BASICS OF HASHING AND HASH TABLES

Hashing is a core technique in CS

"Computer Science has only three ideas: cache, hash, trash." - Prof. Greg Ganger, CMU

We discussed one way to “organize” information for efficient query — sorting

In general, allows O(lg n) access to data

But maintaining sorted data can be hard — O(n) time to insert / remove from a sorted array

We can do better than this:

Red-black tree

Space complexity	
Space	$O(n)$
Time complexity	
Function	**Amortized** / **Worst Case**
Search	$O(\log n)$[1] / $O(\log n)$[1]
Insert	$O(1)$[2] / $O(\log n)$[1]
Delete	$O(1)$[2] / $O(\log n)$[1]

B+-tree

Time complexity in big O notation		
Operation	**Average**	**Worst case**
Search	O(log *n*)	O(log *n*)
Insert	O(log *n*)	O(log *n*)
Delete	O(log *n*)	O(log *n*)
Space complexity		
Space	O(*n*)	O(*n*)

Splay tree

Space complexity		
Space	O(*n*)	
Time complexity		
Function	**Amortized**	**Worst Case**
Search	O(log *n*)[1]:659	O(*n*)[2]:1
Insert	O(log *n*)[1]:659	O(*n*)
Delete	O(log *n*)[1]:659	O(*n*)

Skip list

Time complexity in big O notation		
Operation	**Average**	**Worst case**
Search	$O(\log n)$	$O(n)$[1]
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$
Space complexity		
Space	$O(n)$	$O(n \log n)$[1]

Can we do better?

Hashing is a core technique in CS

Hash table!

Time complexity in big O notation		
Operation	**Average**	**Worst case**
Search	Θ(1)	O(*n*)
Insert	Θ(1)	O(*n*)
Delete	Θ(1)	O(*n*)
Space complexity		
Space	Θ(*n*)[1]	O(*n*)

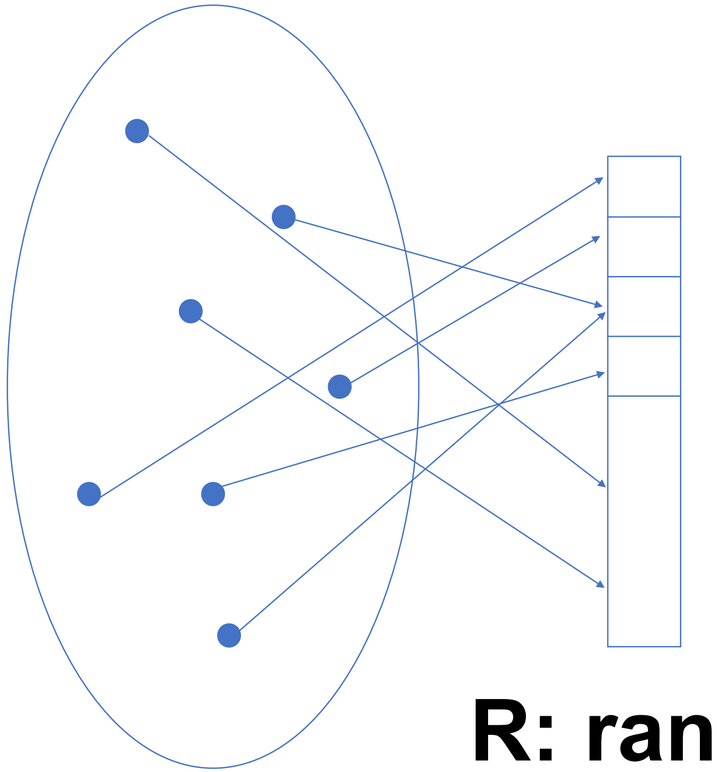
Note — we are concerned here mostly with the “Average” case; the “Worst” case is still bad (but very unlikely).

The hash table data structure provides an *unordered* associative array — a way to associate keys with values so that they can be stored compactly, retrieved quickly, new values can be added and removed quickly.

The array is *unordered*, we don’t need to define an order relationship for this to work, but this also means the structure, in some ways, provides less order than *sorted* data structures.

What is hashing?

D: domain



R: range

We want to map some set of keys from a domain, potentially of very large size, to some continuous range (much smaller).

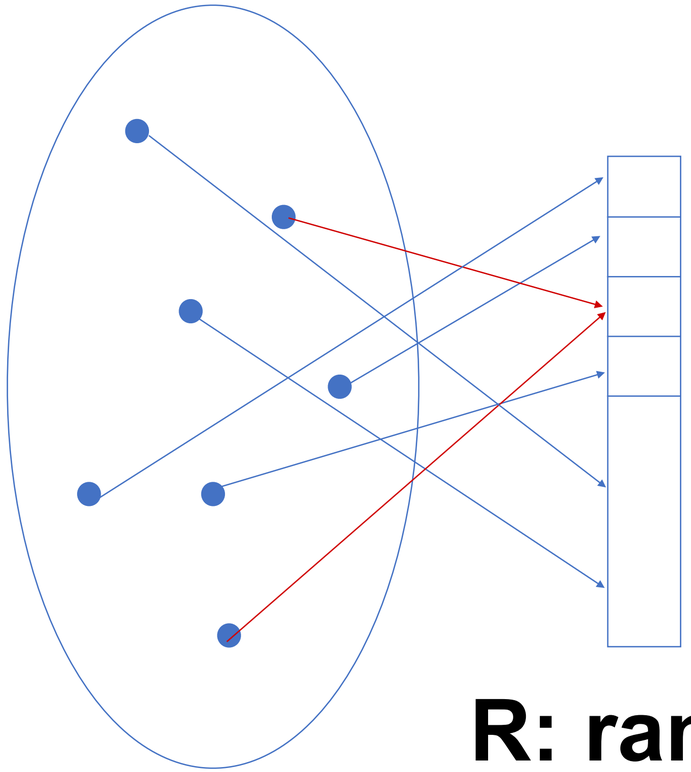
There's a set of keys $K \subseteq D$ that is generally of cardinality much smaller than D. Often, we don't know *what* K is ahead of time.

