CS 3110 Spring 2015
Problem Set 4
Version 2 (last modified April 4, 2016)

Overview

This assignment has two parts. The first part builds upon your implementation of the real numbers in PS3 and shows how modules and functors are powerful tools to extend an existing system. The second part introduces a new data structure, a splay tree, which is similar to a binary search tree but has more interesting properties.

If you consult any written or online sources you need to acknowledge them. Failure to do so is a violation of academic integrity.

You must write your own code. Code copied from outside sources with proper citation does not constitute an AI violation, but will receive minimal credit.

No imperative features are allowed throughout this problem set. This assignment must be done individually.

Objectives

- Be comfortable with using modules and functors to extend an exisisting system.
- Be familiar with a new data structure: splay trees.
- Write formal proofs to reason about your own code.

Recommended reading

The following supplementary materials may be helpful in completing this assignment:

- Lectures 11 12 13 14
- The CS 3110 style guide
- The OCaml tutorial
- Introduction to Objective Caml
- Real World OCaml Chapters 4, 6, 9

What to turn in

Exercises marked [written] should be placed in `written.txt` or `written.pdf` and will be graded by hand. All other exercises should be placed in `fields.ml`, in `geometry.ml`, `utils.ml`, `tests.ml`, or in `splay.ml` and will be graded automatically.

Your files must compile via `cs3110 compile <filename>`.

Geometry

In this part of the problem set you will build upon your knowledge of the real numbers in PS3.

Exercise 1: Revisiting fields.

In PS3 you implemented the real numbers as a field. In this problem set we introduce an ordered field which is just a field equipped with a compare function `cmp` with the following signature:

```
module type FIELD = sig
  type t

  val zero : t
  val one : t

  val add : t -> t -> t

  val multiply : t -> t -> t

  val negate : t -> t

  val inverse : t -> t

  val to_string : t -> string

  val num_of_big_int : big_int -> t

  val num_of_int : int -> t
end

module type OrderedField = sig
  include FIELD

  type cmp = Less | Greater | Equal
  val cmp : t -> t -> cmp
end
```

In `fields.ml` we provided the modules `R` and `Q` both of which are ordered fields. Note that `R.to_string` is unimplemented since this function is for your own testing purposes. We suggest that you implement it yourself if you find it useful.

TODO: Implement `module Floats : OrderedField` using `float` as the number type.

In the next 2 exercises you will build a functor `Geometry (F : OrderedField)` which deals with two geometry problems on the 2 dimensional Euclidean plane where coordinates of points are of type `F.t`.

Exercise 2: Convex Hull.

In Euclidean space, a convex set is a region such that for every pair of points within the region, every point on the straight line segment that joins the pair is also within the region. Given a set of points X on the 2 dimensional plane the convex hull of the set is smallest convex set that contains X. There are various efficient algorithms to compute the convex hull of a set of points. In this exercise you will implement Graham scan which runs in $O(n \log n)$ where n is the number of points.

The algorithm can be described in 3 simple steps:

1. The first step is to find the point with the lowest y-coordinate. If the lowest y-coordinate exists in more than one point in the set, the point with the lowest x-coordinate out of the candidates should be chosen. Call this point $P = (x_P, y_P)$

2. Sort the set in increasing order of the angle they and the point P make with the x-axis.

   In the previous release of this writeup we suggested that you use the cosine function to do this but this requires taking the norm of a 2 dimensional vector which is not too easy to implement. In this version we introduce another function that maps a vector to a number in the interval $[-1, 3]$ which is monotonic with the angle the vector makes with the x-axis.

   Definition 1. *Given a vector (x, y). Let $p = \frac{y}{|x|+|y|}$. The pseudoangle it makes with the x-axis can be computed as:*

   $$p((x,y)) = \begin{cases} p & \text{if } x \geq 0 \\ 2 - p & \text{otherwise.} \end{cases}$$

3. The algorithm proceeds by considering each of the points in the sorted array in sequence. For each point, it is first determined whether traveling from the two points immediately preceding this point constitutes making a left turn or a right turn. If a right turn, the second-to-last point is not part of the convex hull, and lies inside it. The same determination is then made for the set of the latest point and the two points that immediately precede the point found to have been inside the hull, and is repeated until a "left turn" set is encountered, at which point the algorithm moves on to the next point in the set of points in the sorted array minus any points that were found to be inside the hull; there is no need to consider these points again.

