Algorithms and Data Structures

Charles A. Wuethrich

Bauhaus-University Weimar - CogVis/MMC

May 2, 2019

Tables

- In general, a table is nothing else than a 2-dimensional array (or 3-D, or n-D) of data
- Data can be quite complex
- Access like in arrays is ruled through 2-dimensional indexes (up to n-D)

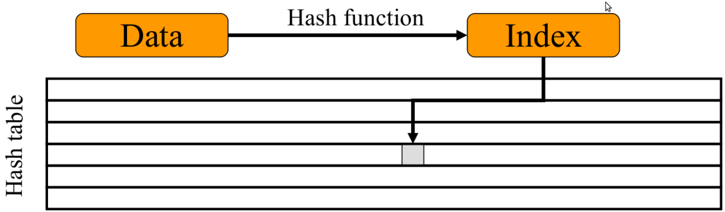
Hash Tables

- Problem with arrays and lists: access not so easy if I search for a particular data and do not know its index
- Example: I know the data I want to look for, but do not know where it is

  – I have to retrieve and read sequentially all data until I find the data I am looking for
- To overcome this problem, Hash Tables were invented
- Hash tables are used to store data in a large table

Hash Tables

- A hash table is nothing else than a big table, where the access is ruled by a function called hash function
- For each data that has to be put in the table, the hash function computes a unique index that is used to insert the data in the table

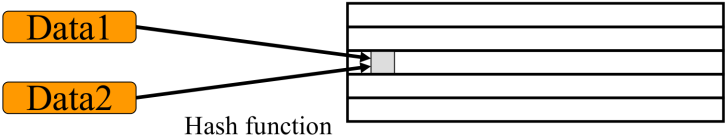


Hash Functions

- Hash tables are usually much bigger than the foreseen data I want to store in them
- A hash function H is a transformation that takes a variable-size input m and returns a fixed-size string, which is called the hash value h (that is, $h = H(m)$).
- A fundamental property of all hash functions is that if two hashes (according to the same function) are different, then the two inputs were different in some way
- The hash value one gets is usually used to compute the index at the hash table where the data will be stored (or, if it is an integer, to compute it directly)

Hash Functions

- However, while hash functions are injective, they are not a bijective function: I.e. given two inputs, the output of the hash function is not necessarily different
- If given two different data, the hash function computes the same value, then the hash function is said to generate a collision
- Obviously, it is important that the hash functions generates as little collisions as possible



Hash Functions

- So what are good hash functions?
- Consider a table containing an index , a name, and a telephone number
- To search this table for Kurt, I would need to search through all of the names
- Which means, in worst case, to search N elements
- On average, I would have to search N/2 elements

Index	Name	Phone
0	Jason	558293
1	Carl	314276
2	William	834562
3	Angus	169278
4	Robert	995386
5	Kurt	635951
6	Leo	239769
7	Empty	
...	Empty	
N-1	Empty	

Hash Functions

- If I sort the names before, then I can do binary search
- That is, I visit
  – first the place at $N/2$,
  – choose the side according to whether Kurt is smaller or biggerg of what I find at $N/2$
  – then visit $N/4$, ...
- In this case, I still need $log_2 N$ to find the element, but I need a preprocessing step of $NlogN$ to do the sorting
- Any successive search will be $log_2 N$

Index	Name	Phone
0	Angus	169278
1	Carl	314276
2	Jason	558293
3	Kurt	635951
4	Leo	239769
5	Robert	995386
6	William	834562
7	Empty	
...	Empty	
N-1	Empty	

Hash Functions

- The idea behind hash functions is that one could use the data to compute the index I store my data at
- Suppose the global size of my table is 13
- Suppose I simply add the ASCII codes of the letters and obtain S
- And suppose I find the index as the S MOD 13
- Note: I use 13 because I always take prime numbers as the table size (more on it later), and my table size is 13

Index	S	S MOD 13
Angus	541	8
Carl	418	2
Jason	539	6
Kurt	454	12
Leo	320	8
Robert	654	4
William	751	10

