Sample Final Exam

Sample Final
CSCI 136: Spring 2018
9 May

This is a *closed book* exam. You have 150* minutes to complete the exam. You may use the back of the preceding page for additional space if necessary, but be sure to mark you answers clearly.

Be sure to give yourself enough time to answer each question— the points should help you manage your time.

In some cases, there may be a variety of implementation choices. The most credit will be given to the most elegant, appropriate, and efficient solutions.

Problem	Points	Description	Score
1	10	Short Answer	
2	10	Queues	
3	10	StackSort	
4	10	Heaps	
5	10	Binary Trees	
6	10	Hashing	
7	10	AVL Trees	
8	10	Time Complexity	
9	10	Graphs	
Total	100		

I have neither given nor received aid on this examination.

Signature: ______________________________

Name: ____________________________

*In fact, 150 minutes is too little time! This is a test suite to help you prepare. The actual final will have fewer questions.

1. *(10 points)* .... Short Answer

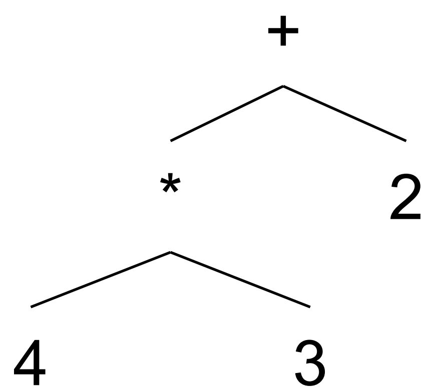
Show your work and justify answers where appropriate.

a. A tree with n elements is both a min-heap and a binary search tree. What does it look like?
   A "degenerate" binary tree where all nodes have only right children.

b. We applied sorting methods primarily to arrays and `Vector`s. Of the following sort algorithms, which are most appropriate to sort a `SinglyLinkedList`: insertion sort, selection sort, quicksort, merge sort?
   Mergsort, as it doesn't require random access.

c. When we rewrite a recursive algorithm to be iterative, we generally must introduce which kind of data structure to aid in simulating the recursion?
   A stack

For parts d. and e., consider the following expression tree (a binary tree where the internal nodes are operators and the leaves are operands):



d. Which tree traversal would you use to print an expression tree in human-readable form (and what is the resulting expression)?
in-order

e. Which tree traversal would you use to *evaluate* an expression tree?
post-order

2. (*10 points*) .... Queues

Recall that the `Queue` interface may be implemented using an array to store the queue elements. Suppose that two `int` values are used to keep track of the ends of the queue. We treat the array as circular: adding or deleting an element may cause the head or tail to "wrap around" to the beginning of the array.

You are to provide a Java implementation of `class CircularQueueArray` by filling in the bodies of the methods below. Note that there is no instance variable which stored the number of elements currently in the queue; you must compute this from the values of `head` and `tail`. You may **not** add any additional instance variables.

```
public class CircularQueueArray {
  // instance variables
  protected int head, tail;
  protected Object[] data;

  // constructor: build an empty queue of capacity n
  public CircularQueueArray(int n) {

    head = 0;
    tail = 0;
    data = new Object[n];
    for (int i = 0; i < n; i++)
        data[i] = null;

  }

  // pre: queue is not fill
  // post: adds value to the queue
  public void enqueue(Object value) {
    data[tail] = value;
    tail = (tail + 1) % data.length;






  }
```

```
  // pre: queue is not empty
  // post: removes value from the head of the queue
  public Object dequeue() {

    Object retVal = data[head];
    data[head] = null;
    head = (head + 1) % data.length;

    return retVal;



  }

  // post: return the number of elements in the queue
  public int size() {

    if (head <= tail)
        return tail-head;

    return data.length - (head - tail);

  }

  // post: returns true iff queue is empty
  public boolean isEmpty() {

    return head == tail && data[head] == null;



  }

  // post: returns true iff queue is full
  public boolean isFull() {


    return head == tail && data[head] != null;



  }
}
```

