Name: ____________________

1. (23 points) Short answers. Show your work and justify answers where appropriate.

a. *Free bonus points given in class.* What is the answer given in class for the first question on exam 2? (2 points)

b. When is it advantageous to use a splay tree instead of a regular binary search tree? (3 points)

c. A tree with n elements is both a min-heap and a binary search tree. What does it look like? (3 points)

d. Which tree traversal would you use to print an expression tree in human-readable form? (2 points)

In order, because That is how we typically write formulas:
$$\begin{tikzcd} & + \arrow[dl, dash] \arrow[dr, dash] & \\ 2 & & 3 \end{tikzcd}$$
is printed as $2 + 3$

e. Which tree traversal would you use to evaluate an expression tree? (2 points)

Post order: evaluate The left subtree, Then The right subtree, and Then Combine The results.

Name: ____________________

f. Given a `SinglyLinkedList` containing n elements, what is the complexity (Big O) of `removeFirst()`? of `removeLast()`? (4 points)

remove First : O(1)
remove Last : O(n) - we must traverse whole list to reach end

g. Given a `DoublyLinkedList` containing n elements, what is the complexity (Big O) of `removeFirst()`? of `removeLast()`? (4 points)

remove First: O(1)
remove Last: O(1) - we have a tail reference to access end in O(1) time.

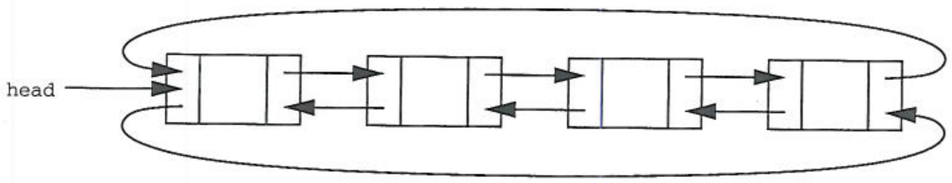
i. We applied sorting methods primarily to arrays and `Vector`s. Of the following sort algorithms, which are most appropriate to sort a `SinglyLinkedList`: insertion sort, selection sort, quicksort, merge sort? (3 points)

All but quicksort would work. Quicksort requires random access to the data, which is ~~[illegible]~~ slow on Singly Linked Lists (ie, get(index) is O(n)).

Merge sort would be the most efficient, since we could still do it in O(n log n) time.

Name: ____________________

2. (15 points) A circular doubly linked list with four elements is represented as in the picture below:



Suppose we have an implementation of such a list, `class CircularDoublyLinkedList`, which includes instance variables:

```
  protected DoublyLinkedListElement head;
  protected int count;
```

Relevant parts of the `class DoublyLinkedListElement` from structure are on page 6.

Consider the `addLast()` method of the `CircularDoublyLinkedList` class:

```
// pre: value is not null
// post: adds value to the tail of the list
public void addLast(Object value);
```

a. What special cases must be considered when writing this method? (5 points)

- when The head is null (ie, The list is empty).

Name: ____________________

b. Write Java code for this method. You need not replicate the pre- and post-conditions specified on the previous page. You may not use `addFirst()` in your code. (10 points)

```
public void addLast(Object value) {
        if (head==null) {
            DLLE newNode= new DLLE(value);
            newNode. setNext(newNode);
            newNode. setPrev(newNode);
            head= newNode;
        } else {
            DLLE newNode= new DLLE (value);
            newNode. setNext( head);
            newNode. setPrev( head. previous());
            head. previous(). setNext(newNode);
            head. setPrevious(newNode);
    }
    count ++;

}
```

```
public class DoublyLinkedListElement {
    protected Object data;
    protected DoublyLinkedListElement nextElement;
    protected DoublyLinkedListElement previousElement;

    public DoublyLinkedListElement(Object v,
                                   DoublyLinkedListElement next,
                                   DoublyLinkedListElement previous) {
    // post: constructs new element with list prefix referenced by
    // previous and suffix referenced by next
        data = v;
        nextElement = next;
        if (nextElement != null) nextElement.previousElement = this;
        previousElement = previous;
        if (previousElement != null) previousElement.nextElement = this;
    }

    public DoublyLinkedListElement(Object v) {
    // post: constructs a single element
        this(v,null,null);
    }

    public DoublyLinkedListElement next() {
    // post: returns the element that follows this
        return nextElement;
    }

    public DoublyLinkedListElement previous() {
    // post: returns element that precedes this
        return previousElement;
    }

    public Object value() {
    // post: returns value stored here
        return data;
    }

    public void setNext(DoublyLinkedListElement next) {
    // post: sets value associated with this element
        nextElement = next;
    }

    public void setPrevious(DoublyLinkedListElement previous) {
    // post: establishes a new reference to a previous value
        previousElement = previous;
    }
}
```

Name: ____________________

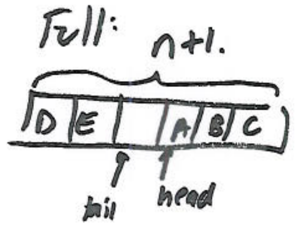
3. (15 points) Recall that the Queue interface may be implemented using an array to store the queue elements. Suppose that two int values are used to keep track of the ends of the queue. We treat the array as circular: adding or deleting an element may cause the head or tail to "wrap around" to the beginning 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 current in the queue; you must compute this from the values of head and tail. You may not add any additional instance variables.

