ANSWERS

A3.5 Hash tables

Data structures · A level · OCR H446 1.4.2, AQA 7517 4.2.1.4, Eduqas A500QS 1.1 · about 20 min

BugBotLab

What this lesson is about

Hashing functions, collisions, rehashing by probing, chaining and load factor: finding markers fast.

- The idea
- Good hashing functions
- Collisions
- Load factor and resizing

Questions

6 marks in all

1. A hash table has 13 slots and the hashing function key MOD 13. In which slot does key 57 belong? [1 mark]

> **Answer:** 5. 57 = 4 x 13 + 5, so the remainder is 5.

2. This program builds a hash table with linear probing. What does it print? [1 mark]

```
SIZE = 7
table = [None] * SIZE
for key in [10, 17, 3, 24]:
    slot = key % SIZE
    while table[slot] is not None:
        slot = (slot + 1) % SIZE
    table[slot] = key
print(table)
```

> **Answer:**
>
> `[None, None, None, 10, 17, 3, 24]`
>
> All four keys hash to 3, so each collides and probes on to the next free slot: 3, 4, 5 and 6.

3. Which are properties of a good hashing function? [1 mark]

Tick every answer that is true.

- ☐ **A** It is quick to calculate
- ☐ **B** The same key always gives the same slot
- ☐ **C** It spreads keys evenly across the table
- ☐ **D** It gives a different slot for every possible key
- ☐ **E** It keeps the keys in sorted order

> **Answer:** A, B, C. There are more possible keys than slots, so collisions cannot be avoided, and hashing scatters keys rather than sorting them.

4. In a hash table with open addressing, why is a deleted slot marked as deleted instead of being emptied? [1 mark]

- ☐ **A** A search for a key further along the probe path would stop at the empty slot and wrongly report it missing
- ☐ **B** Emptying a slot is slower than marking it
- ☐ **C** The hashing function cannot give an empty slot
- ☐ **D** It stops the load factor changing

> **Answer:** A. A search stops at an empty slot, so emptying a slot in the middle of a probe path would hide the keys beyond it.

5. What is it called when two different keys hash to the same slot? [1 mark]

> **Answer:** collision. Collisions are handled by rehashing to another slot, or by chaining.

6. A hash table's load factor has risen above its threshold. What is usually done? [1 mark]

- ☐ **A** A larger table is made and every item is rehashed into it
- ☐ **B** The oldest items are deleted
- ☐ **C** The table is sorted
- ☐ **D** The hashing function is changed to key MOD 2

> **Answer:** A. Each item's slot depends on the table size, so all of them must be hashed again into the bigger table.

The task: markers in a hash table

`MARKERS` lists six markers as `(id, x, y)`: the marker's id and where it is, in cm from the robot's start. `table` is a list of 11 slots, each `None` until used. Build the hash table yourself, without a Python dictionary. 1. Write `put(key, x, y)`: the home slot is `key % SIZE`. If that slot is taken, probe the next slot, wrapping from 10 back to 0, until one is empty. Store the tuple `(key, x, y)` there and print `put <key> in slot <slot>`. 2. Write `get(key)`: follow the same path. Count every slot you look at as one probe, including an empty slot that ends the search. If you find the key, print `found <key> at slot <slot> after <probes> probes` and return its tuple. If you reach an empty slot, print `<key> not found after <probes> probes` and return `None`. 3. Put the six markers in the order listed. Then call `get(42)`, which is not in the table. 4. Call `get(31)`, and drive to marker 31: right by its x, then forward by its y.

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

SIZE = 11
# (marker id, x, y): where each marker is, in cm from the start
MARKERS = [(14, 20, 40), (25, 60, 20), (47, 40, 60), (9, 0, 50), (20, 10, 30), (31, 30,
45)]

table = [None] * SIZE

def put(key, x, y):
    pass

def get(key):
    pass
```

The hint students can ask for: The home slot is the key MOD 11. If that slot is taken, try the next one, wrapping from 10 back to 0. A search follows exactly the same path and counts every slot it looks at: it succeeds when it finds the key, and gives up at an empty slot.

A solution

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

SIZE = 11
# (marker id, x, y): where each marker is, in cm from the start
MARKERS = [(14, 20, 40), (25, 60, 20), (47, 40, 60), (9, 0, 50), (20, 10, 30), (31, 30,
45)]

table = [None] * SIZE

def put(key, x, y):
    slot = key % SIZE
    while table[slot] is not None:
        slot = (slot + 1) % SIZE
    table[slot] = (key, x, y)
    print("put", key, "in slot", slot)

def get(key):
    slot = key % SIZE
    probes = 0
```

```
    while probes < SIZE:
        probes = probes + 1
        if table[slot] is None:
            print(key, "not found after", probes, "probes")
            return None
        if table[slot][0] == key:
            print("found", key, "at slot", slot, "after", probes, "probes")
            return table[slot]
        slot = (slot + 1) % SIZE
    print(key, "not found after", probes, "probes")
    return None

for key, x, y in MARKERS:
    put(key, x, y)

get(42)
record = get(31)
right(50, distance=record[1])
forward(50, distance=record[2])
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.