Hash tables

Hash functions
Open addressing
Chaining

Dictionary ADT

Data structures for ADT implementation

- (Un)ordered arrays/linked lists, BSTs
  - Complexity of insert/remove/lookup operations range from $O(1)$ to $O(n)$

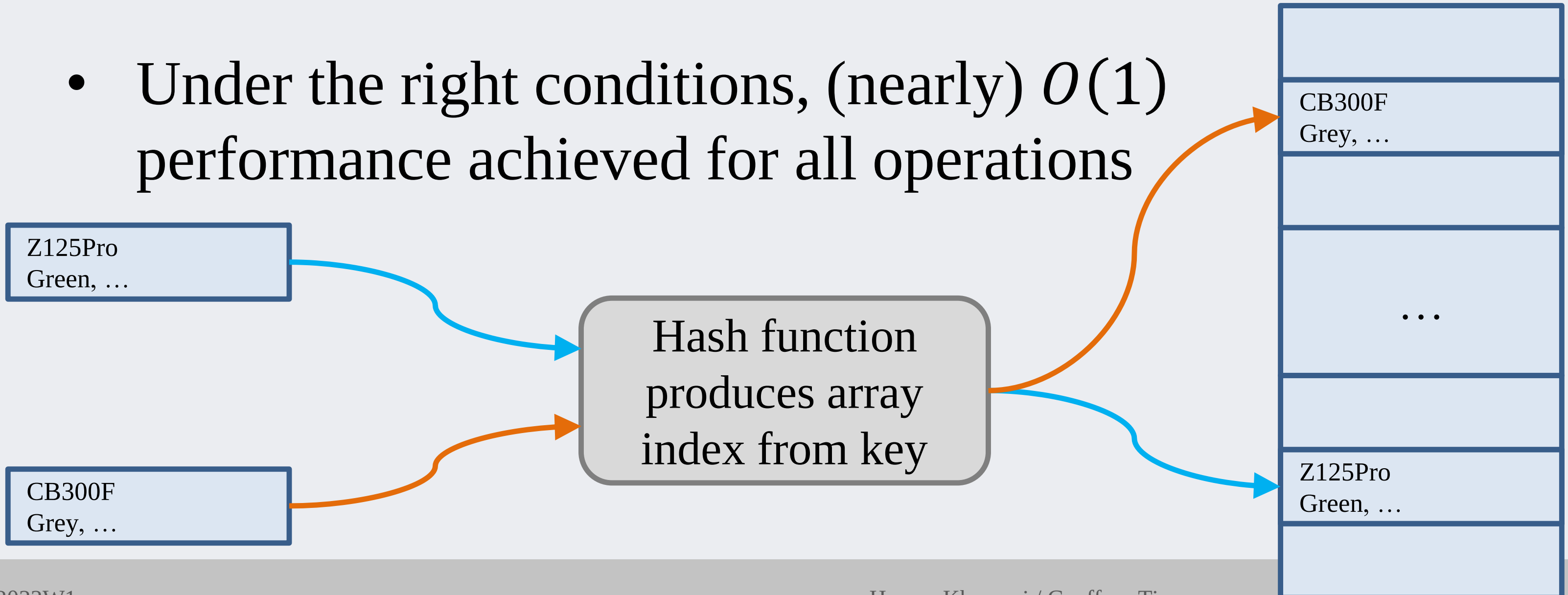
- The quickest access can be achieved by arrays, *if* the index of the desired item is known
  - and if there is no need to shuffle elements to open or fill gaps

Hash tables

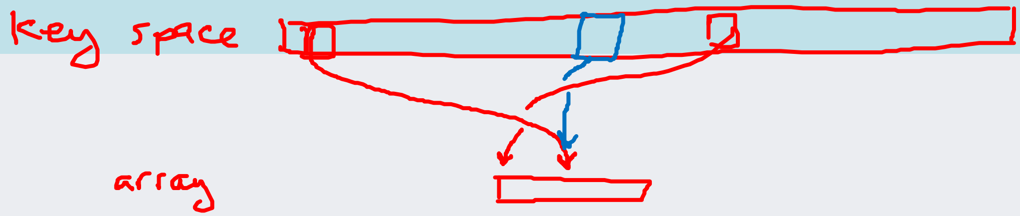
Arrays with gaps and "known" indices

- A hash table consists of an *array* to store data
  - Data often consists of complex types, or pointers to such objects
  - One attribute of the object is designated as the table entry's key
- A *hash function* maps a key to an array index in 2 steps
  - The key should be converted to an integer
  - And then that integer mapped to an array index using some function (often the modulo function)

- Under the right conditions, (nearly) $O(1)$ performance achieved for all operations



Hash function properties



- Hash functions should be:
  - Fast
    - If the hash function is slow to compute, $O(1)$ performance cannot be achieved
  - Deterministic
    - If the hash function maps the same key to different indices at different times, search may not be successful
  - Produce an uniform, random distribution over expected and all possible key values
    - Performance decreases if the array usage is concentrated around certain indices

Collisions

Key space vs array space

- Array has a limited capacity
- Keys may span a very wide range of values
  - Only a portion of these may be chosen to store into the array

- Hash function must map every possible key to some array index
  - Due to pigeonhole principle, any uniform hash function must map several keys to the same index – a collision!

