Lab 5: The Maze Lab

Due: March 26th at 11:59pm

Overview

The goal of this lab is to implement Stack and generate a solve a maze using a depth-first-search routine, as well as implement a Queue to solve a maze using breadth-first search. **This is group lab, groups of two, but you must work with someone in your lab section.**

Deliverables

Your submission should include requisite code necessary to run your maze solver, but the following files will be the focus of grading:

- `Makefile`
- `README`
- `amazing.cpp`
- `maze.[cpp|h]`
- `cell.[cpp|h]`
- `list.h`
- `circ_array_list.inl`
- `stack.h`
- `array_stack.inl`
- `queue.h`
- `array_queue.inl`
- `iterator.[c|h]`
- `test_list.cpp`
- `test_stack.cpp`
- `test_queue.cpp`

All starter code is provided via `update35`.

Submission

You will submit using `handin35`. Be sure to place all relevant files in `cs35/labs/05`.

Generating and Solving Mazes Using Stacks and Queues

In this lab, you will implement a stack and queue, and, using these data structures, you will generate a random maze and then solve the maze. The methodology behind this process is that a stack mimics depth-first-search mechanisms while a queue mimics breadth-first-search mechanisms. In the context of a maze, you can consider depth-first-search as traversing a path through the maze until you reach a dead-end (or the exit), and then backing up until you have a new path choice, and traversing that until you reach a dead-end (or the exit), and then backing up and *etc.*. While breadth-first-search traversal of a maze will step through all path at the same time until they reaches the maze exit. Below is a sample maze solution using depth-first-search, where `@` indicates a cell on the path and `*` indicates a cell visited but not on the path. Note that the routine *backed-out* of the dead-end twice in the upper right-hand quadrant before eventually following the correct downward path towards the exit.

```
+---+---+---+---+---+
  @ | *   *   *   * |
+   +   +   +---+---+
| @ | * | *   *   * |
+   +---+---+---+   +
| @ | @   @   @   @ |
+   +   +---+---+   +
| @   @ | @   @   @ |
+---+---+   +---+---+
|           @   @   @
+---+---+---+---+---+

Path: (3,3) (2,3) (2,4) (3,4) (4,4) (1,2) (2,2)
      (3,2) (4,2) (4,3) (3,3) (2,3) (2,4) (3,4)
```

Starters Abstract This is a fairly involved lab. Although it does not require a lot of new lines of code to complete, there are many interacting parts that you should be aware of. Before we get into specifics, here is an abstract of the required work in the form of a plan of action (note that a complete description of all provided files can be found at the end of this document).

1. Implement the `ensureCapacity()` for the Circular Array List and test using `test_list`.
2. Implement a template stack in `array_stack.inl`
3. Implement maze generation in `maze.cpp`
4. Implement depth-first-search maze solving in `maze.cpp`
5. Implement a template queue and create a new test file for it
6. Implement breadth-first-search maze solving in `maze.cpp`
7. **(Challenge)** Implement path return for breadth-first-search maze solving
8. **(Extra Credit)** Implement other Maze Generation Routines

Like the previous lab, this is a group lab. Remeber: **Working together means coding together**. Meet with your partner regularly, work early and often, and work together.

Representing a Maze in C++

In this lab, we represent a maze using two classes: a *Maze* and a *Cell*. A Maze is just a collection of Cells in a 2-d array. Each cell has at most four neighbors, and there is either a wall between neighbor cells or not. The process of generating a maze is accomplished by removing walls between cells such that there exists a single path to the exit with other paths that lead to dead-ends. The process of solving a maze is involves continually selecting non-walled-off neighbor cells based of the current position until the exit is reached.

A maze is considered to have a *width* and a *height*. Indexes of Cells in the Maze is in the standard CS style, where (0,0) is the upper left-hand corner, and (width-1, height-1) as the lower right-hand corner. Additional, (0,0) is always described as the start of the maze, and (width-1,height-1) as the end (or exit) of the maze. More generally, an index into the 2-D array is as follows, where w is the width and h is the height:

$$\begin{bmatrix} (0,0) & (0,1) & \dots & (0,w-1) \\ (1,0) & \ddots & & \\ \vdots & & & \\ (h-1,0) & (h-1,1) & \dots & (h-1,w-1) \end{bmatrix}$$

You should review `maze.h` and `cell.h` carefully. There is a number of helpful debug functions and implementation details that will greatly aid your development.

