

# 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 your 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

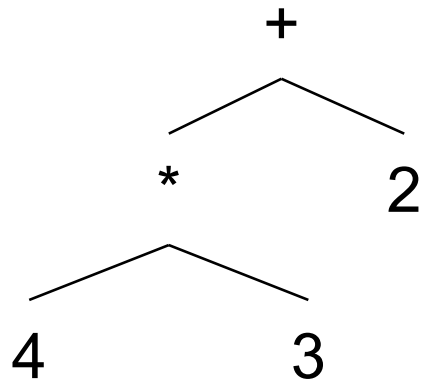
Show your work and justify answers where appropriate.

a. A tree with  $n$  elements is both a min-heap and a binary search tree. What does it look like?

b. We applied sorting methods primarily to arrays and Vectors. Of the following sort algorithms, which are most appropriate to sort a `SinglyLinkedList`: insertion sort, selection sort, quicksort, merge sort?

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?

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



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

c. Which tree traversal would you use to *evaluate* an expression tree?

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) {

    }

    // pre: queue is not full
    // post: adds value to the queue
    public void enqueue(Object value) {

    }
}
```

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

}

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

}

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

}

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

}

}
```

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

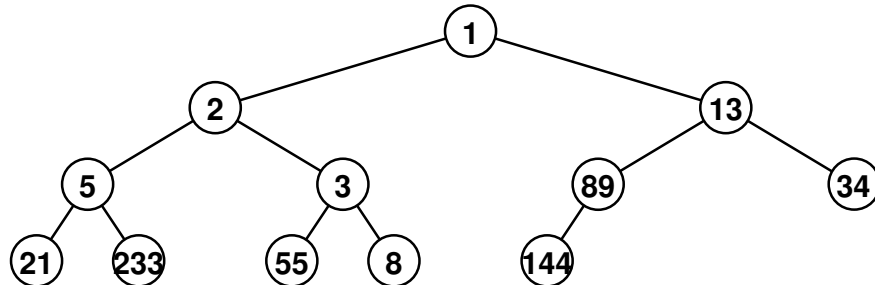
Suppose you are given an iterator that will let you access a sequence of `Comparable` elements. You would like to sort them, but the only data structure available to you is an implementation of the `Stack` interface in the `structure5` package (say, `StackList`). Because the elements are available only 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 StackSort(Iterator iter) {  
    // pre: iter is an Iterator over a structure containing Comparables  
    // post: a Stack is returned with the elements sorted, smallest on top
```

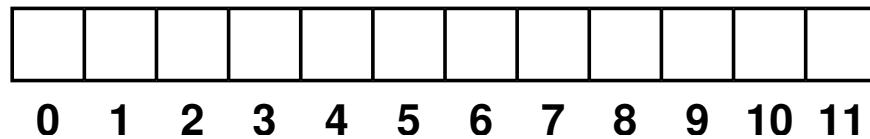
```
}
```

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`.



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

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

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.





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() {
```

```
}
```

e. In class `BinaryTree`, why is the `setLeft()` method public, but the `setParent()` method is protected?



```

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?

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?

c. We also said that a good size for a hash table would be a prime or “almost prime” number. Why?

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.

e. Is the hash table constructed using ordered linear probing as described in part (d) really just an ordered vector? Why or why not?

f. One means of potentially reducing the complexity of computing the hash code for `Strings` is to compute it once – when the `String` is constructed. Future calls to `hashCode()` would return this precomputed value. Since Java `Strings` are immutable, that is, they cannot change once constructed, this could work. Do you think this is a good idea? Why or why not?



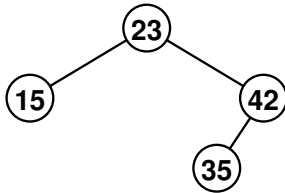
7. (10 points) ..... AVL Trees

*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.*

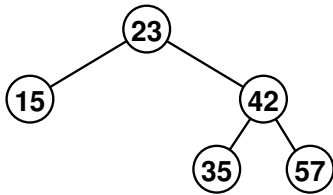
a. Recall that AVL Trees do not necessarily maintain a perfect balance, but require that each tree node's children satisfy the *AVL Condition*. State the AVL condition.

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.



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



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.

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.

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?

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 `Edges` 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?

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.