if (last == to) {
            return s;
        } else if (!visited.contains(last)) {
            visited.add(last);
            for (Edge edge : last.edges()) {
                add(s.extend(edge));
            }
        }
    }
```
public class SearchState implements Iterable<Edge> {
    private Node initial;
    private List<Edge> path;

    public SearchState(Node i) {
        initial = i;
        path = new ArrayList<Edge>();
    }

    public Node first() {
        return initial;
    }

    public Iterator<Edge> iterator() {
        return path.iterator();
    }

    /**
     * Return the last node on this path.
     * If the path is empty, we’re still in the initial node.
     * @return last node on path
     */
    public Node last() {
        ...
    }

    /**
     * Return a new search state which is like this one except
     * with a path extended by the given edge.
     * @param e edge to add to new state’s path.
     * @return new search state.
     */
    public SearchState extend(Edge e) {
        ...
    }
}

1. What role does visited play? What would happen if we moved the call to clear the visited set into the “while” loop?
2. How should add and next be implemented to achieve depth-first search? Breadth-first search? Why?

3 Writing Code

- Suppose we implement a Set with a sorted linked list. According to the comparator, the set should have no duplicates and the elements should be in increasing order. Write the code to check both properties, reporting different error messages for violations of the two properties. It should also return false if there is a cycle in the list. You may assume that the comparator is well-behaved.

```java
class SortedListSet<E> extends AbstractSet<E> {
    ...
private static class Node<T> {
    T data;
    Node<T> next;
}

private Node<E> _head;
private Comparator<E> _comp;

private boolean _report(String s) {
    System.out.println("Invariant Error: " + s);
    return false;
}

private boolean _wellFormed() {
    if (_comp == null) return _report("comparator is null");
    // {Your code here!}
    return true;
}

• Given a binary tree class called TreeSet that uses instances of a class called Node, write a method that writes the nodes on an output stream for debugging purposes. Your method should show the parent-child relationships in the tree.

class TreeSet <T extends Comparable<T>> {
    private static class Node<T> {
        T data;
        Node<T> left, right;
        Node(T d) {data = d; }
    }
    private Node<T> root;

    private void doPrint(Node n, int indent) {
        // {Your code here!}
    }

    print() { doPrint(root,0); }

    ...
};

It is your task to implement doPrint, a helper method for print. This member function should print the tree where each element is indented by two spaces more than the parent, and the data should be printed in order.

For example, in the following picture, the tree on the left should be printed in the manner shown on the right:

• Implement the Dictionary ADT with a single-linked list without a dummy node.
• Suppose we wished to implement a (directed) Graph ADT using edge lists. Nodes are implemented as (small) integers in the range \([0, n]\) where \(n\) is the number of nodes:

```java
public class Graph {
    private int numberOfNodes;
    private ArrayList<ArrayList<Integer>> adjacent;

    public Graph(int size){...}

    public int getNumberOfNodes() {...}
    public void addEdge(int from, int to){...}

    public Iterator<Integer> adjIterator(int node){...}
}
```

– This representation doesn’t give a way to indicate edge weight. Why not? What would have to be changed in order to do so?
– This data structure permits a node to have multiple edges to the same destination node. Explain how this could be changed by changing the data structure.
– Implement the ADT.
– Use the ADT to write a recursive depth-first search.

• A bank keeps track of its collection of customer records and collection of accounts with two nested classes Customer and Account classes:

```java
class Customer {
    public Customer(String name) {...}
    public void setAccount(Account a) {...}
    public Account getAccount() {...}
}
class Account {
    public Account(Money initial) {...}
    public void setBalance(Money b) {...}
    Money getBalance() {...}
}
```

For a new customer that does not yet have an account, both a customer record and an account get dynamically allocated, with an initial deposit of $100 (i.e., \(\text{Money}(100)\)). The customer record and the account must be added to their respective collections in the most convenient way.

Write the body of a function that creates a new customer and account (with 100 dollars) using the customer name passed as a string:

```java
Collection<Customer> customers;
Collection<Account> accounts;
...

Customer createCustomerAndInitialAccount(string name) {
    // {Your code here!}
}
```

• For the same Bank as above, write a `totalDeposits` function that computes the total amount of money in all the accounts. Assume that `Money` has a method:
void add(Money m);
Make sure you don’t modify any of the accounts!

Money totalDeposits()
{
    Money total = new Money(0);

    // Your code here!