Generating a Maze with a Stack

Once you've implemented a stack (details of this are omitted on purpose), you are now ready to start to implement maze generation. You will fill in the function `build()` in `maze.h` with the generation routine. A plain English pseduo-code description of this process is below. It is your task to implement this in C++ using the Maze and Cell classes.

```
Mark maze start as visited;
Push start onto the stack;
while stack is not empty do
    Pop current cell off stack;
    if current cell has unvisited neighbors then
        Choose a random unvisited neighbor;
        Remove wall with neighbor;
        Mark neighbor as visited;
        Push current cell back on stack;
        Push neighbor on stack;
    end
end
```

Note the depth-first-nature of this algorithm: on each iteration, the next current cell (on top of the stack) is the previously found neighbor. The algorithm continues to proceed to random neighbors, removing walls, until all neighbors are visited. Then the routine moves backward by popping from the stack until it finds a cell with more neighbors from which to create paths. This process continues until all cells in the maze are visited, which means the stack will be empty.

To select a random neighbor, you will use the `rand()` function, included from `random.h`. The `rand()` function returns a random double in the range of 0 to $2^{64} - 1$, so you will need to use modulo to bound the randomness between 0 and 4, the number of neighbors.

Solving a Maze

To solve the maze, you will do so using two different methods, depth-first-search and breadth-first-search, using a stack and a queue, respectively. Below, the depth-first-search routine is discussed, but you will need to develop your own method for solving the maze using breadth-first-search.

Depth-First-Search You will first solve the maze using a depth-first-search solution using a stack to maintain the current path being explored. Consider the pseudo-code below:

```
Set maze start as the current cell;
Push current cell on the stack;
while The current cell is not the maze end do
    Set the current cell by peaking on the stack;
    Mark the current cell visited and on the path;
    if the current cell has unvisited and reachable
    neighbors then
        Consistently choose an unvisited a neighbor;
        Push the neighbor onto the stack;
    else
        Set the current cell not on the path;
        Pop the current cell from the stack;
    end
end
return The path that reached the exit
```

Note the depth-first-nature of this routine: the next current cell is again the previously found un-visted neighbor because it was the last cell pushed onto the stack. This process of continually exploring the selected neighbor will proceed until a dead-end is reached, which results in a backing up (via pops) until there is a cell with un-visited neighbors that can be explored, which starts the depth exploration again. This exploration style is often referred to as *backtracking* since the routine greedily explores down a path until it can no longer, either by reaching a dead-end or the maze exit, and if it does reach a dead-end, it will *back-up* until it can explore another path.

In order for depth-first-search to proceed consistently, it is important that you consistently explore the neighbors. That is, for each current cell, you should check each neighbor in a consistent order to see if it is visited and, thus, should be pushed onto the stack and explored next. In `maze.h`, you will note that there is member function called `getNeighbors()` which returns a *new* list of neighbors in the same order with respect to direction. You should use that function to retrieve and consistently choose a neighbor to explore next, but don't forget to delete the returned list otherwise you will have a memory leak.

Breadth-First-Search The breadth-first-search maze solving routine will be very similar to the depth-first-search routine; however, instead of exploring down a single path in the maze and than backtracking, you explore all paths simultaneously. For lack of a better analogy, think of this like a radio-active sludge oozing through the maze. When the sludge reaches a fork in the maze, it oozes out across all paths with the same consistency, until at some point the whole maze is filled with radio-active sludge. Yuck.

The *breadth* part of the breadth-first-search is that you explore out all paths in a broad sweep at the same time, just like the sludge oozing broadly at each fork in the maze. It turns out that a queue is the right data structure to represent this process because it prioritizes the earlier-entered items rather then more

newly-entered items. **Reasoning about this and using a queue to accomplish this task is a big part of this lab**.

One side effect of breadth-first-search is that determining the actual path through the maze is more complicated since multiple paths are being explored at the same time. For breadth-first-search, **you are not required to return the path, but you are required to return the path for depth-first-search**. One of the challenge exercises is developing a routine for returning the correct path —it is non-trivial but is a great learning exercise. I highly recommend you at least attempt the challenge once you complete the other parts of the lab.