Queue is empty if tail == head

Que is full if tail is one spot before head,.. so we need 1 unused spot in array. to Store n values

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

  // constructor: build an empty queue of capacity n
  public CircularQueueArray(int n) {
        data = new Object[n+1];
        tail = 0;
        head = 0;

  }

  // pre: queue is not fill
  // post: adds value to the queue
  public void enqueue(Object value) {
        Assert.pre( ~~[illegible]~~ !isFull(), "..." ~~[illegible]~~ );
        int newTail = (tail + 1) % data.length;
        ~~#~~ data[tail] = value;
        tail = newTail;

  }
```



Name: ____________________

3. (continued)

```
  // pre: queue is not empty
  // post: removes value from the head of the queue
  public Object dequeue() {
        Assert.pre(!isEmpty(), "...");
        int newHead= (head+1) % data.length;
        Object tmp= data[head];
        head=newHead;
        return tmp;

  }

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

        ~~to~~ return (tail+ data.length - head) % data.length.


  }

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

        return size()==0;

  }

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

        return size()== data.length-1

  }
}
```

Name: ____________________

4. (15 points) The *StackSort*. Suppose you need to sort a stream of `Comparable` elements, and the only data structure available to you is an implementation of the `Stack` interface in the structure package (say, a `StackList`). 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.

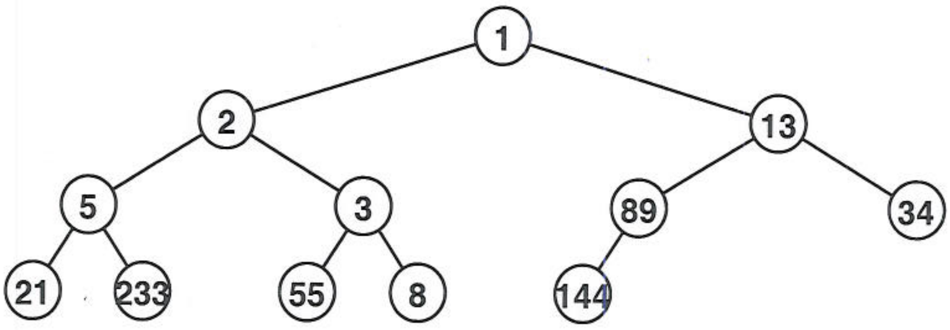
```
public static Stack StackSort(Iterator iter) {
  // pre: iter is an Iterator over a structure containing Comparables
  // post: a Stack is returned with the elements sorted, smallest on top
    Stack data= new Stack();
    While (iter.hasNext()){
        Object value= iter.next();
        Stack temp = new Stack();
        while (!data.isEmpty() && data.peek().compareTo (value) < 0) {
              temp.push(data.pop());
        }
        data.push (value);
        while (!temp.isEmpty()) {
              data.push(temp.pop());
        }
        return data;

