Hashing Algorithms

See: Koffman & Wolfgang chapter 9

Retrieving data based on a key

- Retrieving data based on a key is basic to many applications, and must be done as fast as possible.
  - Retrieving data based on a key is precisely what we do in a searching algorithm, so how can we improve the search to retrieve data even faster !?
- We have looked at a searching algorithm which was O(n)
  - sequential (aka linear) search
- And at a much improved one which was O(log n)
  - Binary search
- But we would like to go *directly* to a position in the array *without having to do any searching at all*
  - so *the time taken to figure out which array position to go to won't change even if we have an array 100 times the size*
  - this would be an O(1) method of accessing data (i.e constant time)
- To do this, we need to 'index' on the key,
  - store the data items at index positions in an array that we can access 'instantly'
  - for example, that is why a DBMS would create an index on columns which are often used as search keys when data must be retrieved from a table.

Example – accessing student data

- Suppose we need an application that can find data about each Tallaght student really fast.
  - We can use some key value in the student data – for example, the X-number of the student.
  - We would like to have a way of getting very quickly to the correct data for any student
- Solution proposed:
  - Have an array, each array position holds student info about a student with a different X number.
  - go ‘directly’ to an index position in the array which has been determined from the X-number.

Example: accessing data for COMP3

X00059970
X00062525
X00060014
X00059519
X00077579
X00062919
X00077087
X00062702
X00060428
X00059366
X00060473
X00062007
X00062933
X00061072
X00046391
X00060877

X00062882
X00061603
X00070581
X00056879
X00059851
X00060479
X00059739
X00062842
X00058586
X00016460
X00062891
X00060835
X00047504
X00012115
X00047587
X00050657
X00077135

- How could we use the X-number as an index into an array?
  - Maybe strip off the 'X' character, and use the rest as an integer index into the array?
- How large an array would we need to do this?
  - Well, what's the highest X-no we can see here? 77579?
- Ideally, how large an array would we like to use?
- There are only 33 students here – an array of 77,579 with 77,546 empty spaces seems a tad like overkill !!
  - ideally an array of size 33
- Any obvious ways to get the answers to the last 2 questions closer together???
  - Subtract the lowest no – so that X000012115 goes to position 0
  - The highest index required is still 65431 (77546 – 12115)

COMP3 X-nos. Possible Hashing Algorithms

ID Number	strip X	mod 33	mod 100	mod 200	mod 500	mod 700
X00059970	59970	9	70	170	470	470
X00062525	62525	23	25	125	25	225
X00060014	60014	20	14	14	14	514
X00059519	59519	20	19	119	19	19
X00077579	77579	29	79	179	79	579
X00062919	62919	21	19	119	419	619
X00077087	77087	32	87	87	87	87
X00062702	62702	2	2	102	202	402
X00060428	60428	5	28	28	428	228
X00059366	59366	32	66	166	366	566
X00060473	60473	17	73	73	473	273
X00062007	62007	0	7	7	7	407
X00062933	62933	2	33	133	433	633
X00061072	61072	22	72	72	72	172
X00046391	46391	26	91	191	391	191
X00060877	60877	25	77	77	377	677
X00062882	62882	17	82	82	382	582
X00061603	61603	25	3	3	103	3
X00070581	70581	27	81	181	81	581
X00056879	56879	20	79	79	379	179
X00059851	59851	22	51	51	351	351
X00060479	60479	23	79	79	479	279
X00059739	59739	9	39	139	239	239
X00062842	62842	10	42	42	342	542
X00058586	58586	11	86	186	86	486
X00016460	16460	26	60	60	460	360
X00062891	62891	26	91	91	391	591
X00060835	60835	16	35	35	335	635
X00047504	47504	17	4	104	4	604
X00012115	12115	4	15	115	115	215
X00047587	47587	1	87	187	87	687
X00050657	50657	2	57	57	157	257
X00077135	77135	14	35	135	135	135

