## Hash Tables

**(a)** Implement class `ChainedHashTable`.

- You will need to modify the `ListNode` struct from previous assignments to include two data members:
  - The key which will be of type `string`
  - The value which is a count and is of type `int`
- Each hash table should have an array capacity of 5000 elements
- Implement and test `ChainedHashTable`

```
// an abstract struct to parent your various hasher classes
struct Hasher{
  virtual int hash(string s, int N) = 0;
};
// your first working hashing class
struct GeneralStringHasher: Hasher{
  int hash(string key, int N){
    unsigned hashVal = 0;
    for (int i = 0; i < key.size(); i++)
      hashVal = (127 * hashVal + key[i]) % 16908799;
    return hashVal % N;
  }
};
class ChainedHashTable{
  private:
    //data members
    //hint: have a data member that is a hasher reference
  public:
    ChainedHashTable(int capacity, Hasher& myHasher){
      //implement constructor
    }
    //copy constructor
    //destructor
    //first testing functions to call from testHashTable function.
    //Run testHashTable for errors
    ////////////////////
    //continue incrementally adding methods to class and test cases
    //to testHashTable
};
//individual method testing functions
void testConstructor(Hasher& hasher){
  //create an empty ChainedHashTable object
}
void testCopyConstructor(Hasher& hasher){
  //create a ChainedHashTable object
```

```
  //create a second object as a copy of the first object
}
//etc
//overall tester function
void testHash(char *inputFileName, Hasher& hasher){
  //call test functions
  //you may want to instantiate a ChainedHashTable object to pass as
  //a reference to some of the more advanced testing features
}
int main(){
  GeneralStringHasher h;
  testHash("random.txt", h);
}
```

- Define `operator[]` on your `ChainedHashTable`
  - If you have a `ChainedHashTable` object called `table` and a string object called `word`, then `table[word]++` should result in incrementing the counter (i.e the `value` of the element with `key` matching `word`. HINT: think carefully about what `operator[]` needs to return for this to work properly.
- For your homework please include the time complexity $O(N)$ of each of the following functions:
  - `ChainedHashTable::insert`
  - `ChainedHashTable::find`
  - `ChainedHashTable::remove`
  - `void insertAll(ChainedHashTable& h,...)`
  - `void findAll(ChainedHashTable& h,...)`
  - `void removeAll(ChainedHashTable& h,...)`
  - `int hash(string s, int N)`
  - `int& operator[](string s)`

**(b)** Testing `ChainedHashTable`

- You will be using two input files of words:
  - `random.txt`
  - `words.txt`
- Create the following three global functions (not class methods) in `testHash.cpp`
  - `insertAll`

  - `findAll`
  - `removeAll`

- Measure the time for each for the above functions as follows
  - For each input file,
    - Starting with $k = 1$, insert $k$/10th of the words $4500 * k$ into your hash table object
    - Using your global functions, measure the time to:
      - Insert all the words from the file into your table
      - Find all the words from the file in your table
      - remove all the words from the file from your table (should support removal of entire hash table entry or decrement the counter)
    - Increment $k$ and repeat until $k = 10$
  - Include two tables of the results in the following format:

| random.txt | | | | |
|---|---|---|---|---|
| | N (number of inputs) | | | |
| | 4500 | 9000 | ... | 45000 |
| insertAll T(N) | .1s | .25s | ... | 1.0s |
| findAll T(N) | .1s | .25s | ... | 1.0s |
| removeAll T(N) | .1s | .25s | ... | 1.0s |

| words.txt | | | | |
|---|---|---|---|---|
| | N (number of inputs) | | | |
| | 4500 | 9000 | ... | 45000 |
| insertAll T(N) | ... | ... | ... | ... |
| findAll T(N) | ... | ... | ... | ... |
| removeAll T(N) | ... | ... | ... | ... |

**(c)** Comparing the Performance of Three Different Hash Functions

- The three hash functions to be compared are:
  - See below. Hash which shifts the lower 6 bits from each character
  - One that sums up the ASCII codes of the string
  - One that multiplies the ASCII codes of the string together

```
int hash(string key, int N){
  const unsigned shift = 6;
  const unsigned zero = 0;
  unsigned mask = ~zero >> (32-shift);  // low 6 bits on
  unsigned result = 0;
```

```
  for (int i = 0; i < key.size(); i++)
    result = (result << shift) | (key[i] & mask);
  return result % N;
}
```

- Create three additional hasher derived structs with respect to the mentioned hash functions

```
//A rough idea of these Hashers - //
//some modifications may be needed//
struct SumHasher : Hasher {
  int hash(string s, int N) {
    int result = 0;
    for (int i=0; i<s.size(); ++i)
      result += s[i];
    return abs(result) % N;
  }
};
struct ProdHasher : Hasher {
  int hash(string s, int N) {
    int result = 1;
    for (int i=0; i<s.size(); ++i)
      result *= s[i];
    return abs(result) % N;
  }
};
```

- For each hash function
  - Insert all the words from `random.txt`, then print out the following statistics (using the functions constructed.
    - min chain length
    - max chain length
    - average chain length
    - standard deviation
  - Report the time to insert, find, and remove all the words using that particular hash function
  - See partial sample console output...

```
Hash function 1 chain length statistics:
        min = 0; max = 400; average = 10.3; st_dev = 4.5
        insertAll = 70 sec
        findAll = 100 sec
        removeAll = 85 sec

Hash function 2 chain length statistics:
        //(etc...)
```

- Please submit your
  - Source code with time complexity analysis
  - a script file with execution under valgrind with no memory leaks (you may use a reduced length input file if execution takes too long, i.e grab 4500 words from `random.txt`)
  - a text file that includes 2 tables showing the results for $N = 4500, 9000, \ldots, 45000$
  - A separate script file showing the console output for each hash function