Hash tables

Basic idea

Goal: We would like to be able to *calculate* the location of our search target without having to actually search for it.

A **hash function** `hash(key)` computes the location from a `key`

Example: Storing international dialing codes in an array:

```
dialCode[hash("UK")] = "+44"
```

In practice, we will need an ADT, rather than a simple array:

```
dialCode.put("UK", "+44")
```

However, some programming languages with built-in hash ADTs, like **Python**, extend the array syntax:

```
dialCode["UK"] = "+44"
```

Maybe you have seen the syntax `dialCode["UK"]` 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 a hash function, we always get the same index.

Example: Storing students' assignments in $O(1)$

In Canvas, say we store students' submissions in a hash table:

- `value` = assignment submission
- `key` = student's ID number

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

Allocate an array `submissions` of size 10^7 and choose the hashcode to be the identity function: `hash(s)` returns s.

Then, to store an assignment:

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

This is in $O(1)$ but **VERY** memory inefficient!

Even if we only need to store assignments of 500 students, we still allocate an array of size 10^7

Example: Hash function based on the size of the array

Allocate an array `arr` of size 500 and compute `hash(s)` as

`s mod 500`.

Now `hash(s)` is one of `0`, `1`, `2`, ... `submissions.length-1`.

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

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

for two different keys `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

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 (at least) the operations:

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

NOTE: 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)`.

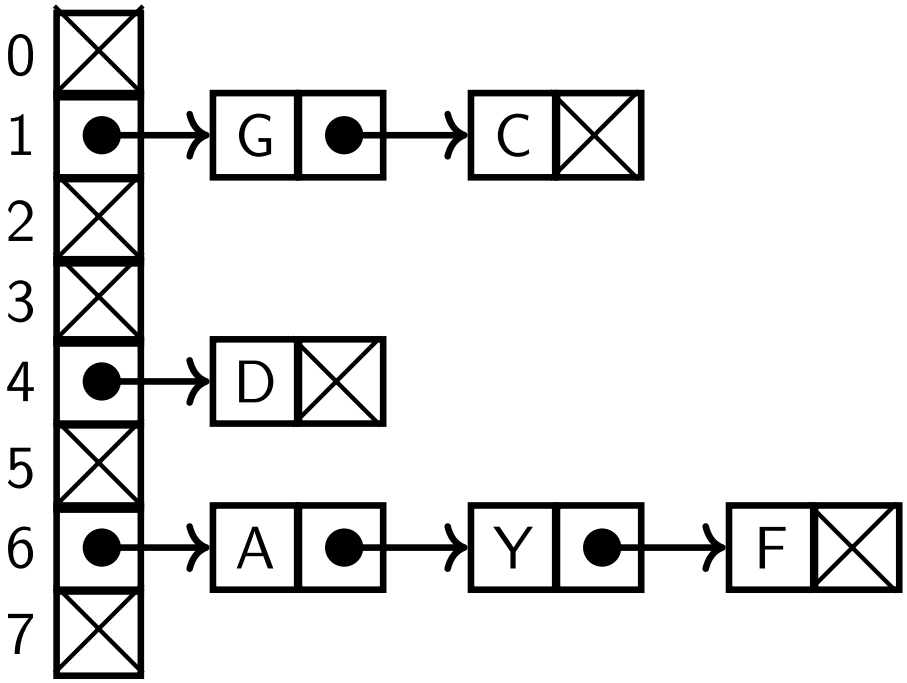
Whereas `lookupValue(key)` returns the value stored on the position given by `key`, `lookup(key)` returns `true` or `false` based on whether `key` is stored in the hash table.

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.

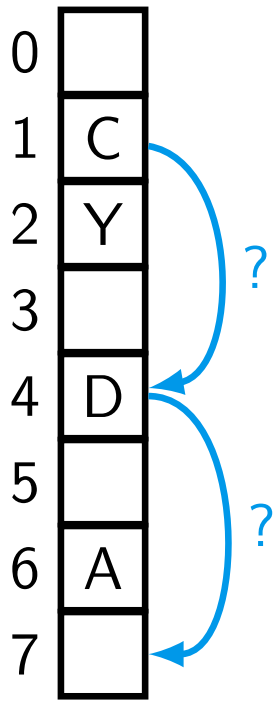
Two types of solutions of hash collisions

Chaining strategy



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

Open Addressing strategy



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

The *chaining* 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 chaining strategy, called *direct chaining*, which uses linked lists to store the values with the same `hash(code)`.

The main idea behind *open addressing* strategies is that, in case of collisions, we find a different address (from a sequence of "fallback" addresses) in the same array for the colliding value to be stored, that is, an address which is currently unused, or *"open"*, hence the name *open addressing*. In this module we consider the following two open addressing strategies:

- Linear probing
- Double hashing