CS51 Problem Set 4: Modules, Functors, and Priority Queues

Due Friday, February 28 at 5pm.

Intro

In this assignment you will learn about OCaml modules and how to use modules to create good data structure interfaces. You will also implement an efficient priority queue with a binary heap.

The questions in this writeup are not graded. They exist to help you check your understanding of the material. You are not expected to include answers to them in your submission, but they are things that we could test on. Answers to most of the questions are in the comments of the problem set.

TESTING: Testing is of course required (though not in modules.ml). Please see testing.ml for how to test.

To get the problem set distribution code, you will want to navigate to your folder where your cloned repository lives (the folder that contains the folders "1", "2", and "3"). Once there, execute:

```
git pull git@code.seas.harvard.edu:cs51-2014/psets.git master
```

Executing this code will download a new directory, called `4`, that contains the distribution code for this problem set. You will want to do a "git push" after downloading this, so that your repo on code.seas is up-to-date.

Partners

You are allowed (and encouraged) to work with a partner on this assignment. You are allowed to work alone if that is your preference. If you are an extension student, you are free to pair up with another extension or on-campus student.

In order to effectively work with a partner, you will want to make the most of git. Decide which of your repos you'd like to use for this assignment (you can also re-follow steps from pset0 to create a new code.sea repo). Then, the owner of that repo should navigate to the repo on code.seas.harvard.edu, click "Manage collaborators" on the right side, click "Add collaborators" on the right side, and add the partner's username with (at least) "Review" and

"Commit" permissions. Then, the partner will want to clone this repo (using git clone and the URL given on the repo's main page, as done in pset0).

You may now both push and pull from this single repo.

Part 1 (5 Points + 1 Challenge point)

Complete the problems in `modules.ml`. You must define a Math module, a LIST signature, and a TF signature.

Part 1.5 (0 Points)

Inside of motivation.ml, FIRSTBINTREE is an interface for binary tree data structures which support *empty*, *insert*, *search*, *delete*, *getmin*, and *getmax* operations. Read and understand FIRSTBINTREE as well as the funny syntax for how to write module signatures. The interface for FIRSTBINTREE is not ideal.

Questions (again, not graded):

- Why is FIRSTBINTREE not ideal?
- How could a call to *delete* give you incorrect behavior for a correctly constructed tree?
- Is FIRSTBINTREE a type?
- How would one *use* FIRSTBINTREE?

Part 2 (10 Points)

BINTREE is a better interface for binary trees. To create this better interface we need to introduce another module type--COMPARABLE--which has its own signature.

Questions:

- Why is this a better interface?
- Why did we need to introduce another module type COMPARABLE?

You must now provide an *implementation* of the BINTREE interface which will be called BinSTree. BinSTree is a special binary search tree where repeated values are compressed into a single node with a list of those values. You must implement BinSTree.

BinSTree is a *functor*. Functors are warm fuzzie things. Also, functors are functions over modules. Functors are not yet modules, they must be applied in order to produce a module. In this case, BinSTree takes a COMPARABLE module *as an argument* and returns a BINTREE

module.

Once we have implemented BinSTree we can create IntTree—a binary search tree of integers—by applying BinSTree to an integer implementation of COMPARABLE.

Part 3 (10 Points)

In this section you will be implementing priority queues. A priority queue is a data structure which allows you to add elements and extract a minimum element. Priority queues are useful when implementing Dijkstra's algorithm efficiently (among other things).

PRIOQUEUE is an interface for priority queues which supports *empty*, *is_empty*, *add*, and *take* operations. *add* inserts an element into the priority queue and *take* removes the minimum element.

Implement ListQueue: a naive implementation of a priority queue. This implementation is not ideal because the *take* operation (or possibly *add*, depending on your implementation) is O(n) complexity.

Next you will implement TreeQueue, which is less naive than ListQueue but still not ideal.

See the assignment file for descriptions and specifications of ListQueue and TreeQueue.

Questions:

- Why is the TreeQueue implementation not ideal?
- What is the worst case complexity of *add* and *take* for a TreeQuee?

