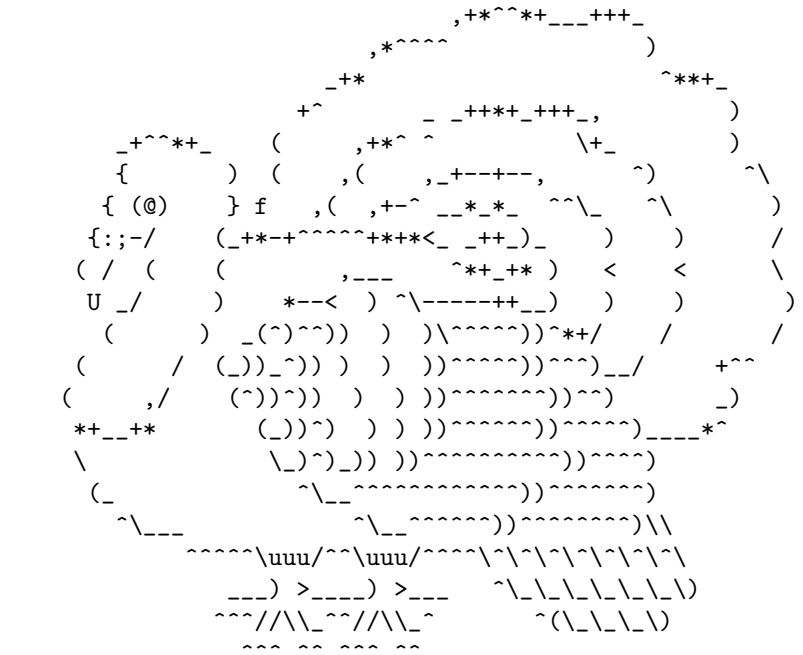
CS15: Hash Tables



From: https://asciiart.website/index.php?art=holiday/thanksgiving

Introduction

Hash tables are often a very fast way to store and retrieve data. The basic idea of hashing is to use an array of buckets — places to put information — and insert values based on an integer value computed from a key values. The

key to efficient hashing (pun!) is a good function to tell you which bucket to use for a given key. For this lab, we will explore a bad and a good hash function, and we'll also implement two strategies to handle when a bucket is already in use (collision).

Review

Collisions

Recall that the input value to a hash function is a key to hash, and the output of the hash function (after compression) is an index of your array. What happens when something is already present at that index? This is known as a **collision**. We will explore two methods of collision resolution in this lab — **linear probing** and **chaining**.

Linear Probing

For linear probing, you simply search for the next open index. So, if your hash function outputs the index 3, and something is occupying 3, then you search at index 4, then 5, and so on. Make sure to wrap around if you go over the end of the table! In other words:

```
indexForKey(x) = (hashFunction(x) + attempt) % table_size
```

where we start with attempt 0, and keep going until we find an empty slot.

Chaining

Handle collisions by storing a linked list in each slot of the array. If there is a collision, simply add the element to the back of the linked list at that slot.

The Lab

Introduction

Hector Hash and his friends are hosting a Thanksgiving dinner. Hector wants to remember the delicious dishes served at the feast by storing them in a hash table. He wants to be able to quickly retrieve information associated with each dish for future gatherings. In particular, he wants to associate each dish name with its serving size and preparation time in minutes (so they know how much time to set aside next year). While discussing hash

functions with his friends, Hector and the group come up with two hash function ideas:

1. Use the length of the dish name as the hash value.
2. Use the `C++ std::hash` facility.

The friends also disagree on which way of handling collisions is best. They consider probing and chaining as options. Hector has started writing two classes that implements both ideas for the hash function, as well as both collision resolution techniques, but he needs your help to finish the job!

