CMSC 341 — Data Structures — Section 01 — Fall 2016

Project 5: Pinball Hash

Due: Tuesday, December 6, 8:59:59pm

Addenda

- Monday, November 21, 13:05. [Already?] Added an implementation note on duplicate auxiliary slots.

Objectives

The objective of this programming assignment is to have you evaluate experimentally the performance of a hash table data structure.

Introduction

For this project, you will implement a variation of Cuckoo Hashing and evaluate its performance experimentally. This hashing scheme, which we will call Pinball Hashing, takes elements of Cuckoo Hashing and combines it with ideas from the study of expander graphs. The entire class will participate in this experimental evaluation. Each student will be assigned a particular version of the hashing scheme and report his/her observations. Collectively, we may be able to produce some advice on how best to implement Pinball Hashing.

Pinball Hashing is an open addressing scheme (like linear probing and quadratic probing) — that is, all of the items are stored in the array of the hash table. There are not any additional data structures — no linked lists — just an array and a size.

Pinball Hashing differs from linear and quadratic probing in that for each item there is a small number of locations where an item might be placed. (In this project, we will experiment with this small number being 5, 7 or 9.) We will call this value the *degree* of the Pinball hash table.

So suppose that we have a Pinball hash table with degree 5. That means there are 5 slots in the hash table where an item may be placed. Let's set aside for now how these 5 slots are determined, except to say that they can be quickly computed from the hash value. To find an item in the hash table, we simply look in all 5 slots. If we find the item, then we are done. If the item is not in any of the 5 slots, then the item is not in the hash table. In contrast, in linear probing and quadratic probing, an item might theoretically be in any slot of the hash table. So searching in a Pinball hash table is very simple.

What about insertion? Suppose we are hashing strings using a hash function $h(\,)$. Suppose that $h($`"aardvark"`$) = 713$. We call 713 the primary slot for `"aardvark"`. If slot 713 is available, then we put `"aardvark"` in that slot and we are done. If 713 is not available, we look to the other 4 *auxiliary* slots for `"aardvark"`. Let's say these are slots number 973, 1516, 72 and 311. (We will consider different ways of computing the locations of the auxiliary slots. That's why we are being a bit vague about how they are determined for now.) If any of these 4 auxiliary slots are available, then we simply put `"aardvark"` in the available auxiliary slot. What if all of the auxiliary slots are taken? Then we randomly choose one of the auxiliary slots, *eject* the string in that auxiliary slot from the table and then put `"aardvark"` in that slot. So, let's say `"bison"` was ejected from slot 1516 and we put `"aardvark"` in slot 1516 instead. What do we do with `"bison"`? Well, we insert it back into the hash table and hope that one of its 4 other slots is available. What if all of the other slots for `"bison"` are also full? Then `"bison"` will eject another item, maybe `"cheetah"`, and we hope that `"cheetah"` has an available slot ...

This process of ejecting items and reinserting them corresponds to a random walk in a graph. (To be technical, the vertices of the graph are the slots of the hash table and two slots/vertices are connected by an edge if one vertex is a primary slot and the other vertex is an auxiliary slot for the same item.) The theory of expander graphs says that if a good fraction of the slots in the

hash table is available, then a random walk will quickly get to an available slot. What is "quickly"? and does the theory actually work? We don't know. That's why we are having you do this experiment.

There are two additional issues to consider. First, it is possible for a random walk to loop back and visit a vertex/slot that it has visited before. In the example above, perhaps re-inserting `"cheetah"` causes `"aardvark"` to be ejected and we are back where we started. Second, it is possible that the hash function maps 6 strings to the same primary slot. In that case, it would be impossible for these 6 strings to be inserted in the primary slot and 4 auxiliary slots. Every time a string gets ejected and re-inserted into the hash table, it will be hashed to the same primary slot again. Again, we are back where we started.

Both of these issues can be handled by limiting the number of ejections that are allowed during a single insert operation. If the limit is exceeded, we just throw up our hands and declare that the hash table is full. (In the implementation, you will throw an exception, not hands.) What should be the limit on the number of ejections resulting from a single insertion? is 8 good? If the limit is too low, then the random walk will not be long enough to find an available slot. If it is too high, then we may be wasting our time in the random walk without finding an available sot. Is 14 better? how about 20? We don't know. That's why we are having you do this experiment.

