Sample Final Exam Solutions

Handout 9
CSCI 136: Fall 2022
Dec 9

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.

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	Iterators	
8	10	Time Complexity	
9	10	Graphs	
Total	90		

I have neither given nor received aid on this examination.

Signature: _____

Name: _____

*In fact, 150 minutes is too little time for the practice exam! We are providing a larger- and harder-than-usual set of questions to help you prepare for the exam. The actual final will be closer to 7 questions.

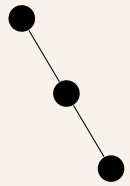
†Problem 10 is harder than any question you should expect to see on the final exam. But think about it anyway: you will be surprised at how well a little stretching prepares you!

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

Show your work and justify answers where appropriate.

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

By the min heap property, the smallest node must be at the top; by the binary search tree property all nodes must be right descendants of the root. Conventionally, in a binary heap, we fill the tree in level order, so the next element to be filled is the left descendent of the root. Therefore, in a binary heap, this tree can only have one element. If we relax the requirement that the heap is filled in level order (not all heaps use implicit trees), then we can obtain a tree with n elements. The above relationship needs to hold recursively for every node. So the final tree must be a path; here is an example with three nodes:



b. Which tree traversal would you use to print an expression tree in human-readable form?

In-order traversal.

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

Post-order traversal. (You need the "solution" from both the left and right side before applying the operator.)

d. Of the following sort algorithms, which are most appropriate to sort a `SinglyLinkedList`: insertion sort, selection sort, quicksort, merge sort?

The simplest is probably merge sort: breaking a linked list into two equal-sized parts can be $O(n)$ time, and merging two linked lists can be done in $O(n)$ time.

Quicksort would require modifying the partition method to work in $O(n)$ time on a linked list. Then, after the two recursive quicksort calls, the two solutions and the pivot element must be carefully grafted back together. This is also a reasonable solution, but requires more modifications than merge sort.

For insertion sort or selection sort, the sorting method would have to be carefully modified to allow linked list items to be swapped efficiently. (Using `get()` to swap two items will lead to $O(n^3)$ rather than $O(n^2)$ running time.) This modification likely requires carefully maintaining two pointers, but is probably ultimately possible.

e. 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

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;
```

```
  }
```

Name: _____

```
  // 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;
```

```
  }
}
```

Name: _____

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`), and memory constraints only allow you to make a small (constant) number of stacks. 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 <E extends Comparable<E> > Stack<E> StackSort(Iterator<E> iter) {
  // pre: iter is an Iterator over a structure containing objects of type E
  // post: a Stack is returned with the elements sorted, smallest on top
```

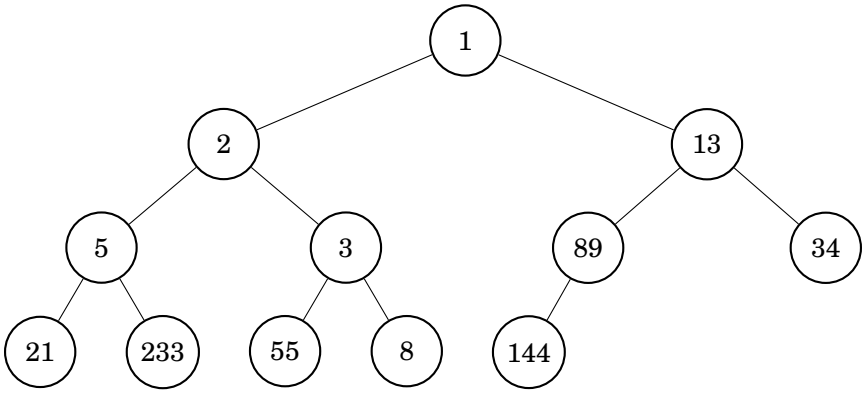
```
  Stack<E> retStack = new StackList<E>();
  Stack<E> tempStack = new StackList<E>();
  // strategy: keep retStack sorted each iteration by putting
  // next() into its proper location in the stack.
  // We use tempStack to temporarily hold everything smaller than next(),
  // then push next, then restore the smaller elements from tempStack.
  // (draw a picture!)
  while (iter.hasNext()) {
        E cur = iter.next();
        // pop everything smaller than cur from retStack
        while (!retStack.empty() && cur.compareTo(retstack.peek()) > 0) {
                tempStack.push(retStack.pop());
        }
        retStack.push(cur);
        while (!tempStack.empty()) {
                retStack.push(tempStack.pop());
        }
  }
  return retStack;
```

```
}
```

