Hashing (I): Introduction

CSCD 300 – Data Structures

Eastern Washington University

EASTERN WASHINGTON UNIVERSITY
start something big

The map of the next few lectures on hashing

- We will start with introducing the concept of the map data structure and its common operations.
- In order to obtain efficient operations for the *map* data structure, we introduce the hashing technique, including hash table and hash function.
- After a basic understanding of the hashing technique, we will learn different techniques such as hashing with chain and open addressing for solving the collision issue that happens in hashing. For each technique, we will introduce several practical techniques for designing good hash functions.
- Finally, we will go over a few practical methods that can convert a non-integer data type to an integer data type, which is what we need for feeding a hash function.

EASTERN WASHINGTON UNIVERSITY
start something big

Outline

The conceptual map data structure

Map

A map is a conceptual data structure to be used as a container to store a set of elements, where each element <k,v> has a unique key k associated with a value v. A map can be implemented by using different physical data structures.

An example

The whole Eastern's database of students profiles can be viewed as a map, where each student's profile is an element whose key is the student's EWUID and whose associated value is the attributes of the student such as name, birthday, gender, etc..

The common operations of a map [1]

- `size()`: return the number of elements in the map.
- `get(k)`: return the element whose key is k, if it exists; otherwise, return `null`.
- `put(k,v)`: if an element with key k exists, update its value to be v and return `null`; otherwise, insert a new element $\langle k, v \rangle$ and return v.
- `remove(k)`: remove the element with key k and return its value, if it exists; otherwise, return `null`.

From now on, we will assume the keys are integer numbers. We will deal with the elements whose keys are not integer numbers at the end of the series of lectures on hashing.

We want to find a way to implement the map data structure, so that the above methods will be time efficient.

[1] You can add and implement any other methods needed by your application.

Inefficient ideas for implementing the map

An example data set: Suppose the elements that we want to save in the map are student profiles. Each profile has the student id which is an integer number as the key and the student's name as the value.

Discussion

Using an array or a linked list to implement a map to contain the above data sets CANNOT make all the methods at page 5 efficient. WHY ?

A better idea: hashing

Suppose we have ten student profiles:

$s_0 = \langle 48567392, Alice \rangle$, $s_1 = \langle 94827503, Bob \rangle$, $s_2 = \langle 50249854, Carol \rangle$,
$s_3 = \langle 48850431, Daniel \rangle$, $s_4 = \langle 49985029, Edward \rangle$, $s_5 = \langle 48092854, Frank \rangle$,
$s_6 = \langle 53962465, George \rangle$, $s_7 = \langle 35765876, Hallen \rangle$, $s_8 = \langle 31843653, Ian \rangle$,
$s_9 = \langle 75986477, Jason \rangle$

Hashing

- We allocate a collection of 10 buckets: $B_0, B_1, \ldots, B_9$ as the physical data structure to host the map.
- Let function $h(x) = x \mod 10$, where x is non-negative integer.
- We save a student profile $\langle id, name \rangle$ into $B_{h(id)}$, assuming each bucket can save more than one element.

	s_3	s_0	s_1, s_8	s_2, s_5	s_6	s_7	s_9		s_4
B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9

(continue ...)

A few terminologies

- The whole table $B[0 \ldots 9]$ is called hash table.
- The function h that we used is called hash function.
- We say that, for example, s_3 is hashed into B_1.
- Two different elements being hashed into the same hash table location is called a hash collision.
- The ratio $\frac{data\ set\ size}{table\ size}$ is called load factor.

Some Observations

- We want to find and use a hash function h, such that all the elements will be hashed into all the hash table locations almost uniformly. In other words, we like hash functions that are less likely to have hash collisions. *It is not easy to find such hash functions and we will discuss more on this topic later.*
- We want the load factor not to be too large, meaning the hash table size cannot be too smaller than the data set size, so that each table location will not have too many elements, assuming a good hash function is given.
- If the table size is $\geq$ data set size and the hash function is quite good, all the operations at page 5 will have constant time cost.

The implementation of the operations

- `size()`: trivial by maintaining a counter as a member field of the map.
- `get(k)`: return the element whose key is k, if such an element exists in $B_{h(k)}$; otherwise, return `null`.
- `put(k,v)`: if an element with key k exists in $B_{h(k)}$, update its value to be v and return `null`; otherwise, insert a new element $\langle k, v \rangle$ into $B_{h(k)}$ and return v.
- `remove(k)`: remove the element with key k from $B_{h(k)}$ and return its value, if it exists; otherwise, return `null`.

EASTERN WASHINGTON UNIVERSITY
start something big

Question

When collisions happen, meaning more than one element are hashed into one table location, how do we save multiple elements in a single table location ? Note that the number of elements in a particular location can vary, depending on the given data set and the picked hash function.

We will continue in the next lecture to solve the above issue.