Final Exam
CSCI 1102 Computer Science II

KEY

Instructor Muller
Boston College
Fall 2016

Before reading further, please arrange to have an empty seat on either side of you if you can. Now that you are seated, please write your name **on the top of the back of this exam**.

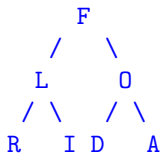
This is a closed-book and closed-notes exam. Computers, calculators and books are prohibited. Partial credit will be given so be sure to show your work. **Please try to write neatly.**

Problem	Points	Out Of
1 Trees, ADTs		15
2 Hash Tables		3
3 Huffman Codes		7
4 Bonus Question		1
Total		26

Part 1: Trees (15 Points Total)

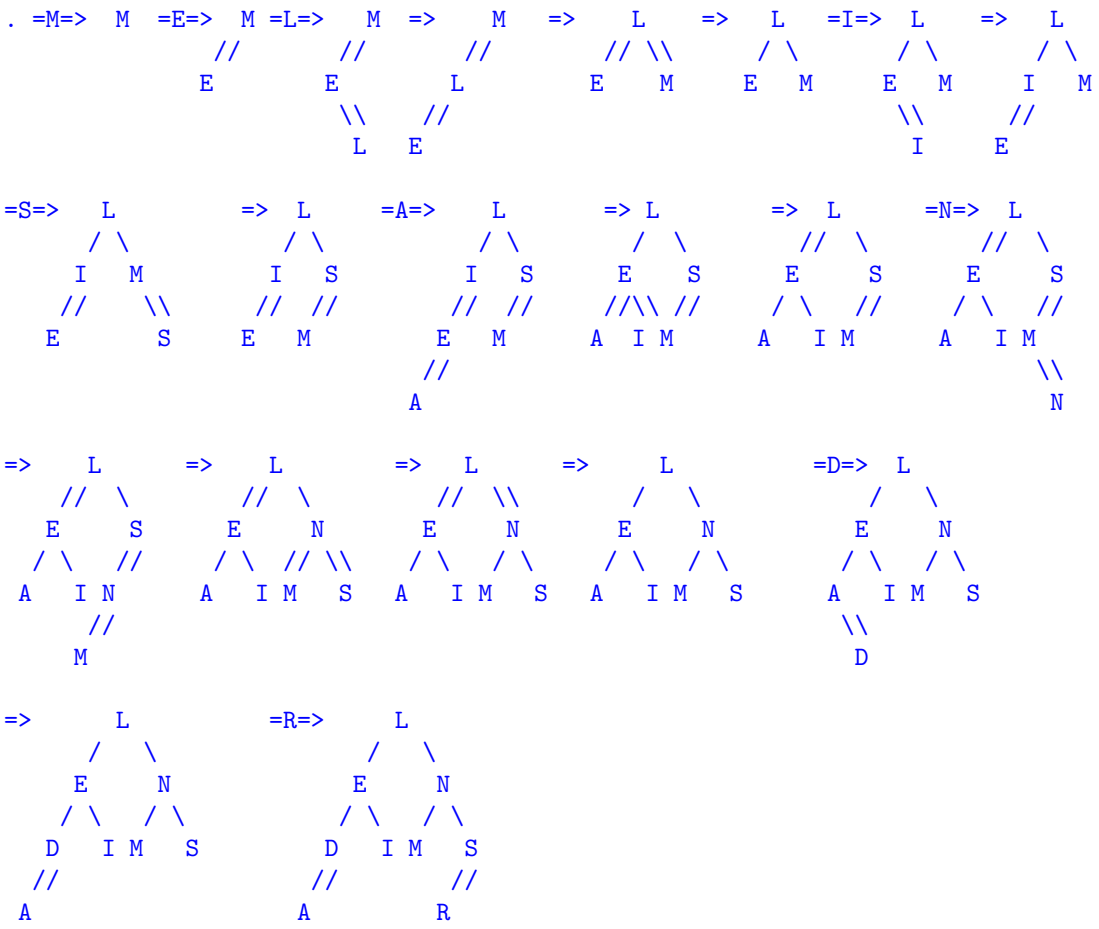
1. (3 Points) In a *full binary tree*, every node has either zero or two children. Alice claims that there is a full binary tree with with postorder traversal `RILDAOF` and with inorder traversal `RLIFDOA`. What is such a tree?

Answer:

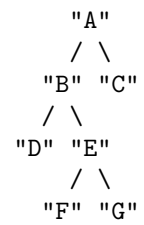


2. (5 Points) Show the succession of left-leaning Red/Black trees resulting from the successive additions of the letters in MELISANDR. Show all of the rotations and color flips.

Answer: MELISANDR



3. (7 Points) Design and implement an ADT for full binary trees supporting values of any type stored in the nodes of the tree. For example, for values of type `String`, we might have:



The ADT can be either mutable or immutable as you like. It should have an operation that checks for empty, an operation for retrieving the value at a node and operations for retrieving the left and right children of a node. In addition, the ADT should have an operation that returns the *height* of the tree (in the example above that would be 3) and it should have an operation that returns a *level-order* traversal of the tree in the form of a `java.util.List<T>` (which is implemented, e.g., by the `java.util.ArrayList<T>` class). For the example, the level-order traversal would be the list:

```
["A", "B", "C", "D", "E", "F", "G"]
```

