Hash Open Indexing

Data Structures and Algorithms

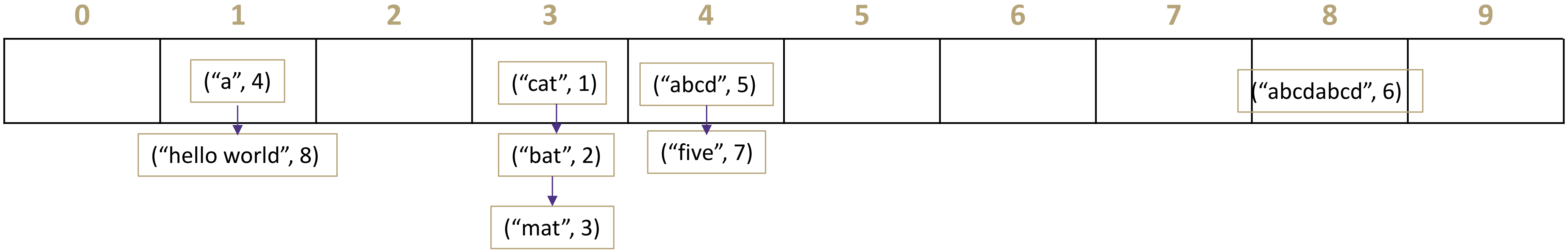
Warm Up

Consider a StringDictionary using separate chaining with an internal capacity of 10. Assume our buckets are implemented using a LinkedList. Use the following hash function:

```
public int hashCode(String input) {
   return input.length() % arr.length;
}
```

Now, insert the following key-value pairs. What does the dictionary internally look like?

("cat", 1) ("bat", 2) ("mat", 3) ("a", 4) ("abcd", 5) ("abcdabcd", 6) ("five", 7) ("hello world", 8)



0 1 2 3 4 5 6 7 8 9

("a", 4)

("hello world", 8)

("cat", 1)

("bat", 2)

("mat", 3)

("abcd", 5)

("five", 7)

("abcdabcd", 6)

Administrivia

HW 2 due

HW 3 out

Midterm Topics

ADTs and Data structures

- Lists, Stacks, Queues, Maps
- Array vs Node implementations of each

Asymptotic Analysis

- Proving Big O by finding C and N_0
- Modeling code runtime with math functions, including recurrences and summations
- Finding closed form of recurrences using unrolling, tree method and master theorem
- Looking at code models and giving Big O runtimes
- Definitions of Big O, Big Omega, Big Theta

BST and AVL Trees

- Binary Search Property, Balance Property
- Insertions, Retrievals
- AVL rotations

Hashing

- Understanding hash functions
- Insertions and retrievals from a table
- Collision resolution strategies: chaining, linear probing, quadratic probing, double hashing

Heaps

- Heap properties
- Insertions, retrievals while maintaining structure with bubbling up

Homework

- ArrayDictionary
- DoubleLinkedList
- ChainedHashDictionary
- ChainedHashSet

Can we do better?

Idea 1: Take in better keys

- Can't do anything about that right now

Idea 2: Optimize the bucket

- Use an AVL tree instead of a Linked List
- Java starts off as a linked list then converts to AVL tree when collisions get large

Idea 3: Modify the array's internal capacity

- When load factor gets too high, resize array
  - Double size of array
  - Increase array size to next prime number that's roughly double the array size
    - Prime numbers reduce collisions when using % because of divisors
  - Resize when $\lambda \approx 1.0$
  - When you resize, you have to rehash

What about non integer keys?

Hash Function

An algorithm that maps a given key to an integer representing the index in the array for where to store the associated value

Goals

Avoid collisions

- The more collisions, the further we move away from O(1)
- Produce a wide range of indices

Uniform distribution of outputs

- Optimize for memory usage

Low computational costs

- Hash function is called every time we want to interact with the data

How to Hash non Integer Keys

Implementation 1: Simple aspect of values

```
public int hashCode(String input) {
    return input.length();
}
```

Pro: super fast O(1)
Con: lots of collisions!

Implementation 2: More aspects of value

```
public int hashCode(String input) {
    int output = 0;
    for(char c : input) {
        out += (int)c;
    }
    return output;
}
```

Pro: fast O(n)
Con: some collisions

Implementation 3: Multiple aspects of value + math!

```
public int hashCode(String input) {
    int output = 1;
    for (char c : input) {
        int nextPrime = getNextPrime();
        out *= Math.pow(nextPrime, (int)c);
    }
    return Math.pow(nextPrime, input.length());
}
```

