Hashing

Section 3.4

Hashing

We have a need… a need for speed!

Hashing

- Array elements can be accessed in O(1)
- Why?
  - the memory address of any element can be calculated mathematically
  - however, this doesn't work for dictionary keys

Hashing

- We can use a nice balanced tree to store the data
- … but that is O(log n) – which is still excellent
- Is it possible to get the time complexity down to O(1)?

Hashing

- What if we come up with a "magic function"?
- So….
  - given a specific key the function would compute the *exact* index of the element
  - this would given dictionaries O(1) access

Hashing

- This function is called a *hash function*
- It takes a key object as an argument and returns a numeric value
- We use this value to store data into a parent array
- Since the value is unique...
  - access is O(1)
  - at least ideally – that is what we want....

Hash Mathematics

- Use hash function to map keys into positions in a hash table
- With element e has key k and h is hash function
  - then e is stored in position $h(k)$ of table
  - To search for e, compute $h(k)$ to locate position
  - If no element, dictionary does not contain e

Using the Function

- Put key/value pairs into the table
- Key is used to find the index
- Value holds the information about the object

Index	Key	Data
0	cat	cat info
1		
2	chicken	chicken info
3	dog	dog info
4		
5		

Example Hash Function

- `hash("Rick") = 5`
- `hash("Morty") = 1`
- `hash("Jerry") = 0`
- `hash("Beth") = 4`

Array	
0	Jerry
1	Morty
2	
3	
4	Beth
5	Rick

But, There are Problems

- Simple hash functions
  - work for implementing dictionaries
  - but most applications have key ranges that are too large for 1-1 mapping between hashes and keys
- Example:
  - key range from 0 to 65,535
  - collection will have no more than 100 items at any given time
  - impractical to use a hash table with 65,536 slots!

Finding the Hash Function

- There is **no** magic function
  - **only** in rare cases, with a limited key range, a perfect function can exist
  - however, for real World cases, there is no function possible
- So, we can take a different approach
  - don't use the hash value as a finishing point
  - use it as a location to start looking

Example Hash Function

- `hash("Isaac") = 5`
- `hash("Mercer") = 1`
- `hash("Bortus") = 0`
- `hash("Yaphet") = 4`
- `hash("Grayson") = 1`

Array	
0	Bortus
1	COLLISION
2	
3	
4	Yaphet
5	Isaac

Collisions

- When two keys hash to the same array location, this is called a *collision*
- What do we do?
  - normally collisions are "first come, first serve"
  - the first key that hashes to the location gets it
  - so, we need to decide what do with the second item that hash to the same location
  - there are <u>two</u> solutions

Closed Hashing

Chaos... good news

Collision Solution: Closed Hashing

- With *closed hashing*, we use the existing array and search for a empty position
- Use the hash value as *start position* – we start searching here

Collision Solution: Closed Hashing

- If the array element is a occupied...search down and look for an empty element
- The search <u>must</u> also…
  - wrap-around to the top
  - and be aware if the search cycles through the entire array – *we ran out of space*

Closed Hash Example

- Adding "Pacman" with a hash value of 0
- This collides with has used by "Dig Dug"
- We search down for the next empty cell

0	Dig Dug
1	Q-Bert
2	
3	
4	Fix-It Felix, Jr
5	Frogger

Closed Hash Example

- Adding "Pacman" with a hash value of 0
- This collides with has used by "Dig Dug"
- We search down for the next empty cell

0	Dig Dug
1	Q-Bert
2	
3	
4	Fix-It Felix, Jr
5	Frogger

Closed Hash Example

- Adding "Pacman" with a hash value of 0
- This collides with has used by "Dig Dug"
- We search down for the next empty cell

0	Dig Dug
1	Q-Bert
2	**Pacman**
3	
4	Fix-It Felix, Jr
5	Frogger

Closed Hashing Clustering

- One problem with the closed hashing is the tendency to form *clusters*
- A *cluster* is a group of continuous used cells – with no open slots
- What happens?
  - the bigger a cluster gets, the more likely it is that new keys will hash into it
  - it then grows larger and larger
  - clusters will eventually degrade the hash to O(n)

