Finding good hash functions

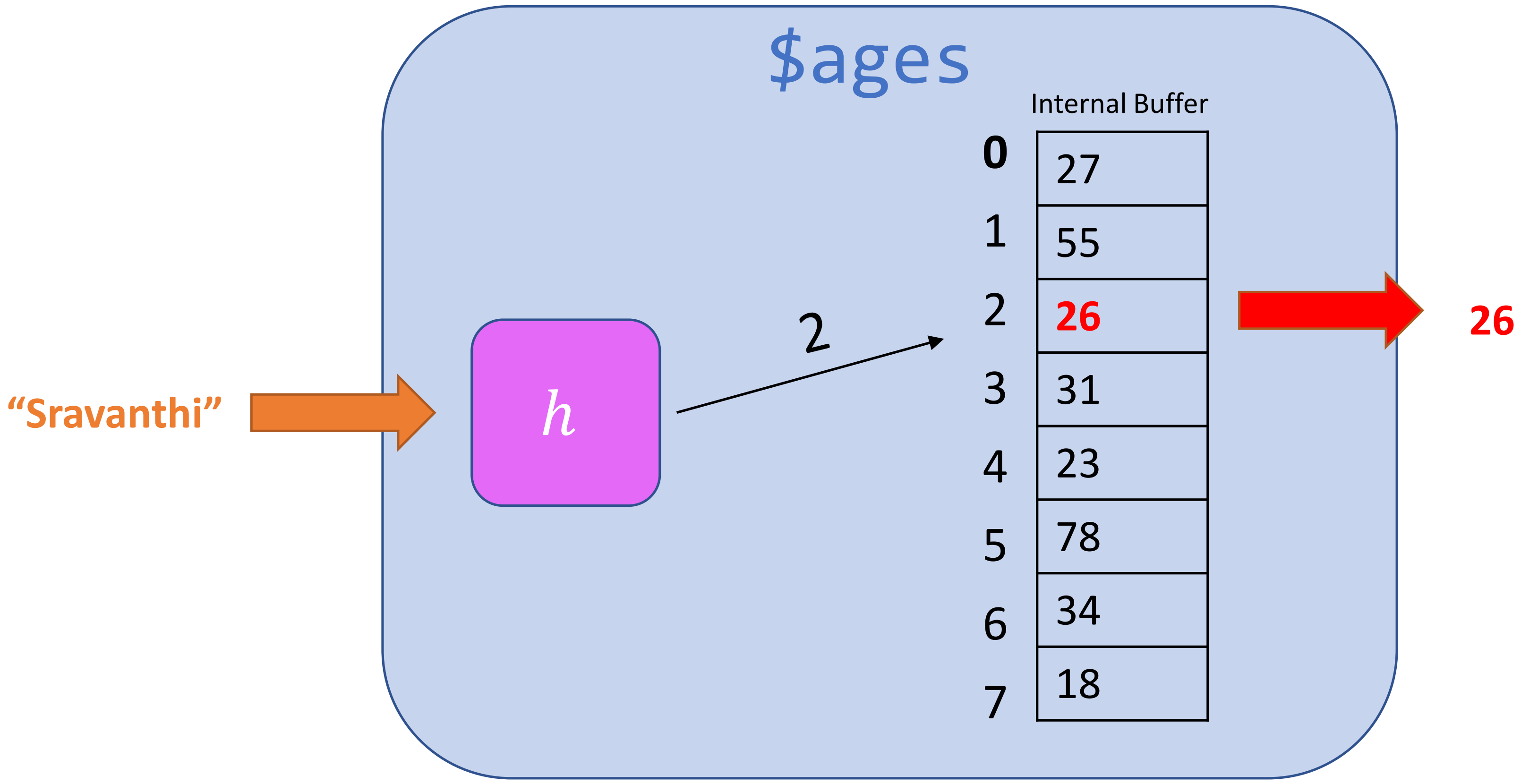
CMSC 420

Hash function requirements

1. Uniformly distribute keys (at least approximately)
   - Reduces “collisions” on the table, allowing for efficient operations.
2. Easy to compute
   - Will be used all the time to delete, insert, search for keys, so it had better be fast to compute.
3. Consistent (“equal” keys should lead to “equal” hashes)
   - Otherwise, **search is broken** (a key that has been inserted can no longer be found)

