Hash Table Lab

The purpose of a hash function is to take a range of key values (whether numeric or alphanumeric) and transform them into index values in such a way that the key values are distributed evenly across all the indices of the hash table. Keys may be completely random or not so random.

Key to address transformation is defined as a mapping or hashing function ***H*** which maps the **key space *K*** (range of key values) to the **address space *A*** (range of available buckets). This function computes a hash by performing some simple mathematical or logical operations on the key or part of the key.

A so-called perfect hash function maps every key into a different bucket. This is only possible for keys that are unusually well behaved and whose range is small enough to be used directly as array indices. In most cases neither of these situations exist, and the hash function will need to compress a larger range of keys into a smaller range of index numbers. This ratio of ***K*** size to ***A*** size will determine the probability of collisions.

Java Hash Code method

This hash will be an integer, so it has potential to be ~2,147,483,647. Obviously, this number will be too large to use directly as a table index, particularly if there are a small number of items to store. The space wasted will be enormous. Therefore, the final stage in this hash will be to normalize the value to be less than the table size.

normalized_hash = hash % table_size;

As there is always a possibility that duplicate hashes can be created by this scheme you have to handle these collisions. What do you do when you have two objects identified by the same table index?

Java String Hash Code: The Java programming language implements the following algorithm to hash strings:

hash = s[0]*31^(n - 1) + s[1]*31^(n - 2) + ... + s[n - 1]

Using int arithmetic, where **s[i]** is the ith character of the string, n is the length of the string, and ^ indicates bitwise exclusive-or. (The hash value of the empty string is zero.) This is one example of a methodology to transform alphanumeric data to a table index. Notice the usage of prime number 31; prime numbers increase the chances of generating well distributed indices due to the lack of factors. Couple this with a table size that is also prime, and you further increase the chances of generating unique keys and hashes with equitable distribution.

Hash Goals: As stated above, the goal of a hash function is to consume a block of data and generate a table index that is evenly distributed across the buckets of the table. Hashing function need to be chose carefully. Let’s examine a couple of cases.

Poor Hash Function:

K = {"abcdef", "bcdefa", "cdefab", "defabc"}
V = {1, 2, 3, 4}
H = sum the ASCII code for each character, modulo 223
A = 7

I'm using a prime number here (223) because I heard that prime numbers help

```
int bad_hash_man(string key){ // O(N) N => length of the string
     int hash = 0;
     for (int i = 0; i < key.length(); ++i)
          hash += static_cast<int>(key[i]) % 223;
     return hash % 7;
}
```

Index	Key + Value
0	
1	
2	
3	
4	"abcdef" : 1
5	"bcdefa" : 2
6	"cdefab" : 3

K_1 = bad_hash_man("abcdef");
H = (97 + 98 + 99 + 100 + 101 + 102) % 223
H = 151
Index = 151 % 7
Index = 4

K_2 = bad_hash_man("bcdefa");
H = (98 + 99 + 100 + 101 + 102 + 97) % 223
H = 151
Index = 151 % 7
Index = 4
We have a collision.

K_3 = bad_hash_man("cdefab");
H = (99 + 100 + 101 + 102 + 97 + 98) % 223
H = 151
Index = 151 % 7
Index = 4
We have a collision.

K_4 = bad_hash_man("defabc");
H = (100 + 101 + 102 + 97 + 98 + 99) % 223
H = 151
Index = 151 % 7
Index = 4
We have a collision.

Every key in this set will map to index 4 in our table. These collisions remove the expected O(1) behavior for insertion and lookup as a supplemental iterative routine must be spawned that locates an available space. The approach used above is **linear probing,** or checking each **slot** > **index** for an opening. The behavior you see above is called clustering and will degrade hash table **efficiency** to **O(n) where n = cluster size**

Better Hash Function:

K = {"abcdef", "bcdefa", "cdefab", "defabc"}
V = {1, 2, 3, 4}
H = multiply the ascii code by its position, sum the series and modulo 223
A = 7

```
int better_hash(string source){ // O(N) N => length of string
     int hash = 0;
     for (int i = 0; i < source.length(); ++i)
          hash += static_cast<int>(source[i]) * i % 223;
     return hash % 7;
}
```

