15-122: Principles of Imperative Computation, Fall 2010
Assignment 4: Virtual Machines, Hash Tables, and Mazes

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

Out: Tuesday, October 5, 2010
Due: Tuesday, October 12, 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 working with hash tables and the JVM. 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 The JVM bytecode interpreter

Refer to the Java Virtual Machine bytecode interpreter given on the course website for Lecture 9.

Exercise 1 (6 pts)**.** Function calls are supported in the JVM with the instructions `invokestatic` and `ireturn`. You will write simplified interpreters for these instructions to go with the bytecode interpreter discussed in class. You may assume that all functions require exactly two arguments, stored in $V[1]$ and $V[2]$ while executing the body of the function.

The format of the `invokestatic` instruction for this problem is `0xB8 <address>`, where `<address>` is one byte representing the absolute address of the first instruction of the called function. Before the `invokestatic` instruction is executed, the arguments for the function to call must be pushed onto the stack. To implement the function call:

1. Pop the arguments off the stack.
2. Push each element of the V array on the stack to save their current values.

3. Push the return address on the stack. (This is the one byte absolute address of the next instruction after the `invokestatic` instruction.)
4. Store the arguments that were popped from the stack into $V[1]$ and $V[2]$.
5. Store the address of the first instruction of the called function in the program counter.

The format of the `ireturn` instruction for this problem is `0xB1`. To implement the return from a function:

1. Pop the return value from the stack. If the stack is empty, `return` this value as the final result from the interpreter (causing the `exec` function to exit).
2. Pop the return address from the stack.
3. Pop the previously saved values for the V array from the stack and store them back into the V array.
4. Set the program counter to the return address popped from the stack.
5. Push the return value back on the stack.

Complete the code for the `invokestatic` and `ireturn` instructions given below.

```
...
else if (inst == 0xB8)  // invokestatic <addr>
   {
       // complete this code
   }

else if (inst == 0xB1) // ireturn
   {
       // complete this code
   }
...
```

Note that the code for `ireturn` will replace the code given in the current version of the interpreter.

Exercise 2 (4 pts)**.** When we discussed merge sort in class, we saw an example of a *recursive* function. A recursive function is a function that calls itself. Here is a simple recursive function that computes $n!$ (n factorial):

```
int factorial(int n) {
    if (n == 0) return 1;
    else return n * factorial(n-1);
}
```

To compute `factorial(4)`, the function executes as follows:

```
factorial(4)
= 4 * factorial(3)
= 4 * (3 * factorial(2))
= 4 * (3 * (2 * factorial(1)))
= 4 * (3 * (2 * (1 * factorial(0))))
= 4 * (3 * (2 * (1 * 1)))
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * 6
= 24
```

Assume that $V[1]$ contains the initial value of n for the first time this function is called and $V[2]$ is 0. ($V[2]$ is not used in this case since there is only one argument needed but the interpreter still requires two arguments.) Convert this function into bytecodes given the states of the stack shown below. The first instruction is given for you.

```
           Bytecode(s):    Instruction:    State of the stack:
       00  0x15 0x01       iload 1         // n
       02  ____________     ____________    // n, 0
           ____________     ____________    // .    (if n == 0, jump to basecase)
           ____________     ____________    // n
           ____________     ____________    // n, n
           ____________     ____________    // n, n, 1
           ____________     ____________    // n, n-1
           ____________     ____________    // n, n-1, 0
           ____________     ____________    // n, factorial(n-1)
           ____________     ____________    // n*factorial(n-1)
           ____________     ____________    // .     (return result)
basecase:  ____________     ____________    // 1
           ____________     ____________    // .     (return result)
```

Note: If you do the two problems above correctly, you should be able to run this bytecode program using the modified version of the bytecode interpreter!

