15-122: Principles of Imperative Computation, Fall 2010
Assignment 6: Trees and Puzzle Solvers

William Lovas (wlovas@cs) Tom Cortina (tcortina@cs)

Out: Wednesday, October 27, 2010
Due: Tuesday, November 2, 2010

(Written part: before lecture,
Programming part: 11:59 pm)

1 Written: (20 points)

The written portion of this week's homework will give you some practice reasoning about red-black trees and tries. You can either type up your solutions or write them *neatly* by hand, and you should submit your work in class on the due date just before lecture begins. Please remember to *staple* your written homework before submission.

1.1 Rotations and Red-Black Trees

Exercise 1 (3 pts)**.** Consider the following function to perform a rotate right operation:

```
tree tree_rotate_right(tree T)
{
    tree root = T->left;
    T->left = root->right;
    root->right = T;
    return root;
}
```

Give `requires` and `ensures` annotations for the `tree_rotate_right` function. Your annotations should include tests on the ordering of the tree and the height of the tree, before and after the rotation. You may assume the following functions are defined for you:

```
bool is_ordered(tree T, elem min, elem max);
int height(tree T);
```

Exercise 2 (3 pts)**.** In an n-node binary search tree, how many single rotations are possible? Explain.

Exercise 3 (3 pts)**.** Show the red-black tree that results after successively inserting the following keys (in the order shown) into an initially empty red-black tree. Refer to the code given in Lecture 17 on the course website.

```
52  49  42  23  30  19
```

Your answer should show the tree after each key is successfully inserted. Also be sure to label each node with "B" or "R" for black or red, respectively.

Exercise 4 (3 pts)**.** Using the invariants for a red-black tree, explain why the depth of any leaf in a red-black tree is at most twice the depth of the leaf closest to the root.

1.2 Tries

Exercise 5 (4 pts)**.** Show the ternary search trie (TST) that results after successfully inserting the following strings (in the order shown) into an initially empty TST. Refer to the code given in Lecture 18 on the course website.

```
car, cart, carts, cope, cup, cut, cute, cab, cop, care
```

Exercise 6 (4 pts)**.** Write a function that prints the strings stored in a TST in alphabetical order. (**Hint:** You will need a recursive helper function.) Your function should work with the code given in Lecture 18 on the course website. Remember that you should be able to write this by hand to test your problem-solving skills. Then you can test it if you wish on the computer.

2 Programming: Lights Out! (30 points)

For the programming portion of this week's homework, you'll implement a brute-force solver for a puzzle game called Lights Out. You'll make use of a "map" abstraction to remember what states you've seen, and you'll have a chance to experiment with several data structures implementing it, exploring the fundamental idea of abstraction: separating interface from implementation.

For this assignment, you'll produce several C_0 files, so each task is labelled with the file to which it pertains. See Figure 2 in the Appendix for a table listing all the tasks and their associated files. (Don't be alarmed! Although there are many tasks for this assignment, most of them are relatively short.)

You should submit your code electronically by 11:59 pm on the due date. Detailed submission instructions can be found below.

Starter code. Download the file `hw6-starter.zip` from the course website.

Compiling and running. For details on how we will compile your code, see the file `COMPILING.txt` included in the starter code. Don't forget to include the `-d` switch if you'd like to enable dynamic annotation checking. **Warning:** *You will lose credit if your code does not compile.*

Submitting. Once you've completed some files, you can submit them by running the command

```
handin -a hw6 <file1>.c0 ... <fileN>.c0
```

You can submit files as many times as you like and in any order. When we grade your assignment, we will consider the most recent version of each file submitted before the due date. If you get any errors while trying to submit your code, you should contact the course staff immediately.

Annotations. Be sure to include appropriate `//@requires`, `//@ensures`, `//@assert`, and `//@loop_invariant` annotations in your program. You should write these as you are writing the code rather than after you're done: documenting your code as you go along will help you reason about what it should be doing, and thus help you write code that is both clearer and more correct. **Annotations are part of your score for the programming problems; you will not receive maximum credit if your annotations are weak or missing.**

Style. Strive to write code with *good style*: indent every line of a block to the same level, use descriptive variable names, keep lines to 80 characters or fewer, document your code with comments, etc. We will read your code when we grade it, and good style is sure to earn our good graces. Feel free to ask on the course bboard (`academic.cs.15-122`) if you're unsure of what constitutes good style.

