Data Structures and Functional Programming	**Problem Set 3**
CS 3110, Spring 2014	**Due: March 6**
Version: 0	**Last Modified: February 21, 2014**

Overview

The exercises in this assignment illustrate the use of the OCaml module system, simple data structures, the substitution model, and formal proofs using structural induction.

Objectives

- Implement functional data structures such as trees and priority queues.
- Gain familiarity implementing larger applications out of modular components.
- Build a compression application based on Huffman trees.
- Use the substitution model and structural induction to prove programs equivalent.

Recommended reading

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

- Lectures 6 and 7
- Recitations 6 and 7
- The CS 3110 style guide
- The OCaml tutorial
- Real World OCaml, Chapters 1-10

What to turn in

You should submit your solutions in files `pQueue.ml`, `solver.ml`, `huffman.ml`, and `reasoning.pdf`. Any comments you wish to make can go in `comments.txt` or `comments.pdf`. If you choose to submit any Karma work, you may submit the file `karma.ml` (be sure to describe what you've done in the comments file).

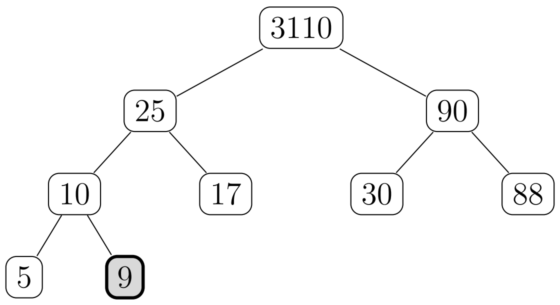


Figure 1: A binary heap. The last node is highlighted.

Part One: Priority Queues

A priority queue is an abstract data type that supports two operations over an ordered collection of elements: (i) inserting an element and (ii) removing the largest element. A common way to implement a priority queue is using a binary heap. Recall that a binary heap is a binary tree satisfying two additional properties:

Completeness: Every level of the tree, with the possible exception of the lowest level, is completely filled, and the nodes in the last level are as far left as possible. We refer to the rightmost of these nodes as the **last** node.

Heap property: The value stored at each non-leaf node is greater than or equal to the values stored in its children.

Given a binary heap, the operations of a priority queue can be implemented as follows:

Insertion: To insert a value `x` into an empty heap, create a tree with `x` as the root and two empty children. To insert `x` into a non-empty heap `h` whose current root is `y`, recursively insert the smaller of `x` and `y` into the child of `h` that will contain the new last node.

For example, when inserting the value 20 into the heap in figure 1, the new last node will be the left subchild of the node numbered 17; so the smaller of 20 and 3110 (i.e., 20) is recursively inserted into the subtree with root 25.

Removal: By construction, the value that should be returned is always located at the root. However, removing the root produces two binary heaps. To merge these into a new binary heap, replace the root value with the value in the last node. If the resulting tree does not satisfy the heap property, then repair it by swapping the root value with the value stored at the root of the larger child, and recursively repair that subtree.

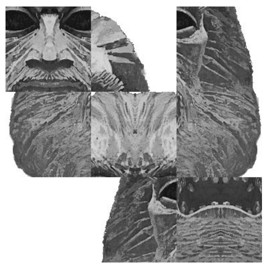
For example, when removing the maximum element from the heap depicted in figure 1, we begin by replacing the root (3110) with the last element (9). Then we swap the root (9) with the larger of its children (90). This causes the right subtree to violate the heap property, so we recursively repair the right subtree.

Exercise 1:

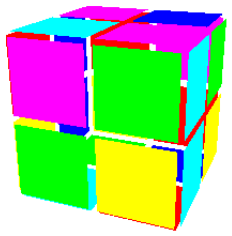
[code] Implement the `PQueue.HeapImpl` module using a binary heap as the data structure. Your implementation should match the specification given in `pQueue.mli`.

To help you, we have provided a function `path_to_last` that computes the path from the root to the rightmost leaf of a complete binary tree of a given size.[1] For example, the heap in Figure 1 contains 9 nodes, and the last node can be reached from the root by going left, then left, then right. Consequently `path_to_last 9` returns `[Left; Left; Right]`. You may find this function helpful for both insertion and removal: for the former you need to push the inserted element towards the last node, while for the latter you need to swap the root with the last node.

[1]Astute observers will note that `path_to_last` is very similar to `int_to_bits` from problem set 2.



(a) Sliding tile puzzle.



(b) Rubik's cube.

