

Week 8 - Wednesday

COMP 2100

Last time

- What did we talk about last time?
- AVL trees
- Balancing trees by construction
- Hash tables

Questions?

Project 3

Assignment 4

Hash Tables

Example

- Determine if a string has any duplicate characters
- Weak!
- Okay, but do it in $O(m)$ time where m is the length of the string

Hash Functions

What are we looking for?

- We want a function that will map data to buckets in our hash table
- Important characteristics:
 - **Efficient:** It must be quick to execute
 - **Deterministic:** The same data must always map to the same bucket
 - **Uniform:** Data should be mapped evenly across all buckets

Division

- We want a function $h(k)$ that computes a hash for every key k
- The simplest way of guaranteeing that we hash only into legal locations is by setting $h(k)$ to be:
- $h(k) = k \bmod N$ where N is the size of the hash table
- To avoid crowding the low indexes, N should be prime
- If it is not feasible for N to be prime, we can add another step using a prime $p > N$:
- $h(k) = (k \bmod p) \bmod N$

Division Pros and Cons

- Pros
 - Simple
 - Fast
 - Easy to do
 - Good if you know nothing about the data
- Cons
 - Prime numbers are involved (What's the nearest prime to the size you want?)
 - Uses no information about the data
 - If the data is strangely structured (multiples of p , for example) it could all hash to the same location

Folding

- Break the key into parts and combine those parts
- **Shift folding** puts the parts together without transformations
 - SSN: 123-45-6789 is broken up and summed $123 + 456 + 789 = 1,368$, then modded by N , probably
- **Boundary folding** puts the parts together reversing every other part of the key
 - SSN: 123-45-6789 is broken up and summed $123 + 654 + 789 = 1,566$, then modded by N , probably

Folding Pros and Cons

- Pros
 - Relatively Simple and Fast
 - Mixes up the data more than division
 - Points out a way to turn strings or other non-integer data into an integer that can be hashed
 - Transforms the numbers so that patterns in the data are likely to be removed
- Cons
 - Primes are still involved
 - Uses no special information about the data

Mid-Square Function

- Square the key, then take the "middle" numbers out of the result
- Example: key = 3,121 then $3,121^2 = 9,740,641$ and the hash value is 406
- One nice thing about this method is that we can make the table size be a power of 2
- Then, we can take the $\log_2 N$ middle bits out of the squared value using bitwise shifts and masking

Mid-Square Pros and Cons

- Pros
 - Randomizes the data a lot
 - Fast when implemented correctly
 - Primes are not necessary
- Cons
 - Uses no special information about the data

Extraction

- Remove part of the key, especially if it is useless
- Example:
 - Many SSN numbers for Indianapolis residents begin with 313
 - Removing the first 3 digits will, therefore, not reduce the randomness very much, provided that you are looking at a list of SSNs for Indianapolis residents

Extraction Pros and Cons

- Pros

- Uses information about the key
- Can be efficient and easy to implement

- Cons

- Requires special knowledge
- Careless extraction of digits can give poor hashing performance

Radix Transformation

- Change the number to a different base
- Then, treat the base as if it were still base 10 and use the division method
- Example: 345 is 423 in base 9
- If $N = 100$, we could take the mod and put 345 in location 23

Radix Transformation Pros and Cons

- Pros

- If many numbers have similar final digits or values mod N (or p), they can be randomized by this method

- Cons

- Choice of base can be difficult
- Effects are unpredictable
- Not as quick as many of the other methods
- Values that didn't collide before might now collide

Collisions

The real problem with hash tables

- What happens when you go to put a value in a bucket and one is already there?
- There are a couple basic strategies:
 - Open addressing
 - Chaining
- **Load factor** is the number of items divided by the number of buckets
 - 0 is an empty hash table
 - 0.5 is a half full hash table
 - 1 is a completely full hash table

Open addressing

- With open addressing, we look for some empty spot in the hash table to put the item
- There are a few common strategies
 - Linear probing
 - Quadratic probing
 - Double hashing

Linear probing

- With linear probing, you add a step size until you reach an empty location or visit the entire hash table
- Let $h(k)$ be the initial hash function
- $h(k,i) = h(k) + ci$, for $i = 0, 1, 2, 3, \dots$

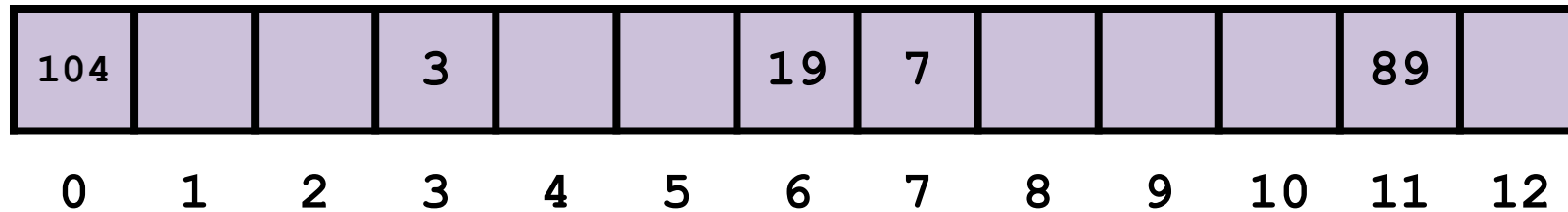
104			3			19	7				89	
0	1	2	3	4	5	6	7	8	9	10	11	12