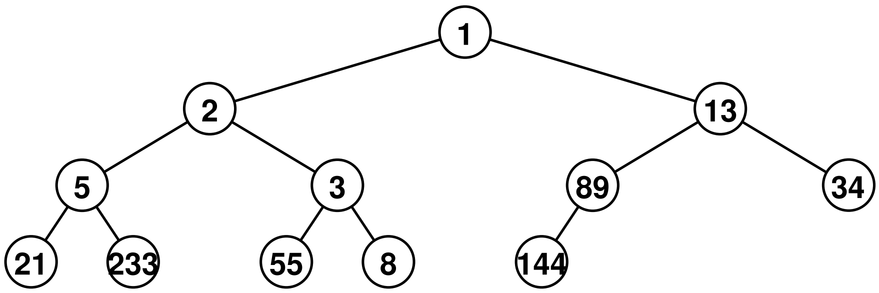
3. (*10 points*) .... Stacks

Suppose you are given an iterator that will let you access a sequence of `Comparable` elements. You would like to sort them, but the only data structure available to you is an implementation of the `Stack` interface in the structure5 package (say, `StackList`). Because the elements are available only through an `Iterator`, so you must process each item as it is returned by the `next()` method of the `Iterator`. The sort method should return a `Stack` containing the sorted elements, with the smallest at the top of the stack. Please fill in the body of the method.

```
public static Stack<Comparable> StackSort(Iterator<Comparable> iter) {
  // pre: iter is an Iterator over a structure containing Comparables
  // post: a Stack is returned with the elements sorted, smallest on top


    Stack<Comparable> retStack = new StackList<Comparable>();
    Stack<Comparable> tempStack = new StackList<Comparable>();

    // strategy: keep retStack sorted each iteration by putting
    // next() into its proper location in the stack.
    // We use tempStack to temporarily hold everything smaller than next(),
    // then push next, then restore the smaller elements from tempStack.
    // (draw a picture!)
    while (iter.hasNext()) {
        Comparable cur = iter.next();

        // pop everything smaller than cur from retStack
        while (!retStack.empty()) {
            if (cur.compareTo(retstack.peek()) <= 0) // found spot for cur
                break;
            tempStack.push(retStack.pop());
        }

        retStack.push(cur);


        while (!tempStack.empty())
            retStack.push(tempStack.pop());

    }

    return retStack;


}
```

4. (*10 points*) .... Heaps

Recall the definition of a min-heap, a binary tree in which each node is no bigger than each of its descendants. For the rest of this question, we presume the Vector implementation of heaps (`class VectorHeap`). Consider the following tree, which is a min-heap.



a. Show the order in which the elements would be stored in the `Vector` underlying our `VectorHeap`.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

1, 2, 13, 5, 3, 89, 34, 21, 233, 55, 8, 144

b. Show the steps involved in adding the value 4 to the heap. **Use drawings of the tree, not the vector.**

We add 4 as the right child of 89 to preserve heap completeness, then we re-establish the heap property. To do this, we "percolateUp": we swap 4 with 89, then swap 4 again with 13. So the final modified path from the root to the leaf is: 1, 4, 13, 89.

c. Using the original tree (not the one with the 4 added), show the steps involved in removing the minimum value of the heap.

We remove the root, and replace it with the rightmost leaf (to preserve the tree's completeness). Then we use "pushDownRoot" to reestablish the heap property: we swap 144 with 2, then again with 3, then again with 8. So the final modified path from the root to the leaf is: 2, 3, 8, 144.

d. Why is the `VectorHeap` implementation of a priority queue better than one that uses a linked list implementation of regular queues, modified to keep all items in order by priority? Hint: Your answer should compare the complexities of the add and remove operations.

