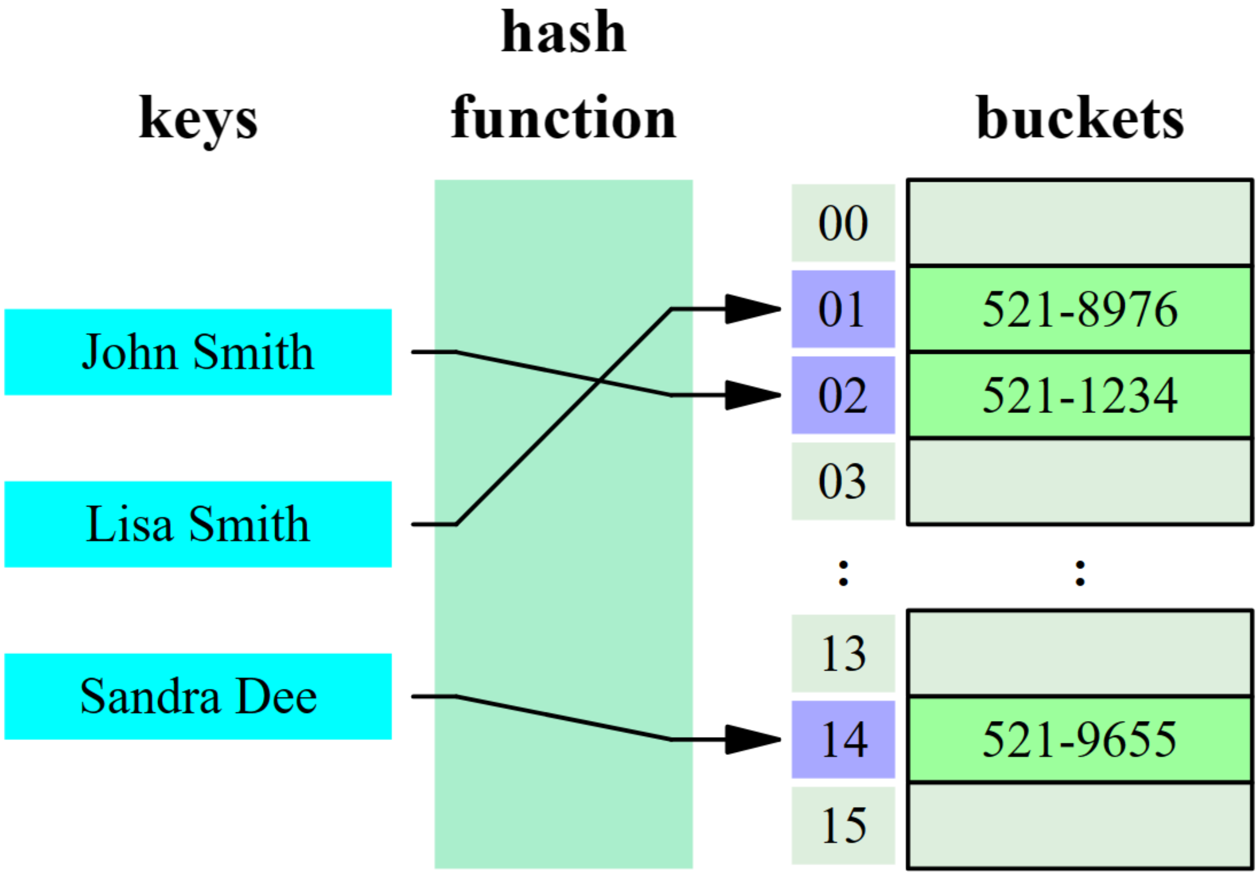
Hash tables

(Slides by Martin Escardo)

Hash Table



A hash table, also known as a hash map, is a data structure used to store and retrieve data efficiently.

It is based on the concept of a hash function, which takes an input (such as a key) and maps it to a fixed-size array index, called a hash code. This hash code determines the location where the data associated with the input will be stored or retrieved from.

Constant-time average-case complexity basic operations like insertion, lookup

Basic idea

The goal: We would like to be able to index arrays by non-integer keys:

```
arr[key] = value
```

(`key` might not be an integer!)