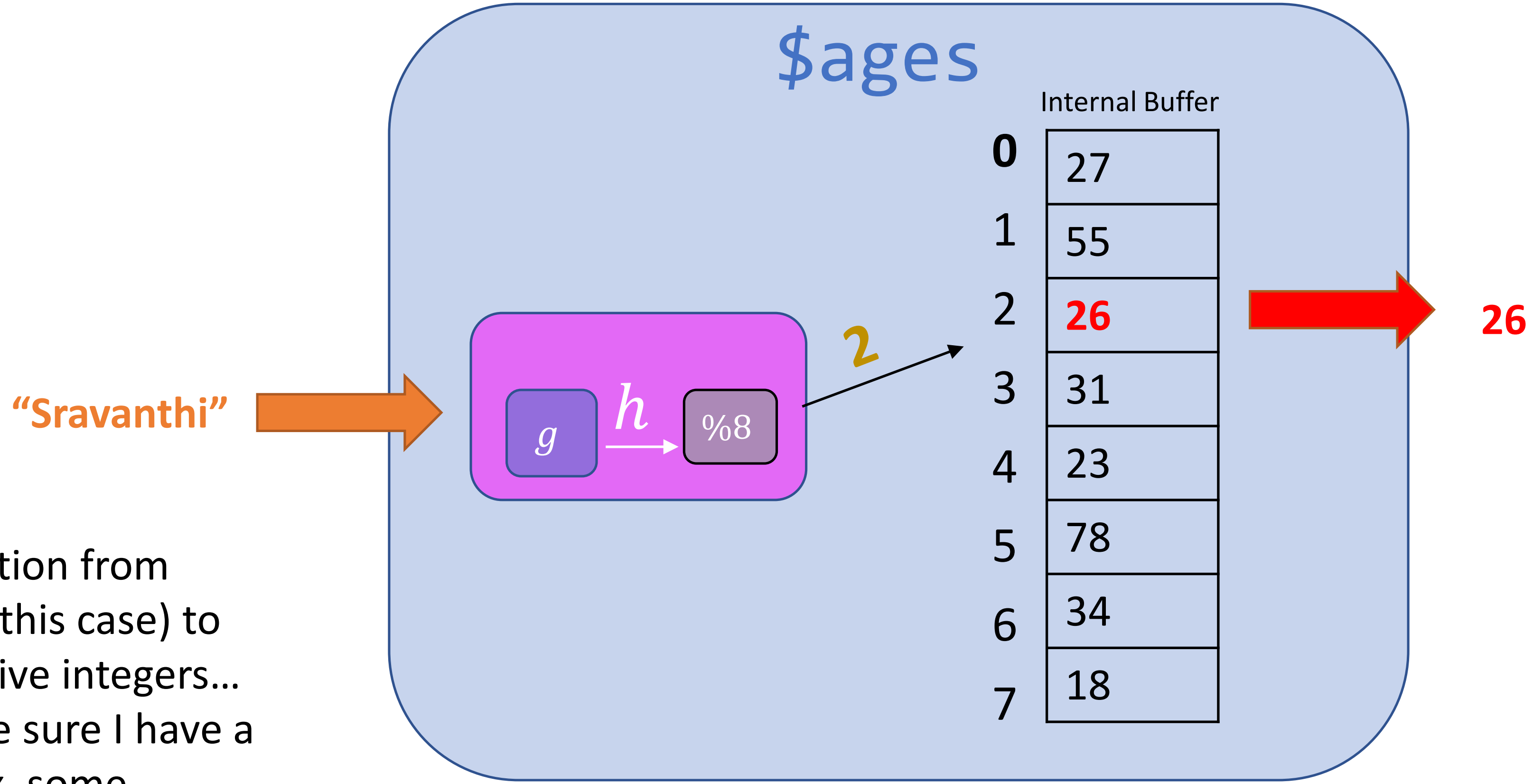
Form of a hash function

- Recall the PHP associative array “incision” we made:



Form of a hash function

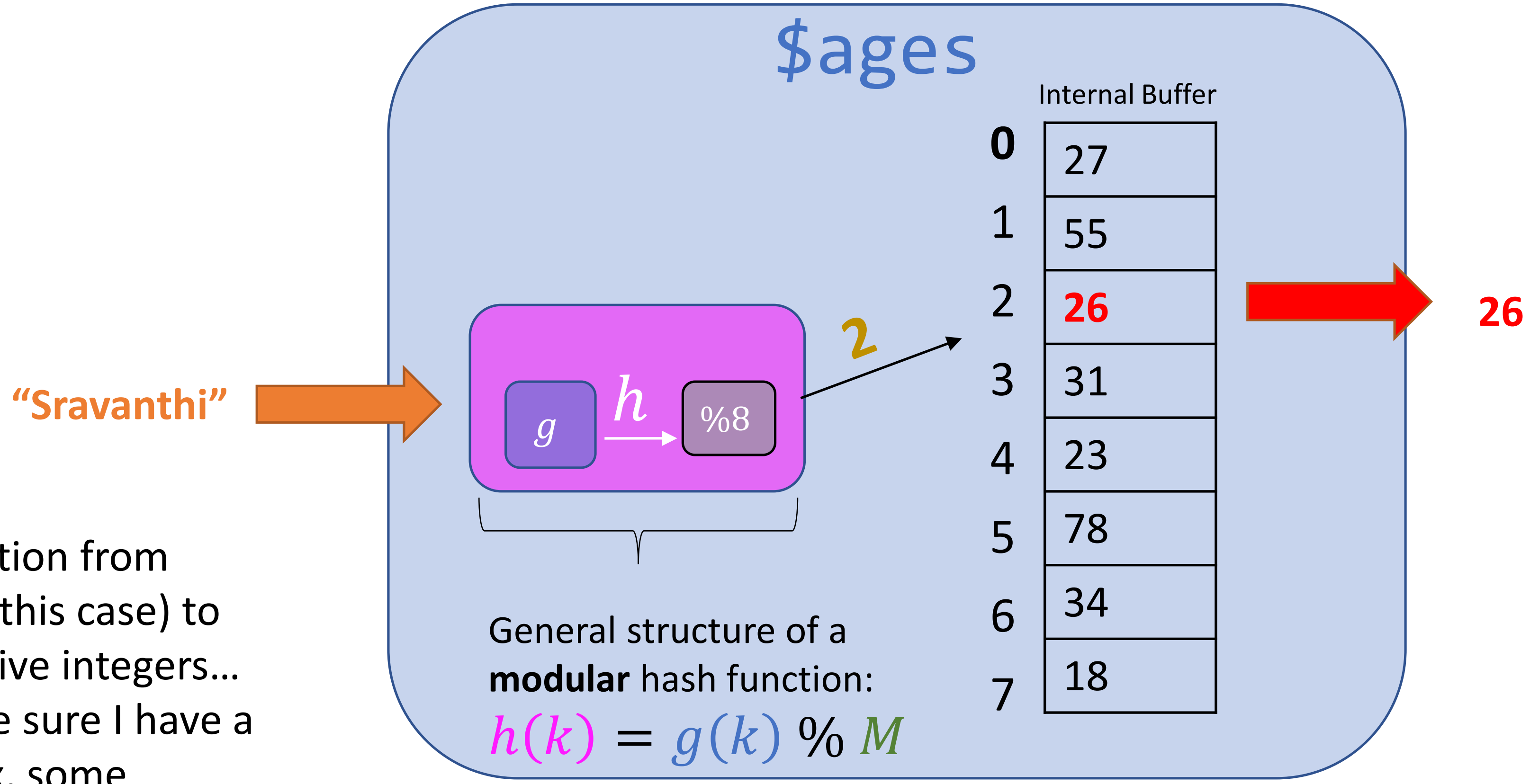
- Recall the PHP associative array “incision” we made:



- h is a function from strings (in this case) to non-negative integers…
- So to make sure I have a valid index, some modding has to take place

Form of a hash function

- Recall the PHP associative array “incision” we made:



- h is a function from strings (in this case) to non-negative integers…
- So to make sure I have a valid index, some modding has to take place

Choice of hash table size

- Which among the following is the best choice for M, the hash table size?

18 | 97 | 1000 | 2000

Choice of hash table size

- Which among the following is the best choice for M, the hash table size?

18 | 97 | 1000 | 2000

- The only given choice that is a prime number!

Choice of hash table size

- Which among the following is the best choice for M, the hash table size?

18

97

1000

2000

- The only given choice that is a prime number!
- Let’s see why this is important…

A simple example

- Suppose that my keys are base-10 integers and my array size is 100.
- Assuming g the identity function, my function formula is

$$h(k) = k\%100$$

A simple example

- Suppose that my keys are base-10 integers and my array size is 100.
- Assuming g the identity function, my function formula is

$$h(k) = k\%100 \quad 10^2$$

- Then, I'm ignoring the information from the most significant $k-2$ **decimal digits** to better disperse my keys ☹
  - Consider: $h(101) = h(547200301) = h(1500000000000001) = 1$

Formal argument

- Suppose that h returns, the following hashed index sequence for some arbitrary keys:

$$\{0, n, 2n, 3n, 4n, \ldots,\} \quad (n \in \mathbb{N}^{\geq 1})$$

- Then, all the keys that correspond to this sequence will be clustered into:

$$\frac{M}{GCD(M, n)}$$

M is the hash table **capacity** (the number of its cells).

buckets!

Evaluating the argument

- Example: Suppose $M = 100$ and $h(k) = k \;\%\; M$. Then, the following sequence of 501 keys:

$$\{0, 20, 40, 60, 80, 100, \ldots, 10000\}$$

- Can only be hashed to indices 0, 20, 40, 60, 80, 0, 20, 40, 60, 80, 0, 20, 40, 60, 80, …
- So we only have $\frac{100}{GCD(100,20)} = \frac{100}{20}$ =5 possible buckets for 501 keys, which means only 1% of keys can be given unique positions in storage ☹

Evaluating the argument

- Suppose now that $M = 101$ (prime) and $h(k) = k \ \% \ M$. Then, the same sequence of 501 keys:

$$\{0, 20, 40, 60, 80, 100, 120, 140, 160, 180, \ldots, 9900, 9920, 9940, 9960, 9980, 10000\}$$

- Then, the sequence will be hashed to a total of $\frac{101}{GCD(101,20)} = 101$ buckets, which is 25% of the total #keys!
  - And all it took was increasing M by 1! ☺

Evaluating the argument

- Suppose now that $M = 101$ (prime) and $h(k) = k \ \% \ M$. Then, the same sequence of 501 keys:

$$\{0, 20, 40, 60, 80, 100, 120, 140, 160, 180, \dots, 9900, 9920, 9940, 9960, 9980, 10000\}$$

- Then, the sequence will be hashed to a total of $\frac{101}{GCD(101,20)} = 101$ buckets, which is 25% of the total #keys!
  - And all it took was increasing M by 1! ☺

- Picking M=149 (prime) yields: $\frac{149}{GCD(149,20)}$ =149 buckets, 29% of the total #keys.

