CIS 240 – Data Structure Project
Due: April 30, 2023 at 11:59 p.m.

For this project, students can work in pair to complete the tasks specified. You must submit your source files with names of group members and short description of the files. Make sure your names are in each file submitted.

Introduction

The hash table makes searching a lot faster. As long as the table does not become too full, the time for adding and finding an element will be O(1). This performance does not come without a cost. The obvious penalty is that there will be space in the table that is wasted. Another penalty is that the items in the hash table are not in any particular order. It is an inherent property of the hash table that items are not ordered. In fact, as more items are added to the hash table, the size of the table may be increased to maintain the performance. In this case, the items will be rehashed and will no longer be in the same locations or order.

General Collision Resolution

To place an item in a hash table of size m, a hash function H(k) is applied to the key k. An integer value between 0 and m–1 will be returned and will be the location of the object. If there is already an object in that location, a collision has occurred and must be resolved. In a hash table with open addressing, collisions are resolved by trying other locations until an empty slot is found. One way of viewing this process is that there is a series of hash functions $H_0()$, $H_1()$, $H_2()$, $H_3()$, ... $H_i()$, ..., which are applied one at a time until a free slot is found.

Linear Probing

For linear probing, slots in the hash table are examined one after another. From the view of the general scheme, the hash functions are

$H_0(k) = H(k)$
$H_1(k) = (H(k)+ 1) \bmod m$
$H_2(k) = (H(k) + 2) \bmod m$
...
$H_i(k) = (H(k)+ i) \bmod m$

The mod operation is required to keep the values in the range from 0 to m–1. While you could use these formulas to compute each of the hash locations, usually the previous value is used to compute the next one.

$H_i(k) = (H_{i-1}(k) + 1) \bmod m$

Linear hashing has the advantage of a simple computational formula that guarantees all the slots will be checked. The performance of linear hashing is affected by the creation of clusters of slots that are filled. Linear hashing tends to create relatively few clusters that are large in size. If there is a collision with a slot inside a cluster, finding an empty slot outside of the cluster will then require a large number of probes. The hash table will have its best performance if the free slots are distributed evenly and large clusters avoided.

Double Hashing

Double hashing is scheme for resolving collisions that uses two hash functions H(k) and h(k). It is similar to linear probing except that instead of changing the index by 1, the value of the second hash function is used.

In the view of the general scheme, the hash functions are

$H_0(k) = H(k)$
$H_1(k) = (H(k)+ h(k)) \bmod m$
$H_2(k) = (H(k)+ 2\, h(k)) \bmod m$
...
$H_i(k) = (H(k)+ i\, h(k)) \bmod m$

As with linear hashing, the hash function can be defined in terms of the previous values.

$H_i(k) = (H_{i-1}(k)+ h(k)) \bmod m$

You must be careful when defining the second hash function.

Since you really want to probe the entire table, the value returned by the second hash function has some limitations. The first condition is that it should not be 0. The second condition is that it should be relatively prime with respect to m. A common way to guarantee the second condition is to choose a table size that is a prime.

Double hashing can still be affected by clustering (though to a lesser extent than linear hashing). Every key that has the same value for the second key will probe the table in the same pattern and can still be affected by clusters.

Tasks to Do

In this project, students need to implement two dynamic array based hash tables, one with linear probing and the other with double hashing. The hash tables implemented should be able to perform insertion, removal, and search. Also, expand the hash table when there are too much data.

Add the following as a private member into your two implementations of hash table. The arrays within the hash table should be arrays of Entry.

```
class Entry
{
    private:
        int data;
        bool occupied;
    public:
        Entry(){
            data = 0;
            occupied = false;
        }
        void setData(int d)
        {
              data = d;
              occupied = true;
         }
         int getData() const
         {
             return data;
         }
         bool isOccupied() const
         {
              return occupied;
          }
         void setToRemoved()
          {
               occupied = false;
          }
          bool isUndefined() const
          {
               return (!occupied && data==0);
           }
};
```

Below is a list of public function members that both hash tables should have:

- Constructor(initial table size)
- Copy constructor
- Assignment operator
- Destructor
- Insert(data)
- Remove(data)
- Search(data)
- Count() – returns the number of data in the hash table
- isEmpty()
- isFull()

Use the following hash functions in your hash tables:

- H(k) = k % table_size while table_size is a prime number
- h(k) = k % (table_size – 2) + 1

Add the following function as a public members in your hash tables.

```
int probe(Entry key) {
        int totalProbes=0;
        boolean found = false;
        int index = HashFunction(key);

        int hashDelta = 1;  // need to modify this line for the hash table with double hashing

        while (!found && !arr[index].isUndefined()) {
            if (arr[index].isOccupied()) {
                if (key == (arr[index].getData())) {
                    found = true; // Key found
                } else{  // Follow probe sequence
            index = (index + hashDelta) % table_size;
          }
        } else // Skip entries that were removed
        {
          index = (index + hashDelta) % table_size;
        }
        totalProbes++;
      } // end while

      if (!found) totalProbes++;

      return totalProbes;
    }
```

<u>Suggestion: implement some private function members within your hash table classes to reduce repetition of codes.</u>

Write a program that asks the user for initial table size, number of data to insert. The program should then generate random data and insert those data into the hash tables. Finally, call the probe() function to obtain the statistics needed and complete the tables below.

<u>Number of Probes for theTwo Kinds of Hash Tables:</u>
The initial table size should be set to 5

NUMBER OF ITEMS INSERTED	TOTAL PROBES FOR LINEAR PROBING	TOTAL PROBES FOR DOUBLE HASHING
10	65	65
20	230	230
30	495	477/495
40	860	815
50	1325	1294
60	1831	1799
70	2440	2355
80	3101	3106
90	3926	3787
100	5150	5094
110	6215	6013

<u>Number of Probes with Respect to the Initial Table Size:</u>
Use 1000 for the number of items to insert

INITIAL TABLE SIZE	AVERAGE PROBES FOR LINEAR PROBING	AVERAGE PROBES FOR DOUBLE HASHING
50	9135	13715
100	4343	6633
250	1227	2181

500	613	1461
1000	185	1083
2000	92	502