1.2 Hash Tables

Exercise 3 (6 pts)**.** In Java, strings are hashed using the following function:

$$(s[0] * 31^{n-1} + s[1] * 31^{n-2} + ... + s[n-2] * 31^1 + s[n-1] * 31^0) \,\%\, m$$

where $s[i]$ is the ASCII code for the ith character of string s, n is the length of the string, and m is the number of chains in the hash table.

(a) If 15122 strings were stored in a hash table with 4401 chains, what would the load factor of the table be? If the strings above were equally distributed in the hash table, what does the load factor tell you?

(b) Using the hash function above with a table size of 4401, give an example of two strings that would "collide" in the hash table and would be stored in the same chain. Show your work. (Think of short strings please!)

(c) Why did the Java designers use 31 in the formula above? Investigate several sources for your answer and report on the validity of the sources. (For this question, you are free to use the Internet, but be sure to cite your sources!)

Exercise 4 (4 pts)**.** Refer to the hash table code posted on the course website for Lecture 11. The hash table stores elements of type `elem` consisting of a `word` and an associated `count`. The elements are hashed into the table using the word as the key.

Consider extending the implementation with the following `table_delete` function which deletes an entry for a given key from the table.

```
elem table_delete(table H, key k)
//@requires is_table(H);
//@ensures is_table(H);
//@ensures table_search(H, k) == NULL;
//@ensures \result == NULL || equal(elem_key(\result), k);
{
    int h = hash(k, H->size);
    chain C = H->array[h];
    if (C == NULL) {
        return NULL;
    } else {
        elem e = chain_delete(C, k);
        if (e == NULL) return NULL;
        H->num_elems--;
        return e;
    }
}
```

Write the code for the function `chain_delete` which is used by the `table_delete` function. If the key is in the chain, remove the element containing the key from the chain and return this element. Otherwise, return `NULL`.

```
elem chain_delete(chain C, key k)
//@requires is_chain(C);
//@ensures is_chain(C);
{
      // complete this function
}
```

Give both the worst-case and the average-case run-time complexity of the function using big-*O* notation assuming there are n elements and m chains in the hash table. For the average-case analysis, assume that the hash function is ideal, i.e., that it distributes keys equally across all m chains.

You should make good use of `//@`-annotations to help guide you to the correct code: If you have a loop, what is its invariant? Which pointers might be `NULL` and which are known not to be? In either case, what additional information do we also know?

2 Programming: Mazes (30 points)

For the programming portion of this week's homework, you'll write three C_0 files to search through mazes using stacks and queues, called `mazesearch1.c0`, `mazesearch2.c0`, and `mazesearch3.c0`.

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 `hw4-starter.zip` from the course website. It contains some code for reading and working with mazes, described below, as well as some sample input mazes.

Compiling and running. For this homework, you will use the standard `cc0` command with command line options. For this assignment, you will need to use libraries for file I/O (`file`), console I/O (`conio`), parsing (`parse`) and strings (`string`). Don't forget to include the `-d` switch if you'd like to enable dynamic annotation checking.

