Hashing

CSE 332 Summer 2020

Instructor: Richard Jiang

Teaching Assistants:

Hamsa Shankar	Kristin Li	Winston Jodjana
Maggie Jiang	Hans Zhang	Michael Duan
Jeffery Tian	Annie Mao	

Lecture Q&A: pollev.com/332summer

Lecture clarifications: tinyurl.com/332-07-15A

Announcements

- Make sure you do checkpoint 1 to fix your GitLab pipeline!
- More office hours coming for international time zones
- Keep giving us feedback through office hours, quizzes, or anonymous feedback form

pollev.com/332summer :: tinyurl.com/332-07-15A

Lecture Outline

- **B-Trees Wrapup**

- Balanced Tree Wrapup

- Hash Tables
  - Designing Hash Function
  - Hashing Applications
  - Hash Table Operations
  - Collision *Avoidance* Concepts
  - Collision Resolution: Separate Chaining

pollev.com/332summer :: tinyurl.com/332-07-15A

B+ Tree Add Algorithm (1 of 2)

1. Add the value to its **leaf** in key-sorted order

2. If the **leaf** now has *L*+1 items, *overflow:*
   - Split the **leaf** into two leaves:
     - Original **leaf** with $\lceil (L+1)/2 \rceil$ smaller items
     - New **leaf** with $\lfloor (L+1)/2 \rfloor = \lceil L/2 \rceil$ larger items
   - Attach the new **leaf** to its parent
     - Add a new key (smallest key in new leaf) to parent in sorted order

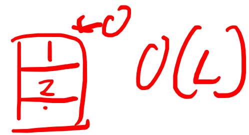
3. If step (2) caused the parent to have *M*+1 children, …

B+ Tree Add Algorithm (2 of 2)

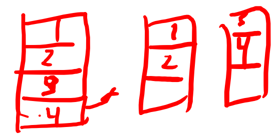
3. If step (2) caused an **internal node** to have *M*+1 children
   - Split the **internal node** into two nodes
     - Original **node** with $\lceil (M+1)/2 \rceil$ smaller keys
     - New **node** with $\lfloor (M+1)/2 \rfloor = \lceil M/2 \rceil$ larger keys
   - Attach the new **internal node** to its parent
     - Add a new key (smallest key in new node) to parent in sorted order
   - If step (3) caused the parent to have *M*+1 children, repeat step (3) on the parent
     - If the **root** overflows, make a new **root** with two children
     - This is the only case that increases the tree height

B+ Tree Add: Efficiency (1 of 2)

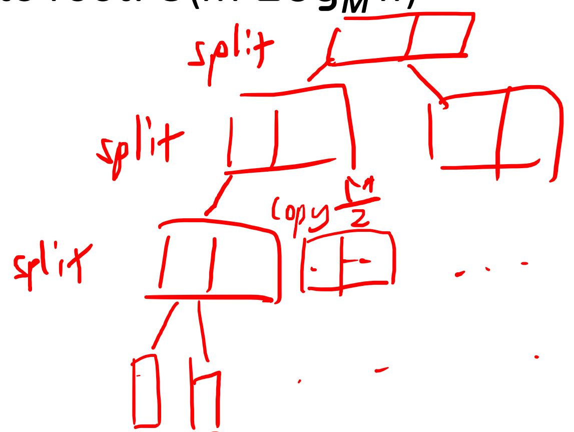
- Find correct leaf: $O(\texttt{log}_2 M \texttt{ log}_M n)$
- Add (key, value) pair to leaf: $O(L)$
  - Why? Shifting leaf elements



- Possibly split leaf: $O(L)$
  - Why? Copying half of elements to new node



- Possibly split parents all the way up to root: $O(M \texttt{ log}_M n)$
  - Why? $O(\frac{M}{2} \cdot \log_M n) \rightarrow O(M \cdot \log_M N)$



- Total: $O(L + M \texttt{ log}_M n)$

B+ Tree Add: Efficiency (2 of 2)

- Worst-case runtime is $O(L + M \log_M n)$!

