Hashing #1

CptS 223 - Fall 2017 - Aaron Crandall

Today’s Agenda

- Announcements
- Thing of the day
- Hashing - Love me some hashing!

Announcements

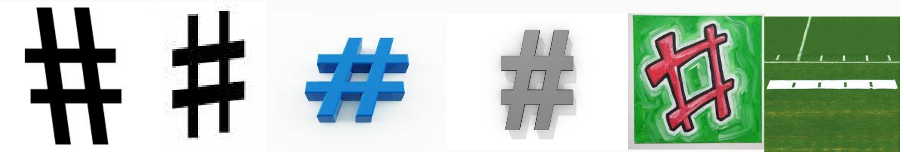
- Will be trying to get the midterm graded for Friday. Who needs sleep?
- How did the job fair go?
- Friday we’ll go over the midterm

TotD - Google's AI Software Learns to Make AI Software

- Google and others think software that learns to learn could take over some work done by AI experts: https://goo.gl/SJYBCP
- … software design a machine-learning system to take a test used to benchmark software that processes language. What it came up with surpassed previously published results from software designed by humans.
- This isn't actually a new concept, nor research area, but oh! the advances!

Hashing - Monarch of O(1) time!

- Hashing stores records in (optimally) O(1) time for:
  - Access, Insert, Search
- Also called:
  - Dictionary (python), Map (C++11 STL), Hash (PERL), Hashtable (Java)
- Just like trees, records are indexed by a key
  - Hash key is normally not the original key value, though!
  - Key can be strings instead of integers, allowing for nice code (python here):
    - myDict = []
    - myDict["mykey"] = record1

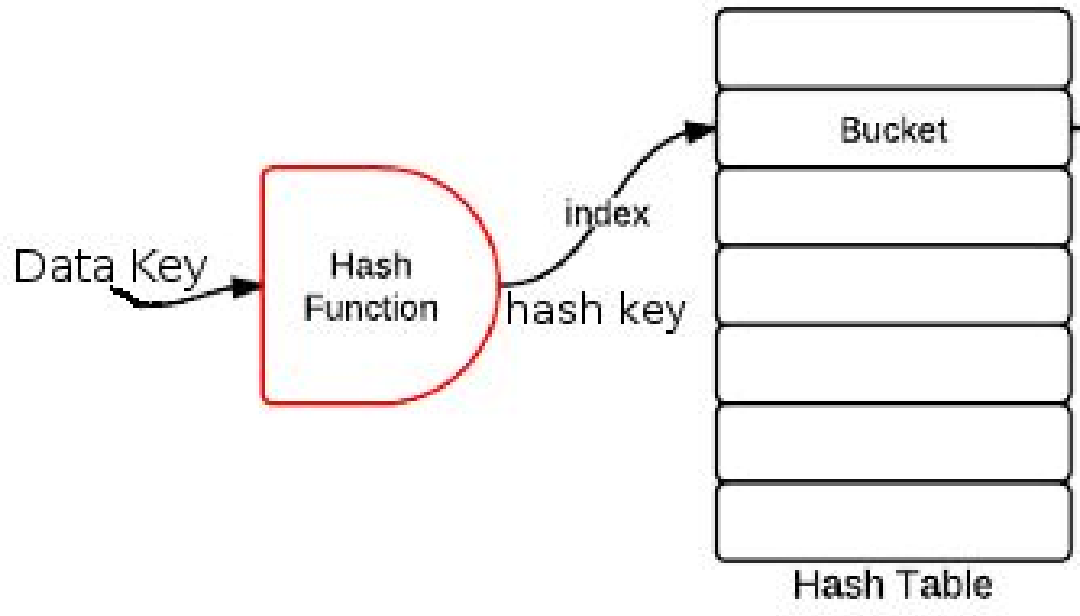


Hashing has several key elements

1) Hash table
   a) Normally a vector (array) of elements indexed by hashed key
   b) Array is of TableSize (TS) size: T hashtable[TableSize]
2) Hash Function
   a) Takes a record's key and returns the index in the hash table to place the record
   b) Ideally this puts elements uniformly throughout the hash table
3) Collision resolution
   a) What do you do if two elements want the same index?
   b) Normally inf number of possible keys keys, but only finite indexes