Efficiency of Closed Hashing

- Hash tables are surprisingly efficient
- Although collisions cause searching, tables, items can found near O(1)
- Even if the table is nearly full (leading to long searches), efficiency is still quite high

Closed Hashing Pitfalls

- Closed hashing is not the best solution
- It requires a static array
  - the array cannot be increased at runtime (or the hash fails)
  - as a result, the array can fill up
- Clustering causes O(n) degradation

Closed Hashing Pitfalls

- You cannot delete items
  - it creates empty slots in clusters!
  - this can prevent an item, *added in a cluster*, from being found below the gap
  - there are work-rounds, but it gets **convoluted**

Closed Delete Problem

- "Dig Dug" and "Pacman" have the same hash value of 0
- When "Pacman" was added, "Dig Dug" caused "Pacman" to be stored at 2

0	Dig Dug
1	Q-Bert
2	Pacman
3	
4	Fix-It Felix, Jr
5	Frogger

Closed Delete Problem

- If "Dig Dug" is deleted, the array element is empty
- If there is a search for "Pacman", it will hash to 0 and it won't be found

0	
1	Q-Bert
2	Pacman
3	
4	Fix-It Felix, Jr
5	Frogger

Open Hashing

The Merging of Concepts

Open Hashing

- With *open hashing*, we don't store individual objects in each array cell
- Instead, each array element is a linked list or tree

Open Hashing

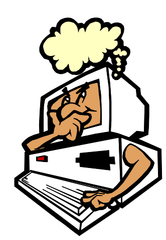
- So, our hash table is an array of either linked lists or trees
- This approach is also known as *bucket hashing* since each list/tree acts a "hash bucket"

Collision Solution: Open Hashing

- When a collision occurs the item is added to the list/tree
- So, this list/tree will contain **all** the objects with the same hash value
- We don't need to look for conflicts

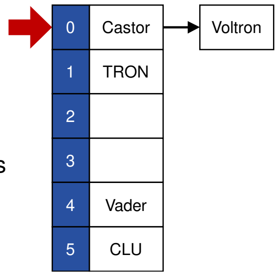
When an Object is Looked Up…

- Compute the hash value
- And then search the targeted list/tree
- So, the time complexity is minimalized



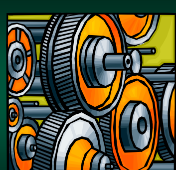
Open Hash Example

- Adding "Voltron" with a hash of 0
- This collides with "Castor"
- Open hashing just adds the item to the linked list



Open Hashing Benefits

- Open hashing will not fill up the array and can grow *indefinitely*
- Far faster access time than closed hashing
- No clustering
- Objects can be deleted

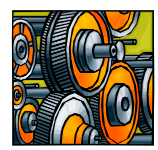


Hash Functions

Techniques for Spreading the Data

Hash Functions

- A hash function can be *anything*
- However, it is best to...
  - find one that spreads items evenly over the array
  - ... and one that limits collisions



Random Hashing

- Most hashing algorithms use a pseudo-random number generator
- This essentially scatters the items “randomly” throughout the hash table
- ... but there is no real "random" numbers in computers – only chaotic series

Popular Algorithm: Division

- Uses the formula: $h(k) = k \bmod N$
  - k is a raw key value produced by some internal function
  - for all purpose, we don't care "how" this was produced
  - N is the size of the array
- Selecting N
  - table size N is usually chosen as a prime number
  - it prevents patterns – which can cause collisions

Popular Algorithm: MAD

- Based on multiply, add, and divide (MAD)
- Uses the formula: $h(k) = (a * k + b) \bmod N$
  - a and b are both constants
  - eliminates patterns provided $a \bmod N \neq 0$
  - this is the same formula used to create (pseudo) random number generators