    // System.out.println("The total amount is " + total);
    return total;
}

• How can an array be used to implement a Stack ADT? Please complete the class declaration, then give
  the member function implementations separately:

  class Stack<T> {
      T[] contents;
      int used;

      private T[] makeArray(int n) {
          contents = (T[])new Object[n];
      }

      public Stack() { ... }
      public boolean isEmpty() { ... }
      public void push(T x) { ... }
      public T pop() { ... }
  };

• Implement a map that uses an (unsorted) linked list of entries to keep the map data. Don’t worry
  about efficiency at all – just get the map working. You may assume the existence of MyEntry
  NB: You should use both AbstractMap and AbstractSet. (Why?)

• Implement a depth-first search of a graph using a recursive function. The result should print the nodes
  along the path. Use the following node class.

  /*
   * Each node permits iteration to the adjoining nodes.
   */
  class Node implements Iterable<Node> { 
      public String getName();
      ...
  };

  /*
   * Return whether a path can be found to the goal.
   * Return null if no path can be found.
   */
  List<Node> findPath(Node from, Node to, Set<Node> visited, List<Node> sofar) {
      ...
  }
• Implement a hash table with quadratic hashing.

• Write a routine to check the invariant of a hashtable using chained hashing. Assume the following data structure:

```java
class HashTable<K,V> ... {  
    static class HashEntry<K,V> ... {  
        K key;
        V value;
        HashEntry<K,V> next;
    };

    HashEntry<K,V>[] contents;
    int numEntries;

    int hash(K key) { ... } // return value in range [0,contents.length)
}
```

You can assume the existence of boolean _report(String s). You don’t need to check that there are no duplicates.

• Write a routine to check the invariant of a hashtable using linear probing. Assume the following data structure:

```java
class HashSet<K> ... {  
    K[] contents;
    int numEntries;

    K nullObject = (K) new Object(); // fill-in for null
    K noObject = (K) new Object(); // place holder for removed entry

    int hash(K key) { ... } // return value in range [0,contents.length)
}
```

NB: Make sure you handle the result of probing. You don’t need to check that there are no duplicates. Hint: you don’t need to treat nullObject specially. (Why not?) (NB: This question by itself is too hard for a final exam.)

• Implement insertion sort / merge-sort / quick-sort on an array or on a linked list.

4 Debugging

• We have written the following code to implement cloning of a Set implemented with binary search trees (with parent pointers):

```java
class Set<T> {  
    private static class Node<T> {  
        Node<T> parent;
        T data;
        Node<T> left,right;
        Node(Node<T> p, T d) {  
            parent = p;
            data = d;
        }
    }
```
private Comparator<T> comparator;
private Node<T> root;
private int numItems = 0;

...

@ SuppressWarnings("unchecked")
public Set<T> clone()
{
    Set<T> result = null;
    try
    {
        result = (Set<T>) super.clone();
    }
    catch (CloneNotSupportedException e)
    {
        throw new RuntimeException("This class does not implement Cloneable");
    }
    result.root = new Node<T>(null,result.data);
    result.root.left = result.left;
    result.root.right = result.right;

    return result;
}

This code works fine when the set has exactly one element, but crashes if it is empty, and strange
things sometimes seem to handle if there is more then one element. For instance the invariant might
fail suddenly at the start of a public method.

1. Why does this code crash if the set is empty?
2. Why doesn’t it work correctly when there is more than one node?
3. Give example client (outside) code which will cause an invariant error to happen. Explain why.

• The following series of classes was designed using inheritance:

    public class Person {
        private int id;
        private String firstName, lastName;

        public Person() {}

        public Person(int id, String firstName, String lastName) {
            this.id = id;
            this.firstName = firstName;

        }
```java
    this.lastName = lastName;
}

public String getFormattedName() {
    return firstName + " " + lastName;
}

public class Student extends Person {
    private double gpa;

    public Student(int id, String firstName, String lastName, double gpa) {
        this.gpa = gpa;
    }

    public double getGPA() {
        return gpa;
    }

    public class Teacher extends Person {
        private String title;
        private boolean isTenured;

        public Teacher(int id, String firstName, String lastName,
                        String title, boolean isTenured) {
            this.title = title;
            this.isTenured = isTenured;
        }

        public boolean getIsTenured() {
            return isTenured;
        }

        @Override
        public String getFormattedName() {
            return title + " " + firstName + " " + lastName; // error #1
        }
    }

    public class Main {

        public static void main(String[] args) {
            Teacher a = new Teacher(100, "Mary", "Jo", "Dr.", true);
            Person b = new Student(200, "John", "Doe", 3.8);

            printPersonInfo(a);
            printPersonInfo(b);
        }

        public static void printPersonInfo(Person p) {
            System.out.println(p.getFormattedName());
```
if(p instanceof Student) {
    System.out.println("GPA: "+ p.getGPA()); // error #2
} else if(p instanceof Teacher) {
    System.out.println("Tenured: "+ p.getIsTenured()); // error #3
}
}