Debugging Your Maze Generating/Solving Algorithms

To help your debugging process, a number of debug functions are included. These functions return useful string representations of the associated objects that will allow you to quickly debug common errors. Any member function that begins with "debug" is something that you should pay attention to and use. I also recommend that you add your own debug functions as you need them. It is perfectly fine to submit these as long as they do not interfere with the lab execution. The functions below are particularly useful:

- Cell : `debugString()` : Convert the Cell into a debug string that is useful for tracking changes to the maze
- Cell : `strIndex()` : Returns a human readable index for this cell as a string, e.g., "(0,0)".
- Maze : `toString()` : Converts the Maze into a string representations based on the current state of the cells.
- Circular Array List : `debugPrint()` : Prints information about the current head of list and current location of data within the array.

Further, one strategy that I recommend all groups use is that while developing both the generating and solving routine, stick a maze printing function in each iteration of the loop, that is the `toString()` member function. This will allow you to visually track the progress of your routines, and debug them more easily. A visited (but not on-path) cell is represented by a $*$ and a cell that is on-path is represented by a @. Don't forget to remove these debug prints before submission.

Grading

Rubric

This lab will be graded on a 40 point scale:

- (5 Points) Completion of Circular Array List
- (5 Points) Stack and Queue Implementation and Tests
- (10 Points) Generating a Solving a Maze using a stack
- (10 Points) Solving a Maze using a Queue
- (10 Points) Coding Style and Memory Management

Note that there is points dedicated to aspects of your lab that are not part of the solution to the maze. Do pay attention to coding style and managing memory, design and lay out your code effectively so that it is readable and easily debugged. Run your program through `valgrind` and plug any memory leaks.

Challenge and Extra Credit

- **(Challenge)** Develop a routine for returning the path in breadth-first-search
- **(2.5 Points EC)** Implements Kruskal's Algorithm for maze generatin and add a flag to `amazing` to use it instead of the standard one. See `http://en.wikipedia.org/wiki/Maze_generation_algorithm` for a description.
- **(2.5 Points EC)** Implement the recursive division method for maze gneration and add a flag to `amazing` to use it instead of the standard one. See `http://en.wikipedia.org/wiki/Maze_generation_algorithm` for a definition.

Provided Code

Below is the provided source and header files and short description of their purpose. Note that files you need to create yourself are not included.

- `amazing.cpp` : The main program. You should read this file but will probably not need to edit it.
- `maze.[cpp|h]` : The source and header file for the maze. You will need to edit this file, particularly the source file.
- `cell.[cpp|h]` : The source and header file for maze cells. You will not need to edit this file to complete the lab, but you should read it.
- `list.h` : The header file for lists. The class definition for the pure-virtual List and Circular Array Lists are in this file. You will not need to edit this file, but you should read it.
- `circ_array_list.inl` : The inline-source file for the circular array lists. You will need to edit this file to complete `ensureCapacity()`.
- `stack.h` : The header file for a stack. You will need to edit this file.
- `array_stack.inl` : The inline-source file for a an array based stack implementation. You will need to edit this file.
- `test_list.cpp` : The test file for the Circular Array List implementation. You will not need to edit this file, but you should read and use it to check your implementation.
- `test_stack.cpp` : The test file for a your array stack implementation. You will not need to edi this file, but you should read and use it to check your implementation.
- `Makefile` : You will need to edit the Makefile as you develop your test.

Compilation

Compilation will occur using `make`. To compile the amazing

```
bash> make
```

If you add extra functionality contained in other files, you should edit the `Makefile`, but it is not necessary. The result of that compilation will be executable, `player`.

To compile the test code:

```
bash> make test
```

Which will compile an executable `test_list` at first, you should add more tests as you develop.

Execution

Here are the command line arguments for `amazing`:

```
[aviv@guineapig] code >./amazing -?
amazing [OPTIONS]
         -w width        set the width of the maze (dflt: 10)
         -h height       set the height of the maze (dflt: 10)
         -r rseed        set the seed of the random number generator (dflt: 100)
         -s              solve the maze default (dflt: false)
         -b              solve the maze using bread-first-search
                         instead of depth-first-search (dflt: false)
```

Note that the "-?" prints the help screen.