CSCI 240 PA 7 Submission

Due Date: 4/23/25

Name(s): Benjamin Garcia

Exercise 1 -- need to submit source code and I/O
-- check if completely done ✅ ; otherwise, discuss issues below
Pseudocode below if applicable:

Source code below:

```
#include <iostream>
#include <unordered_map>
#include <string>
#include <vector>
#include <chrono>
#include <fstream>
using namespace std;

string generateStringFromKey(int n){
  string reversed = "";
  if (n == 0){
    return "0";
  }

  while (n > 0){
    int digit = n % 10;
    reversed += to_string(digit);
    n /= 10;
  }

  return reversed;
}

void found(int n, string k){
  cout << "key: " << n << endl;
  cout << "value: " << k << endl;
}

int main() {
  unordered_map<int, string> myMap;
  vector<int> vec = {13, 21, 5, 37, 15};
```

```
for(int i = 0; i < vec.size(); i++){
  myMap[vec.at(i)] = generateStringFromKey(vec.at(i));
} // generates the unordered map with the key & string

// deleting / finding

for(auto it = myMap.begin(); it != myMap.end(); ){
  if(it->first == 10 || it->first == 21){
    found(it->first, it->second);
    ++it;
  } else if (it->first == 20 || it->first == 37){
    it = myMap.erase(it);
  } else {
    ++it;
  }
}

cout << endl;

cout << "--- checking hash map again after changes ---" << endl;

cout << endl;

for(auto it = myMap.begin(); it != myMap.end(); ++it){
  found(it->first, it->second);
}

cout << endl;

cout << "--- small 1k portion ---" << endl;

ifstream file("small1k.txt");
vector<int> small1kdata;
int val;
while(file >> val){
  small1kdata.push_back(val);
}
file.close();

unordered_map<int, string> small1kMap;
```

```
  small1kMap.reserve(1000 / 0.75);

  auto start = chrono::high_resolution_clock::now();

  for (int value : small1kdata) {
    small1kMap[value] = generateStringFromKey(value);
  }

  auto end = chrono::high_resolution_clock::now();
  chrono::duration<double> elapsed = end - start;
  cout << "Time to insert 1000 entries: " << elapsed.count() << "
seconds" << endl;

  cout << endl;

  cout << "--- large 100k portion ---" << endl;

  ifstream file2("large100k.txt");
  vector<int> large100kdata;
  int val2;
  while(file2 >> val2){
    large100kdata.push_back(val2);
  }
  file2.close();

  unordered_map<int, string> large100kMap;
  large100kMap.reserve(100000 / 0.75);

  auto start2 = chrono::high_resolution_clock::now();

  for (int value : large100kdata) {
    large100kMap[value] = generateStringFromKey(value);
  }

  auto end2 = chrono::high_resolution_clock::now();
  chrono::duration<double> elapsed2 = end2 - start2;
  cout << "Time to insert 100,000 entries: " << elapsed2.count() <<
" seconds" << endl;

  return 0;
```

```
}
```

Input/output below:

```
PS C:\Users\benja\VSCODEFILES\csci-240>  & 'c:\Users\benja\.vscode\extensions\ms-vscode.cpptools-1.24.5-win32-x64
\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-zrn2wyuv.pw3' '--stdout=Microsoft-MIE
ngine-Out-o35kzopw.txa' '--stderr=Microsoft-MIEngine-Error-wiu411oq.qun' '--pid=Microsoft-MIEngine-Pid-kgiqgdfc.j
nw' '--dbgExe=C:\msys64\ucrt64\bin\gdb.exe' '--interpreter=mi'
key: 21
value: 12

--- checking hash map again after changes ---

key: 15
value: 51
key: 5
value: 5
key: 21
value: 12
key: 13
value: 31

--- small 1k portion ---
Time to insert 1000 entries: 0.0013992 seconds

--- large 100k portion ---
Time to insert 100,000 entries: 0.536983 seconds
PS C:\Users\benja\VSCODEFILES\csci-240>
```

Exercise 2 -- need to submit source code and I/O
 -- check if completely done ✅ ; otherwise, discuss issues below
Pseudocode below if applicable:

Source code below:

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <chrono>
#include <list>
#include <fstream>

using namespace std;

template <typename Key, typename Value>
class AbstractMap {
  public:
    //-------- nested Entry class ---------
    class Entry {
        friend AbstractMap;
      private:
        Key k;
        Value v;
```