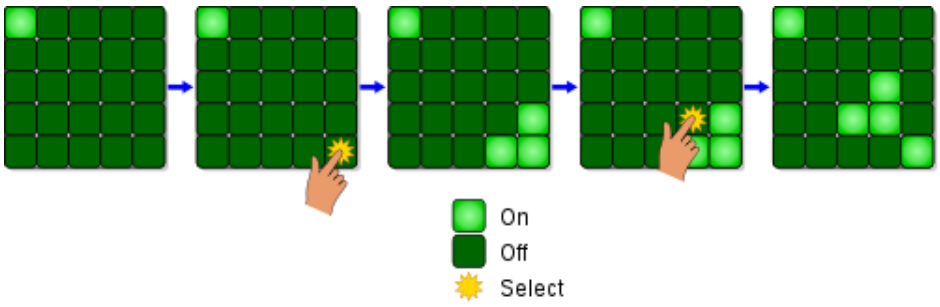

Figure 1: A sequence of moves in Lights Out. (Image: Wikipedia)

2.1 Lights Out

Lights Out is an electronic game consisting of a grid of lights, usually 5 by 5. The lights are initially presented in a random pattern of on and off, and the objective of the game is to turn all the lights off. The player interacts with the game by touching a light, which toggles its state and the state of all of its cardinally adjacent neighbors. It is often quite tricky to see how to solve a given configuration, if it is solvable at all, since the path to the solution may involve seemingly-backward progress as lights must be turned on to point the way towards a final all-off state.

A simple solver would proceed by a brute-force search similar to the maze search you implemented in Homework 4, but with one essential difference: the graph is not explicitly represented. As a consequence, many things that we leveraged the graph to represent (e.g., neighbors of a node, the search-order predecessor of a node) must be handled externally instead using auxiliary computations and data structures. In the remainder of this assignment, you'll develop a Lights Out solver incrementally and then evaluate its performance using several different underlying data structures.

2.2 Representing the Board

To warm up, we'll briefly discuss the representation of a Lights Out board. We represent a Lights Out board as a bit array along with its width and height. We represent a bit array in packed form as a C_0 integer.

```
typedef int bitarray;

typedef struct board* board;

struct board {
    int width;
    int height;
    bitarray lights;
};
```

Since the bit array representing the state of the lights is given as a 32-bit `int`, it is important that the total area of the board be less than or equal to 32, an invariant that's checked by the specification function `is_board`.

We represent the state of the board as an integer rather than as an array of boolean values for two reasons. First, we will soon wish to store the state in various data structures, and we want to be careful not to let any other piece of code change the state after its been stored in a data structure. By convention, arrays are treated as highly mutable, so we would not expect to count on an array not changing unless we made our own separate copy of it. And second, making many copies of an array puts a significant drain on our space performance, which as discussed in class can have a serious impact on run-time performance due to effects like cache locality.

Bit arrays support operations similar to ordinary arrays, but in a "value-oriented" fashion: to "update" a bit array, a function must return a new bit array. The operations of getting the value of a bit, setting a bit to a particular value, and flipping a bit are easily implemented using bitwise operators. We interpret the indices starting from the least significant bit, such that an index i refers to the 2^i bit of the integer when interpreted unsigned.

$$b_{31}b_{30}\ldots b_2b_1b_0$$

Task 1 (3 pts, `board.c0`)**.** Implement the functions `bitarray_get`, `bitarray_set`, and `bitarray_flip` as specified in `board.h0`.

The bit array is interpreted as a grid in the usual fashion: position (x, y) of a $w \times h$ grid is b_{wy+x}. For example, if `0` represents a light in the "off" state and `#` represents a light in the "on" state, the board

```
#0#
0##
#00
```

would be represented as 001110101, i.e., 117.

2.3 The "Map" Abstraction

In this section, you'll show how hash tables and binary search trees support a common interface, that of a "map" associating keys to elements.

First, recall the common pattern we have seen over and over again in our data structure implementations, where a library requires some code to be specified by the client in order to be complete. For example, the hash table code we wrote in Lecture 11 was completely independent of the actual type of elements, provided that they supported the operations of key extraction, key equality, and key hashing. So to actually build a real hash table, the client has to specify an element type and definitions for the three operations.