- Example: Add 6 with a step size of 5

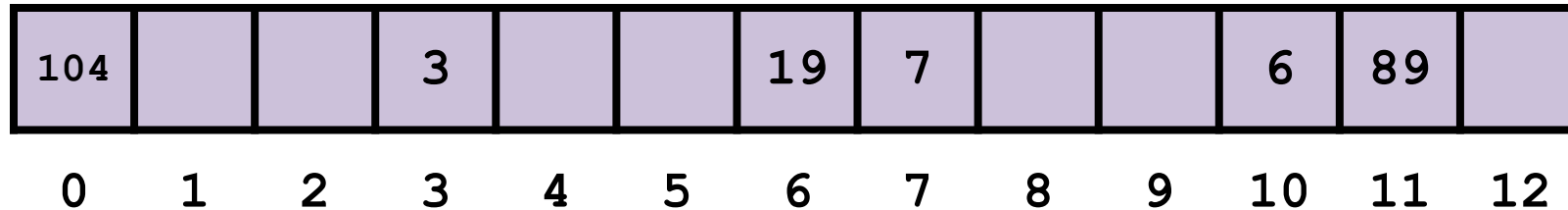
104			3			19	7	6			89	
0	1	2	3	4	5	6	7	8	9	10	11	12

Quadratic probing

- For quadratic probing, use a quadratic function to try new locations:
- $h(k,i) = h(k) + c_1i + c_2i^2$, for $i = 0, 1, 2, 3\dots$



- Example: Add 6 with $c_1 = 0$ and $c_2 = 1$



Double hashing

- For double hashing, do linear probing, but with a step size dependent on the data:
- $h(k,i) = h_1(k) + i \cdot h_2(k)$, for $i = 0, 1, 2, 3, \dots$

104			3			19	7				89	
0	1	2	3	4	5	6	7	8	9	10	11	12

- Example: Add 6 with $h_2(k) = (k \bmod 7) + 1$

104	6		3			19	7				89	
0	1	2	3	4	5	6	7	8	9	10	11	12

Open addressing pros and cons

- Open addressing schemes are fast and relatively simple
- Linear and quadratic probing can have clustering problems
 - One collision means more are likely to happen
- Double hashing has poor data locality
- It is impossible to have more items than there are buckets
- Performance degrades seriously with load factors over 0.7

Chaining

- Make each hash table entry a linked list
- If you want to insert something at a location, simply insert it into the linked list
- This is the most common kind of hash table
- Chaining can behave well even if the load factor is greater than 1
- Chaining is sensitive to bad hash functions
 - No advantage if every item is hashed to the same location

Deletion

- Deletion can be a huge problem
- Easy for chaining
- Highly non-trivial for open addressing
- Consider our linear probing example with a step size of 5

104			3			19	7	6			89	
0	1	2	3	4	5	6	7	8	9	10	11	12

- Delete 19
- Now see if 6 exists

Perfect Hash Functions

- If you know all the values you are going to see ahead of time, it is possible to create a minimal perfect hash function
- A minimal perfect hash function will hash every value without collisions and fill your hash table
- Cichelli's method and the FHCD algorithm are two ways to do it
- Both are complex
- Look them up if you find yourself in this situation

Hash Table Implementation

Recall: Symbol table ADT

- We can define a symbol table ADT with a few essential operations:
 - `put(Key key, Value value)`
 - Put the key-value pair into the table
 - `get(Key key):`
 - Retrieve the value associated with key
 - `delete(Key key)`
 - Remove the value associated with key
 - `contains(Key key)`
 - See if the table contains a key
 - `isEmpty()`
 - `size()`
- It's also useful to be able to iterate over all keys

Chaining hash table

```
public class HashTable {
    private int size = 0;
    private int power = 10;
    private Node[] table = new Node[1 << power];

    private static class Node {
        public int key;
        public Object value;
        public Node next;
    }
    ...
}
```


Easy methods

- Get the number of elements stored in the hash table

```
public int size()
```

- Say whether or not the hash table is empty

```
public boolean isEmpty()
```

Hashing function

- It's useful to have a function that finds the appropriate hash value
- Take the input integer and swap the low order 16 bits and the high order 16 bits (in case the number is small)
- Square the number
- Use shifting to get the middle **power** bits

```
private int hash(int key)
```

Upcoming

Next time...

- Finish implementing hash tables
- **Map** in the JCF
 - **HashMap**
 - **TreeMap**
- Introduction to graphs

Reminders

- Start Project 3
 - Form teams!
- Start Assignment 4
- Keep reading 3.4
- Read 4.1