   Again, determining whether three points constitute a "left turn" or a "right turn" does not require computing the actual angle between the two line segments, and can actually be achieved with simple arithmetic only as follows.

   For three points $P_1 = (x_1, y_1), P_2 = (x_2, y_2)$ and $P_3 = (x_3, y_3)$, simply compute the z-coordinate of the cross product of the two vectors $\overrightarrow{P_1P_2}$ and $\overrightarrow{P_1P_3}$, which is given by the expression $(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)$. If the result is 0, the points are collinear; if it is positive, the three points constitute a "left turn" or counter-clockwise orientation, otherwise a "right turn" or clockwise orientation (for counter-clockwise numbered points).

TODO: : Implement the function `convex_hull : point list -> point list` in `geometry.ml` which returns the vertices of the convex hull that contains points in the input list in the counter-clockwise order.

We have provided some point sets using both modules `Q` and `R` in file `tests.ml`. Write test cases using these point sets and add a few more tests to make sure all cases are tested for. Recall that it is impossible to decide equality in $\mathbb{R}$, the set of real numbers. The `R.cmp` function provided will terminate if its inputs are distinct numbers but will loop forever otherwise. For this reason there are 2 cases `convex_hull` can fail on a point set.

- Points do not have distinct coordinates.
- There are 3 colinear points

In the real-coordinate point sets provided we ensure that these 2 cases do not happen. You need to keep this in mind while writing your test cases.

Exercise 3: Area of a polygon.

The convex hull in the previous exercise is an example of a convex polygon. In general a polygon is defined to be the area bounded by straight line segments which form a closed loop.

A polygon is self-crossing if one edge crosses over another edge of the same polygon. We will not consider this type of polygon.

Given the vertices of a non-self-crossing polygon, there is a simple algorithm that computes its area.

Let $[(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots, (x_{n-1}, y_{n-1})]$ be the vertices in **clockwise** direction.

Let $(x_n, y_n) = (x_0, y_0)$. The area A can be computed using the following formula:

$$A = \left(\sum_{i=1}^{n} (x_{i-1} + x_i).(y_{i-1} - y_i) \right) /2$$

If the list of vertices is in the counter-clockwise direction the result has the same magnitude as the area but with a negative sign in which case we can take the absolute value of A.

More explantion on the correctness of this algorithm can be found here.

TODO: Implement the function `area : point list -> t` in `geometry.ml` that computes the area of a polygon given a list of vertices in the clockwise or counter-clockwise direction. The return area must be positive.

Make sure to test your code thoroughly in `tests.ml`

Splay Trees

A splay tree is an efficient implementation of a binary search tree that takes advantage of locality in the keys used in incoming lookup requests. For many applications, there is excellent key locality. A good example is a network router. A network router receives network packets at a high rate from incoming connections and must quickly decide on which outgoing wire to send each packet, based on the IP address in the packet. The router needs a big table (a map) that can be used to look up an IP address and find out which outgoing connection to use. If an IP address has been used once, it is likely to be used again, perhaps many times. Splay trees can provide good performance in this situation.

Importantly, splay trees offer amortized $O(\log n)$ performance; a sequence of M operations on an n-node splay tree takes $O(\log n)$ time.

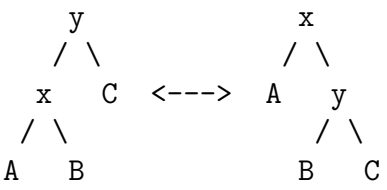
A splay tree is a binary search tree. It has one interesting difference, however: whenever an element is looked up in the tree, the splay tree reorganizes to move that element to the root of the tree, without breaking the binary search tree invariant. If the next lookup request is for the same element, it can be returned immediately. In general, if a small number of elements are being heavily used, they will tend to be found near the top of the tree and are thus found quickly.

We have already seen a way to move an element upward in a binary search tree in recitation: tree rotation. When an element is accessed in a splay tree, tree rotations are used to move it to the top of the tree. This simple algorithm can result in extremely good performance in practice.

There are three kinds of tree rotations that are used to move elements upward in the tree. These rotations have two important effects: they move the node being splayed upward in the tree, and they also shorten the path to any nodes along the path to the splayed node. This latter effect means that splaying operations tend to make the tree more balanced.