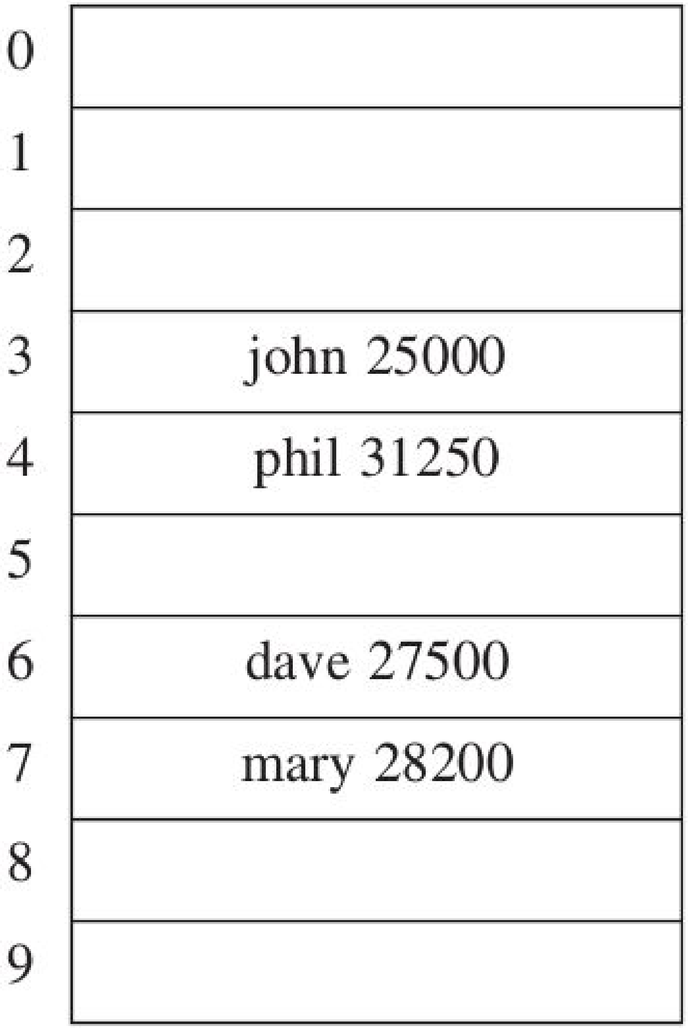
The simplest insert or lookup is thus:

1) Data key into hash function
    a) Outputs hash key
2) Use hash key to index table
3) Insert/lookup data
    @ hash key



Hash table: an array

- Usually an array of pointers to data records
  - Simplest form is pointers to records, but can be more complex ADTs
- Why an array? What's the Big-O for indexing?
- TableSize (TS) is decided by several factors
  - Number of elements to hash (N)
  - Collision resolution function and load calculations
- Normally TS is a prime number
  - More explained later about why
  - This is actually *very* important



Hash function

- Is an algorithm to convert the keys to hash table indexes
  - Called “hash keys”
- Simplest form is just mod by TS:
  - Hashkey = key % TS
  - But if table and key have bad properties? TS = 10 & keys are integers ending in 0?
    - All would hash to index 0!
  - For this first reason, TS should normally be prime
    - With random dataset of integer keys, this gives a nice uniform distribution
- What if TS is large, but data is all small keys? Badness.
  - For example, if doing sum of strings when lengths are 8 chars or less, but TS == 10,007?
    - The keys all fit into the first 0->1,016 hashkeys

A simple integer hash function

```
int hash( const int key, int tableSize ) {
        return( key % tableSize );
}
```

- For ints this will work just fine, but what if we wanted to key on a string?
- That way we could do lookups by names instead of WSU ID values.
- What does this print? Let’s look, shall we?

```
char myChar = ‘A’;
std::cout << (int)myChar << std::endl;
```

Enter ASCII, Extended ASCII, and Unicode!

- What actually *is* a char data type? How big? Is it actually a character?

