CS 51 Code Review 4

Modules and Functors in OCaml

Sam Green and Gabbi Merz

Harvard University

Table of contents

Modules and Abstraction

Modules

- A **module** is a collection of values (and remember, functions are values) and types.
- A **module signature** or **module type** describes the contents of a module.
- While not precisely true, this analogy may help:

$$\frac{\texttt{type}}{\texttt{value}} \cong \frac{\texttt{signature}}{\texttt{module}}$$

Modules

Here's a definition of the Math module:

```
# module Math  =
    struct
        let pi = 3.14159
        let cos = cos
        let sin = sin
        let sum = (+.)
        let max (lst : float list) =
          match lst with
          | [] -> None
          | hd :: tl -> Some (List.fold_right max tl hd)
    end ;;
```

Important syntax here: `module`, `struct`, `end`. What are these for? Are they analogous to other syntax we've seen so far?

Modules (soln)

Solution:

- `module` is similar to `let`. It's used for binding a module identifier to the collection of values (the module) its going to identify.
- `struct`, `end` "wrap" the contents of a module. Note that modules can be anonymous, just like functions. Exercise: see what happens when you put just the `struct ...end` portion of the `Math` module in.

Module Signatures

The `type` analog for modules is the module signature. For example:

```
# module type TF =
  sig
    type info
    val info : info
    val hometown : string
    val print_info : unit -> unit
    val grade_assignment : int -> string
    val favorite_function : float -> float -> float
    val fold : int list -> int -> int
  end ;;
```

We could then apply this signature to the `Sam` or `Gabbi` module. (Remember, files are module by default!)

```
module TFGabbi = Gabbi : TF ;;
```

A `BigNum` Module.

What happened when you typed `#mod_use ''ps3.ml''` into `utop` while you were working on problem set 3?

Something like this (output omitted for space):

```
module Ps3 :
    sig
    type bignum = { neg : bool; coeffs : int list; }
    val base : int
    val negate : bignum -> bignum
    val equal : bignum -> bignum -> bool
...
```

(Notice that `Ps3` has an anonymous module signature applied to it! This signature is the default, which exposes every defined type and values. It's inferred automatically by the compiler!)

What are some undesirable design properties of this default?

BigNum Observations

To name a couple:

- Most importantly, the implementation of the type `bignum` is exposed! Any client of the code could violate the invariant that we worked so hard to establish.
- Even the `base` in which numbers are represented is included.
- There are many, many more values than included in the `Ps3` module than we would ideally want to include in a real `BigNum` module.

Exercise: A `BigNum` Module Signature

The solution to these problems is to write a module signature and use it to enforce an abstraction barrier between client code and the `Ps3` module.

Let's imagine that the only values we wanted to include were `toInt`, `fromInt`, `plus`, `negate`, and `times`. What would the signature be? How would we create a `BigNum` module out of this signature and the `Ps3` module?

Solution (I): A `BigNum` Module

Here's a possible signature:

```
# module type BIGNUM =
    sig
        type bignum
        val fromInt : int -> bignum
        val toInt : bignum -> int option
        val negate : bignum -> bignum
        val plus : bignum -> bignum -> bignum
        val times : bignum -> bignum -> bignum
    end ;;
```

Solution (II): A `BigNum` Module

And it's application:

```
# module BigNum : BIGNUM = Ps3 ;;
module BigNum : BIGNUM
```

Note, importantly, that we can no longer crack into the list of coefficients or directly manipulate the boolean sign:

```
# let b = BigNum.fromInt 5 ;;
val b : BigNum.bignum = <abstr>

# b.coeffs ;;
Error: Unbound record field coeffs
```

Functors

Functor: Definition and Purpose

Informally, a **functor** is a "function" from modules to modules. More precisely, a functor is a module that is **parameterized** by another module.

Some possible uses:

- Enforce abstractions.
- Make code more generic.
- Make code more extensible.

Functor: Motivation

Imagine we needed a stack data structure as part of a system we were building. What's the "simplest" way to use the following signature for stacks with elements of several different types?

```
# module type STACK =
  sig
    exception Empty
    type element
    type stack
    val empty : unit -> stack
    val push : element -> stack -> stack
    val top : stack -> element
    val pop : stack -> stack
    val serialize : stack -> string
  end ;;
```

Functor: Motivation (soln)

What's the easiest way to write a stack module for `ints`, `strings`, `floats`?

A first step would be copy and paste whatever implementation we chose first, change the types, and change the name. Clearly bad – what if there was a bug? What if we discover a better implementation?