Pro: few collisions
Con: slow, gigantic integers

3 Minutes

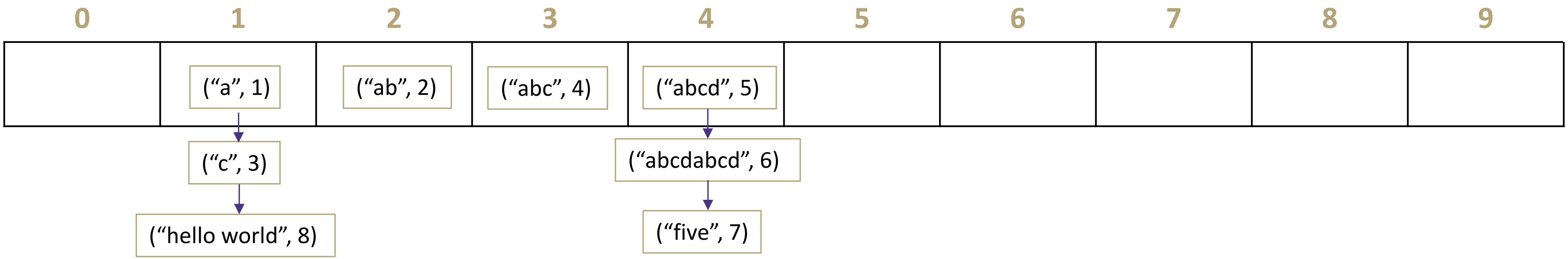
Practice

Consider a StringDictionary using separate chaining with an internal capacity of 10. Assume our buckets are implemented using a LinkedList. Use the following hash function:

```
public int hashCode(String input) {
   return input.length() % arr.length;
}
```

Now, insert the following key-value pairs. What does the dictionary internally look like?

(“a”, 1) (“ab”, 2) (“c”, 3) (“abc”, 4) (“abcd”, 5) (“abcdabcd”, 6) (“five”, 7) (“hello world”, 8)



0

1: (“a”, 1) → (“c”, 3) → (“hello world”, 8)

2: (“ab”, 2)

3: (“abc”, 4)

4: (“abcd”, 5) → (“abcdabcd”, 6) → (“five”, 7)

5

6

7

8

9

Review: Handling Collisions

Solution 1: Chaining

Each space holds a "bucket" that can store multiple values. Bucket is often implemented with a LinkedList

Operation		Array w/ indices as keys
put(key,value)	best	O(1)
	average	O(1 + λ)
	worst	O(n)
get(key)	best	O(1)
	average	O(1 + λ)
	worst	O(n)
remove(key)	best	O(1)
	average	O(1 + λ)
	worst	O(n)

Average Case:
Depends on average number of elements per chain

Load Factor λ

If n is the total number of key-value pairs

Let c be the capacity of array

Load Factor λ = $\frac{n}{c}$

Handling Collisions

Solution 2: Open Addressing

Resolves collisions by choosing a different location to tore a value if natural choice is already full.

Type 1: Linear Probing

If there is a collision, keep checking the next element until we find an open spot.

```
public int hashFunction(String s)
    int naturalHash = this.getHash(s);
    if(natural hash in use) {
        int i = 1;
        while (index in use) {
            try (naturalHash + i);
            i++;
```

Linear Probing

Insert the following values into the Hash Table using a hashFunction of % table size and linear probing to resolve collisions
1, 5, 11, 7, 12, 17, 6, 25

0	1	2	3	4	5	6	7	8	9
	1 11	12			25	6	7 17		

Linear Probing

3 Minutes

Insert the following values into the Hash Table using a hashFunction of % table size and linear probing to resolve collisions
38, 19, 8, 109, 10

	0	1	2	3	4	5	6	7	8	9
8	10								38	19 109

Problem:

- Linear probing causes clustering
- Clustering causes more looping when probing

Primary Clustering

When probing causes long chains of occupied slots within a hash table

2 Minutes

Runtime

When is runtime good?

Empty table

When is runtime bad?

Table nearly full

When we hit a “cluster”

Maximum Load Factor?

λ at most 1.0

When do we resize the array?

λ ≈ ½

Can we do better?

Clusters are caused by picking new space near natural index

Solution 2: Open Addressing

Type 2: Quadratic Probing

If we collide instead try the next i^2 space

```
public int hashFunction(String s)
    int naturalHash = this.getHash(s);
    if(natural hash in use) {
        int i = 1;
        while (index in use) {
            try (naturalHash + i * i);
            i++;
```

Quadratic Probing

Insert the following values into the Hash Table using a hashFunction of % table size and quadratic probing to resolve collisions
89, 18, 49, 58, 79

0	1	2	3	4	5	6	7	8	9
		58	79					18	89 49

(49 % 10 + 0 * 0) % 10 = 9
(49 % 10 + 1 * 1) % 10 = 0

(58 % 10 + 0 * 0) % 10 = 8
(58 % 10 + 1 * 1) % 10 = 9
(58 % 10 + 2 * 2) % 10 = 2

(79 % 10 + 0 * 0) % 10 = 9
(79 % 10 + 1 * 1) % 10 = 0
(79 % 10 + 2 * 2) % 10 = 3

Problems:

If λ≥ ½ we might never find an empty spot

- Infinite loop!