```
      public:
        Entry(const Key& k = Key(), const Value& v = Value()) :
k(k), v(v) {}
        const Key& key() const { return k; }       // read-only
access
        const Value& value() const { return v; }   // read-only
access
        Value& value() { return v; }               // allow change
to value
    };  // end of Entry class

  protected:
    //---------- custom iterator representation ----------
    class abstract_iter_rep {
      public:
        virtual const Entry& entry() const = 0;
        virtual void advance() = 0;
        virtual bool equals(const abstract_iter_rep* other) const =
0;
        virtual abstract_iter_rep* clone() const = 0;
        virtual ~abstract_iter_rep() {}
    }; //------ end of abstract_iter_rep ----------

  public:
    //---------- const_iterator ----------
    class const_iterator {
        friend AbstractMap;

      private:
        abstract_iter_rep* rep{nullptr};      // a pointer to an
underlying iterator representation

      public:
        const Entry& operator*() const { return rep->entry(); }
        const Entry* operator->() const { return &rep->entry(); }
        const_iterator& operator++() { rep->advance(); return *this;
}
        const_iterator operator++(int) { const_iterator temp{*this};
rep->advance(); return temp; }
```

```
        bool operator==(const const_iterator& other) const { return
rep->equals(other.rep); }
        bool operator!=(const const_iterator& other) const { return
!rep->equals(other.rep); }

        const_iterator(abstract_iter_rep* r = nullptr) : rep{r} {}
        const_iterator(const const_iterator& other) :
rep{other.rep->clone()} {}
        ~const_iterator() { delete rep; }
        const_iterator& operator=(const const_iterator& other) {
            if (this != &other and rep != nullptr) {
                delete rep;
                rep = other.rep->clone();
            }
            return *this;
        }
    }; //------ end of const_iterator ----------

  protected:
    // necessary utilities for our inheritance
    abstract_iter_rep* get_rep(const_iterator iter) const { return
iter.rep; }
    void update_value(const Entry& e, const Value& v) {
const_cast<Entry&>(e).v = v; }

  public:
    //---------- pure virtual functions -----------
    virtual int size() const = 0;
    virtual const_iterator begin() const = 0;
    virtual const_iterator end() const = 0;
    virtual const_iterator find(const Key& k) const = 0;
    virtual const_iterator put(const Key& k, const Value& v) = 0;
    virtual const_iterator erase(const_iterator loc) = 0;

    //---------- concrete functions -----------
    bool empty() const { return size() == 0; }     // Returns true
if the map is empty, false otherwise
    bool contains(const Key& k) const { return find(k) != end(); }
// Returns true if map contains key
```

```
    // Returns the value associated with key k,  or throws
out_of_range exception if k not found
    const Value& at(const Key& k) const {
        const_iterator it{find(k)};
        if (it == end())
            throw std::out_of_range("key not found");
        return it->value();
    }

    // Erases entry with given key (if one exists); returns true if
entry was removed
    bool erase(const Key& k) {
        const_iterator it{find(k)};
        if (it == end())
            return false;
        erase(it);
        return true;
    }

    virtual ~AbstractMap() {}                    // expected when
declaring other virtual functions
};

template <typename Key, typename Value>
class UnorderedListMap : public AbstractMap<Key,Value> {
  private:
    typedef AbstractMap<Key,Value> Base;
// shorthand for templated base type
  public:
    using typename Base::Entry, typename Base::const_iterator,
Base::erase;
  private:
    using typename Base::abstract_iter_rep, Base::get_rep;
    typedef std::list<Entry> EntryList;
// shorthand for a list of entries
    typedef typename EntryList::const_iterator LCI;
// shorthand for list's const_iterator

    EntryList storage;
// map entries are stored in a list
```

```
  protected:
    // position within our map is described by an iterator in the
underlying list
    class iter_rep : public abstract_iter_rep {
// specialize abstract version
      public:
        LCI list_iter{nullptr};
// data member
        iter_rep(LCI it) : list_iter(it) {}
// constructor

        const Entry& entry() const { return *list_iter; }
        void advance() { ++list_iter; }
        abstract_iter_rep* clone() const { return new
iter_rep(list_iter); }
        bool equals(const abstract_iter_rep* other) const {
            const iter_rep* p = dynamic_cast<const
iter_rep*>(other);  // cast abstract argument
            return p != nullptr && list_iter == p->list_iter;
        }
    };  //----- end of class iter_rep -----

  public:
    UnorderedListMap() {}
// Creates an empty map

    int size() const { return storage.size(); }
// Returns number of map entries

    // Returns iterator to first entry
    const_iterator begin() const { return const_iterator(new
iter_rep(storage.begin())); }

