CS 600.226: Data Structures

Michael Schatz

Nov 7, 2018

Lecture 29: HashTables pt2

HW7

Assignment 7: Whispering Trees

Out on: November 2, 2018

Due by: November 9, 2018 before 10:00 pm

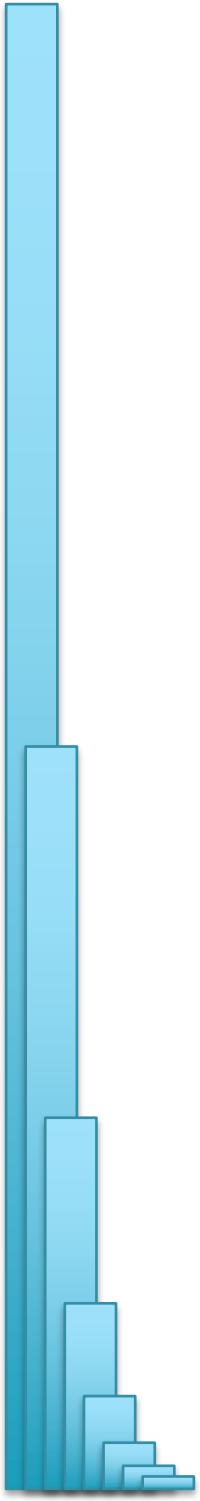
Collaboration: None

Grading:

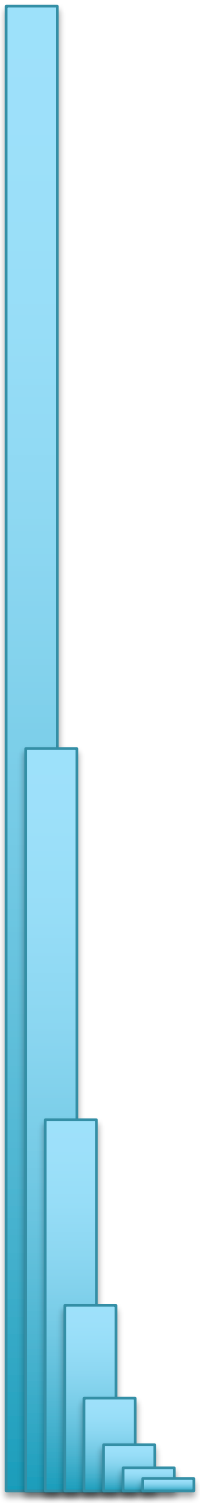
- Packaging 10%,
- Style 10% (where applicable),
- Testing 10% (where applicable),
- Performance 10% (where applicable),
- Functionality 60% (where applicable)

Overview

The seventh assignment is all about ordered maps, specifically fast ordered maps in the form of balanced binary search trees. You'll work with a little program called Words that reads text from standard input and uses an (ordered) map to count how often different words appear. We're giving you a basic (unbalanced) binary search tree implementation of OrderedMap that you can use to play around with the Words program and as starter code for your own developments.



Part 1: Hash Tables



Maps

aka dictionaries

aka associative arrays

```
Mike      -> Malone 323
Peter     -> Malone 223
Joanne     -> Malone 225
Zack      -> Malone 160 suite
Debbie     -> Malone 160 suite
Yair      -> Malone 160 suite
Ron       -> Garland 242
```

Key (of Type K) -> Value (of Type V)

Note you can have multiple keys with the same value,
But not okay to have one key map to more than 1 value

Maps, Sets, and Arrays

Sets as Map<T, Boolean>

```
Mike     -> True
Peter    -> True
Joanne   -> True
Zack     -> True
Debbie   -> True
Yair     -> True
Ron      -> True
```

Array as Map<Integer, T>

```
0   -> Mike
1   -> Peter
2   -> Joanne
3   -> Zack
4   -> Debbie
5   -> Yair
6   -> Ron
```

Maps are extremely flexible and powerful,
and therefore are extremely widely used

Built into many common languages: Awk, Python, Perl, JavaScript…

Could we do everything in O(lg n) time or faster?
=> Balanced Search Trees

Maps, Sets, and Arrays

Sets as Map<T, Boolean>

```
Mike      -> True
Peter     -> True
Joanne    -> True
Zack      -> True
Debbie    -> True
Yair      -> True
Ron       -> True
```

Array as Map<Integer, T>

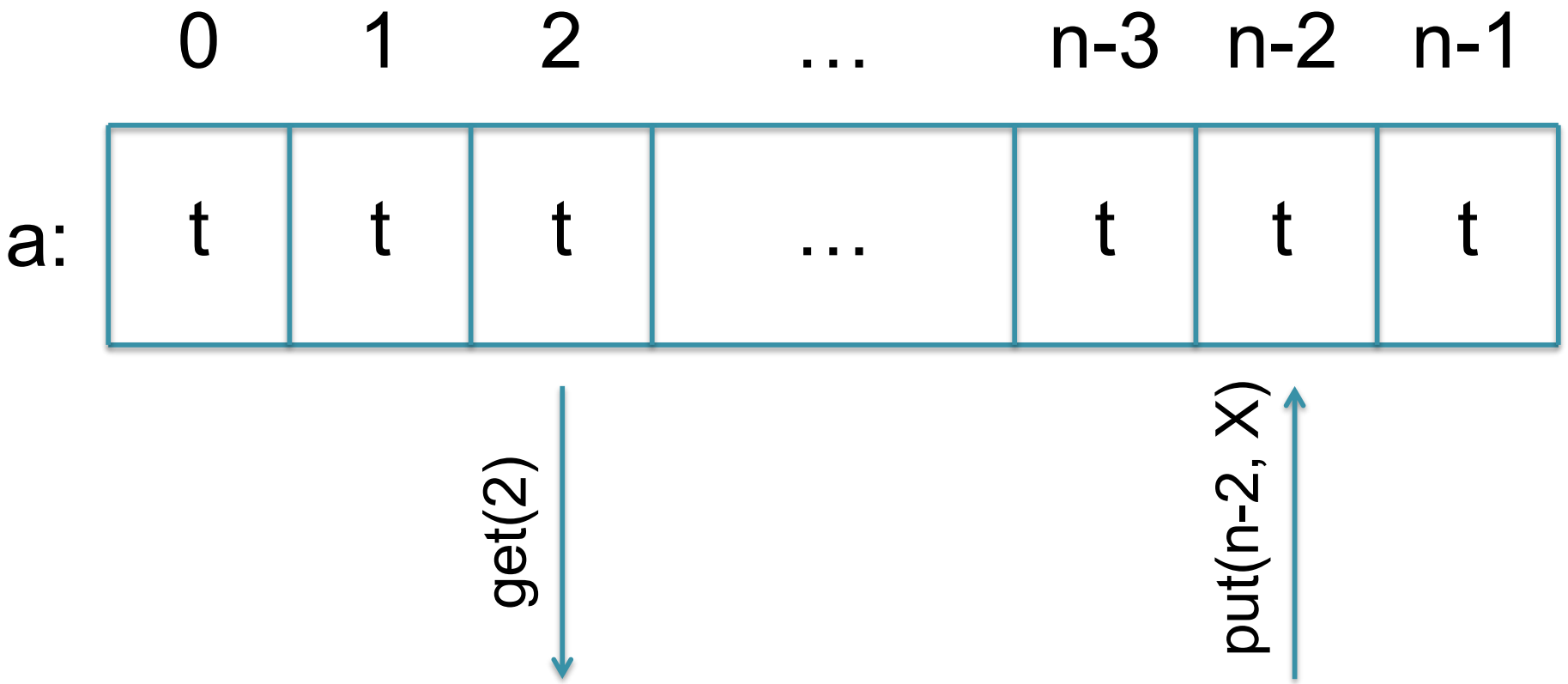
```
0   -> Mike
1   -> Peter
2   -> Joanne
3   -> Zack
4   -> Debbie
5   -> Yair
6   -> Ron
```

Maps are extremely flexible and powerful,
and therefore are extremely widely used

Built into many common languages: Awk, Python, Perl, JavaScript…

Could we do everything in O(lg n) time or faster?
=> Balanced Search Trees

ADT: Arrays



- Fixed length data structure
- Constant time get() and put() methods
- Definitely needs to be generic ☺

Hashing

Array[13] = 10; Array[42] = 15 ***O(1)***

Array["Mike"] = 10; Array["Peter"] = 15 ***BST:O(lg n) -> Hash:O(1)***

Hash Function enables Map<K, V> as Map<Integer, V> as Array<V>

- h(): K -> Integer for any possible key K
- h() should distribute the keys uniformly over all integers
- if k_1 and k_2 are "close", $h(k_1)$ and $h(k_2)$ should be "far" apart

Typically want to return a small integer, so that we can use it as an index into an array

- An array with 4B cells in not very practical if we only expect a few thousand to a few million entries
- How do we restrict an arbitrary integer x into a value up to some maximum value n?

```
0 <= x % n < n
```

Compression function: c(i) = abs(i) % length(a)

Transforms from a large range of integers to a small range (to store in array a)

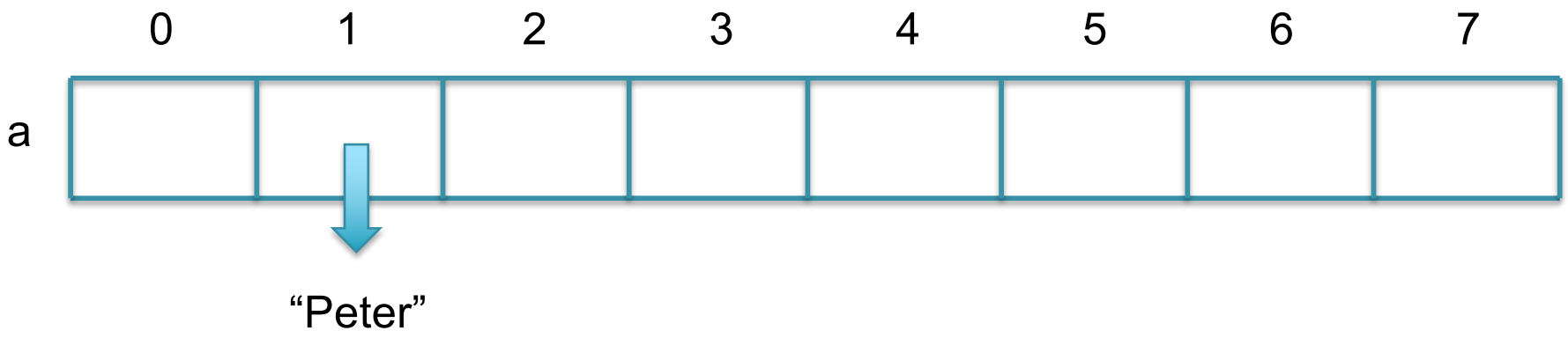
Collisions

Collisions occur when 2 different keys get mapped to the same value

- Within the hash function h():
  - Rare, the probability of 2 keys hashing to the same value is 1/4B.
- Within the compression function c():
  - Common, 4B integers -> n values

Example: Hashing integers into an array with 8 cells

- h(i) = i
- c(i) = i % 8



insert(1, "Peter"): c(h(1)) = c(1) = 1

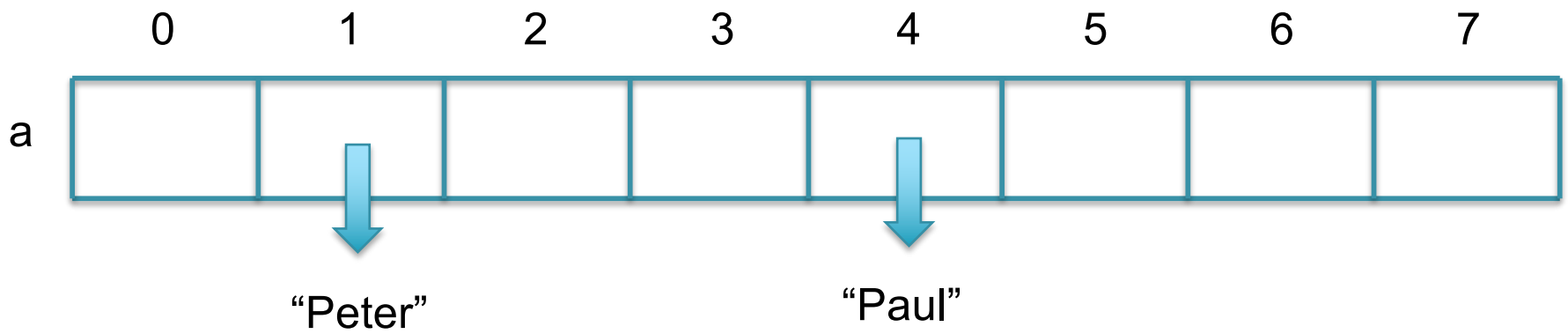
Collisions

Collisions occur when 2 different keys get mapped to the same value

- Within the hash function h():
  - Rare, the probability of 2 keys hashing to the same value is 1/4B.
- Within the compression function c():
  - Common, 4B integers -> n values

Example: Hashing integers into an array with 8 cells

- h(i) = i
- c(i) = i % 8



insert(20, "Paul"): c(h(20)) = c(20) = 4

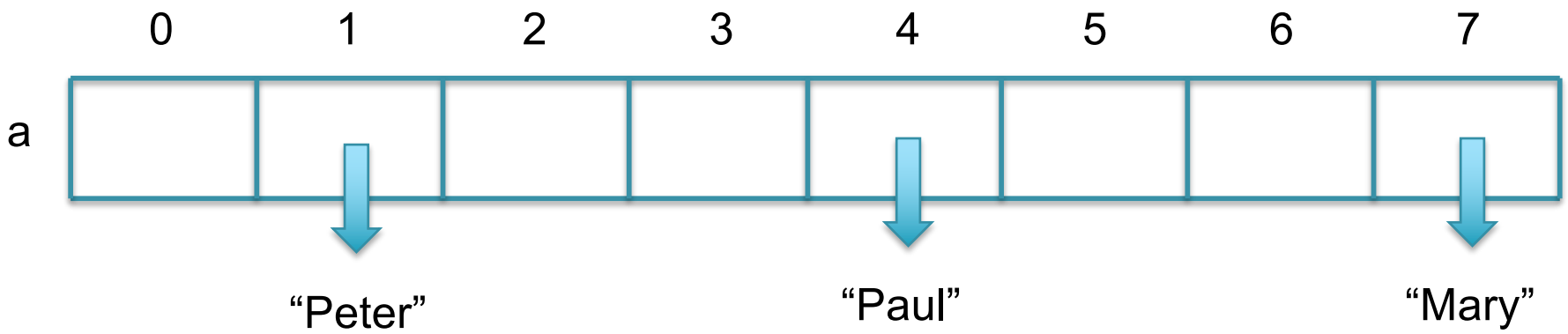
Collisions

Collisions occur when 2 different keys get mapped to the same value

- Within the hash function h():
  - Rare, the probability of 2 keys hashing to the same value is 1/4B.
- Within the compression function c():
  - Common, 4B integers -> n values

Example: Hashing integers into an array with 8 cells

- h(i) = i
- c(i) = i % 8



insert(15, “Mary”): c(h(15)) = c(15) = 7

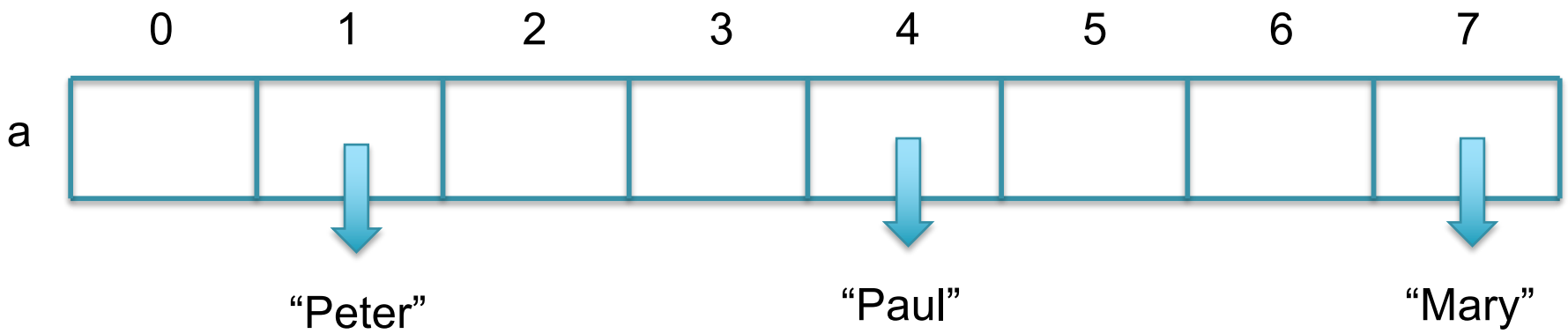
Collisions

Collisions occur when 2 different keys get mapped to the same value

- Within the hash function h():
  - Rare, the probability of 2 keys hashing to the same value is 1/4B.
- Within the compression function c():
  - Common, 4B integers -> n values