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	0	000	NULL	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	Start of Header	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	Start of Text	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	End of Text	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	End of Transmission	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	Enquiry	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	Acknowledgment	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	Bell	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	Backspace	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	Horizontal Tab	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	Line feed	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	Vertical Tab	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	Form feed	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	Carriage return	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	Shift Out	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	Shift In	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	Data Link Escape	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	Device Control 1	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	Device Control 2	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	Device Control 3	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	Device Control 4	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	Negative Ack.	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	Synchronous idle	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	End of Trans. Block	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	Cancel	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	End of Medium	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	Substitute	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	Escape	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	File Separator	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	Group Separator	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	Record Separator	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	Unit Separator	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	Del

asciicharstable.com

These days you’ll deal with Unicode too

- “Unicode is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems”: http://unicodefor.us/
- UTF-8: 1-6 bytes each
- UTF-32: 4 bytes each
- But… still numbers you can sum up!
- Has things like interrobang‽ YES□ ☠

WATCHING THE UNICODE PEOPLE TRY TO GOVERN THE INFINITE CHAOS OF HUMAN LANGUAGE WITH CONSISTENT TECHNICAL STANDARDS IS LIKE WATCHING HIGHWAY ENGINEERS TRY TO STEER A RIVER USING TRAFFIC SIGNS.

Hash function continued

- Can sum up ASCII values of strings to make int
  - Allows quick hashing of strings for keys
  - Very nice! Love to index by strings
  - BADNESS->English words aren’t random and most are too short to hash well
- So use Horner’s Rule for polynomial key size (use a prime like 37) instead

$$\sum_{i=0}^{KeySize-1} Key[KeySize - i - 1] \cdot 37^i$$

The code for this is nice and short

Can cap the KeySize to a max num of chars if the strings are very long

Good string-based algorithm

- Hashes uniformly
- Unsigned int?
  - Overflow is okay!
    - WHY?
  - No negative index. :-)
- Can add a limit
  - int lim = m
  - Prevents long calcs
  - O(1) doesn't mean T(n) is actually fast

```
/**
 * A hash routine for string objects.
 */
unsigned int hash( const string & key, int tableSize )
{
    unsigned int hashVal = 0;

    for( char ch : key )
        hashVal = 37 * hashVal + ch;

    return hashVal % tableSize;
}
```

Example in C++11 STL Map

```
// constructing maps
#include <map>
int main ()
{
  std::map<char, int> first;
  first['a']=10;        first['b']=30;
  first['c']=50;        first['d']=70;

  return(0);
}
```

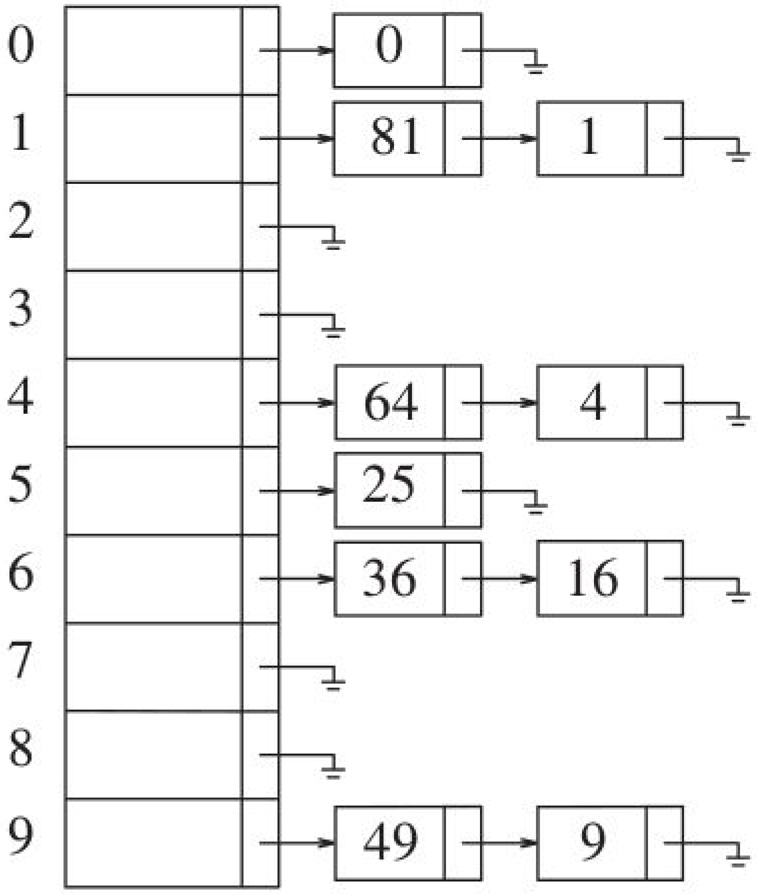
- Keyed by char here
- Could be a string just as easily!
- Does rehashing, sizing, etc for you
- Can store pointers to objects

Collision function?