- But the worst-case isn't that common!
  - Splits are uncommon
    - Only required when a node is *full*
    - M and L are likely to be large and, after a split, nodes will be half empty
  - Splitting the **root** is extremely rare
  - Remember that our goal is minimizing disk accesses! Disk accesses are still bound by $O(\log_M n)$

B+ Tree Remove Algorithm (1 of 2)

1. Remove the item from its **leaf**

2. If the **leaf** now has $\lceil L/2 \rceil - 1$, *underflow:*
   - If a neighbor has > $\lceil L/2 \rceil$ items, *adopt* and update parent
   - Else, *merge* **leaf** with neighbor
     - Guaranteed to have a legal number of items
     - Parent now has one less **leaf**

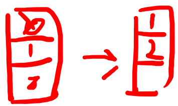
3. If step (2) caused the parent to have $\lceil M/2 \rceil - 1$ children, …

B+ Tree Remove Algorithm (2 of 2)

3. If step (2) caused an **internal node** to have $\lceil M/2 \rceil - 1$ children
   - If a neighbor has > $\lceil M/2 \rceil$ keys, *adopt* and update parent
   - Else, *merge* with neighbor node
     - Guaranteed to have a legal number of keys
     - Parent now has one less node, may need to continue up the tree
   - If step (3) caused the parent to have $\lceil M/2 \rceil - 1$ children, repeat step (3) on the parent
     - If **root** went from 2 children to 1 child, make the child the new **root**
     - This is the only case that decreases the tree height

B+ Tree Remove: Efficiency (1 of 2)

- Find correct **leaf**: $O(\log_2 M \log_M n)$
- Remove item from **leaf**: $O(L)$
  - Why? Shifting leaf elements



- Possibly adopt from or merge with neighbor **leaf**: $O(L)$
  - Why? Shifting to adopt or copying from merge
- Possibly adopt or merge **parent node** up to **root**: $O(M \log_M n)$
  - Why? $\log_M N$ is from height each operation is $O(M)$

except with M instead of L

- Total: $O(L + M \log_M n)$

B+ Tree Remove: Efficiency (2 of 2)

- Worst-case runtime is $O(L + M\ \mathtt{log}_M\ n)$!

- But the worst-case isn't that common!
  - Merges are uncommon
    - Only required when a node is ***half empty*** (🤔 half full?)
    - M and L are likely large and, after a merge, nodes will be completely full
  - Shrinking the height by removing the **root** is extremely rare
  - Remember that our goal is minimizing disk accesses! Disk accesses are still bound by $O(\mathtt{log}_M\ n)$

B+ Trees in Java?

- For most of our data structures, we encourage writing high-level, reusable code. Eg, using Java generics in our projects

- It's a bad idea for B+ Trees, however
  - Java can do balanced trees! It can even do other B-Trees, such as the 2-3 tree (which resembles a B+ Tree with M=3)
  - Java wasn't designed for things like managing disk accesses, which is the whole point of B+ Trees
  - The key issue is Java's extra *levels of indirection*…

