

Data Structures and Algorithms for Engineers

Module 10: Hashing

Lecture 1: Dictionaries, hashing, hash functions, collision resolution, complexity, applications.

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

Containers and Dictionaries

- **Containers** are data structures that permit storage and retrieval of data items independent of content
- **Dictionaries** are data structures that **retrieve** data based on **key** values (i.e, **content**)

Containers and Dictionaries

Main dictionary operations are

- $\text{Search}(D, k)$ Given a search key k , return a pointer to the element in dictionary D whose key value is k , if one exists
- $\text{Insert}(D, x)$ Given a data item x , add it to the dictionary D
- $\text{Delete}(D, x)$ Given a pointer to a given data item x in the dictionary D , remove it from D

Containers and Dictionaries

Some dictionary data structures also **efficiently** support other useful operations

- $\text{Max}(D)$ Retrieve the item with the largest key from D
- $\text{Min}(D)$ Retrieve the item with the smallest key from D

These operations allows the dictionary to serve as a **priority queue**

Containers and Dictionaries

Some dictionary data structures also **efficiently** support other useful operations

- $\text{Predecessor}(D, x)$ Retrieve the item from D whose key is immediately before x in sorted order
- $\text{Successor}(D, x)$ Retrieve the item from D whose key is immediately after x in sorted order

These operations enable us to iterate through the elements of the data structure

Containers and Dictionaries

- We have defined these container and dictionary operations in an **abstract** manner, without reference to their implementation or the implementation of the structure itself
- There are many implementation options
 - Unsorted arrays
 - Sorted arrays
 - Singly-linked lists
 - Doubly-linked lists
 - Binary search trees
 - Balanced binary search trees
 - Heaps
 - **Hash tables**
 - ...

Hashing

- Hash tables are a **very** practical way to maintain a dictionary
- It takes a constant amount of time to look up an item in an array, **if** you have its index ... $O(1)$
- A hash function is a mathematical function to maps keys to integers
 - This integer is used as an index into an array
 - We store our item at that position

Hashing

	values
[0]	Empty
[1]	4501
[2]	Empty
[3]	7803
[4]	Empty
.	.
.	.
.	.
[97]	Empty
[98]	2298
[99]	3699

HandyParts company makes no more than 100 different parts. But the parts all have four digit numbers.

This hash function can be used to store and retrieve parts in an array.

$\text{Hash}(\text{key}) = \text{partNum} \% 100$

Placing Elements in the Array

	values
[0]	Empty
[1]	4501
[2]	Empty
[3]	7803
[4]	Empty
.	.
.	.
.	.
[97]	Empty
[98]	2298
[99]	3699

Use the hash function

$$\text{Hash}(\text{key}) = \text{partNum} \% 100$$

to place the element with

part number 5502 in the

array.

Placing Elements in the Array

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
.	.
.	.
.	.
[97]	Empty
[98]	2298
[99]	3699

Next place part number
6702 in the array.

Hash(key) = partNum % 100

$$6702 \% 100 = 2$$

But values[2] is already
occupied.

COLLISION OCCURS

How to Resolve the Collision?

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
.	.
.	.
.	.
[97]	Empty
[98]	2298
[99]	3699

One way is by linear/sequential probing.
This uses the rehash function

$$(\text{HashValue} + 1) \% 100$$

repeatedly until an empty location
is found for part number 6702.

Resolving the Collision

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
.	.
.	.
.	.
[97]	Empty
[98]	2298
[99]	3699

Still looking for a place for 6702
using the function

$$(\text{HashValue} + 1) \% 100$$

Collision Resolved

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
.	.
.	.
.	.
[97]	Empty
[98]	2298
[99]	3699

Part 6702 can be placed at the location with index 4.

Collision Resolved

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	6702
.	.
.	.
.	.
[97]	Empty
[98]	2298
[99]	3699

Part 6702 is placed at the location with index 4.

