Lecture 12: Open Addressing

Data Structures and Algorithms

Administrivia

Exercise 2 due tonight.

- Make sure you're assigning pages properly please!

Exercise 3 out sometime tonight.

Midterm in one week!

For the midterm, you are allowed one 8.5"x11" sheet of paper (both sides) for notes

- I strongly recommend you handwrite your note sheet.
- But you are free to generate it with a computer if you prefer.

Idea for note sheet: in the real-world you can often google stuff,
write down what you would lookup. It should also help you study.

We will provide you identities, we'll post the sheet in the exam resources early next week.

Midterm Topics (not exhaustive)

ADTs and Data structures

- Lists, Stacks, Queues, Dictionaries
- Array vs Node implementations of each
- Design decisions!

Asymptotic Analysis

- Proving Big O by finding c and n_0
- Modeling code runtime
- Finding closed form of recurrences using tree method and master theorem
- Looking at code models and giving simplified tight 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

Projects

- ArrayDictionary
- DoubleLinkedList

Resizing

Our running time in practice depends on λ. What do we do when λ is big?

Resize the array!

- Usually we double, that's not quite the best idea here
- Increase array size to next prime number that's roughly double the current size
  - Prime numbers tend to redistribute keys, because you're now modding by a completely unrelated number.
  - If % TableSize = k then `%2*TableSize` gives either k or k +`TableSize`.
- Rule of thumb: Resize sometime around when λ is somewhere around 1 if you're doing separate chaining.
  - When you resize, you have to rehash everything!

pollEV.com/cse373su19

Can we just copy over our old chains?

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 store a value if natural choice is already full.

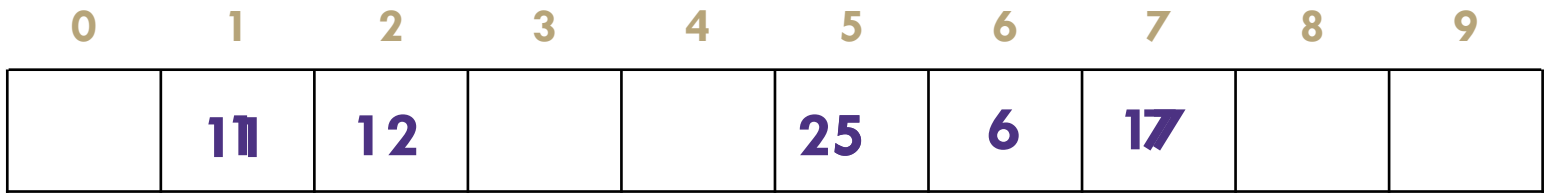
Type 1: Linear Probing

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

```
int findFinalLocation(Key s)
      int naturalHash = this.getHash(s);
      int index = natrualHash % TableSize;
      while (index in use) {
            i++;
            index = (naturalHash + i) % TableSize;
      }
      return index;
```

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



3 Minutes

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

2 Minutes

Runtime

When is runtime good?

When we hit an empty slot

- (or an empty slot is a very short distance away)

When is runtime bad?

When we hit a "cluster"

Maximum Load Factor?

λ at most 1.0

When do we resize the array?

$\lambda \approx ½$ is a good rule of thumb

Can we do better?

Clusters are caused by picking new space near natural index

Solution 2: Open Addressing

Type 2: Quadratic Probing

Instead of checking i past the original location, check i^2 from the original location.

```
int findFinalLocation(Key s)
      int naturalHash = this.getHash(s);
      int index = natrualHash % TableSize;
      while (index in use) {
            i++;
            index = (naturalHash + i*i) % TableSize;
      }
      return index;
```

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, 27

0	1	2	3	4	5	6	7	8	9
		58	79				27	18	89

(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

Now try to insert 9.

Uh-oh

Problems:

If $\lambda \geq \frac{1}{2}$ we might never find an empty spot

Infinite loop!

Can still get clusters

Quadratic Probing

There were empty spots. What gives?

Quadratic probing is not guaranteed to check every possible spot in the hash table.

The following is true:

If the table size is a prime number p, then the first $p/2$ probes check distinct indices.

Notice we have to assume p is prime to get that guarantee.

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

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

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 relatively prime to table size

```
int findFinalLocation(Key s)
      int naturalHash = this.getHash(s);
      int index = natrualHash % TableSize;
      while (index in use) {
            i++;
            index = (naturalHash + i*jumpHash(s)) % TableSize;
      }
      return index;
```

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 anything except a multiple of the TableSize

Resizing: Open Addressing

How do we resize? Same as separate chaining

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

When to resize?

- Depending on our load factor λ AND our probing strategy.
- Hard Maximums:
  - If $\lambda = 1$, `put` with a new key fails for linear probing.
  - If $\lambda > 1/2$ `put` with a new key **might** fail for quadratic probing, even with a prime `tableSize`
    - And it might fail earlier with a non-prime size.
  - If $\lambda = 1$ `put` with a new key fails for double hashing
    - And it might fail earlier if the second hash isn't relatively prime with the `tableSize`

Running Times

What are the running times for:

`insert`

Best: $O(1)$

Worst: $O(n)$ (we have to make sure the key isn't already in the bucket.)

`find`

Best: $O(1)$

Worst: $O(n)$

`delete`

Best: $O(1)$

Worst: $O(n)$

In-Practice

For open addressing:

We'll **assume** you've set λ appropriately, and that all the operations are $\Theta(1)$.

The actual dependence on λ is complicated – see the textbook (or ask me in office hours)

And the explanations are well-beyond the scope of this course.

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

Summary

Separate Chaining

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

Open Addressing

- Uses less memory (usually).
- Various schemes:
- Linear Probing – easiest, but lots of clusters
- Quadratic Probing – middle ground, but need to be more careful about λ.
- 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

Hash functions with some additional properties

Cryptographic hash functions: A small change in the key completely changes the hash.

- Commonly used in practice: SHA-1, SHA-265
- 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.
- Digital signatures
- Lots of other crypto applications

Other Applications of Hashing

Locality Sensitive Hashing – hash functions that map similar keys to similar hashes.

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

Extra optimizations

Idea 1: Take in better keys

- Really up to your client, but if you can control them, do!

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 buckets get large

Idea 3: Modify the array's internal capacity

- When load factor gets too high, resize array
  - Increase array size to next prime number that's roughly double the array size
  - Let the client fine-tune the λ that causes you to resize

Wrap Up

Hash Tables:

- Efficient find, insert, delete **in practice, 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.