Hashing algorithms

- A hashing algorithm (or function) provides a way to map from a key value to an index position.
  - We can use the algorithm initially to figure out where to insert the key-value in the array.
  - And use the same algorithm to figure out where to search for it.
- Our first attempt at figuring an index position in the array from the X-no was simply to strip off the X and use the rest as an integer index.
  - This is a simple hashing algorithm.
  - it maps from the key value (the X-number) to an index position
  - If the X number is not found at that index position, then it's not stored in the array.
- But this algorithm would lead to an extremely sparse array (ie an array with a lot of 'holes' in it where no data has been inserted)
  - Good hashing algorithms will hash into a much tighter space than this.
- If we have inside knowledge about the type of data we are storing, we may be able to tighten up the size of the array a bit.
  - Maybe X-numbers always have an integer part greater than 10000? That would get us down to an array of size 67,579
  - Maybe X-number are always even? – that would halve the size of the array?
- And if we are prepared for the occasional foul-up, we could reduce the size of the array dramatically even if we have no inside info about the data
  - e.g hash to an index position between 0 and 699 by taking modulus 700 of the integer key.
  - Or hash to an array of size 365 (366?) by forgetting about X-numbers and taking the birthday of a student as the index position
  - The foul-ups happen when 2 keys hash to the same position in the array. This is called a hash collision.

Good Hashing Algorithms

- It is not usually possibly to devise a ‘perfect’ hashing algorithm
- The best ones will hash to an array of a relatively small size
  - Relatively means compared with the number of data items we intend to store in the array
- They will also cause very few ‘hash collisions’
  - Remember, a hash collision occurs when 2 data items hash to the same position in the array
- Looks like using an algorithm that says

  Strip the X number

  Convert rest of string to integer

  Return modulus 700 of the integer

Will do the job for our Comp-3 example. But could we guarantee that if 3 more students join the class there will still be no hash collisions?

It might help if we chose 701 instead of 700 – hash tables with a size that is a prime number turn out to give better results

Hash Collisions

- There are hashing algorithms much more sophisticated that just taking the modulus
  - They guarantee the best possible spread of data values through the array, not bunched up at any particular position …
  - … so they have the best possible performance in avoiding hash collisions
  - But still can't *guarantee* no hash collision
  - They usually involve looking at each 'bit' of the bytes storing the integer separately.
- Sophisticated hashing techniques will include sophisticated ways of dealing with hash collisions
  - But they still fall into 2 basic categories

Dealing with Hash Collisions 1: open addressing

- **If when you attempt to hash a new value into the array you find that the array position is already 'taken' then 're-hash' to another position.**
  - Must guarantee that the re-hashing algorithm will cover the whole array if it is applied enough times
    - So an empty position for the new entry will eventually be found
  - For example, a very simple re-hashing algorithm would say

    add 1 to the index you last tried

    wrap to the beginning if you are at the last index and it's already taken.
- The same re-hashing algorithm will be applied when searching for a value in the array
  - If there is a value already at the first position, and its not the one we're looking for, then re-hash to the next possibly position.
  - We know that the value isn't in the array if we come to an empty position without finding it. (Because that's were it would have been put)

Hash Collision management: Some draw backs of open addressing

Problem 1:

We said: We know that the value isn't in the array if we come to an empty position without finding it. (Because that's were it would have been put)

But suppose that the list is full even though the element we want isn't there?

- how will we know when to stop looking for the key?
- We will be in an infinite loop situation.

- Solution?

We could *extend the size of the hash table* after an insertion if the occupancy ratio exceeds a specified thresh-hold

- That's a technique sometimes used
- A big overhead when it happens, as everything must be re-indexed!

Problem 2:

What happens when we delete an item? If another item entered in the array had collided with it originally, then when we come to the (new) empty position, we cant assume that that means the element we are looking for isn't there!

Solution?

We could use a dummy value in such a position?

- the array will need to be a lot bigger over time as items are deleted and replaced by dummy values

We could extend the size of the hash table as above to cope with the growth of the table,

we could use the space to add another item later

- We would have to check that it wasn't in the array already.