Example: Hashing integers into an array with 8 cells

- h(i) = i
- c(i) = i % 8



get(15): get(c(h(15))) = get(c(15)) = get(7) => "Mary" ☺

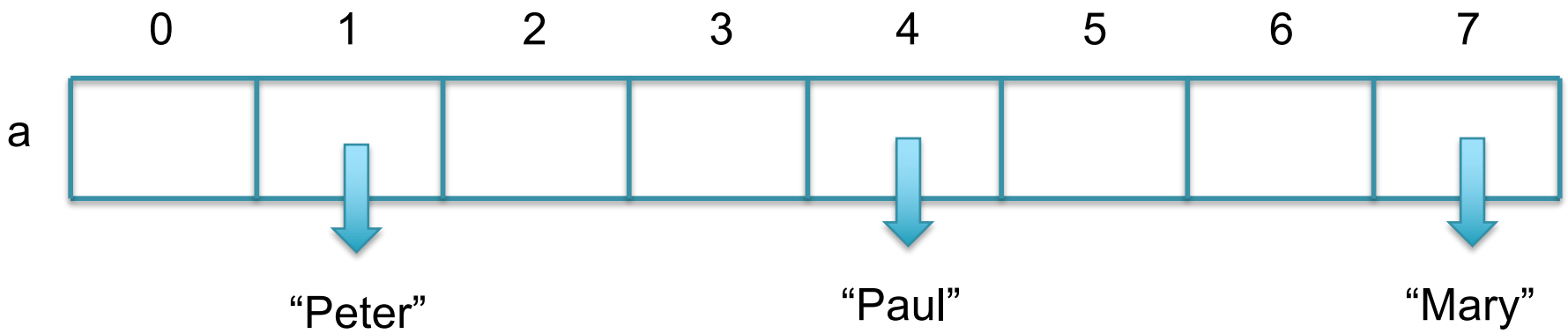
Collisions

Collisions occur when 2 different keys get mapped to the same value

- Within the hash function h():
  - Rare, the probability of 2 keys hashing to the same value is 1/4B.
- Within the compression function c():
  - Common, 4B integers -> n values

Example: Hashing integers into an array with 8 cells

- h(i) = i
- c(i) = i % 8



has(4): has(c(h(4)) = has(c(4)) = has(4) => "Paul" ☹

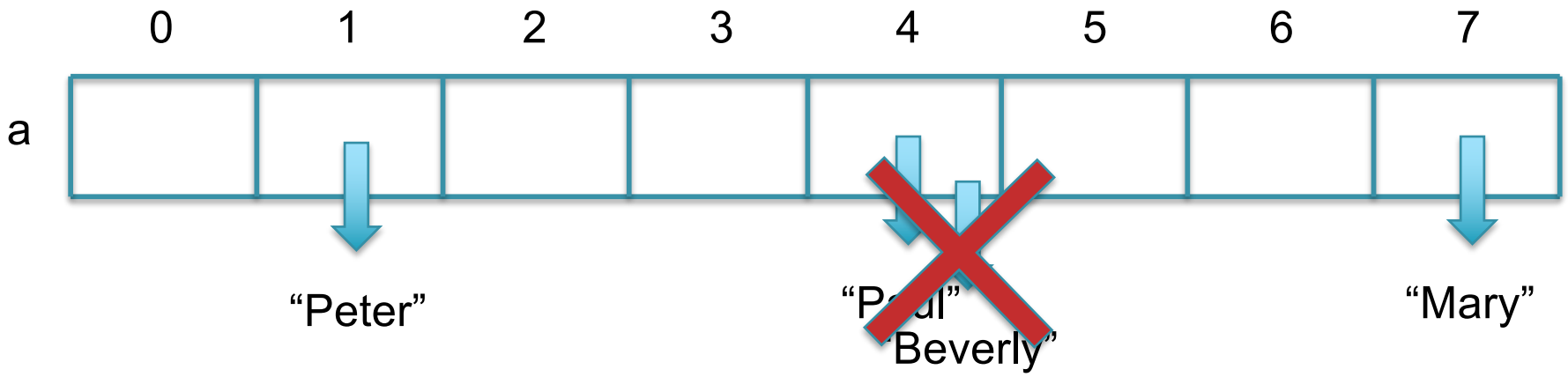
Collisions

Collisions occur when 2 different keys get mapped to the same value

- Within the hash function h():
  - Rare, the probability of 2 keys hashing to the same value is 1/4B.
- Within the compression function c():
  - Common, 4B integers -> n values

Example: Hashing integers into an array with 8 cells

- h(i) = i
- c(i) = i % 8



insert(4, "Beverly"): c(h(4)) = c(4) = 4 ☹

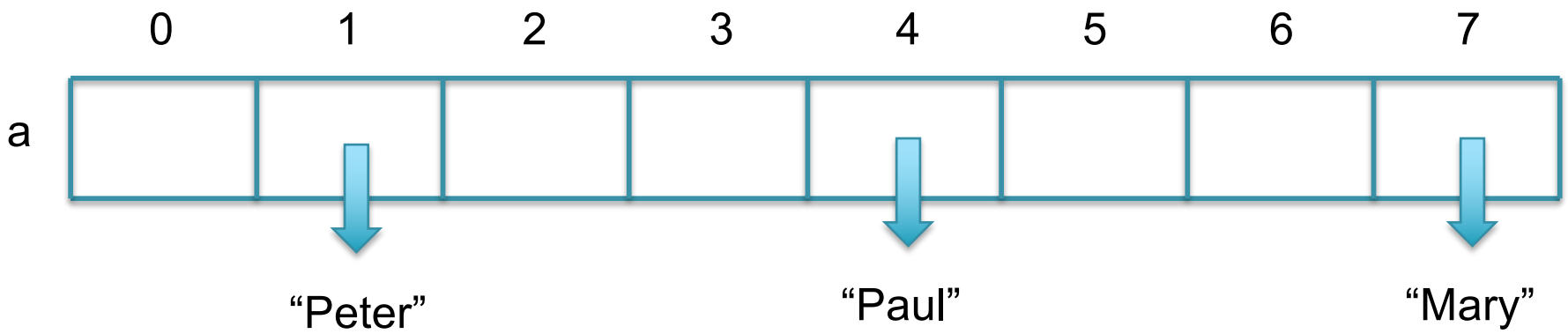
Collisions

Collisions occur when 2 different keys get mapped to the same value

- Within the hash function h():
  - Rare, the probability of 2 keys hashing to the same value is 1/4B.
- Within the compression function c():
  - Common, 4B integers -> n values

Example: Hashing integers into an array with 8 cells

- h(i) = i
- c(i) = i % 8



Two problems caused by collisions:

False positives: How do we know the key is the one we want?

Collision Resolution: What do we do when 2 keys map to same location?

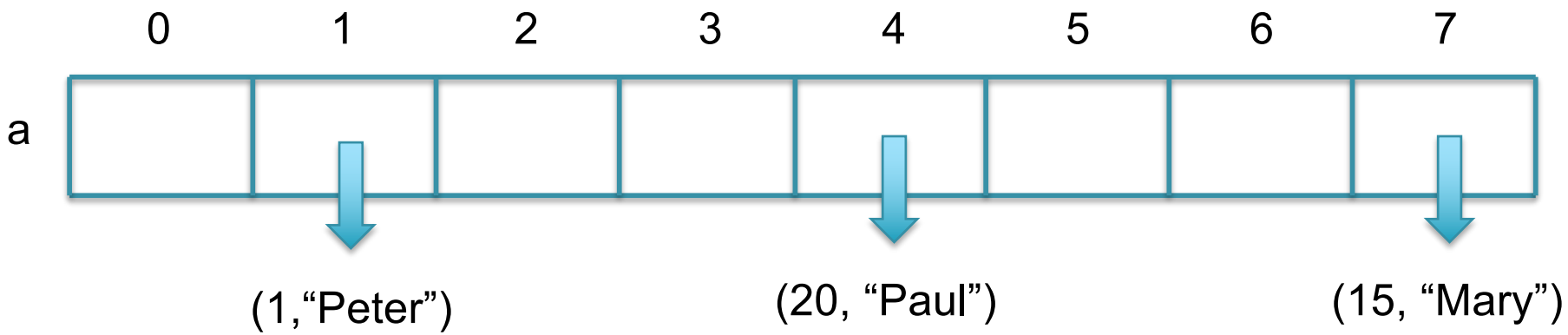
Collisions

Collisions occur when 2 different keys get mapped to the same value

- Within the hash function h():
  - Rare, the probability of 2 keys hashing to the same value is 1/4B.
- Within the compression function c():
  - Common, 4B integers -> n values

Example: Hashing integers into an array with 8 cells

- h(i) = i
- c(i) = i % 8



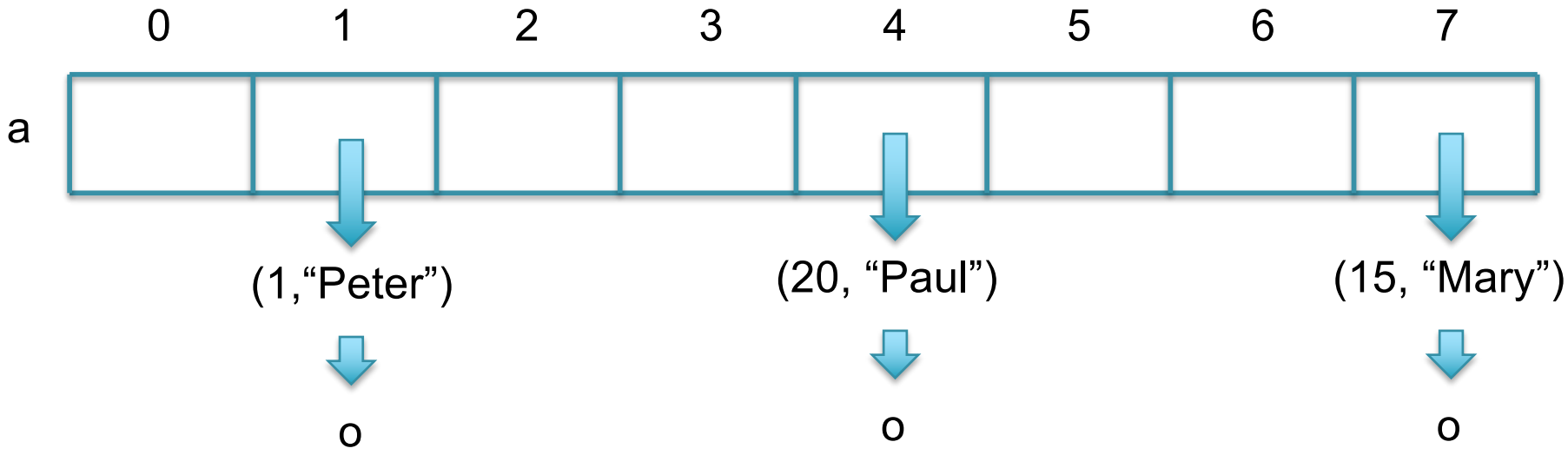
Two problems caused by collisions:

False positives: How do we know the key is the one we want?

Collision Resolution: What do we do when 2 keys map to same location?

Separate chaining

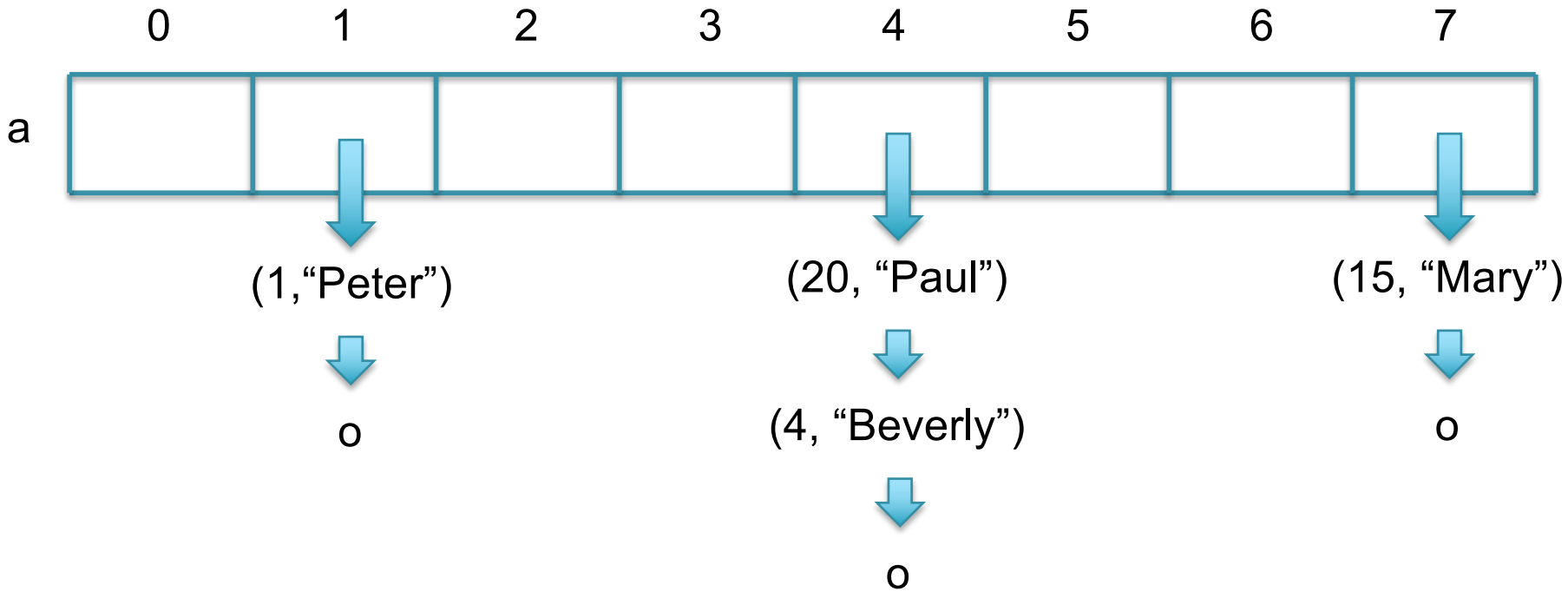
Use Array<List<V>> instead of an Array<V> to store the entries



insert(4, “Beverly”): c(h(4)) = c(4) = 4

Separate chaining

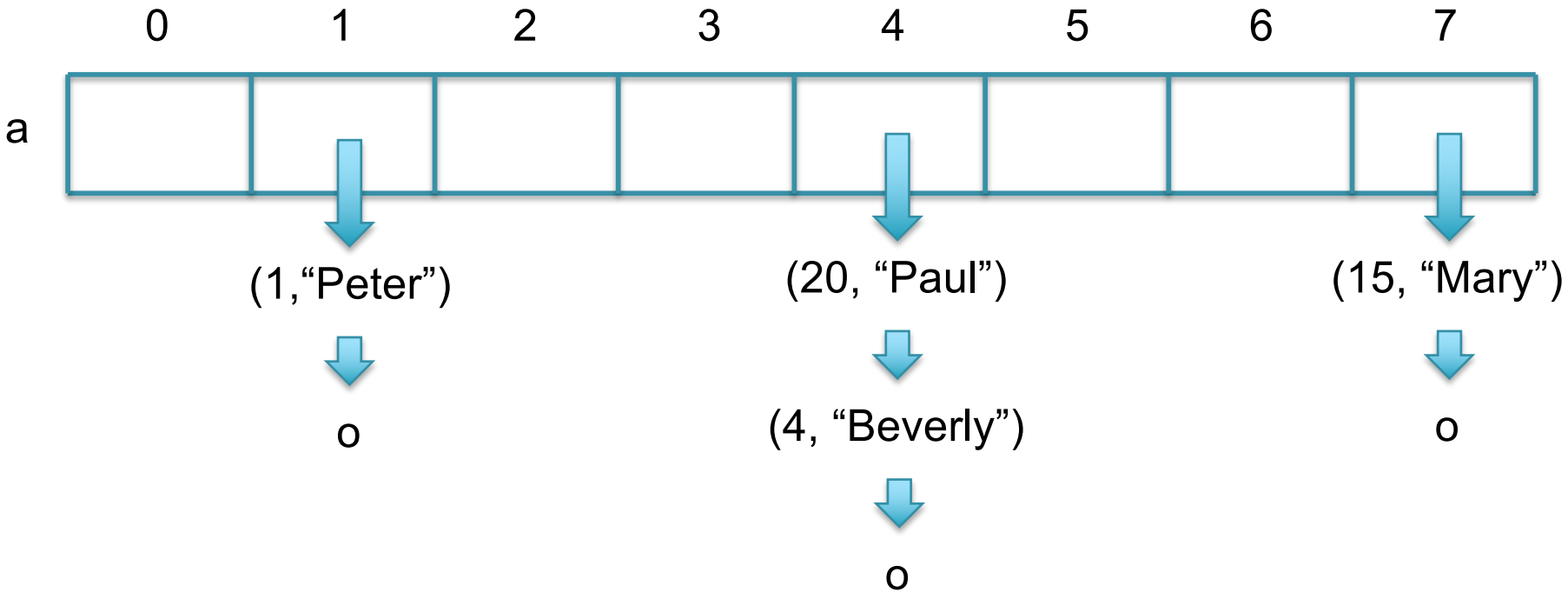
Use Array<List<V>> instead of an Array<V> to store the entries



insert(4, “Beverly”): c(h(4)) = c(4) = 4

Separate chaining

Use Array<List<V>> instead of an Array<V> to store the entries



Using separate chaining we could implement the Map<K,V> interface ☺

Seems fast, but how fast do we expect it to be?

