Sample Final Exam

Handout 11
CSCI 136: Fall 2019
6 December

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	90		

I have neither given nor received aid on this examination.

Signature: _____

Name: _____

*In fact, 150 minutes may be too little time for these problems! This is a problem suite to help you prepare for the exam. The actual final will most likely have 7-8 questions.

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

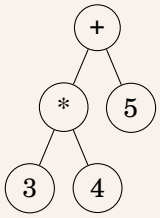
Show your work and justify answers where appropriate.

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

The tree consists of $n - 1$ nodes that have a right child, but no left child. There is also a single leaf node with no children. For example, here is the tree for $n = 3$:

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

In-order traversal. This would mean that the operation would appear in between the two operands, as humans generally expect. For example, the following expression tree:



Would give the equation $3 * 4 + 5$.

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

Post-order traversal. As we saw in class, post-order traversal allows for easy stack-based expression evaluation. And it avoids potential issues with order of operations.

d. We applied sorting methods primarily to arrays and `Vector`s. Of the following sort algorithms, which would you use to sort a `SinglyLinkedList`: insertion sort, selection sort, quicksort, merge sort? Briefly explain your answer.

Merge sort is likely the easiest, as it does not require random access. We do need to break the linked list in half, but this can easily be done with a traversal.

Brief discussion of other options:

- Quicksort would need a new `partition` method that works on linked lists (this is possible, especially if the partition method creates two new linked lists to partition into).
- Selection sort can be done if implemented carefully: at the ith `for` loop iteration, you must find the smallest element among elements $i + 1, \ldots n - 1$ stored in the linked list. This can be placed at the head of the list.
- Insertion sort is very difficult to implement, but is likely possible with sufficient care.

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(isFull())
        return data.length;

    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 Iterator over structure containing elements of type E
  // post: a Stack is returned with the elements sorted, smallest on top
```

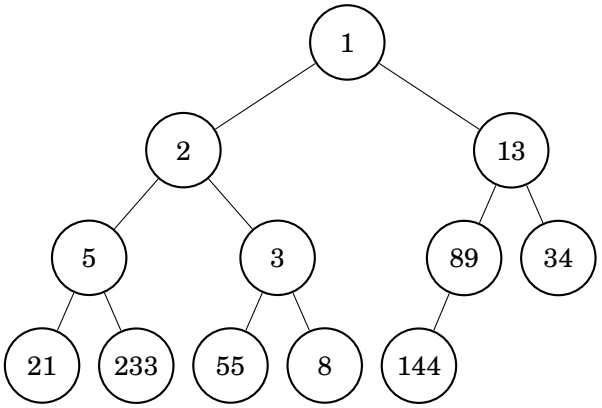
```
  Stack<E> retStack = new StackList<E>();
  Stack<E> tempStack = new StackList<E>();
  while(iter.hasNext()) {
    E cur = iter.next();

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

    while(!tempStack.empty())
      retStack.push(tempStack.pop());
  }
  return retStack;
```