Note that the probability of looping back is reduced if we increase the degree. In the first case, since we pick an item among the auxiliary slots randomly, a larger degree implies a lower probability of picking an auxiliary slot that leads to looping back. A bigger degree also means more strings have to hash to the same primary slot before the hash table is declared full. On the other hand, recall that a larger degree does mean a slower search, since the search function must look in all the auxiliary slots. So, is a degree of 5 good enough? would 7 be better? or maybe even 9? or is 9 too big? We don't know. That's why we are having you do this experiment.

Finally, we still need to describe how the auxiliary slots are determined. The short answer is randomly. (The longer answer involves expander graphs and the theorem that randomly generated graphs tend to have good expansion properties.) This is somewhat counter-intuitive because we have to be able to recover the locations of the auxiliary slots every time. Otherwise search would not work. How can we find the auxiliary slots if they are "random"? We approach this in three ways: pseudo-random, a little bit random and not very random.

- **Option #1: pseudo-random**: in this approach we store a seed for the random number generator at each slot of the hash table. Suppose that seed `s` is stored in slot `t`. Every time we work with slot `t` as the primary slot, we call `srand(s)` to set the random number generator. (We do this for the insert, delete and search functions of the hash table.)
  The calls to `rand()` after setting this seed will always produce the same values. If the degree of the hash table is 5, then the 4 calls to `rand()` after setting the seed will always produce the same 4 values in the same order. We will take those 4 values to be the indices of the auxiliary slots.

  Where do we get these seeds to store in the hash table? We generate them randomly, of course. (See A Note on Pseudo-Random Generators and Implementation Notes for details.)

- **Option #2: a little bit random**: The first approach requires additional storage to save the random seed. In this second approach, we simply use the index of the primary slot as the random seed. The rest is the same as the first approach.

- **Option #3: not very random**: Another way to avoid storing a random seed is to randomly generate a few *offsets* for the entire hash table. In a degree 5 hash table, we generate 4 random offsets. Suppose these offsets are 30, 72, 316 and 996. Then the auxiliary slots for primary slot 10 are 40, 82, 326 and 1006. (We just add the offsets to 10.) Similarly,the auxiliary slots for primary slot 34 are 64, 106, 350 and 1030.

Do note that in each of these approaches, the index of the auxiliary slots have to be taken modulo the table size (which should be a prime number).

Which of these 3 approaches work better? Should we just store a random seed in each slot? use the index as the seed? or just use the same offsets for the entire table? We don't know. That's why we are having you do this experiment.

Your Assignment

Your assignment is to implement *and test* a version of the Pinball Hashing scheme described above. You should have received an individual email message from Prof. Chang that assigned a specific version to implement. The specification includes:

- The **degree** of the Pinball Hash table. (Recall that the degree is the number of auxiliary slots + 1.)
- The **maximum number of ejections** allowed when inserting a new item in the hash table.
- How much randomness to use when determining the auxiliary slots (i.e., one of Option #1 pseudo-random, Option #2 a little bit random and Option #3 not very random).

Your implementation must be compatible with the header file given below. You may add additional data members and member functions, but you must not change the members and functions already given. Note that the `hashCode()` member function is already implemented in the header file as an inline definition. This is so we all use the same hash code function. Otherwise, the differences in the experiments that you conduct may be due to differences in the hash code used. Details on the required member functions are given in the Additional Specifications section.

```
#ifndef _PINBALL_H_
#define _PINBALL_H_

#include <string>
#include <stdexcept>
using namespace std ;

class PinballHashFull : public std::out_of_range {

public:
   PinballHashFull(const string& what) : std::out_of_range(what) { }
} ;


class Pinball {

public:
   Pinball(int n=1019) ;    // default capacity is prime
   ~Pinball() ;

   void insert(const char *str) ;
   int find(const char *str) ;
   const char * at(int index) ;
   char * remove(const char *str) ;
   void printStats() ;

   int size() { return m_size ; }

   // add public data members and
   // public member functions as needed


private:

   char ** H ;        // the actual hash table
   int m_size ;       // number of items stored in H
   int m_capacity ;   // number slots allocated in H

   int m_degree ;     // use degree assigned to you
   int m_ejectLimit ; // use ejection limit assigned to you


   // Made inline definition of hashCode() so everyone uses
   // the same function.
   //
   unsigned int hashCode(const char *str) {

      unsigned int val = 0 ;
      const unsigned int thirtyThree = 33 ;  // magic number from textbook

      int i = 0 ;
```

```
      while (str[i] != '\0') {
         val = val * thirtyThree + str[i] ;
         i++ ;
      }
      return val ;
   }


   // add private data members and
   // private member functions as needed

} ;


#endif
```

