CSCI 136
Data Structures &
Advanced Programming

Hashing: Loose Ends

Video Outline

- Growing hashtables
- Choosing an appropriate hashtable size
- Ideal hash function properties and examples
- Revisiting hashtable performance

Hashtable Size

- Vectors are useful because, when a Vector “runs out of space”, the Vector grows
  - It’s very clear when we need to grow a vector: excess capacity = 0
- What does it mean for a hashtable to “run out of space”?
- Even ignoring correctness, performance is slowed by “full” hashtables

Hashtable Size

- The right size for our hashtable will make a trade-off between space and performance
  - We want our table size to be large to minimize collisions (and run/chain lengths) : ↑ ↓
  - We want our table size to be small to minimize wasted space (empty slots): ↑ ↓
- In addition, we would like some flexibility in case we make a bad initial guess for our size

Hashtable Fullness: Load Factor

- Suppose a hashtable with M slots stores N elements
- Load factor is a measure of how full the hash table is
  - LF = (# elements) / (# slots) = N / M
- A smaller load factor means the hashtable is less full, which likely gives better performance

Calculating Load Factor

- To track a hashtable's load factor, we can keep a running count of its elements
  - Every successful `remove()` decrements the count
    - Careful with reserved slots!
    - May want to use chaining if you anticipate many deletes
  - *Some* `put()` operations increment the count
    - Only increment when putting new keys: replacing the value associated with an existing key doesn't change the hashtable's count

- Load factor is then (`count / table.length`)

Using Load Factor

- Given a hashtable's load factor, what should we do?
  - If the load factor is low, nothing!
    - A low load factor should give good performance
  - If the load factor is high (.6?) , grow our table
    - Increase the number of slots without changing the number of elements (LF = N / M)
- How to grow?
  - Vectors: `ensureCapacity()`
    - Allocate new `Object` array, then copy elements to same index within new (larger) array
      - Does this work for hashtables?

Load factors

- Idea: always keep load factor below a certain fraction
- Usually .5 to .75
  - Java HashMap (uses external chaining) uses .75
  - structure5 Hashtable (uses linear probing) uses .6
- Plan: keep track of the load factor. Once it gets too high, make more slots

Doubling Array

- Cannot just copy values
  - Why?
    - Canonical slot may change
  - Example: suppose (`key.hashCode() == 11`)
    - 11 % 8 = 3;
    - 11 % 16 = 11;
- **Result**: to grow our array, we must recompute the hashcode for each item, then reinsert each item into new array

Array sizes

- Some people like using hash tables whose size is a prime
- Reason: taking mod the hash table size (if it's a prime) can help "spread out" our items
- Downside: need to find a prime size when "doubling"

Good Hashing Functions

- Important point:
  - All of our performance hinges on using "good" hash functions that spread keys "evenly"
- Good hash functions:
  - Are fast to compute
  - Uniformly distribute keys across the range
- General rules of thumb?
  - Not really. We almost always have to test "goodness" empirically.