Part 4 (25 Points)

You will now implement a priority queue using a binary heap. BinaryHeap will be awesome because it has O(log(n)) complexity for both *add* and *take*.

Binary (min)heaps are complete binary trees for which the root element is smaller than all other nodes in the tree (this holds for all subtrees). Binary heaps are also *balanced* trees which means that all levels in the tree are completely filled, with the possible exception of the last. Note that the usual definition of binary heaps (and complete binary trees) asserts that the last level should be filled from left to right. Here, for a given node, we will work with the stronger invariant that its left branch is either the same size as or has one more node than the right branch. So, a balanced tree is ODD or EVEN (these aren't conventional terms). A tree is ODD if its left branch has one more node than its right branch. A tree is EVEN if its branches are of

equal size. *add and take must return balanced trees for this definition of balanced*.

We have defined the type for implementing the binary heap. Functions over the type will often need to respect one of these representation invariants:

- WEAK invariant: your tree is balanced. For any given node in the tree, there are equally as many nodes in the left subtree as the right subtree, or the left has exactly 1 more. The tree cannot have more nodes on the right than the left.
- STRONG invariant: your tree is balanced, and for every branch, the value of the node of that branch is less than or equal to the values of its child nodes.

add and *take* must take in and return trees which respect the STRONG invariant. We have provided stubs for helper functions which operate on trees that must only respect the WEAK invariant.

TIP: your branch nodes should track whether they are ODD or EVEN. This will help you keep your tree balanced at all times! Notice that we have encoded the difference between ODD and EVEN branch nodes in the type that we've given you!

You should probably first write a size function for your tree type. This will help you check your representation invariant. You should *not* be calling size in the implementation of *take*; rather, you should be using it to test it.

We have provided you with the implementation of *add* and a partial implementation of *take*. Below are some guidelines when implementing *take* and its helper functions, as well as in understanding *add*.

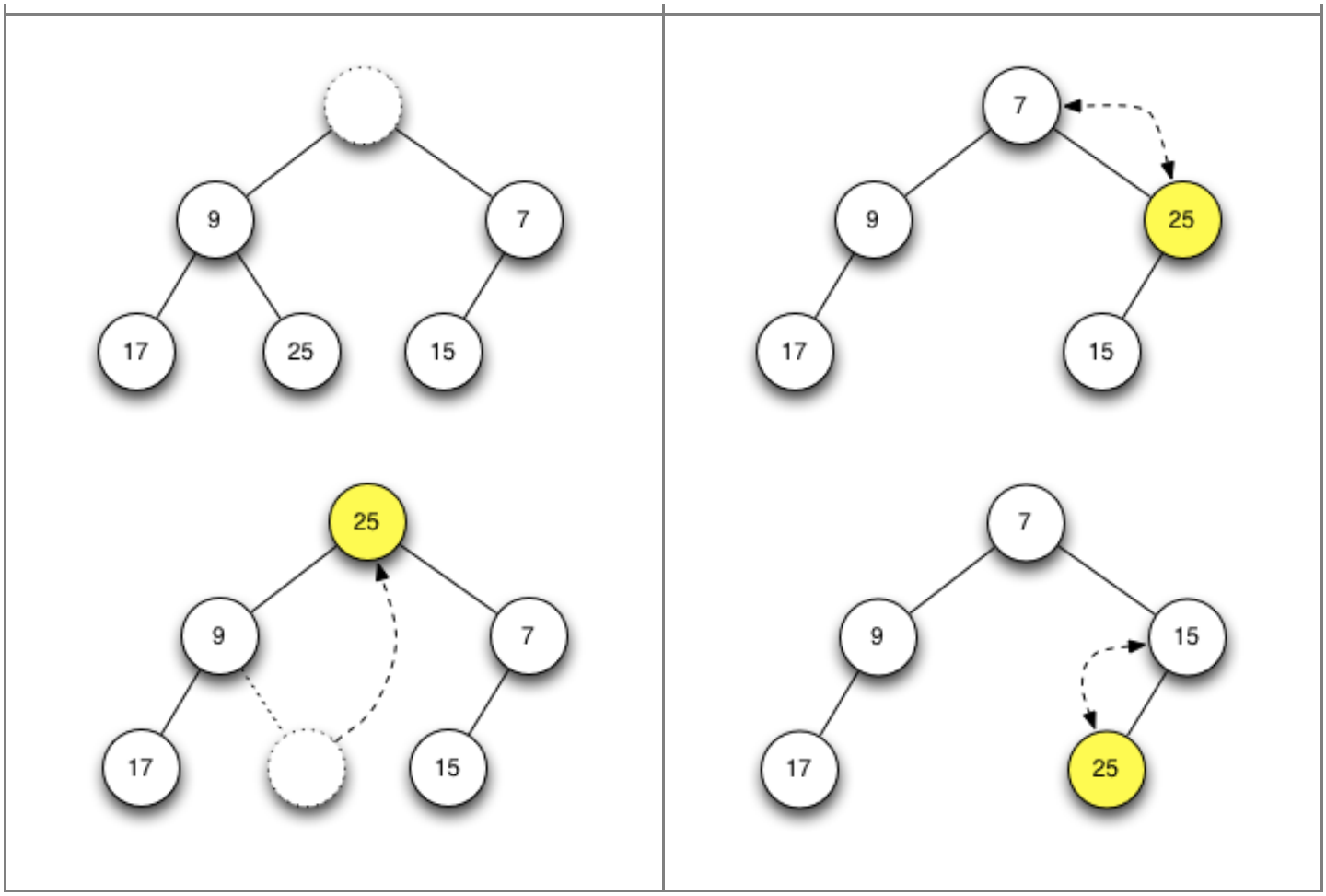
add:

- *add* inserts a node into a spot that will either turn the main tree from ODD to EVEN or from EVEN to ODD. We implement this function for you, but you should understand how it works.

take:

- *take* removes the root of the tree (the minimum element) and replaces it by a leaf of the tree that, when removed, turns the tree from ODD to EVEN or from EVEN to ODD.
- After removing and replacing the root node your tree will respect the WEAK invariant. You must "fix" the tree to respect the STRONG invariant.

Take	Fix



Questions:

- How do we know that our binary heap stays balanced?
- How might you test your binary heap?
- How might you test the helper functions used in implementing your binary heap?
- Why is it useful to use ListQueue, TreeQueue, and BinaryHeap behind a PRIOQUEUE interface?

Now that your priority queues are implemented, we give you an example of how to use them to implement sort functions based on a generic PRIOQUEUE. You should use these for testing (in addition to testing within the modules).

Part N.1 (Challenge, 2 points)

Write a functor for sorting which takes a COMPARABLE module as an argument and has a sort function. You should abide by the following interface:

```
type c
val sort : c list -> c list
```

You should use your BinaryHeap implementation, and test it!

Part N.2 (Challenge, 2 points)

Benchmark the running times of heapsort, treesort, and selectionsort. Arrive at an algorithmic complexity for each sorting algorithm. Record the results of your tests. Be convincing with your data and analysis when establishing the algorithmic complexity of each sort.

Submit!

Before submitting, it is **very important** that you compile your code by running the `make` command while in the directory with your ps4.ml and modules.ml, and then run your compiled programs using `./ps4.native` and `./modules.native`.
Unless you deliberately added print statements for debugging, there should be no output from running the program. If you do not see any output, your asserts passed and you are good to go. Evaluating definitions in the toplevel, while useful during development, **is not sufficient** for these purposes.

ONLY ONE PARTNER NEEDS TO SUBMIT THE CODE. Be sure that both partner's names are written atop ps4.ml. To submit, click here, log in with your HUID, click "Upload New Submission", click "Add files" and select your ps4.ml and modules.ml, add an appropriate description if you want, and click "Start Upload". On the left side of the screen you are brought to, you should be able to see the two files. If you click on the down-arrow button, you should be able to download. You can do all of this from Chrome in the Appliance, so you don't need to worry about transferring the file from your appliance to your local computer.

There is *no substitute* for compiling your code with "make" and then running the executables to test your submission.