Figure 2: Two puzzle games.

Part Two: Puzzle Solver

The goal of many puzzles is to find a sequence of moves that transform an initial state into a final state. For example, in the sliding tile puzzle (Figure 2a), the goal is to transform a scrambled matrix of tiles into an unscrambled one by shifting tiles one at a time, while in a Rubik's cube (Figure 2b), the goal is to transform a cube whose faces contain randomly colored squares into a cube whose face contains squares of a single color using rotations.

Abstractly, these puzzles can be represented as (very large!) graphs whose nodes correspond to configurations of the puzzle, and edges correspond to feasible moves between states. A solution to the puzzle is a path from the initial to final state in the graph. Figure 3 depicts a portion of the graph for the sliding tile puzzle.

In this exercise you will implement a puzzle solver that searches for a winning strategy in an arbitrary graph. Your solver will be generic in the sense that it will work on the abstract representation of a puzzle as a graph, so it will not need to know about the specific details of the puzzle being solved. Hence, it will be possible to use the same solver with sliding tile puzzles, Rubick's cubes, and others.

A simple way to search in a graph is to use depth-first or breadth-first search. However, although these traversal strategies are guaranteed to find a solution (if one exists), because the graphs representing a puzzle tend to be very large, traversing the entire graph is not feasible in general. A much better idea is to implement a guided search strategy that explores the most promising paths first.

Such a search proceeds in the same manner as DFS or BFS: it maintains a worklist of nodes to visit and a list of the nodes that have been seen so far. At each step, it removes an element from the worklist and processes it. However, unlike DFS (which removes the most recently added node) or BFS (which removes the least recently added node), guided search removes the most promising node. To break ties, if two nodes are equally promising, it removes the node with the shorter overall path from the initial node. This strategy is sometimes called best-first search.

A good way to maintain the worklist for best-first search is to use a priority queue ordered by goodness. We recommend that you re-use `PQueue.ListImpl` or `PQueue.HeapImpl` from the last exercise.

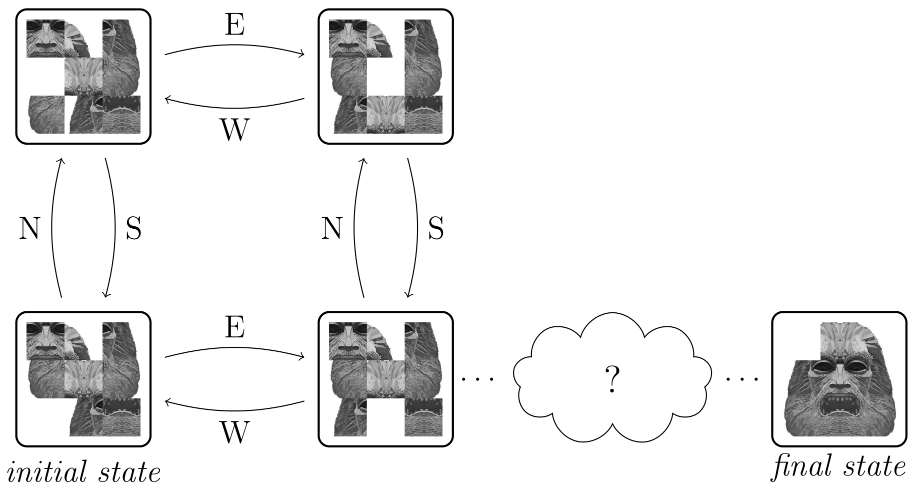


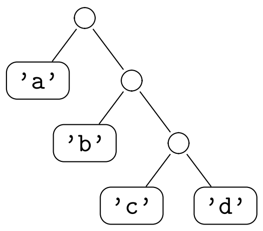
Figure 3: Partial graph for the sliding tile puzzle. The edges between nodes are labeled by the compass direction that the "open" tile needs to be shifted to move between the nodes.

Exercise 2:

[code] Implement the `solve` function in the `Solver.Make` functor (in the file `solver.ml`). It takes a puzzle state that has been instantiated in a non-goal state, and outputs `Some l`, where `l` is a list of moves that solve the puzzle, or `None` if no solution exists. Your solver should make use of the types and functions in the `Solver.PUZZLE` signature which define the graph representation of a puzzle, and the heuristic function for determining whether a node is promising. See the documentation provided in `solver.mli` for further details.