Name: _____

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

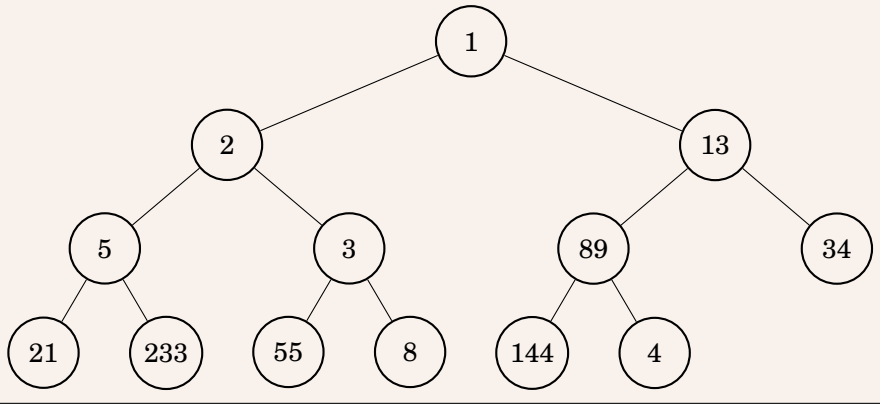
Recall the definition of a min-heap, a binary tree in which each node's value is no bigger than that of each of its descendants. For the rest of this question, we presume the Vector implementation of heaps (`class VectorHeap`). Consider the following tree, which is a min-heap.

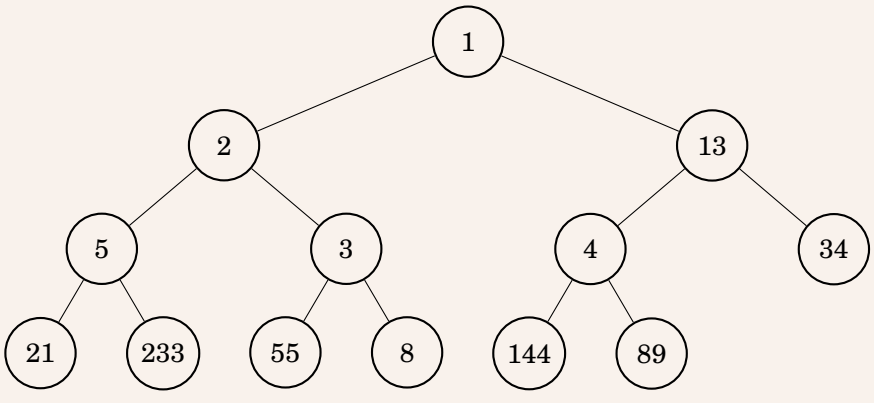


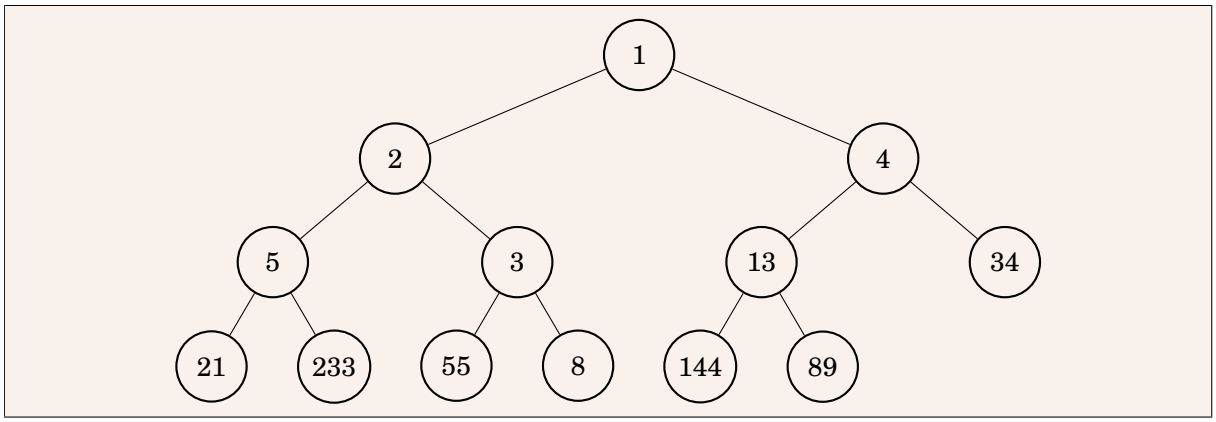
a. Show the order in which the elements would be stored in the `Vector` underlying our `VectorHeap`.