Separate Chaining Analysis

Assume the table has just 1 cell:

All n items will be in a linked list => O(n) insert/find/remove ☹

Assume the hash function h() always returns a constant value

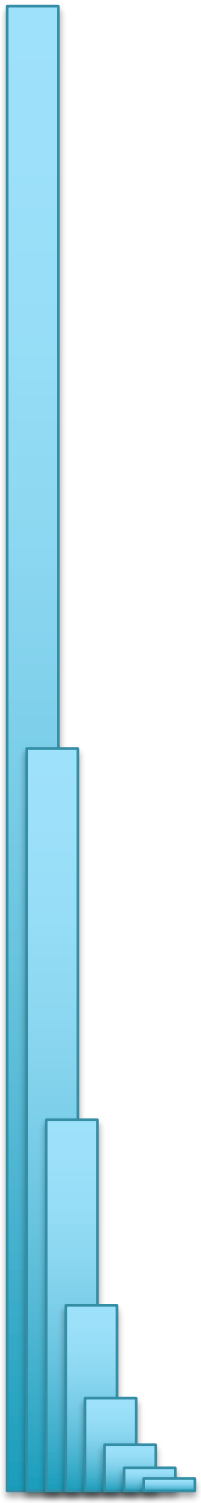
All n items will be in a linked list => O(n) insert/find/remove ☹

Assume table has m cells AND h() evenly distributes the keys

- Every cell is equally likely to be selected to store key k, so the n items should be evenly distributed across m slots
- Average number of items per slot: n/m
  - Also called the ***load factor*** *(*commonly written as α)
  - Also the probability of a collision when inserting a new key
    - Empty table: 0/m probability
    - After 1st item: 1/m
    - After 2nd item: 2/m

Expected time for unsuccessful search:	O(1+n/m)
Expected time for successful search:	O(1+n/m/2) => O(1+n/m)

If n < c*m, then we can expect constant time! ☺

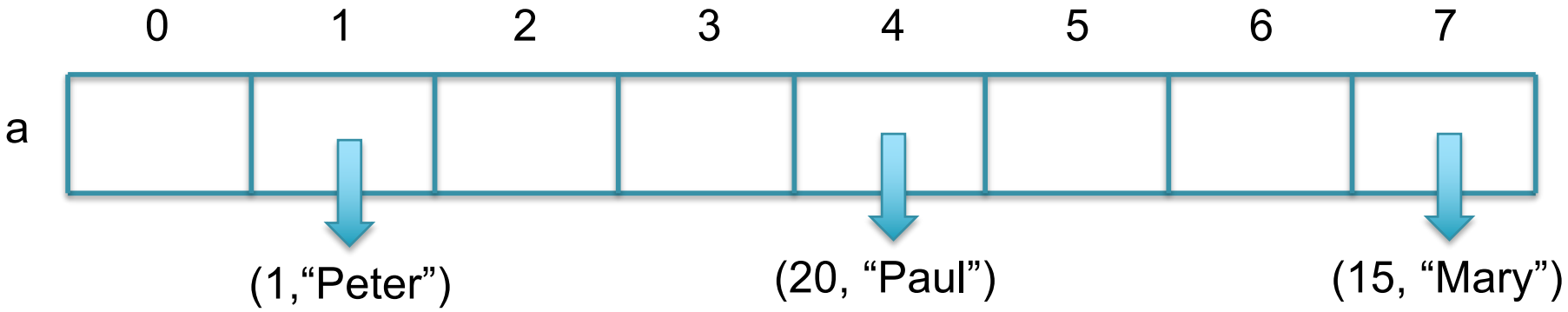


Part 2: Alternate Collision Strategies aka Open addressing

Alt 1: Linear Probing

Fixed overhead for objects, rather than many dynamic lists

Use an Array<V> to store the entries,
but go to next available cell on collision.



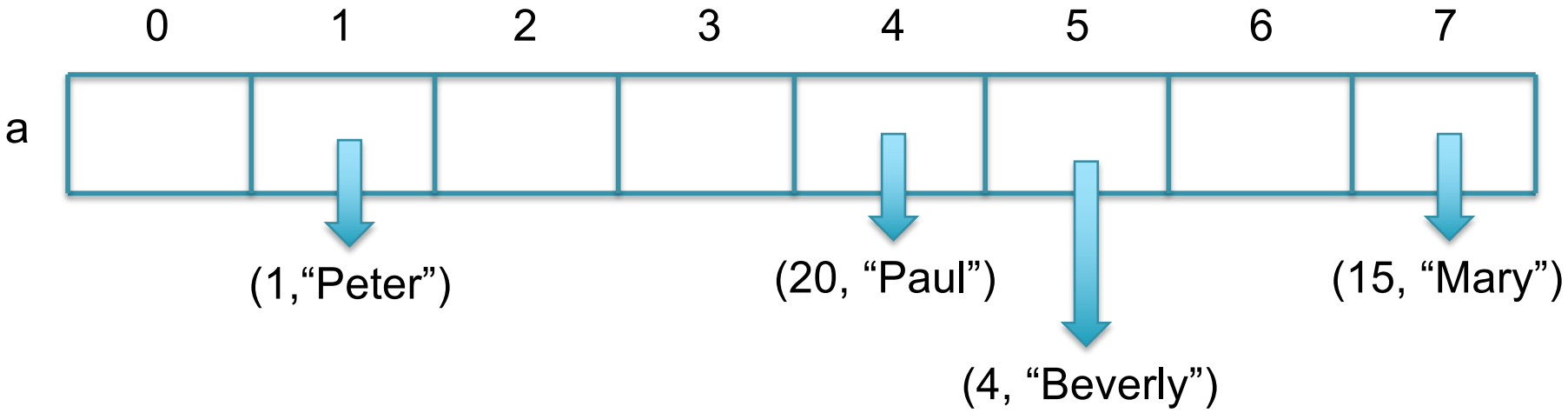
insert(4, “Beverly”): c(h(4)) = c(4) = 4

A[4] is already filled, so go to A[5]

Alt 1: Linear Probing

Fixed overhead for objects, rather than many dynamic lists

Use an Array<V> to store the entries,
but go to next available cell on collision.



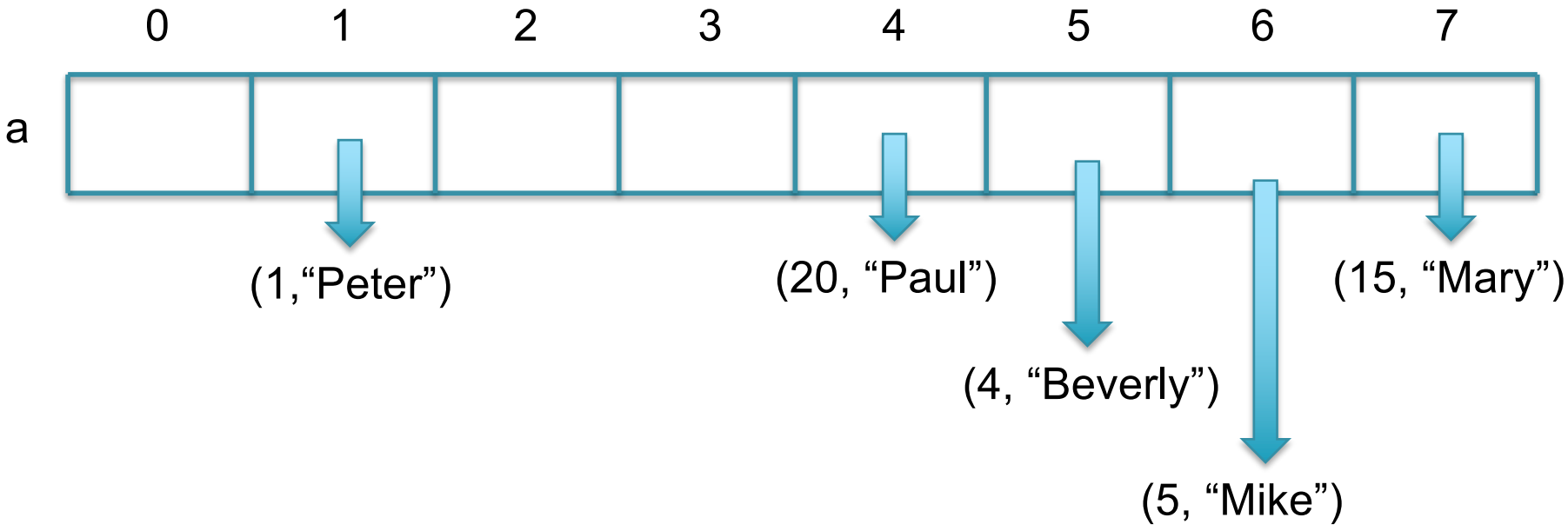
Insert(5, “Mike”): c(h(5)) = c(5) = 5

A[5] is already filled, so go to A[6]

Alt 1: Linear Probing

Fixed overhead for objects, rather than many dynamic lists

Use an Array<V> to store the entries,
but go to next available cell on collision.



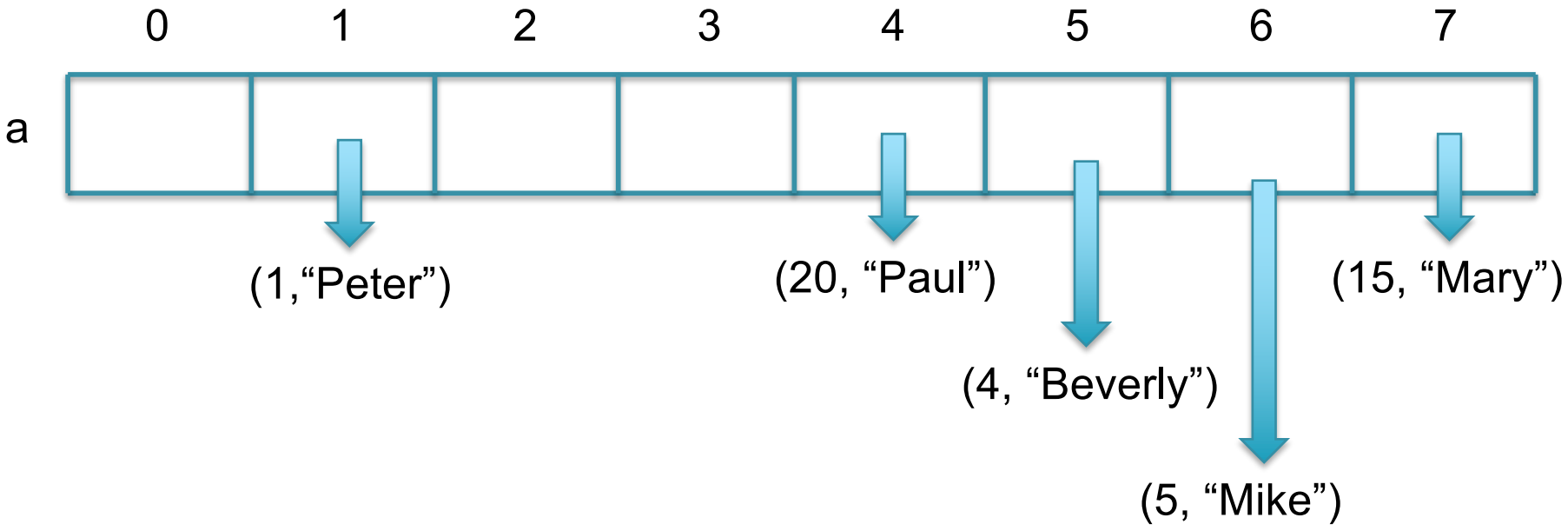
Insert(5, “Mike”): c(h(5)) = c(5) = 5

A[5] is already filled, so go to A[6]

Alt 1: Linear Probing

Fixed overhead for objects, rather than many dynamic lists

Use an Array<V> to store the entries,
but go to next available cell on collision.



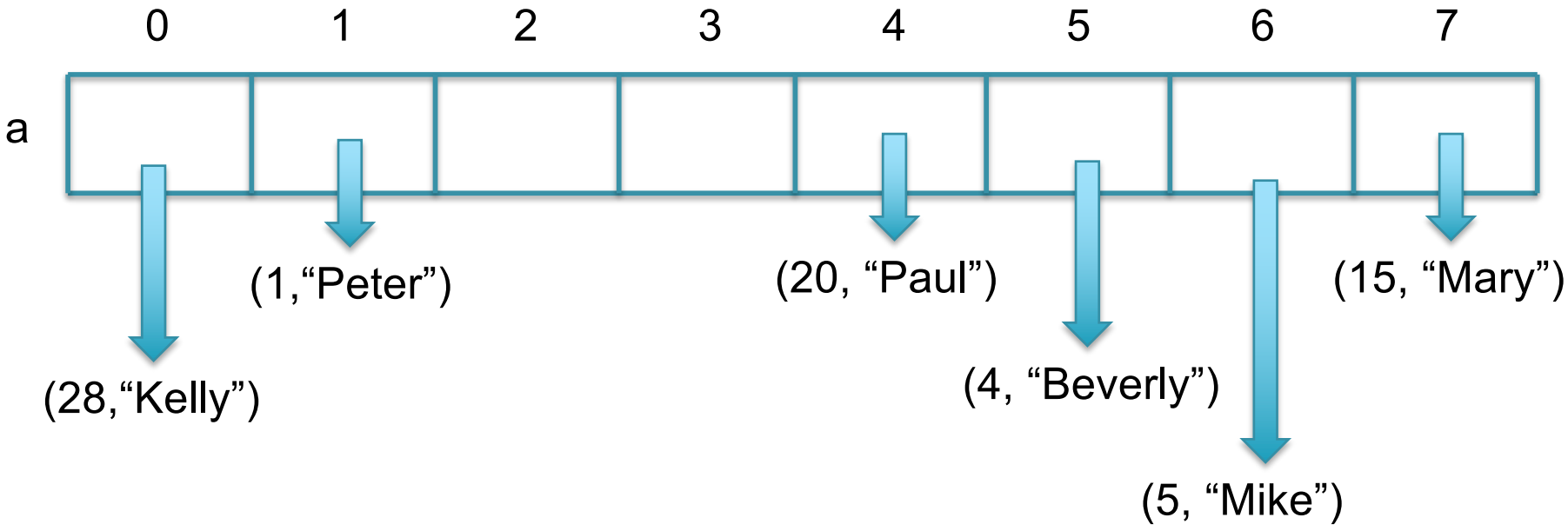
Insert(28, “Kelly”): c(h(28)) = c(4) = 4

A[4] is already filled, so go to ???

