CM2035 Topic 5 Hashing

Ian Sanders

October 2025 Session

UNIVERSITY OF LONDON

Recap

In the module videos we have seen

- introduction to hashing,
- extend and rehash,
- direct addressing,
- chained hashing, and
- open address hashing.

In this webinar we will take a slightly different look at these ideas.

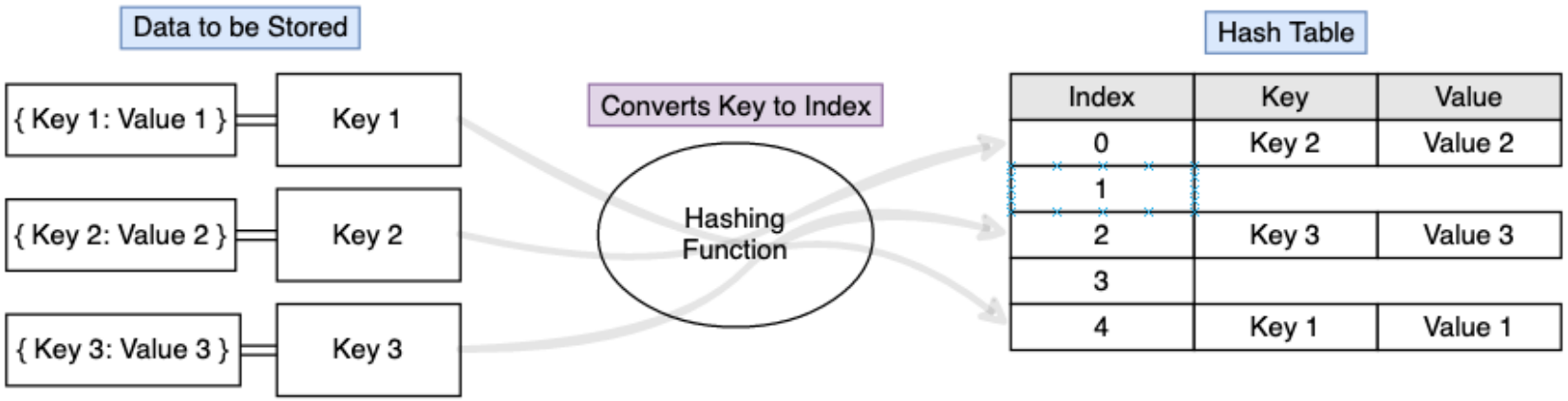
UNIVERSITY OF LONDON

Efficient Data Structures

In many algorithms we want to be able to store information, typically referenced by a key, in a way that aids very fast look up.

dictionary — a data structure that allows efficient *insertion*, *searching* and *deletion*

A *hash table* is an example of such a data structure



UNIVERSITY OF LONDON

So how does hashing work?

- Direct Addressing
- Closed Address Hashing
- Open Address Hashing

UNIVERSITY OF LONDON

Direct addressing

Assign a *unique* array position to *every possible key* which might be used in our application.

Consider a small company in a small building where each employee has their own office. The office number is the key and the value is the employee's name (or possibly the purpose of the room).

key	value
0	Bob
1	
2	Fred
3	Lucy
4	Printer
5	Monde
6	
7	Bill
8	Susan
9	Tea Room

UNIVERSITY OF LONDON

Direct addressing continued

Searching, inserting and deleting can be done very quickly and easily.
Works well if the "key space" is quite small.
A problem is that the "key space" could be very large and only a very small percentage of the keys would actually be used.
Lots of wasted space!
Often simply not feasible to allocate that amount of space.
So we need a better approach

Hashing

Map some key from the *key space* onto an integer which is an index into a *hash table*
The mapping is done using a *hash function*
The result of the mapping is called the *hash code* of the key.
For some hash function h, the key k *hashes* to slot $h(k)$.
Possible for two different keys to give the same hash code — a *collision*.

UNIVERSITY OF LONDON

Example

Key space is pets' names and the hash table is of size 9

A possible *hash function*, is
$h(name) = (\sum_{i=1}^{length(name)} value(name_i)) \mod 9$

$h(\text{“spot”}) = (19 + 16 + 15 + 20) \mod 9 = 70 \mod 9 = 7$
$h(\text{“fred”}) = (6 + 18 + 5 + 4) \mod 9 = 33 \mod 9 = 6$
$h(\text{“mutt”}) = (19 + 16 + 15 + 20) \mod 9 = 74 \mod 9 = 2$
$h(\text{“jeff”}) = (10 + 5 + 6 + 6) \mod 9 = 27 \mod 9 = 0$
$h(\text{“pina”}) = (16 + 9 + 14 + 1) \mod 9 = 40 \mod 9 = 4$
$h(\text{“flopsy”}) = (6 + 12 + 15 + 16 + 19 + 25) \mod 9 = 93 \mod 9 = 3$
$h(\text{“becca”}) = (2 + 5 + 3 + 3 + 1) \mod 9 = 14 \mod 9 = 5$
$h(\text{“fido”}) = (6 + 9 + 4 + 15) \mod 9 = 34 \mod 9 = 7.$

UNIVERSITY OF LONDON

hash code	key(s)
0	“jeff”
1	
2	“mutt”
3	“flopsy”
4	“pina”
5	“becca”
6	“fred”
7	“spot”, “fido”
8	

Some questions:

1. What is a good hash function?
2. How do we handle collisions?

A good hash function should

1. spread the hash codes for the input keys over the range of available indices
2. give as few collisions as possible and
3. is often dependent on the application for which the hash table is being designed.

