Final Exam

CMP-338 110 Points Total Spring 2018

Instructions

- **Answer all questions on the yellow paper.**
- **One question per page.**
- **Use only one side of the yellow paper.**

1. (20 Points) Given the `LinkedList` class that has the following attributes:

```
private Node<I> head = null;
private Node<I> tail = null;
private int listSize = 0;
```

Where `I extends Comparable<? super I>>`. Re-write the following methods from the `LinkedList` class and **fix all 10 logical errors**:

```
// add the given element at the given index
// return true if successful, false otherwise
public boolean add(I element, int index) {
  boolean rc = false;
  Node<I> newNode = new Node<I>(element);
  Node<I> curNode = tail;

  if (index == 0) {
    if (head == null) {
      tail = newNode;
    }
    newNode.setNext(head);
    tail = newNode;
    listSize++;
    rc = true;
  } else if (index > listSize) {
    add(element);
    rc = true;
  } else  if (index < listSize) {
    for ( int j = 1  ; j < index ; j++ ) {
      curNode = curNode.getNext();
    }
    newNode.setNext(curNode);
    curNode.setNext(newNode);
    listSize--;
    rc = true;
  }
  return rc;
}
```

```
 // remove the element at the specified index
 // return true if successful, false otherwise
 public boolean remove(int index) {
   boolean rc = false;
   Node<I> curNode = head;
   Node<I> prevNode = null;

   if (index > listSize) {
     if (index == 0) {
       tail = head.getNext();
       rc = true;
     } else {
       for ( int i = 0 ; i < index ; i++ ){
         prevNode = curNode;
         curNode = prevNode.getNext();
       }
       prevNode.setNext(curNode.getNext());
       if (index == (listSize - 1)) {
         head = prevNode;
       }
       rc = true;
     }
     listSize--;
   }
}
```

2. (30 Points) Write a method to reverse the contents of a list. The `List` class implements the `ListInterface` given below. You can also assume that you have access to a `Queue` class that implements the `QueueInterface` below and a `Stack` class that implements the `StackInterface` below.

```
public interface ListInterface<I>
{
   public int size();
   public boolean isEmpty();
   public void add(I e);
   public I get(int index);
   public void remove(int index);
}
```

```
public interface QueueInterface<I>
{
   public int size();
   public boolean isEmpty();
   public void enqueue(I e);
   public I peek();
   public I dequeue();
   public I dequeue(int index);
}
```

```
public interface StackInterface<I>
{
   public int size();
   public boolean isEmpty();
   public void push(I e);
   public I peek();
   public I pop();
}
```

The method should accept a `List` as a parameter and return the reversed `List` . The method should have the following signature:

```
public List<I> reverseList(List<I> list) {

}
```

3. (20 Points) Given an array of **TreeItem** elements that is sorted in ascending order, write a method to create a balanced **BinarySearchTree** from all the elements of the array. The **BinarySearchTree** implements the **BinarySearchTreeInterface** given below. You method should have the following signature:

```
public BinarySearchTree<K,V> createBalancedTree(TreeItem<K,V>[] items) {

}
```

BinarySearchTreeInterface, **TreeNode** and **TreeItem** are as follows:

```
public interface BinarySearchTreeInterface <K extends Comparable<? super K>, V> {
   public TreeNode<K, V> getRoot();
   public void setRoot(TreeNode<K, V> root);
   public boolean isEmpty();
   public TreeItem<K,V> getRootItem();
   public TreeItem<K,V> find(K key);
   public void insert(TreeItem<K,V> treeItem);
   public void delete(K key);
}
```

```
public class TreeNode
              <K extends Comparable<? super K>, V> {
   private TreeItem<K,V> treeItem;
   private TreeNode<K,V> leftChild;
   private TreeNode<K,V> rightChild;
   private TreeNode<K,V> parent;

   public TreeNode(TreeItem<K,V> treeItem) {
      this.treeItem = treeItem;
      this.leftChild = null;
      this.rightChild = null;
      this.parent = null;
   }

   public TreeNode<K, V> getLeftChild() {
      return leftChild;
   }
   public void setLeftChild(TreeNode<K,V> leftChild) {
      this.leftChild = leftChild;
   }

   public TreeNode<K, V> getRightChild() {
      return rightChild;
   }
   public void setRightChild(TreeNode<K,V> rightChild) {
      this.rightChild = rightChild;
   }

   public TreeNode<K, V> getParent() {
      return parent;
   }
   public void setParent(TreeNode<K, V> parent) {
      this.parent = parent;
   }

   public TreeItem<K, V> getTreeItem() {
      return treeItem;
   }
   public void setTreeItem(TreeItem<K, V> treeItem) {
      this.treeItem = treeItem;
   }
}
```

```
public class TreeItem
    <K extends Comparable<? super K>, V> {
   private K key;
   private V value;

   public TreeItem(K key, V value) {
      this.key = key;
      this.value = value;
   }

   public K getKey() {
      return key;
   }

   public V getValue() {
      return value;
   }

   public void setValue(V value) {
      this.value = value;
   }
}
```

4. (20 Points) A heap can be represented by an array **heapArray**.

An element at index **n** in **heapArray** has it's left child at index **2n + 1** and it's right child at index **2n + 2**. Also, an element at index **m** in **heapArray** has it's parent at index **(m – 1) / 2**.

A Max-Heap is a heap where every node in the heap is bigger than it's children.

Given a random array **heapArray**, write the **heapify()** method that converts **heapArray** into a Max-Heap. You can assume the following:

```
T heapArray[];
```

And

```
<T extends Comparable<? super T>>
```

The method should have the following signature:

```
private void heapify() {

}
```

5. (20 Points) Given the following list of numbers:

80, 70, 10, 100, 90, 130, 60, 120, 50, 110, 20, 30, 140, 150, 40

being inserted in the given order.

a. (5 Points) Draw the resulting Binary Search Tree.
b. (5 Points) Draw the resulting 2-3 Tree.
c. (5 Points) Draw the resulting 2-3-4 Tree.
d. (5 Points) What order should the numbers be inserted in order to obtain a Full Binary Search Tree?