- A good hash function reduces the number and effect of collisions
  - general principle: scatter data across the entire array, and "similar" keys should not map to "similar" indices
  - this should apply to all keys in the key space, as well as the subset of keys chosen for insertion

All keys vs inserted keys

Consider a hash table whose array has capacity = 676, where the keys will be people's names (strings in lower-case). The hash function is $h(str) = 26 \times str[0] + str[1]$, where $str[0]$ is the first character of the string, and $str[1]$ is the second character, and characters have values $a = 0, b = 1$, etc.

"charlie"
2 8

$26 \times 2 + 8 = 60$

Is this a good hash function to use for this hash table?

fast ✓
deterministic ✓
uniform, random? ✗

A. This is great! This is uniformly distributed across all possible combinations of two starting letters.

$\boxed{\text{B.}}$ This is bad! For… reasons.

C. It's OK. Not great, but not that bad either.

D. Can't answer now. Japan vs Croatia.

"brian"

"adam"

"adrian"

most of the indices will be unused

Collision resolution

- Collisions inevitably occur (e.g. attempting to insert two keys into the hash table which both map to the same index)
  - so the hash table must include a mechanism to resolve collisions

- Open addressing
  - each array index stores the data type of the data value to be inserted
  - When attempting to insert into an array index which is already occupied, insert at some other index (following some prescribed method for locating a free space, called probing)

see sample open addressing hash table code on course website

- Chaining
  - each array index stores a collection structure of the data value's data type (e.g. a linked list)
  - When inserting into an array index, add the data value to the linked list residing at that index

Open addressing

First version: linear probing

- The hash table is searched sequentially
  - **Starting with the original hash location**
  - For each time the table is probed (for a free location) add one to the index (modulo array capacity)
    - Search *h*(*search key*) + 1, then *h*(*search key*) + 2, and so on until an available location is found
    - If the sequence of probes reaches the last element of the array, wrap around to *arr*[0]
- Linear probing leads to *primary clustering*
  - The table contains groups of consecutively occupied locations
  - These clusters tend to get larger as time goes on
    - Reducing the efficiency of the hash table

Linear probing example

- Hash table is size 23 – it's good to make array capacity a prime number
- The hash function, $h(x) = x \bmod 23$, where x is the search key value
- Some existing search key values are shown in the array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58									21	

Linear probing example

- Insert 81, $h = 81 \bmod 23 = 12$
- Which collides with 58 so use linear probing to find a free space
- First look at 12 + 1, which is free so insert the item at index 13

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58									21	

Linear probing example

- Insert 35, $h = 35 \bmod 23 = 12$
- Which collides with 58 so use linear probing to find a free space
- First look at 12 + 1, which is occupied so look at 12 + 2 and insert the item at index 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81								21	

Linear probing example

- Insert 60, $h = 60 \text{ mod } 23 = 14$
- Note that even though the key doesn't hash to 12 it still collides with an item that did
- First look at 14 + 1, which is free

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81	35							21	

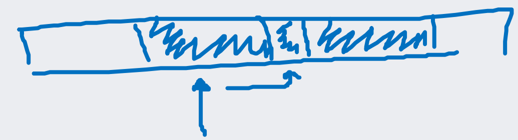
Linear probing example

- Insert 12, $h = 12 \bmod 23 = 12$
- The item will be inserted at index 16
- Notice that primary clustering is beginning to develop, making insertions less efficient

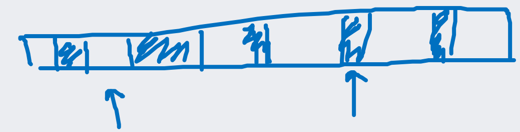
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81	35	60						21	

Try It!