Example Hash Functions

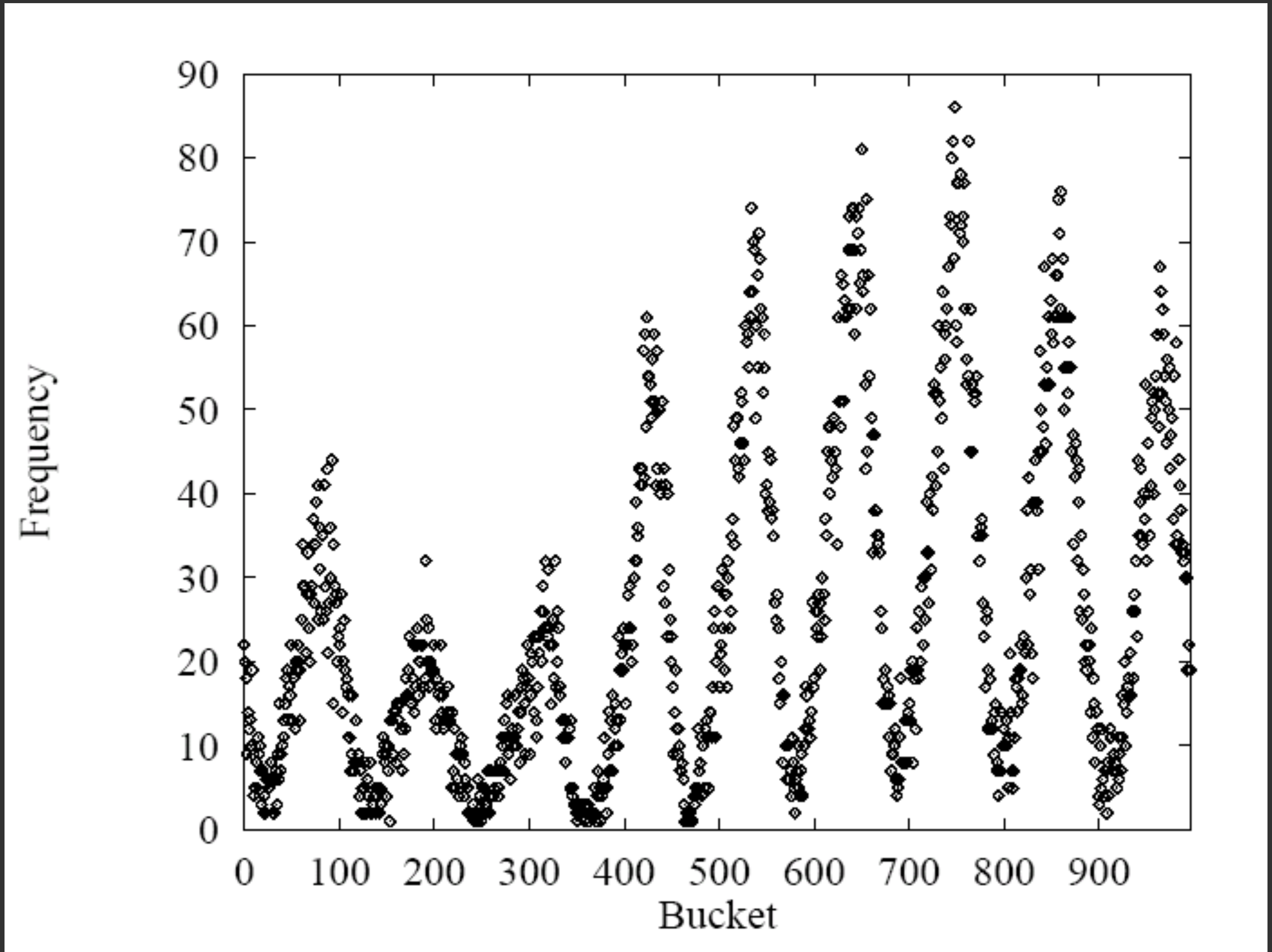
- What are some feasible hash functions for Strings?
  - Use the first char's ASCII value?
    - 0-255 only
    - Not uniform (some letters more popular than others)
  - Sum of all characters' ASCII values?
    - Not uniform - lots of small words
    - Doesn't give coverage over large array sizes
    - Not good at avoiding collisions – e.g., smile, limes, miles, and slime are all the same
    - Let's look at how this works in practice