For example, indexing by strings: `museums["Bham"] = 13`.

But arrays are *only* indexed by integers.

$\Longrightarrow$ We need a **hash function** `hash(key)` which computes the index in `arr` for a given `key`:

```
arr[hash(key)] = value
```

Example 1: storing student assignments in $\mathcal{O}(1)$

When implementing Canvas, we store assignments of students in a hash table:

- `value`s = assignments
- `key`s = students
- `hash(s)` = the student ID of student `s`

Student IDs of the form 2183201, 1526020, ... 7-digit numbers

Allocate an array `arr` of size 10^7, then to store an assignment:

```
arr[hash(s)] = assignment
```

This is in $\mathcal{O}(1)$

Example 1: storing student assignments in $\mathcal{O}(1)$

When implementing Canvas, we store assignments of students in a hash table:

- `value`s = assignments
- `key`s = students
- `hash(s)` = the student ID of student `s`

Student IDs of the form 2183201, 1526020, ...   7-digit numbers

Allocate an array `arr` of size 10^7, then to store an assignment:

```
arr[hash(s)] = assignment
```

This is in $\mathcal{O}(1)$ but memory inefficient! :-(

Even if we only need to store assignments of 170 students, we still allocate an array of size 10^7!

Example 2: hash function based on the size of the array

Allocate an array `arr` of size 170 and compute `hash(s)` as

`studentID(s) mod 170`.

This way `hash(s)` is one of `0`, `1`, `2`, ... `arr.length-1`.

Example

Input

Student ID	Assignment Marks
2177147	85
2051025	60
2157143	75

hash(s) = studentID(s) mod 10

arr[hash(s)] = assignment

Example

Input

Student ID	Assignment Marks
2177147	85
2051025	60
2157143	75

hash(s) = studentID(s) mod 10

arr[hash(s)] = assignment

2. Store the assignment at index

Hash Table (array)

Key	Value
0	
1	
2	
3	75
4	
5	60
6	
7	85
8	
9	

1. Calculate hash value

2177147 mod 10 = 7

2051026 mod 10 = 5

2157143 mod 10 = 3

Example 2: hash function based on the size of the array

Allocate an array `arr` of size 170 and compute `hash(s)` as

`studentID(s) mod 170`.

This way `hash(s)` is one of `0`, `1`, `2`, … `arr.length-1`.

We might introduce **hash collisions**. That is, we can have

`hash(key1) == hash(key2)`

for two different keys/students `key1` and `key2`.

Collisions will happen even if we double/triple the size of `arr`.

$\Longrightarrow$ We need a mechanism for dealing with hash collisions.

Example

Input

Student ID	Assignment Marks
2177147	85
2051025	60
2157143	75
2000147	66

hash(s) = studentID(s) mod 10

arr[hash(s)] = assignment

1. Calculate hash value

2177147 mod 10 = 7

2051026 mod 10 = 5

2157143 mod 10 = 3

2000147 mod 10 = 7

Hash Collision

2. Store the assignment at index

Hash Table (array)

Key	Value
0	
1	
2	
3	75
4	
5	60
6	
7	85? 66?
8	
9	

Summary + Disclaimer

In summary, a **hash table** consists of

1. an array `arr` for storing the values,
2. a hash function `hash(key)`, and
3. a mechanism for dealing with collisions.

It implements the operations:

`set(key, value)`, `delete(key)`, `lookupValue(key)`.

Summary + Disclaimer

In summary, a **hash table** consists of

Exercise 1
(5 mins)

1. an array `arr` for storing the values,
2. a hash function `hash(key)`, and
3. a mechanism for dealing with collisions.

It implements the operations:

`set(key, value)`, `delete(key)`, `lookupValue(key)`.

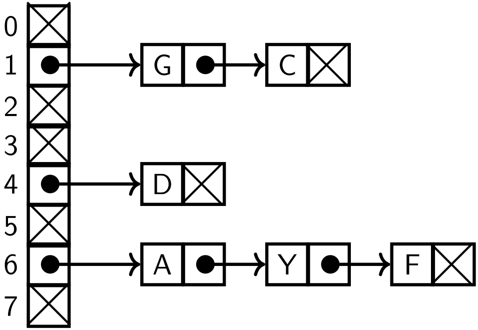
Disclaimer: We will consider a simplified situation where `key`s and `value`s are the same. For example, an assignment is always:

`arr[hash(key)] = key`.