- Insert the items into a hash table of 29 elements using linear probing:
  - 61, 19, 32, 72, 3, 76, 5, 34



- Using a hash function: $h(x) = x \bmod 29$
- Using a hash function: $h(x) = (x * 17) \bmod 29$



Searching

Example, linear probing

- Searching for an item is similar to insertion
- Find 59, $h = 59 \text{ mod } 23 = 13$, index 13 does not contain 59, but is occupied
- Use prescribed probe method to find 59 or an empty space
- Conclude that 59 is not in the table

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81	35	60	12					21	

- Search must use the same probe method as insertion
- Terminates when item found, empty space, or entire table searched

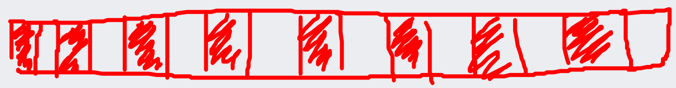
Hash table efficiency

- When analyzing the efficiency of hashing it is necessary to consider *load factor*, λ
  - λ = *number of items* / *table size*
  - As the table fills, λ increases, and the chance of a collision occurring also increases
    - Performance decreases as λ increases
  - Unsuccessful searches make more comparisons
    - An unsuccessful search only ends when a free element is found
- It is important to base the table size on the largest possible number of items
  - The table size should be selected/adjusted so that λ does not exceed 1/2

Suppose we have a hash table containing n elements, using linear probing for collision resolution, and the load factor is $\lambda = {}^1\!/_2$

What is the "best" worst-case complexity of a single insertion at this time?

Keys are uniformly distributed

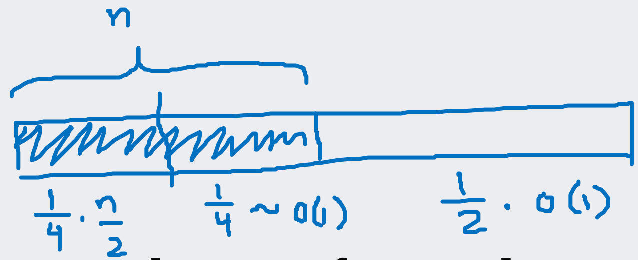


A. $O(1)$

B. $O(\log n)$

C. $O(n)$

D. $O(n^2)$



What is the "worst" worst-case complexity of a single insertion?

Clusters and load factor

- Primary clusters lead to performance degradation towards $O(n)$ as the load factor increases beyond $^1/_2$
  - As clusters get larger, new insertions are probabilistically more likely to land in a cluster
  - If a new insertion lands in a cluster, it is added to the end of the cluster, increasing its size and leading to a snowball effect
  - Clusters may even join together into large clusters
- Array can be resized, but existing elements must be re-inserted using updated hash function with new array capacity

Removals and open addressing

- Removals add complexity to hash tables
  - It is easy to find and remove a particular item
  - But what happens when you want to search for some other item?
  - The recently empty space may make a probe sequence terminate prematurely
- One solution is to mark a table location as either empty, occupied or removed (tombstone)
  - Locations in the removed state can be re-used as items are inserted
    - After confirming non-existence

Tombstones and performance

- After many removals, the table may become clogged with tombstones which must still be scanned as part of a cluster
  - it may be beneficial to periodically re-hash all valid items
- Example with linear probing and $h(x) = x \bmod 23$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
	X	X				X	29	X	X	X	X	54	X	60	X	X	35	X	X		21	X

$\lambda = \frac{5}{23} \sim 0.217$

`search(75)`

requires 15 probes!

After rehashing:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29		54				35		60							21	

$\lambda = \frac{5}{23} \sim 0.217$

`search(75)`

requires 2 probes 🙂

Open addressing

Other collision resolution schemes

- Assume:
  - $capacity$: array capacity
  - x: key
  - $h(x)$: initial hashed index of x
  - p: probe number, increments with each unsuccessful probe

- Item is inserted at index:
  - linear probing: $(h(x) + p) \bmod capacity$
  - quadratic probing: $(h(x) + p^2) \bmod capacity$
  - double hashing: $\left(h(x) + p \cdot h_2(x)\right) \bmod capacity$
    - where $h_2(x)$ is an auxiliary hash function producing a small offset value

Quadratic probing

- Quadratic probing is a refinement of linear probing that prevents primary clustering
  - For each probe, p, add p^2 to the original location index
    - 1st probe: $h(x) + 1^2$, 2nd: $h(x) + 2^2$, 3rd: $h(x) + 3^2$, etc.
- Results in secondary clustering
  - The same sequence of probes is used when two different values hash to the same location
  - This delays the collision resolution for those values
- Analysis suggests that secondary clustering is not a significant problem

Quadratic probing example

- Hash table is size 23
- The hash function, $h = x \bmod 23$, where x is the search key value
- The search key values are shown in the table