We have provided implementations of the sliding tile puzzle and the Rubik's cube, along with graphical interfaces for visualizing games being played. You can run them by running `cubeMain.ml` and `tileMain.ml` respectively (using the `cs3110` tool!). These programs generate a random puzzle, invoke your solver to solve it, and then display an animation of the solution. To run the animations, you must update the `cs3110` tool and install the SDL graphics library. To do so, execute the following commands from the command-line in the virtual machine:

```
% cd 3110-tools/cs3110-cli/student
% git pull
% make install
% sudo apt-get install libsdl1.2-dev libsdl-image1.2-dev libsdl-ttf2.0-dev
% opam install ocamlsdl
```



Character	Encoding
'a'	0
'b'	10
'c'	110
'd'	111

Figure 4: A Huffman tree and the corresponding encoding

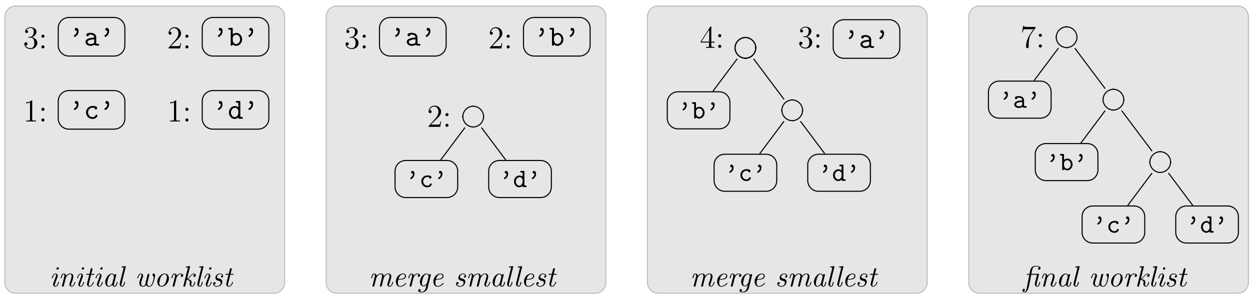
Part Three: Huffman Coding

Huffman coding is a data compression technique that analyzes the frequencies of bytes occuring in a string. Characters that occur frequently are encoded using shorter bitstrings, while those that occur infrequently are encoded using longer bitstrings.

A huffman encoding can be represented as a binary tree, often called a **Huffman tree**, with characters at the leaves. The encoding of a given character is given by the path from the root to that character: each left branch adds a 0 bit to the bitstring and each right branch adds a 1 bit. Figure 4 depicts a Huffman tree and the corresponding encoding of a single character.

Huffman tree encodings have the property that a bit sequence formed by concatenating the encodings of letters can be uniquely decoded. For example, using the encoding represented by figure 4, the bit string `1100111010100` is decoded as `"cadabba"`.

If the frequencies of the characters in a file are known, an optimal Huffman tree can be built using a simple greedy algorithm that maintains a priority queue of partial Huffman trees. Initially the queue contains a leaf for each character, with its priority given by the frequency of that character in the original string. Then, at each step, we remove the two trees with the lowest frequencies, and combine them into a single tree. This tree is then reinserted into the queue with priority given by the sum of their frequencies. This process is repeated until only a single tree remains; this tree is the optimal Huffman tree for the given input frequencies. For example, the tree in figure 4 can be built from the string `"cadabba"` using the following sequence of steps:



Exercise 3:

[code] In the file huffman.ml, implement the build_tree function to construct a Huffman tree from the provided character list. If the input list is empty, the build_tree function should return Empty. It should use one of the implementations of priority queues: either PQueue.ListImpl or PQueue.HeapImpl.[2]

Exercise 4:

[code] Implement the encode and decode functions in huffman.ml. These functions are specified in detail in huffman.mli. We have provided you with driver programs compress.ml and decompress.ml that compress and decompress whole files. You can test your implementation by running these commands using cs3110 tool. For example:

```
> cs3110 compile compress.ml
> cs3110 compile decompress.ml
> cs3110 run compress.ml ../data/Conrad.txt > Conrad.huff
> du -b ../data/Conrad.txt
213304
> du -b Conrad.huff
118727
> cs3110 run decompress.ml Conrad.huff > Conrad.out
> diff -s ../data/Conrad.txt Conrad.out
Files ../data/Conrad.txt and Conrad.out are identical
```

This example shows that the compressed file is about 55% of the size of the original. This is a typical savings for Huffman coding on large files; smaller files typically get less benefit from compression because there is less diversity in character frequency and because the Huffman tree itself must also be stored in the file. The data/ directory contains a number of compressed and uncompressed files that you can use to test your implementation.

