Hash tables

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`

Maybe you have seen the syntax `arr["Bham"]` in Python, JavaScript, PHP or other programming languages. Even though it looks like those languages allow indexing of arrays by strings, internally it is always implemented by using hash tables.

It is important that every time we compute the index of a key by hash function, we get the same index.

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`.

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.

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)`.

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)`.

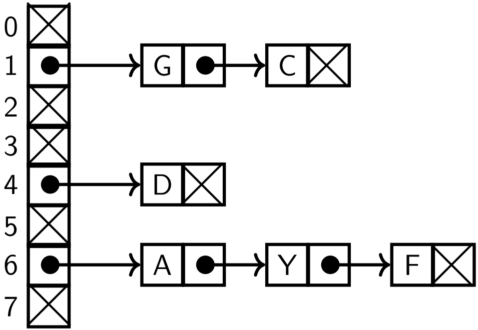
Whereas `lookupValue(key)` returns the value stored on the position given by `key`, `lookup(key)` returns `true` or `false` based on whether `key` is stored in the hash table.

The reason why we explain the simplified situation is because it is easier to illustrate the main ideas this way. However, this simplified situation is also often useful on its own. In Java there is even a class called `HashSet` which works exactly this way.

Note: The only difference between the simplified and unsimplified situations is that, instead of storing the key only, we need to store both the key and the value.

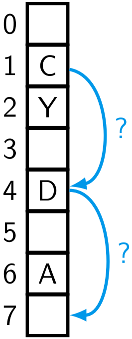
Two types of solutions of hash collisions

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.

The *sticking out* strategies store an extra data structure on each position of the hash table. Those could be linked lists, another hash table, or even something completely different. In the following we only consider one sticking out strategy called *direct chaining*, which uses linked lists to store the values with the same `hash(code)`.

The main idea behind *tucked in* strategies is that, in case of collisions, we find a different position (from a sequence of “fallback“ positions) in the same array. In this module consider the following two tucked in strategies:

- Linear probing
- Double hashing

Example: Direct chaining (= a sticking out strategy)

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

Table size: 10

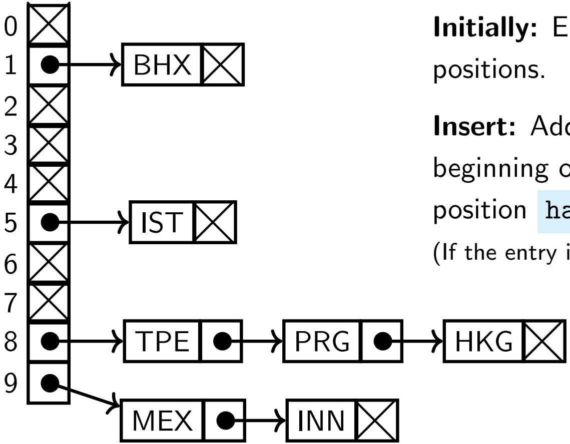
Hash function:

- 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 $0, 1, 2, 3, \ldots$, 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

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.)

To insert, we always first check if the `key` which we are inserting is in the linked list on position `hash(key)`. If it isn't, we the `key` at the beginning of that list.

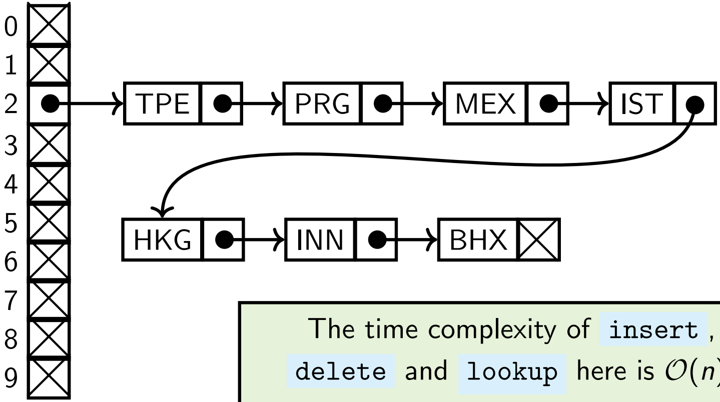
(We are inserting without duplicates.)

To `delete(key)` we delete `key` from the linked list stored on position `hash(key)`, if it is there. Similarly, `lookup(key)` returns `true` / `false` depending on if `key` is stored in the list on position `hash(key)`.

Note: The choice to insert the `key` at the beginning of the list and not at the end is not so important. Inserting at the beginning is more common (probably) because, in practice, the just inserted `key` is more likely to be accessed soon again, as opposed to the key at the end of the list.

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**.

We see that the hash function assigns `2` to all keys. Then, when inserting a new `key` we first check if `key` is stored in the linked list on position `hash(key) = 2`. This requires to go through all the elements already stored in the hash table $\implies \mathcal{O}(n)$ time complexity.

Similarly, `delete` and `lookup` are also in $\mathcal{O}(n)$.

To tackle this, we require to have a **good** hash function which uniformly distributes the keys among positions. In other words, given a random `key`, it ought to have the same probability of being stored on every position.

Remark: Notice that whether a function is good or not also depends on the *distribution* of your data/keys. (You don't want the two most likely keys to share the same hash key, for example.) When the distribution is not known, one assumes that all keys are equally likely.

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.

The load factor represents how full the hash table is. Assuming we have a good hash function, the load factor 0.25 represents 25% probability of getting a collision.

A consequence of having a good hash function is that, if the linked list on position `hash(key)`, for a randomly selected `key`, has *expected* length $\frac{n}{T}$.

The word "expected" has a well-defined meaning in probability theory. Intuitively speaking, it means that the list stored on position `hash(key)` might be longer, it might be shorter, but the length of it will most likely be approximately $\frac{n}{T}$ (for a randomly selected `key`).