    // Returns iterator representing the end
    const_iterator end() const { return const_iterator(new
iter_rep(storage.end())); }

    // Returns a iterator to the entry with a given key, or end() if
no such entry exists
```

```
    const_iterator find(const Key& k) const {
        LCI walk{storage.begin()};
        while (walk != storage.end() && walk->key() != k)
            ++walk;
        return const_iterator(new iter_rep(walk));
    }

    // Associates given key with given value; if key already exists
previous value is overwritten
    const_iterator put(const Key& k, const Value& v) {
        const_iterator loc{find(k)};
        if (loc != end()) {
            this->update_value(*loc,v);
// overwrite existing value
            return loc;
        } else {
// key is new
            storage.push_back(Entry(k,v));
            return const_iterator(new iter_rep(--storage.end()));
// newest entry is last on the list
        }
    }

    // Removes entry referenced by given iterator, and returns
iterator to next entry in iteration order
    const_iterator erase(const_iterator loc) {
        LCI list_iter =
dynamic_cast<iter_rep*>(Base::get_rep(loc))->list_iter;
        return const_iterator(new
iter_rep(storage.erase(list_iter)));
    }
};

template <typename Key, typename Value, typename Hash>
class AbstractHashMap : public AbstractMap<Key,Value> {
  protected:
    typedef AbstractMap<Key,Value> Base;

  public:
```

```
    using typename Base::Entry, typename Base::const_iterator,
Base::begin, Base::end;

  protected:
    Hash hash;                                          // hash
function
    int sz{0};                                          // total
number of entries
    int table_sz{17};                                   // current
number of buckets

    // compute compressed hash function on key k
    int get_hash(const Key& k) const { return hash(k) % table_sz; }

    // Change table size and rehash all entries
    void resize(int new_table_size) {
        vector<Entry> buffer;                           //
temporary copy of all entries
        for (Entry e : *this)
            buffer.push_back(e);
        table_sz = new_table_size;
        create_table();                                 // based on
updated capacity
        sz = 0;                                         // will be
recomputed while reinserting entries
        for (Entry e : buffer)
            put(e.key(), e.value());                    // reinsert
into this map
    }

    //---------- pure virtual functions -----------
    virtual void create_table() = 0;          // creates empty table
having length equal to num_buckets;
    virtual const_iterator bucket_find(int h, const Key& k) const =
0;          // find(k) for bucket h
    virtual const_iterator bucket_put(int h, const Key& k, const
Value& v) = 0; // put(k,v) for bucket h
    virtual const_iterator bucket_erase(int h, const_iterator loc) =
0;         // erase(v) for bucket h
```

```
  public:
    // Returns the number of entries in the map
    int size() const { return sz; }

    // Returns a const_iterator to the entry with a given key, or
end() if no such entry exists
    const_iterator find(const Key& k) const { return
bucket_find(get_hash(k), k); }

    // Removes entry referenced by given iterator; returns iterator
to next entry (in iterator order)
    const_iterator erase(const_iterator loc) {
        int h{get_hash(loc->key())};
        return bucket_erase(h, loc);
    }

    // Associates given key with given value. If key already exists
previous value is overwritten.
    // Returns an iterator to the entry associated with the key
    const_iterator put(const Key& k, const Value& v) {
        const_iterator result{bucket_put(get_hash(k), k, v)};
        if (sz > table_sz / 2)                          // keep
load factor <= 0.5
            resize(2 * table_sz - 1);                   // will
be pow(2,j) + 1 for some j
        return result;
    }
};

template <typename Key, typename Value, typename Hash =
std::hash<Key>>
class ChainHashMap : public AbstractHashMap<Key, Value, Hash> {
  protected:
    typedef AbstractHashMap<Key, Value, Hash> Base;
  public:
    using typename Base::Entry;                                 //
make nested Entry public
  protected:
    using typename Base::abstract_iter_rep, Base::get_rep,
Base::table_sz, Base::sz;
```