Evaluating the argument

- Suppose now that $M = 101$ (prime) and $h(k) = k \ \% \ M$. Then, the same sequence of 501 keys:

$$\{0, 20, 40, 60, 80, 100, 120, 140, 160, 180, \dots, 9900, 9920, 9940, 9960, 9980, 10000\}$$

- Then, the sequence will be hashed to a total of $\frac{101}{GCD(101,20)} = 101$ buckets, which is 25% of the total #keys!
  - And all it took was increasing M by 1! ☺

- Picking M=149 (prime) yields: $\frac{149}{GCD(149,20)}$ =149 buckets, 29% of the total #keys.
- Picking M=337 (prime) yields: $\frac{337}{GCD(337,20)}$ =337 buckets, 67% of the total #keys

Evaluating the argument

- Suppose now that $M = 101$ (prime) and $h(k) = k \% M$. Then, the same sequence of 501 keys:

$$\{0, 20, 40, 60, 80, 100, 120, 140, 160, 180, \dots, 9900, 9920, 9940, 9960, 9980, 10000\}$$

- Then, the sequence will be hashed to a total of $\frac{101}{GCD(101,20)} = 101$ buckets, which is 25% of the total #keys!
  - And all it took was increasing M by 1! ☺

- Picking M=149 (prime) yields: $\frac{149}{GCD(149,20)}$ =149 buckets, 29% of the total #keys.
- Picking M=337 (prime) yields: $\frac{337}{GCD(337,20)}$ =337 buckets, 67% of the total #keys
- Picking M = 502 (***composite greater than the #keys!!!***), yields:

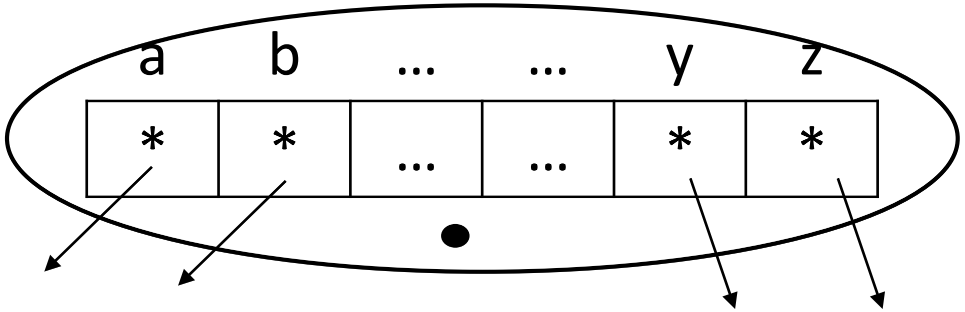
  $\frac{502}{GCD(502,20)}$ =251 buckets < 337 buckets for M=337 < 502!!!!

Hash functions: Keys are Integers

- When your keys are ints, a simple modular hash function will do.
- This is the approach that we will follow in class.

Hash functions for characters

- A data structure known as a *trie* (“try”) indexes by character!

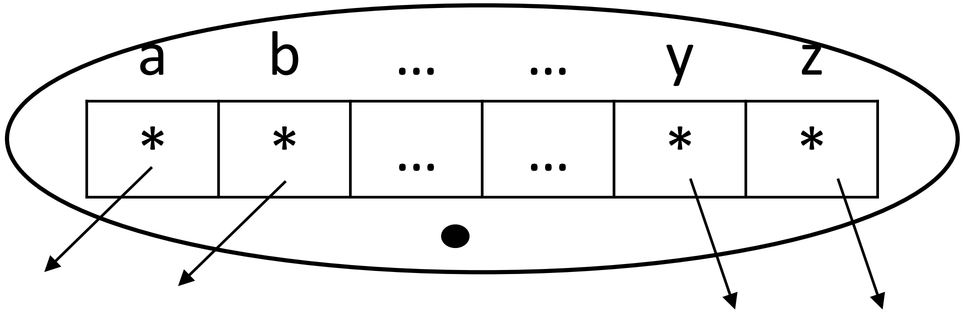


$$h(k) = (int)k - 97,$$
(where $k \in \{'a', 'b', \ldots, 'z'\}$)

- Note that this is not a **modular** hash function!