Can still get clusters

3 Minutes

Secondary Clustering

Insert the following values into the Hash Table using a hashFunction of % table size and quadratic probing to resolve collisions

19, 39, 29, 9

0	1	2	3	4	5	6	7	8	9
39			29					9	19

Secondary Clustering

When using quadratic probing sometimes need to probe the same sequence of table cells, not necessarily next to one another

Probing

- h(k) = the natural hash
- h'(k, i) = resulting hash after probing
- i = iteration of the probe
- T = table size

Linear Probing:

$h'(k, i) = (h(k) + i) \% T$

Quadratic Probing

$h'(k, i) = (h(k) + i^2) \% T$

For both types there are only O(T) probes available

- Can we do better?

Double Hashing

Probing causes us to check the same indices over and over- can we check different ones instead?

Use a second hash function!

h'(k, i) = (h(k) + i * g(k)) % T

<- Most effective if g(k) returns value prime to table size

```
public int hashFunction(String s)
    int naturalHash = this.getHash(s);
    if(natural hash in use) {
        int i = 1;
        while (index in use) {
            try (naturalHash + i * jump_Hash(key));
            i++;
```

Second Hash Function

Effective if g(k) returns a value that is *relatively prime* to table size

- If T is a power of 2, make g(k) return an odd integer
- If T is a prime, make g(k) return any smaller, non-zero integer
    - g(k) = 1 + (k % T(-1))

How many different probes are there?

- T different starting positions
- T – 1 jump intervals
- O(T^2) different probe sequences
    - Linear and quadratic only offer O(T) sequences

Resizing

How do we resize?

- Remake the table
- Evaluate the hash function over again.
- Re-insert.

When to resize?

- Depending on our load factor λ
- Heuristic:
  - for separate chaining λ between 1 and 3 is a good time to resize.
  - For open addressing λ between 0.5 and 1 is a good time to resize.

Separate chaining: Running Times

What are the running times for:

`insert`

Best: $O(1)$

Worst: $O(n)$ (if insertions are always at the end of the linked list)

`find`

Best: $O(1)$

Worst: $O(n)$

`delete`

Best: $O(1)$
Worst: $O(n)$

Linear probing: Average-case insert

If $\lambda < 1$ we'll find a spot eventually.

What's the average running time?

Uniform Hashing Assumption

for any pair of elements x, y

the probability that h(x) = h(y) is $\frac{1}{TableSize}$

If find is unsuccessful: $\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$

If find is successful: $\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)}\right)$

We won't ask you to prove these

Summary

1. Pick a hash function to:

- Avoid collisions
- Uniformly distribute data
- Reduce hash computational costs

2. Pick a collision strategy

- Chaining
  - LinkedList
  - AVL Tree
- Probing
  - Linear
  - Quadratic
- Double Hashing

No clustering
Potentially more "compact" (λ can be higher)

Managing clustering can be tricky
Less compact (keep λ < ½)
Array lookups tend to be a constant factor faster than traversing pointers

Summary

Separate Chaining

- Easy to implement
- Running times $O(1 + \lambda)$

Open Addressing

- Uses less memory.
- Various schemes:
- Linear Probing – easiest, but need to resize most frequently
- Quadratic Probing – middle ground
- Double Hashing – need a whole new hash function, but low chance of clustering.

Which you use depends on your application and what you’re worried about.

Other applications of hashing

- Cryptographic hash functions: Hash functions with some additional properties
  - Commonly used in practice: SHA-1, SHA-265
  - To verify file integrity. When you share a large file with someone, how do you know that the other person got the exact same file? Just compare hash of the file on both ends. Used by file sharing services (Google Drive, Dropbox)
  - For password verification: Storing passwords in plaintext is insecure. So your passwords are stored as a hash.
  - For Digital signature
  - Lots of other crypto applications

- Finding similar records: Records with similar but not identical keys
  - Spelling suggestion/corrector applications
  - Audio/video fingerprinting
  - Clustering

- Finding similar substrings in a large collection of strings
  - Genomic databases
  - Detecting plagiarism

- Geometric hashing: Widely used in computer graphics and computational geometry

Wrap Up

Hash Tables:

- Efficient find, insert, delete **on average, under some assumptions**
- Items not in sorted order
- Tons of real world uses
- …and really popular in tech interview questions.

Need to pick a good hash function.

- Have someone else do this if possible.
- Balance getting a good distribution and speed of calculation.

Resizing:

- Always make the table size a prime number.
- λ determines when to resize, but depends on collision resolution strategy.