CPSC 221, 2015W2
Programming Project 2
A MORE AMAZING Maze Solver

Final Submission Due: 9PM Friday 2016-Apr-8

Note that the 9pm deadline is subject to the same late penalty as on the written assignments: divide the number of minutes late (past 9pm) your assignment is by 5, take the floor function, and then raise 2 to the resulting power. That number out of 64 is the fraction of points you'll lose. E.g. if you're 10 minutes late, you lose 6.25%, but if you're 25 minutes late, you lose 50%.

Objectives

- Work through the details of AVL tree and hash table implementations.
- See how asymptotically better algorithms and efficient coding let you solve problems that were previously unsolvable.

Overview

For this project, we revisit the Amazing Maze Solver that you worked on in Project 1. In particular, now that you've learned some new, more powerful data structures, we can go back and solve problems we couldn't solve before!

In Project 1, we focused on different implementations for the data structure that holds the "active states", i.e., the states from which we intend to continue searching for a solution. As you discovered, if you use a stack, you get depth-first search; if you use a queue, you get breadth-first search; and if you use a priority queue, you get best-first search.

The puzzle solver also needed a dictionary (the `seenStates` dictionary) to keep track of whether the solver had seen a state already. If so, the solver skips putting that state into the activeStates; this prevents the solver from getting stuck in an infinite loop, because we'll never try to explore the same state more than once. In addition, the seenStates dictionary is used to keep track of how we got to each state: each state keeps track of its predecessor, so that when the solver eventually finds a solution, it can follow the trail of predecessors back to the start state, thereby spelling out a step-by-step solution to the puzzle. However, back when we did Project 1, we hadn't learned anything about the Dictionary ADT yet, so we implemented this with just a simple, unordered linked list.

Now that you've learned more, we can now make the maze solver *better!* In particular, you have learned about AVL trees and hash tables as efficient data structure for the Dictionary ADT. Therefore, for this project, your job will be to write 3 new dictionary implementations (AVL Tree, Hash Table with Linear Probing, Hash Table with Double Hashing), and compare their performance to the LinkedListDict.

By the way, be sure to use the code provided with this project, not Project 1! If you don't do this, your code might be much slower than necessary.

Assignment

1. Get the supplied project files (see below), similarly to how you did it for Project 1. Be sure to use the newer files (the ones supplied in Project 2, not the ones from Project 1).
2. Fill in the missing code in AVLDict.hpp and AVLDict.cpp to implement the AVL Tree class. You can search for the string "TODO" in the source files to find places

you need to add code, but you might have to add code elsewhere to get things to work. Your implementation must be from scratch, not using any C++ STL libraries. You might want to modify code from your labs, though, and that's allowed. You must use the rotate_left and rotate_right functions we have named (but that you'll complete), and you must not modify any of the tracing code that's labeled "DO NOT CHANGE". When you implement double rotations, you must perform them using our single rotation functions (again, similarly to how it was done in your lab).

You should also notice that find() should keep track of how deep it goes each time it is called, and record that number by calling `record_stats`.

3. Fill in the missing code in LinearHashDict.hpp and LinearHashDict.cpp to implement a hash table with linear probing. Your implementation must be from scratch, not using any C++ STL libraries (in particular, you may not use STL hashtable functions, nor the STL vector class). Do not modify any of the tracing code that's labeled "DO NOT CHANGE". Also, do not modify the hash function, nor the list of primes.

   You will also need to add code to find() to count how many probes are needed, and to track the statistics by calling `record_stats`.

   At the beginning of your add() function, note the check if the add will push the load factor over 3/4; if so, add() will call rehash() to rehash into the next bigger sized hash table before performing the insertion. Do not modify this code.

4. Fill in the missing code in DoubleHashDict.hpp and DoubleHashDict.cpp to implement a hash table with double hashing. You'll find that this is very similar to LinearHashDict, so do that first and make sure you have it fully debugged, and then you can re-use almost all of that code for DoubleHashDict. The same rules about the implementation apply.

You also must answer some questions in the HANDIN.txt file.

Important marking constraints: We will mark your code semi-automatically. Therefore, you must ensure that your code will compile and link correctly by simply running `make` on the ugrad Linux servers, with our supplied Makefile. Also, your code must respect the supplied interface types, with no modifications. And do not modify the tracing code marked with "DO NOT CHANGE".

