Nate Johnson, Joshua Greene

Data Structures

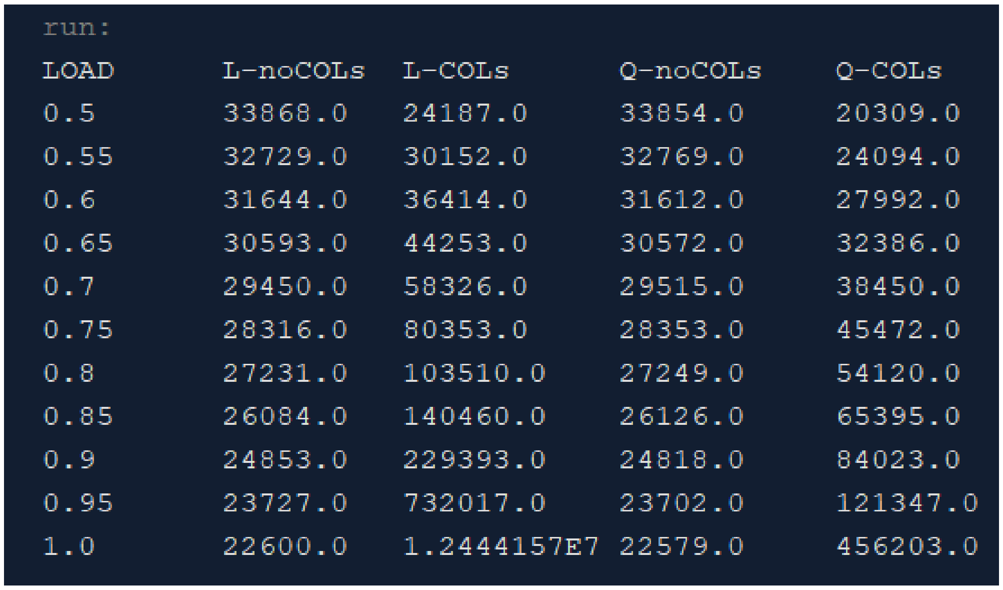
Dr. Baas

10/11/20

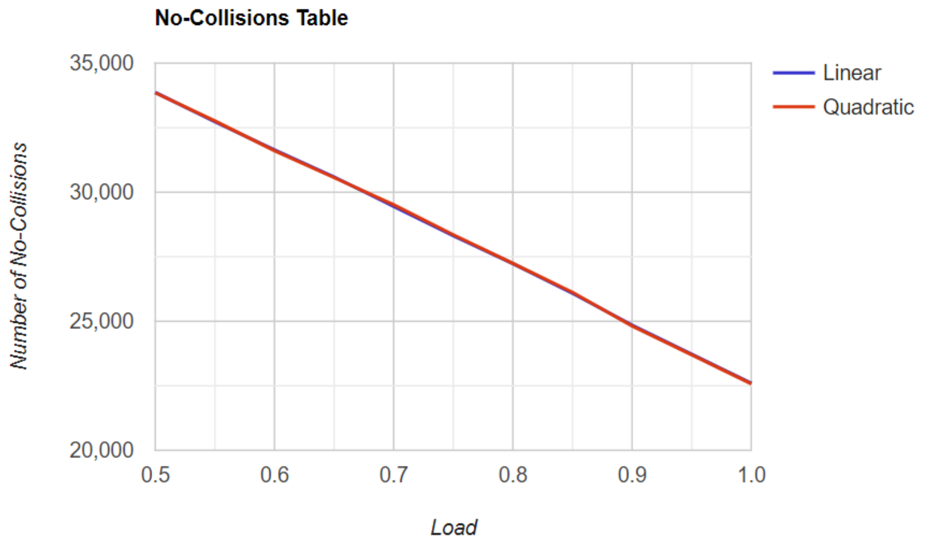
Open Addressing Lab

This lab had us implement our own open addressing hash table using an array structure. The purpose of this lab was to compare the number of collisions when performing a linear probe vs. a quadratic probe. The test was performed starting with a load factor of 50% and incrementing every 5% until our hash table was the same size as the number of values we were inserting. The data we used was provided by a .txt file called "dictionary.txt" which consisted of 45,403 words and was read into the program. Below is the output of the code showing the number of no-collisions and collisions for both the linear and quadratic probing methods.