And the operations change to: `insert(key)`, `delete(key)`, `lookup(key)`.

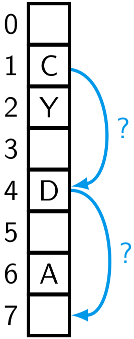
Two types of solutions of hash collisions

Sticking out strategy



Entries with the same `hash(key)` are stored in a linked list.

Tucked in strategy



If the position is occupied, we try different “fallback” positions.

Example: Direct chaining (= a sticking out strategy)

Entries: airport codes, e.g. BHX, INN, HKG, IST, ...

Table size: 10

Hash function:

- We treat the codes as a number in base 26
  (A=0, B=1, ..., Z=25).
  Example: ABC $= 0 * 26^2 + 1 * 26 + 2 = 28$
- The hashcode is computed `mod 10`
  (to make sure that the index is $0, 1, 2, 3, ...,$ or 9).
  Example:
  `hash(BHX) =(1*26*26 + 7*26 + 23)mod 10 = 1`

key	BHX	INN	HKG	IST	MEX	PRG	TPE
hash	1	9	8	5	9	8	8

A	B	C	D	E	F	G	H	I	J	K	L	M
$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$
0	1	2	3	4	5	6	7	8	9	10	11	12
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$
13	14	15	16	17	18	19	20	21	22	23	24	25

BHX $= 1 * 26^2 + 7 * 26^1 + 23 * 26^0 = 676 + 182 + 23 = 881$

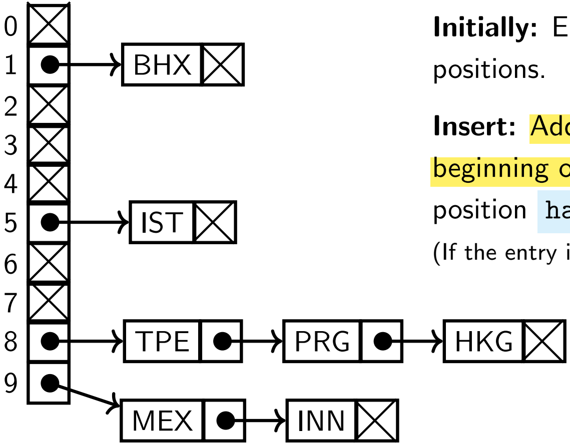
Hash (BHX) = (881) mod 10 = 1

DXB $= 3 * 26^2 + 23 * 26^1 + 1 * 26^0 = 2{,}028 + 598 + 1 = 2627$

Hash (DXB) = (2627) mod 10 = 7

Example: Direct chaining

`key`	BHX	INN	HKG	IST	MEX	PRG	TPE
`hash`	1	9	8	5	9	8	8



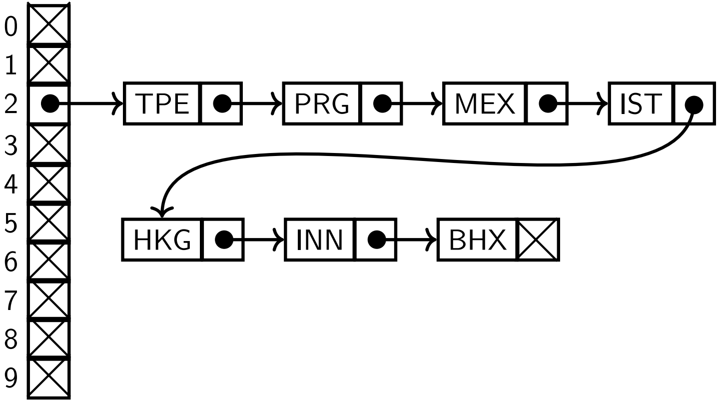
Initially: Empty lists on all positions.

Insert: Add a new node at the beginning of the list stored on position `hash(key)`.

(If the entry is not already in the list.)