1. We get the following compiler errors:
   error #1 The fields firstName and lastName are not visible.
   error #2 The method getGPA is undefined for the type Person.
   error #3 The method getIsTenured is undefined for the type Person.

   Explain each of these error messages.

2. Fix the compiler errors by only changing the lines marked with errors. Either edit the code neatly
   or place the fixes below:

3. After the compiler errors are fixed, the program runs and prints the following unexpected output:

   Dr. null null
   Tenured: true
   null null
   GPA: 3.8

   Why? Please explain!

4. Fix the problems.

   • The following code is supposed to implement remove for a BST implementation of Set, but it has a
     number of independent bugs:

   11 public class TreeSet<T> extends AbstractSet<T> {
   12    private static class Node<E> {
   13        E data; // never null
   14        Node<E> left, right;
   15        Node(E d) { data = d; }
   16    }
   ...
   83    private Comparator<T> _comparator; // never null
   84    private Node<T> _root;
   85    private int _size;
   86    private int _version;
   ...
   117    @Override
   118    public boolean remove(Object x) {
   119        assert _wellFormed() : "invariant broken at start of remove";
   120        T key = (T)x; // NB: Warning: unchecked cast
   121        Node<T> lag = _root;
   122        Node<T> n = _root;
   123        while (!key.equals(n.data)) {
   124            lag = n;
   125            if (_comparator.compare(key, n.data) < 0) {
   126                n = n.left;
   127            } else {
   128                n = n.right;
   129            }
   130            if (n == null) return false;
   131        }
   132        if (n.left == null && n.right == null) {
```java
    if (lag.left == n) lag.left = null;
    else lag.right = null;
    }
    else {
        Node<T> p = n.left;
        lag = n;
        while (p.right != null) {
            lag = p;
            p = p.right;
        }
        n.data = p.data;
        if (lag.left == p) lag.left = p.left;
        else lag.right = p.right;
    }
    --_size;
    ++_version;
    assert _wellFormed() : "invariant broken at the end of remove";
    return true;
}
...
```

(a) The code fails the following test:
```java
    public void test0() {
        assertFalse(c.remove("bagels"));
    }
```

The failure trace is:
```
java.lang.NullPointerException
at edu.uwm.cs351.TreeSet.remove(TreeSet.remove:123)
at TestOrderedSet.test0(TestOrderedSet.java:23)
```

Why does the code crash on this test? (Diagnose the problem.)

(b) The code fails the following test:
```java
    public void test1() {
        c.add("flour");
        assertFalse(c.remove("lettuce"));
        assertTrue(c.remove("flour"));
    }
```

The failure trace is:
```
java.lang.AssertionError: invariant broken at the end of remove
at edu.uwm.cs351.TreeSet.remove(TreeSet.remove:148)
at TestOrderedSet.test1(TestOrderedSet.java:29)
```

The console gets the following message
```
Invariant error found: _size is 0 but should be 1
```

Why does the code crash on this test? (Diagnose the problem.)

(c) The code fails the following test:
```java
    public void test2() {
        c.add("apples");
        c.add("bread");
        c.add("cheese");
        assertTrue(c.remove("bread"));
    }
```

The failure trace is:
java.lang.NullPointerException
    at edu.uwm.cs351.TreeSet.remove(TreeSet.remove:138)
    at TestOrderedSet.test2(TestOrderedSet.java:36)
Why does the code crash on this test? (Diagnose the problem.)

(d) The code fails the following test:
```java
39    public void testWeird() {
40        c.add("eggs");
41        assertFalse(c.remove(42));
42    }
```
The failure trace is:
```java
java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
    at java.lang.String$CaseInsensitiveComparator.compare(String.java:1178)
    at edu.uwm.cs351.TreeSet.remove(TreeSet.remove:125)
    at TestOrderedSet.testWeird(TestOrderedSet.java:41)
```
Why does the code crash on this test? (Diagnose the problem.)