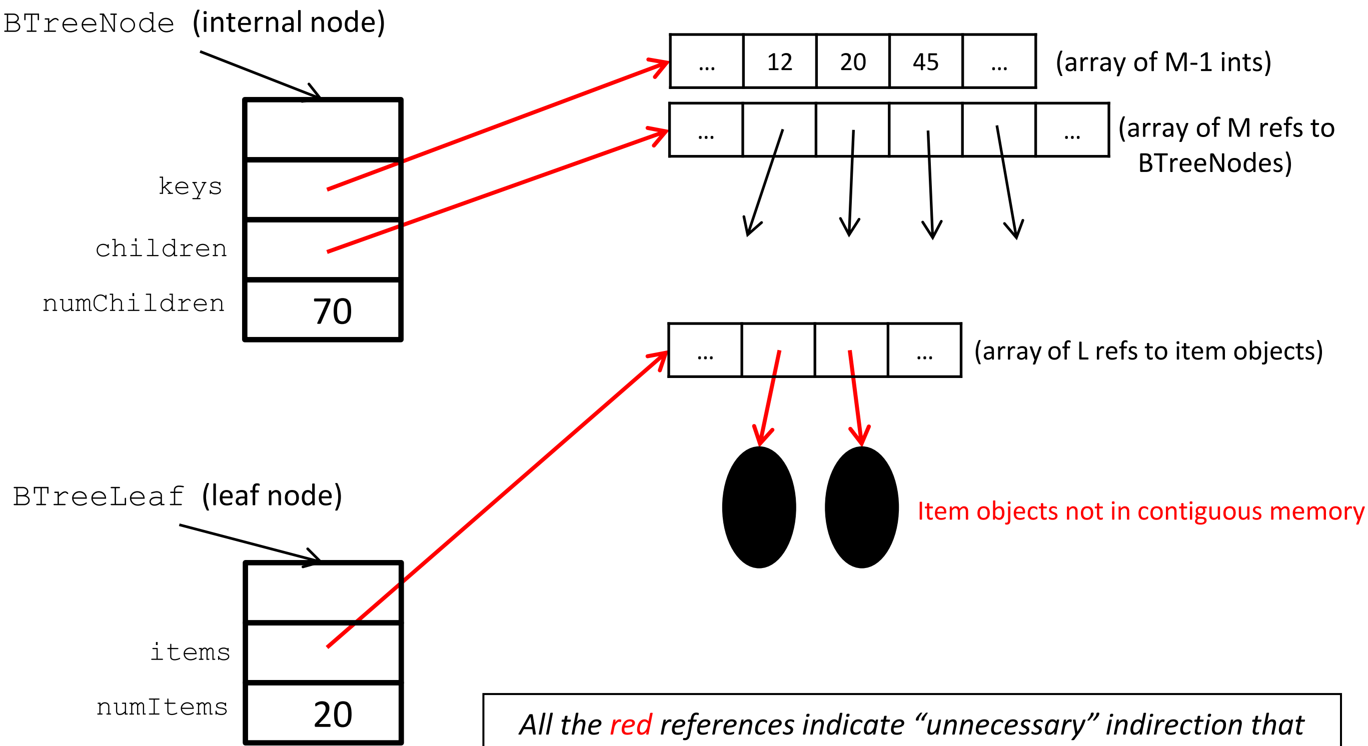
Possible Java Implementation: Code

Even if we assume **`int`** keys, Java's data representation doesn't match what we want out of a B+ Tree

```
class BTreeNode<E> {  // internal node
  static final int M = 128;
  int[]           keys        = new int[M-1];
  BTreeNode<E>[]  children    = new BTreeNode[M];
  int             numChildren = 0;
  …
}

class BTreeLeaf<E> {  // leaf node
  static final int L = 32;
  E[] items      = (E[])new Object[L];
  int numItems = 0;
  …
}
```

Possible Java Implementation: Box-and-Arrows



All the red references indicate "unnecessary" indirection that might be avoided in another programming language!

B+ Trees in Java: The Moral of the Story

- The whole idea behind B+ trees was to keep related data in contiguous memory

- But this runs counter to the code and patterns Java encourages
  - Java's implementation of generic, reusable code is not want you want for your performance-critical web-index

- Other languages (e.g., C++) have better support for "flattening objects into arrays" in a generic, reusable way

- Levels of indirection matter!

Lecture Outline

- B-Trees Wrapup
- **Balanced Tree Wrapup**
- Hash Tables
  - Designing Hash Function
  - Hashing Applications
  - Hash Table Operations
  - Collision *Avoidance* Concepts
  - Collision Resolution: Separate Chaining

pollev.com/332summer :: tinyurl.com/332-07-15A

Summary: Search Trees (1 of 2)

- **Binary Search Trees** make good dictionaries because they implement **`find`**, **`add`**, and **`remove`** as well as a number of useful operations such as **`flattenIntoSortedList`** or **`successor`**
  - Essential and beautiful computer science

- *Balanced* search trees guarantee logarithmic-time operations
  - … if you can maintain balance within the time bound
  - **AVL trees** maintain balance by tracking height and allowing all children to differ in height by at most 1
  - **B trees** maintain balance by keeping nodes at least half full and all leaves at same height

Summary: Search Trees (2 of 2)

- Other great balanced trees (see text; worth knowing they exist)
  - **Red-black trees**: all leaves have depth within a factor of 2
  - **Splay trees**: self-adjusting; amortized guarantee; no extra space for height information

- Next up: dictionaries that don’t rely on trees at all!

Lecture Outline

- B-Trees Wrapup
- Balanced Tree Wrapup
- Hash Tables
  - **Designing Hash Function**
  - Hashing Applications
  - Hash Table Operations
  - Collision *Avoidance* Concepts
  - Collision Resolution: Separate Chaining

pollev.com/332summer :: tinyurl.com/332-07-15A

What is Hashing?