Alt 1: Linear Probing

Fixed overhead for objects, rather than many dynamic lists

Use an Array<V> to store the entries,
but go to next available cell on collision.

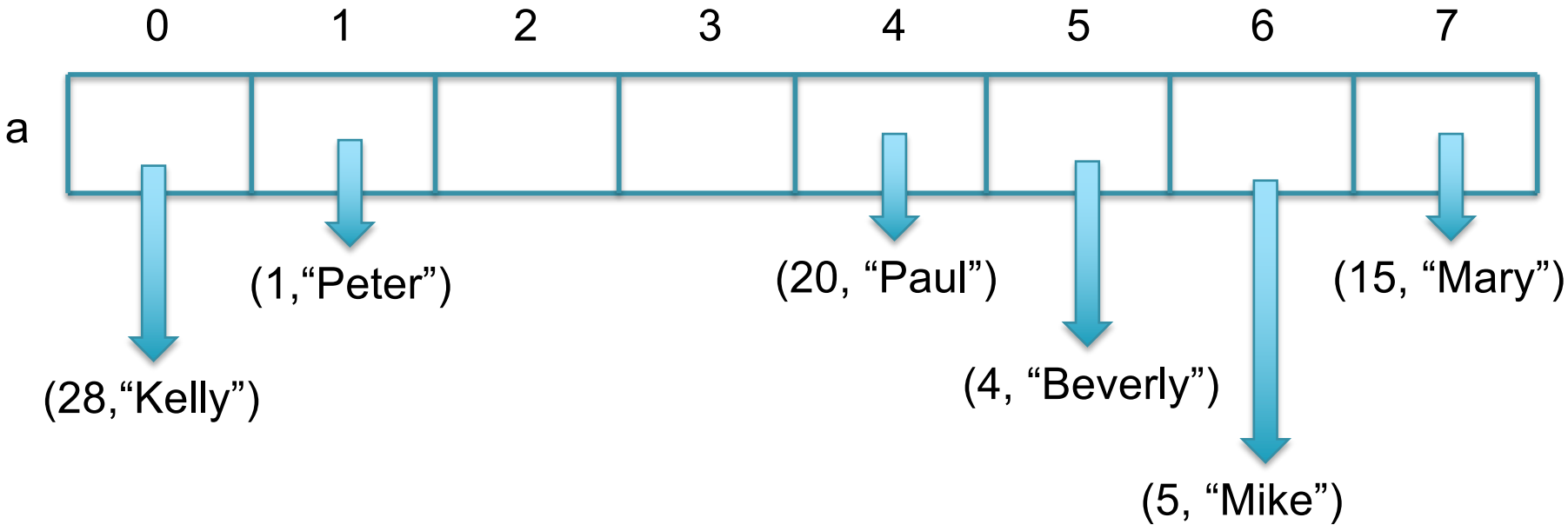


A[4] is already filled, so go to A[0]

Alt 1: Linear Probing

Fixed overhead for objects, rather than many dynamic lists

Use an Array<V> to store the entries,
but go to next available cell on collision.

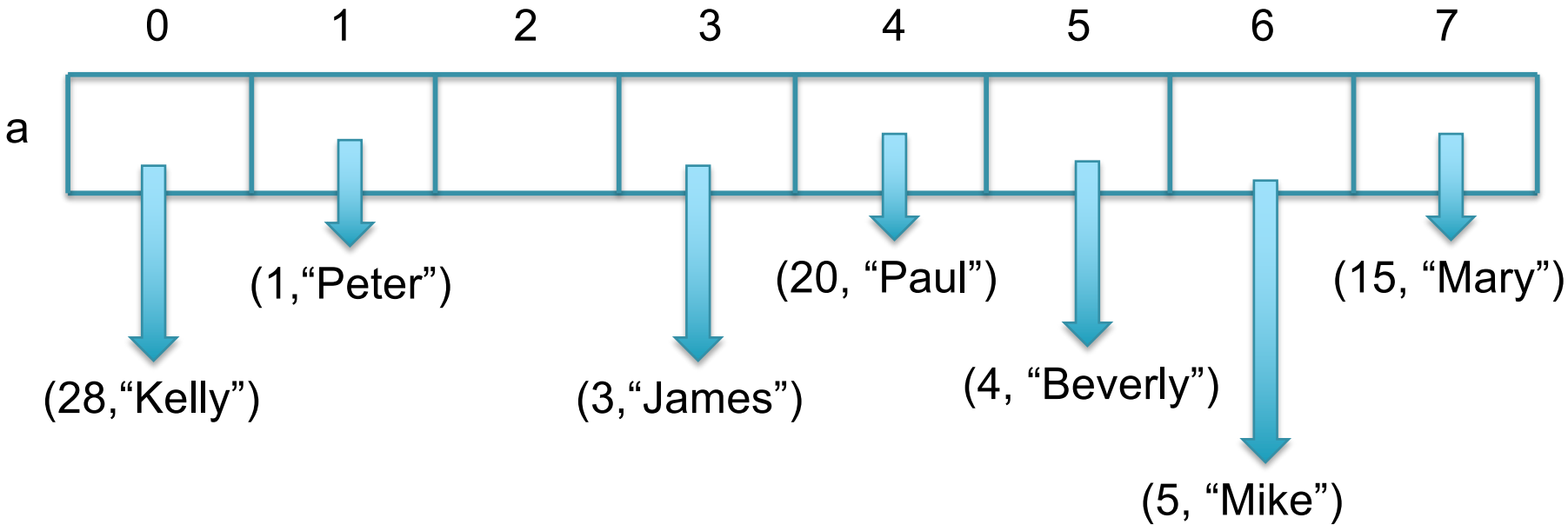


Insert(3, “James”)

Alt 1: Linear Probing

Fixed overhead for objects, rather than many dynamic lists

Use an Array<V> to store the entries,
but go to next available cell on collision.

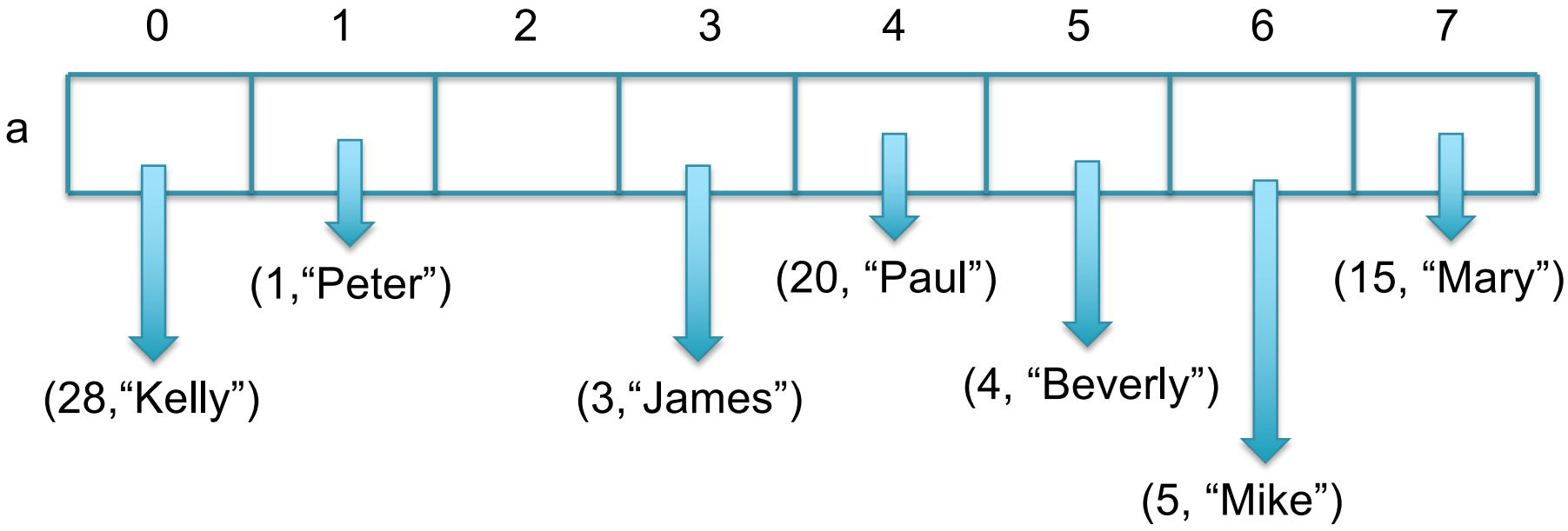


Insert(3, “James”)

Alt 1: Linear Probing

Fixed overhead for objects, rather than many dynamic lists

Use an Array<V> to store the entries,
but go to next available cell on collision.

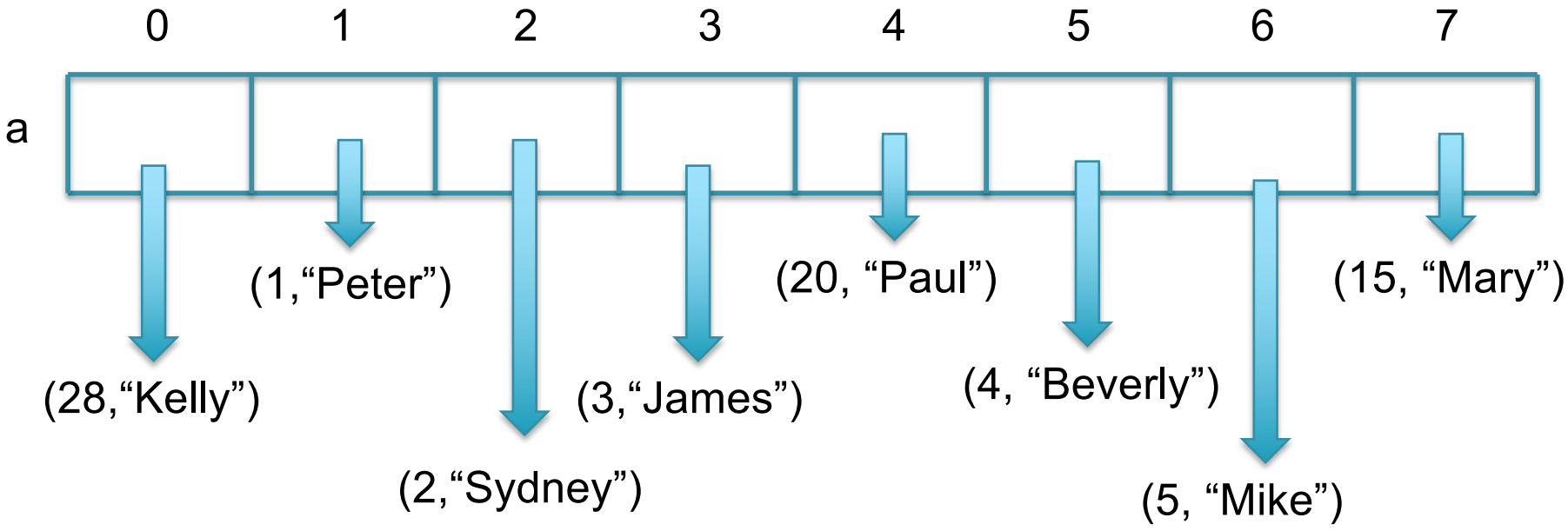


Insert(2, “Sydney”)

Alt 1: Linear Probing

Fixed overhead for objects, rather than many dynamic lists

Use an Array<V> to store the entries,
but go to next available cell on collision.

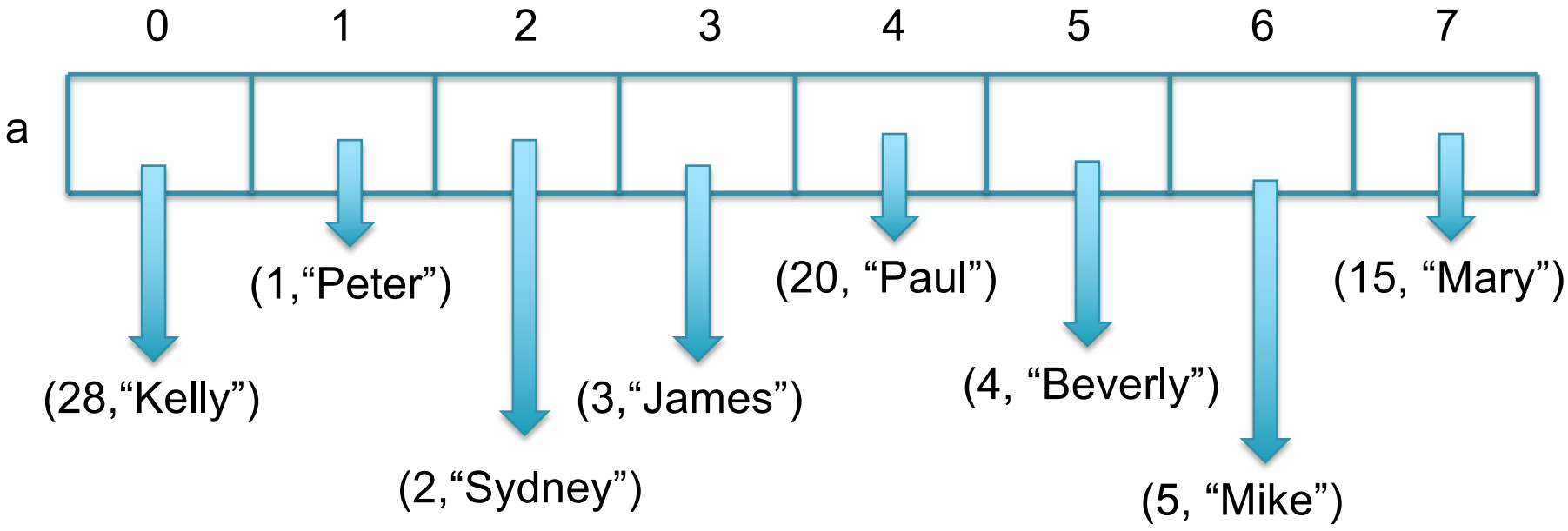


Insert(2, “Sydney”)

Alt 1: Linear Probing

Fixed overhead for objects, rather than many dynamic lists

Use an Array<V> to store the entries,
but go to next available cell on collision.