Where would the part with number 4598 be placed using linear probing?

Hashing

- In general, the keys are not so conveniently defined (e.g., part numbers) and they have to be computed
- Typically, they are some alphanumeric string S
- The first step of a hash function is to map each key to a big integer
- Let α be the size of the alphabet in which S is written
- Let $char(c)$ be a function that maps each symbol of the alphabet to a unique integer from 0 to $\alpha - 1$

Hashing

- The hash function

$$H(S) = \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \times \text{char}(s_i)$$

maps each string to a unique (but large) integer by treating the characters of the string as “digits” in a base- α number system

- The result is unique identified numbers, but they are so large they will quickly exceed the number of slots m in the hash table
- We reduce this number to an integer between 0 and $m-1$ by taking the remainder of $H(S) \bmod m$
- If m , the size of the hash table, is selected well, the resulting hash value will be fairly uniformly distributed

Hashing

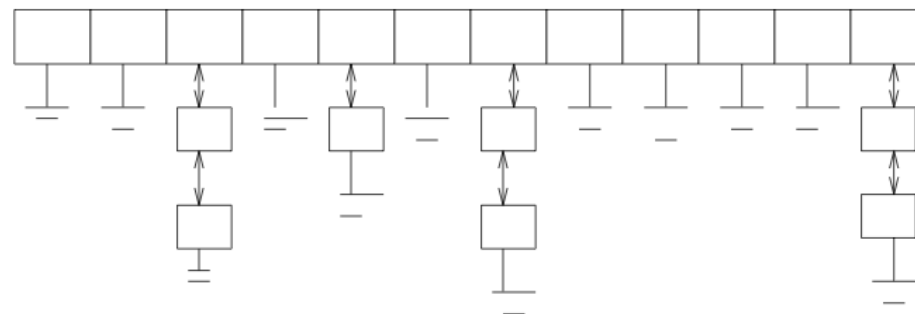
Collisions

- No matter how good the hash function is, there will sometimes be collisions: two keys mapping to the same number/index
- One approach to collision resolution: open addressing
 - On insertion, check to see if the desired position is empty
 - If so, insert it
 - If not, find some other place ...
 - Simplest approach is sequential probing: look for the next open spot in the table

Hashing

Collisions

- No matter how good the hash function is, there will sometimes be collisions: two keys mapping to the same number/index
- Alternative approach: chaining



Hashing

Complexity of operations in a hash table

- Assuming chaining with doubly-linked lists
- m -element hash table
- n keys

	Hash table (expected)	Hash table (worst case)
Search(L, k)	$O(n/m)$	$O(n)$
Insert(L, x)	$O(1)$	$O(1)$
Delete(L, x)	$O(1)$	$O(1)$
Successor(L, x)	$O(n + m)$	$O(n + m)$
Predecessor(L, x)	$O(n + m)$	$O(n + m)$
Minimum(L)	$O(n + m)$	$O(n + m)$
Maximum(L)	$O(n + m)$	$O(n + m)$

Hashing

- A hash table is often the best data structure to maintain a dictionary
- Also useful for other applications
 - Efficient string matching via hashing

Problem: given a text string t and a pattern string p , does t contain the pattern p as a substring, and if so where?

Hashing

- A hash table is often the best data structure to maintain a dictionary
- Also useful for other applications

String matching

Simplest algorithm:

- Overlay p in t at every position in the text
- Check whether every pattern character matches the text character
- Complexity $O(nm)$, where $n = |t|$ and $m = |p|$

Hashing

String matching

Rabin-Karp algorithm:

- Basic idea: if two strings are identical, so are their hash values
- If the two strings are different, the hash values are **almost certainly** different (may need to check, but not often)
- With some clever computation of the hash function (to reduce complexity to constant) the algorithm will usually run in $O(n + m)$

Hashing

Related applications

- Is a given document different from all the rest in a large corpus?
- Is part of this document plagiarized from a document in a large corpus?