When we were storing elements that contained `string`s as `key`s and associated integer `count`s, we wrote the following type definitions and functions to complete the hash table implementation:

```
typedef key string;
typedef struct elem* elem;

struct elem {
  string word;
  int count;
};

key elem_key(elem e) {
  return e->word;
}

bool equal(key s1, key s2) {
  return string_equal(s1, s2);
}

int hash(string s, int m)
//@requires m > 1;
//@ensures 0 <= \result && \result < m;
{
  int a = 1664525; int b = 1013904223; /* inlined random number generator */
  int r = 0xdeadbeef;                  /* initial seed */
  int len = string_length(s);
  char[] C = string_to_chararray(s);
  int i; int h = 0;             /* empty string maps to 0 */
  //@assert \length(C) == len+1;
  for (i = 0; i < len; i++)
  //@loop_invariant 0 <= i && i <= \length(C);
  {
    h = r*h + char_ord(C[i]); /* mod 2^32 */
    r = r*a + b;              /* mod 2^32, linear congruential random no */
  }
  h = h % m;                    /* reduce to range */
  //@assert -m < h && h < m;
  if (h < 0) h += m;            /* make positive, if necessary */
  return h;
}
```

Recall that we used an inlined pseudorandom number generator to "smear" strings uniformly over all possible hash values. The specially chosen constants `a = 1664525` and `b = 1013904223` ensure that small changes in the input string result in unpredictable changes in the hash value. Also note that we didn't write a `compare` function for hash tables since they only require equality, but binary search trees need a total

order comparison.

In what follows, we'll map a game state to an (x, y) move and the preceding state: `key`s are `bitarray`s, and `elem`s contain a key state, an (x, y) position, and a preceding `bitarray` state. From `board.h0`:

```
typedef bitarray key;
typedef struct elem* elem;

struct elem {
    bitarray state;
    int x;
    int y;
    bitarray previous;
};
```

Task 2 (3 pts, `board.c0`)**.** Implement the "client code" functions required to use the `elem` type defined in `board.h0` with hash tables and binary search trees: `elem_key`, `equal`, `hash`, and `compare`. For a reminder of the types and contracts of the functions, see `hashtables.h0` and `bst.h0`. When implementing `hash`, be sure to use some form of randomness to "smear" keys uniformly across hash values.

An essential concept in computational thinking is *abstraction*: the separation of *interface* from *implementation*. When a program is composed of many independent components whose boundaries are mediated by carefully specified interfaces, then the program can be updated in a modular fashion: at any time, one implementation of a component can be replaced by another without changing the overall meaning of the program, provided that the new implementation adheres to the same interface as the old. Conversely, when interfaces are left unspecified or violated, things can go horribly awry: arguably, some of the most egregious software errors of all time were caused by a careless confusion of interface and implementation.

We can demonstrate the power of separating interface from implementation by observing that two data structures we've defined over the course of the semester implement the same interface: hash tables and binary search trees both embody the idea of a map between keys and values. We can make this observation precise by delineating a precise `map` interface and showing how to implement it using either data structure. We show a sample of the interface here; the full specification including function contracts is found in the file `map.h0`.

```
typedef struct map* map;

bool is_map(map M);
map map_new();
bool map_contains(map M, key k);
elem map_search(map M, key k);
void map_insert(map M, elem e);
```

The `elem` and `key` types are the familiar "client code" types we saw above.

Task 3 (3 pts, `hashmap.c0`)**.** Implement the `map` interface as specified in `map.h0` using hash tables. Be sure your implementation satisfies the invariants specified in the interface! For reference, the hash table interface can be found in the file `hashtables.h0`.

Task 4 (3 pts, `treemap.c0`)**.** Implement the `map` interface as specified in `map.h0` using binary search trees. Be sure your implementation satisfies the invariants specified in the interface! For reference, the binary search tree interface can be found in the file `bst.h0`.

2.4 Solving the Puzzle

Employing computational thinking once again, we can view the problem of searching for a solution to a Lights Out puzzle as the problem of searching for a path in a graph:[1] the nodes of the graph are board states, and the neighbors of a node are all the board states that are reachable with a single move. This game graph is enormous, though—there are 2^{25} different possible game states in the usual 5×5 game, and each state has 25 neighbors, one for each light. So when we search through this graph, we do not represent the graph explicitly in memory. Instead, we compute pieces of it lazily as we require them, and consequently, we must make use of some interesting data structures to support our search.