Insert(55, “Katherine”)

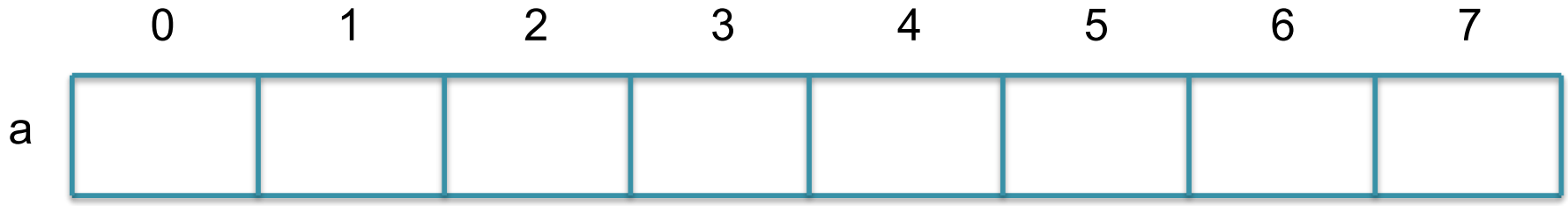
No place left to insert anything else!
A load factor of 1 is a hard bound

What should we do? | Array double & rehash | When? | Around 50% load

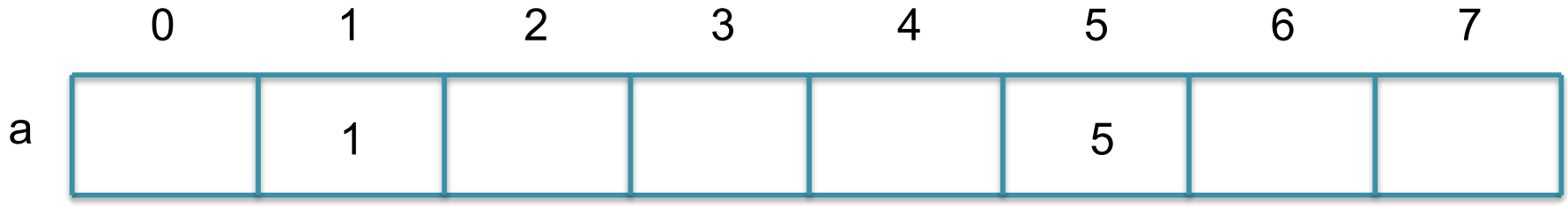
Primary clustering

Once you have a collision, you will have more and more collisions that grow into clusters around the first one

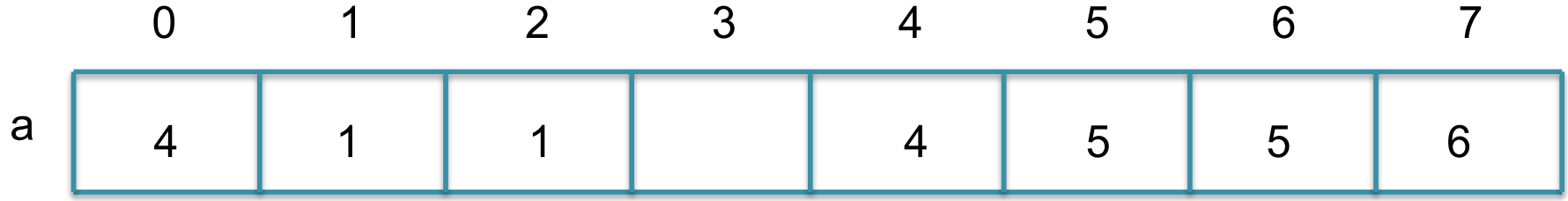
Empty Table



Some Inserts: 1, 5



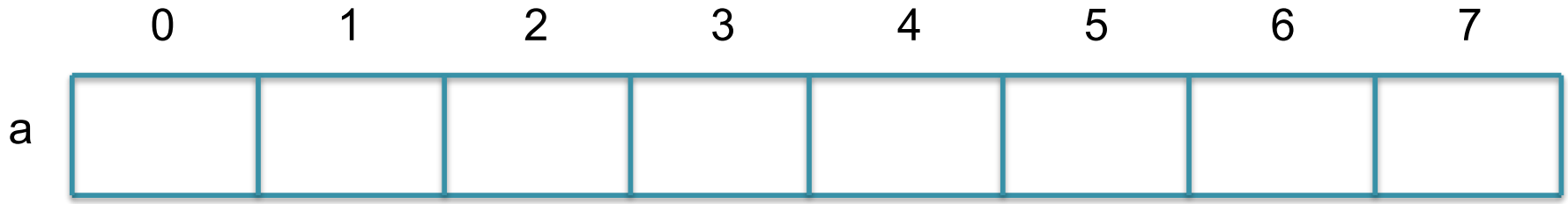
More Inserts: 1, 5, 1, 4, 5, 6, 4



Primary clustering

Once you have a collision, you will have more and more collisions that grow into clusters around the first one

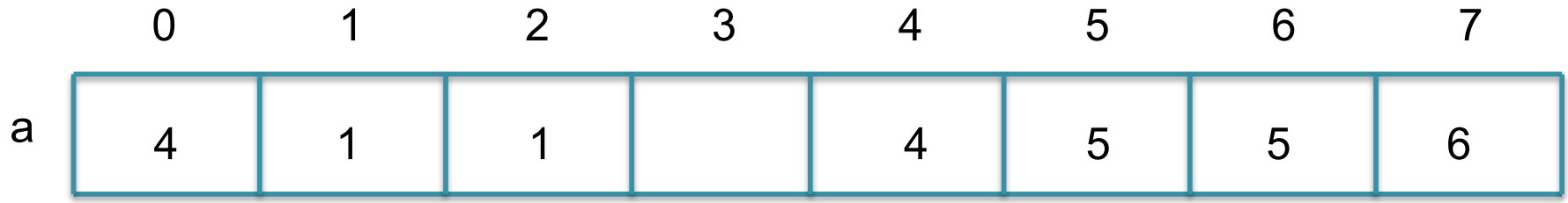
Empty Table



Some Inserts: 1, 5

Notice that the next available cell for h = 0, 1, 2, 3, 4, 5, 6, or 7 is slot 3!

More Inserts: 1, 5, 1, 4, 5, 6, 4



Primary clustering

Once you have a collision, you will have more and more collisions that grow into clusters around the first one

Empty Table

0 1 2 3 4 5 6 7

How would we look for key 12?

Start looking in 12 % 8, and keep checking until the first empty cell data[4].key, data[5].key, data[6].key, data[7].key, data[0].key, data[1].key, data[2].key, data[3].key => not there!

More Inserts: 1, 5, 1, 4, 5, 6, 4

	0	1	2	3	4	5	6	7
a	4	1	1		4	5	5	6

Primary clustering

Once you have a collision, you will have more and more collisions that grow into clusters around the first one

Empty Table

"Primary clustering" means that if a key hashes to anywhere inside an existing cluster, than the cluster will grow

As the load gets higher and higher, your searches will get longer and longer since the keys will interfere with each other

Run time will be much worse than O(1+n/m) at high load

More Inserts: 1, 5, 1, 4, 5, 6, 4

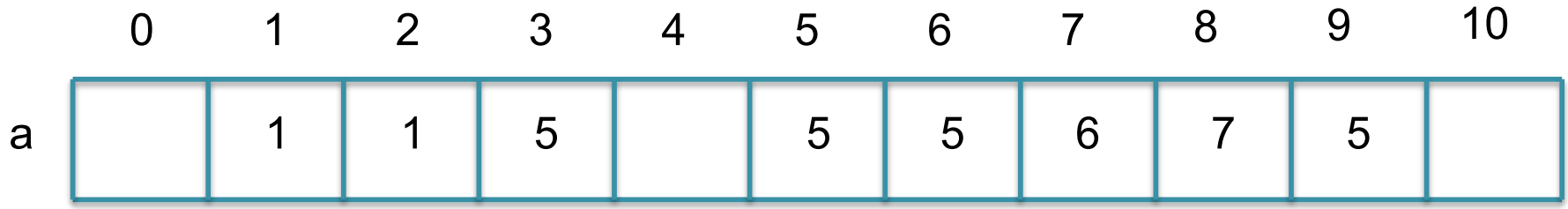
	0	1	2	3	4	5	6	7
a	4	1	1		4	5	5	6

Alt 2: Quadratic Probing

Use a different probe sequence to create gaps between elements

- Quadratic probing: 1, 4, 9, 16, 25… (note: 1, 2^2, 3^2, 4^2, 5^2, …)
- Requires that the table size is a prime number & keep load factor below 0.5 to ensure the sequence will cover half of the slots in the table

Insert this sequence: 1, 5, 1, 5, 6, 7, 5, 5



Spreads out the keys better than linear probing, but still subject to "secondary clustering" meaning the cluster grows if keys hash to the same starting value (else cluster probably wont grow)

Alt 3: Double Hashing

Determine the step-size for probing from the key itself

- Map from the key to a small integer -> double hashing ☺
- Often gives better spread than linear or quadratic probing

Have to be careful to select the step-size that will visit (nearly) every cell:

- Hash table of size 8 with a step size of 2 only visits even or odd slots ☹

Option 1: Primes

- Select hash table size to be some prime p > 2
- Secondary hash function produces values in the range [1..p-1]
- Example: p=7
  - +1 visits every slot
  - +2 visits 0, 2, 4, 6, 1, 3, 5, 0
  - +3 visits 0, 3, 6, 2, 5, 1, 4, 0

Option 2: Powers of 2

- Select hash table size to be $t=2^x$ for some x
- Secondary hash function produces odd values in the range $[1..2^x-1]$
- Example: p=8
  - +1 visits every slot
  - +3 visits 0, 3, 1, 4, 7, 2, 5, 0
  - …

Alt 3: Double Hashing

Determine the step-size for probing from the key itself

- Map from the key to a small integer -> double hashing ☺
- Often gives better spread than linear or quadratic probing

Have to be careful to select the step-size that will visit (nearly) every cell:

- Hash table of size 8 with a step size of 2 only visits even or odd slots ☹

Option 1: Primes

- Select hash table size to be some prime p > 2
- Secondary hash function produces values in the range [1..p-1]
- Example: p=7
  - +1 visits every slot
  - +2 visits 0, 2, 4, 6, 1, 3, 5, 0
  - +3 visits 0, 3, 6, 2, 5, 1, 4, 0

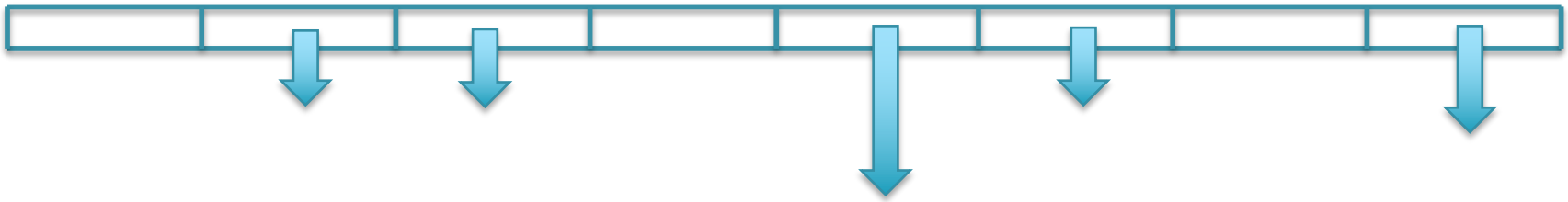
Option 2: Powers of 2

- Select hash table size to be $t=2^x$ for some x
- Secondary hash function produces odd values in the range $[1..2^x-1]$
- Example: p=8
  - +1 visits every slot
  - +3 visits 0, 3, 1, 4, 7, 2, 5, 0
  - …

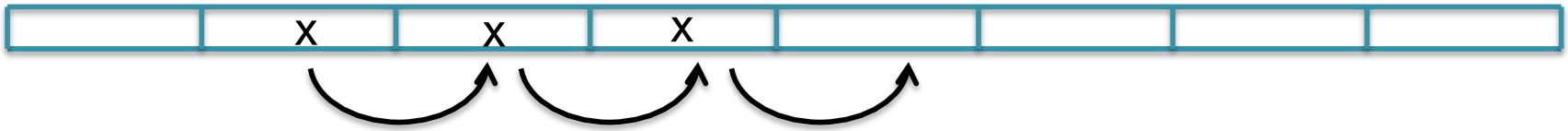
Also tricky to implement generically in Java since we don't know the type. Object.hashCode() returns an int

Collision Strategies

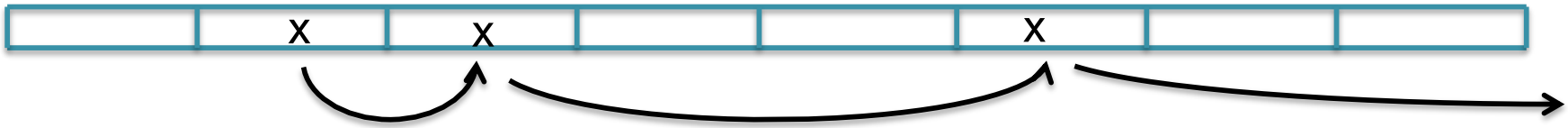
1. Separate chaining:
More object overhead, but degrades gracefully as load approaches 1



2. Linear Probing
Minimal overhead, easy to implement, but tends to form large clusters



3. Quadratic Probing
Slightly more complex, but skips over larger and larger amounts to find holes

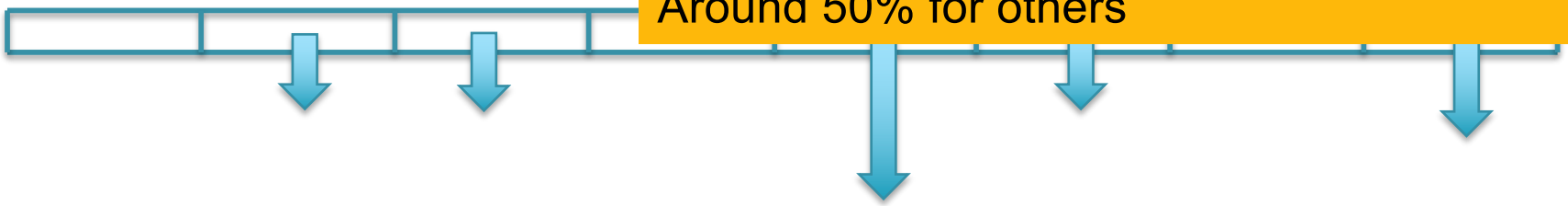


4. Double Hashing
Stride length depends on h_2(key)

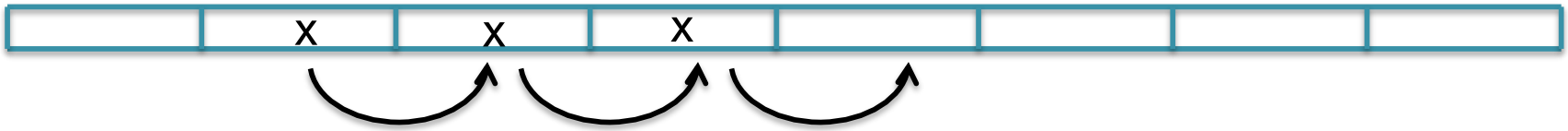
Collision Strategies

1. Separate chaining:
More object overhead, but degrad

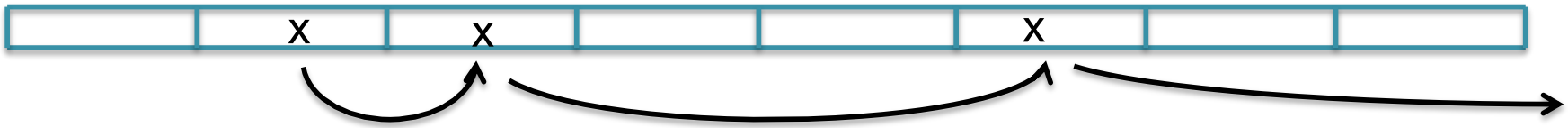
Rule of thumb:
Resize at ~90% load for separate chaining,
Around 50% for others



2. Linear Probing
Minimal overhead, easy to implement, but tends to form large clusters



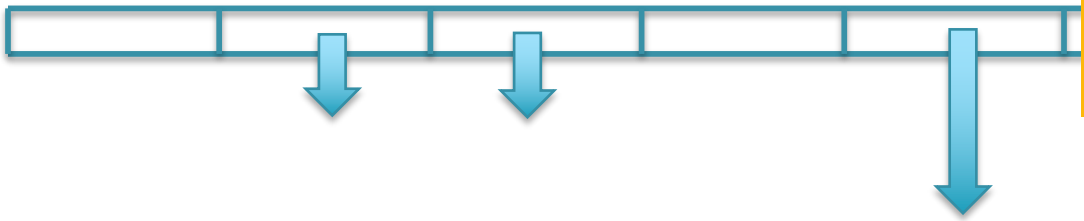
3. Quadratic Probing
Slightly more complex, but skips over larger and larger amounts to find holes



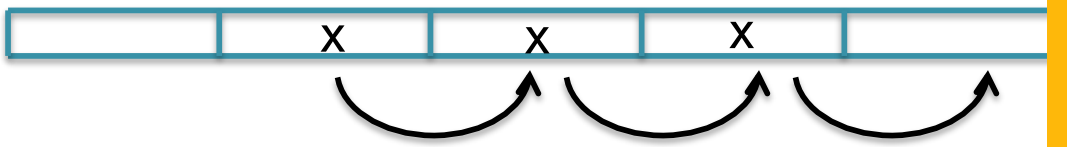
4. Double Hashing
Stride length depends on h_2(key)

Removing elements

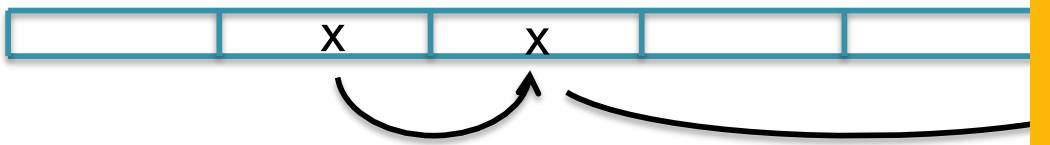
1. Separate chaining:
More object overhead, but degrades gracefully as l