There are two common ways of dealing with collisions

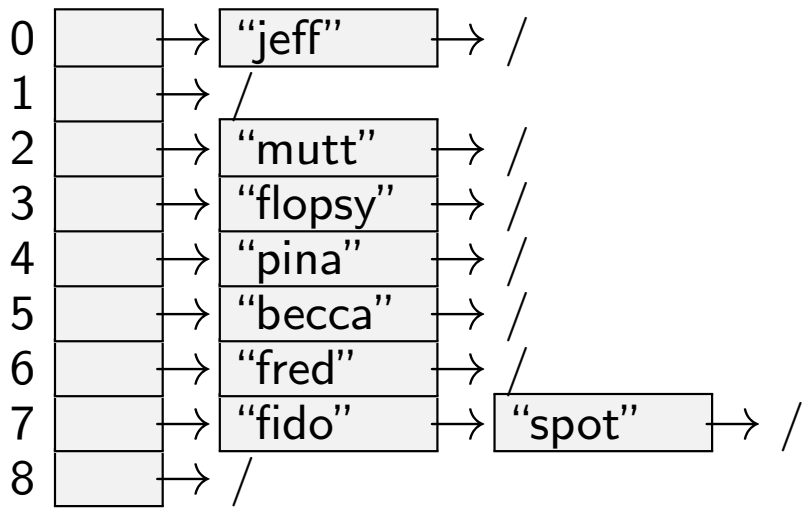
- *closed address* hashing also called *chained* hashing
- *open address* hashing

UNIVERSITY OF LONDON

Closed address hashing

Each position in the table is a pointer to the head of a linked list. Initially all the lists are empty, i.e. the pointer to the head of the list is nil.

To insert a key, we first calculate its hash code to get the index into the hash table and then we insert a node representing the key into the beginning of the linked list in that position.



The cost of an insertion is thus clearly O(1)
The cost of searching and the cost of deletion in the best case are both O(1)
In the worst case searching and deletion are both dependent on the length of the list in the slot being considered.
If the hash table is being used to stored n items in h linked lists then the *load factor* of the table, α, is n/h and this is the *average* number of items in each linked list.
If we have chosen a good hash function for our table then the actual number of items in each list will be very close to α.
Average search cost is $O(1 + \alpha)$.
The average cost of a deletion is also $O(1 + \alpha)$.
Worst case for searching and deletion is if all of the keys hash to the same slot (a poor hash function) and are stored in the same linked list — $O(n)$.

Open address hashing

The keys are actually stored in the slots in the hash table not in a linked list.
Each slot holds a key or a *nil*.
If a collision occurs then we *rehash*.
The simplest rehashing is called *linear probing*.
If there is a collision at position/index j then $rehash(j) = (j + 1) \mod h$

UNIVERSITY OF LONDON

Example: Suppose we have inserted all the keys up to “becca” then our hash table would be

0	“jeff”
1	nil
2	“mutt”
3	“flopsy”
4	“pina”
5	“becca”
6	“fred”
7	“spot”
8	nil

Now we look at inserting "fido"

$h(\text{"fido"}) = (6 + 9 + 4 + 15) \mod 9 = 34 \mod 9 = 7.$

This is a collision! In slot 7.

So we would rehash....

$rehash(7) = (7 + 1) \mod 9 = 8 \mod 9 = 8$

And our table becomes....

0	"jeff"
1	nil
2	"mutt"
3	"flopsy"
4	"pina"
5	"becca"
6	"fred"
7	"spot"
8	"fido"

UNIVERSITY OF LONDON

Complexity of open address hashing

Best case for searching is O(1)
Ditto for insertion.
Generally performance is good when searching a hash table with a low load factor.
The average number of slots which have to be accessed approaches $\sqrt{n}$ if the load factor is high i.e. close to 1.

We need to be careful when doing deletions from the hash table as we might be trying to delete a key which has caused a subsequent key to be rehashed to another slot in the table.
For example, suppose we had the following keys and hash codes
A, 3 B, 4 C, 5 D, 3 E, 7 F, 3 G, 0 H, 1 I, 2
Then we would get the following table after insertion with rehashing where necessary.

0	G
1	H
2	I
3	A
4	B
5	C
6	D
7	E
8	F
9	nil

UNIVERSITY OF LONDON

If A was deleted and its entry replaced by “nil” then the chain would be broken and a subsequent search for D or F would say that neither were in the table.

0	G
1	H
2	I
3	nil
4	B
5	C
6	D
7	E
8	F
9	nil

UNIVERSITY OF LONDON

If an "obsolete" marker was used then we would continue searching until a "nil" was found.

0	G
1	H
2	I
3	obsolete
4	B
5	C
6	D
7	E
8	F
9	nil

If we now insert another key, say P, which hashes to 3 we would get ...

0	G
1	H
2	I
3	P
4	B
5	C
6	D
7	E
8	F
9	nil

Summary

Direct addressing Unique array position for every possible key.
Searching, inserting and deleting are O(1).
Works well if the "key space" is quite small.
If "key space" is very large, could have wasted space.

UNIVERSITY OF LONDON

Summary

Chained hashing (Closed address hashing) Each position in the table is a pointer to the head of a linked list.
The cost of an insertion is O(1)
The best case of searching and deletion is O(1)
Average search cost is $O(1+\alpha)$.
Average deletion cost is $O(1+\alpha)$.
Worst case for searching and deletion is $O(n)$.

UNIVERSITY OF LONDON

Summary

Open address hashing The keys are actually stored in the slots in the hash table.
If a collision occurs then we *rehash*.
The simplest rehashing is called *linear probing*.
Best case for insertion, searching and deletion is $O(1)$.
Generally performance is good when searching a hash table with a low load factor.

UNIVERSITY OF LONDON

Finally

I will post the webinar recording and the slides on my website as soon as I can.
Thanks and have a good weekend.

UNIVERSITY OF LONDON