The hash is the map $H : D \rightarrow R$

Example:
D = k-mers for k = 31 (4.611686e+18)
K = k-mers in human genome (~2,500,000,000)
R = [0, m * 2,500,000,000] for "small" m (e.g. < 2)

What is hashing?

D: domain



R: range

The hash is the map H : D → R

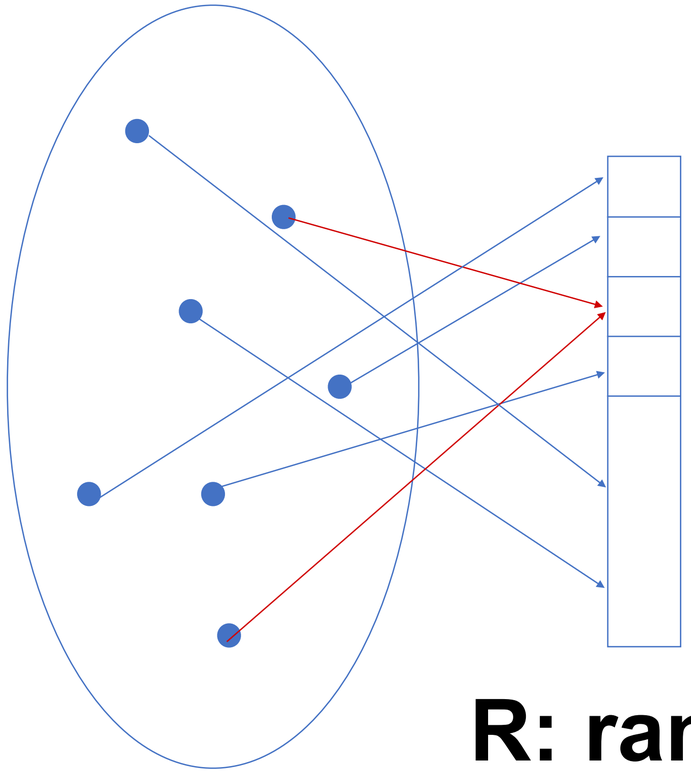
In general, this map can have *collisions*, but a "good" hash has a small random collision probability. A hash function with no collisions is known as a *perfect* hash function (will discuss in future lectures).

Overall, designing good, fast, hash functions is an art and a science — but we have many!

Example:
D = k-mers for k = 31 (4.611686e+18)
K = k-mers in human genome (~2,500,000,000)
R = [0, m * 2,500,000,000] for "small" m (e.g. < 2)

What is hashing?

D: domain



R: range

Overall, designing good, fast, hash functions is an art and a science — but we have many!

smhasher

SMhasher

Linux Build status build passing build passing

Hash function	MiB/sec	cycl./hash	cycl./map	size	Quality problems
donothing32	11149460.06	4.00	-	13	bad seed 0, test NOP
donothing64	11787676.42	4.00	-	13	bad seed 0, test NOP
donothing128	11745060.76	4.06	-	13	bad seed 0, test NOP
NOP_OAAT_read64	11372846.37	14.00	-	47	test NOP
BadHash	769.94	73.97	-	47	bad seed 0, test FAIL
sumhash	10699.57	29.53	-	363	bad seed 0, test FAIL
sumhash32	42877.79	23.12	-	863	UB, test FAIL
multiply_shift	8026.77	26.05	226.80 (8)	345	bad seeds & 0xffffffff0, fails most tests
pair_multiply_shift	3716.95	40.22	186.34 (3)	609	fails most tests
crc32	383.12	134.21	257.50 (11)	422	insecure, 8590x collisions, distrib, PerlinNoise
md5_32	350.53	644.31	894.12 (10)	4419	
md5_64	351.01	656.67	897.43 (12)	4419	
md5-128	350.89	681.88	894.03 (13)	4419	
sha1_32	353.03	1385.80	1759.94 (5)	5126	Sanity, Cyclic low32, 36.6% distrib
sha1_64	353.03	1385.80	1759.94 (5)	5126	Sanity, Cyclic low32, 36.6% distrib
sha1-160	364.95	1470.55	1794.16 (13)	5126	Comb/Cyclic low32
sha2-224	147.13	1354.81	1589.92 (12)		Comb low32
sha2-224_64	147.60	1360.10	1620.93 (13)		Cyclic low32
sha2-256	147.80	1374.90	1606.06		

https://rurban.github.io/smhasher/

Catalog of performance of different hashes

2 general and good hashes are

Wyhash:

https://github.com/wangyi-fudan/wyhash

xxHash (XXH3):

https://github.com/Cyan4973/xxHash

What is hashing?

Let’s look at some code!

https://github.com/umd-bioi607/notebooks/blob/main/hashing/Hashing.ipynb

What is hashing?

Great — so we can get “random” but consistent indices for our keys, but what do we do if we have collisions?