2. Linear Probing
Minimal overhead, easy to implement, but tends to



3. Quadratic Probing
Slightly more complex, but skips over larger and l



4. Double Hashing
Stride length depends on h_2(key)

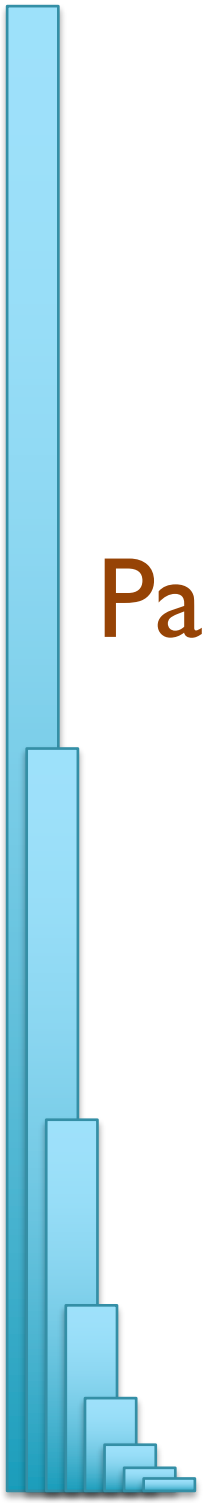
Separate Chaining:

Just remove the element from the appropriate bucket

Open Addressing:

Very tricky, may have to move many elements around

Often easier to just mark the element as deleted and periodically rehash everything once too many nodes have been deleted (aka Lazy deletion)



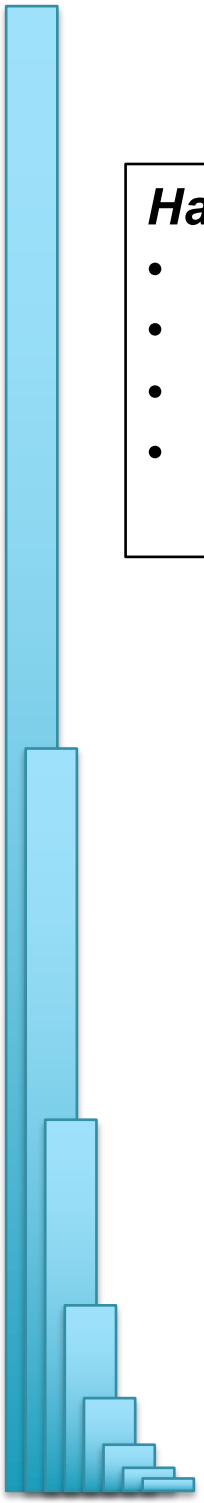
Part 4: Hash functions, hash tables, and other practicalities ☺

Hash Functions

Hash Function enables Map<K, V> as Map<Integer, V> as Array<V>

- h(): K -> Integer for any possible key K
- h() should distribute the keys uniformly over all integers
- if k_1 and k_2 are "close", $h(k_1)$ and $h(k_2)$ should be "far" apart
- Should be fast, as we are aiming for O(1) performance
  - Unlike cryptographic hashes that are slow but give very few collisions

How do we apply a mathematical function to non-mathematical keys (String, Color, Student, Fruit, …)?



ASCII Table

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

How should we hash strings?

(Bad) String Hash Functions

Hash Function enables Map<K, V> as Map<Integer, V> as Array<V>

- h(): K -> Integer for any possible key K
- h() should distribute the keys uniformly over all integers
- if k_1 and k_2 are "close", $h(k_1)$ and $h(k_2)$ should be "far" apart
- Should be fast, as we are aiming for O(1) performance
  - Unlike cryptographic hashes that are slow but give very few collisions

Hash of String: First char

- h(s) = (int) s.at(0)
- Super simple, but does not map to all possible integers, does not separate out values

Hash of String: sum of chars

$$h(s) = \sum_{i=0}^{s.length()-1} s.at(i)$$

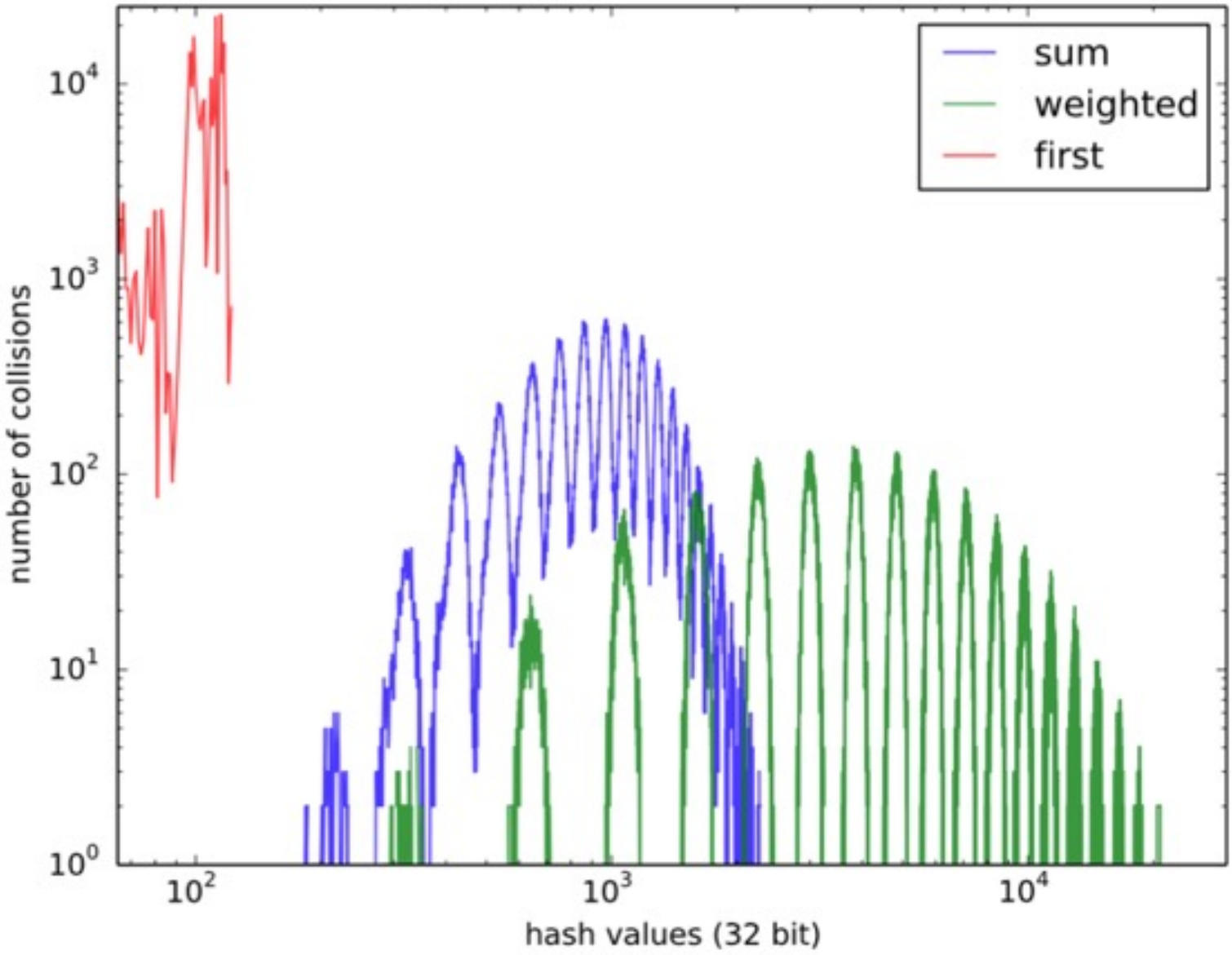
- Simple, but anagrams map to same value (stop, pots, tops, ..)

Hash of String: weighted sum of chars

$$h(s) = \sum_{i=0}^{s.length()-1} (i+1) \times s.at(i)$$

- Better: depends on both characters used and their order
- Small values for typical words

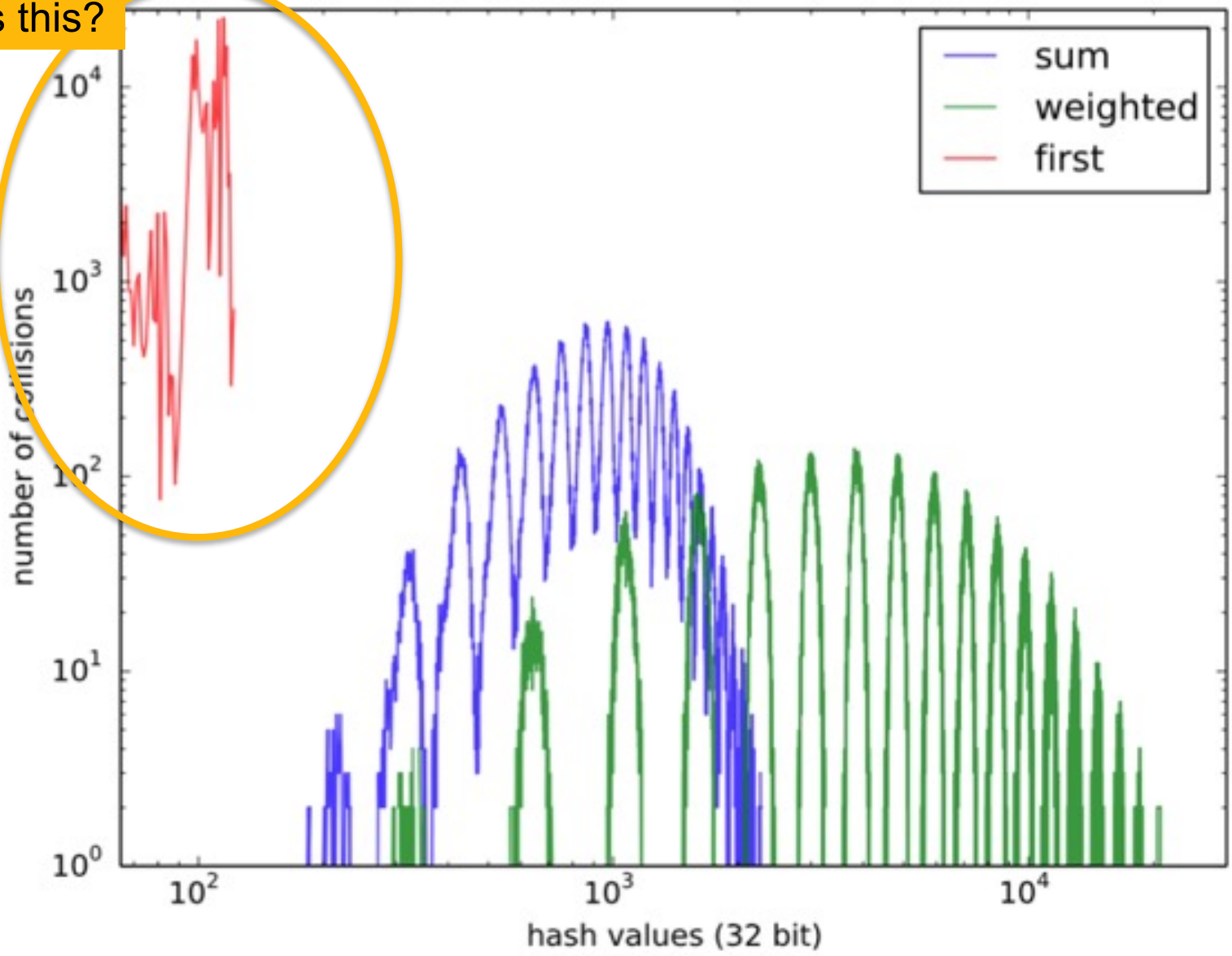
Bad Hash Function Performance



Hash every word in dictionary.txt and plot #collisions versus hash value

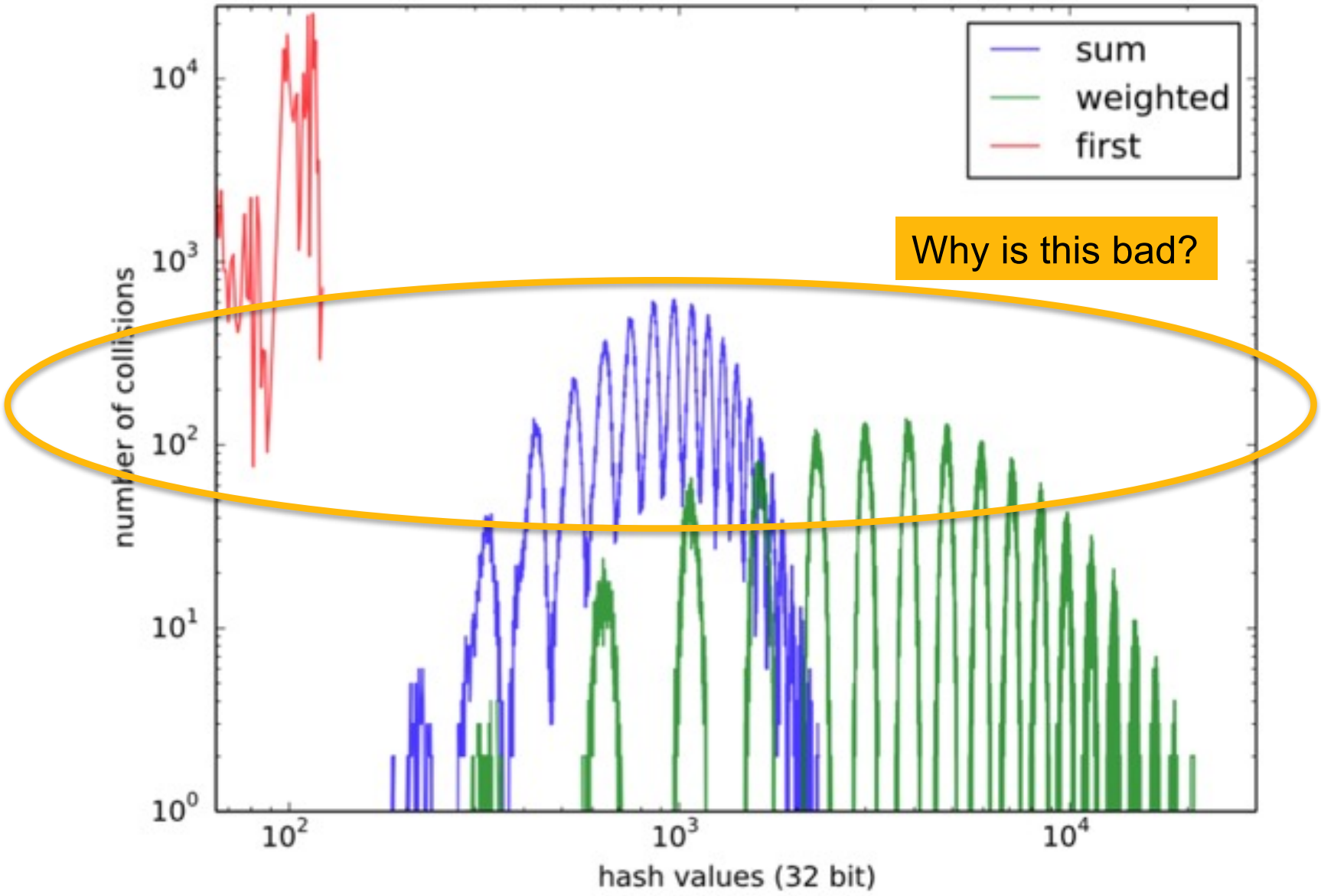
Bad Hash Function Performance

What is this?