Hash functions for characters

- A data structure known as a *trie* ("try") indexes by character!



$$h(k) = (int)k - 97,$$

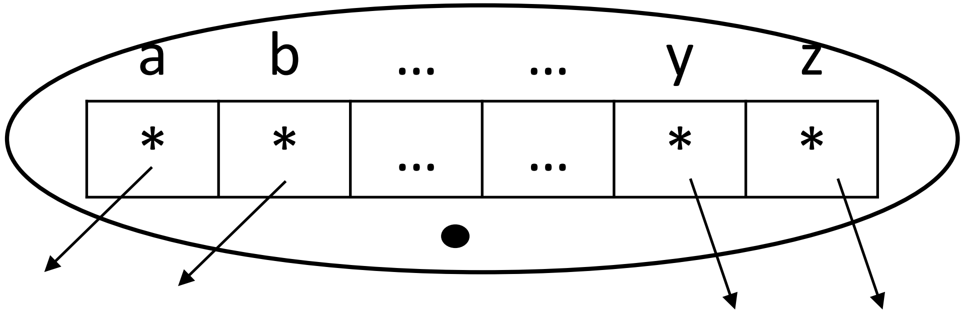
(where $k \in \{'a', 'b', \ldots, 'z'\}$)

A subtrahend of 97 allows ASCII 'a' (lowercase 'A') to be hashed to 0! ☺

- Note that this is not a **modular** hash function!

Hash functions for characters

- A data structure known as a *trie* (“try”) indexes by character!



A subtrahend of 97 allows ASCII ‘a’ (lowercase ‘A’) to be hashed to 0! ☺

$$h(k) = (int)k - 97,$$

(where $k \in \{'a', 'b', \ldots, 'z'\}$)

- Note that this is not a **modular** hash function!

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Hash functions for floating-point keys

- Humans typically write floating-point numbers in base 10. Example:

$$\pi_{(10)} \approx 3.14159\ldots_{(10)} = 3 \times 10^0 + \frac{1}{10} + \frac{4}{10^2} + \frac{1}{10^3} + \frac{5}{10^4} + \frac{9}{10^5} + \cdots$$

Hash functions for floating-point keys

- Humans typically write floating-point numbers in base 10. Example:

$$\pi_{(10)} \approx 3.14159\ldots_{(10)} = 3 \times 10^0 + \frac{1}{10} + \frac{4}{10^2} + \frac{1}{10^3} + \frac{5}{10^4} + \frac{9}{10^5} + \cdots$$

- We can similarly represent floating-point numbers in base 2! Example:

$$\pi_{(2)} = 11.00100100011\ldots_{(2)}$$
$$= 1 \times 2^1 + 1 \times 2^0 + \frac{0}{2^1} + \frac{0}{2^2} + \frac{1}{2^3} + \frac{0}{2^4} + \frac{0}{2^5} + \frac{1}{2^6} + \frac{0}{2^7} + \frac{0}{2^8} + \frac{0}{2^9} + \frac{1}{2^{10}} + \frac{1}{2^{10}} + \cdots$$

Hash functions for floating-point keys

- Exercise!

$$2.75_{(10)} = ?_{(2)}$$
$$3.25_{(10)} = ?_{(2)}$$

Hash functions for floating-point keys

- Exercise!

$$2.75_{(10)} = 10.11_{(2)}$$
$$3.25_{(10)} = 11.01_{(2)}$$

Hash functions for floating-point keys

- Exercise!