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58									21	

Quadratic probing example

- Insert 81, $h = 81 \bmod 23 = 12$
- Which collides with 58 so use quadratic probing to find a free space
- First look at $12 + 1^2$, which is free so insert the item at index 13

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58									21	

Quadratic probing example

- Insert 35, $h = 35 \text{ mod } 23 = 12$
- Which collides with 58
- First look at $12 + 1^2$, which is occupied, then look at $12 + 2^2 = 16$ and insert the item there

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81								21	

Quadratic probing example

- Insert 60, $h = 60 \bmod 23 = 14$
- The location is free, so insert the item

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81			35					21	

Quadratic probing example

- Insert 12, $h = 12 \bmod 23 = 12$, which is occupied
- First check index $12 + 1^2$,
- Then $12 + 2^2 = 16$,
- Then $12 + 3^2 = 21$ (which is also occupied),
- Then $12 + 4^2 = 28$, wraps to index 5 which is free

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81	60		35					21	

Quadratic probe sequences

- Note that after some time a sequence of probes repeats itself
  - In the preceding example *h*(*key*) = *key* % 23 = 12, resulting in this sequence of probes (table size of 23)
    - 12, 13, 16, 21, 28(**5**), 37(**14**), 48(**2**), 61(**15**), 76(**7**), 93(**1**), 112(**20**), 133(**18**), 156(**18**), 181(**20**), 208(**1**), 237(**7**), ...
- This generally does not cause problems if
  - The data is not significantly skewed,
  - The hash table is large enough (around 2 * the number of items), and
  - The hash function scatters the data evenly across the table
- Quadratic probing may potentially fail at $\lambda > \frac{1}{2}$
  - proof is outside the scope of this course

Double hashing

- In linear probing the probe sequence is independent of the key
- Double hashing produces *key dependent* probe sequences
  - In this scheme a second hash function, h_2, determines the probe sequence
- The second hash function must follow these guidelines
  - $h_2(\text{key}) \neq 0$
  - $h_2 \neq h_1$
  - A typical h_2 is $p - (\text{key mod } p)$ where p is a prime number

Double hashing example

- Hash table is size 23
- The hash function, $h = x \bmod 23$, where x is the search key value
- The second hash function, $h_2 = 5 - (\text{key} \bmod 5)$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58									21	

Double hashing example

- Insert 81, $h = 81 \text{ mod } 23 = 12$
- Which collides with 58 so use h_2 to find the probe sequence value
- $h_2 = 5 - (81\ mod\ 5) = 4$, so insert at 12 + 4 = 16

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58									21	

Double hashing example

- Insert 35, $h = 35 \bmod 23 = 12$
- Which collides with 58 so use h_2 to find a free space
- $h_2 = 5 - (35 \bmod 5) = 5$, so insert at 12 + 5 = 17

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58				81					21	

Double hashing example

- Insert 60, $h = 60 \bmod 23 = 14$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58				81	35				21	

Double hashing example

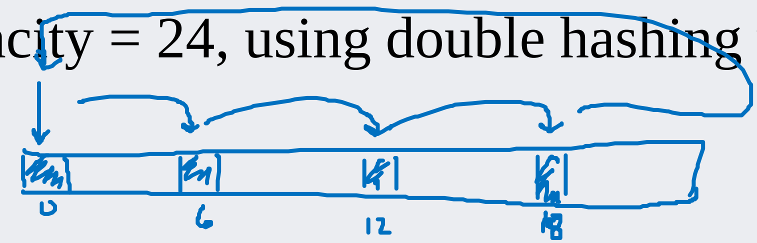
- Insert 83, $h = 83 \bmod 23 = 14$
- $h_2 = 5 - (83 \bmod 5) = 2$, so insert at 14 + 2 = 16, which is occupied
- The second probe increments the insertion point by 2 again, so insert at 16 + 2 = 18

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58		60		81	35				21	

iClicker 12.4

Consider a hash table with array capacity = 24, using double hashing with $h_2(key) = 7 - (key \bmod 7)$

Will this be an effective hash table?