Download: Pinball.h

We will be hashing strings. The strings are given in a global array `words[]`. None of the member functions in your implementation of the `Pinball` class should use this global array. The driver programs should be the only code that uses the `words[]` array. The global variable `numWords` has the size of the `words[]` array.

```
#ifndef _WORDS_H_
#define _WORDS_H_

const int numWords = 58110 ;

const char *words[numWords] = {
"aardvark",
"aardwolf",
"aaron",
"aback",
"abacus",
"abaft",
"abalone",
"abandon",
"abandoned",
"abandonment",
"abandons",
.
.
.
"zoomed",
"zooming",
"zooms",
"zooplankton",
"zoos",
"zulu",
"zulus"
} ;

#endif
```

Download: words.h.

Note that the items in the `words[]` array are `const char *` strings and not C++ strings. When you store a string in `H`, you should make a copy of the given string.

Here is an example driver program that uses the Pinball class:

```
/* File: driver.cpp

   Main program used to test Pinball Hash Tables.
*/

#include <iostream>
#include <time.h>
#include <sys/resource.h>
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "words.h"
#include "Pinball.h"

using namespace std ;


// Get amount of user time since the program began in milliseconds.
//
double getmsecs() {
   struct rusage use ;
   struct timeval utime ;

   getrusage(RUSAGE_SELF, &use) ;
   utime = use.ru_utime ;
   return ((double) utime.tv_sec)*1000.0 +  ((double) utime.tv_usec)/1000.0 ;
}


// A "bug-proof" way to call rand() that preserves the state of the
// random seed between calls to rand().
// Set the random seed by passing a non-zero parameter.
//
int myRand(int seed=0) {

   // static local variables persist between calls
   static int savedSeed ;

   if (seed) savedSeed = seed ;

   int othersSeed = rand() ;  // save other people's seed
   srand(savedSeed) ;         // restore my seed
   int result = rand() ;
   savedSeed = result ;       // save for next time
   srand(othersSeed) ;        // restore other people's
   return result ;
}


// Exercise the Pinball Hash table with size slots
// inserting reps items.
//
void test(int size, int reps) {

   int index ;
   int slot ;
```

```
   double startTime, stopTime ;

   Pinball PH(size) ;

   startTime = getmsecs() ;

   // Insert reps randomly chosen items
   //
   for (int i=0 ; i < reps ; i++) {

      // don't choose items already in the hash table
      do {
         index = myRand() % numWords ;
         slot = PH.find(words[index]) ;
      } while(slot > -1) ;
      try {

         PH.insert(words[index]) ;  // might throw exception


         // double check that inserted items were stored properly
         // for debugging.

         /*
         slot = PH.find(words[index]) ;
         if (slot >=0 && strcmp(PH.at(slot),words[index]) != 0) {
            cout << "Inserted word not stored at slot = " << slot << endl ;
            cout << "   words[" << index << "] = " << words[index] << endl ;
            cout << "   PH.at(" << slot << ") = " << PH.at(slot) << endl ;
         }
         */

      } catch (PinballHashFull &e) {
         cout << e.what() << endl ;
         break ;
      } catch (...) {
         cout << "Unknown error\n" ;
      }
   }

   stopTime = getmsecs() ;
   printf("Elapsed time = %.3lf milliseconds\n", stopTime - startTime) ;

   PH.printStats() ;
}


int main() {

   // set random seed to wall clock time in milliseconds
   //
   struct timeval tp;
   gettimeofday(&tp, NULL);
   long int ms = tp.tv_sec * 1000 + tp.tv_usec / 1000;
   printf("Random seed set to: %ld\n\n", ms) ;
   myRand(ms) ;
   srand(ms*37) ;   // other uses

   // When debugging/developing, uncomment a line below and
   // use a fixed random seed.
```

```
   //
   // myRand(82170) ; srand(82170*37) ;
   // myRand(3170325890) ; srand(3170325890*37) ;
   // myRand(9132919912) ; srand(9132919912*37) ;

   test(5003,4000) ;   // >5,000 slots, ~80% full
   printf("\n\n") ;

   test(10037,8000) ;  // >10,000 slots, ~80% full
   printf("\n\n") ;

   test(20101,16000) ;  // >20,000 slots, ~80% full
   printf("\n\n") ;

   return 0 ;
}
```

Download: driver.cpp.