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.**



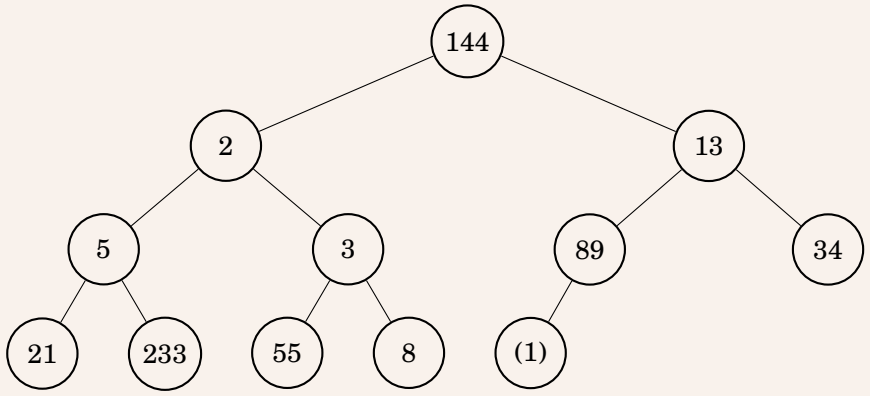


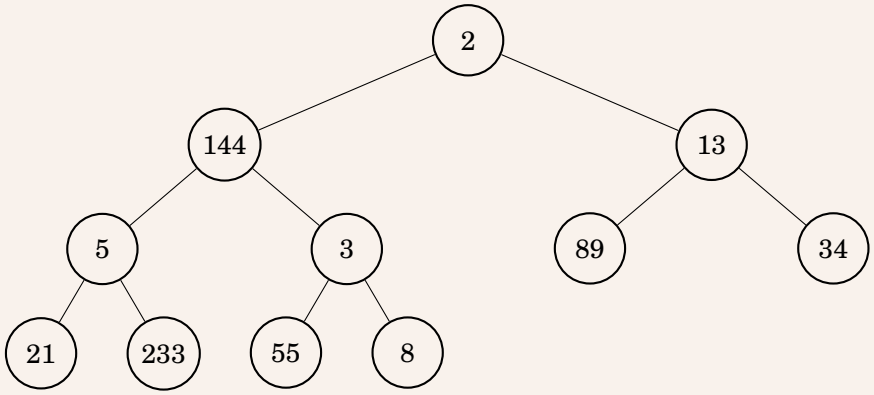

1
2
4
5
3
13
34
21
233
55
8
144
89

Name: _____

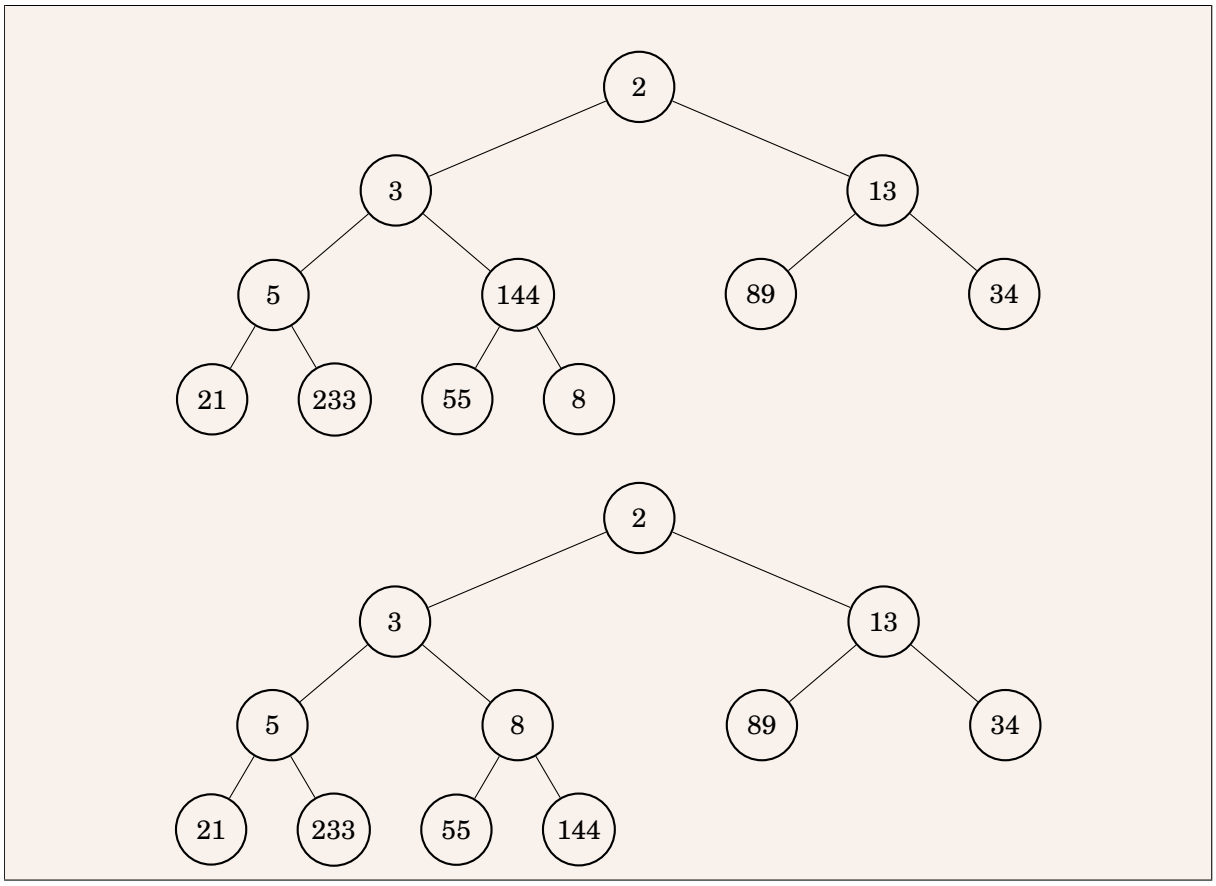
c. Using the original tree (not the one with the 4 added), show the steps involved in removing the minimum value of the heap.

1, the minimum value, is immediately removed after this step (so will not be in future trees):





(Continued on next page:)


2
3
13
5
144
89
34
21
233
55
8
2
3
13
5
8
89
34
21
233
55
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.

The `VectorHeap` implementation requires $O(\log n)$ time to add and remove.

A linked list that keeps items in order requires $O(n)$ time to add, since all items must be scanned to find the correct location. However, removing the minimum-priority item is $O(1)$ time.

If we add and then remove $O(n)$ items, the `VectorHeap` requires $O(n \log n)$ total time for all operations, whereas the sorted linked list requires $O(n^2)$ time.

Name: _____

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`. As a guide, relevant sections of `BinaryTree.java` from the `structure5` package are included after this question. 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();

        Comparable maxChild;
        if(maxLeft != null && maxLeft.compareTo(maxRight) > 0 ) {
                maxChild = maxLeft;
        else {
                maxChild = maxRight;
        }

        Comparable thisVal = (Comparable) val;

        if(maxChild != null && maxChild.compareTo(thisVal) > 0) {
                return maxChild;
        } else {
                return thisVal;
        }
```

```
}
```

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?

$O(n)$

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();
}
```

As written, 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`.

A binary search tree must have value $\geq$ all of its left descendants, and value $<$ all of its right descendants. This method does not compare any values.

```
public boolean isBST() {
```

```
        if(this == EMPTY) {
                return true;
        }

        Comparable compVal = (Comparable) val;

        if(compVal.compareTo(left.maximum()) < 0) {
                return false;
        }

        if(compVal.compareTo(right.minimum()) >= 0) {
                return false;
        }

        return left.isBST() && right.isBST();
}
```

Name: _____

```
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;
    }
}
```

Name: _____

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

a. What is meant by the "load factor" of a hash table?

Number of elements divided by the number of slots.

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?

Hash codes are often used to implement a `Map` using a hash table. The `get` method of `Map` asks the user to look up a key that is `.equals()` to the requested key. We must ensure that the hash codes are the same so that `get()` looks in the correct position.

(There would be issues with `contains()` as well, and even `put()`, which is required to search for duplicate values.)

c. A hash table with *ordered linear probing* maintains an order among keys considered during the rehashing process. For example, we may store the keys in a run in sorted order in the hash table (while ensuring that each key occurs on or after the slot it initially hashed to).

