Lab 10 - Hash Tables

CS 251, Fall 2024

Copyright Notice

Exam 3

This exam is on **paper**. Please follow your TAs' instructions. Don't open the exam until instructed to do so.

Learning Goals

By the end of this lab, you'll:

- Have implemented a memory-safe rehashing algorithm for an externally chaining hash table

Starter Code

lab10-starter.zip

Tasks

For this lab, work on the tasks in the **given order**. We strongly encourage collaboration in labs. Work with your peers, and ask your TAs questions!

As seen in Lectures 25 and 26, one way of implementing a hash table is with chaining: storing multiple elements in each array index. We've provided starter code for a hash map that uses chaining with linked list buckets.

Let's run through a few of the "interesting" bits of the code:

`chaining.h`:

```
struct Entry {
  int k;
  int v;
```

```
  Entry* next;
};
```

In lecture, we presented hash tables as a way of implementing the **set** ADT. To implement the **map** ADT, we need to store both the key and the value. The `next` pointer makes the `Entry` struct into a linked list.

```
Entry** buckets
```

This is similar to how we stored data in a vector. Since we're storing an array of linked lists, and linked lists use pointers, we need to store `Entry*`.

`chaining.cpp`:
We won't cover `insert` in detail – it doesn't diverge from what we covered in lecture (inserting at the back of the correct linked list if the key isn't already present). Ask us your questions about what's going on in there!

```
int ChainingHashMap::get_bucket(int key) {
  return ((key % this->capacity) + this->capacity) % this->capacity;
}
```

Why did we write the "bucket index" function this way instead of just using `key % capacity`? In C++ (and in most languages), the `%` operator is the *remainder* – it keeps the sign of the dividend. This means that if we try to store negative keys, `-1 % capacity` will be negative, which isn't a valid array index!

The `get_bucket` function computes the *modulus* instead, which is non-negative (i.e. `get_bucket(-1)` should be index `capacity - 1` (Quick check – why does that bucket index make sense?)).

Destructor

`ChainingHashTable::~ChainingHashTable()`

Implement the destructor, which cleans up all allocated memory for the current `ChainingHashTable`. Remember that each bucket is a linked list!

Test your implementation with `make test_no_rehash`. These tests should not leak any memory.

Rehashing

`ChainingHashTable::maybe_rehash()`

Implement the rehasher. This function checks if a rehash is necessary (with the current `size`, not `size+1`) by checking that the **load factor** is at least 1. Only if the current load factor is at least one, then double the capacity of the hash table and place each `Entry` appropriately.

Do not allocate any `new Entry` inside this function. When rehashing, you can put elements at either the front or back of the correct bucket's linked list.

Remember to set all data in the new array to an initial value to avoid wild pointers.

Test your implementation with `make test_all`. The previous tests should still pass, and there should be no memory leaks.

Deliverables

- A completely passing test suite with no memory leaks.