Running the test program might produce output that looks something like this:

```
Random seed set to: 1479302151417

Elapsed time = 3.158 milliseconds
Hash statistics report:
   randomness level = pseudo-random
   capacity = 5003
   size = 4000
   degree = 7
   ejection limit = 14
   number of primary slots  = 2740
   average hits to primary slots = 1.459854
   maximum hits to primary slots = 6
   total number of ejections = 230
   maximum number of ejections in a single insertion = 11


*** Exception: Maximum ejections reached!
Elapsed time = 3.982 milliseconds
Hash statistics report:
   randomness level = pseudo-random
   capacity = 10037
   size = 5981
   degree = 7
   ejection limit = 14
   number of primary slots  = 4546
   average hits to primary slots = 1.315882
   maximum hits to primary slots = 6
   total number of ejections = 70
   maximum number of ejections in a single insertion = 15


Elapsed time = 8.854 milliseconds
Hash statistics report:
   randomness level = pseudo-random
   capacity = 20101
   size = 16000
   degree = 7
```

```
ejection limit = 14
number of primary slots  = 11044
average hits to primary slots = 1.448750
maximum hits to primary slots = 6
total number of ejections = 1099
maximum number of ejections in a single insertion = 9
```

Here are two more sample main programs and output. (Your output may look different.)

- smalltest.cpp and smalltest.txt
- mediumtest.cpp and mediumtest.txt

After you have successfully implemented the Pinball class **and** have your code fully debugged **and** `valgrind` reports that there are no memory leaks, run your program **on GL** with the hash table sizes 5003, 10037 and 20101 (these are prime numbers) with the hash table at 50%, 60%, 70%, 80% and 90% full. At each size and load factor, run your program 10 times and compute the average. (Hint: modify the driver program to do this and report the average.) Finally, report your results in these Google forms:

- CMSC 341 Proj5 Data, N=5003 (Red),
- CMSC 341 Proj5 Data, N=10037 (Green),
- CMSC 341 Proj5 Data, N=20101 (Blue),

You can either have your program compute the averages, or fill in this Proj5 Data Worksheet (make a copy).

Additional Specifications

Here are some additional specifications for the `Pinball` class member functions that you have to implement. You will need to add data members and member functions to the `Pinball` class, but the prototypes for the member functions listed here should not be changed.

- ```
  Pinball(int n=1019) ;
  ```

  This is the default constructor for the `Pinball` class. The parameter `n` is the size of the hash table. If no size is given, use 1019 which is a prime number. You must allocate space for the `H` array and initialize it (and also for other data members that you create).

- ```
  ~Pinball() ;
  ```

  This is the destructor. Make sure you deallocate all memory for this object. Strings in the `H` array must be deallocated using `free()` since they are C strings (i.e., don't use `delete`).

- ```
  void insert(const char *str) ;
  ```

  This function inserts a copy of the C string `str` into the hash table. It has no return value. (Note: use `strdup()` to copy C strings.) If the hash table is full or the maximum number of ejections was exceeded, then `insert()` should throw a `PinballHashFull` exception. (This exception is already defined in `Pinball.h`.)

  Calling `insert()` with a string that is already in the hash table should have no effect. (I.e., do not insert a second copy of
  ```

the same value.)

- ```
  int find(const char *str) ;
  ```

  The `find()` function looks for `str` in the hash table. If found, the index of `str` is returned. If `str` is not in the hash table, `find()` should return -1.

  The location returned by `find()` is only valid until the next call to `insert()` or to `remove()`.

- ```
  const char * at(int index) ;
  ```

  The `at()` function returns a pointer to the string stored at the `index` slot of the hash table. If the `index` is invalid (i.e., less than 0 or greater than or equal to `m_capacity`), then `at()` should throw an `out_of_range` error (already defined in `stdexcept`).

  The pointer returned has type `const char *` to prevent the string stored in the hash table from being changed. The calling function can make a copy if desired.

