Direct chaining (= a chaining strategy)

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

Table size: 10

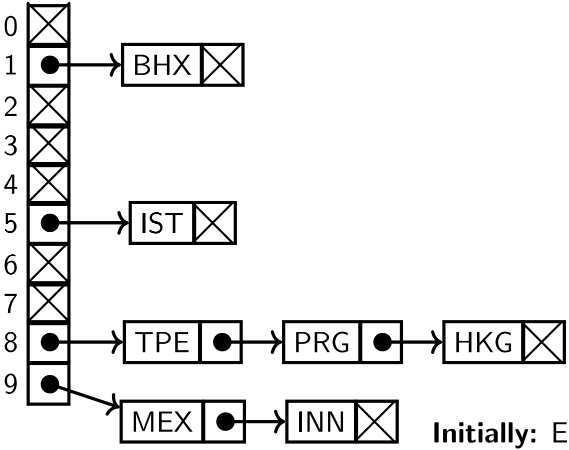
Hash function:

- 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

Direct chaining

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



Initially: Empty lists on all positions.

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 insert the `key` at the beginning of that list.

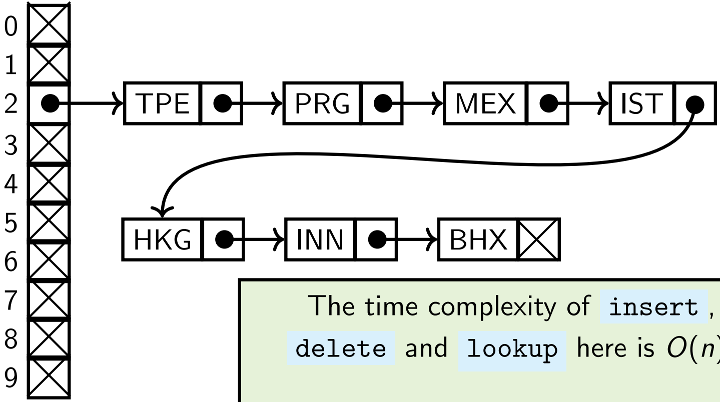
(We are inserting without duplicates.)

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.

Bad hash functions

`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 $O(n)$!

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 $\implies O(n)$ time complexity.

Similarly, `delete` and `lookup` are also in $O(n)$.

To tackle this, we need 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:

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

Unsuccessful lookup of `key`:

- `key` is not in the table.
- Location `hash(key)` stores $\frac{n}{T}$ entries, *on average*.
- $\implies$ 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 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 it's length 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 $k = \frac{n}{T}$ entries on average.
- On average, A linear search in a linked list of k elements takes $\frac{1}{k}(1 + 2 + \cdots + k) = \frac{k(k+1)}{2k} = \frac{(k+1)}{2}$ comparisons

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

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, insert `key` at the beginning of the list stored on `hash(key)`.

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

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

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

To summarise, we made two assumptions:

1. We have a *good hash function*.
2. We assume a *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 $O(1)$.

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 “chaining” strategies

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

We will take a look at two **open addressing strategies** which avoid those problems:

- Linear probing
- Double hashing