}
```

pop of values until top one is larger than value.

push all values Back on to data.

Name: ____________________

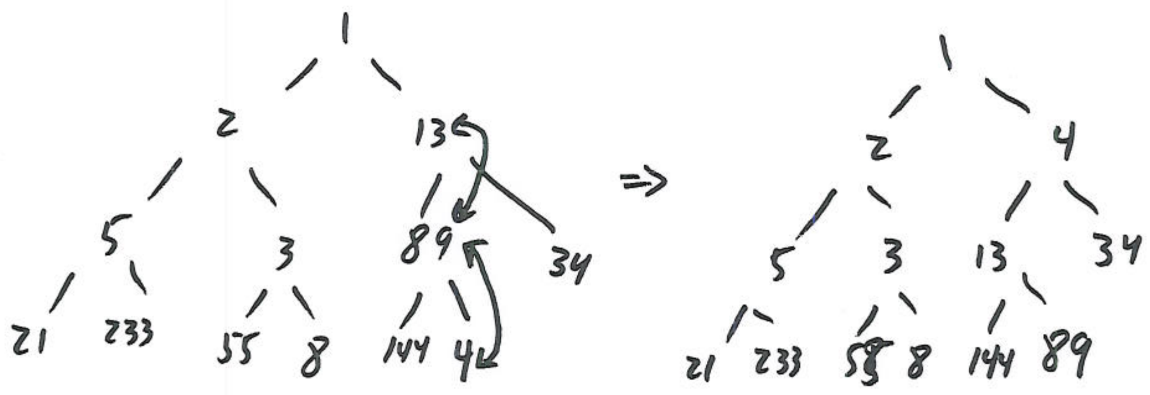
5. (16 points) Recall the definition of a min-heap, a binary tree in which each node is smaller than any of its descendants. For the rest of this question, we presume the Vector implementation of heaps (`class VectorHeap`). Consider the following tree, which is a min-heap.



a. Show the order in which the elements would be stored in the `Vector` underlying our `VectorHeap`. (4 points)

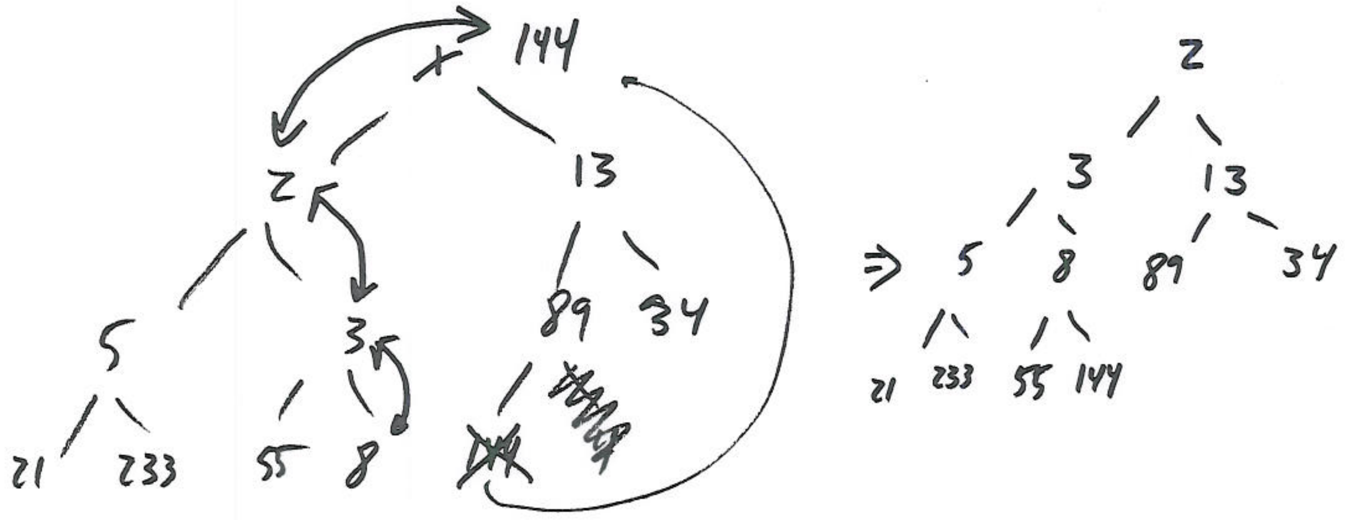
1	2	13	5	3	89	34	21	233	55	8	144
0	1	2	3	4	5	6	7	8	9	10	11

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



Name: ____________________

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



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. (4 points)

add /remove are both O(log n) for VectorHeap,

but [illegible]

add would be O(n) to maintain ordering

on a linked list implementation.

Name: ____________________

6. (18 points) 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 structure package are included on pages 14–16 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. (6 points)

```
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 leftMax = left.maximum();
   Comparable rightMax = right.maximum();
   Corporable max = ~~left~~ value();
   if ( max.compareTo (leftMax) < 0) {max = leftMax; }
   if ( max.compareTo( rightMax) < 0) { max = rightMax; }
   return max;

}
```

b. What is the worst-case complexity of `maximum` on a tree containing n values? (2 points)

$O(n)$ - we must look at every value.

c. What is the complexity of `maximum` on a full tree containing n values? (2 points)

$O(n)$ - being full doesn't change anything.

Name: ____________

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. (6 points)

```
public boolean isBST() {
        if (isEmpty()) {
            return true;
        } else {
① No larger values in left subtree     return !((left.maximum()!=null && left.maximum().compareTo(value()) > 0)
② No smaller value in right subtree    && !((right.minimum()!=null && right.minimum().compareTo(value()) ≤ 0)
③ left subtree is BST                  && left.isBST()
④ right subtree is BST.                && right.isBST();
        }
}
```

e. In class BinaryTree, why is the `setLeft()` method public, but the `setParent()` method is protected? (2 points)

To prevent the client from violating the invariant that a node's left child's parent is the node:

node.left().parent() == node.

```
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;
    }
}
```