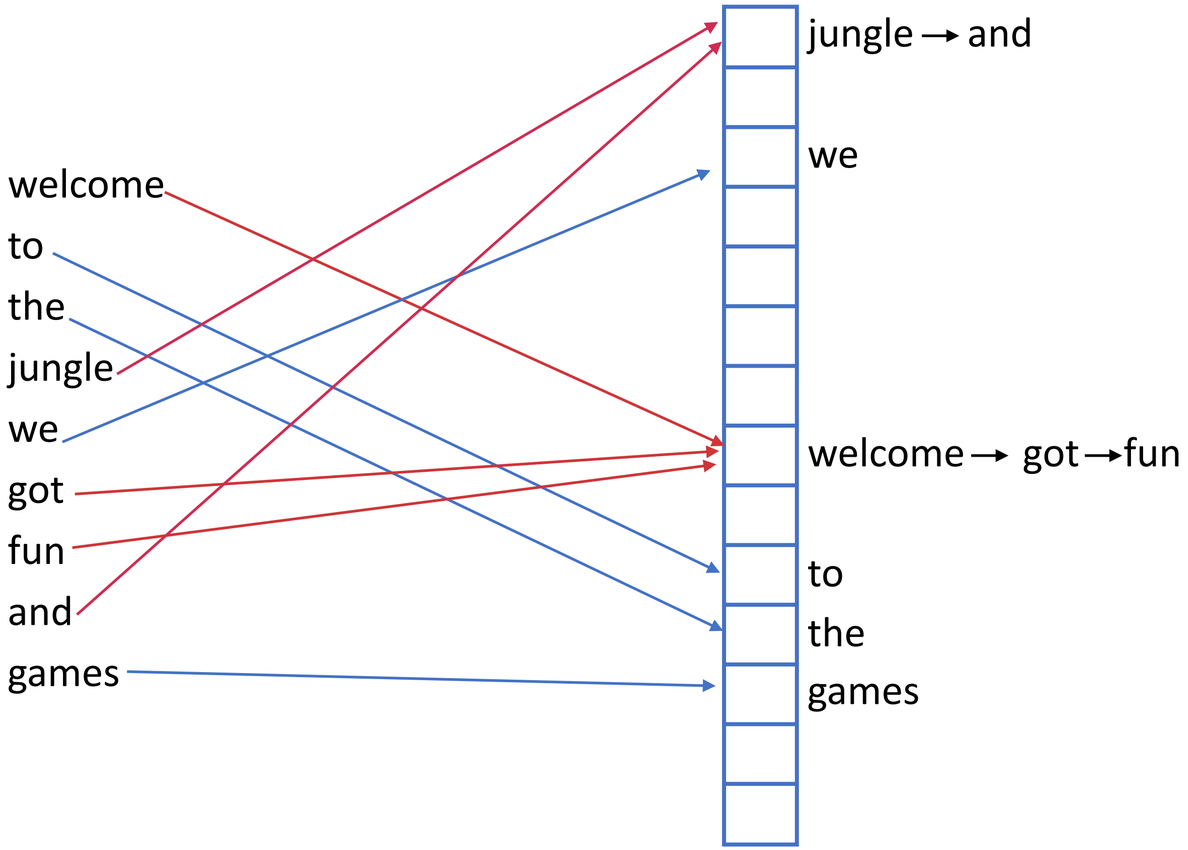
There’s a lot of work in practical hash table design that deals with this question, but we’ll discuss the two classic *collision resolution* methods.

Collision Resolution

Separate Chaining (closed addressing)

Collision Resolution

Separate Chaining (closed addressing)



Each “cell” in the hash table holds it’s own list/vector/array

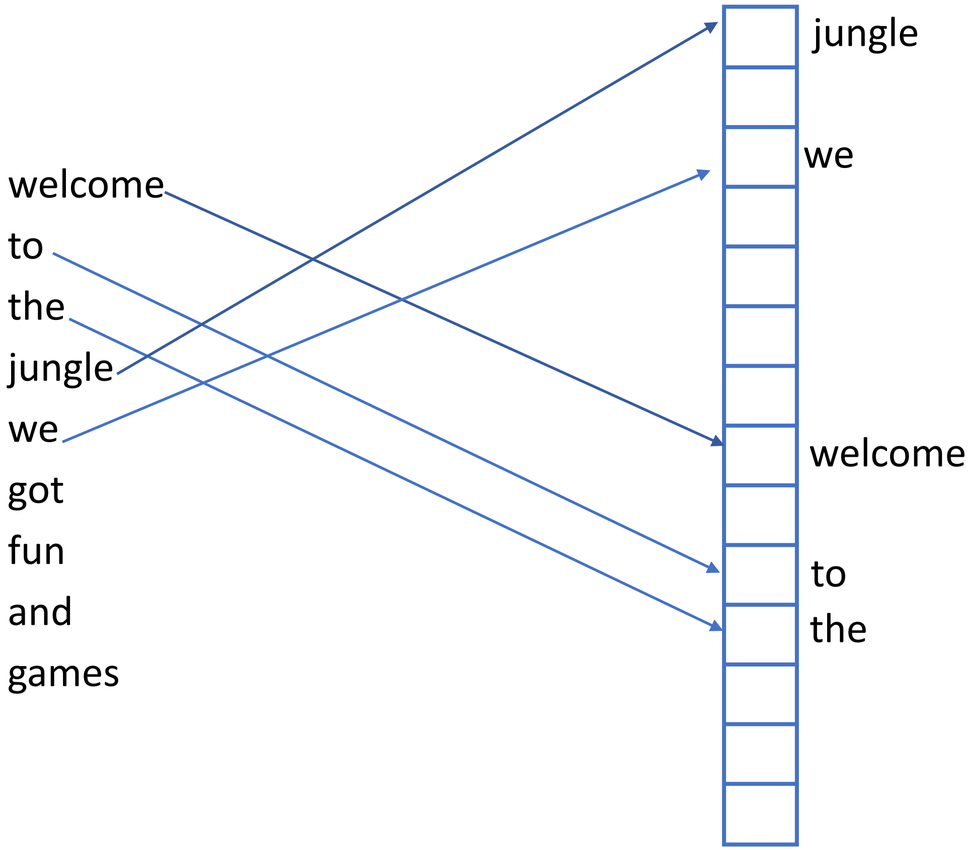
When we have a collision, we add the current key to the end of the list for it’s cell