- **Hashing** is taking data of arbitrary size and type and converting it to an fixed-size integer (ie, an integer in a predefined range)

- Running example: design a hash function that maps strings to 32-bit integers [-2147483648, 2147483647]

- A good hash function exhibits the following properties:
  - *Deterministic*: the same input should generate the same output
  - *Efficiency*: it should take a reasonable amount of time
  - *Uniformity*: inputs should be spread “evenly” over its output range

Bad Hashing

```
int hashFn(String s) {
  return
    Random.nextInt();
}
```

```
int hashFn(String s) {
  int retVal = 0;

  for (int i = 0;
    i < s.length();
    i++) {

    for (int j = 0;
      j < s.length();
      j++) {
     retVal += helperFn(
       s, i, j);
    }
  }

  return retVal;
}
```

```
int hashFn(String s) {
  if (s.length()%2 == 0)
    return 17;
  else
    return 42;
}
```

Deterministic?	*Efficient?*	*Uniform?*

Attempt #1: hash("cat")

- One idea: Assign each letter a number, use the first letter of the word
  - a = 1, b = 2, c = 3, …, z = 26
  - hash("cat") == 3

- What's wrong with this approach?
  - Other words start with c
    - hash("chupacabra") == 3
  - Can't hash "=abc123"

Attempt #2: hash("cat")

- Next idea: Add together all the letter codes, add new values for symbols
  - hash("cat") == 99 + 97 + 116 == 312
  - hash("=abc123") == 505
- What's wrong with this approach?
  - Other words with the same letters
    - hash("act") == 97 + 99 + 116 == 312

33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o		
48	0	64	@	80	P	96	`	112	p		

Attempt #3: hash("cat")

- Max possible value for English-only text (including punctuation) is 126
- Another idea: Use 126 as our base to ensure unique values across all possible strings
  - hash("cat") == $99*126^0 + 97*126^1 + 116*126^2 == 232055937$
  - hash("act") == $97*126^0 + 99*126^1 + 116*126^2 == 232056187$
- What's wrong with this approach?
  - Only handles English!

Attempt #4: hash("cat")

- If we switch to another character set we can encode strings such as "¡Hola!"
  - The Unicode "Basic Multilingual Plane" contains 65,472 codepoints

- hash("cat") == $99*65472^0 + 97*65472^1 + 116*65472^2$ == 497,249,953,827

- What's wrong with this approach?
  - Our range was [-2,147,483,648, 2,147,483,647]
    - 497,249,953,827 % 2,147,483,647 == 1,181,231,370 == hash("覑")
  - We could use the modulus operator (%) to "wrap around", but now we've introduced the possibility of collisions 😢
  - The BMP excludes most emoji (💉), characters outside the "Han Unification" (兩 vs 两 vs 両 vs 网), and much, much more

hash("cat"): Lessons Learned

- Writing a hash function is hard!
  - So don't do it ☺

- Common hash algorithms include:
  - MD5
  - SHA-1
  - SHA-256
    - the only one that hasn't been proven to be *cryptographically insecure* (yet)
  - xxHash
  - CityHash
  - SuperFastHash

Aside: Combining hash functions

- A few rules of thumb / tricks:
  - Use all 32 bits (careful, that includes negative numbers)
  - Use different overlapping bits for different parts of the hash
    - This is why a factor of 37^i works better than 256^i
    - When smashing two hashes into one hash, use bitwise-xor
      - bitwise-and produces too many 0 bits
      - bitwise-or produces too many 1 bits
      - Rely on expertise of others; consult books and other resources
- If keys are known ahead of time, choose a *perfect hash*

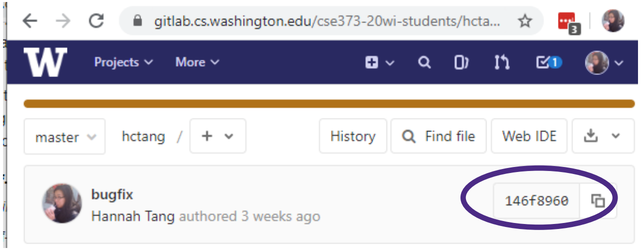
Lecture Outline

- B-Trees Wrapup
- Balanced Tree Wrapup
- Hash Tables
  - Designing Hash Function
  - **Hashing Applications**
  - Hash Table Operations
  - Collision *Avoidance* Concepts
  - Collision Resolution: Separate Chaining

pollev.com/332summer :: tinyurl.com/332-07-15A

Content Hashing: Applications

- Caching:
  - You've downloaded a large video file. You want to know if a new version is available. Rather than re-downloading the entire file, compare your file's hash value with the server's hash value.
- File Verification / Error Checking:
  - Same implementation
  - Can be used to verify files on your machine, files spread across multiple servers, etc.



