Hash Open Indexing

Data Structures and Algorithms

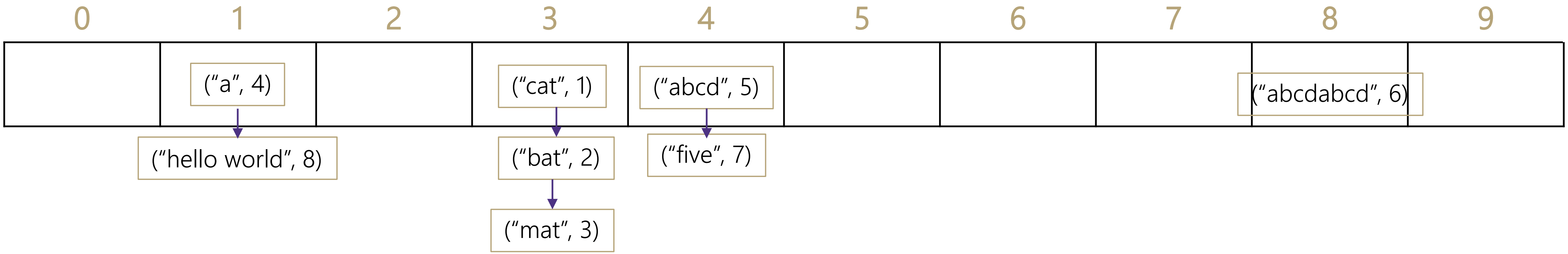
Warm Up

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)



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 $\lambda = \frac{n}{c}$

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, 7, 6, 25

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

Linear Probing

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

8

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

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?

$\lambda \approx ½$

Average number of probes for successful probe:

$$\frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$$

Average number of probes for unsuccessful probe:

$$\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

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

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

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 $\lambda < ½$)
Array lookups tend to be a constant factor faster than traversing pointers