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.

When we detect a collision, we will shift the remainder of the run to make room for the new object so that it can be stored in sorted order. (Another way of saying this: we will treat the run as an OrderedVector, and place each new item into the correct place within the run as if it were shorted, shifting the other elements down.)

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

No; the hash table will not be sorted by the keys. It will only be sorted within runs.

e. 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. Many (if not most) `String`s are created and used without ever being stored in a hash table. If we compute the hash code every time we create the string, then all strings are forced to pay this high cost.

(That said, of course, in a use case where most `String`s get hashed repeatedly, this would save time.)

7. (*10 points*) ..... Iterators

Given a pair of iterators that return data in sorted order, a `Mergerator` merges them into a single iterator that returns all data in sorted order. Here is a simple example of how it might be used:

```
String a[] = { "brown", "cow", "moo" };
String b[] = { "another", "sentence", "sorted" };

AbstractIterator<String>> iter1 = ArrayIterator<String>(a);
AbstractIterator<String>> iter2 = ArrayIterator<String>(b);

Iterator<String> merger = new Mergerator(iter1,iter2);
while (merger.hasNext()) {
  System.out.println(merger.next());
}
```

This code prints out:

```
another brown cow moo sentence sorted
```

This problem asks about implementing the `Mergerator` class to provide the `AbstractIterator` methods: `reset()`, `next()`, `hasNext()`. (You do not need to implement `get()`; also, your `next()` method should not rely on `get()`.)

Complete the code for the `Mergerator` class. Declare any instance variables you will use in your class and implement the constructor and methods (`reset`, `hasNext`, and `next`). Your class should be parameterized by the type `E` of elements that will be returned by the underlying iterators.

```
public class Mergerator <E extends Comparable<E>> extends AbstractIterator<E> {


AbstractIterator<E> iter1;
AbstractIterator<E> iter2;

public Mergerator(AbstractIterator<E> theIter1, AbstractIterator<E> theIter2) {
    iter1 = theIter1;
    iter2 = theIter2;

}

  public void reset() {

    iter1.reset();
    iter2.reset();

  }
  public boolean hasNext() {

    return iter1.hasNext() || iter2.hasNext();

  }
  public E next() {
```

```
    if(iter1.hasNext() && get() == iter1.get()) {
      return iter1.next();
    } else {
      return iter2.next();
    }
```

```
  }
  public E get() {
```

```
    if(iter1.hasNext() &&
      (!iter2.hasNext() || iter1.get().compareTo(iter2.get())) <= 0) {
      return iter1.get();
    } else {
      return iter2.get();
    }
```

```
  }
```

Name: _____

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.

Finding the smallest element of all lists is $O(n)$ time. We repeat this $O(n^2)$ times, for $O(n^3)$ total time.

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.

Merging two lists of size k requires $O(k)$ time (as discussed in class). Therefore, merging *all* pairs of lists (to obtain $n/2$ lists of size $2n$) requires $O(n^2$ time; similarly, merging all pairs of lists of size $2n$ to obtain $n/4$ lists of size $4n$ requires $O(n^2)$ time, and so on.

Each time we merge all pairs, the number of lists is divided by two. We begin with n lists, and end up with 1 list, so we need to merge all pairs $O(\log n)$ times.

We perform $O(\log n)$ merges, each of which takes $O(n^2)$ time, for $O(n^2 \log n)$ time in total.

Name: _____

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

a. Recall the `Trie` structure you implemented for the Lexicon lab. 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 implementation. The number of edges in this graph is generally going to be much less than the number of vertices squared, so this is the space efficient choice. Furthermore, for the lexicon lab, we traverse the tree by iterating over the children of a node; the child whose stored letter is equal to the next character in the word is the next child we visit. This corresponds to iterating over the neighbors of a vertex in a graph. The adjacency list implementation is significantly more efficient at iterating over all neighbors of a vertex, so it is more time efficient as well.

The space efficiency of an adjacency list graph with n vertices and m edges is $O(n+m)$. The space efficiency of a trie with n' nodes, that have a total of m' children, is $O(n' + m')$. Note that $n = n'$ and $m = m'$, so the space efficiency is the same.

The time to obtain a child of a node in the `Trie` with d children is $O(d)$. The time to iterate through all neighbors of a node in an adjacency list with d incident edges is $O(d)$. Therefore, the time to traverse also remains the same.