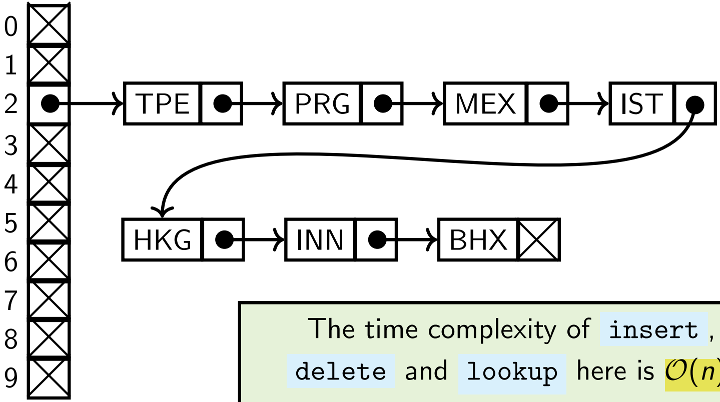
Example 2: Bad hash function

key	BHX	INN	HKG	IST	MEX	PRG	TPE
hash	2	2	2	2	2	2	2



Example 2: Bad hash function

key	BHX	INN	HKG	IST	MEX	PRG	TPE
hash	2	2	2	2	2	2	2



The time complexity of `insert`, `delete` and `lookup` here is $\mathcal{O}(n)$!

A **good** hash function `hash(key)` assigns indexes to keys **uniformly**.

Time Complexity of Direct Chaining, part 1

The **load factor** of a hash table is the *average* number of entries stored on a location:

$n =$ the total number of stored entries

$$\frac{n}{T}$$

$T =$ the size of the hash table

If we have a *good* hash function, a location given by `hash(key)` has the *expected* number of entries stored there equal to $\frac{n}{T}$.

Time Complexity of Direct Chaining, part 1

The **load factor** of a hash table is the *average* number of entries stored on a location:

$$\frac{n}{T}$$

$n =$ the total number of stored entries

$T =$ the size of the hash table

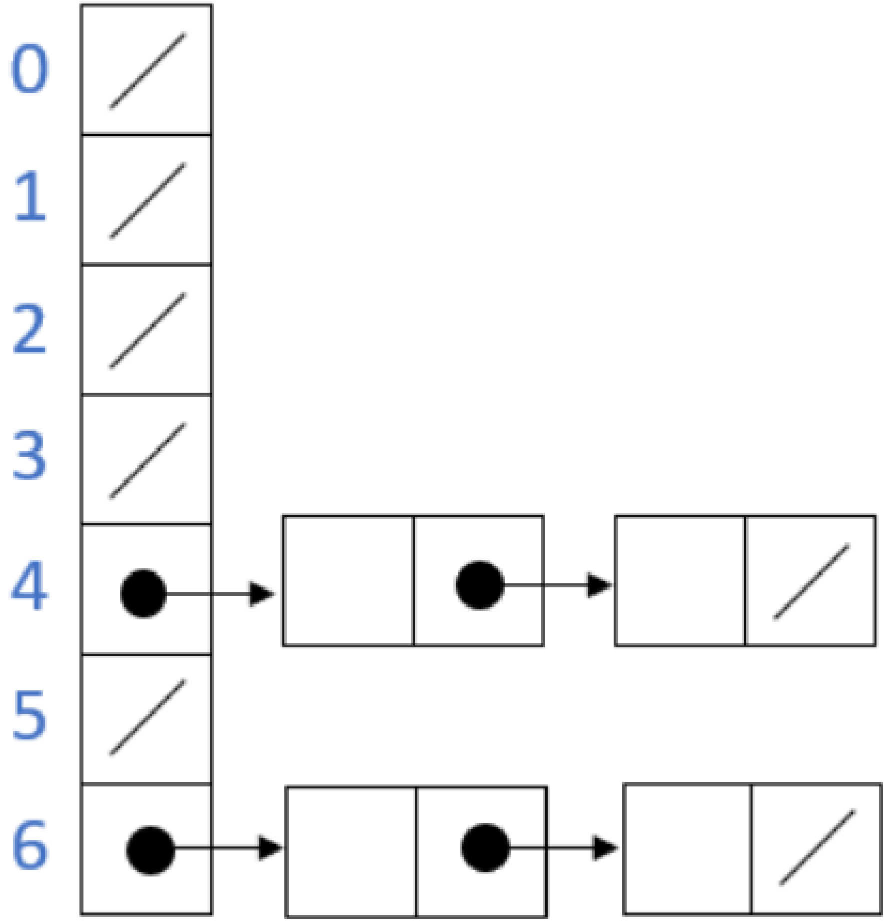
If we have a *good* hash function, a location given by `hash(key)` has the *expected* number of entries stored there equal to $\frac{n}{T}$.