Answer:

```
// File: BTree.java

import java.util.*;

public interface BTree<T> {
  boolean isEmpty();
  T getInfo();
  BTree<T> getLeft();
  BTree<T> getRight();
  int height();
  List<T> levelorder();
}

// File: Empty.java

import java.util.*;

public class Empty<T> implements BTree<T> {

  public boolean isEmpty() { return true; }
  public T getInfo() { return null; }

  public BTree<T> getLeft() { throw new NoSuchElementException(); }
  public BTree<T> getRight() { throw new NoSuchElementException(); }
  public int height() { return 0; }
  public List<T> levelorder() { return new ArrayList<T>(); }
}
```

```
// File: Node.java

import java.util.*;

public class Node<T> implements BTree<T> {

  private T info;
  private BTree<T> left, right;

  public Node(T info, BTree<T> left, BTree<T> right) {
    this.info = info;
    this.left = left;
    this.right = right;
  }

  public boolean isEmpty() { return false; }
  public T getInfo() { return this.info; }
  public BTree<T> getLeft() { return this.left; }
  public BTree<T> getRight() { return this.right; }

  private boolean isLeaf() {
    return this.left.isEmpty() && this.right.isEmpty();
  }

  public int height() {
    if (this.isLeaf())
      return 0;
    else
      return 1 + Math.max(this.getLeft().height(), this.getRight().height());
  }

  public List<T> levelorder() {
    List<BTree<T>> work = new ArrayList<BTree<T>>();
    List<T> answer = new ArrayList<T>();

    work.add(this);

    while (!work.isEmpty()) {
      BTree<T> t = work.remove(0);
      if (!t.isEmpty()) {
        answer.add(t.getInfo());
        work.add(t.getLeft());
        work.add(t.getRight());
      }
    }
    return answer;
  }

  public static void main(String[] args) {

    BTree<String> empty = new Empty<String>(),
      g = new Node<String>("G", empty, empty),
      f = new Node<String>("F", empty, empty),
      e = new Node<String>("E", f, g),
      d = new Node<String>("D", empty, empty),
      c = new Node<String>("C", empty, empty),
```

```
      b = new Node<String>("B", d, e),
      a = new Node<String>("A", b, c);
      List<String> lo = a.levelorder();
      for (String s : lo)
        System.out.println(s);
  }
}
```

Part 2: Hash Tables (3 Points)

Linear probing is a simple way to resolve collisions when hash tables are organized using open addressing.

$$h(k, i) = (h'(k) + i) \bmod m$$

The downside of linear probing is *clustering*. Lets say the helper function h' is simply $h'(k) = k \bmod m$. Explain clustering by showing a simple example. Give a sentence or two to convey the intuition.

Part 3: Huffman Coding (7 Points)

1. (5 Points) Lets say that a Huffman codec respects the following invariants:

   (a) Items are initially entered into the priority queue in ascending order, i.e., A before B, B before C, etc,

   (b) When ties occur, the new entry is inserted at the back of the group with the same priority,

   (c) When traversing a Huffman Tree, a left branch is a zero and a right branch is a 1.

   Given these protocols, what would a decompression algorithm return when decompressing a .zip file containing the following frequency table and bit stream? Please show all of your work.

   ```
   [A=3, D=1, G=2, L=1, Y=1]
   101111000001110111
   ```

2. (2 Points) The Huffman coding algorithm works on any kind of input data: text, acoustic signals, RGB-values, etc. So it would seem to stand to reason that we could apply the Huffman algorithm to a .zip file, to obtain a doubly-compressed version of an input file. In fact, we can imagine doing this over and over again. In a sentence or two, how would this work out as a way to produce smaller and smaller representations of the input file?

Part 4: Bonus Question (1 Point)

Sedgewick and Wayne are interested in analyzing the resource consumption of algorithms in a finer-grained manner than is provided by asymptotic big O notation. For this purpose, in section 1.4, they define a relation $f \sim g$. What is this relation?

java.util.List⟨E⟩

1. `boolean add(E e)` Appends the specified element to the end of this list.
2. `void add(int index, E element)` Inserts the specified element at the specified position in this list.
3. `boolean addAll(Collection<? extends E> c)` Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
4. `boolean addAll(int index, Collection<? extends E> c)` Inserts all of the elements in the specified collection into this list at the specified position.
5. `void clear()` Removes all of the elements from this list.
6. `boolean contains(Object o)` Returns true if this list contains the specified element.
7. `boolean containsAll(Collection<?> c)` Returns true if this list contains all of the elements of the specified collection.
8. `boolean equals(Object o)` Compares the specified object with this list for equality.
9. `E get(int index)` Returns the element at the specified position in this list.
10. `int hashCode()` Returns the hash code value for this list.
11. `int indexOf(Object o)` Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
12. `boolean isEmpty()` Returns true if this list contains no elements.
13. `Iterator<E> iterator()` Returns an iterator over the elements in this list in proper sequence.
14. `int lastIndexOf(Object o)` Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
15. `ListIterator<E> listIterator()` Returns a list iterator over the elements in this list (in proper sequence).
16. `ListIterator<E> listIterator(int index)` Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
17. `E remove(int index)` Removes the element at the specified position in this list.
18. `boolean remove(Object o)` Removes the first occurrence of the specified element from this list, if it is present.
19. `boolean removeAll(Collection<?> c)` Removes from this list all of its elements that are contained in the specified collection.
20. `boolean retainAll(Collection<?> c)` Retains only the elements in this list that are contained in the specified collection.
21. `E set(int index, E element)` Replaces the element at the specified position in this list with the specified element.
22. `int size()` Returns the number of elements in this list.
23. `List<E> subList(int fromIndex, int toIndex)` Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
24. `Object[] toArray()` Returns an array containing all of the elements in this list in proper sequence (from first to last element).
25. `<T> T[] toArray(T[] a)` Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.