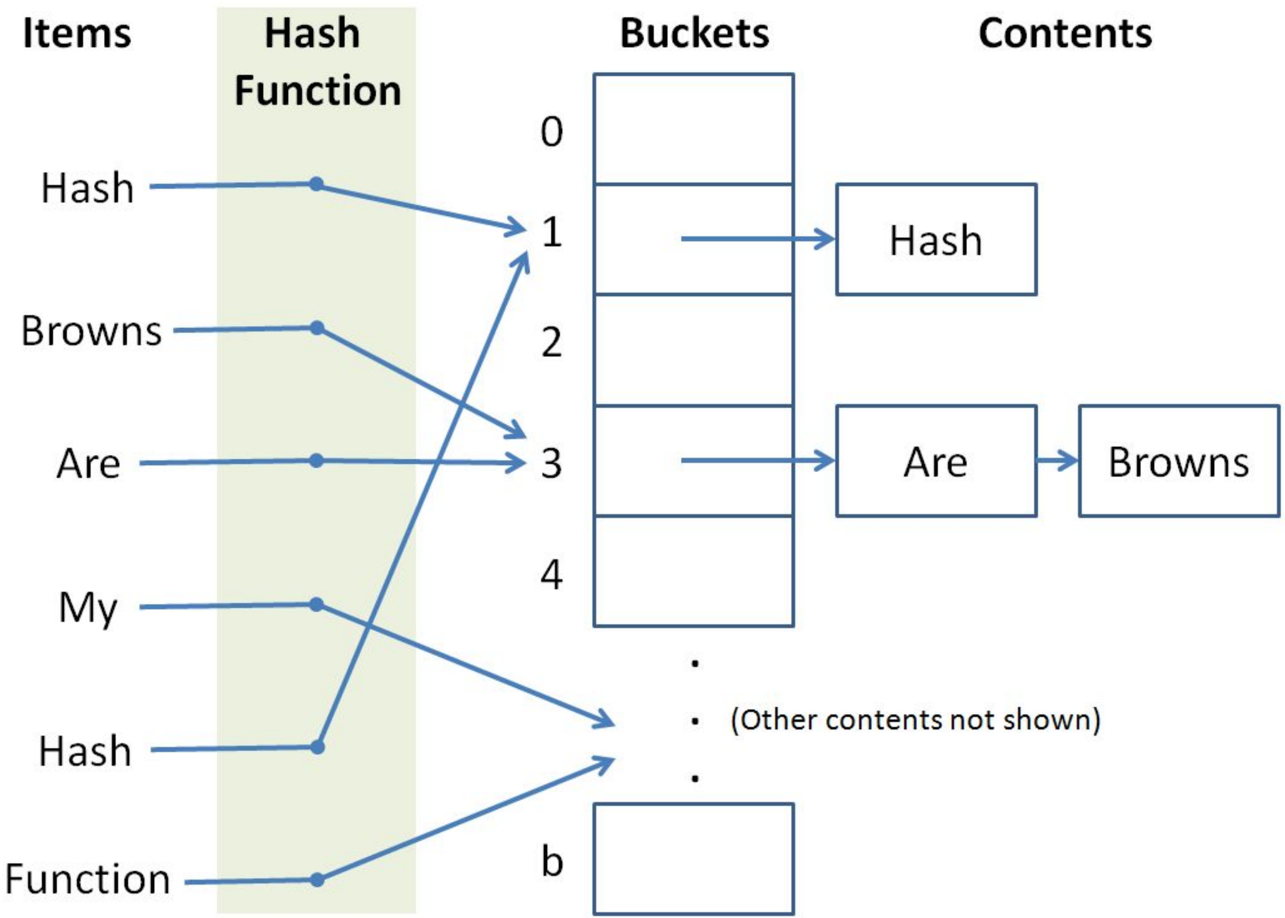
- What do we do if two elements hash to the same index?
  - Rehash right away with a larger TS
  - Just wipe out the old data?
  - Forget the new data?
  - We’ll look at more functions for this later
- Classic approach is Separate Chaining

Separate Chaining

- Hash table elements point to linked lists of records
- Use the STL list for the lists (why not vec?)
- Often insert new records at list head
  - Exploits the principle of locality
- Allows of huge growth
- Still needs a good TS and can be rehashed
- IF TS is good, and hash func good
  - Then... lists aren't too long (how long is "too"?)