Unsuccessful lookup of `key`:

- `key` is not in the table.
- Location `hash(key)` stores $\frac{n}{T}$ entries, *on average*.
- $\Longrightarrow$ We have to traverse them all.

Load Factor Example

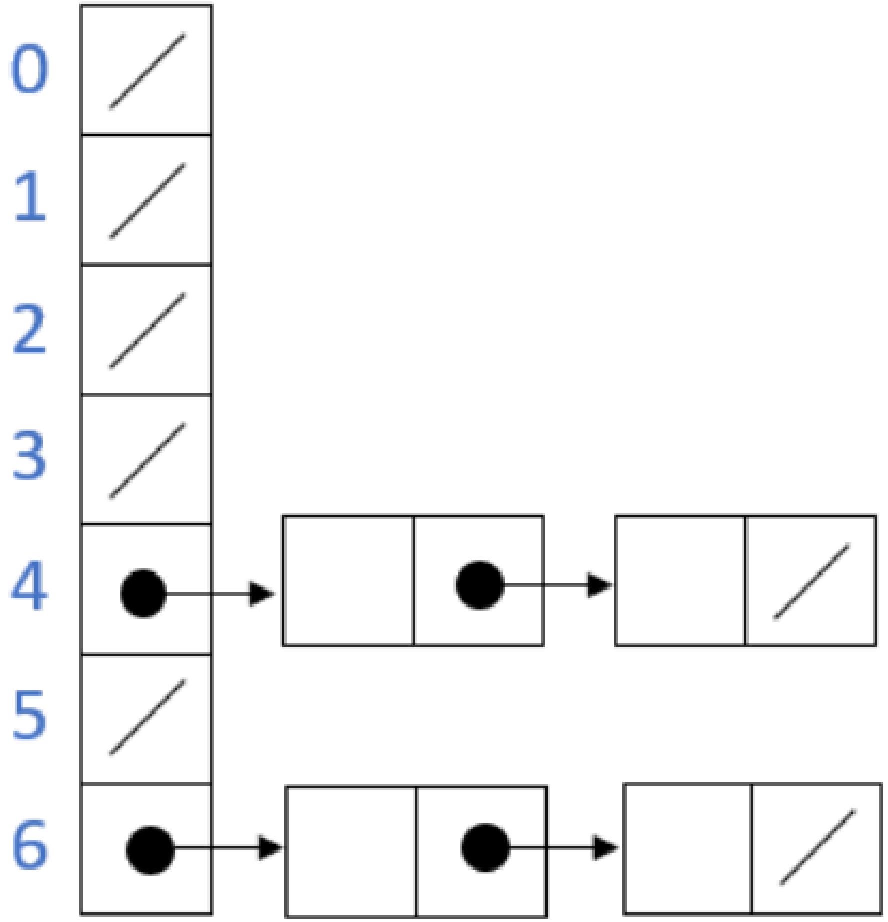
What is the load factor?



- 4/7
- 2/6
- 2/3
- 2/7

Load Factor Example

What is the load factor?



- 4/7
- 2/6
- 2/3
- 2/7

Time Complexity of Direct Chaining, part 2

Successful lookup of `key`:

- Location `hash(key)` stores $\frac{n}{T}$ entries, *on average*.
- The *expected* position of `key` the list is in the middle
  $\implies$ we traverse $\frac{1}{2}(1 + \frac{n}{T})$ many entries, *on average*.

Time Complexity of Direct Chaining, part 2

Successful lookup of `key`:

- Location `hash(key)` stores $\frac{n}{T}$ entries, *on average*.
- The *expected* position of `key` the list is in the middle

  $\implies$ we traverse $\frac{1}{2}(1 + \frac{n}{T})$ many entries, *on average*.