For details on how we will compile your code, see the file `COMPILING.txt` included in the starter code. **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 hw4 <file1>.c0 ... <fileN>.c0
```

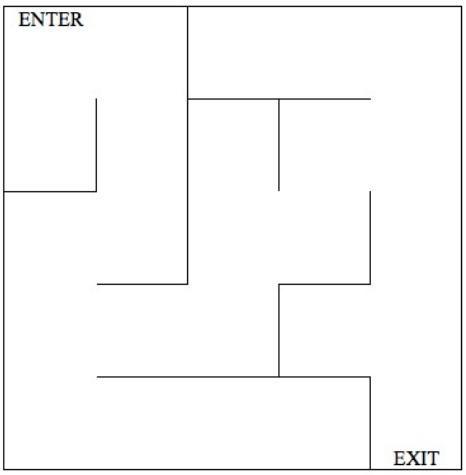
You can submit files as many times as you like and in any order. When we grade your assignment, we will consider the most recent version of each file submitted before the due date. If you get any errors while trying to submit your code, you should contact the course staff immediately.

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.**

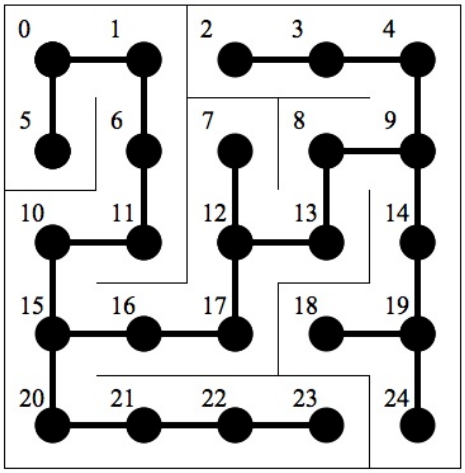
Style. Strive to write code with *good style*: indent every line of a block to the same level, use descriptive variable names, keep lines to 80 characters or fewer, document your code with comments, etc. We will read your code when we grade it, and good style is sure to earn our good graces. Feel free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

2.1 Mazes

A maze is an $m \times n$ grid of "rooms" where you always start in the top left room and want to walk to the room at the bottom right corner, without walking through a wall.



One important aspect of computational thinking is the use of *abstraction*. By finding a general representation of the data without the specific details of the maze, we can solve the problem using well-known techniques. In this case, we can represent this maze as a *graph*. A graph is a data structure consisting of a set of nodes and a set of edges that connect pairs of nodes. For a maze, the graph consists of a node for each room and a connecting edge between two nodes if the two rooms are touching and there is no wall between them.



In the example above, we label each room with a number from 0 to the number of rooms - 1, row by row, left to right. Using abstraction, finding a path in the maze

from ENTER to EXIT is the same as finding a path in the graph from node 0 to node 24. The graph could be a maze, or a sewer system, or a social network. We don't care.

In this assignment, the maze will be stored in a text file using the following format:

```
<number of rows>  <number of columns>
<number of rooms connected to room 0>  <list of rooms connected to room 0>
<number of rooms connected to room 1>  <list of rooms connected to room 1>
etc.
```

Compare the data file below with the graph (maze) on the previous page to make sure you understand its format:

```
5 5

2  1 5
2  0 6
1  3
2  2 4
2  3 9

1  0
2  1 11
1  12
2  9 13
3  4 8 14

2  11 15
2  6 10
3  7 13 17
2  8 12
2  9 19

3  10 16 20
2  15 17
2  12 16
1  19
3  14 18 24

2  15 21
2  20 22
2  21 23
1  22
1  19
```

2.2 Building the Maze

An $m \times n$ maze consists of a set of rooms where each room has a identifying number between 0 and $mn - 1$ and a list of neighboring rooms. (The other fields will be described later in the assignment.)

```
struct room {
  int number;
  list neighbors;
  /* bookkeeping info: */
  room predecessor;
  bool visited;
};
typedef struct room* room;
```

The maze is just an array of rooms along with the number of rooms and distinguished ENTER and EXIT rooms.

```
struct maze {
  int numrooms;
  room[] rooms;
  room enter;
  room exit;
};
typdef struct maze* maze;
```

You are given code in the file `mazeinput.c0` that reads in data from a text file formatted as described above and returns a `maze`.

2.3 Search the Maze using a Queue

In this programming exercise, you will implement a maze search in C_0 that uses a queue. Place your code for this section in a file called `mazesearch1.c0`.

One way to search for the exit of the maze is to use what we call a *breadth-first search*, or BFS. First, enqueue the `enter` room onto a queue. Then while the queue is not empty, dequeue a room and if the room is not the `exit` room, enqueue each of its neighboring rooms in the order they're given, but don't enqueue any rooms that you've already visited. To keep track of which rooms are visited, set the `visited` field of a room to true when it's put on the queue.

Task 1 (10 pts)**.** Write a function `mazesearch1` that searches for a path in a maze from its `enter` room to its `exit` room using a queue. The function should return a list of the rooms along the path from `enter` to `exit`.