Lookup requires computing h(k), visiting the corresponding cell, and walking the list until we find the key, or reach the end

The lookup speed depends on the average length of these lists, or (in the worst case) the longest such list.

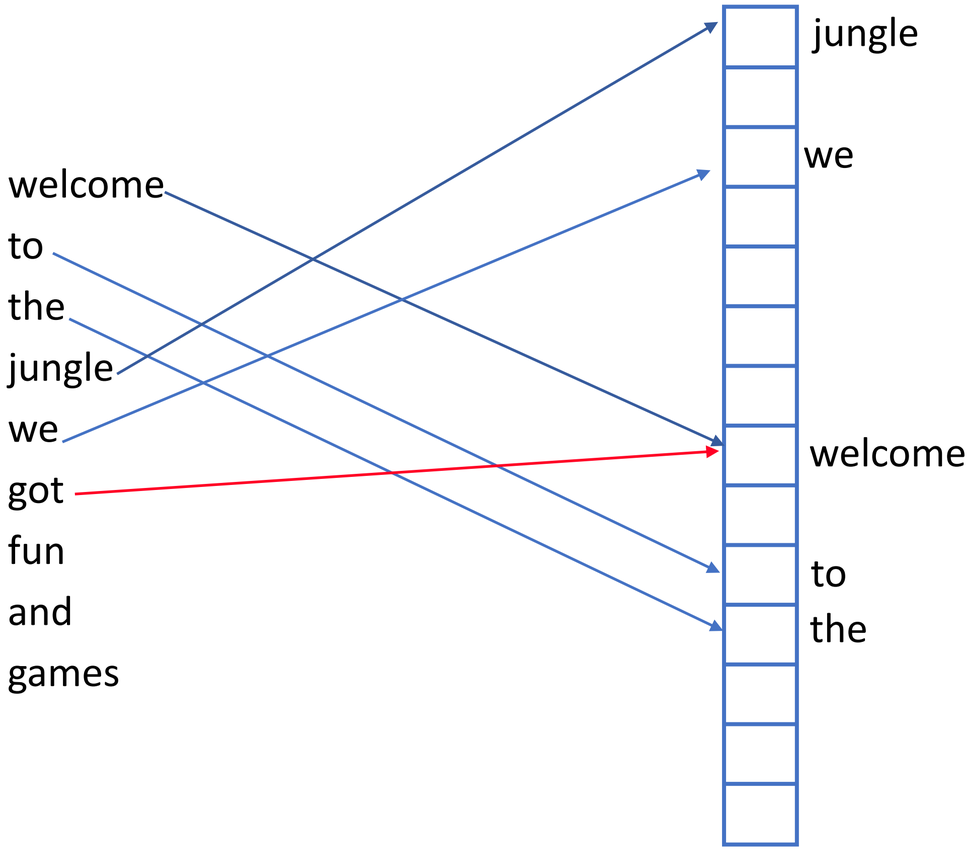
Collision Resolution

Linear Probing (open addressing)



