ECE250 Project 2 Design Document- Hash Tables

Devon Miller-Junk 20763552 October 11, 2019

Project 2: Hash Tables

For the design of project 2, my plan was to create two different implementations for a hash table which contain the same functions and operations, but different methods to handle collisions. The goal was to design both versions to have constant average run times for the insertion, deletion, and search operations.

**Note:** To compile and run my program please use the following (replacing openhttest with unorderedhttest for the chaining implementation):

g++ -std=c++11 -o openhttest openhttest.cpp

./ openhttest <inputFile >outputFile

1. **Open Addressing using Linear Probing Implementation**

The open addressing implementation of the hash table uses an array of nodes that contain the values in the hash table. The strategy to avoid collisions is to check consecutive nodes until you find an empty node, where the value can be inserted.

* 1. **Node Struct**

A node is the basic unit used to hold the values in the hash table. Each node contains its value (an integer) and a state, which is an enum. The state can be initialized, deleted, or uninitialized. The state allows for a deleted node to be differentiated from that of an uninitialized node, to help with operations.

The node struct contains a destructor and 3 constructors, an empty constructor, a copy constructor, and a value constructor. The equality and equals operators were overloaded to provide easy checking of equality and easy assigning of value. The isInitialized and IsUninitialized functions return a check for if the state matches a certain condition. The deletion function sets the state of a node to deleted, and the getValue and getState functions return the state and value of a node. All these operations take constant time.

* 1. **HashTableLinProb Class**

The hash table class implements an array of nodes called table. It also has a size of the table and a hash function that takes a value and maps the value to an index in the table. Since each index in the table can only contain one node, then if the index is full, the hash table class will continue to look for an empty spot to insert the new value into.

There is one destructor and three constructors for the hash table, the default constructor creates a hash table of size 10, the constructor that takes an integer as a parameter creates a hash table of that integer size, there is also a copy constructor. Every constructor takes linear times to create all the nodes.

The insert, search and delete functions are the core three functions of the hash table, and they all take an integer value as a parameter. The insert function finds the first location that the value should go in by using the hash function. It then sequentially searches the hash table for an empty space to insert the value into the hash table. If the insertion function finds an empty space, it returns the value true. The insertion function does not need to check all values if the table is full because a numElements value is used to store the number of elements in the array. If the numElements equals the size of the array, then nothing can be inserted. The search function finds the location that the value should be using the hash function, then sequentially check each next node until it finds either the node it is looking for or a non-deleted, non-empty node. If it does not find the value before finding the non-deleted, non-empty node then the value is not in the hash table. The search function returns the index of the value or -1 if nothing was found. The delete function works the same as the search function, except that it deletes that value once it has found it, and it returns true/false depending on if the value was successfully deleted. If we assume uniform hashing, then any one location can only have one value mapped to it, and thus there can not be any collisions. Therefore, when inserting/searching/deleting, then only a single index needs to be checked, meaning that if we assume uniform hashing these operations take constant time.

There is also a print function used for developmental purposes.

1. **Separate Chaining Implementation**

The chaining implementation of the hash table uses an array of linked lists to store the values in the hash table. The strategy to avoid collisions is to always append to a linked list, which is effectively storing multiple values at a single index of the table.

* 1. **Doubly Linked List Class**

The doubly linked list class stores a pointer to the starting node, a pointer to the back node and a size of the linked list. Each node then contains a pointer to the node in front of it and a pointer to the node behind it as well as the value of the node.

The Linked list class contains a default constructor, a constructor that transforms an array of integers into a linked list, and a destructor. All these functions except for the default constructor (which takes constant time) take linear time.

The doubly linked list has a push function, a pop function, and a remove function. The push function appends to the linked list, the pop function removes from the end of the linked list, and the remove function takes a pointer to a node as a parameter and removes that node from the linked list. These all take constant time.

The getStart, getBack, and getSize functions all just return their corresponding values and take constant time. The print function takes linear time and is used for developmental purposes.

* 1. **Hash Table Class**

The hash table class implements a hash table that uses chaining for collision handling. The hash table contains an array of linked lists, the size of the array, and a hash function that takes an integer value and returns an integer value between 0 and the size of the linked list.

This hash table class also has 2 constructors and a destructor. The default destructor makes a hash table of size 10, the constructor that takes an integer as a parameter makes a hash table with the size passed in. The destructor and both constructors take linear time.

The insertion, deletion, and search functions are the core three functions of this hash table as well. The insertion function takes an integer value and returns true if a value was successfully inserted. The insertion function works by finding the correct index in the table to go to, and then linearly searches the linked list for the value. If the value is not found, then insert the value. The search function takes a value as a parameter, finds the index of the correct linked list to use using the hash function, then linearly searches the linked list for the value. It returns the index of the linked list if it found the value. The delete function searches for the value, and if the value is found, removes it from the linked list and returns true. These functions all take linear time based on the length of the linked list. To determine the true time these algorithms, take, we must figure out how long each linked list will be. If we assume uniform hashing, which causes no collisions, then each linked list would be of size 1, resulting in a constant average time for these functions.

The print function is used for development and just prints the linked lists. It takes linear time.

1. **Utility Function**

I used a utility file to hold a few functions that are useful for string parsing and input handling. These function just modify the strings received from the input file.

1. **References**

Tahvildari, L. (2019, Fall). *ECE 250 Lectures*. *ECE 250 Lectures*. Waterloo.

Ward, P. (2018, Fall). *ECE 150 Lectures*. *ECE 150 Lectures*. Waterloo.