- Fingerprinting
  - Git hashes ("identification")
  - Ad tracking ("identification"): see https://panopticlick.eff.org/
  - YouTube ContentID ("duplicate detection")

Content Hashing: Defining a Salient Feature

- Hash function implementors can choose what's salient:
  - hash("cat") == hash("CAT") ???

- What's salient in detecting that an image or video is unique?

- What's salient in determining that a user is unique?

Content Hashing vs Cryptographic Hashing

- In addition to the properties of “regular” hash functions, cryptographic hashes must also have the following properties:
  - It is infeasible to find or generate two different inputs that generate the same hash value
  - Given a hash value, it is infeasible to calculate the original input
  - Small changes to the input generate an uncorrelated hash values

- Security is *very hard* to get right!
  - If you don’t know what you’re doing, you’re probably making it worse
  - Most algorithms, including MD5 and SHA-1, are not cryptographically secure

Lecture Outline

- B-Trees Wrapup
- Balanced Tree Wrapup
- Hash Tables
  - Designing Hash Function
  - Hashing Applications
  - **Hash Table Operations**
  - Collision *Avoidance* Concepts
  - Collision Resolution: Separate Chaining

pollev.com/332summer :: tinyurl.com/332-07-15A

Review: Set and Dictionary Data Structures

- We've seen several implementations of the Set or Dictionary ADT
- Search Trees give good performance – log N – as long as the tree is reasonably balanced
  - Which doesn't occur with sorted or mostly-sorted input
  - So we studied two categories of search trees whose heights are bounded:
    - **B-Trees** (eg, B+ Trees) which grow from the root and are "mostly full" M-ary trees
    - **Balanced BSTs** (eg, AVL Trees) which grow from the leaves but rotate to stay balanced

	Find	Add	Remove
LinkedList Dict	Θ(N)	Θ(N)	Θ(N)
BST Dict	h = Θ(N)	h = Θ(N)	h = Θ(N)
AVL Tree Dict	h = Θ(log N)	h = Θ(log N)	h = Θ(log N)
B+ Tree Dict	h = Θ(log N)	h = Θ(log N)	h = Θ(log N)

Hash Table: Idea (1 of 2)

- Thanks to hashing, we can convert objects to large integers
- Hash tables can use these integers as array indices

```
HashTable h;
h.add("cat", 100);
h.add("bee", 50);
h.add("dog", 200);
```

```
hashFunction("cat") == 2;
hashFunction("bee") == 2525393088;
hashFunction("dog") == 9752423;
```

0	-
1	-
2	100
3	-
…	-
9752423	200
…	-
2525393088	50
…	

Hash Table: Idea (2 of 2)

- We can convert objects to large integers
- Hash Tables use these integers as array indices
  - To force our numbers to fit into a reasonably-sized array, we'll use the modulo operator (%)

```
HashTable h;
h.add("cat", 100);
h.add("bee", 50);
h.add("dog", 200);
```

```
hashFunction("cat") == 2;
2 % 5 == 2
hashFunction("bee") == 2525393088;
2525393088 % 5 == 3
hashFunction("dog") == 9752423;
9752423 % 5 == 3
```

0	-
1	-
2	100
3	50
4	-

Poll Everywhere

pollev.com/332summer

How should we handle the "bee" and "dog" collision at index 3?

A. Somehow force "bee" and "dog" to share the same index
B. Overwrite "bee" with "dog"
C. Keep "bee" and ignore "dog"
D. Put "dog" in a different index, and somehow remember/find it later
E. Rebuild the hash table with a different size and/or hash function
F. I'm not sure …

0	-
1	-
2	100
3	50
4	-

Hash Table Components

```
HashTable h;
h.add("cat", 100);
h.add("bee", 50);
```

0	-
1	-
2	100
3	50
4	-

- Implementing a hash table requires the following components:

K —hashFunction→ int —%→ table-index —collision?→ resolved table-index

```
hashFunction("cat") == 2;
2 % 5 == 2
hashFunction("bee") == 2525393088;
2525393088 % 5 == 3
```