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!

If ℓ denotes the length of the linked list on position `hash(key)`, then a random key stored in this linked list is *on average* stored in the middle of this linked list, that is, on position

$$\frac{1}{2}(1+\ell).$$

Next, because we assumed that we have a *good* hash function, the *expected* length of the linked lists on position `hash(key)` is $\frac{n}{T}$. In other words, it is expected that

$$\ell = \frac{n}{T}.$$

Consequently, a successful lookup traverses, *on average*, $\frac{1}{2}(1+\frac{n}{T})$ entries of the linked list stored on position `hash(key)`.

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 $\mathcal{O}(1)$.

To summarise, we made two assumptions:

1. We have a *good hash function*.
2. We assume *maximal load factor*.

A consequence of the first assumption is that the expected length of chains is $\frac{n}{T}$ and the second one is that $\frac{n}{T} \leq \lambda$, for some fixed constant number λ.

By assuming those two conditions, we have computed that the operations of hash tables are all in $\mathcal{O}(1)$.

Whether a hash function is *good* depends on the distribution of the data. On the other hand, making sure that the load factor is bounded by some λ can be done automatically. We will show how to do this later on. The consequence of our approach will be that the constant time complexity will be (only) *amortized*.

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:

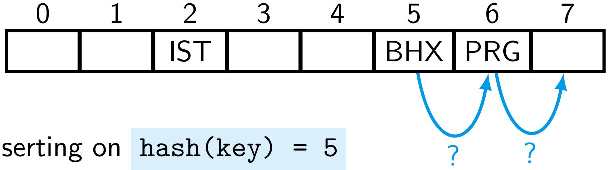
- Linear probing
- Double hashing

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`, ...

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`:

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

? ? ?

Replace with **#**

Note that in the step 1. we skip over all tombstones.

This means, when initialising an empty hash table we denote *all* positions as *empty*. Then, after a sequence of insertions and deletions some positions might be denoted as occupied or empty.

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**

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)`

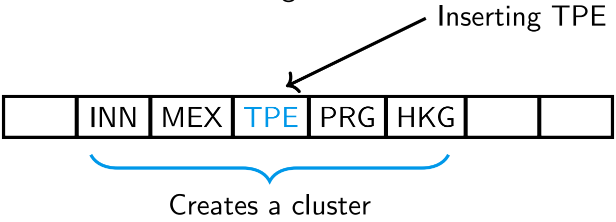
A	F			B	C	**E**	#

(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:



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:

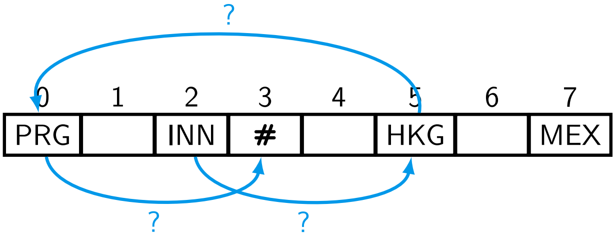
1. `hash1(key) + 1*hash2(key) mod T`
2. `hash1(key) + 2*hash2(key) mod T`
3. `hash1(key) + 3*hash2(key) mod T`
4. ...

`T` is the table size

(until we find an available space)

Example

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



Double hashing is an improvement of linear probing. The only difference is that every `key` has a different sequence of "fallback" positions given by the secondary hash function.

Except for how we calculate the fallback positions, all the operations (`insert`, `delete` and `lookup`) work the same way; we use tombstones to mark deleted keys, when looking up we skip over those tombstones etc.

Linear probing's fallback positions are:

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

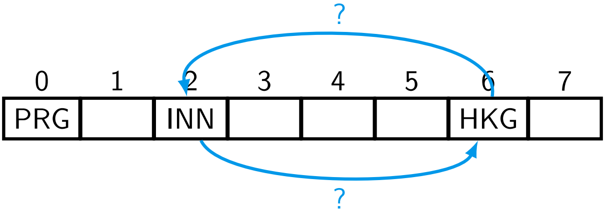
whereas double hashing's fallback positions are:

`hash1(key) + i*hash2(key) mod T` for `i = 1, 2, 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!

Two solutions:

(a) T is a prime number.

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

Maths break:

- Two numbers a and b are said to be *coprime* if no number, other than 1, divides both a and b
- *Prime numbers* are the numbers which are divisible only by 1 and themselves.

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.)

This combines well with our extra assumption that $T = 2^k$ in order to avoid short cycles (from slide 19). If we start from an empty hash table of such size (for example, we initially have $T = 2^3 = 8$), then doubling the size always ensures that $T = 2^k$ for some (natural) number k.

Remark: If we double the size of the hash table, we also need to change the (primary) hash function to make sure that it is *good* again. In practice, `hash(key)` is usually compute as

`bigHash(key) mod T` (where `bigHash` computes a "big" hashcode).

Then, after doubling the size of our hash table we only modify `hash(key)` as follows

`bigHash(key) mod 2*T`.

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.

We see that no matter whether we use direct chaining, linear probing or double hashing to deal with collisions, either way, all operations will be in the constant *amortized* time complexity. The only reason why double hashing is the best is that the constant, which is hidden by the big-$\mathcal{O}$ is the best in case of double hashing. (Because `allocate_memory` usually has a large constant.)

Remark: It is desirable to keep track of how many tombstones are there in the hash table. If this number exceeds some threshold, we also rehash but without doubling the size. (If it was too many tombstones, we might even decrease the size of the hash table by one half.) As a consequence also `delete` is also $\mathcal{O}(1)$ *amortized* time complexity.