```
}
```

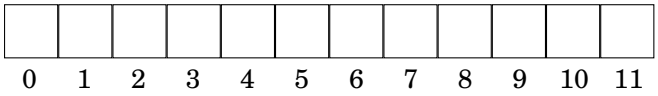
Name: _____

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.

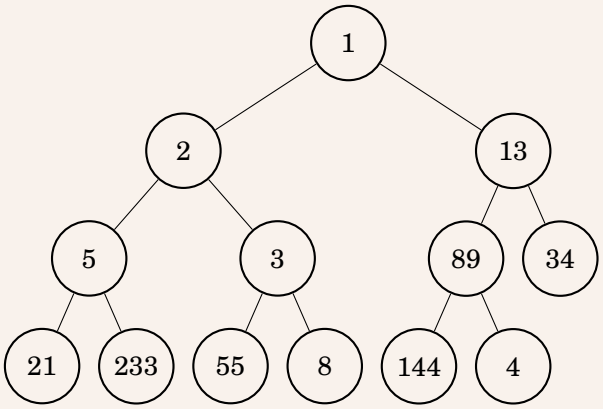


a. Show how the elements would be stored in the `Vector` underlying our `VectorHeap`.

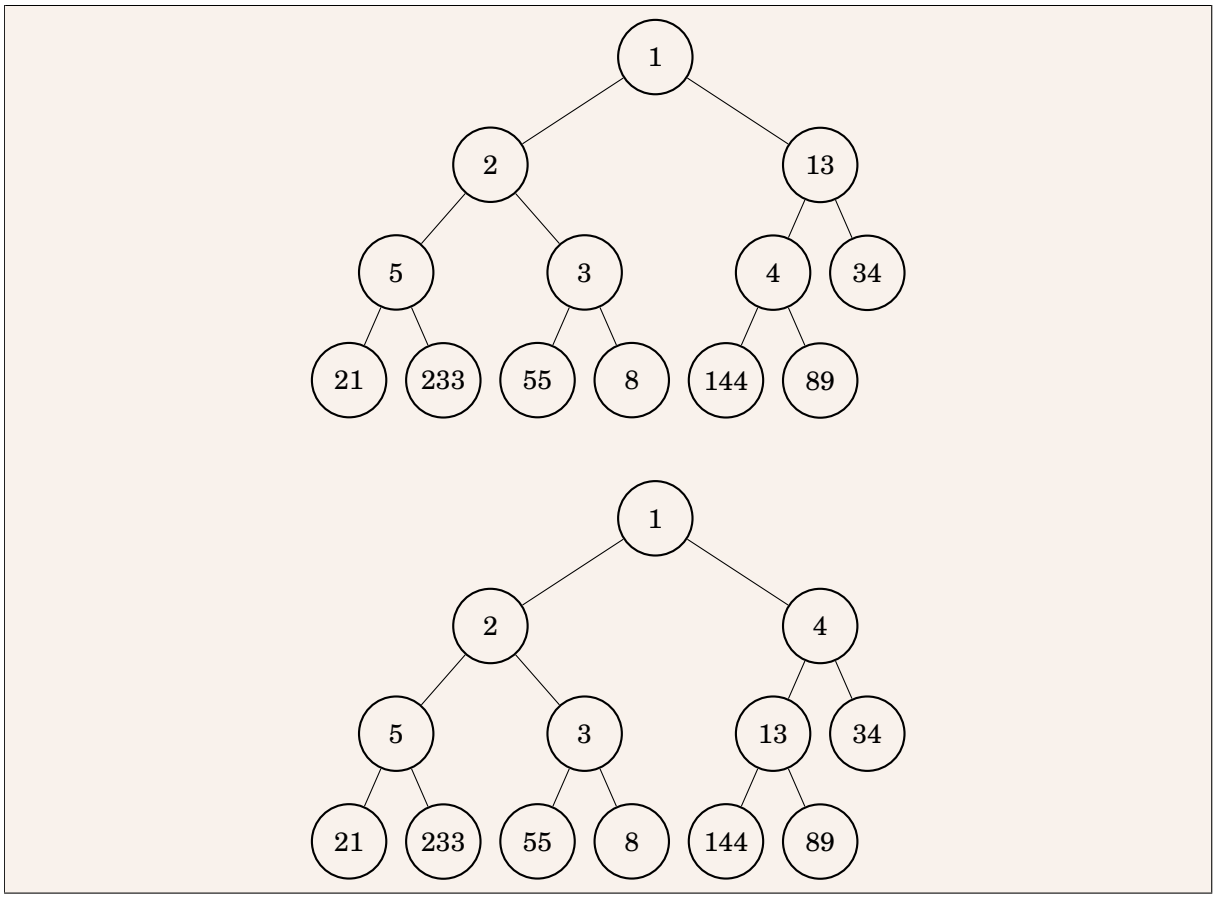


b. Show the steps involved in adding the value 4 to the heap. **Use drawings of the tree, not the vector.**

We insert 4 as the leftmost leaf on the last level, and then percolate up.

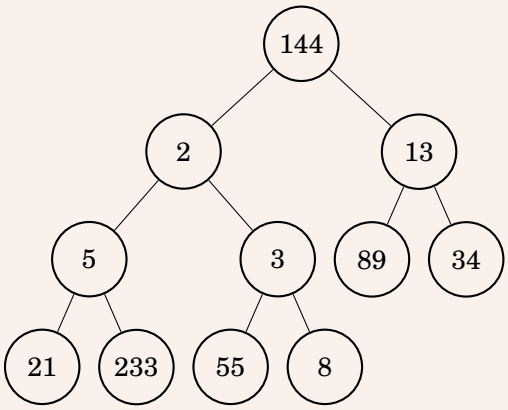


(Answer continued on next page...)

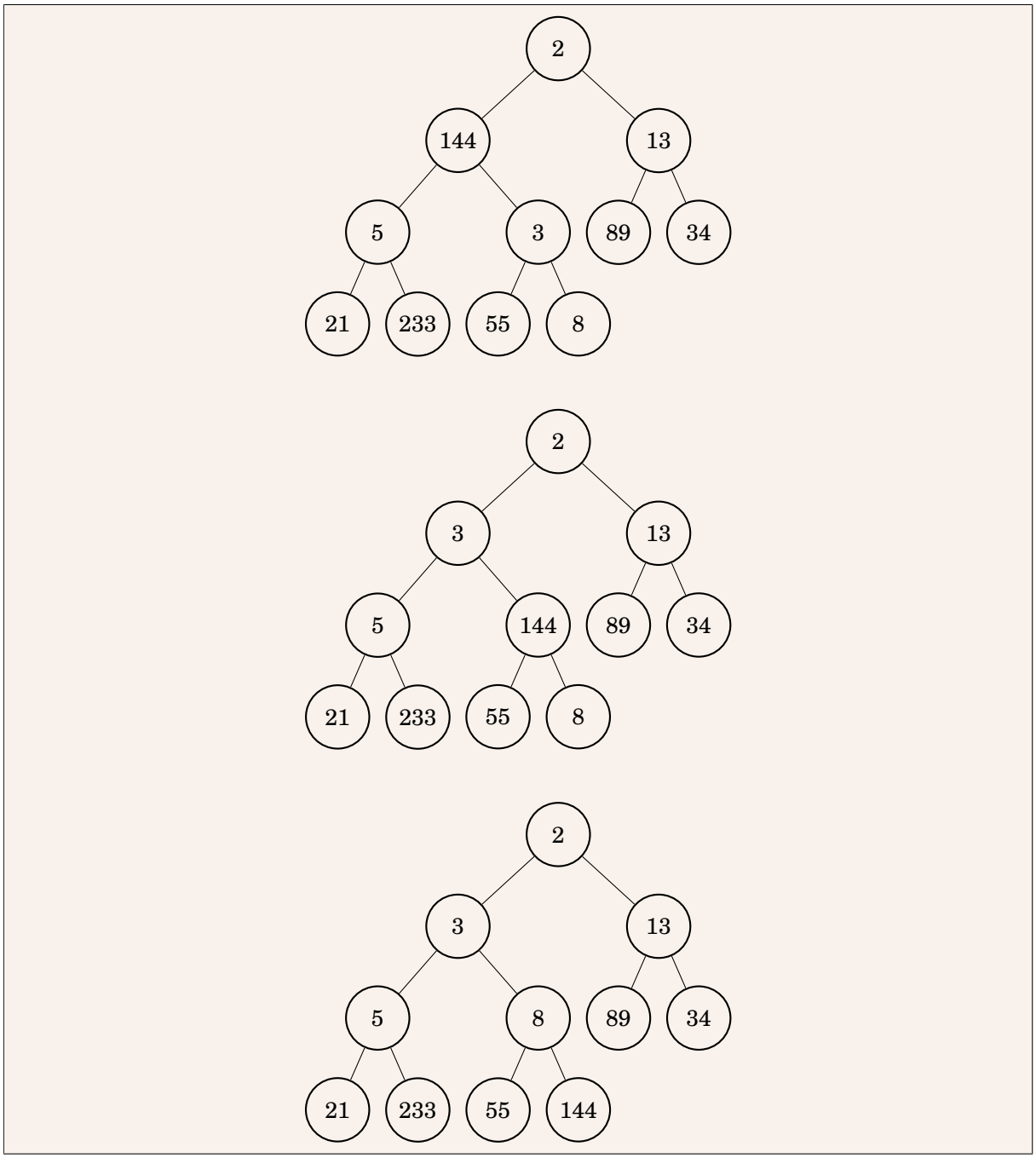


c. Using the original tree (not the one with the 4 added), show the steps involved in removing the minimum value of the heap.

First we swap the root with the rightmost leaf at the last level and remove it. Then we push down, repeatedly swapping the inserted element with its smaller child until it is larger than both of its children.



(Answer continued on next page...)


2
144
13
5
3
89
34
21
233
55
8
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 for both `add` and `remove`. A linked list implementation would require $O(n)$ time for `add` and $O(1)$ time for `remove`. This means that a `VectorHeap` is much better—the huge increase in time for `add` is not made up by the improved time for `remove`.

Name: _____

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

Suppose we have a `BinaryTree<E>` 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 static <E extends Comparable<E> > E maximum(BinaryTree<E> tree) {
  // post: the maximum value in the tree is returned
```

```
  if(tree.isEmpty())
    return null;
  E maxLeft = maximum(tree.left());
  E maxRight = maximum(tree.right());
  E largestChild;
  if(maxRight == null ||
          (maxLeft != null && maxLeft.compareTo(maxRight) > 0)) {
    largestChild = maxLeft;
  } else {
    largestChild = maxRight;
  }

  if(largestChild == null || tree.value().compareTo(largestChild) > 0)
    return tree.value();
  return largestChild;
}
```