Assume **maximal load factor** λ, that is, $\frac{n}{T} \leq \lambda$

(For example, in Java $\lambda = 0.75$)

The *average case* time complexities:

- unsuccessful lookup: $\frac{n}{T} \leq \lambda$ comparisons $\implies \mathcal{O}(1)$
- successful lookup: $\frac{1}{2}(1 + \frac{n}{T}) \leq \frac{1}{2}(1 + \lambda)$ comparisons $\Rightarrow \mathcal{O}(1)$

λ is a constant number!

Time Complexity of Direct Chaining, part 3

The time complexity of `insert(key)` is the same as unsuccessful lookup:

- First check if the `key` is stored in the table.
- If it is not, append `key` at the beginning of the list on stored on `hash(key)`.

In total: $\frac{n}{T} + 1 \leq \lambda + 1 \implies \mathcal{O}(1)$.

Time Complexity of Direct Chaining, part 3

The time complexity of `insert(key)` is the same as unsuccessful lookup:

- First check if the `key` is stored in the table.
- If it is not, append `key` at the beginning of the list on stored on `hash(key)`.

In total: $\frac{n}{T} + 1 \leq \lambda + 1 \implies \mathcal{O}(1)$.

The time complexity of `delete(key)` is the same as successful lookup.

$\implies$ The time complexities of `insert`, `delete`, `lookup` are all $\mathcal{O}(1)$.

Disadvantages of “sticking out” strategies

1. Typically, there is a lot of hash collisions, therefore a lot of unused space.
2. Linked lists require a lot of allocations (`allocate_memory`), which is slow. (Also, for caching reasons.)

We will take a look at two **tucked-in strategies** which avoid those problems:

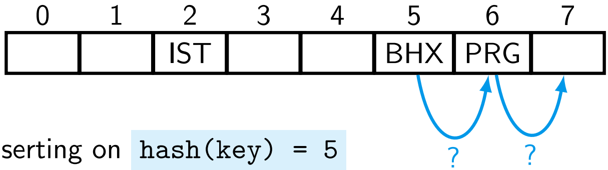
- Linear probing
- Double hashing

Linear probing (= a tucked in strategy)

Insertion (initial idea): If the primary position `hash(key)` is occupied, search for the first *available* position to the right of it.

If we reach the end, we wrap around!

Example



Inserting on `hash(key) = 5`

We use `mod` to compute the "fallback" positions:

`hash(key)+1 mod T`, `hash(key)+2 mod T`, `hash(key)+3 mod T`, ...

Deletion (idea):

1. Find whether the `key` is stored in the table:

   Starting from the primary position `hash(key)`, go the right, until the `key` or an empty position is found.

2. If the `key` is stored in the table, replace it with a **tombstone** (marked as **#**).

Example

Deleting `key = TPE` such that `hash(key) = 0`:

0	1	2	3	4	5	6	7
MEX	INN	**#**	TPE	HKG		PRG	

Linear probing, deletion

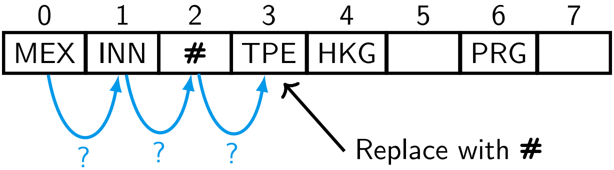
Deletion (idea):

1. Find whether the `key` is stored in the table:

   Starting from the primary position `hash(key)`, go the right, until the `key` or an empty position is found.
2. If the `key` is stored in the table, replace it with a **tombstone** (marked as **#**).

Example

Deleting `key = TPE` such that `hash(key) = 0`:



Searching:
Starting from the primary position `hash(key)`, search for the `key` to the right. We skip over all **tombstones #**.

If we reach an empty position, then the `key` is not in the table.

Inserting (more accurately):
First check if `key` is stored in the table, and if it is not and its the primary position `hash(key)` is occupied by a different key, search for the first **empty or tombstone** position to the right of it.

Store the `key` there.

Remark
Every positions is either **empty**, or it stores a **tombstone** or a **key**. Moreover, initially are all positions marked as *empty*.