```
list mazesearch1(maze mz);
```

In order to determine the path, as you put a room on the queue, store the room you're coming from in its `predecessor` field. For example, in the maze shown earlier, when you remove room 0 from the queue, put rooms 1 and 5 on the queue, each with a `predecessor` value of room 0. (Note that the `enter` room will not have a predecessor since it is the first room along the path. Its value will be `NULL`.)

Once your search reaches the `exit` room, you can then build the successful path through the maze. This path will be a linked list of rooms you get by traversing through the `predecessor` fields starting from the `exit` room until you get to the `enter` room. Your function should return this list of rooms, but remember that the list should be ordered from the `enter` room to the `exit` room.

You must include annotations for the precondition(s), postcondition(s) and loop invariant(s) for each function. You may include additional annotations for assertions as necessary. **2 points of this task will be based on the strength of your annotations.**

You may include any auxiliary functions you need in the same file, but you should not include a `main()` function in this file.

2.4 Search the Maze using a Stack

In this programming exercise, you will implement a maze search in C_0 that uses a stack. Place your code for this section in a file called `mazesearch2.c0`.

Another way to search for the exit of the maze is to use what we call a *depth-first search*, or DFS. First, push the `enter` room onto a stack. Then while the stack is not empty, pop a room off the stack and if the room is not the `exit` room, push each of its neighboring rooms onto the stack in the order they're given, but don't push any rooms that you've already visited. To keep track of which rooms are visited, set the `visited` field of a room to true when it's put on the stack.

Task 2 (10 pts)**.** Write a function `mazesearch2` that searches for a path in a maze from its `enter` room to its `exit` room using a stack. The function should return a list of the rooms along the path from `enter` to `exit`.

```
list mazesearch2(maze mz);
```

As in the previous task, you should return a list of rooms that make up a successful path from the `enter` room to the `exit` room in the maze. As you push a room onto the stack, set its `predecessor` field to the room you're coming from.

You must include annotations for the precondition(s), postcondition(s) and loop invariant(s) for each function. You may include additional annotations for assertions as necessary. **2 points of this task will be based on the strength of your annotations.**

You may include any auxiliary functions you need in the same file, but you should not include a `main()` function in this file.

2.5 Search the Maze using Recursion

In this programming exercise, you will implement a maze search in C_0 that uses recursion. Place your code for this section in a file called `mazesearch3.c0`.

Finding a path through a maze can also be expressed recursively. A path exists from the current room to the exit if either the current room *is* the exit or there is path from one of its neighboring rooms of the maze to the exit. (This definition is recursive; do you see it?)

Task 3 (8 pts)**.** Write a function `mazesearch3` matching the following prototype:

```
list mazesearch3(maze mz);
```

This function will call an auxiliary function that returns a list of a successful path in the maze from the given current room to the exit:

```
list search(maze mz, room current_room);
```

It is this auxiliary function that will be recursive. Think about how `mazesearch3` should call `search`; what is the initial "current room" when you're solving a maze?

You must include annotations for the precondition(s), postcondition(s) and loop invariant(s) for each function. You may include additional annotations for assertions as necessary. **2 points of this task will be based on the strength of your annotations.**

You may include any auxiliary functions you need in the same file, but you should not include a `main()` function in this file.

2.6 Analyzing the Searches

Task 4 (2 pts)**.** Create a text file named `README.txt` that answers the following questions:

1. Which search technique (BFS or DFS) is being used in the recursive task? Explain.
2. Which search technique (BFS or DFS) will always return the shortest path from the entrance to the exit of a maze? Explain.

2.7 Optional: Generating a Random Maze

Write a function `make_maze` that has a file name as its argument along with the number of rows and columns for the maze. The function should create a text file that has data for a valid maze given the number of rows and columns. The maze should be randomly generated (so the function doesn't generate the same maze each time it is called). Document this code carefully to explain how it works. The winner of the King's Prize is the person who creates the most elegant, well-documented maze generator.