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

$O(n)$—the method recursively searches every node of the tree, and performs only $O(1)$ operations at each node.

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

$O(n)$—the method recursively searches every node of the tree, and performs only $O(1)$ operations at each node.

Name: _____

d. Consider the following method:

```
public static <E extends Comparable<E>> boolean isBST(BinaryTree<E> tree){
  // post: returns true iff the tree rooted here is a binary search tree
  if (tree.isEmpty()) return true;
  return isBST(tree.left()) && isBST(tree.right());
}
```

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`.

The method does not check that the root is larger than its left child and smaller than its right child, so we do not know if the BST property is maintained.

```
public static <E extends Comparable<E>> boolean isBST(BinaryTree<E> tree){
  // post: returns true iff the tree rooted here is a binary search tree
  if (isEmpty()) return true;
  return isBST(tree.left()) && isBST(tree.right()) &&
      (tree.left().isEmpty()
                || tree.value().compareTo(tree.left().value()) >= 0) &&
      (tree.right().isEmpty()
                || tree.value().compareTo(tree.right().value()) < 0);
}
```

Note that we need to put checks in to ensure that the left and right child of `tree` are nonempty, as otherwise their values are `null` and the call to `compareTo` may give an error.

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

The `setParent()` method is `protected` to ensure that the references are correctly maintained by the tree. In particular, we want a tree `BinaryTree<E> parent` to be the parent of a tree `BinaryTree<E> tree` exactly when `tree` is a child of `parent`. By keeping this method protected, we ensure that `setParent()` is only called by `setLeft()` or `setRight()`.

```
public class BinaryTree<E>{
    protected E val; // value associated with node
    protected BinaryTree<E> parent; // parent of node
    protected BinaryTree<E> left; // left child of node
    protected BinaryTree<E> right; // right child of node

    // 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(E value) {
        val = value;
        parent = null;
        left = new BinaryTree<E>();
        right = new BinaryTree<E>();
    }

    // Constructs a tree node with tree children.  Value of the node
    // and subtrees are provided by the user
    public BinaryTree(E value, BinaryTree<E> left, BinaryTree<E> right) {
        this(value);
        if (left == null) { left = new BinaryTree<E>(); }
        setLeft(left);
        if (right == null) { right = new BinaryTree<E>(); }
        setRight(right);
    }

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

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

    // Get reference to parent of this node
    public BinaryTree<E> 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<E> 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<E> newRight) {
```

```
    if (isEmpty()) return;
    if (right != null && right.parent() == this) right.setParent(null);
    right = newRight;
    right.setParent(this);
}

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

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

// Returns reference to root of tree containing n
public BinaryTree<E> 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 (isEmpty()) 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 (isEmpty()) return true;
    if (left().height() != right().height()) return false;
    return left().isFull() && right().isFull();
}

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

// 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 (isEmpty()) 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 (isEmpty()) return true;
        return (Math.abs(left().height()-right().height()) <= 1) &&
                left().isBalanced() && right().isBalanced();
    }

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

Name: _____

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

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

The ratio of the number of objects stored in the table to the number of slots/buckets. (Bailey pg 379)

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?

If this was not the case, then two objects with equivalent keys may map to different places in the hash table. This would cause several problems: `contains()` would fail, `get()` would fail, and `put()` may 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?

When using quadratic probing, using a prime number guarantees that half the buckets are searched when attempting to find an empty slot of a given element.

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.

This follows some of the same ideas as an `OrderedVector`. We search linearly through the cluster of stored values to find the first value in the cluster that is larger than the new key. This value and all subsequent values are then shifted down one slot, making room for the key to be inserted into the correct position.