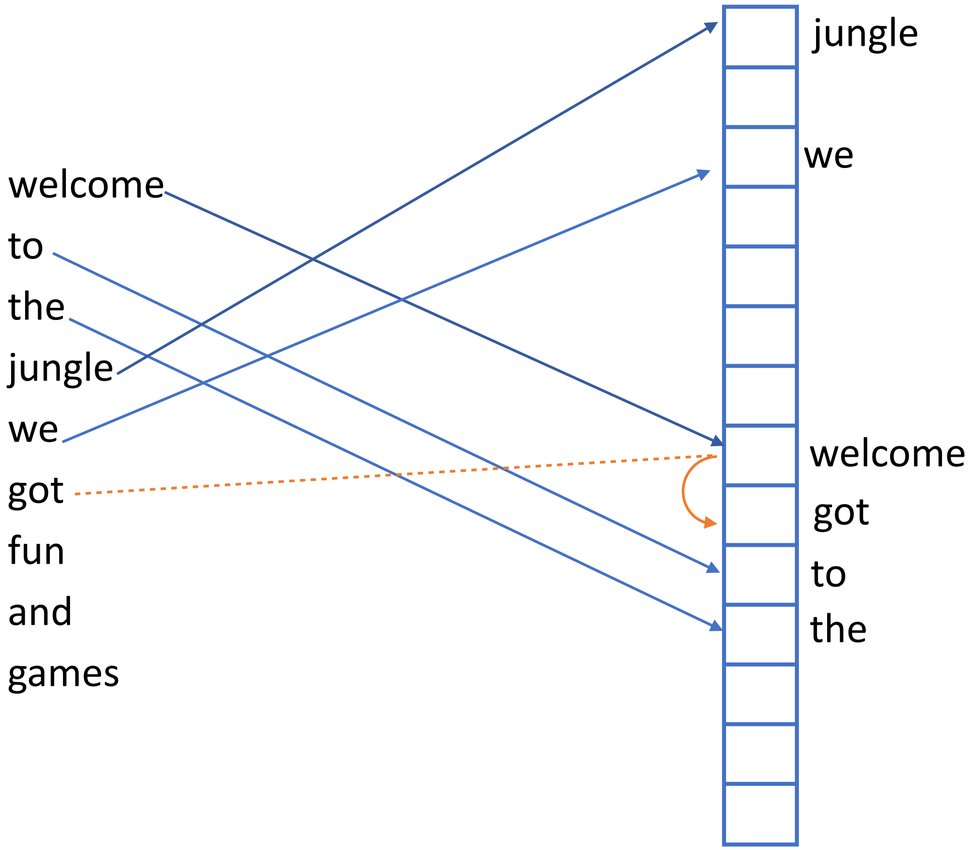
Collision Resolution

Linear Probing (open addressing)



Collision Resolution

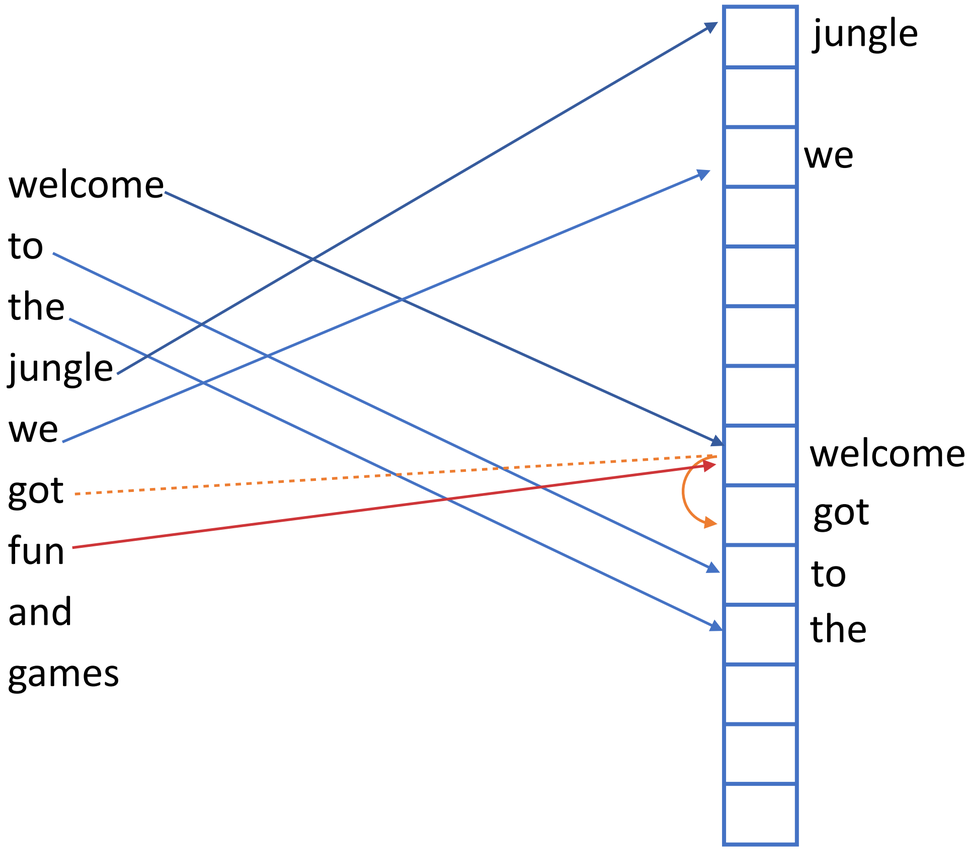
Linear Probing (open addressing)



When we observe a collision, scan down the table for the next *open* slot.

Collision Resolution

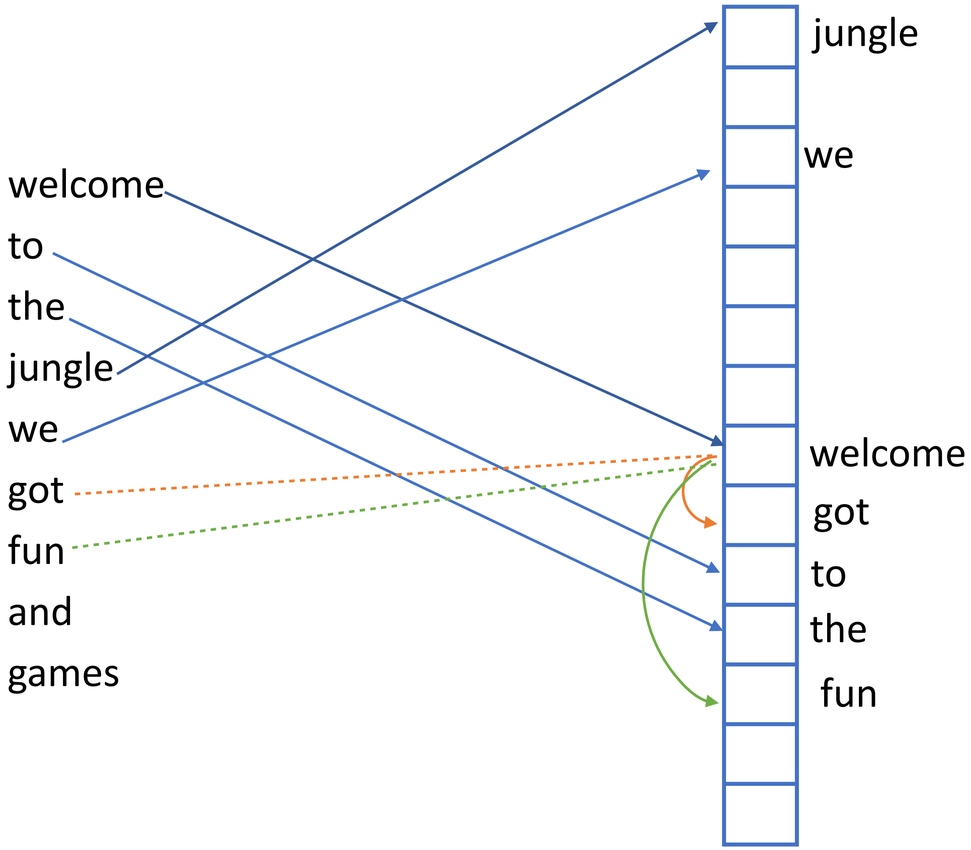
Linear Probing (open addressing)