Functor: Example

```
module type SERIALIZE =
  sig
    type t
    val serialize : t -> string
  end ;;

module MakeStack (Element: SERIALIZE)
    : (STACK with type element = Element.t) =
    struct
    type element = Element.t ;;
    ...
    end ;;
```

Exercise: Functors

Create a module to handle stacks of `(int, int)` values.

```
# module IntIntSerialize =
    struct
        type t = int * int
        let serialize (x, y) =
            "(" ^ string_of_int x ^ ", "
                ^ string_of_int y ^ ")"
    end ;;

# module IntIntStack = MakeStack(IntIntSerialize);;
```

Do you find this compelling?

Functors: Toy Example[1]

Imagine we have a module that implements this signature:

```
# module type X_int =
    sig
        val x : int
    end ;;
```

How can we write a functor that takes a module of type `X_int` and create a new module of type `X_int` with the value `x` incremented?

[1] Thanks to Niamh Mulholland!

Toy Functor Example

```
# module Increment (M : X_int) : X_int =
    struct
        let x = M.x + 1
    end ;;
```

We can then create a module to start with, and then increment it.

```
# module Three : X_int = struct let x = 5 end ;;
module Three : X_int
# module Four : X_int = Increment(Three) ;;
module Four : X_int
# Four.x - Three.x ;;
- : int = 1
```

Binary Heaps (for Ps4)

Why Binary Heaps?

The conceptual idea for this week's problem set is about modules and functors. It's also the first foray into a real abstraction.

The motivating problem is a priority queue. The implementation progression is:

1. List based.
2. Binary search tree based.
3. Binary heap based.

The binary heap implementation allows for $O(\log n)$ operations that could be worst case $O(n)$ in other implementations.

Binary Heap: Definition

A binary (min)heap is a binary search tree that satisfies additional representation invariants.[2]

The first is an **ordering** invariant:

The value stored at the root of any subtree (including the root of the whole tree) must be smaller than all values stored in the subtrees below the root.

The second is a **balance** invariant:

For any node, its left child tree is either the same size as (in number of nodes) or exactly one node larger than its right child tree.

[2] Recall the definition of representation invariant!

Heap Operations

The biggest challenge in implementing the binary heap is understanding the rules for inserting and popping so that these invariants are enforced. So let's this sequence of operations to get comfortable:

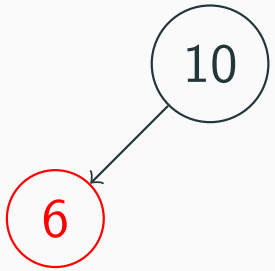
1. `Insert` 10.
2. `Insert` 6.
3. `Insert` 6.
4. `Insert` 5.
5. `Insert` 11.
6. `Insert` 1.
7. `Take`.

Insert 10

1. `Insert 10.`

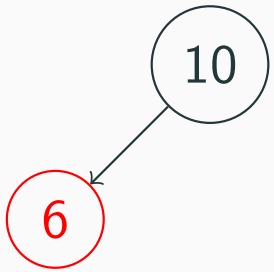
Insert 6 for Balance

1. `Insert 10.`
2. `Insert 6.`

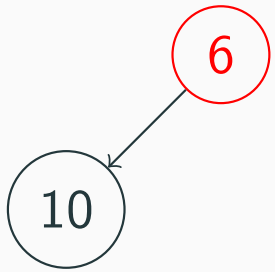


Fix for Order

1. `Insert 10.`
2. `Insert 6.`



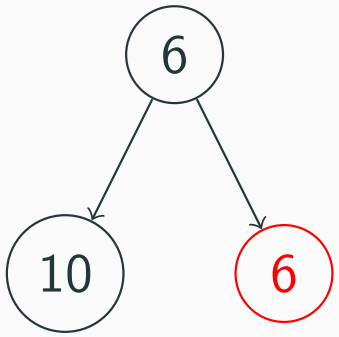
(a) Unfixed.



(b) Fixed.

Add another 6

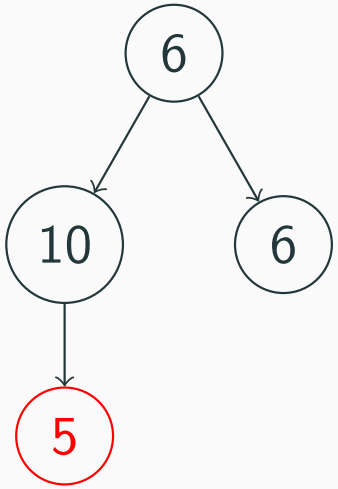
1. `Insert 10.`
2. `Insert 6.`
3. `Insert 6.`



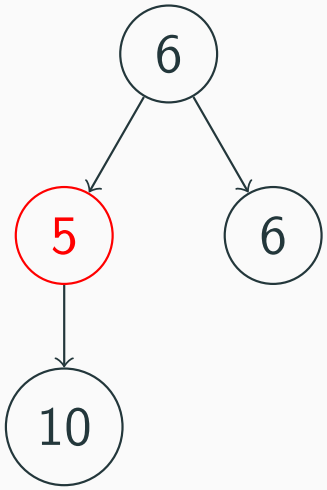
No fix required.