Example: Linear probing

`key`	A	B	C	D	E	F
`hash`	0	4	5	6	5	4

0	1	2	3	4	5	6	7
A				B	C	D	

1. `insert(E)`

0	1	2	3	4	5	6	7
A				B	C	D	**E**

Example: Linear probing

key	A	B	C	D	E	F
hash	0	4	5	6	5	4

0	1	2	3	4	5	6	7
A				B	C	D	

1. `insert(E)`

0	1	2	3	4	5	6	7
A				B	C	D	**E**

2. `insert(F)`

A	**F**			B	C	D	E

Example: Linear probing

key	A	B	C	D	E	F
hash	0	4	5	6	5	4

0	1	2	3	4	5	6	7
A				B	C	D	

1. `insert(E)`

0	1	2	3	4	5	6	7
A				B	C	D	**E**

2. `insert(F)`

A	**F**			B	C	D	E

3. `delete(D)`

A	F			B	C	#	E

Example: Linear probing

key	A	B	C	D	E	F
hash	0	4	5	6	5	4

0	1	2	3	4	5	6	7
A				B	C	D	

1. `insert(E)`

0	1	2	3	4	5	6	7
A				B	C	D	**E**

2. `insert(F)`

A	**F**			B	C	D	E

3. `delete(D)`

A	F			B	C	#	E

4. `delete(E)`

A	F			B	C	#	#

Example: Linear probing

key	A	B	C	D	E	F
hash	0	4	5	6	5	4

0	1	2	3	4	5	6	7
A				B	C	D	

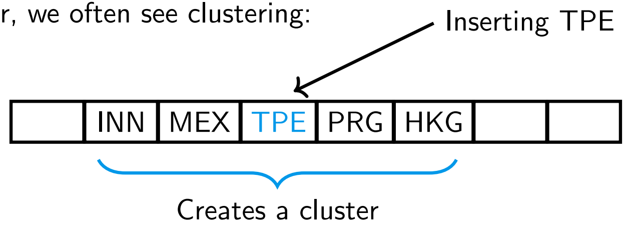
		0	1	2	3	4	5	6	7
1.	`insert(E)`	A				B	C	D	**E**
2.	`insert(F)`	A	**F**			B	C	D	E
3.	`delete(D)`	A	F			B	C	#	E
4.	`delete(E)`	A	F			B	C	#	#
5.	`insert(E)`	A	F			B	C	**E**	#

(first we checked that `E` is not stored, we searched until position 2)

The time complexity and disadvantages

`insert`, `search` and `delete` have the time complexity $\mathcal{O}(1)$. (This is much more difficult to calculate.)

However, we often see clustering:



Primary clusters are clusters caused by entries with the same hash code, but these form even bigger **secondary clusters** when the attempt to insert bumps into another cluster.

Clusters are more likely to get bigger and bigger, even if the load factor is small. To make clustering less likely, use **double hashing**.

Double hashing

Use primary and secondary hash functions `hash1(key)` and `hash2(key)`, respectively.

Insertion: We try the primary position `hash1(key)` first and, if it fails, we try fallback positions:

1. `hash1(key) + 1*hash2(key) mod T`
2. `hash1(key) + 2*hash2(key) mod T`
3. `hash1(key) + 3*hash2(key) mod T`
4. ... (until we find an available space)

`T` is the table size

Double hashing

Use primary and secondary hash functions `hash1(key)` and `hash2(key)`, respectively.

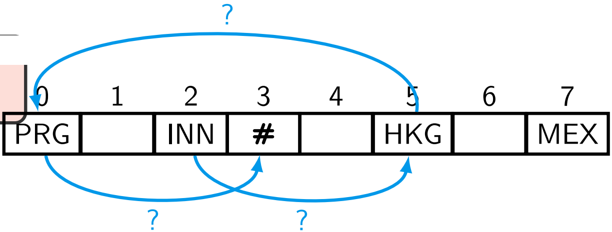
Insertion: We try the primary position `hash1(key)` first and, if it fails, we try fallback positions:

1. `hash1(key) + 1*hash2(key) mod T`
2. `hash1(key) + 2*hash2(key) mod T`
3. `hash1(key) + 3*hash2(key) mod T`
4. …

(until we find an available space)

Example

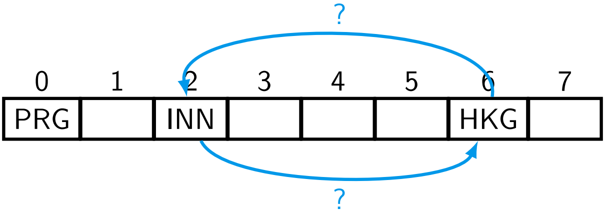
If `key = TPE`,
`hash1(key) = 2`,
`hash2(key) = 3`:



Avoiding short cycles

We can have short cycles!