```
    typedef UnorderedListMap<Key,Value>  Bucket;                    //
each bucket will be a simple map
    typedef typename Bucket::const_iterator BCI;                    //
bucket const_iterator

    vector<Bucket> table;

    void create_table() {
        table.clear();
        this->table.resize(Base::table_sz);                         //
fills with empty buckets
    }

    class iter_rep : public abstract_iter_rep {                     //
specialize abstract version
      public:
        const vector<Bucket>* tbl{nullptr};                         //
need table to advance
        int bkt_num{0};                                             //
which bucket in table?
        BCI bkt_iter;                                               //
which location within that bucket?

        iter_rep(const vector<Bucket>* t, int b, BCI it) : tbl{t},
bkt_num{b}, bkt_iter{it} {}

        const Entry& entry() const { return *bkt_iter; }
        abstract_iter_rep* clone() const { return new
iter_rep(*this); }

        void advance() {
            ++bkt_iter;                                             //
try advancing within current bucket
            while (bkt_iter == (*tbl)[bkt_num].end()) {
                ++bkt_num;                                          //
advance one bucket
                if (bkt_num == tbl->size()) break;                  //
no buckets left
                bkt_iter = (*tbl)[bkt_num].begin();                 //
start at beginning of bucket
```

```
            }
        }

        bool equals(const abstract_iter_rep* other) const {
            const iter_rep* p = dynamic_cast<const
iter_rep*>(other);  // cast abstract argument
            if (p == nullptr) return false;    // failed cast
            return tbl == p->tbl && bkt_num == p->bkt_num &&
bkt_iter == p->bkt_iter;
        }
    }; // end class iter_rep

  public:
    using AbstractMap<Key,Value>::erase;                          //
makes key-based version accessible
    using typename AbstractMap<Key,Value>::const_iterator;

    // Creates an empty map
    ChainHashMap(int n = 17) {
      Base::table_sz = n;
      create_table();
  } // initializes table of buckets

    const_iterator begin() const {
        iter_rep* p = new iter_rep(&table, 0, table[0].begin());
        if (table[0].empty()) p->advance();                        //
advance to first actual entry (or end)
        return const_iterator(p);
    }

    const_iterator end() const {
        return const_iterator(new iter_rep(&table, table.size(),
table[table.size() - 1].end()));
    }

  protected:
    const_iterator bucket_find(int h, const Key& k) const {
// searches for k in bucket h
        BCI here{table[h].find(k)};
```

```
        if (here != table[h].end())
// found it!
            return const_iterator(new iter_rep(&table, h, here));
        else
            return end();
    }

    const_iterator bucket_put(int h, const Key& k, const Value& v)
{  // calls put(k,v) on bucket h
        int old_size{table[h].size()};
        BCI result{table[h].put(k,v)};
        Base::sz += (table[h].size() - old_size);
// one more if new key
        return const_iterator(new iter_rep(&table, h, result));
    }

    const_iterator bucket_erase(int h, const_iterator loc) {
// calls erase(loc) on bucket h
        const_iterator next{loc};
        ++next;
// precompute next location

table[h].erase(dynamic_cast<iter_rep*>(Base::get_rep(loc))->bkt_iter
);
        Base::sz--;
        return next;
    }
};

string generateStringFromKey(int n){
  string reversed = "";
  if (n == 0){
    return "0";
  }

  while (n > 0){
    int digit = n % 10;
    reversed += to_string(digit);
    n /= 10;
  }
```

```
  return reversed;
}

int main() {
  ChainHashMap<int, string> cmap(14);          // 11 / 0.75 = ~14.6
  // generating

  vector<int> testKeys = {13, 21, 5, 37, 15};
  for (int k : testKeys) {
    cmap.put(k, generateStringFromKey(k));
  }

  for (int key : {10, 21}) {
    auto it = cmap.find(key);
    if (it != cmap.end())
      cout << "Found key " << key << ": " << it->value() << endl;
    else
      cout << "Key " << key << " not found." << endl;
  }

  cmap.erase(20);
  cmap.erase(37);

  cout << "--- cmap contents after erase ---" << endl;
  for (auto it = cmap.begin(); it != cmap.end(); ++it) {
    cout << "Key: " << it->key() << ", Value: " << it->value() <<
endl;
  }

  ifstream file("small1k.txt");
  vector<int> smallData;
  int num;
  while (file >> num) smallData.push_back(num);
  file.close();

