15-122: Principles of Imperative Computation, Spring 2011 Assignment 5: Hash Tables, Heaps, and Puzzle Solvers

Jacob Potter (jdpotter@andrew) Sri Raghavan (srikrish@andrew)
William Lovas (wlovas@cs) Tom Cortina (tcortina@cs)

Out: Sunday, March 6, 2011
Due: Thursday, March 17, 2011

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

1 Written: (25 points)

This week's written portion will give you practice with formalizing data structure invariants, as well as heaps and priority queues. 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 Hash Tables, Revisited

Refer to the code (below) from the hash table interface we wrote in lecture. We wrote a specification function, `is_ht`, to check that a given `ht` is actually a hash table.

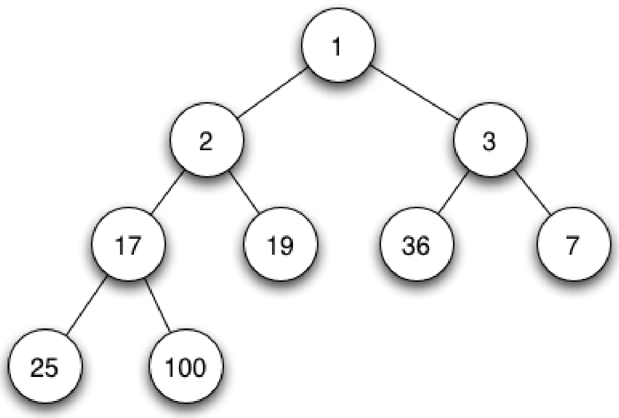
```
bool is_ht(ht H) {
    if (H == NULL) return false;
    if (!(H->size > 0)) return false;
    //@assert H->size == \length(H->A);
    return true;
}
```

An obvious invariant of our hashtables is that every element of a chain hashes to the index of that chain. This specification function is incomplete, then: we never test that the *contents* of the hash table hold to this invariant. That is, we test only on the `struct ht`, and not the properties of the array within.

Exercise 1 (4 pts)**.** Extend `is_ht` from above, implementing code to check that every element in the hash table matches the chain it is located in, and that each chain is non-cyclic. You may find the circularity-checking code we wrote in lecture useful.

1.2 Heaps

We represent heaps, conceptually, as trees. For example, take the min-heap below. (Image: Wikipedia. Stolen shamelessly by Sri.)



Exercise 2 (3 pts)**.** We can perform operations with respect to the heap above.

(a) What is the result of `insert`ing an element with value 0 into the heap? Draw your answer as a tree, of the same form. You may find it helpful to write out the steps of the algorithm, but only submit the final result.

(b) From the original tree, what is the result of a `delete_min` operation? Draw your answer as another tree. Again, you may find it helpful to write out the steps, but only submit the final result.

(c) Again consider the original tree, and the method by which we represent it in a program (as an array). At what index is the element with value 36 stored? At what index is its parent stored?

Exercise 3 (5 pts)**.** Consider runtime complexity (using big-O notation) with regards to heaps.

(a) We have n elements. What is the runtime complexity of building a heap from these elements by adding them one at a time to an array? Assume that the array is large enough to hold the final heap. Justify your answer.

(b) With the same n elements, what is the runtime complexity of building a heap by adding them one by one to an unbounded array (uba) of initial length 1? Justify your answer using amortized analysis on the uba. **Hint:** What is the amortized cost of a single insert?

Exercise 4 (2 pts)**.** Given a min-heap (that is, every element is less than or equal to its children) how might we find the maximum element? Describe and explain an algorithm to do so; a high-level sketch is sufficient. **Hint:** think about the heap invariant and how our implementation represents a heap in memory.

Exercise 5 (2 pts)**.** True or false: an array that is sorted from smallest to largest is also a min-heap. Explain.

Exercise 6 (2 pts)**.** We can implement a sorting algorithm using a heap. Explain, on an algorithmic level, how this might work (no code is necessary). Analyze the runtime complexity of this algorithm.

1.3 Priority Queues

Exercise 7 (2 pts)**.** How might you use a priority queue as an ordinary FIFO queue, so that the enq and deq operations are as efficient as possible? Explain. If necessary, you can add additional data to the priority queue structure. What is the runtime complexity of each of these operations?

Exercise 8 (2 pts)**.** How does a priority queue (implemented with a heap, as in class) behave if all of the elements added have identical priorities? That is, can we draw parallels to a data structure we've already covered in lecture? Explain.

1.4 Unrelated (But Important) Loop Invariant Practice

Below is code for a simple iterative factorial function.

```
int factorial(int n)
//@requires n >= 0;
{
    int i;
    int t = 1;
    for (i = 0; i < n; i++)
    //@loop_invariant ________________ ;
    {
        t = (i+1) * t;
    }
    return t;
}
```

We define the factorial sequence (f_n) mathematically using a recurrence:

$$f_n = \begin{cases} 1 & \text{if } n = 0 \\ n * f_{n-1} & \text{if } n > 0 \end{cases}$$

and this recurrence can be represented in C_0 by a recursive function:

```
int f(int n)
//@requires n >= 0;
{
    if (n == 0)
        return 1;
    else //@assert (n > 0);
        return n * f(n-1);
}
```

Exercise 9 (1 pt)**.** Add *useful* loop invariants to the above `factorial` function (you may use as many lines as necessary).

