Homework 4 Write-Up:

Group Members:

- Sam Graler
- Randy Hucker
- Jeff Edwards

Instructions:

- Download and unzip the submitted folder “Homework 4” (or open however you’d like)
    - This contains 4 complete visual studio projects. The one entitled “Homework 4” contains all the class .h and .cpp files, as well as the driver program. There are three unit test projects, one for each data structure used in the homework.
    - You should be able to compile and run them like any other Visual Studio Project, including the unit test files. The file that has the main function is called “HW4.cpp”

Group Member Contributions:

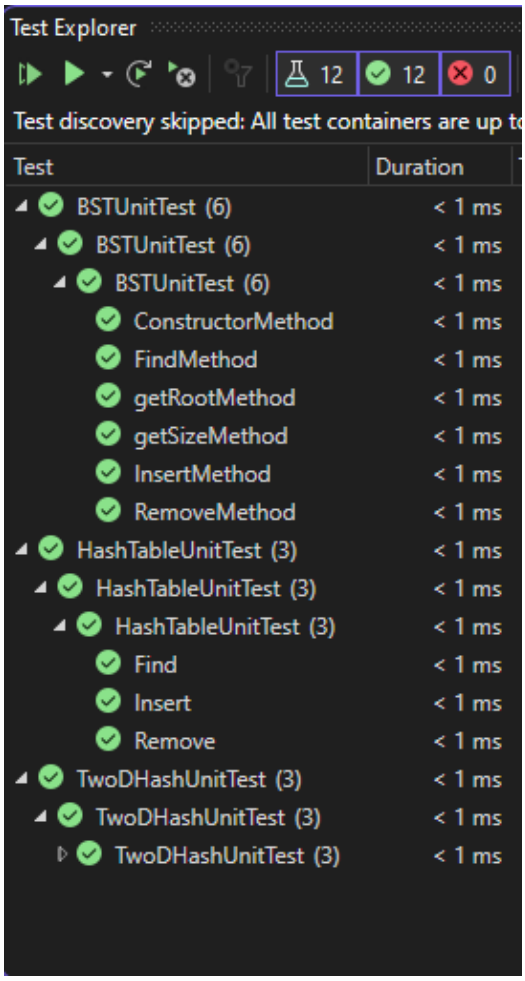
- Work for this homework was divided evenly:
    - 33.33% for Randy Hucker
    - 33.33% for Sam Graler
    - 33.33% for Jeff Edwards

Results Analysis:

Results Screenshot:

```
Initial 50 Insertion
Binary Tree Checks: 173
Hash Table Checks: 51
2D Hash Table Checks: 51

Initial 50 Removal
Binary Tree Checks: 48
Hash Table Checks: 9
2D Hash Table Checks: 10

Secondary 50 Insertion
Binary Tree Checks: 262
Hash Table Checks: 51
2D Hash Table Checks: 61

Secondary 50 Find
Binary Tree Checks: 38
Hash Table Checks: 7
2D Hash Table Checks: 11

__Total Checks__
Hash Table: 118
2D Hash Table: 133
Binary Tree: 521
```

Unit Test Screenshot:



Did it meet your expectations?

Binary Tree - Yes, this met our expectations. We assumed that hash tables (of either kind) would outperform the binary tree here because they don't need to make as many comparisons during lookups, insertions, and deletions of data as long as there aren't many collisions. This is due to hash tables using a hash function to map keys to indices in an array, which allows for constant-time access to elements. Essentially, once the hash function has been calculated, the position of the data in the hash table can be determined instantly.

Hash Table - This did meet our expectations. Since hash tables are so efficient in storing and retrieving data, we assumed that their checks would be quite low. We were initially quite worried about collisions, but since we are only inserting 50 elements into a table with a size of 500, collisions were rare. This meant the performance stayed high.

2D Hash Table - This didn't meet our expectations. We hypothesized that the 2D hash table would outperform the other data structures in this use case because the additional memory that 2D hash tables require would be minimal in our use and also, the collisions would also be reduced because of the increased space. We believe that the small difference in efficiency would be due to us needing to traverse arrays to find our insertion point instead of linear probing. Which in turn increased the number of checks versus the linear probing method we took with the normal hash table.

Which performed best?

The normal Hash Table was our best performer. We believe this to be the case because:

- 2D hash tables require more memory than normal hash tables.
- The 2D hash function needs to take into account multiple keys/indices to generate a unique index for the data.
- Binary trees require logarithmic time for lookup, insertion, and deletion operations.

What situations make each ideal and less ideal?

Binary Tree:

Ideal - Large data sets where a hash function would cause many collisions and therefore become less efficient. Also trees are more memory efficient.
Not Ideal - A large number of lookup, insertion, and deletion operations need to be carried out. Also when we have small data sets that wouldn't create a lot of collisions

Hash Table:

Ideal - If fast lookups, insertions, and deletions are a priority and the data does not need to be sorted. If the table is significantly larger than the amount of data we want to store (fewer collisions)
Not Ideal - When you need to traverse in order or if your memory is limited.

2D Hash Table:

Ideal - When multiple keys or indices need to be used to access the data. (Ex. to store game objects based on their position in the grid, with the x and y coordinates serving as the keys.)
Not Ideal - When memory usage needs to be minimal and the data set is collision prone.