WORKSHEET

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

Name _____ Class _____ Date _____

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]

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)
```

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

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

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

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

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
```

Plan your program here, then type it in and press Run.

Do it on the robot

www.bugbotlab.com/learn/a3-5-hash-tables/

The simulator checks it and tells you when it passes. Nothing to install, no account.

Challenges

1. Before running your program, work out on paper which slot each marker goes in. Which markers collided, and how many probes did each need?
2. Change `put` to probe 3 slots on each time instead of 1. Does every key still find a slot? What if the size were 12 instead of 11?
3. Rewrite the table with chaining. How many probes does `get(31)` take now?