You are responsible for deallocating any memory that you allocate. (The rule of thumb is: if you create something with `new`, you should destroy it with `delete` when you've finished with it. If you used `new` with brackets to make an array, use `delete []` rather than just `delete`.) You may want to use the program `valgrind` to help check for memory leaks, but valgrind is no substitute for understanding what you're doing.

Important Programming Advice: Test your code!!!!!!!! This means writing tests, just like on Project 1. Do not just rely on the solver to test your code. Your code must implement the Dictionary find and add functions of the Dictionary ADT properly. Do your best to make sure your code really works correctly, even for corner cases!

Teams

You may work on a team of at most two on this assignment, subject to the course's Academic Conduct policy, and we strongly encourage that you do so. We recommend that you work (literally) together as much as possible, but you may choose how to divide the work under three conditions: (1) document each team member's effort in the `HANDIN.txt` file; (2) work together on and *both* understand your `HANDIN.txt`; (3) understand how your team member's code is structured and why it works. Remember to test your team's code as an integrated whole! Except in extreme cases, all team members will receive the same grade for the project. Check out All I Need to Know about Pair Programming I Learned in Kindergarten for advice on working in a pair.

What we've provided

As before, we've provided a whole bunch of files::

Makefile
: The command `make` will use this to build your program. You should not need to modify the makefile, even if you add new `.cpp` and `.hpp` files. DO NOT MISS the following commands: `make handin-proj2` hands in your final project 2 submission (run early and often). These commands must be executed on our undergrad linux servers, in a directory that contains the relevant HANDIN.txt, and various .cpp and .hpp files you wish to hand in.

HANDIN.txt
: Key documentation file for your final submissions. Read it and fill in the TODO items! Also, answer the questions there.

solve.cpp
: This is the main code that solves the mazes (and puzzles). You will edit this code to choose which maze or puzzle to solve, set up the initial configuration, and select which search strategy and dictionary implementation to use. You'll make a only few changes to it to select different puzzle sizes and data structures..

BagOfMazeStates.hpp, etc.
: Most files are exactly the same as in Project 1.

AVLDict.hpp, AVLDict.cpp
: Skeleton code for AVL tree. Implement it!

LinearHashDict.hpp, LinearHashDict.cpp
: Skeleton code for hash table with linear probing. Implement it!

DoubleHashDict.hpp, DoubleHashDict.cpp
: Skeleton code for hash table with double hashing. Implement it!

All provided files are included in this zip archive: project2-files.zip

As before, use `make` to compile the code. Note that you might have to put in some placeholder code in the classes that you haven't written yet, in order to allow the compilation to complete. Also, you may ignore any type conversion warnings in the code that we've supplied.

Final Submission

You are required to hand in your project2 directory with your implementations of the Dictionary ADT. Included in this directory should be at least these files:

- AVLDict.{hpp,cpp}, LinearHashDict.{hpp,cpp}, DoubleHashDict.{hpp,cpp},
- Your completed HANDIN.txt file

You should not need to hand in changes to any other files in the project.

Turn your files in electronically using `handin`, by running `make handin-proj2`.

Checking Your Stats:

If you want to check whether your statistics collection code is working correctly, here are the search cost stats (number of nodes in the AVL tree traversed, or number of hash buckets checked -- see the individual .cpp files for more details) you should see when running on the Maze 1 with the supplied HeapPriorityQueue.

For the AVLDict:

```
Depth Statistics for find():
```

```
0: 0
1: 2
2: 10
3: 10
4: 22
5: 32
6: 83
7: 112
8: 81
9: 49
10: 0
...
```

For the LinearHashDict (with the primes[] array):

```
Probe Statistics for find():
0: 0
1: 204
2: 51
3: 16
4: 18
5: 2
6: 9
7: 2
8: 3
9: 3
10: 10
11: 3
12: 2
13: 3
14: 0
15: 3
16: 2
17: 0
18: 0
19: 0
20: 2
21: 4
22: 9
23: 2
24: 7
25: 2
26: 2
27: 4
28: 4
More: 34
```

For the DoubleHashDict (with the primes[] array and hash2()):

```
Probe Statistics for find():
0: 0
1: 223
2: 86
3: 31
4: 30
5: 13
6: 10
7: 2
8: 0
9: 2
10: 0
11: 2
12: 2
13: 0
```

...

Last updated: 2015-03-22 13:13:39 cs221