$h(x) = 0$ $h_2(x) = 6$

A. Yes, it's basically the same thing as the previous example with the numbers changed

B. No, because… reasons…again

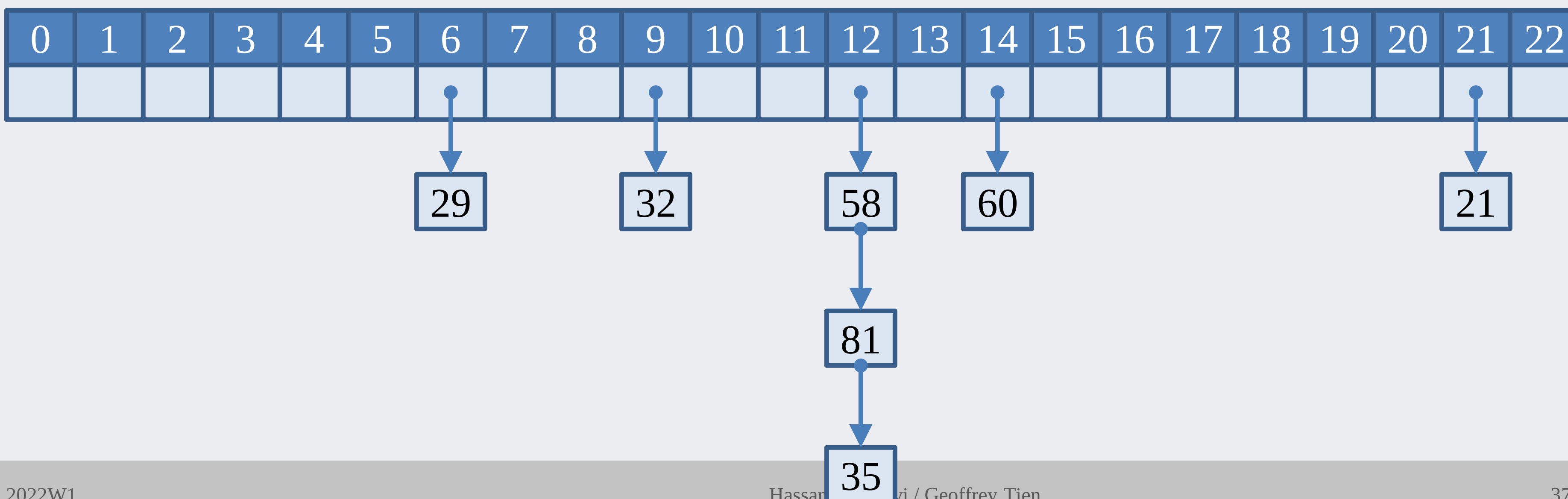
C. Can't wait for the course to be over!

Separate chaining

- Separate chaining takes a different approach to collisions
- Each entry in the hash table is a pointer to a linked list (or other dictionary-compatible data structure)
  - If a collision occurs the new item is added to the end of the list at the appropriate location
- Performance degrades less rapidly using separate chaining
  - with uniform random distribution, separate chaining maintains good performance even at high load factors $\lambda > 1$

Separate chaining example

- $h(x) = x \bmod 23$
- Insert 81, $h(x) = 12$, add to back (or front) of list
- Insert 35, $h(x) = 12$
- Insert 60, $h(x) = 14$



Hash table discussion

- If λ is less than ½, open addressing and separate chaining give similar performance
  - As λ increases, separate chaining performs better than open addressing
  - However, separate chaining increases storage overhead for the linked list pointers
- It is important to note that in the worst case hash table performance can be poor
  - That is, if the hash function does not evenly distribute data across the table

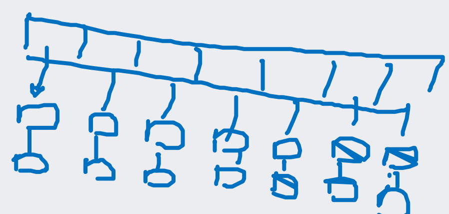
Chaining performance

with capacity n

- Suppose our table contains n keys. What would the worst case search?

capacity = n   expected cost: $\frac{1}{n}\cdot n + \frac{n-1}{n}\cdot 1$ close to 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22



$\lambda = 2$, max search = 2 probes

But what if the next key that we search for lands in any one of the other indices?

Then, what is the average length of each list?

λ, and thus this will also be the *expected* number of probes to make for a (failed) search or insertion that checks for duplicates

This is how chaining achieves such performance, e.g. even at $\lambda = 2$, it means we expect to make at most 2 probes for any search, whereas a load factor this high is impossible to achieve with open addressing

Removals and chaining

- With open addressing, we had to handle removals by setting flags in the array
- Removals are much simpler with chaining, assuming the work of implementing the chaining structure is already done
  - just call a removal method on the list at the hashed index!

Readings for this lesson

- Thareja
  - Chapter 15.5.1 (Linear probing, quadratic probing, double hashing)
  - Chapter 15.5.2 (Chaining)
  - Have a look at the open addressing code sample on the course webpage
  - See if you can implement a hash table with singly-linked list chaining

- Congratulations, we're done!
  - Geoff's office hours during exam period:
    - Usual hours on Monday