ASCII TABLE

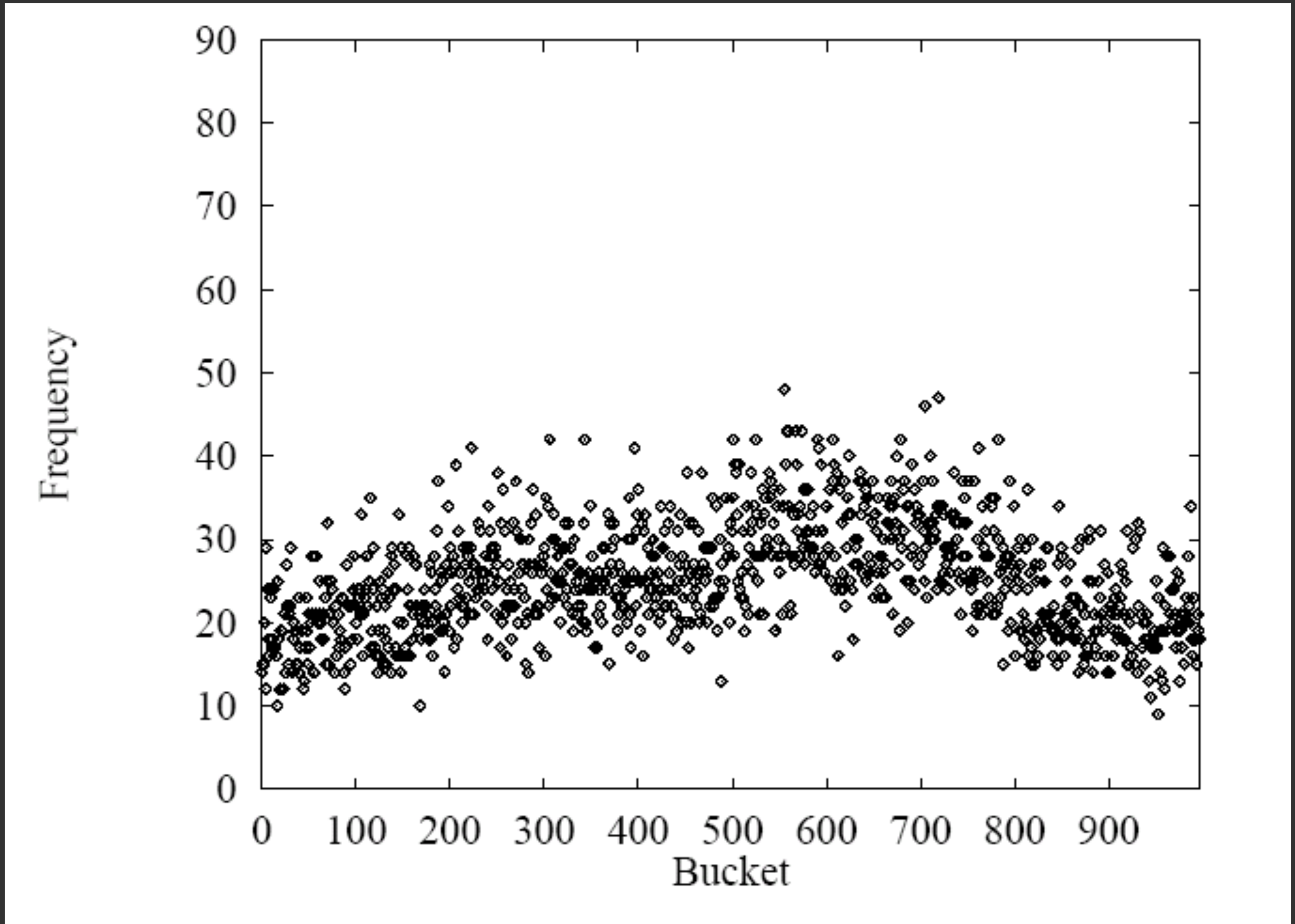
Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	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]

$$\sum_{i=0}^{s.length()} s.charAt(i)$$

Hash of all words in UNIX spelling dictionary (997 buckets)