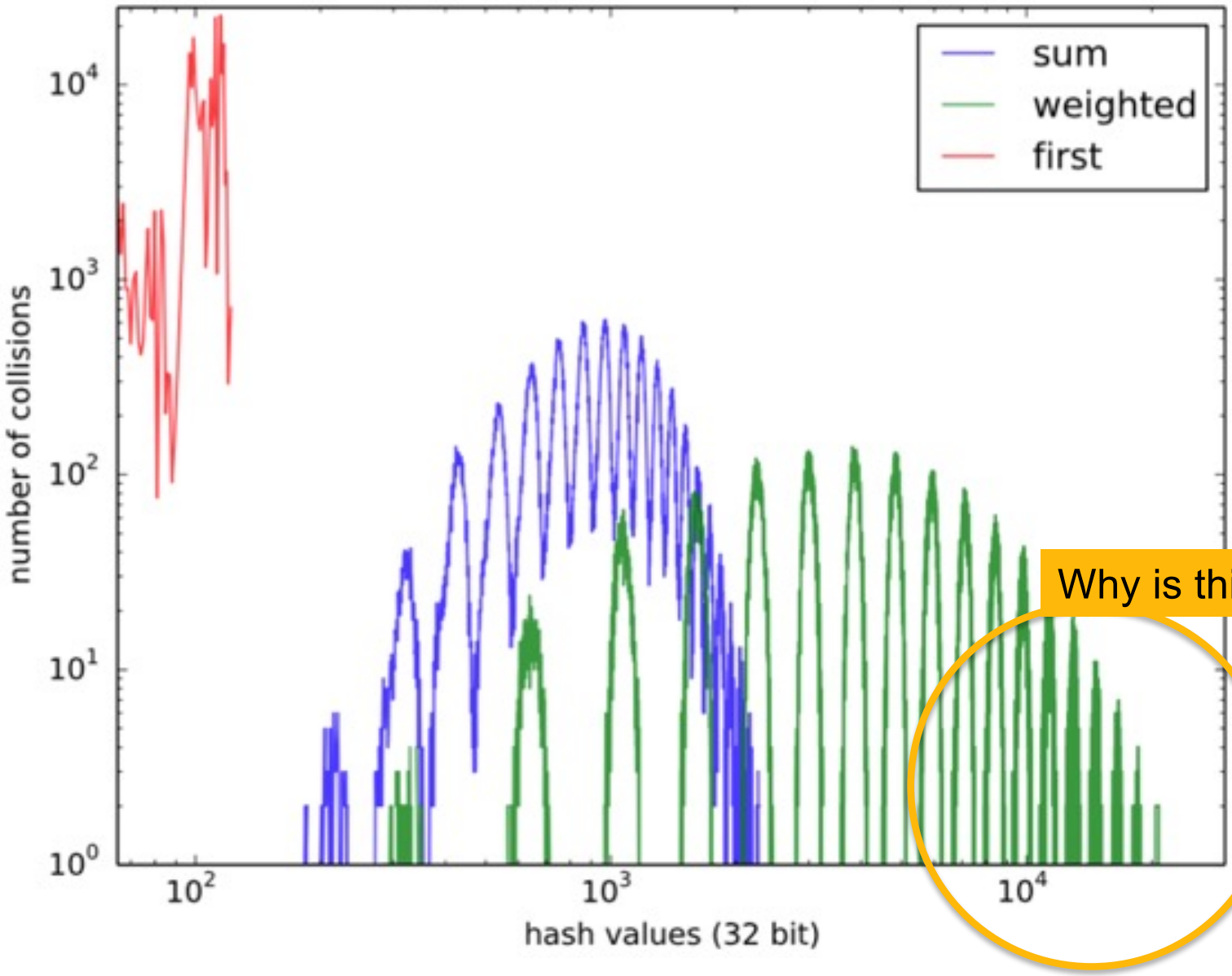
Hash every word in dictionary.txt and plot #collisions versus hash value

Bad Hash Function Performance



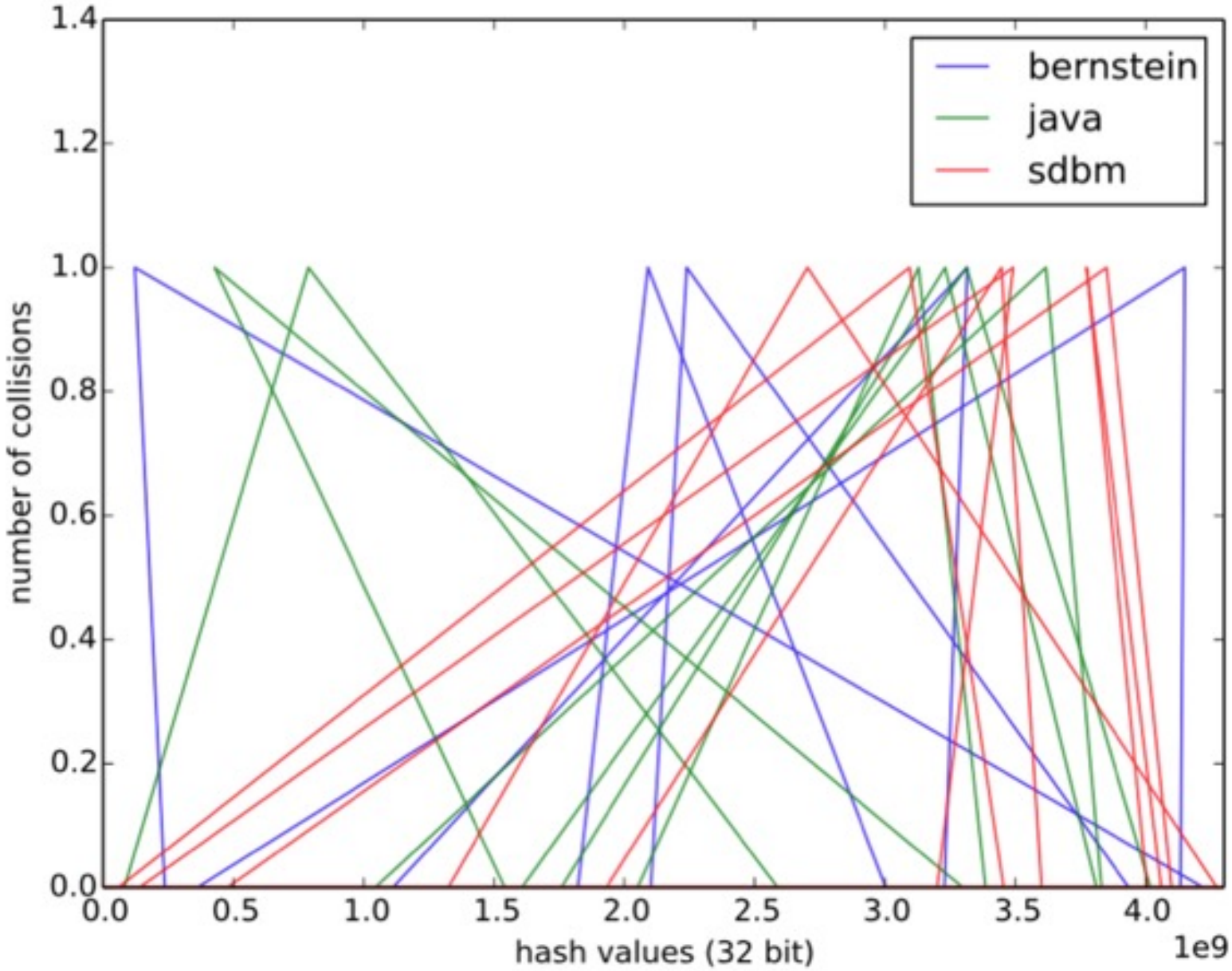
Hash every word in dictionary.txt and plot #collisions versus hash value

Bad Hash Function Performance



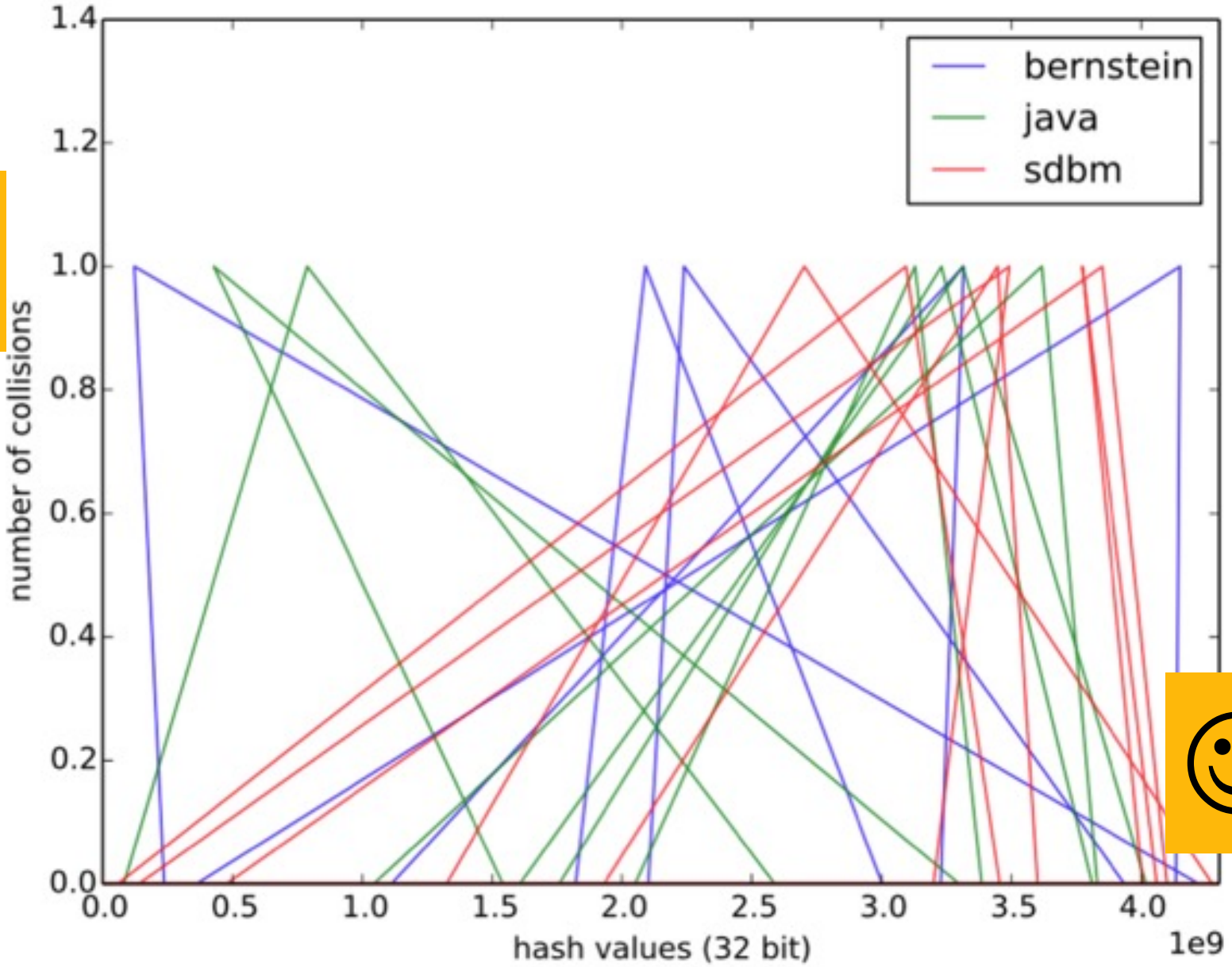
Hash every word in dictionary.txt and plot #collisions versus hash value

Good Hash Function Performance



Hash every word in dictionary.txt and plot #collisions versus hash value

Good Hash Function Performance



Hash every word in dictionary.txt and plot #collisions versus hash value

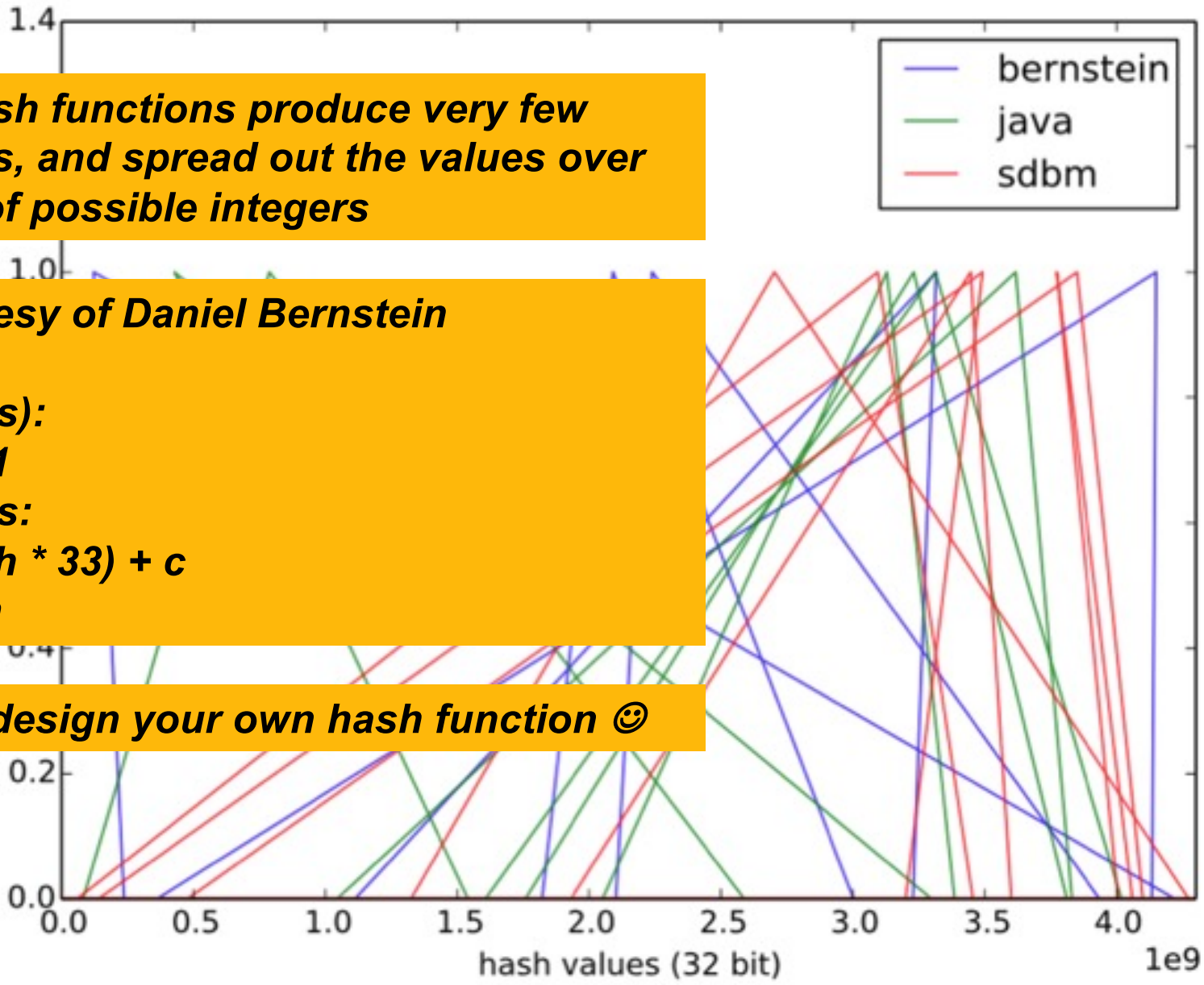
Good Hash Function Performance

Good hash functions produce very few collisions, and spread out the values over billions of possible integers

Courtesy of Daniel Bernstein

```
djbhash(s):
  h = 5381
  for c in s:
      h = (h * 33) + c
  return h
```