- ```
  char * remove(const char *str) ;
  ```

  The `remove()` function removes `str` from the hash table and returns the pointer. If `str` is not in the hash table, `remove()` returns NULL.

  It is the responsibility of the code that calls `remove()` to deallocate the string that is returned. (Again, use `free()`, not `delete` to deallocate.)

- ```
  void printStats() ;
  ```

  Print out some statistics about the hash table. (See sample output.) This requires cooperation from `insert()` and `remove()` and additional data members in the class. Some clarifications:

  - A slot is a primary slot if some item that is currently in the hash table hashes to that slot. The **number of primary slots** is the number of slots in the table that are currently considered primary slots. This is less than the number of items in the hash table because of collisions. I.e., some slots are filled because they are being used as auxiliary slots.
  - **average hits to primary slots** is just `m_size` divided by the number of primary slots. This gives us an idea of the number of collisions.
  - **maximum hits to primary slots** is the maximum number of items that hash to the same primary slot considering only the items that are currently in the hash table. If this number exceeds the degree, we have to rehash.
  - **total number of ejections** is the number of ejections performed in all insertions since the hash table was created.
  - if the **maximum number of ejections in a single insertion** exceeds the ejection limit, then `insert()` would have thrown an exception. This number tells us if we were getting close to the ejection limit.

A Note on Pseudo-Random Generators

Library functions like `rand()` are called pseudo-random generators because they provide some semblance of randomness but the output they produce is not truly random in a mathematical sense. (There are several definitions of "random".)

A typical pseudo-random generator will output a predetermined sequence of numbers in a fixed order:

$r_1, r_2, r_3, \ldots r_i, \ldots$

Calling `srand()` with a seed value simply tells the `rand()` function to start at a different place in the sequence. Where it will start in the sequence varies from system to system depending on which version of `rand()` is installed. For example, on GL,

after calling

```
srand(191332) ;
```

the next few calls to `rand()` will always return the values:

```
2116609228
177487921
450639930
1944808177
1241803817
635976569
```

(On some systems, calling `srand(2116609228)` means the next call to `rand()` will give `177487921`, but this doesn't happen with the `rand()` function installed on GL.)

This presents a problem for us because we want to use the `rand()` function in several ways. In Option #1 and #2, we need to reset the random seed to a particular value so we can locate the same auxiliary slots every time. But that would mean all subsequent calls to `rand()` will always produce the same values! That's not very random.

Consider a concrete example. Suppose that we have degree=5 in our Pinball Hash table and we are implementing Option #1. Also, suppose that we want to insert in primary slot #54 and that we stored the seed 191332 for slot #54. (These numbers are from the example above.) When we look for the auxiliary slots for slot #54, we call `srand(191332)` and the next 4 calls to `rand()` will give us 2116609228, 177487921, 450639930 and 1944808177 which we will mod out by the table size to get the indices for the 4 auxiliary slots. This is all as expected. We want to get the same 4 random numbers every time we look for the 4 auxiliary slots for slot #54.

就是如果使用固定的随机数种子，那么每次如果发生了collision，eject的时候低n+1次都是一样的，

Now suppose that all the slots are full for this insertion. We want to randomly pick an auxiliary slot for ejection. If we just call `rand()` again, we will get 1241803817 every time because that is always the value returned by the fifth call to `rand()` after setting the seed to 191332. That means we will always pick the same auxiliary slot for ejection. This is very bad.

We can work around this by "saving some randomness" before we set the random seed and "restoring" the randomness later.

```
int savedSeed = rand() ;
srand(191332) ;
aux[0] = rand() % N ;   // first auxiliary slot
aux[1] = rand() % N ;   // second auxiliary slot
aux[2] = rand() % N ;   // third auxiliary slot
aux[3] = rand() % N ;   // fourth auxiliary slot
...
srand(savedSeed) ;       // restore some randomness
pickAux = rand() % 4 ;   // pick an aux. slot
```

This will not put the pseudo-random number generator in exactly the same state as prior to the call to `srand()`, but we do not need to have that. We just want to make sure that we don't pick the same auxiliary slot each time.

A similar trick is used in the `myRand()` function in driver.cpp and mediumtest.cpp. The `myRand()` function is used to pick a random word from the global `words[]` array. Calls to the Pinball Hash `insert()` and `find()` functions might leave the random number generator in a bad state (as in a not very random state). So, the `myRand()` function saves its own random seed with each call.