When we observe a collision, scan down the table for the next *open* slot.

Collision Resolution

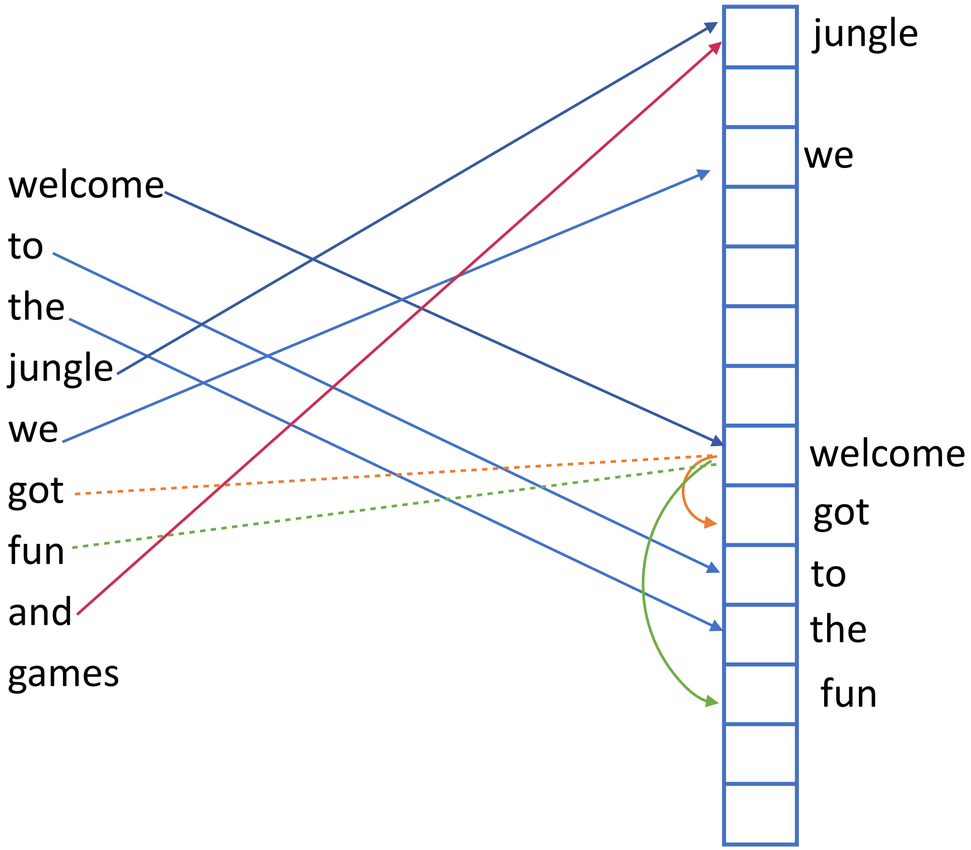
Linear Probing (open addressing)



When we observe a collision, scan down the table for the next *open* slot.

Collision Resolution

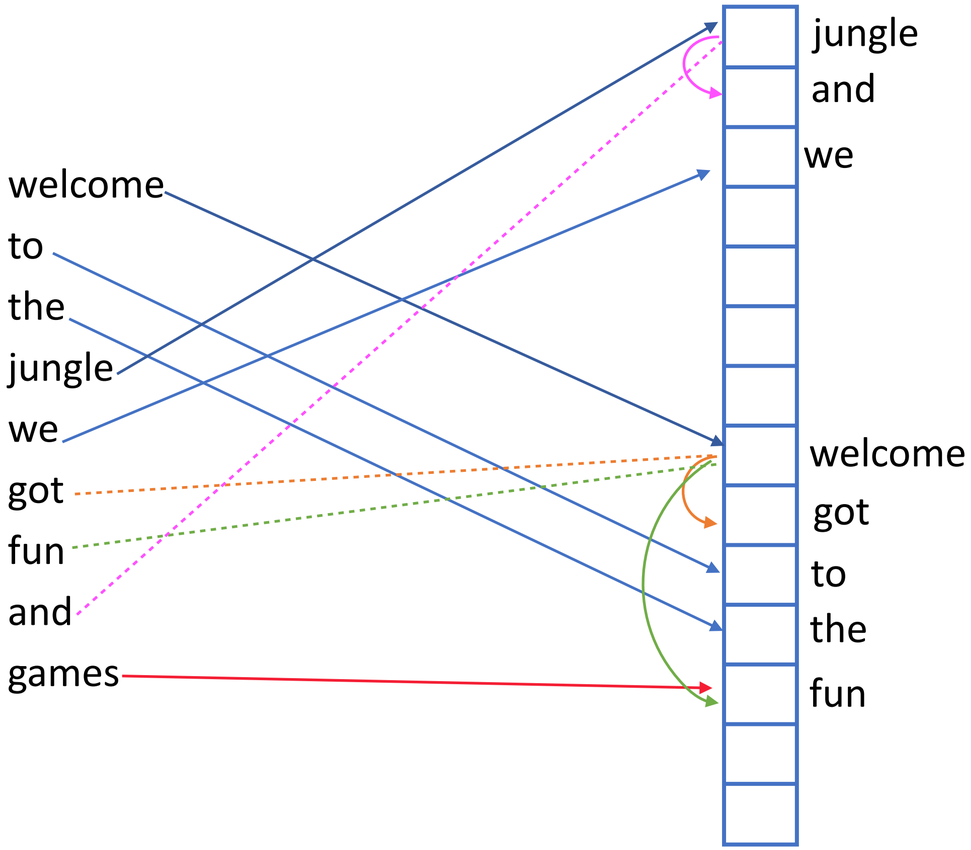
Linear Probing (open addressing)