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. An ordered vector is always (entirely) sorted by keys. The hash table we constructed only sorts within each cluster. Keys that are consecutive in a global ordered might not be in the same cluster.

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?

There is a tradeoff in this decision, but it is probably not a good idea for Java Strings.

The benefit of this is that this method would work—each hashcode would only need to be computed once. However, this means that *every* String created in Java would have the overhead of calculating and storing this hash (instead of just the ones used in a hash table).

Name: _____

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

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.

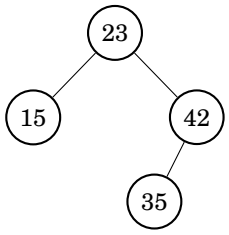
For every node in the tree, the height of its left child must be within 1 of the height of its right child.

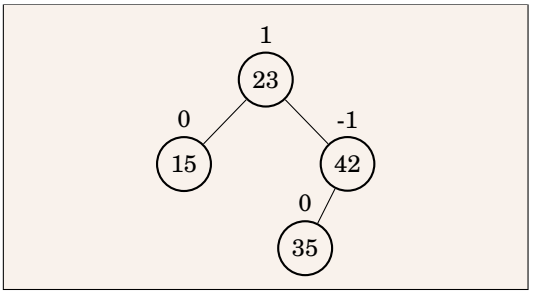
b. We could make our balance criteria more strict and require that the tree must have minimum height for its size, but this is rarely done. Give two reasons why this is the case.

There are many reasons why this is a bad idea. We give some examples.

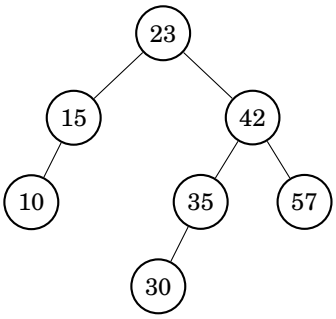
- It seems that this would be difficult to maintain with rotations. Instead, the tree would need to be rebalanced in some other way (possibly rebuilding from scratch).
- It seems unlikely that this property could be maintained in $O(\log n)$ time. Each insert to the tree may require rebalancing the entire tree.
- The benefits of this are minor. The minimum height for a tree is $\log_2 n$ and we showed in class that the height of an AVL tree is $\log_{3/2} n$, which is $O(\log n)$.

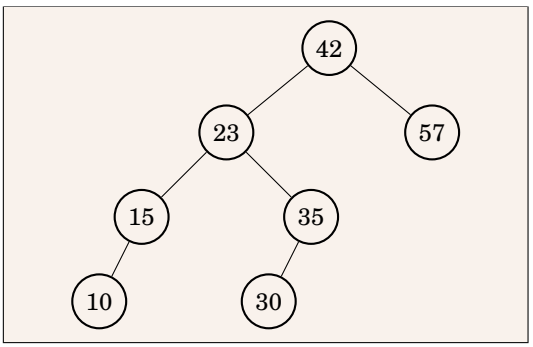
c. Label each node in the following tree with its balance factor. Is this tree an AVL tree?





d. Show the result of performing a single left rotation on nodes 23 and 42 in the following tree. This rotation should make 42 the root of the resulting tree.





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 minimum of the n smallest elements takes $O(n)$ time. We must do this for each of the n^2 elements in L. Multiplying, we obtain $O(n^3)$ 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 with n' total elements takes $O(n')$ time, as we saw in class. Each time we do a round of merges, we merge all n^2 elements; therefore each round takes $O(n^2)$ time.

Each round of merges divides the size of each list by 2. Therefore, this process takes $O(\log n)$ rounds.

Multiplying the number of rounds by the cost of each round, we obtain total time $O(n^2 \log n)$.

Name: _____

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

a. In studying Prim's 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?

When the edge was originally inserted, the vertex was not visited. However, before the edge is removed from the priority queue, it could be that the vertex was visited via another, cheaper edge.

b. 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?

As we discussed in class, a list representation is much better when the graph is relatively sparse (the number of edges is not much more than the number of verticies). In the `Trie` structure we implemented, each node had at most 26 children—almost always much smaller than the number of nodes.

In fact, we can show that the number of edges is always small. Each edge connects a node to its parent, but we know that in a `Trie` (in fact in any tree) every node has at most one parent. Therefore, the total number of edges is at most the number of nodes.

The time and space complexity is similar. In fact, my `Trie` stored the children of each node in an Ordered-Vector, and the `GraphList` structure stores them in a list. In both cases, we potentially scan all the children for `add` or `contains`; in both cases the total space is $O(|V| + |E|)$.

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, and there cannot be two different edges between the same pair of vertices).

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

Base Case: A graph with one vertex has no edges, and $1(1-1)/2 = 0$.

Inductive hypothesis: Any graph G' with $n-1$ vertices has at most $(n-1)(n-2)/2$ edges.

Inductive step: Consider a graph G with n vertices. Remove one of the vertices, and all incident edges. Let x be the number of edges removed; because edges are between distinct vertices we must have $x \leq n-1$. We are left with a graph G' with $n-1$ vertices. By the inductive hypothesis, the number of edges in G' is at most $(n-1)(n-2)/2$.

The number of edges in G is at most x plus the number of edges in G'. Since $x \leq n-1$, and the number of edges in G' is at most $(n-1)(n-2)/2$, the number of edges in G is at most $n-1+(n-1)(n-2)/2$.

Simplifying,

$$n-1+\frac{(n-1)(n-2)}{2} = \frac{2n-2}{2} + \frac{n^2-3n+2}{2} = \frac{n^2-n}{2} = n(n-1)/2$$