Representing Dishes

For the table, Hector is using `C++ string`s to represent the dish name (the keys), and the following `struct` (defined in `DishInfo.h`) to represent the values:

```
struct DishInfo {
	int servingSize;
	int prepTime;

	/* Constructors for struct */
	DishInfo()
	{
		servingSize = prepTime = -1;
	}

	DishInfo(int servings, int minutes)
	{
		servingSize = servings;
		prepTime = minutes;
	}
};
```

Hash Table Classes

Hector is writing two hash table classes, `TurkeyChainingTable` and `TurkeyLinearTable`, which maintain a hash table that handles collisions with chaining and linear probing, respectively.

Implementation Details

You'll notice a few details in the `TurkeyChainingTable` and `TurkeyLinearTable` classes. For instance,

```
typedef std::string KeyType;
typedef DishInfo ValueType;
```

These type definitions allow us to abstract the hash table classes over the details of the key and value type. In the implementation, the only code that depends on knowing the types is the print function, which could be removed after debugging. Then we could make this a template! Similarly, defined in `HashFunction.h`

```
enum HashFunction { BAD_HASH_FUNCTION, GOOD_HASH_FUNCTION };
```

This allows us to use variables of type `HashFunction`, whose values are either `GOOD_HASH_FUNCTION` or `BAD_HASH_FUNCTION`. This is simply a convenience to make our code more readable.

TurkeyChainingTable

Hector is using `chainedTable`, a `std::vector`, to implement chaining:

```
    std::vector<std::list<ChainNode>> chainedTable;
```

Each index of the of the `vector` holds a `std::list` of `ChainNodes`. Declaring a `TurkeyChainingTable` instance will create, resize, and initialize `chainedTable` to empty `lists`.

Index	Chain (std::list of ChainNodes)
0	[ ]
1	[ ]
2	[ ]
3	[ ]
4	[ ]

After a few calls to `insertChaining`, `chainedTable` might look something like this:

Index	Chain (std::list of ChainNodes)
0	“Dish A”: {5, 128}
1	[ ]
2	[ ]
3	“Dish B”: {30, 117} → “Dish C”: {21, 126}
4	[ ]

TurkeyLinearTable

Hector is using `linearTable`, a `std::vector`, to implement linear probing:

```
std::vector<TableEntry> linearTable;
```

Each index of the of the `vector` holds a `TableEntry`. Declaring a `TurkeyLinearTable` instance will create, resize, and initialize `linearTable` to empty `TableEntrys`.

Index	0	1	2	3	4
Bucket	`Empty`	`Empty`	`Empty`	`Empty`	`Empty`

After a few calls to `insertProbing`, `linearTable` might look something like this:

Index	0	1	2	3	4
Bucket	`Empty`	`"Dish A": {5, 128}`	`"Dish C": {30, 117}`	`Empty`	`"Dish B": {21, 126}`

Functions to Write

Your job will be to write the functions described below (along with any helper functions you'd like to write). The functions you will write for this lab are:

```
TurkeyLinearTable::insertProbing(KeyType key, ValueType value,
HashFunction hashFunction);
```

Adds a dish and its information to the hash table using using either the `GOOD_HASH_FUNCTION` or the `BAD_HASH_FUNCTION`. This function handles collisions by using the linear probing method.

```
TurkeyChainingTable::insertChaining(KeyType key, ValueType value,
HashFunction hashFunction);
```

Adds a dish and its information to the hash table using either the `GOOD_HASH_FUNCTION` or the `BAD_HASH_FUNCTION`. This function handles collisions by using the chaining method.
Finally:

```
TurkeyLinearTable::expand();
TurkeyChainingTable::expand();
```

Expand both tables to accommodate more values. You should expand when the load factor, which is the number or items in the table divided by the table capacity, exceeds something like `0.7`.

Other Notes

Make a directory for this lab and get the files from the usual place. You can use the `make` command to compile and link the files. The executable generated is named `turkey`.

You will need to submit the following files:

```
main.cpp
TurkeyChainingTable.cpp
TurkeyChainingTable.h
TurkeyLinearTable.cpp
TurkeyLinearTable.h
Makefile
README
```

You must submit them using Gradescope to the assignment `lab_hash_tables`. Submit these files directly, do not try to submit them in a folder.

Well done!