When we observe a collision, scan down the table for the next *open* slot.

Collision Resolution

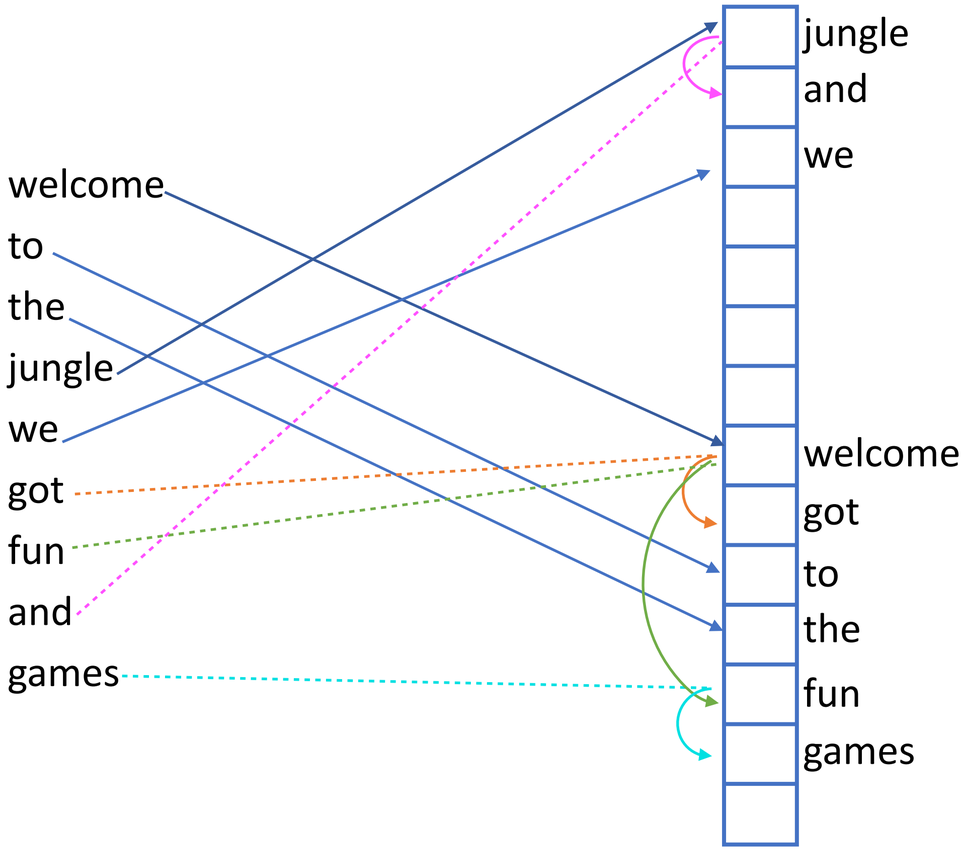
Linear Probing (open addressing)



When we observe a collision, scan down the table for the next *open* slot.

Collision Resolution

Linear Probing (open addressing)