K_1 = better_hash("abcdef");
H = (97*0 + 98*1 + 99*2 + 100*3 + 101*4 + 102*5) % 223
= (0 + 98 + 198 + 300 + 404 + 510) % 223
= 1510 % 223
H = 172
Index = 172 % 7
Index = 4
K_2 = better_hash("bcdefa");
H = (98*0 +99*1 + 100*2 + 101*3 + 102*4 + 97*5) % 223
= (0 + 99 + 200 + 303 + 408 + 485) % 223
= 1495 % 223
H = 157
Index = 157 % 7
Index = 3
K_3 = better_hash("cdefab");
H = (99*0 +100*1 + 101*2 + 102*3 + 97*4 + 98*5) % 223
= (0 + 100 + 202 + 306 + 388 + 490) % 223
= 1486 % 223
H = 148
Index = 148 % 7
Index = 1
K_4 = better_hash("defabc");
H = (100*0 + 101*1 + 102*2 + 97*3 + 98*4 + 99*5) % 223
= (0 + 101 + 204 + 291 + 392 + 495) % 223
= 1483 % 223
H = 145
Index = 145 % 7
Index = 5

Index	Key + Value
0	
1	"cdefab" : 3
2	
3	"bcdefa" : 2
4	"abcdef" : 1
5	"defabc" : 4
6	

Hash Functions: These are some common hashing approaches that have demonstrated reliable key distribution. These are just examples. You should always experiment with the function before deploying due to the potential unique characteristics of each key space.

1. **Division Method:** One of the earliest hashing functions. This function performs well at preserving the uniformity that exists in a key space. Keys that are closely related or clustered are mapped to unique indices.

    a. Division method is defined as (for some positive integer *m)*
    b. **H(x) = Math.abs(x) % m**
    c. Yields a hash belonging to the set {0, 1 . . . m – 2, m – 1}
    d. ***m*** being a large prime number, aids in uniqueness due to lack of factors
    e. Even number divisors are to be avoided

2. **Midsquare Method:**

    a. Multiply the key by itself and select an appropriate number of bits or digits from the middle of the square. The same positions must be used for all products
    b. **Example:**
        i. key → 12345
        ii. key^2 → 152, 399, 025
        iii. hash → 399

3. **Digit Analysis:** This approach forms an address by selecting and shifting digits or bits from the original key

    a. **Example:**
        i. Key → 7546123
        ii. Select digits 3 to 6 and reverse the order
        iii. Hash →2164
    b. For a given key space the same positions and techniques should be used consistently
    c. Ideally some analysis is performed before hand to determine which digits have the most uniform distribution across the key space

4. **Length Dependence:** The length of the key along with some predetermined portion of the key is used to form a table address directly or often an intermediary key which is further hashed by the division method to form the table address.

    a. One string approach that has performed well sums the binary representation of the first and last characters, added to the length of the key, shifted left 4 bits. In decimal parlance this is akin to multiplying the value by 16
    b. **STUDENT** becomes
        i. int('S') + int('T') + (key.length() << 4)
        ii. 83 + 84 + (7 << 4) or 83 + 84 + (7 x 16)
        iii. This yields 279 as an intermediate key
        iv. Apply the division method with a large-ish prime divisor: 279 % 61 → 35

Open Addressing Collision Resolution Schemes:

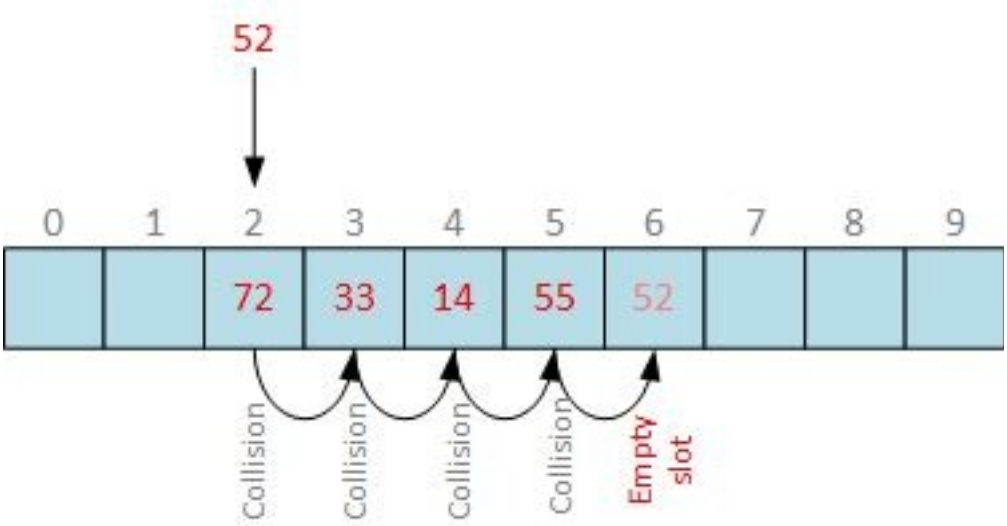
An open addressing scheme advances through the table from the collision point, searching for an open slot. These schemes have a direct effect on the deletion of an item from a hash table. Simply removing an item from a cluster can derail further lookup attempts by introducing a blank bucket. Search routines could see this and bail erroneously with the incorrect assumption of a terminated search. Due to this, open addressing schemes either need to shift clusters (inefficient) to fill opened spaces or install a special **deleted** reference.

