

## Extra Credit Problems

Below are a number of purely optional extra credit problems. Each problem is marked with a number of points that will be awarded for a complete and correct solution. Partial credit will be awarded for problems that are partially complete and/or partially correct.

In most problems, you are expected to write all code, including any testing code, from scratch. If you are interested in a problem but are having difficulty any details of its “from scratch” nature (e.g., you don’t know exactly how to start or how to get your code working with other code we’ve been using in class) please contact Lyn.

Solutions to any of these problems will be accepted through Monday, Dec. 23. To submit a problem, send Lyn an email message that contains (1) a writeup of the problem (i.e., the “hard-copy”) and (2) indicates where the code for the problem lives (i.e., the “softcopy”). Note that you can access all the departments Linux machines remotely, and so may work on these problems at home if you wish (and have the capability to work remotely).

### Extra Credit 1 [25]: Cyclic Lists

You saw in Problem 3 of Problem Set 6 that mutable lists can be used to create cyclic lists – lists in which later nodes point back to earlier nodes. In this problem, you will explore ways to create cyclic lists and tests if lists are cyclic. Assume that all methods are defined in a class named `Cyclic`.

#### a. [5]: Cyclify

Write the following class method:

```
public static void cyclify (ObjectMList L);
```

Given a non-empty, non-cyclic mutable list L, modifies L so that the tail of the last node in L points to the first node in L.

Test your method to show that it works as expected. (This is a bit tricky. A cyclic list is effectively infinite, so you can’t just print out all the elements!)

#### b. [10]: isCyclic

Write the following class method:

```
public static boolean isCyclic (ObjectMList L);
```

Returns `true` if L is a cyclic list – i.e., if L contains some node whose tail points to itself or to a node earlier in the list. Otherwise, returns `false`.

*Notes:*

- To determine cyclicity, `isCyclic` should maintain a collection of all the nodes visited so far and should return `true` if the current node is in the collection. It should return `false` if the empty list is reached without ever finding a node that is a member of the already-visited nodes.
- An implementation of `isCyclic` should ignore the values in the head slots of nodes and instead focus on the nodes themselves. Use `==` to compare if two nodes are the very same instance.

- You may use auxiliary methods if you find them helpful.
- Test your method to show that it works as expected.

**c. [10]: isCyclicTwoFinger**

A disadvantage of the `isCyclic` algorithm mentioned above is that it requires  $\Theta(n)$  space to maintain the collection of already-visited nodes. There is a constant-space means of testing if a given mutable list is cyclic known as the *two-finger algorithm*: imagine that two fingers traversing a list from the beginning, but one finger travels twice as fast as the other (i.e., one finger takes two tails for every tail taken by the other). A list is cyclic if and only if the two fingers meet at the same node after the first step.

Implement and test this idea via a method named `isCyclicTwoFinger`.

**Extra Credit 2 [25]: Node Counting**

Consider the following `size` method for counting the number of nodes in an integer tree:

```
public static int size (IntTree t) {
    if (IT.isLeaf(t)) {
        return 0;
    } else {
        return 1 + (size (IT.left(T)))
            + (size (IT.right(t)))
    }
}
```

**d. [5]** Due to possible sharing involving tree nodes, the `size` method can return a number that is larger than the actual number of distinct nodes in the tree. For example, consider the tree `t1` constructed by the following code:

```
IntTree t3 = IT.node(3, IT.leaf(), IT.leaf());
IntTree t2 = IT.node(2, IT.leaf(), t3);
IntTree t1 = IT.node(1, t2, t3);
```

The tree `t1` contains only three distinct nodes. But what is the result returned by `size(t1)`?

**e. [5]** Design integer trees `t5` and `t7` that contain only three distinct nodes, but for which `size(t5)` returns 5 and `size(t7)` returns 7. Draw pictures of the trees and write the Java code that constructs them.

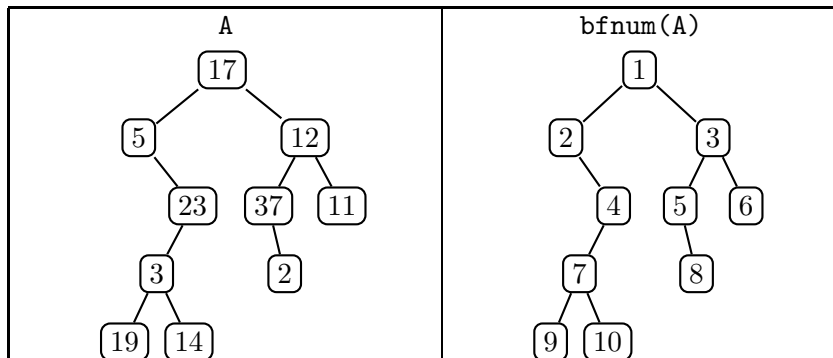
**f. [15]** Write a Java class method `countNodes` that takes an integer tree as an input and returns the number of *distinct* nodes in the tree. For example, `countNodes` should return 3 for the trees `t1`, `tb`, and `tc` described above. *Hints*: Ignore the values in the heads of the nodes. Instead concentrate on the nodes themselves. You can test the equality of two list nodes via `==`.