Both add and remove in a heap are $O(\log n)$. *If we instead kept all of our elements in a sorted LinkedList, we would need to do* $O(n)$ *work to insert an element.*

5. (*10 points*) .... Binary Trees

Suppose we have a `BinaryTree` that contains only `Comparable` values.

a. It is often useful to find the minimum and maximum values in the tree. Implement the method `maximum` as a member of `class BinaryTree`. Relevant sections of `BinaryTree.java` from the structure5 package are included on pages 12–14 to guide you. Your method should return the `Comparable` that is the maximum value in the tree. It should return `null` if called on an empty tree.

```
public Comparable maximum() {
  // pre: the values in this tree are all Comparable
  // post: the maximum value in the tree is returned

    if (isEmpty())
        return null;

    Comparable maxLeft = left.maximum();
    Comparable maxRight = right.maximum();

    Compable maxChild = maxLeft.compareTo(maxRight) > 0 ?
                        maxLeft : maxRigth;

    return maxChild.compareTo(val) > 0 ? maxChild : val;



}
```

b. What is the worst-case complexity of `maximum` on a tree containing n values?
$O(n)$

c. What is the complexity of `maximum` on a full tree containing n values?
$O(n)$

d. Consider the following method, which I propose as a member of `class BinaryTree`:

```
public boolean isBST() {
  // post: returns true iff the tree rooted here is a binary search tree
  if (this == EMPTY) return true;
  return left().isBST() && right().isBST();
}
```

This method will not always return the correct value. Explain why, then provide a correct method. You may use `minimum()` and `maximum()` from part (a), as well as any other methods of `BinaryTree`.

```
public boolean isBST() {

    if (this == EMPTY)
        return true;

    return left.isBST() && right.isBST() &&
           value.compareTo(left.val) >= 0 &&
           value.compareTo(right.val) <= 0;









}
```

e. In `class BinaryTree`, why is the `setLeft()` method `public`, but the `setParent()` method is `protected`?
Otherwise, we may break the tree structure by setting node A as the parent of node B without setting node B as a child of node A.

```
public class BinaryTree {
    protected Object val; // value associated with node
    protected BinaryTree parent; // parent of node
    protected BinaryTree left; // left child of node
    protected BinaryTree right; // right child of node
    // The unique empty node
    public static final BinaryTree EMPTY = new BinaryTree();

    // A one-time constructor, for constructing empty trees.
    private BinaryTree() {
        val = null; parent = null; left = right = this;
    }

    // Constructs a tree node with no children.  Value of the node
    // is provided by the user
    public BinaryTree(Object value) {
        val = value; parent = null; left = right = EMPTY;
    }

    // Constructs a tree node with no children.  Value of the node
    // and subtrees are provided by the user
    public BinaryTree(Object value, BinaryTree left, BinaryTree right) {
        this(value);
        setLeft(left);
        setRight(right);
    }

    // Get left subtree of current node
    public BinaryTree left() {
        return left;
    }

    // Get right subtree of current node
    public BinaryTree right() {
        return right;
    }

    // Get reference to parent of this node
    public BinaryTree parent() {
        return parent;
    }

    // Update the left subtree of this node.  Parent of the left subtree
    // is updated consistently.  Existing subtree is detached
    public void setLeft(BinaryTree newLeft) {
        if (isEmpty()) return;
        if (left.parent() == this) left.setParent(null);
        left = newLeft;
        left.setParent(this);
    }

    // Update the right subtree of this node.  Parent of the right subtree
    // is updated consistently.  Existing subtree is detached
    public void setRight(BinaryTree newRight) {
        if (isEmpty()) return;
        if (right.parent() == this) right.setParent(null);
        right = newRight;
        right.setParent(this);
    }
```

```
// Update the parent of this node
protected void setParent(BinaryTree newParent) {
    parent = newParent;
}

// Returns the number of descendants of node
public int size() {
    if (this == EMPTY) return 0;
    return left().size() + right.size() + 1;
}

// Returns reference to root of tree containing n
public BinaryTree root() {
    if (parent() == null) return this;
    else return parent().root();
}

// Returns height of node in tree.  Height is maximum path
// length to descendant
public int height() {
    if (this == EMPTY) return -1;
    return 1 + Math.max(left.height(),right.height());
}

// Compute the depth of a node.  The depth is the path length
// from node to root
public int depth() {
    if (parent() == null) return 0;
    return 1 + parent.depth();
}

// Returns true if tree is full.  A tree is full if adding a node
// to tree would necessarily increase its height
public boolean isFull() {
    if (this == EMPTY) return true;
    if (left().height() != right().height()) return false;
    return left().isFull() && right().isFull();
}

// Returns true if tree is empty.
public boolean isEmpty() {
    return this == EMPTY;
}

// Return whether tree is complete.  A complete tree has minimal height
// and any holes in tree would appear in last level to right.
public boolean isComplete() {
    int leftHeight, rightHeight;
    boolean leftIsFull, rightIsFull, leftIsComplete, rightIsComplete;
    if (this == EMPTY) return true;
    leftHeight = left().height();
    rightHeight = right().height();
    leftIsFull = left().isFull();
    rightIsFull = right().isFull();
    leftIsComplete = left().isComplete();
    rightIsComplete = right().isComplete();

    // case 1: left is full, right is complete, heights same
    if (leftIsFull && rightIsComplete &&
```

```
            (leftHeight == rightHeight)) return true;
        // case 2: left is complete, right is full, heights differ
        if (leftIsComplete && rightIsFull &&
            (leftHeight == (rightHeight + 1))) return true;
        return false;
    }

    // Return true iff the tree is height balanced.  A tree is height
    // balanced iff at every node the difference in heights of subtrees is
    // no greater than one
    public boolean isBalanced() {
        if (this == EMPTY) return true;
        return (Math.abs(left().height()-right().height()) <= 1) &&
                left().isBalanced() && right().isBalanced();
    }

    // Returns value associated with this node
    public Object value() {
        return val;
    }
}
```