[2]Note that because the remove operation returns the greatest element, you will have to invert the usual ordering on integers so that smaller frequences rank higher.

Part Four: Formal Reasoning

In this part of the assignment, you will investigate proofs of program equivalence using structural induction and the substitution model. Your solutions to these exercises should be submitted in a single file `reasoning.pdf` that can be created using any software you like.

Exercise 5:

[written] Recall the binary tree data type:

```
type 'a bintree =
    Leaf
  | Node of 'a bintree * 'a * 'a bintree
```

Let $P(t)$ be an arbitrary predicate on binary trees. State the structural induction principle for the `'a bintree` datatype as a logical formula of the form "$(\ \ldots\ ) \Longrightarrow \forall t.\, P(t)$."

Exercise 6:

[written] Consider the following recursive function,

```
let rec tree_sum =
  (fun t ->
    match t with
     | Leaf -> 0
     | Node(l,x,r) -> x + (tree_sum l) + (tree_sum r)) in
tree_sum t
```

and an equivalent function written using `tree_fold`:

```
let rec tree_fold =
  (fun a -> fun f -> fun t ->
    match t with
     | Leaf -> a
     | Node(l,x,r) -> f x (tree_fold a f l) (tree_fold a f r)) in
tree_fold 0 (fun x l r -> x + l + r) t
```

Assume that we have already generated fresh variables `tree_sum'` and `tree_fold'`, and added new reduction rules to handle the recursive definitions:

```
tree_fold' ⟶ (fun a -> fun f -> fun t ->
                  match t with
                       | Leaf -> a
                       | Node(l,x,r) -> f x (tree_fold' a f l) (tree_fold' a f r))

tree_sum' ⟶ (fun t ->
                  match t with
                    | Leaf -> 0
                    | Node(l,x,r) -> x + (tree_sum' l) + (tree_sum' r))
```

Give a careful proof of the following equivalence:

```
let tree_fold = tree_fold' in tree_fold 0 (fun x l r -> x + l + r) t =
let tree_sum = tree_sum' in tree_sum t
```

You should show that both sides of the equality reduce to the same value in the substitution model. To complete the proof, you will likely need to prove a lemma $\forall t.\, P(t)$, where $P(t)$ is the predicate:

$P(t) \triangleq$

```
(fun a -> fun f -> fun t ->
     match t with
       | Leaf -> a
       | Node(l,x,r) -> f x (tree_fold' a f l) (tree_fold' a f r))
0 (fun x l r -> x + l + r) t
=
(fun t ->
     match t with
       | Leaf -> 0
       | Node(l,x,r) -> x + (tree_sum' l) + (tree_sum' r))
t
```

Your proof of this lemma should use the structural induction principle on binary trees.

Getting Started

Because this is a larger assignment, we have provided you with a lot of code to get started.

- `writeup/` this writeup.
- `doc/` automatically generated ocamldoc documentation for the starter code provided in `release`.
- `data/` data files for testing your Huffman implementation
- `release/`
  - `huffman.{ml,mli}`, `pQueue.{ml,mli}`, `solver.{ml,mli}` modules to implement.
  - `animation.{ml,mli}`, `tilePuzzle.{ml,mli}`, `cubePuzzle.{ml,mli}`, `zardoz.png` support files for the animations for the puzzle solver.
  - `compress.ml`, `decompress.ml`, `tileMain.ml` `cubeMain.ml` driver programs for testing your Huffman and solver implementations.
  - `util.{ml,mli}` utility functions you may find useful.

Comments

We would like to know how this assignment went for you. Were there any parts that you didn't finish or wish you had done in a better way? Which parts were particularly fun or interesting? Did you do any Karma problems?

Karma suggestions

- Implement more puzzles for your solver. If you're feeling adventurous, add animations!
- For the Rubik's cube puzzle, we intentionally generated inputs with short solutions (fewer than 8 moves) because the search space grows exponentially. Similarly, we only generate 8-tile puzzles. Performance can be improved, by using better heuristics, algorithms, or data structures. Implement a solver that is efficient enough to handle much larger instances of the puzzles.
- See what impact the OCaml native code compiler `ocamlopt` has on the performance of your priority queue implementations.
- The Wikipedia entry on Huffman coding contains a wealth of variations on Huffman encoding. Find one that interests you and implement it.
- We did not specify how to break ties when selecting trees in the `build_tree` function, so there are multiple valid implementations. Determine whether different choices can produce encodings with a different length.