```
int myRand(int seed=0) {

   // static local variables persist between calls
   static int savedSeed ;

   if (seed) savedSeed = seed ;

   int othersSeed = rand() ;  // save other people's seed
   srand(savedSeed) ;         // restore my seed
```

```
    int result = rand() ;
    savedSeed = result ;       // save for next time
    srand(othersSeed) ;        // restore other people's
    return result ;
}
```

Because of the different `rand()` functions installed on different systems, please run your experiments on GL only. That way when we compare results reported by different students, we can be assured that the differences are not due to different pseudo-random generators used.

Implementation Notes

- Remember to mod out by the table size when you are working with hash table indices.
- You are working with C null-terminated strings, not C++ strings. If you haven't used C strings in a while, please review. For example, a C string is just an array of `char`, so the type is `char *`. A dynamically allocated array of C strings has type `char * *` since it is a pointer to an array of pointers to `char`. Here are Mr. Lupoli's notes on C strings.
- You must use the `strcmp()` function to make string comparisons. You cannot use the == operator. That will do pointer comparison, not string comparison. To use `strcmp()`, make sure you

  ```
  #include <string.h>
  ```

- Your hash table should make a copy of the string inserted, and not just store the pointer. Copies should be made using the `strdup()` function.
- The `strdup()` function allocates memory using `malloc()` instead of `new`. To deallocate this memory, you must use `free()` instead of `delete`. While it is possible to allocate an array of `char` using `new` like this:

  ```
  char *str = new char[15] ;
  ```

  it would be very confusing for one program to sometimes use `malloc()` and sometimes use `new` to allocate strings. So, just stick to using `malloc()` and `free()` to allocate and deallocate C strings.
- Some of the strings that you are working with have type `const char *`. This means the string being pointed to cannot be changed. (The string is immutable in Python-speak.) Make sure that you know what this means. If you assign a `const char *` pointer to a `char *` pointer, the compiler will give you an error.
- When you eject a string and re-insert it, make sure there is no memory leak.
- Discussion topic #3 says that it is possible for an auxiliary slot to appear multiple times in the list of auxiliary slots for a single primary slot. For most implementations, this won't change anything. Think about your code and convince yourself that it doesn't. (And if it does change things, then modify your code so it doesn't.)
- When your `insert()` function is past the ejection limit, before you throw the `PinballHashFull` exception, you must free or delete any dynamically allocated local variables or parameters. Otherwise, you could have a memory leak.
- Don't forget to update statistics when you remove.
- On GL, in order to use the `rand()` and `srand()` functions, you must:

  ```
  #include <stdlib.h>
  ```

- Doing several runs of the same program only makes sense if the random values generated differ from run to run. You should set the random seed to the clock value at the beginning of `main()` as shown in the example driver program. For debugging, you should set the random seed to a fixed value. Otherwise, you cannot reproduce your errors.

Files

Here are the available files all in one place:

- Pinball.h

- driver.cpp
- driver.txt
- mediumtest.cpp
- mediumtest.txt
- smalltest.cpp
- smalltest.txt
- words.h (big!)

What to Submit

You must submit the following files to the `proj5/src` directory.

- `Pinball.h`
- `Pinball.cpp`
- `myDriver.cpp`
- `output.txt`

The `output.txt` file should be the concatenation of the output from all runs of your program. You can concatenate files together using the Unix `cat` command:

```
cat run1.txt run2.txt run3.txt  > output.txt
```

Don't forget to report your data in the Google form:

- CMSC 341 Proj5 Data, N=5003 (Red),
- CMSC 341 Proj5 Data, N=10037 (Green),
- CMSC 341 Proj5 Data, N=20101 (Blue),

(You must be logged into your UMBC Google account.)

Discussion Topics

Here are some topics to think about:

1. The description above only picks among the auxiliary slots for ejection. Can we eject the string stored in the primary slot? Explain.
2. How can you determine if the number of items that hash to the same primary slot exceeds the degree? Is it worthwhile to maintain this information in the "production" code?
3. Is it possible for two auxiliary slots picked in the manner described above to end up being the same index in `H`? (Hint: Yes.) How? Does it matter? (Hint: Shouldn't.) What if an auxiliary slot picked randomly just happens to have the index of the primary slot?