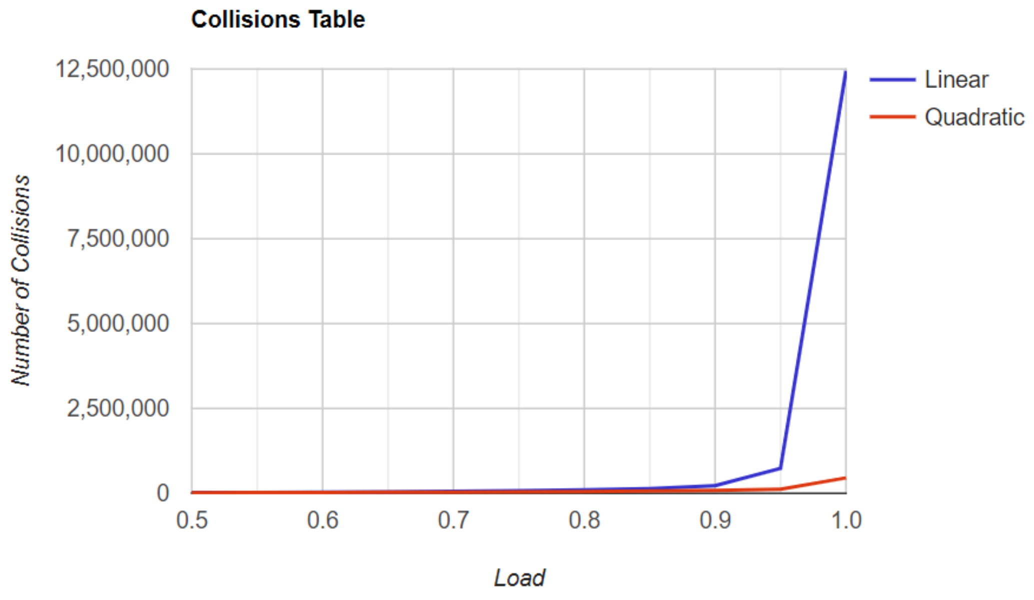
Output:



```
run:
LOAD        L-noCOLs   L-COLs      Q-noCOLs    Q-COLs
0.5         33868.0    24187.0     33854.0     20309.0
0.55        32729.0    30152.0     32769.0     24094.0
0.6         31644.0    36414.0     31612.0     27992.0
0.65        30593.0    44253.0     30572.0     32386.0
0.7         29450.0    58326.0     29515.0     38450.0
0.75        28316.0    80353.0     28353.0     45472.0
0.8         27231.0    103510.0    27249.0     54120.0
0.85        26084.0    140460.0    26126.0     65395.0
0.9         24853.0    229393.0    24818.0     84023.0
0.95        23727.0    732017.0    23702.0     121347.0
1.0         22600.0    1.2444157E7 22579.0     456203.0
```



When looking at the graphs there are a couple things to draw your attention to. First, when looking at the graph displaying the “number of no-collisions” (successful insertions) we see that both the linear and the quadratic probing methods have a linear decrease in no-collision insertions as the load factor increases. This is to be expected since as the load factor increases, there is less empty space for the next word to be placed in.



When taking a look at the collisions table, you can see both the linear and the quadratic probing methods produce relatively the same number of collisions up until the hash table is around 75% full. From here, the linear method slowly increases in collision count over the quadratic method. When the hash table is 90% full, this is where the linear and quadratic methods start to really differ in collision count. The linear probe method skyrockets right at the 95% full mark reaching over 12,000,000 million collisions, while the quadratic doesn't even reach 500,000 collisions! From this lab, we can conclude that linear and quadratic probing both produce similar results when the table size is around 75% full. However, around the 90% full mark, quadratic probing produces a far more reasonable performance. So, in conclusion, if you can allow for 25% of your hash table to be empty simply go with linear probing. Otherwise, if you can't afford to waste 25% your hash table, you better use quadratic probing!