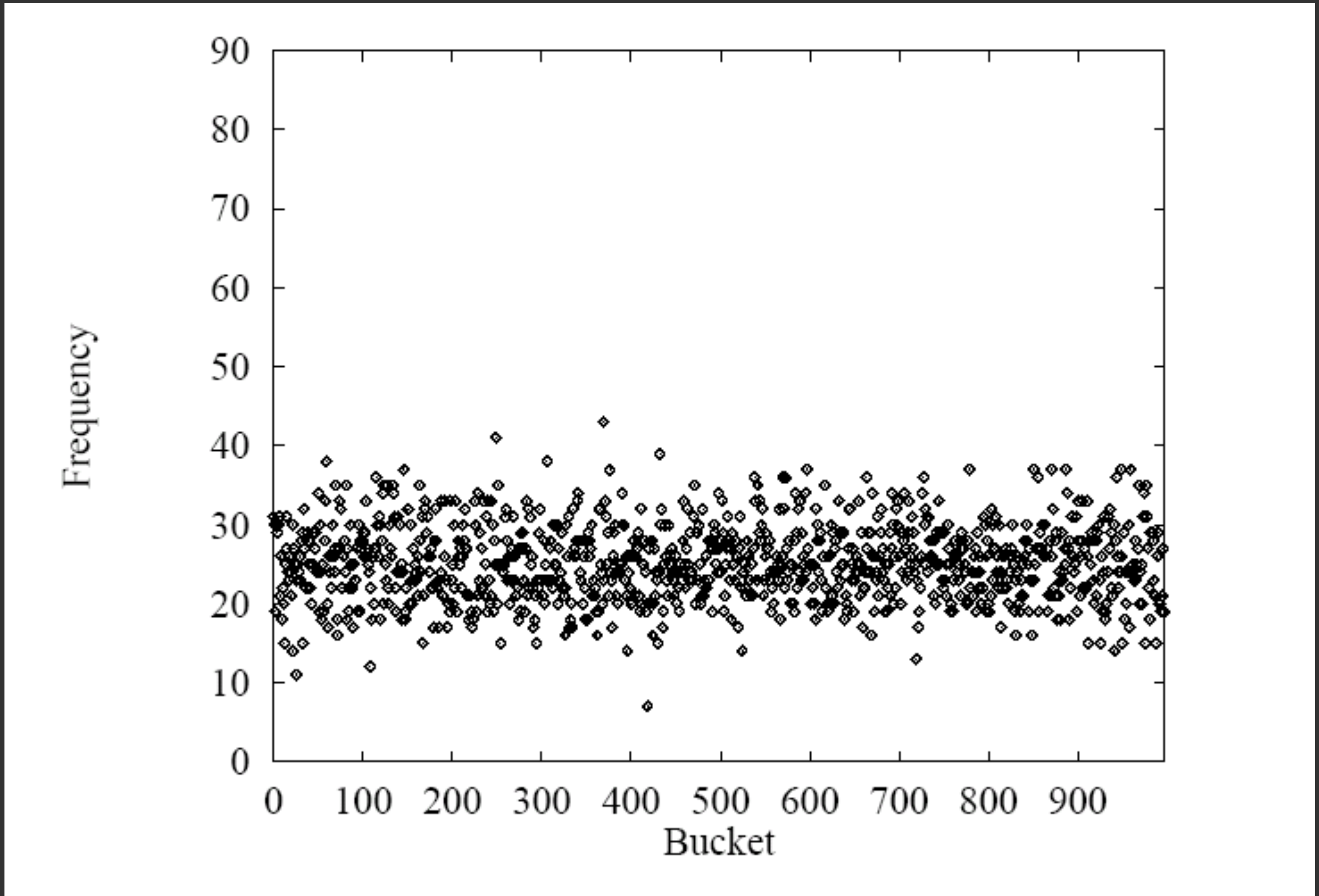
$$\sum_{i=0}^{s.length()} s.charAt(i) * 2^i$$



Better, but buckets are still pretty uneven (middle quite a bit bigger than ends)