6. (*10 points*) .... Hashing

a. What is meant by the "load factor" of a hash table?
Bailey pg 379): The ratio of the number of variables stored to the number of slots/buckets in the array

b. We take care to make sure our hash functions return the same hash code for any two equivalent (by the `equals()` method) objects. Why?
Bailey pg 386): We cannot have duplicate keys map to different places within the same hashtable. Otherwise contains(key) would fail, get(key) would fail, and put(key, value) might not detect duplicate keys.

c. We also said that a good size for a hash table would be a prime or "almost prime" number. Why?
We store Objects in a hashtable by using modular arithmetic to convert a hashcode to a bucket: (hash(key) % array.length). If we don't use a prime array length, multiples of an index will still collide if we resize our array by doubling.

d. A hash table with *ordered linear probing* maintains an order among keys considered during the rehashing process. When the keys are encountered, say, in increasing order, the performance of a failed lookup approaches that of a successful search. Describe how a key might be inserted into the ordered sequence of values that compete for the same initial table entry.
High level idea: When we detect a collision, we will shift to make room within a cluster of collided objects instead of scanning to the end of a cluster. We insert a key within its cluster (i.e., all keys that belong to the same bucket) at the position it would fail if all collided objects were sorted.

e. Is the hash table constructed using ordered linear probing as described in part (d) really just an ordered vector? Why or why not?
No. A sorted vector is globally sorted by keys. A hashtable with ordered linear probing only sorts within runs of collided objects. Keys that are consecutive in a global ordering might not have colliding hash codes.

f. One means of potentially reducing the complexity of computing the hash code for `String`s is to compute it once – when the `String` is constructed. Future calls to `hashCode()` would return this precomputed value. Since Java `String`s are immutable, that is, they cannot change once constructed, this could work. Do you think this is a good idea? Why or why not?
Probably not. Most strings are created and used without ever being stored in a hashtable. If we compute the hashcode whenever we create a string then ALL strings are forced to pay that cost.

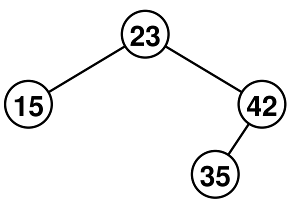
7. (*10 points*) .... AVL Trees

Note: we did not go into as much detail on AVL trees this semester, but we left this in the sample exam because it is still an interesting BST practice problem to work through.

a. Recall that AVL Trees do not necessarily maintain a perfect balance, but require that each tree node's children satisfy the *AVL Condition*. State the AVL condition.
An AVL tree is a binary search tree which has the following properties: (1) The sub-trees of every node differ in height by at most one. (2) Every sub-tree is an AVL tree.
The balance factor of a node is the height of its right subtree minus the height of its left subtree. An AVL tree is balanced if all nodes have a balance factor of -1, 0, *or* $+1$.