  ChainHashMap<int, string> smallMap(1334);    // 1000 / 0.75 = ~1334

  auto start = chrono::high_resolution_clock::now();
  for (int n : smallData)
```

```
    smallMap.put(n, generateStringFromKey(n));
  auto end = chrono::high_resolution_clock::now();

  chrono::duration<double> elapsed = end - start;
  cout << "Time to insert 1k into ChainHashMap: " << elapsed.count()
<< " seconds" << endl;

  ifstream file2("large100k.txt");
  vector<int> largeData;
  int num2;
  while(file2 >> num2) largeData.push_back(num2);
  file2.close();

  ChainHashMap<int, string> largeMap(133334); // 100000 / 0.75 =
~133334

  auto start2 = chrono::high_resolution_clock::now();
  for(int i : largeData)
    largeMap.put(i, generateStringFromKey(i));
  auto end2 = chrono::high_resolution_clock::now();

  chrono::duration<double> elapsed2 = end2 - start2;
  cout << "Time to insert 100k into ChainHashMap: " <<
elapsed2.count() << " seconds" << endl;

  return 0;
}
```

Input/output below:

```
PS C:\Users\benja\VSCODEFILES\csci-240>  & 'c:\Users\benja\.vscode\extensions\ms-vscode.cpptools-1.24.5-win32-x64
\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-xtlclvhu.lj5' '--stdout=Microsoft-MIE
ngine-Out-1i4vb3pf.y2l' '--stderr=Microsoft-MIEngine-Error-idnsnux4.oig' '--pid=Microsoft-MIEngine-Pid-aw0eipta.g
4b' '--dbgExe=C:\msys64\ucrt64\bin\gdb.exe' '--interpreter=mi'
Key 10 not found.
Found key 21: 12
--- cmap contents after erase ---
Key: 15, Value: 51
Key: 5, Value: 5
Key: 21, Value: 12
Key: 13, Value: 31
Time to insert 1k into ChainHashMap: 0.016784 seconds
Time to insert 100k into ChainHashMap: 3.05101 seconds
PS C:\Users\benja\VSCODEFILES\csci-240>
```

Exercise 3 -- need to submit source code and I/O
-- check if completely done ✅ ; otherwise, discuss issues below
Pseudocode below if applicable:

Source code below:

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <set>
#include <map>
#include <vector>

using namespace std;

const int BIG_PRIME = 2147483647; // 2^31 - 1

// Polynomial hash function: hash(s) = s[0]*a^(n-1) + ... +
s[n-1]*a^0 mod BIG_PRIME
int polynomial_hash(const string& s, int a) {
    unsigned long long hash = 0;
    for (char c : s) {
        hash = (hash * a + c) % BIG_PRIME;
    }
    return static_cast<int>(hash);
}

int main() {
    ifstream file("USDeclIndFormatted.txt");
    if (!file) {
        cerr << "Failed to open usdeclar.txt" << endl;
        return 1;
    }

    set<string> unique_words;
    string word;
    while (file >> word) {
        unique_words.insert(word);
    }
    file.close();

    cout << "Unique words: " << unique_words.size() << endl << endl;
```

```
    vector<int> a_values = {1, 37, 40, 41};

    for (int a : a_values) {
        map<int, vector<string>> hash_map;

        for (const string& w : unique_words) {
            int h = polynomial_hash(w, a);
            hash_map[h].push_back(w);
        }

        int collision_count = 0;
        for (const auto& [hash_val, word_list] : hash_map) {
            if (word_list.size() > 1)
                collision_count++;
        }

        cout << "a = " << a << " → collisions: " << collision_count
<< endl;
    }

    return 0;
}
```

Input/output below:

```
PS C:\Users\benja\VSCODEFILES\csci-240>  & 'c:\Users\benja\.vscode\extensions\ms-vscode.cpptools-1.24.5-win32-x64
\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-tt041yay.vxz' '--stdout=Microsoft-MIE
ngine-Out-weerhhmq.pqw' '--stderr=Microsoft-MIEngine-Error-vox04bjl.z23' '--pid=Microsoft-MIEngine-Pid-p5wmdt3z.h
cs' '--dbgExe=C:\msys64\ucrt64\bin\gdb.exe' '--interpreter=mi'
Unique words: 539

a = 1 → collisions: 126
a = 37 → collisions: 0
a = 40 → collisions: 0
a = 41 → collisions: 0
PS C:\Users\benja\VSCODEFILES\csci-240>
```