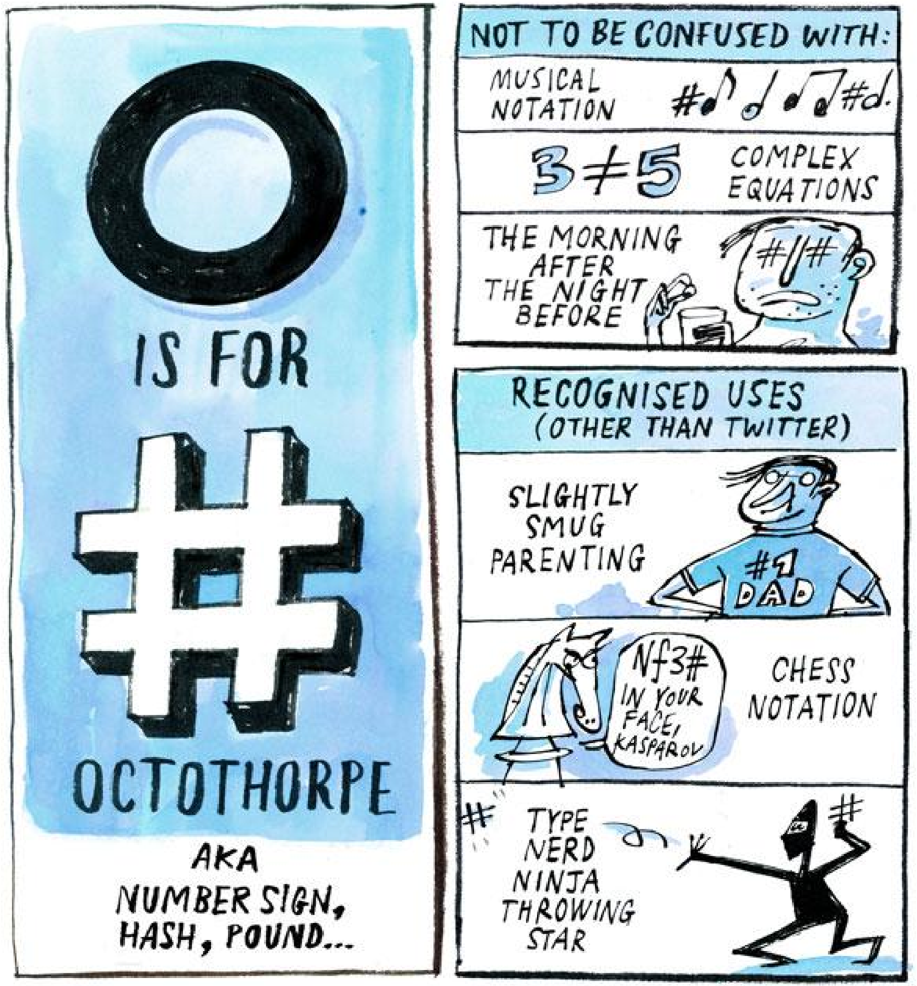


Str example


Items
Hash Function
Buckets
Contents
Hash
Browns
Are
My
Hash
Function
0
1
2
3
4
b
Hash
Are
Browns
(Other contents not shown)

Chaining can also be other storage ADTs

- Instead of lists, the storage can be:
  - BSTs
  - Other hash tables
    - ensure different hash function!
  - Heaps
  - Take your pick!



I want a nerd ninja throwing star. :-(

With Separate Chaining we do have a load factor, but it can go higher than probing

- This factor, λ, helps us decide if we need to rehash
- It's the ratio of N:TS, which is the average length of the lists
  - Assuming a good hash function!
  - For example, if we have N elements and TS = 2N, the lists are (on average) ½ nodes long
- Search time for this is:
  - O(1) for lookup (hash function + table index) + O(λ) to walk the list length
  - Unsuccessful search: O(1) + O(λ)
  - Successful search: O(1) + O(λ/2) -> Remember average search time for a list?

Proving (showing this?)

- Searching a list is for the node + others in list
  - N elements in hash
  - M lists
  - Others in list ~= (N-1)/M == λ - 1/M
    This is essentially λ when M >> 0
- On average ½ of the others are searched so…
  - O(1 + λ/2) to successfully search
- TS doesn't matter here, but load factor does
- Best λ is when TS ~~ N in data
  - If λ gets > 1 do a rehash