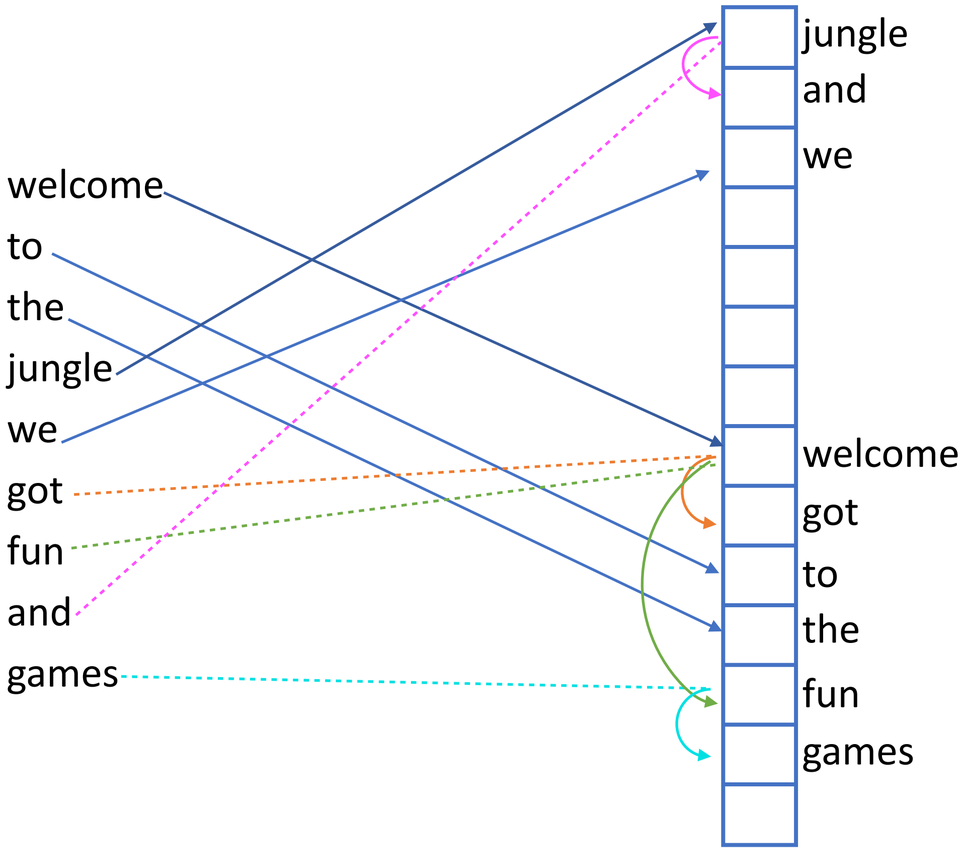


When we observe a collision, scan down the table for the next *open* slot.

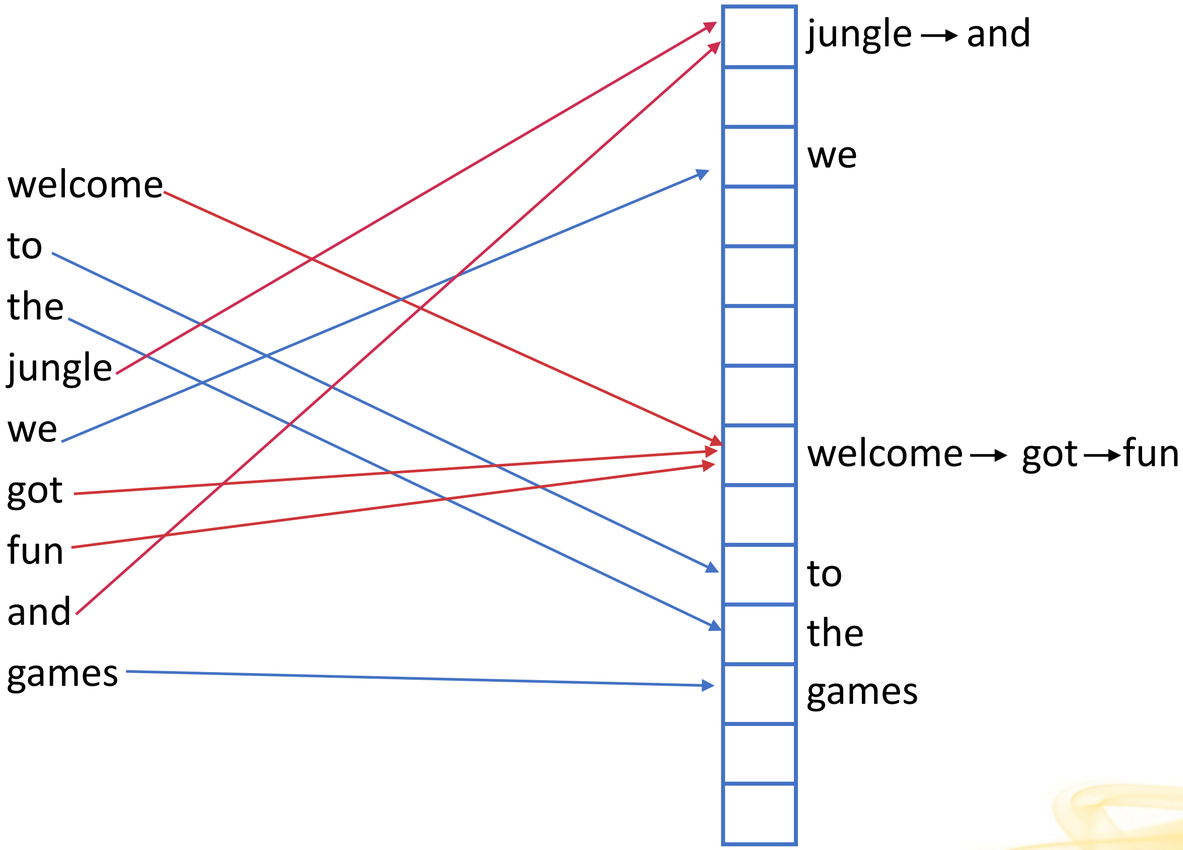
When we query, we start at the cell given by h(k), and then we must scan in the table until we either see k, or we observe an empty cell.

Collision Resolution

Linear Probing (open addressing)



Separate Chaining (closed addressing)



These provide different tradeoffs, depending on your use case

Most modern high-performance hash tables use **open addressing**

Back to the notebook!

Let’s look at some code!

https://github.com/umd-bioi607/notebooks/blob/main/hashing/Hashing.ipynb

Hashing as a basic indexing tool

Hashing (and hash tables) provide a general and useful primitive for building fast and efficient *associative arrays*.

We can associate a collection of keys with values, and be able to query / retrieve the key in O(1) time in the expected case.

Well designed hash tables can work at high load factors $\alpha =$ (m = elements in the table / n = table size) close to 1.

In fact, hash tables can provide state-of-the-art indices if we choose our keys wisely and store information efficiently.

We'll return later to talk about *minimal perfect hashes* — hash tables that guarantee that there are no collisions, and that m = n, but only when the key set is static and known at construction time.