$$\sum_{i=0}^{s.length()} s.charAt(i) * 256^i$$

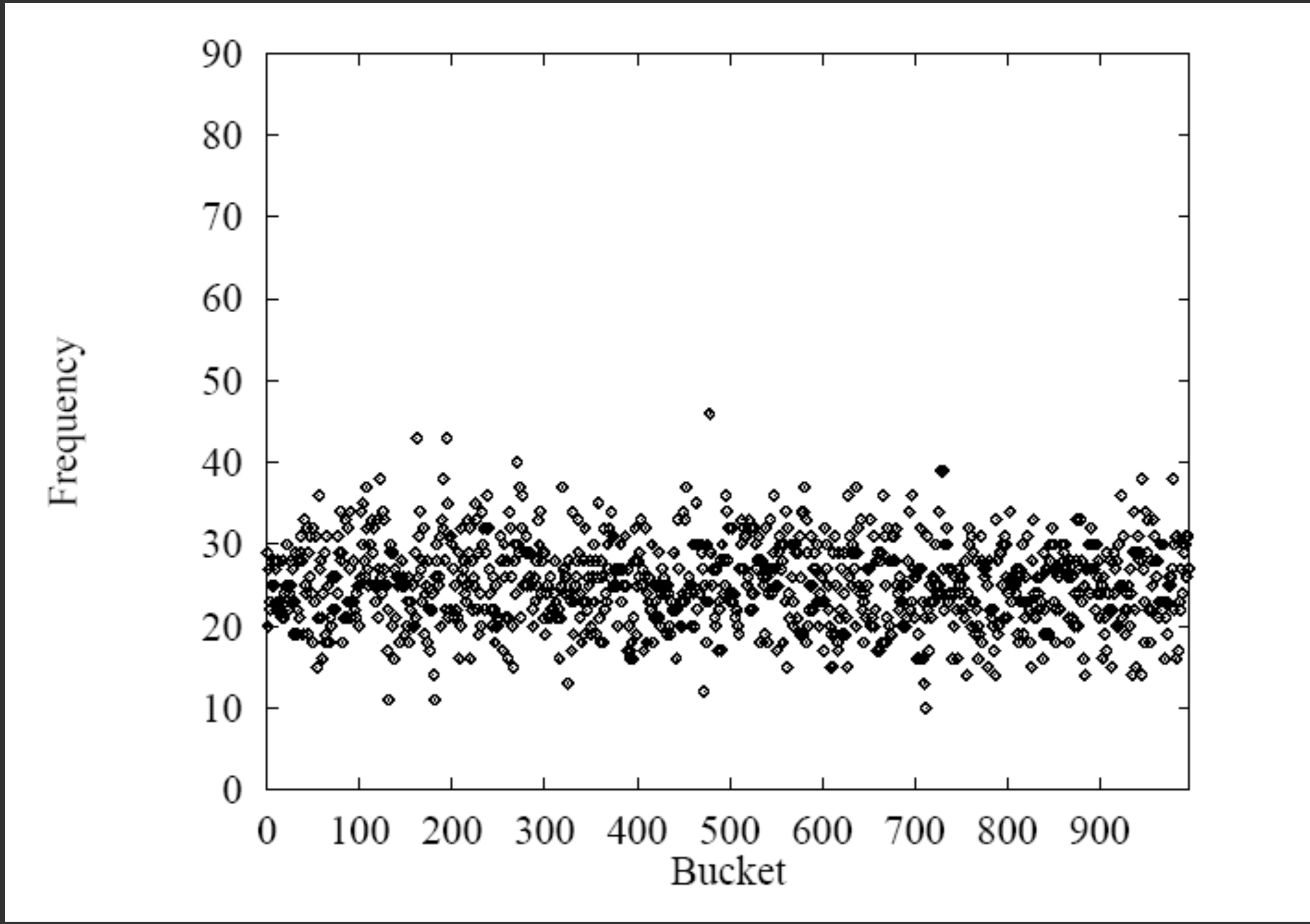
This looks pretty good, but 256^i is big…



$$\sum_{i=0}^{s.length()} s.charAt(i) * 31^{i}$$

Java uses: (here n = s.length())

$$\sum_{i=0}^{n} s.charAt(i) * 31^{n-i-1}$$



Other Objects?

- Integer: i.hashCode() is i
  - That might be very bad depending on your data!
  - May want to use another hashCode() method in that case
    - Perhaps a wrapper class to give a new method
- Character, Long similar
- For your own classes: write your own methods!
  - Test empirically to make sure elements are spread out

Hashtables: O(1) operations?

- How long does it take to compute a String's hashCode?
  - O(s.length())
  - (Doesn't depend on table size)
- Given an object's hash code, how long does it take to find that object?
  - O(run length) or O(chain length) times cost of .equals() method to compare keys

Impact on performance

- Let's say we have constant load factor
  - Number of slots is a constant factor greater than the number of elements
- And we have a good hash function
  - Spreads objects out "like random"

- Then: an average bucket has constant chain length
- An average bucket is in a run of constant length
- (Worst case is O(log n) for both---but this is very rare)
- Usually we say that hash tables have O(1) performance

Summary

	put	get	space
unsorted vector	O(n)	O(n)	O(n)
unsorted list	O(n)	O(n)	O(n)
sorted vector	O(n)	O(log n)	O(n)
balanced BST	O(log n)	O(log n)	O(n)
hashtable	O(1)*	O(1)*	O(n)*