*Below is the Source Code used in this lab:

HASH TABLE CLASS:

```
/*
 * HashTable.java
 * Nate Johnson & Joshuah Greene
 * COSC2203-01 Data Structures
 * 10-12-2020
 * This program will enter a listing of words from "dictionary.txt" into a
 * hash table using either linear or quadratic probing algorithms.
 * This program will keep a running count of the number of both the collisions
 * that occur during insertion and the number of insertions that did not result
 * in a collision.
 */
// HashTable.java
```

```
package hashtable;
import java.util.*;
import java.lang.*;
import java.io.*;

public class HashTable
{
    int tableSize;
    int numElms;
    boolean linear;
    int noCollisions = 0;
    int collisions = 0;
    String[] table;


    /*
     * Hash Table(int, int, boolean)
     * int NumElms: The quantity of elements to be iserted.
     * int load: The load value for the table.
     * boolean linear: Set to true for linear, false for quadratic methods.
     * Returns: Constructor; No return type.
     * Description: Initializes the size of the table and determine which
algorithms
     * should be used (linear/quadratic).
    */
    public HashTable(int numElms, float load, boolean linear)
    {
        this.numElms = numElms;
        this.tableSize = nextPrime((int)Math.ceil(((numElms * 1 / load ))));
        this.table = new String[tableSize];
        this.linear = linear;

        //System.out.println((linear ? "Linear " : "Quadratic ") + "Table Size:"
+ tableSize);
    }

    /*
     * nextPrime(int)
     * int num: The number from which the next prime is to be calculated.
     * Returns: The next prime number (int).
     * Description: This method tests whether the num is prime, returns it if
     * it is, and repeats the process with num incrememnted if not.
    */
    static int nextPrime(int num)
    {
        boolean prime = false;
        int check = num;
        //Run until a prime is found
        while (!prime)
        {
             //Determine if the number is already prime
            for (int i = 1; i <= Math.ceil(check / 2); i++)
            {
                //Break loop if the check number is not prime
                if (check % i == 0)
                    if (i != 1)
                    {
                        check++;
                        break;
                    }
                //If by the last call the if statement has not been broken, set
prime true
                if (i == Math.ceil(check / 2))
                    prime = true;
            }
        }
        //Return the prime
        return check;
    }

    /*
     * hash(String)
     * String key: the String from which to derive the hashCode.
     * Returns: the hashCode as an int.
     * Description: This method performs a hashing algorithm to create an
indexable
     * hash code from a given String.
    */
    public int hash(String key)
    {
        return Math.abs(key.hashCode()) % tableSize;
    }
    /*
     * load(String)
     * String filePath: The path to the file containing entries.
     * Returns: Void; no return type.
     * Description: This method reads entries from a file and inserts them into
     * the table. */
```

```
public void load(String file_path)
{
    try
    {
    //Open a new file object from parameter file path.
    File file = new File(file_path);
    //Create a buffer reader from the file.
    BufferedReader in = new BufferedReader(new FileReader(file));
    //Declare a String for file contents to be stored in.
    String fileData;
    //Read through file contents and hash each word.
    while((fileData = in.readLine()) != null)
        //If no index can be found, notify the user and quit loading.
        if (insert(fileData) != true)
        {
            System.out.println("Loading failed at: " + fileData);
            break;
        }
    }
    catch (IOException e)
    {
        System.out.println("Failed to open: " + file_path);
        System.out.println(e);
    }
}

/*
 * insert(key)
 * String key: The key String to be inserted for.
 * Returns: True if insertion successful, false if not.
 * Description: This method passes the key to the appropriate insert method.
*/
public boolean insert(String key)
{
    if (linear)
        return lProbeInsert(key);
    else
        return qProbeInsert(key);
}
/*
 * collisionCount()
 * No parameters.
 * Returns: The int number of collisions.
 * Description: This method returns the collision count.
*/
```

```
    public int collisionCount()
    {
        return collisions;
    }

    /*
     * collisionCount()
     * No parameters.
     * Returns: The int number of collisionless insertions.
     * Description: This method returns the collisionless insertion count.
    */
    public int noCollisionCount()
    {
        return noCollisions;
    }

     /*
     * qProbeInsert(String)
     * String key: The String to be inserted into the table.
     * Returns: True if the key String was inserted; false otherwise.
     * Description: This method performs a quadratic search of the table and
inserts
     * the key String into an empty index.
    */
    boolean qProbeInsert(String key)
    {
        //Get hashcode
        int hash =  Math.abs(key.hashCode());
        int probe = hash % tableSize;

        if (table[probe] == null)
        {
            noCollisions++;
            table[probe] = key;
            return true;
        }

        for (int i = 1; i < tableSize; i++)
        {
            collisions++;
            probe = Math.abs(hash + i * i) % tableSize;
            if (table[probe] == null) {
                table[probe] = key;
                return true;
            }
```

```java
        }
        //If no open index was found, return false
        return false;
    }

    /*
     * lProbeInsert(String)
     * String key: The String to be inserted into the table.
     * Returns: True if the key String was inserted; false otherwise.
     * Description: This method performs a linear search of the table and inserts
     * the key String into an empty index.
    */
    boolean lProbeInsert(String key)
    {
        //Get hash
        int hash = Math.abs(key.hashCode());
        int probe = hash % tableSize;

        if (table[probe] == null)
        {
            table[probe] = key;
            noCollisions++;
            return true;
        }

        for (int i = 1; i < tableSize; i++)
        {
            collisions++;
            probe = (hash + i) % tableSize;

            if (table[probe] == null)
            {
                table[probe] = key;
                return true;
            }
        }

        //If no open index was found, return false
        return false;
    }
}
```