Consider inserting a `key` such that `hash1(key) = 2` and `hash2(key) = 4`:

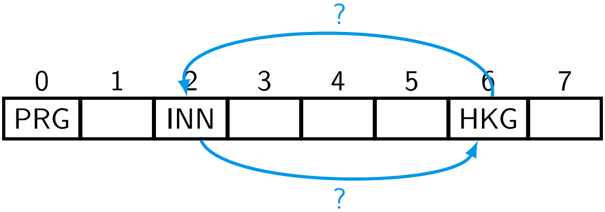


The table size T and `hash2(key)` have to be coprime!

Avoiding short cycles

We can have short cycles!

Consider inserting a `key` such that `hash1(key) = 2` and `hash2(key) = 4`:



The table size T and `hash2(key)` have to be coprime!

Two solutions:

(a) T is a prime number.

(b) $T = 2^k$ and `hash2(key)` is always an odd number. **(preferred)**

What to do if the table is full?

We say that a hash table is **full** if the load factor is more than the maximal load factor, that is,

$$\frac{n}{T} > \lambda.$$

What to do if the table is full?

We say that a hash table is **full** if the load factor is more than the maximal load factor, that is,

$$\frac{n}{T} > \lambda.$$

Rehashing (idea): If the table becomes full after an insertion, allocate a new twice as big table and `insert` all elements from the old table into it.

What to do if the table is full?

We say that a hash table is **full** if the load factor is more than the maximal load factor, that is,

$$\frac{n}{T} > \lambda.$$

Rehashing (idea): If the table becomes full after an insertion, allocate a new twice as big table and `insert` all elements from the old table into it.

Consequences for `insert`:

- the Worst Case time complexity is $\mathcal{O}(n)$ (when rehashing) but
- the *amortized* time complexity is $\mathcal{O}(1)$!

(Rehashing can be used for direct chaining, linear probing, or double hashing and always leads to constant amortized time complexities.)

Summary

Hash tables consist of an array `arr`, a primary hash function `hash1(key)` (and secondary hash function `hash2(key)`.)

All operations are in $\mathcal{O}(1)$ (amortized time) if

1. `hash1` (and `hash2`) computes indexes uniformly,
2. we `rehash` whenever the table becomes full,
3. ($T = 2^k$ for some k, and `hash2` gives odd numbers).

Summary

Hash tables consist of an array `arr`, a primary hash function `hash1(key)` (and secondary hash function `hash2(key)`.)

All operations are in $\mathcal{O}(1)$ (amortized time) if

1. `hash1` (and `hash2`) computes indexes uniformly,
2. we `rehash` whenever the table becomes full,
3. ($T = 2^k$ for some k, and `hash2` gives odd numbers).

Comparison with trees
AVL Trees require keys to be *comparable* and the operations are in $\mathcal{O}(\log n)$, best, worst and average case.

Hash tables, on the other hand, require *good hash functions*.
Then, operations are in $\mathcal{O}(1)$ *amortized* time complexity.

A	B	C	D	E	F	G	H	I	J	K	L	M
$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$
0	1	2	3	4	5	6	7	8	9	10	11	12

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$
13	14	15	16	17	18	19	20	21	22	23	24	25

BHX $= 1 * 26^2 + 7 * 26^1 + 23 * 26^0 = 676 + 182 + 23 = 881$

Hash (BHX) = (881) mod 10 = 1

DXB $= 3 * 26^2 + 23 * 26^1 + 1 * 26^0 = 2{,}028 + 598 + 1 = 2627$

Hash (DXB) = (2627) mod 10 = 7