**Extra Credit 3 [25]: Breadth-First Numbering**

Write the following method:

```
public static IntTree bfnun (IntTree t);
Returns an integer tree that has the same “shape” as t, but in which every node is numbered by the order in which it would be visited in a breadth-first traversal of t.
```

For example, given the tree A shown on the left below, `bfnum` should return the tree shown on the right below.



Notes:

- Test your method to show that it behaves as expected.
- For full credit, your solution should use no mutable data structures (e.g., no mutable integer lists, mutable queues, etc.). Partial credit will be awarded for solutions that use mutable data structures.

#### Extra Credit 4 [35]: Integer Tree Identification

This semester we studied many special kinds of integer binary trees. In this problem, you are asked to write methods that determine if a given binary tree satisfies the conditions of one of the “special” trees.

Notes:

- In addition to writing the method, you should test it to show that it works as expected.
- It is possible for each method to take  $\Theta(n)$  time for an  $n$ -node tree. Full credit will be given only for  $\Theta(n)$  solutions. Partial credit will be given for less efficient solutions.

g. [5]

```
public static boolean isMaxHeap (IntTree t);
Returns true if t is a max heap and false otherwise.
```

h. [10]

```
public static boolean isLeftist (IntTree t);
Returns true if t is leftist and false otherwise.
```

i. [10]

```
public static boolean isFull (IntTree t);
Returns true if t is full and false otherwise.
```

j. [10]

```
public static boolean isComplete (IntTree t);  
Returns true if t is complete and false otherwise.
```

### Extra Credit 5 [20]: In-place Heap Sort

At the bottom of p. 10 of Handout #24 is a brief section entitled *Heapsort Revisited* that describes an *in-place* version of heap sort for vectors – i.e., one that does not use any additional vector (or any other  $\Theta(n)$ -size) storage other than the vector supplied as an argument to the heap sort method.

Based on this description and the other information on complete heaps in Handout #24, define a `HeapSort` class with the following single public method:

```
public static void heapSort (Comparator c, Vector v);  
Uses an in-place heap sort algorithm to sorts the elements of v from low to high according to  
the order specified by c.
```

*Notes:*

- Your `HeapSort` class should contain a `main` method that tests `heapSort`. You should use this method to show that your `heapSort` method works as expected.
- In addition to the single public class method `heapSort`, your class can contain any number of private class methods. You may even find it helpful to define instance variables for `HeapSort` and create `HeapSort` instances that are used by the `heapSort` method.

### Extra Credit 6 [40]: Leftist Heaps

Handout #24 gives pictures and words explaining how to implement operations on leftist heaps, but does not give any Java code. In this problem, you will remedy this lack by defining a `MaxPQLeftistHeap` class that inherits from `CollectionImpl`, implements `Collection`, and represents a max priority queue as a leftist heap.

*Notes:*

- You should represent a leftist heap as an `ObjectTree` where each value is an instance of a `LeftistValue` class that contains the following two public instance variables:
  1. `rank`: the rank of the node holding this instance of `LeftistValue`.
  2. `elt`: the element residing at the node holding this instance of `LeftistValue`.
- You should test your class to show that it behaves as expected.
- In addition to the usual `Collection` instance methods, your `MaxPQLeftistHeap` class should also include the following class method:

```
public static MaxPQLeftistHeap fromVector (Comparator c, Vector v);  
Uses the linear-time algorithm described on the last page of Handout #24 to build an  
instance of MaxPQLeftistHeap from the elements of vector v, using the comparator c to  
compare elements.
```

### Extra Credit 7 [40]: 2-3 Trees

Handout #28 gives words and pictures describing how to implement 2-3 trees, but does not give any Java code. In this problem, you will remedy this lack by defining an `Int23Tree` class that implements *immutable* 2-3-trees with integers as values. Your class should have the following public class methods:

```
public static Int23Tree leaf ();
```

Returns an empty 2-3 tree.

```
public static Int23Tree insert (int x, Int23Tree t);
```

Returns the 2-3 tree that results from inserting the integer `x` into `t`.

```
public static Int23Tree delete (int x, Int23Tree t);
```

Returns the 2-3 tree that results from deleting the integer `x` from `t`. If `x` does not exist in `t`, returns a tree equivalent to `t`.

```
public static boolean isLeaf (Int23Tree t);
```