Linear Probing: If faced with a collision situation, the linear probing algorithm will iteratively look at subsequent elements until the first free space is found. This traversal is known as probing the table; and it proceeds one element at a time. The probing begins at the original hashed index and proceeds element by element. This is an iterative approach and thus begins to degrade the performance of the table.

A downside to linear probing is that clusters can form which degrade the performance of extraction. This is called **primary clustering**. If the cluster size is large, the cost of the search increases. For linear probing it is a bad idea to let the hash table get nearly full, because performance is degraded as the hash table gets filled. A hash table attempts to guarantee O(1) performance of insertion and lookup but an overloaded hash table can generate to O(n) due to clustering

The ratio of **elements / table_size** is called the **"load factor"**. This measure is key to the resizing of a hash table. A common approach is to increase the size of the table and rehash the elements when a load factor threshold is reached.

hash_code = key % 10
Insert 52



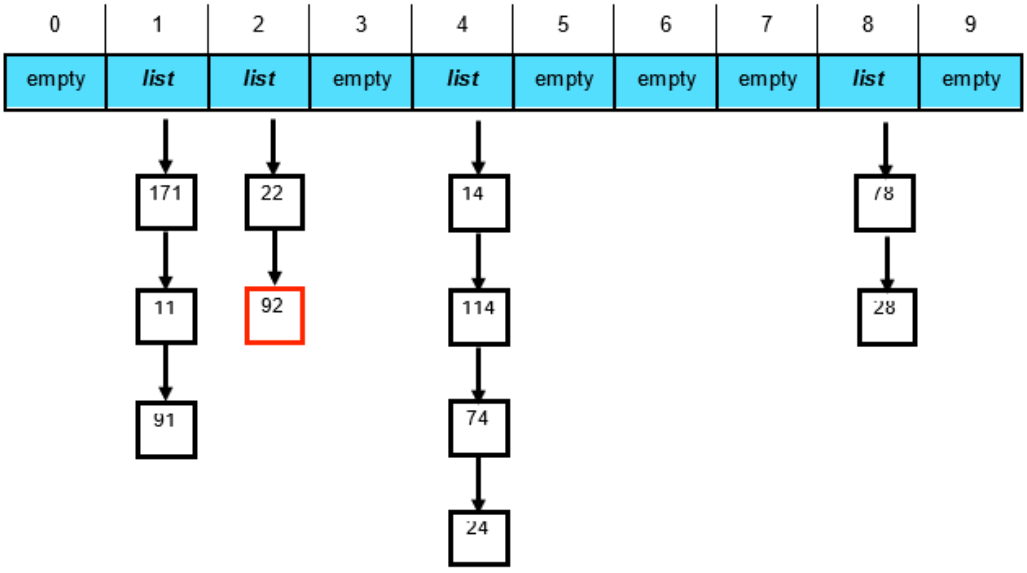
Quadratic Probing: Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found. Quadratic probing can be a more efficient algorithm, since it better avoids the primary clustering problem that can occur with linear probing, although it is not immune because keys that hash to the same index will in fact travel the same probe path and **secondary clusters** can develop. The size of the table should be a prime number so that the probing scheme is guaranteed to eventually visit every slot in the table. Can you prove that this is true?

The Probe Step Is the Square of the Step Number: In a linear probe, if the primary hash index is x, subsequent probes go **x+1, x+2, x+3, . . .**
In squared quadratic probing, probes go **x+1, x+4, x+9, x+16, x+25, . . .**
The distance between probes is the square of the step number:
$x+1^2$, $x+2^2$, $x+3^2$, $x+4^2$, $x+5^2$, . . . $x+N^2$

Closed Addressing Collision Resolution Scheme:

Separate Chaining: In open addressing, collisions are resolved by probing for an open slot in the hash table. A different approach is to install a linked list or tree at each index in the hash table. A data item's key is hashed to the index in the usual way, and the item is inserted into the linked list at that index. Other items that hash to the same index are simply added to the linked list; there's no need to search for empty cells in the primary array.