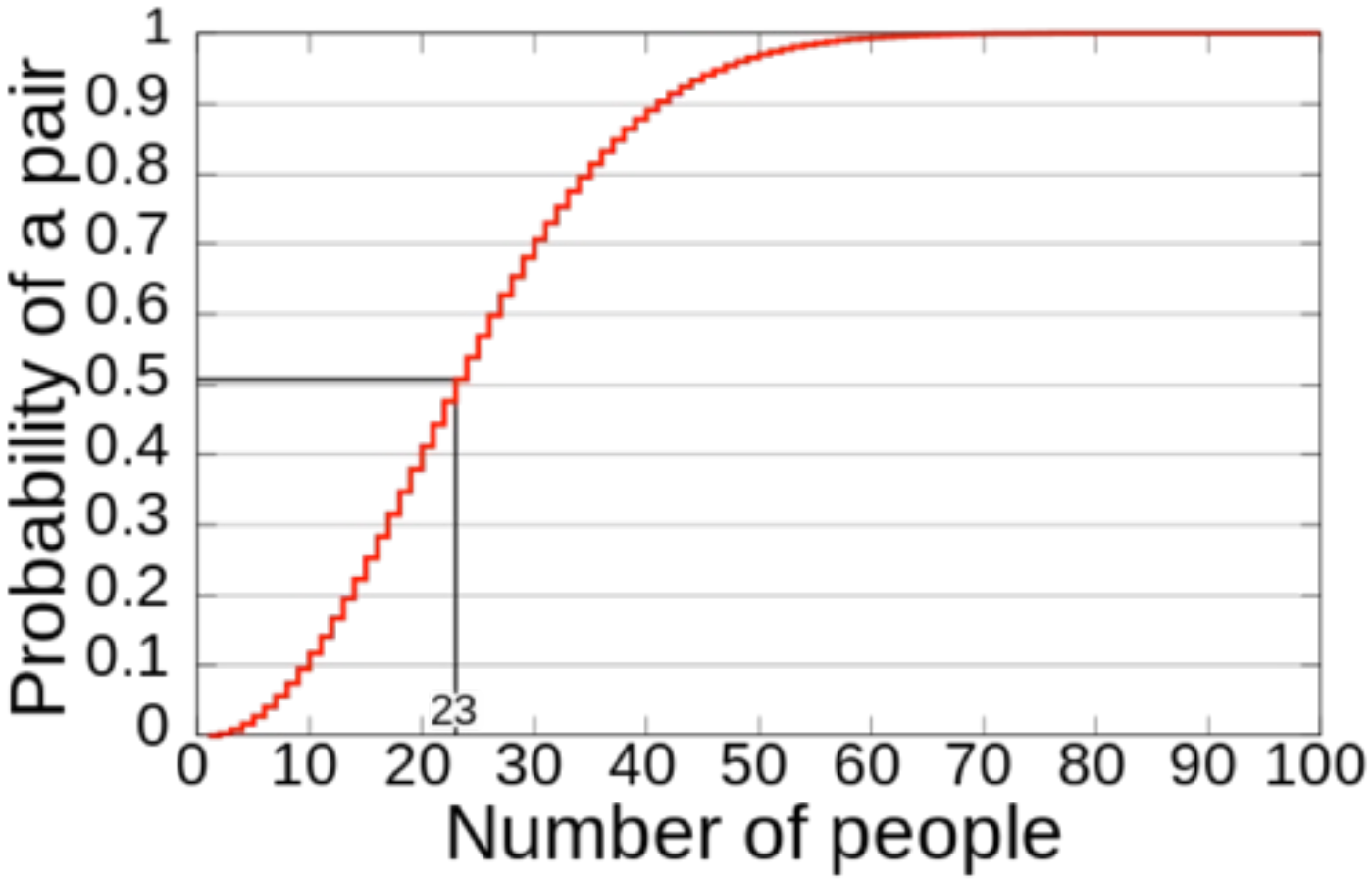
Don't design your own hash function ☺



Hash every word in dictionary.txt and plot #collisions versus hash value

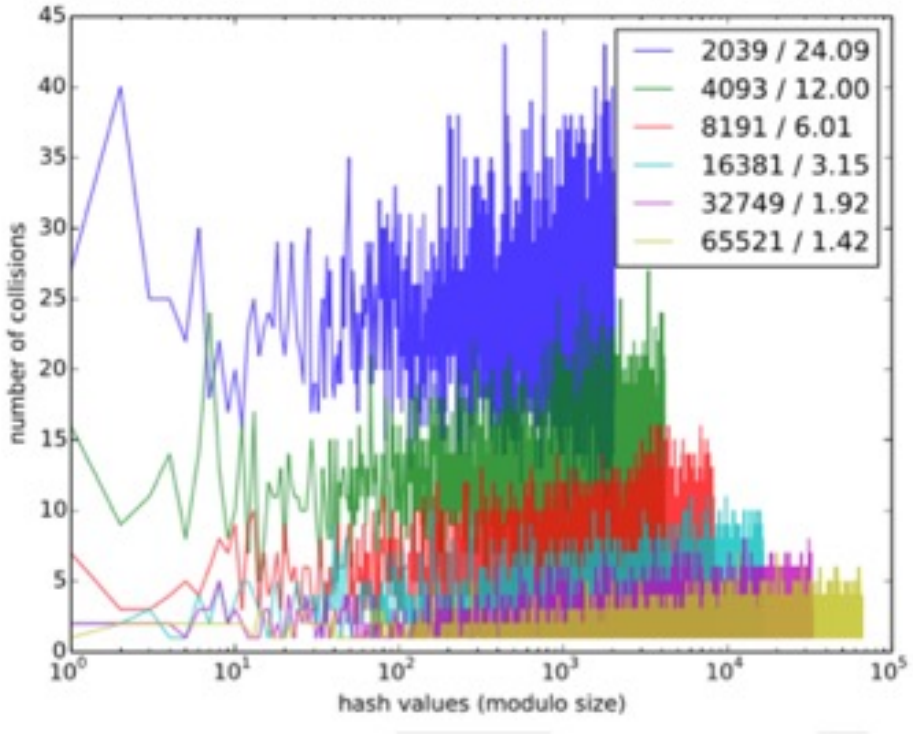
Birthday Paradox

Some collisions are essentially unavoidable at modest load

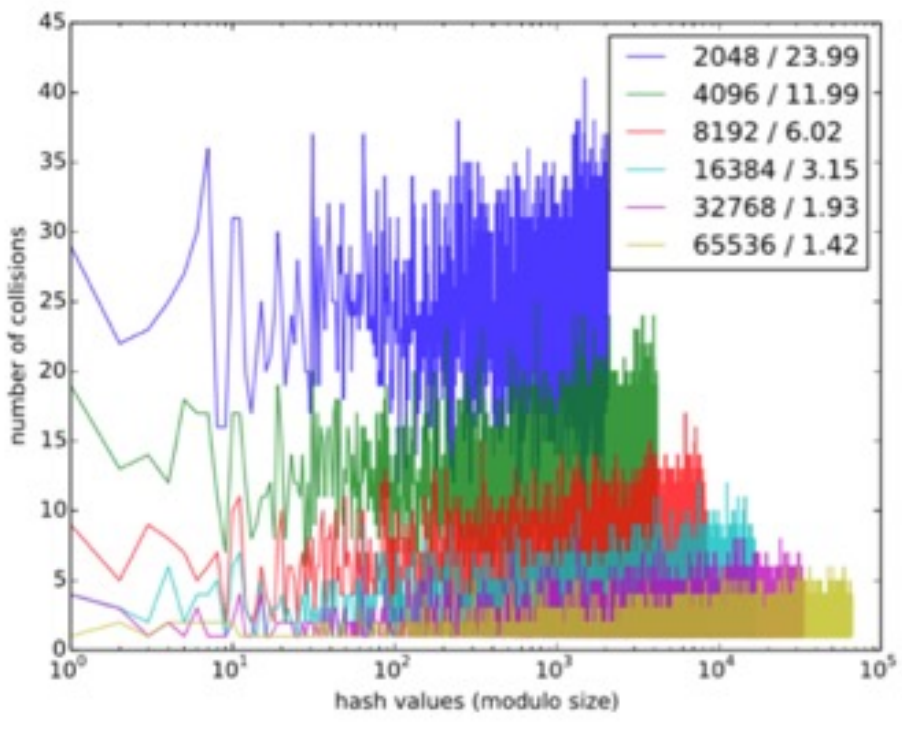


There is a 364/365 chance 2 people don't share the same birthday
There is a 364/365 * 363/365 chance 3 people don't share
There is a 364/365 * 363/365 * 362/365 chance 4 people don't share
…
There is a >90% chance two people in this room share the same birthday

Hash Table Size (Berstein Hash)



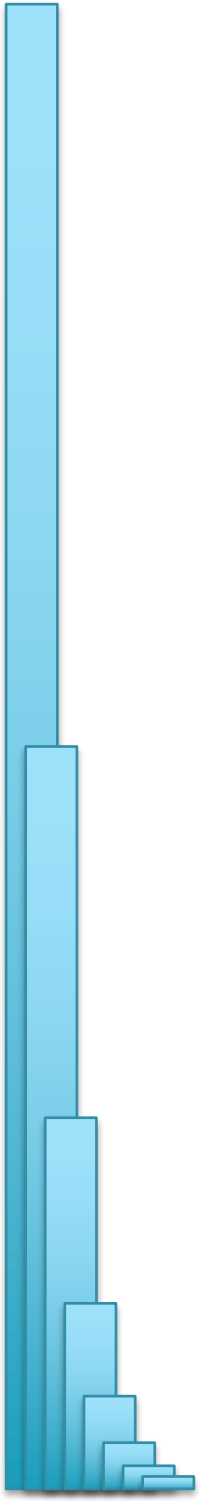
Prime Hash Table Sizes



Powers of Two

In theory, using a prime number as the size of your hash table will have fewer collisions, but in practice using powers of two works about as well.

The major exception is if you are using quadratic probing: with a load factor < 0.5, a prime size will guarantee you explore at least half of the possible slots, but a non-prime size make no guarantees



Part 5: HashMap Implementation

HashMap<K,V> (1)

```
import java.util.ArrayList;
import java.util.Iterator;

// doesn't resize/rehash

public class HashMap<K, V> implements Map<K, V> {

    private static class Entry<K, V> {
        K key;
        V value;

        Entry(K k, V v) {
            this.key = k;
            this.value = v;
        }

        public boolean equals(Object that) {
            return (that instanceof Entry)
                && (this.key.equals(((Entry) that).key));
        }

        public int hashCode() {
            return this.key.hashCode();
        }
    }
```

Uses an ArrayList<ArrayList>> to store the data

Nested Entry<K,V> class

Equals checks the key only

Use java hashCode() function on the key
For Strings:

$$h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$$

HashMap<K,V> (2)

```
 // inner class, not nested! look ma, no static!
private class HashMapIterator implements Iterator<K> {
    private int returned = 0;
    private Iterator<ArrayList<Entry<K, V>>> outer;
    private Iterator<Entry<K, V>> inner;
    HashMapIterator() {
        this.outer = HashMap.this.data.iterator();
        this.inner = this.outer.next().iterator();
    }
    public boolean hasNext() {
        return this.returned < HashMap.this.size;
    }
    public K next() {
        if (this.inner.hasNext()) {
            this.returned += 1;
            return this.inner.next().key;
        } else {
            while (!this.inner.hasNext() && this.outer.hasNext()) {
                this.inner = this.outer.next().iterator();
            }
            if (this.inner.hasNext()) {
                this.returned += 1;
                return this.inner.next().key;
            } else {
                return null;
            }
        }
    }
    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Iterator is a bit complicated because it has to iterate over the outer list buckets and then the inner list of Entry's in each bucket

HashMap<K,V> (3)

```
private static final int INITIAL_SIZE = 8;
private ArrayList<ArrayList<Entry<K, V>>> data;
private Entry<K, V> fake;
private int size;

public HashMap() {
    this.data = new ArrayList<>(); // Java 7
    for (int i = 0; i < INITIAL_SIZE; i++) {
        this.data.add(new ArrayList<Entry<K, V>>());
    }
    this.fake = new Entry<K, V>(null, null);
}

public boolean has(K k) {
    return this.find(k) != null;
}

private Entry<K, V> find(K k) {
    int slot = this.hash(k);
    this.fake.key = k;
    int index = this.data.get(slot).indexOf(this.fake);
    if (index == -1) {
        return null;
    } else {
        return this.data.get(slot).get(index);
    }
}
```

Use 8 buckets, initialized with an empty list in each

fake helps with searching so we can call ArrayList.indexOf() which uses Entry.equals()

HashMap<K,V> (4)

```
private int hash(Object o) {
    return this.abs(o.hashCode()) % this.data.size();
}

private int abs(int i) {
    if (i < 0) {
        return -i;
    } else {
        return i;
    }
}

public void put(K k, V v) throws UnknownKeyException {
    Entry<K, V> e = this.findForSure(k);
    e.value = v;
}


public V get(K k) throws UnknownKeyException {
    Entry<K, V> e = this.findForSure(k);
    return e.value;
}
```

hash() returns
hashCode % num_buckets

findForSure guarantees the
reference is not null

HashMap<K,V> (5)

```
    private Entry<K, V> findForSure(K k) throws UnknownKeyException {
        Entry<K, V> e = this.find(k);
        if (e == null) {
            throw new UnknownKeyException();
        }
        return e;
    }

    public void insert(K k, V v) throws DuplicateKeyException {
        if (this.has(k)) {
            throw new DuplicateKeyException();
        }
        Entry<K, V> e = new Entry<K, V>(k, v);
        int slot = this.hash(k);
        this.data.get(slot).add(e);
        this.size += 1;
    }

    public V remove(K k) throws UnknownKeyException {
        Entry<K, V> e = this.findForSure(k);
        int slot = this.hash(k);
        this.data.get(slot).remove(e);
        this.size -= 1;
        return e.value;
    }

    public Iterator<K> iterator() {
        return new HashMapIterator();
    }
}
```

findForSure guarantees the reference is not null

Just adds the item to the appropriate bucket, but does NOT resize!

Removes the item from the containing bucket using ArrayList.remove

Benchmarks

BinarySearchTreeMap

```
## These are the standard, unbalanced binary search trees.

$ java Words <manifesto.txt >bla
0.527914894
$ java Words <wealth.txt >bla
1.629583552
$ java Words <war.txt >bla
1.865477425
$ java -Xss2048k Words <cracklib.txt >bla
43.321007802
$ java -Xss16384k Words <dictionary.txt >bla
2905.769206485
```

HashMap

```
## First for a bucket array with only eight slots

$ java Words <manifesto.txt >bla
0.612431685
$ java Words <wealth.txt >bla
4.413707022
$ java Words <war.txt >bla
7.679160966
$ java Words <cracklib.txt >bla
16.050761321
$ java Words <dictionary.txt >bla
541.798612936
```

Even a small hash table outperforms a naïve BST

Benchmarks

HashMap

```
## Now for 4096 slots
## That's still a pretty high load factor:
## dictionary.txt has 234937 distinct words so alpha = 57.36

$ java Words <manifesto.txt >bla
0.529792145
$ java Words <wealth.txt >bla
1.485476444
$ java Words <war.txt >bla
1.647926909
$ java Words <cracklib.txt >bla
1.562509081
$ java Words <dictionary.txt >bla
4.221646673
```

AvlTreeMap

```
$ java Words <manifesto.txt >bla
0.64004931
$ java Words <wealth.txt >bla
1.744705285
$ java Words <war.txt >bla
2.062956075
$ java Words <cracklib.txt >bla
2.256693694
$ java Words <dictionary.txt >bla
7.379325237
```

HashMap is noticeably faster than AVLTreeMap even though AVLTreeMap guarantees O(lg n) performance

Benchmarks

HashMap

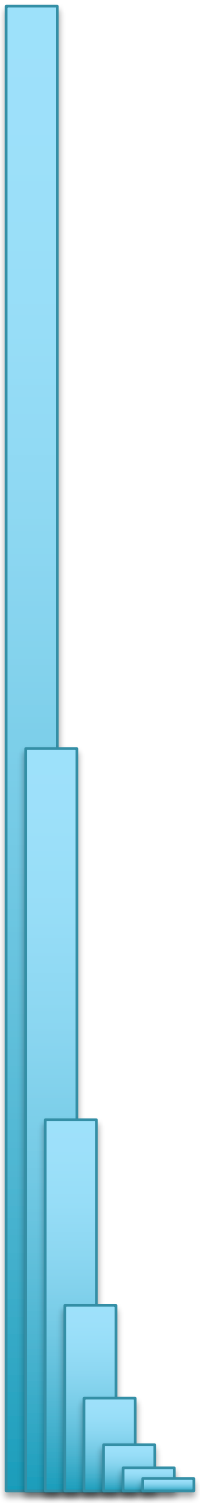
```
## Now for 4096 slots
## That's still a pretty high load factor:
## dictionary.txt has 234937 distinct words so alpha = 57.36

$ java Words <manifesto.txt >bla
0.529792145
$ java Words <wealth.txt >bla
1.485476444
$ java Words <war.txt >bla
1.647926909
$ java Words <cracklib.txt >bla
1.562509081
$ java Words <dictionary.txt >bla
4.221646673
```

TreapMap

```
$ java Words <manifesto.txt >bla
0.543315182
$ java Words <wealth.txt >bla
1.788253795
$ java Words <war.txt >bla
2.196759232
$ java Words <cracklib.txt >bla
1.5308888
$ java Words <dictionary.txt >bla
3.602697315
```

The only thing faster than HashMap is the TreapMap

Random almost always wins ☺



Next Steps

1. Reflect on the magic and power of Hash Tables!

2. Assignment 7 due Friday November 9 @ 10pm