Answer for Question 1

Yes, the collected times for Excercise 1 and 2 make sense given the differences in implementation. In Ex. 1, the C++ unordered_map is highly optimized and uses efficient internal hashing strategies, resulting in very, very, fast insertions. In contract, Exercise 2 uses a custom ChainHashMap with separate chaining via UnorderedListMap buckets. This complexity introduces overhead from dynamic memory management, pointers, and polymorphism. Which explains the noticeably longer runtime, despite the slower speed, the time still grows linearly with the number of insertions, which align with the expected average-case performance for hash tables using chaining.

Answer for Question 2
Compression functions are essential in hashing due to the need of converting potentially large hash codes into valid indices within a fixed-size table. Without them, hash values could exceed the table bounds or distribute unevenly, resulting in clustering & poor performance. A good compression functions ensures a uniform distribution of keys, reducing collisions and maintaining efficient access times. Common compression methods inclue the modulo method (h(k) % N) an dmore advanced ones like the MAD method. In exercise 3, using different base values in the polynomial hash affected the collision rate, showing how both hashing and compression influence performance.

Extra Credit – provide if applicable
Pseudocode below if applicable:

Source code below:

```
#include <iostream>
#include <fstream>
#include <string>
#include <set>
#include <map>
#include <vector>

using namespace std;

const int BIG_PRIME = 2147483647; // 2^31 - 1

// Polynomial hash function: hash(s) = s[0]*a^(n-1) + ... +
s[n-1]*a^0 mod BIG_PRIME
int polynomial_hash(const string& s, int a) {
    unsigned long long hash = 0;
    for (char c : s) {
        hash = (hash * a + c) % BIG_PRIME;
    }
```

```
    return static_cast<int>(hash);
}

int cyclic_shift_hash(const string& s) {
  unsigned int hash = 0;
  for (char c : s) {
      hash = (hash << 5) | (hash >> 27); // rotate left 5 bits
      hash += c;
  }
  return static_cast<int>(hash % BIG_PRIME);
}


int main() {
    ifstream file("USDeclIndFormatted.txt");
    if (!file) {
        cerr << "Failed to open usdeclar.txt" << endl;
        return 1;
    }

    set<string> unique_words;
    string word;
    while (file >> word) {
        unique_words.insert(word);
    }
    file.close();

    cout << "Unique words: " << unique_words.size() << endl << endl;

    vector<int> a_values = {1, 37, 40, 41};

    for (int a : a_values) {
        map<int, vector<string>> hash_map;

        for (const string& w : unique_words) {
            int h = polynomial_hash(w, a);
            hash_map[h].push_back(w);
        }

        int collision_count = 0;
```

```
        for (const auto& [hash_val, word_list] : hash_map) {
            if (word_list.size() > 1)
                collision_count++;
        }

        cout << "a = " << a << " → collisions: " << collision_count
<< endl;
    }

    map<int, vector<string>> cylic_hash_map;
    for (const string& w : unique_words) {
      int h = cyclic_shift_hash(w);
      cylic_hash_map[h].push_back(w);
    }

    int cyclic_collision_count = 0;
    for (const auto& [hash_val, word_list] : cylic_hash_map) {
      if (word_list.size() > 1)
        cyclic_collision_count++;
    }

    cout << "Cyclic shift hash collisions: " <<
cyclic_collision_count << endl;

    return 0;
}
```

Input/output below:

```
PS C:\Users\benja\VSCODEFILES\csci-240> & 'c:\Users\benja\.vscode\extensions\ms-vscode.cpptools-1.24.5-win32-x64
\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-cjvmbyrf.0mx' '--stdout=Microsoft-MIE
ngine-Out-psvmsvsa.z1c' '--stderr=Microsoft-MIEngine-Error-1mnrpnuj.lbe' '--pid=Microsoft-MIEngine-Pid-cw4sgqro.y
ob' '--dbgExe=C:\msys64\ucrt64\bin\gdb.exe' '--interpreter=mi'
Unique words: 539

a = 1 → collisions: 126
a = 37 → collisions: 0
a = 40 → collisions: 0
a = 41 → collisions: 0
Cyclic shift hash collisions: 0
PS C:\Users\benja\VSCODEFILES\csci-240>
```