Ascii codes

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	*[NULL]*	32	20	*[SPACE]*	64	40	**@**	96	60	**`**
1	1	*[START OF HEADING]*	33	21	**!**	65	41	**A**	97	61	**a**
2	2	*[START OF TEXT]*	34	22	**"**	66	42	**B**	98	62	**b**
3	3	*[END OF TEXT]*	35	23	**#**	67	43	**C**	99	63	**c**
4	4	*[END OF TRANSMISSION]*	36	24	**$**	68	44	**D**	100	64	**d**
5	5	*[ENQUIRY]*	37	25	**%**	69	45	**E**	101	65	**e**
6	6	*[ACKNOWLEDGE]*	38	26	**&**	70	46	**F**	102	66	**f**
7	7	*[BELL]*	39	27	**'**	71	47	**G**	103	67	**g**
8	8	*[BACKSPACE]*	40	28	**(**	72	48	**H**	104	68	**h**
9	9	*[HORIZONTAL TAB]*	41	29	**)**	73	49	**I**	105	69	**i**
10	A	*[LINE FEED]*	42	2A	*****	74	4A	**J**	106	6A	**j**
11	B	*[VERTICAL TAB]*	43	2B	**+**	75	4B	**K**	107	6B	**k**
12	C	*[FORM FEED]*	44	2C	**,**	76	4C	**L**	108	6C	**l**
13	D	*[CARRIAGE RETURN]*	45	2D	**-**	77	4D	**M**	109	6D	**m**
14	E	*[SHIFT OUT]*	46	2E	**.**	78	4E	**N**	110	6E	**n**
15	F	*[SHIFT IN]*	47	2F	**/**	79	4F	**O**	111	6F	**o**
16	10	*[DATA LINK ESCAPE]*	48	30	**0**	80	50	**P**	112	70	**p**
17	11	*[DEVICE CONTROL 1]*	49	31	**1**	81	51	**Q**	113	71	**q**
18	12	*[DEVICE CONTROL 2]*	50	32	**2**	82	52	**R**	114	72	**r**
19	13	*[DEVICE CONTROL 3]*	51	33	**3**	83	53	**S**	115	73	**s**
20	14	*[DEVICE CONTROL 4]*	52	34	**4**	84	54	**T**	116	74	**t**
21	15	*[NEGATIVE ACKNOWLEDGE]*	53	35	**5**	85	55	**U**	117	75	**u**
22	16	*[SYNCHRONOUS IDLE]*	54	36	**6**	86	56	**V**	118	76	**v**
23	17	*[ENG OF TRANS. BLOCK]*	55	37	**7**	87	57	**W**	119	77	**w**
24	18	*[CANCEL]*	56	38	**8**	88	58	**X**	120	78	**x**
25	19	*[END OF MEDIUM]*	57	39	**9**	89	59	**Y**	121	79	**y**
26	1A	*[SUBSTITUTE]*	58	3A	**:**	90	5A	**Z**	122	7A	**z**
27	1B	*[ESCAPE]*	59	3B	**;**	91	5B	**[**	123	7B	**{**
28	1C	*[FILE SEPARATOR]*	60	3C	**<**	92	5C	****	124	7C	**|**
29	1D	*[GROUP SEPARATOR]*	61	3D	**=**	93	5D	**]**	125	7D	**}**
30	1E	*[RECORD SEPARATOR]*	62	3E	**>**	94	5E	**^**	126	7E	**~**
31	1F	*[UNIT SEPARATOR]*	63	3F	**?**	95	5F	**_**	127	7F	*[DEL]*

- Now I fill in my 12 elements table exactly at the place said by S MOD 13

Index	S	S MOD 13
Angus	541	8
Carl	418	2
Jason	539	6
Kurt	454	12
Leo	320	8
Robert	654	4
William	751	10

Hash value	Name	Phone
0		
1		
2	Carl	314276
3		
4	Robert	995386
5		
6	Jason	558293
7		
8	Angus -Leo	169278 - 239769
9		
10	William	834562
11		
12	Kurt	635951

Collision! #?@#%&!!!!

Hash Functions

- And now ?!

- There are alternative strategies that can be taken:
  - The simplest one would be to look if the next cell is free,
  - if it is, store the data there,
  - if not, look forward
- This approach is called *open addressing*
- A smarter version of this approach sets a chained list at each hash table position.
- This approach is called bucketing, because at each position you have a bucket containing some elements instead of a single element

Hash Functions

- Another approach is to use a second hash function H2 for handling collisions *(double hashing)*
- Obviously, the second function needs to be completely different from the first one
- For example, take one plus the bitwise exclusive or of all codes in a name (again taken as all lowercase) mod N, where N is the size of the hash table
- If the first hash function H_1 gives a collision, use the second hash function H_2 to generate a new hash index, and add it (mod N) to the result of H_1

- Why do we usually take hash tables with prime numbers?
- The reasons are deeply rooted in math: given a cyclic group Z_p, with p prime, then any element a of Z_p is a generator of the group
- In other words, the sums a, (a+a), (a+a+a),... are such that after p sums I reobtain a, and the sums have covered ALL elements of Z_p.

Hash Functions

- What are the advantages of a hash table?
- First and foremost, it takes ONE application of the hash function to generically find an element of my table
- Obviously, the more a hash table is filled, the more collisions one gets.
- At some point of filling, it might be a good idea to enlarge the hash table and use another hash function, of course, better suited for the larger table