Add a 5

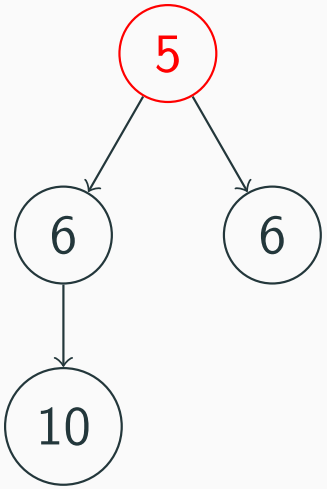
1. `Insert 10.`
2. `Insert 6.`
3. `Insert 6.`
4. `Insert 5.`



(a) Add 5 here for balance.



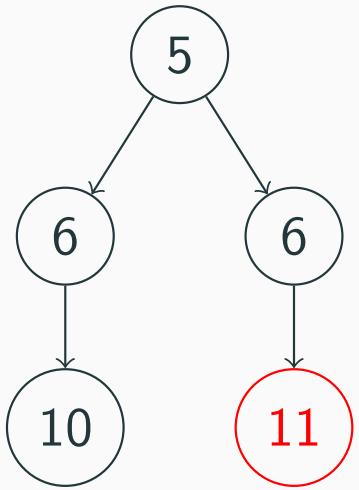
(b) 5 bubbles up.



(c) 5 bubbles up again.

Add an 11

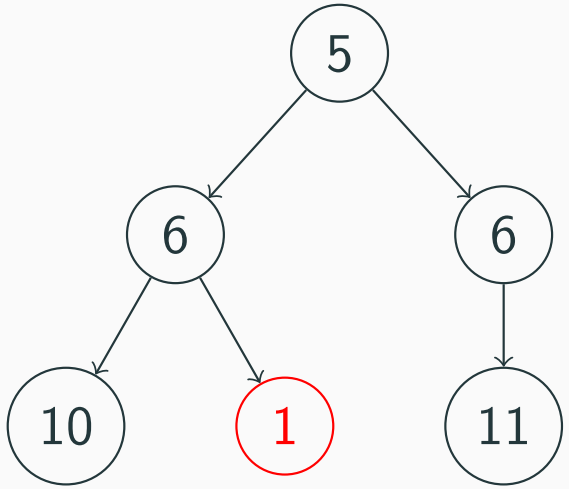
1. `Insert 10.`
2. `Insert 6.`
3. `Insert 6.`
4. `Insert 5.`
5. `Insert 11.`



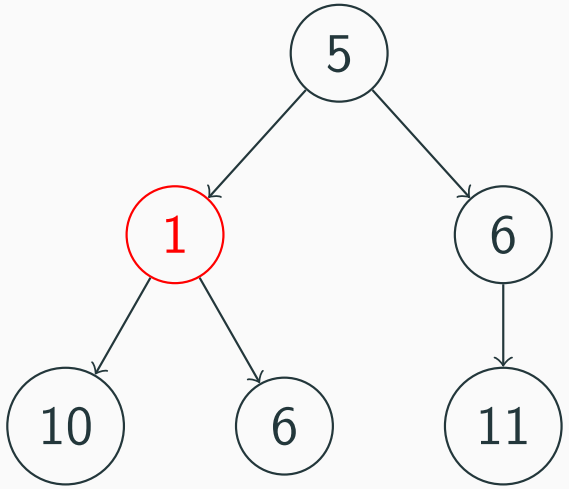
Add 11 here for balance, no fix required.