In order to find shortest solutions, we'll implement breadth-first search using a queue. The overall idea behind the algorithm is quite close to what you implemented for searching through a maze in Homework 4, and your code will consequently look quite similar. Instead of searching for a path through a maze, though, you are searching for a sequence of moves (x, y) that lead from an initial state to the all-"off" state. Instead of having a neighbors list at each node, you will compute the neighbors by considering every possible move you might make. And instead of maintaining a "visited" mark and a search predecessor for each node, you will maintain a map from board states to the moves and previous states which led to them.

To remind you how the search works, here is a high-level sketch. Throughout the algorithm you keep a queue of states to return to and a map from states to moves and previous states.

1. Begin by enqueueing the initial board state. Add it to the map as well, mapping it to a sentinel move $(-1, -1)$.

2. Repeat the following as long as the work queue isn't empty: dequeue a state, compute all of its neighboring states, and for each one you haven't already seen, enqueue it and add it to the map, mapping it to the move that got to it and the current state.

[1] Recall that a graph is a collection of *nodes* and *edges*, where each edge connects a pair of nodes.

3. If you ever encounter a winning state—all lights in the "off" position—reconstruct and return a winning sequence of moves by working backward through the solution using the map.

Task 5 (10 pts, `lightsout.c0`)**.** Implement the solver outlined above as the function `solve` specified in `lightsout.h0`. Be sure to include annotations for the function's preconditions, postconditions, and for any relevant invariants in its body. You may find it convenient to use some of the functions defined in `board.h0`, but you are not required to.

Thanks to abstraction, your implementation of the solution algorithm is actually three implementations! Depending on which implementation of the "map" interface you compile against, your algorithm will have very different performance characteristics.

Task 6 (5 pts, `README.txt`)**.** Empirically evaluate the performance of your solver using three different implementations of the "map" abstraction: hash tables, unbalanced binary search trees, and red-black trees. Compare the behavior on several example inputs and record the execution times in a file `README.txt`. Then write a short summary of your thoughts with respect to the performance of your solver using each map abstraction. Which implementation performs best? Does it matter whether the input has a solution? Speculate on reasons for any differences you find. Remember to compile *without* `-d` when you're comparing timings! (**Note:** This task is intentionally open-ended, but it is worth a full 5 points, so be sure to give it some thought.)

2.5 Boards from External Input

When testing your code, it may be convenient to read starting boards from files. We can represent boards in files just as we showed above: each "off" light is an `0` character and each "on" light is a `#` character, and each row is a single string with no spaces on its own line. Here are three sample boards included with the starter code: a single light on in the center, the four corners lit up, and a small, empty square:

```
00000    #000#    00000
00000    00000    0###0
00#00    00000    0#0#0
00000    00000    0###0
00000    #000#    00000
```

Task 7 (3 pts, `board.c0`)**.** Implement a `read_board` function as specified in `board.h0` that reads a board from a file in the format described above. You may use anything from `readfile.c0` included in the starter code, or you may use the `file` library directly, whichever you prefer. Be sure your implementation satisfies the specified postcondition! If you're stumped, look to previous assignments or code from lecture for examples of reading from files.

2.6 Optional: Judges' Prize

The brute-force exhaustive approach you're asked to implement above is quite naive. Although it produces short solutions, it sometimes takes a very long time to produce them! Design and implement a better algorithm that can solve larger problems efficiently. Document your solution and the optimizations you implement. You may find some useful domain-specific insights in Sutner's works [1, 2].

References

[1] Klaus Sutner. Linear cellular automata and the Garden-of-Eden. The Mathematical Intelligencer, 11(2):49–53, 1989.

[2] Klaus Sutner. The σ-game and cellular automata. The American Mathematical Monthly, 97(1):24–34, January 1990.

A Appendix: Files for each task

Task	Content of Task	File
Task 1	Bitarray ops: `bitarray_*`	`board.c0`
Task 2	"Client code" for maps	`board.c0`
Task 3	`hashmap` implementation	`hashmap.c0`
Task 4	`treemap` implementation	`treemap.c0`
Task 5	Lights Out solver: `solve`	`lightsout.c0`
Task 6	Empirical analysis of solver	`README.txt`
Task 7	File input: `read_board`	`board.c0`

Figure 2: Tasks in this homework and the files they should go in.