Exercise 10 (2 pts)**.** Prove (using your loop invariants) that the value returned from `factorial(n)` (for arbitrary $n \geq 0$) is in fact equal to f_n.

2 Programming: Lights Out! (25 points)

For the programming portion of this week's homework, you'll improve the hash table implementation discussed in lecture in two important ways. Then, you'll implement a brute-force solver for a puzzle game called Lights Out. You'll reuse the same hash table code that we previously used to map strings to word counts, but instead storing a mapping involving moves and game states. This code reuse depends on 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 `hw5-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.* Remember that you should make a habit of developing your code with annotation checking enabled.

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

```
handin -a hw5 <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.

2.1 Resizing

The hash table code that we have handed out is incomplete in several important fashions. First, the array `A` is allocated with a specified initial size and never grows. This means that as the number of elements in the hash table becomes very large, the load factor will also grow very high. If `A` is of fixed size m, then increasing the number of elements in the hash table much above m means that we can no longer treat the average case performance of hash table functions as $O(1)$. Consider the simplest case where the array is of size 1. In this case, the hash table is almost exactly like a linked list, with $O(n)$ access time! A hash table with an array of size 2 resembles *two* linked lists, with the hash function determining which of the two will be used - but this is still $O(n)$.

Second, note that if `ht_insert()` is called to insert an element with a key that is already present in the array, it will leave the previous value still present in the chain but inaccessible. This is bad, because the old value is "leaked": its memory cannot be reclaimed.

Task 1 (5 pts, `hashtables.c0`)**.** Modify `ht_insert()` to double the size of the array whenever the hash table's load factor exceeds 1 (in other words, `num_elems > size`). Note that this will require calculating a new hash for each element in the hash table, since `m` will have changed.

Task 2 (4 pts, `hashtables.c0`)**.** Modify `ht_insert()` to check whether an element with a matching key already exists, and if so, replace the old element instead of creating a new `list`.

2.2 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 word ladder search you implemented in Homework 3, but with one essential difference: sequences of moves are not explicitly stored. In Homework 3, sequences of words were stored

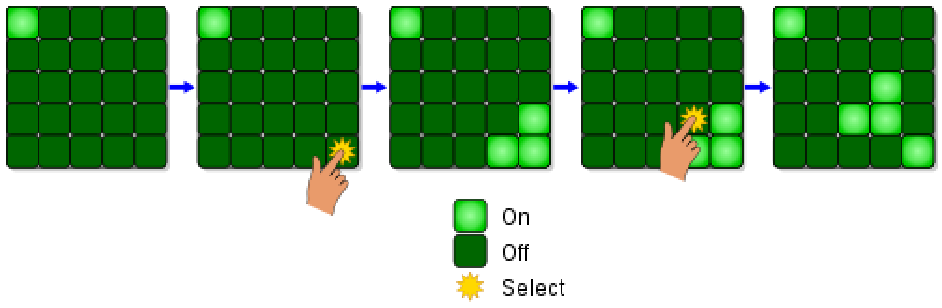


Figure 1: A sequence of moves in Lights Out. (Image: Wikipedia)

in stacks, and once the destination was found, the corresponding stack already contained all intermediate words. In this assignment, the process of finding the search-order predecessor of a game position 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 out of several components.

2.3 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 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 3 (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.

2.4 The "Map" Abstraction

In this section, you'll write the necessary interface code to use a hash table to store the pieces of game state that we'll be tracking.

First, recall the pattern from the lecture on interfaces, 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 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 struct wcount* elem;
typedef string key;

struct wcount {
    string word;
    int count;
};

int hash(string s, int m) {
    return hash_string(s, m); /* from hash-string.c0 */
}

bool key_equal(string s1, string s2) {
    return string_equal(s1, s2);
}

string elem_key(struct wcount* wc) {
    return wc->word;
}
```

Recall that we used a pseudorandom number generator, not reproduced here but shown in the lecture notes, 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.

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 4 (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`, and `hash`. For a reminder of the types and contracts of the functions, see `hashtables.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.

Although for this assignment we will only be using hash tables, we can still design the code such that another data structure could be used instead easily. The interface to our hash table is best described as a *map* from keys to values, and as we will find later in the course, hash tables are not the only data structure suitable for implementing maps. The hash table interface is much like a more general `map` library. In future assignments, we will see how multiple different implementations can ascribe to the same interface.

2.5 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 for a word ladder in Homework 3, and your code will consequently look quite similar. Instead of searching for a sequence of words, though, you are searching for a sequence of moves (x, y) that lead from an initial state to the all-"off" state. Instead of considering the next possible moves to be the set of words one letter different from a given word, you will compute every possible move you might make. And instead of maintaining a stack of words, 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.

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.

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

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. Your `solve` function should return `NULL` if given a board that has no solution, and you may assume that the given board is not already solved.

2.6 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
```

We have provided a `read_board` function in `board.c0` that reads a board from a file in the format described above.

2.7 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	`ht_insert()` resizing	`hashtables.c0`
Task 2	`ht_insert()` duplicate handling	`hashtables.c0`
Task 3	Bitarray ops: `bitarray_*`	`board.c0`
Task 4	“Client code” for maps	`board.c0`
Task 5	Lights Out solver: `solve`	`lightsout.c0`

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