Returns `true` if `t` is a leaf and `false` otherwise.

```
public static boolean isTwoNode (Int23Tree t);
```

Returns `true` if `t` is a 2-node and `false` otherwise.

```
public static boolean value2 (Int23Tree t);
```

If `t` is a 2-node, returns its value. Otherwise, throws an exception.

```
public static boolean left2 (Int23Tree t);
```

If `t` is a 2-node, returns its left subtree. Otherwise, throws an exception.

```
public static boolean right2 (Int23Tree t);
```

If `t` is a 2-node, returns its right subtree. Otherwise, throws an exception.

```
public static boolean isThreeNode (Int23Tree t);
```

Returns `true` if `t` is a 3-node and `false` otherwise.

```
public static boolean leftValue3 (Int23Tree t);
```

If `t` is a 3-node, returns its left value. Otherwise, throws an exception.

```
public static boolean rightValue3 (Int23Tree t);
```

If `t` is a 3-node, returns its right value. Otherwise, throws an exception.

```
public static boolean left3 (Int23Tree t);
```

If `t` is a 3-node, returns its left subtree. Otherwise, throws an exception.

```
public static boolean middle3 (Int23Tree t);
```

If `t` is a 3-node, returns its middle subtree. Otherwise, throws an exception.

```
public static boolean right3 (Int23Tree t);
```

If `t` is a 3-node, returns its right subtree. Otherwise, throws an exception.

```
public static int height (Int23Tree t);
```

Returns the height of `t`, which is the length of *all* paths from the root of the tree to a leaf.

```
public static int size (Int23Tree t);
```

Returns the number of values in `t`.

*Notes:*

- The API for `Int23Tree` does not include any public methods for directly constructing 2-nodes or 3-nodes. Instead, there are public methods for inserting and deleting integers from an `Int23Tree`. An implementation is likely to have *private* methods for constructing 2-nodes and 3-nodes, but these are not exported to help guarantee that every instance of `Int23Tree` is a valid 2-3 tree – i.e., (1) it respects the ordering condition and (2) every path from the root to a leaf has the same length.
- You should extensively test your implementation to show that it works as expected.
- A challenging aspect of this problem is designing the representation of an instance of `Int23Tree`.

### Extra Credit 8 [25]: Character Bags

Restricting the elements of a collection can sometimes lead to specialized collections for which operations are very efficient. In this problem, you should implement a Java class `CharBag` that models bags of characters with 8-bit ASCII values. Instances of `CharBag` can be efficiently represented as objects with three instance variables:

1. `elts`: An integer array with 256 slots. The contents of slot  $i$  is the number of occurrences of the character with ASCII value  $i$ .
2. `size`: The total number of character occurrences in the bag.
3. `count`: The number of distinct characters in the bag.

The `CharBag` class should support all the instance methods supported by the `Bag` interface, except that references to elements of type `Object` should be replaced by references to type `char`. You should implement the following operations so that they run in *constant* time – i.e., independent of the number of characters in the bag: `choose`, `deleteChosen`, `insert`, `delete`, `deleteAll`, `size`, `count`, `clear`, `clone`, `union`, `intersection`, and `difference`. Note that iterating through all elements of the 256-slot array can be done in “constant” time, because the time does not depend on the number of elements in the bag.

Test all the methods of the bag to show that they work as expected.

### Extra Credit 9 [40]: Array Sorting

This semester we studied versions of *insertion sort*, *selection sort*, *merge sort*, and *quick sort* that sort *lists* of integers. In this problem you should develop versions of these algorithms that sort *arrays* of integers. For each algorithm, you should write a `void` Java class method that takes an arbitrary integer array as its single argument and modifies the order of the integers in the array to make them sorted from low to hi.

*Notes:*

- Try to use as little additional storage as possible. In particular, don’t use any intermediate arrays unless they are absolutely necessary.
- Try to use as few auxiliary methods as possible. For example, rather than invoking a `swap` method, you should instead write a sequence of statements that swaps the contents of two array slots. As another example, in insertion sort, rather than writing an auxiliary `insert` method, instead use a `while` loop to achieve the same purpose. This will help to make your sorting implementations as efficient as possible.

- You should test each method to show that it works as expected.
- Compare the actual running times of each array method on arrays of  $n$  elements to the running times of each corresponding list method on a lists of the same  $n$  elements in the same order. You can find the list method implementations in `~cs230/download/Sorting`. You should use the `javatest` command in this directory (rather than `java`) when doing testing. (The `javatest` command uses “interpreted mode” rather than Sun’s “just-in-time” compiler. This makes it easier to compare execution times.)