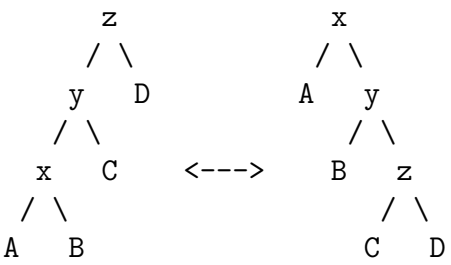
Rotation 1: Simple rotation

The simple tree rotation used in treaps is also applied at the root of the splay tree, moving the splayed node x up to become the new tree root. Here we have $A < x < B < y < C$, and the splayed node is x in the tree on the left and y in the tree on the right. The splayed node depends on which direction the rotation is.



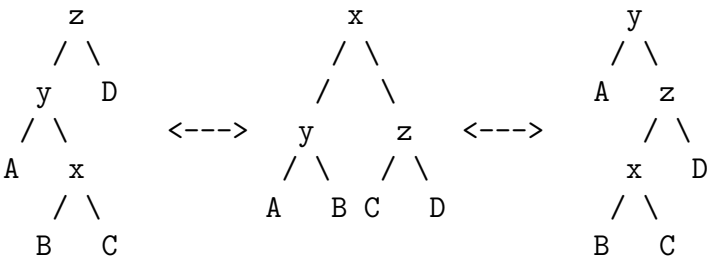
Rotation 2: Zig-Zig and Zag-Zag

Lower down in the tree rotations are performed in pairs so that nodes on the path from the splayed node to the root move closer to the root on average. In the *zig-zig* case, the splayed node is the left child of a left child or the right child of a right child (*zag-zag*).



Rotation 3: Zig-Zag

In the *zig-zag* case, the splayed node is the left child of a right child or vice-versa. The rotations produce a subtree whose height is less than that of the original tree. Thus, this rotation improves the balance of the tree. In each of the two cases shown, x is the splayed node:



See this page for a nice visualization of splay tree rotations and a demonstration that these rotations tend to make the tree more balanced while also moving frequently accessed elements to the top of the tree.

The classic version of splay trees is an imperative data structure in which tree rotations are done by imperatively updating pointers in the nodes. This data structure can only implement a mutable set/map abstraction because it modifies the tree destructively.

Unlike the classic imperative splay tree, your code should build a new tree upon insertion, deletion, and lookup. **Note: You may not use any imperative features for this section.**

Exercise 4: Splay.

Many of the operations performed on splay trees require splaying. You will first implement splay - you may find the rotations described above to be useful.

TODO: Implement `splay` as defined `splay.mli`.

Exercise 5: Lookup.

Remember that when an element has been looked up, it is splayed to the top of the tree. If the element does not exist in the tree, splay the last non-leaf node reached during lookup.

TODO: Implement `lookup` as defined in `splay.mli`.

Exercise 6: Insertion.

Splay trees use the same method as binary search trees to insert nodes with one exception - after the item is inserted, a splay is performed to move the inserted node to the top of the tree.

TODO: Implement `insert` as defined in `splay.mli`.

Exercise 7: Deletion.

If the removed node is not the root, the parent of the removed node is splayed to the top of the tree. Additionally, if the element does not exist in the tree, the last non-leaf node encountered during search is splayed. Otherwise, splay trees use the same method as binary search trees to delete nodes.

TODO: Implement `delete` as defined in `splay.mli`.

Exercise 8: Proving Correctness.

[written] Prove the following:

Given an n-node splay tree t and an element x, `insert t x` returns a tree t' that satisfies the binary search tree invariants. That is, each node in t' has two distinguished subtrees, and each node's value is greater than all the values stored in the left subtree and less than all the values stored in the right subtree.

Comments

[written] At the end of the file, please include any comments you have about the problem set, or about your implementation. This would be a good place to list any problems with your submission that you weren't able to fix or to give us general feedback about the problem set.

Release files

The accompanying release file `ps4.zip` contains the following files:

- `writeup.pdf` is this file.
- Folder `geometry` contains files needed in the first part of the problem set.
  - `fields.mli` and `fields.ml` contain the interface and implementation of 3 different ordered fields.
  - `geometry.mli` and `geometry.ml` contain the interface and implementation of the solution to the 2 geometry problems.
  - `utils.ml` contains some utility functions for the real module. These functions are handy in implementing test cases in `tests.ml`
  - `.ocamlinit` and `.cs3110` contain configuration options to include the `nums` package when running in `utop` or compiling with `cs3110 compile` without needing to specify it explicitly.
- Folder `splaytree`
  - `splay.mli` and `splay.ml` contain the interface and implementation of splay trees.