$$2.75_{(10)} = 10.11_{(2)}$$
$$3.25_{(10)} = 11.01_{(2)}$$

- To hash a floating point number, I *could* round to the nearest integer and then use a modular hash function on integers…

Hash functions for floating-point keys

- Exercise!

$$2.75_{(10)} = 10.11_{(2)}$$
$$3.25_{(10)} = 11.01_{(2)}$$

- To hash a floating point number, I *could* round to the nearest integer and then use a modular hash function on integers…
- But doing so would mean that the keys 2.75 and 3.25 would collide ☹
  - The general case is much worse: **All keys in [2.5, 3.5) would collide**!

Hash functions for floating-point keys

- Exercise!

$$2.75_{(10)} = 10.11_{(2)}$$
$$3.25_{(10)} = 11.01_{(2)}$$

- To hash a floating point number, I *could* round to the nearest integer and then use a modular hash function on integers…
- But doing so would mean that the keys 2.75 and 3.25 would collide ☹
  - The general case is much worse: **All keys in [2.5, 3.5) would collide**!
- **SOLUTION**: Use a hash function on the **binary representation of those numbers**!

Hash functions for floating-point keys

- Exercise!

$$2.75_{(10)} = 10.11_{(2)}$$
$$3.25_{(10)} = 11.01_{(2)}$$

- To hash a floating point number, I *could* round to the nearest integer and then use a modular hash function on integers…
- But doing so would mean that the keys 2.75 and 3.25 would collide ☹
  - The general case is much worse: **All keys in [2.5, 3.5) would collide**!

- **SOLUTION**: Use a hash function on the **binary representation of those numbers**!
- But, binary representations are strings….
- So now we **need hash functions for string data types!**

Hash functions for strings

- The following is what Java uses, with $R = 31$ (a prime) in most cases.
- Remember that `char`s in Java are 2-byte `unsigned int`s (so the first $2^{16} =$ 65536 Unicode codepoints).

```
int hash = 0;
for (int i = 0; i < s.length(); i++)
     hash = (R * hash + s.charAt(i)) % M;
```

- This allows us to treat s as an $|s|$-digit base-R integer mod M.
- Modding at every iteration allows us to maintain small intermediate values for "hash"
  - If you're familiar with **modular exponentiation**, the exact same reasoning is applied for modding by M at every step!

Using Java's **hashCode( )**

hashCode

```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

As much as is reasonably practical, the hashCode method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)

Returns:

a hash code value for this object.

See Also:

`equals(java.lang.Object)`, `System.identityHashCode(java.lang.Object)`

Using Java's **hashCode()**

hashCode

`public int hashCode()`

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinc integer results for unequal objects may improve the performance of hash tables.

As much as is reasonably practical, the hashCode method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required the Java™ programming language.)

Returns:

a hash code value for this object.

See Also:

`equals(java.lang.Object)`, `System.identityHashCode(java.lang.Object)`

- Remember: `k1.equals(k2)` ⇒ (`k1.hashCode() == k2.hashCode()`) is an implication (⇒) **not** a logical equivalence! (⇔)
- So, while Java tries, there are ***no guarantees*** that

  `(!k1.equals(k2))` ⇒ `(k1.hashCode() != k2.hashCode())`

Using Java's **hashCode()**

So what about **negative** hashCode()s?

hashCode

`public int hashCode()`

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

As much as is reasonably practical, the hashCode method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)

Returns:

a hash code value for this object.

See Also:

`equals(java.lang.Object)`, `System.identityHashCode(java.lang.Object)`

- Remember: `k1.equals(k2)` ⇒ (`k1.hashCode() == k2.hashCode()`) is an implication (⇒) **not** a logical equivalence! (⇔)
- So, while Java tries, there are ***no guarantees*** that

  `(!k1.equals(k2))` ⇒ `(k1.hashCode() != k2.hashCode())`

Using Java's `hashCode()`: Negative hashes

- To ensure that the values that `hashCode()` returns are useable as array indices (e.g in a modular hash function), we need to mask the top bit!

Using Java's hashCode(): Negative hashes

- To ensure that the values that hashCode() returns are useable as array indices (e.g in a modular hash function), we need to mask the top bit!
  - This means that Java hash tables can not grow beyond 2^{31} positions!

Using Java's `hashCode()`: Negative hashes

- To ensure that the values that `hashCode()` returns are useable as array indices (e.g in a modular hash function), we need to mask the top bit!
  - This means that Java hash tables can not grow beyond 2^{31} positions!
  - To do this masking, simply do a bitwise **AND** with `0x7fffffff`:

```
private int hash(Key x, int M){
    return (x.hashCode() & 0x7fffffff) % M;
}
```

Using Java's `hashCode()`: Custom data types

- Recall: Default `hashCode()` based on object's address in memory.
- Also, let's not forget our contract:

$$\texttt{k1.equals(k2)} \Rightarrow (\texttt{k1.hashCode() == k2.hashCode()})$$

Using Java's `hashCode()`: Custom data types

- Recall: Default `hashCode()` based on object's address in memory.
- Also, let's not forget our contract:

`k1.equals(k2)` ⇒ `(k1.hashCode() == k2.hashCode())`

Demo time!

Take-home message from `hashCode()` and custom data types

- Overriding the default `hashCode()` should have the same priority as :
  - Creating appropriate constructors
  - Overriding `equals()` in a type-safe manner
  - Overriding `compareTo()` (for `Comparable`s).

Take-home message from `hashCode()` and custom data types

- Overriding the default `hashCode()` should have the same priority as :
  - Creating appropriate constructors
  - Overriding `equals()` in a type-safe manner
  - Overriding `compareTo()` (for `Comparable`s).

- Jason considers this a design flaw of Java (and any other language that might replicate this behavior).
  - Many developers go through their entire careers without ever having had to understand the inner workings of `Object.hashCode()`, and that's ok!
  - Yet, if client code uses their data types as keys, inconsistencies can arise because equality has not been extended to the hash code level…

Resizing our hash table

- At some point, no matter how large of a prime number we have as M, or no matter how "uniform" h is, our performance **will** hurt with our current M ☹
- So we have to **enlarge our table** (increase M somehow)

Resizing our hash table

- At some point, no matter how large of a prime number we have as M, or no matter how "uniform" h is, our performance **will** hurt with our current M ☹
- So we have to **enlarge our table** (increase M somehow)
- **Question:** What's a good resizing strategy for M?

`M <- 2*M`

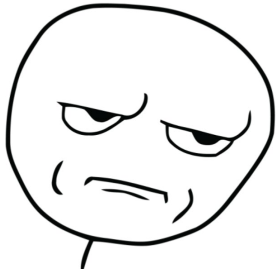
`M <- 3*M`

`M <- 4*M`

`M <- 17*M`

Resizing our hash table

- At some point, no matter how large of a prime number we have as M, or no matter how “uniform” h is, our performance **will** hurt with our current M ☹
- So we have to **enlarge our table** (increase M somehow)
- **Question:** What’s a good resizing strategy for M?



NONE OF THEM.
ALL OF THEM make M composite!

`M <- 2*M`   `M <- 3*M`   `M <- 4*M`   `M <- 17*M`

Best option: Maintain a large look-up table of primes and increase the choice of M to the next prime each time!

When to resize?

- It’s your choice.
- Tradeoff: Resizing implies **reinserting everything based on the new M.**
  - So, if you resize at 40% capacity, you are paying for re-insertions much more often than resizing at an 80% capacity.
  - But! You end up with a very efficient hash table (few collisions)
- In practice: some people online mention 70% as a good compromise
- Jason found Cython source for a `dict()` implementation somewhere (based on *double hashing*) which resized at 60 or 66% (can’t remember exact percentage)