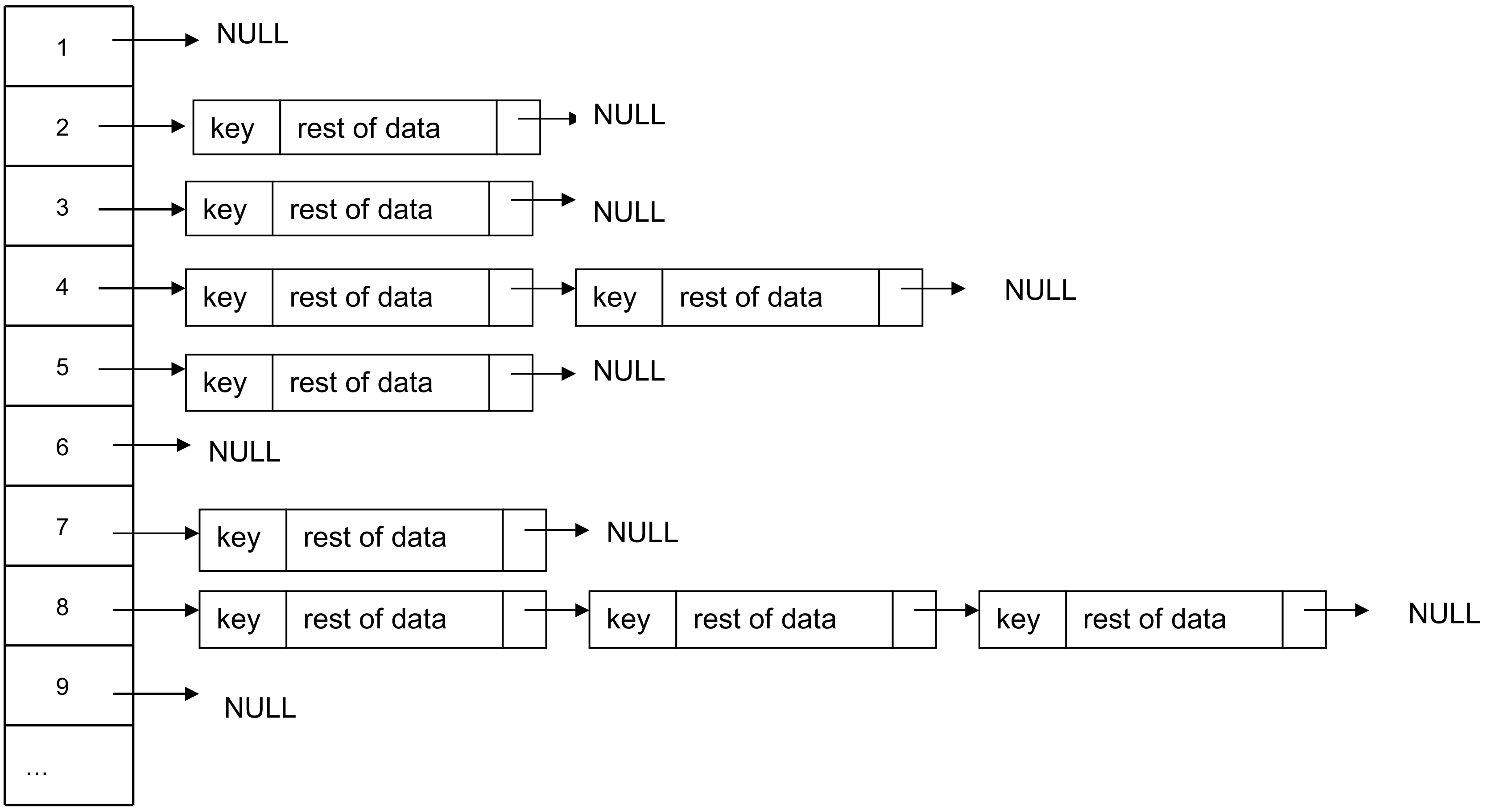
Dealing with Hash Collisions 2: Buckets and Chaining

- The idea of a ‘hash bucket’ is that instead of storing a single value at any position in the array, we store all the values that hash to that position
  - So we will have to do some sort of search through the bucket to find the one we want
  - But it will only be a search of a very few things (the ones that ‘collided’ at that position)

- How do we implement the bucket?
  - It could be any container type data structure

- Commonly, the data structure for the bucket is a linked list
  - Each index position stores a pointer to the head of a list
  - Most lists are either empty or have only one entry (assuming a good hashing algorithm was used)
  - But sometimes there might be 2 or 3 entries in the list (or even more)
  - We call this method of dealing with collisions ‘**chaining**’

Using a linked list to handle hash collisions



Index positions 2, 3, 5 and 7 have one entry

1, 6 and 9 are empty

4 and 8 handle hash collisions

Pros and cons of hashing algorithms: some after thoughts!

1. We can retrieve data in O(1) time 'most of the time'
   – As long as we have a good hashing algorithm for the data

2. Worst case might be O(n)
   – If all items collide at the same position!

- Hashing is only a good technique if we are looking for an 'exact key' match
  – For example, it wouldn't be a good algorithm for finding all the data items with the key in a certain range.
  – Or for finding a data item with a value *approximately equal* to something