DRIVER CLASS:

```
/*
 * HashTable.java
 * Nate Johnson & Joshuah Greene
 * COSC2203-01 Data Structures
 * 10-12-2020
 * This program will load a file "dictionary.txt" into a HashTable object
 * using both linear and quadratic probing over ten iterations, with a load size
_* range of 0.5, 0.55, 0.66 ... 1.0.
 * This program will output the collisions and non-collision insertion counts in
_* columns and in CSV compatible formatting.
 */


package hashtable;

public class Driver
{

    public static void main(String[] args) {

        // report[][] stores the data for each search method in a table
        float[][] report = new float[11][5];

        // Denote each column for output
         System.out.println(String.format("%-9s %-9s %-9s %-9s %-9s",
                 "LOAD", "L-noCOLs", "L-COLs", "Q-noCOLs", "Q-COLs"));

        //Perform tests with load sizes [0.5, 0.55, 0.6 ... 1.0]
        for (int i = 0; i <= 10; i++)
        {
            //Calculate load
            float load = (float)(0.50 + (0.05 * i));
            report[i][0] = load;
            HashTable lTable = new HashTable(45403, load, true);
            HashTable qTable = new HashTable(45403, load, false);
            lTable.load("dictionary.txt");
            report[i][1] = lTable.noCollisionCount();
            report[i][2] = lTable.collisionCount();
            qTable.load("dictionary.txt");
            report[i][3] = qTable.noCollisionCount();
            report[i][4] = qTable.collisionCount();
            //Output report row
            System.out.println(String.format("%-9.4s %-9s %-9s %-9s %-9s",
                    report[i][0], (int)report[i][1], (int)report[i][2],
(int)report[i][3], (int)report[i][4]));
        }

        System.out.println("\nCSV Formatting: ");

        System.out.println(String.format("%s, %s, %s, %s, %s",
                 "LOAD", "L-noCOLs", "L-COLs", "Q-noCOLs", "Q-COLs"));
        for (int i = 0; i <= 10; i++)
        {
            System.out.println(String.format("%s, %s, %s, %s, %s",
                    report[i][0], (int)report[i][1], (int)report[i][2],
(int)report[i][3], (int)report[i][4]));
        }
    }
}
```