Add a 1

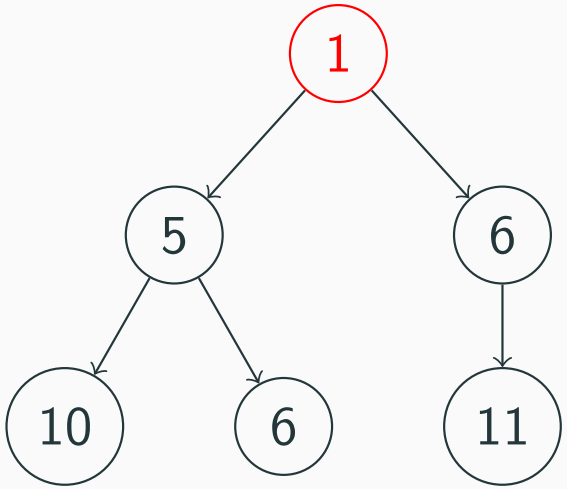
1. `Insert 10.`
2. `Insert 6.`
3. `Insert 6.`
4. `Insert 5.`
5. `Insert 11.`
6. `Insert 1.`



(a) Add 1 here for balance.



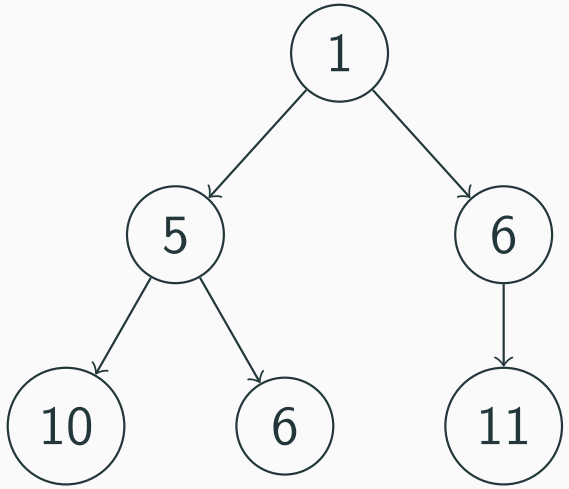
(b) Swap 1 and 6 for ordering



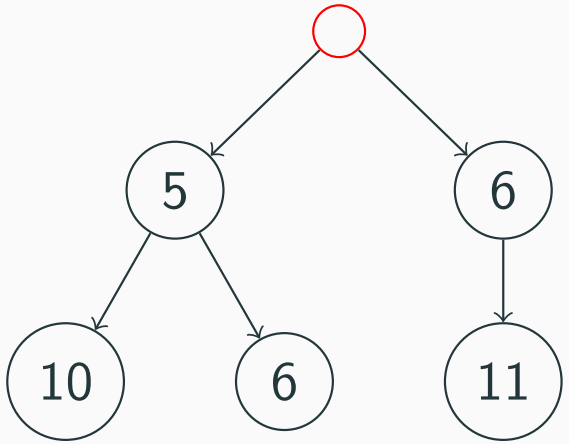
(c) Swap 1 and 5 for ordering

Take the Lowest Element.

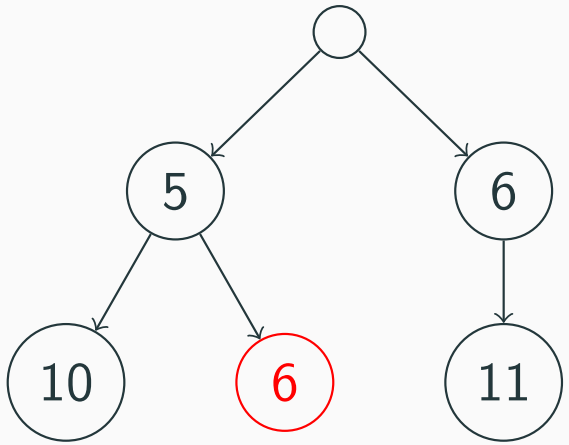
1. `Insert 10.`
2. `Insert 6.`
3. `Insert 6.`
4. `Insert 5.`
5. `Insert 11.`
6. `Insert 1.`
7. `Take.`



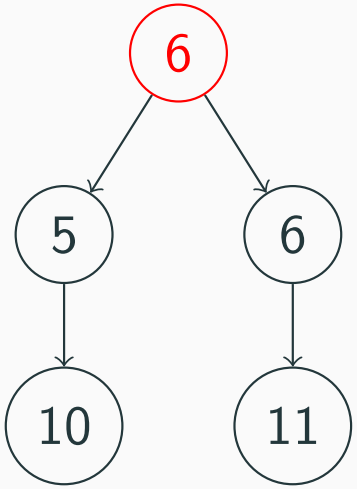
(a) Starting tree.



(b) Remove root.



(c) Use balance to identify element to move.



(d) Move this element to the top, then fix.

Takeaways

1. Understand the invariants.
2. Enforce the invariants separately.
3. Identify the recursive structure. Note that invariant properties apply to subtrees! (Hint.)
4. Draw pictures.
5. (Try not to code until you understand the data structure)

Questions?

Remember to fill out the form!

`http://sa.muel.green/cs51`