Insert 92
=======
key = 92
hashed_index = key % table_size
92 % 10 = 2
table[hashed_index].push_back(key); // table[2].push_back(92);



Tasks

1. Implement a **HASH TABLE:**

    1. You have been provided with two class skeletons for two hash table implementations
        1. An **Open Addressing** scheme that will use linear or quadratic probing for collision resolution
            1. This class will work with an ***array of HashNode*** for the memory management

        2. A **Closed Addressing** scheme that will use ***separate chaining*** for collision resolution
            1. This class will require ***your*** LinkedList (from previous labs) for the chaining that happens at each table slot. You may need to modify certain behaviors of your list or add new behaviors. You may not use C++ STL classes here. All structures must be your own.
            2. The underlying memory management of this HashTable will be an ***array of LinkedLists***

        3. You are required to stick with arrays so that you can focus on the minutia of the resize behavior. Do not use vectors

        4. The classes are each defined in their own ***hpp*** file

    2. At the heart of these classes is a **struct** that represents a ***hash node.*** The hash node will encapsulate various details of the hash tables payload. In both you will find
        1. The ***key***, which will be a string in this lab (we are not worried about fully templating both the key and value)
        2. The ***value***, which will be a templated type
        3. The hash node designs vary a bit depending on the addressing scheme and you are expected to understand and explain why. Study the implementations closely.

    3. In each class you will notice public method skeletons for the abstract map behaviors
        1. put → puts value in table with key. If the key is already in the table, simply update the value
        2. get → gets value from table with key. Does not remove
        3. remove → removes value from table with key
        4. contains → boolean method. Is key present in table?
        5. Overloaded **[]** to provide ***table[key]*** behavior. Same as **put** but without method call
        6. These are marked as **TO DO** and your task is to design implementations

    4. In each class you will notice private helper function skeletons to help with some of the work
        1. hash → reduces key to a number
        2. should_resize → boolean method to signify resizing should occur
        3. resize → resizes by 50% and then closest prime, rehashes. Clean up memory

4. load_factor → threshold for resize
5. find_next_prime → calculates the next prime number after 50% increase
6. These are marked as **TO DO** and your task is to design implementations

5. Design implementations for the four classes of hash functions mentioned in this document. You will notice that the keys for these exercises are strings, so you will need to also include some hashing that reduces the string to a number and then experiment with the numeric hashing approaches.

6. **Load Factor:** A critical statistic for a hash table is the ***load factor (λ)***, defined as
    1. Where
        1. n is the number of entries occupied in the hash table
        2. k is the number of buckets
        3. $\lambda = n / k$

    2. When the load factor gets high the performance of the hash table degrades. You will be experimenting with this in the exercises.

    3. The load factor for Java 10s HashMap is 0.75. When capacity reaches this point the table is resized and the elements are rehashed and reinserted. **Explain in comments why the values cannot simply be left where they were.**

7. **PROOF OF CONCEPT:** Create a **open_address_main.cpp** and prove that your open addressing hash tables function properly. There is some rudimentary print logic in the hash table classes. Use this to assist in debugging. Prove that your HashTable can deal with values of different types. Use the provided **Contact.h** class specification to prove that Contact instances can be added to the table

8. **PROOF OF CONCEPT:** Create a **closed_address_main.cpp** and prove that your closed addressing hash tables function properly. Clearly display using both **linear and quadratic probing.** There is some rudimentary print logic in the hash table classes. Use this to assist in debugging.

9. The most important piece of information we need to analyze the use of a hash table is the load factor, λ. Conceptually, if λ is small, then there is a lower chance of collisions, meaning that items are more likely to be in the slots where they belong. If λ is large, meaning that the table is approaching capacity, then there are more and more collisions. This means that collision resolution is more difficult, requiring more comparisons to find an empty slot. The hash table performance degrades from **O(1)** to **O(n)** where n = the collision cluster size.

    With chaining, increased collisions means an increased number of items on each chain. Understand that the underlying table does not necessarily need to resize because only pointers to lists are being stored. The list at each slot grows, which means there's the possibility that $\lambda > 1$. Obviously this is bad and the table should still be resized.

1. **Formulas:**
    1. For a successful search using open addressing with linear probing, the average number of comparisons is approximately $\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right)$
    2. An unsuccessful search gives $\frac{1}{2}\left(1+\left(\frac{1}{1-\lambda}\right)^2\right)$
    3. Chaining gives $1+\frac{\lambda}{2}$ average number of comparisons for a successful search
    4. And λ for an unsuccessful search

10. **GRAPHING: Hash Tables are another algorithm to solve the "linear list search" problem.** Revisit the linear and binary search from earlier in the semester. Include these algorithms as part of the search analysis for this lab. Design an experiment and plot linear and binary search tests in the same plot with the hash table using the same data sets. To make this easier you can approach this experiments with values of type int.

    Create an analysis document that includes the graphs clearly labeled and a detailed description of the analysis for each algorithm. Include details of exactly how the Big O notation is derived for each algorithm.

Submission: All necessary C++ files, screen shots of the various plots descriptively labeled and correct. All code must be error and leak free, and the HashTable classes needs to demonstrate that they can work with various data types.