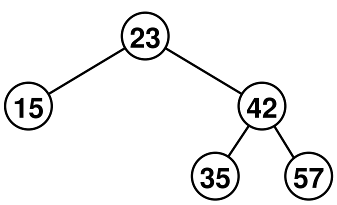
b. We could make our balance criteria more strict and require that the tree must have minimum height for its size, but this is rarely done. Give two reasons why this is the case.

c. Insert the value 33 into the given AVL Tree. Show all steps.



33 is added as a left child of 35. Then we rotate 42. 23's right child becomes 35.

d. Insert the value 39 into the given AVL Tree. Show all steps.



39 is added as a right child of 35. Then we rotate right around 42. (35 becomes the right child or the 23, 42 becomes the right child of 35, 39 the left child of 42, and 57 the right child of 42.) Then we rotate left around the root. (35 becomes the root. 23 is the left child of 35. 42 is the right child of 35. 39 is the left child of 42. 57 is the right child of 42.)

8. *(10 points)* .... Time Complexity

Suppose you are given n lists, each of which is of size n and each of which is sorted in increasing order. We wish to merge these lists into a single sorted list L, with all n^2 elements. For each algorithm below, determine its time complexity (Big O) and justify your result.

a. At each step, examine the smallest element from each list; take the smallest of those elements, remove it from its list and add it to the end of L. Repeat until all input lists are empty.

It takes n comparisons to pick one element, and there are n^2 elements.
$n * n^2 = O(n^3)$

b. Merge the lists in pairs, obtaining $\frac{n}{2}$ lists of size $2n$. Repeat, obtaining $\frac{n}{4}$ lists of size $4n$, and so on, until one list remains.

A total $log_2 n$ levels of merges are required, and merging two sorted lists containing k total items has cost $O(k)$.
$O(log_2(n) * n^2)$

9. (*10 points*) .... Graphs

a. In studying Prim Algorithm, we saw that some edges removed from the priority queue are not useful in that they lead to a vertex we have already visited. How is this possible, given that we insert only edges leading to unvisited vertices into the priority queue?

As we grow our set of vertices through successive iterations of the algorithm, our priority queue may have edges that we had previously inserted but that did not get selected because they had high cost. Since there are often multiple paths to any vertex, we might grow our set of vertices by selecting an alternate path, making a previously inserted edge obsolete.

b. Recall the `Trie` structure you implemented for Lab 8. It was a general tree, where a node in the tree could have an arbitrary number of children. Trees are nothing more than graphs with some restrictions on the edges allowed. You could store the same information in a `Graph` by making a `Vertex` for each tree node and adding `Edge`s representing the links to the children. Which `Graph` implementation would you use for this, and why? How does its time and space complexity compare to your `Trie` implementation?

I would use an adjacency list representation because the LexiconTrie is a sparse graph. An adjacnecy list only stores the edges that are present in the graph, rather than every possible edge that could exist but doesn't.

c. Consider the following definition of a graph.

Def: A graph G consists of a set V, whose members are called the vertices of G, together with a set E, of edges, which are pairs of *distinct* vertices from V (no edges from a vertex back to itself).

Prove by induction that an undirected graph G with n vertices has at most $n(n-1)/2$ edges.

Base cases: 0 vertex has no edges. $n(n-1)/2 = 0(0-1)/2 = 0$. 1 vertex has no edges. $n(n-1)/2 = 1(1-1)/2 = 0$.

Inductive Hypothesis: Assume that for 0 to n vertices, there are at most $n(n-1)/2$ edges.

Then show... Suppose we have any graph with $n+1$ vertices. The vertex with the most edges can be connected to all other vertices, so it has at most n edges. If we remove any one vertex, we have a graph with n vertices, and by our inductive hypothesis, this graph has at most $n(n-1)/2$ edges.

If we add the vertex back, we can have at most $n + (n(-1)/2) = (2n + n^2 - n)/2 = (n^2 + n)/2 = (n(n+1))/2$. This is what we wanted to show.