A Note on Terminology

- We and the book use the terms
  - "chaining" or "separate chaining"
  - "open addressing"

- Very confusingly
  - "open hashing" is a synonym for "chaining"
  - "closed hashing" is a synonym for "open addressing"

Reminder: a dictionary maps *keys* to *values*; an *item* or *data* refers to the (key, value) pair

Lecture Outline

- B-Trees Wrapup
- Balanced Tree Wrapup
- Hash Tables
  - Designing Hash Function
  - Hashing Applications
  - Hash Table Operations
  - **Collision *Avoidance* Concepts**
  - Collision Resolution: Separate Chaining

pollev.com/332summer :: tinyurl.com/332-07-15A

Key Space vs Value Space vs Table Size

- There are *m* possible keys
  - *m* typically large, even infinite
- A hash function will map those keys into a large set of integers
- We expect our table to have only *n* items
  - *n* is much less than *m* (often written *n* << *m*)

- Many dictionaries have this property
  - Database: All possible student names vs. students enrolled
  - AI: All possible chess-board configurations vs. those considered by the current player
  - …

Collision Avoidance: Hash Function Input

- As usual: our examples use int or string keys, and omit values

- If you have aggregate/structured objects with multiple fields, you want to hash the "identifying fields" to avoid collisions
  - Hashing just the first name = bad idea
  - Hashing everything = too granular? Too slow?

```
class Person {
  String first; String middle; String last;
  Date birthdate;
}
```

- As we saw earlier, the hard part is deciding *what* to hash
  - The *how* to hash is easy: we can usually use "canned" hash functions

Collision Avoidance: Table Size (1 of 3)

- With "`x % TableSize`", the number of collisions depends on
  - the keys inserted (see previous slide)
  - the quality of our hash function (don't write your own)
  - `TableSize`

- Larger table-size tends to help, but not always!
  - Eg: 70, 24, 56, 43, 10 with `TableSize` = 10 and `TableSize` = 60

- *Technique*: Pick table size to be prime. Why?
  - Real-life data tends to have a pattern
  - "Multiples of 61" are probably less likely than "multiples of 60"
  - Some collision *resolution* strategies do better with prime size

Collision Avoidance: Table Size (2 of 3)

- Examples of why prime table sizes help:
- If **`TableSize`** is 60 and…
  - Lots of keys hash to multiples of 5, we waste 80% of table
  - Lots of keys hash to multiples of 10, we waste 90% of table
  - Lots of keys hash to multiples of 2, we waste 50% of table
- If **`TableSize`** is 61…
  - Collisions can still happen, but multiples of 5 will fill table
  - Collisions can still happen, but multiples of 10 will fill table
  - Collisions can still happen, but multiples of 2 will fill table

Collision Avoidance: Table Size (3 of 3)

- If **`x`** and **`y`** are "co-prime" (means **`gcd(x,y)==1`**), then

  `(a * x) % y == (b * x) % y` iff **`a % y == b % y`**

- Given table size **`y`** and key hashes as multiples of **`x`**, we'll get a decent distribution if **`x`** & **`y`** are co-prime
  - So choose a **`TableSize`** that has no common factors with any "likely pattern" **`x`**
  - And choose a decent hash function

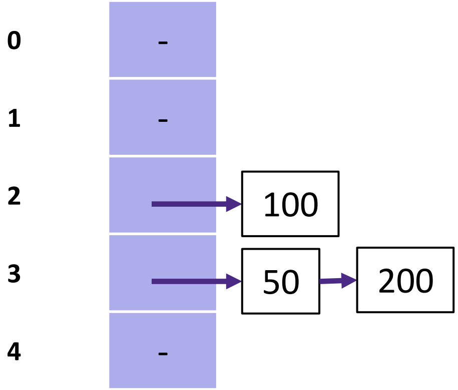
Lecture Outline

- B-Trees Wrapup
- Balanced Tree Wrapup
- Hash Tables
  - Designing Hash Function
  - Hashing Applications
  - Hash Table Operations
  - Collision *Avoidance* Concepts
  - **Collision Resolution: Separate Chaining**

pollev.com/332summer :: tinyurl.com/332-07-15A

Separate Chaining Idea

- All keys that map to the same table location are kept in a list
  - (a.k.a. a "chain" or "bucket")

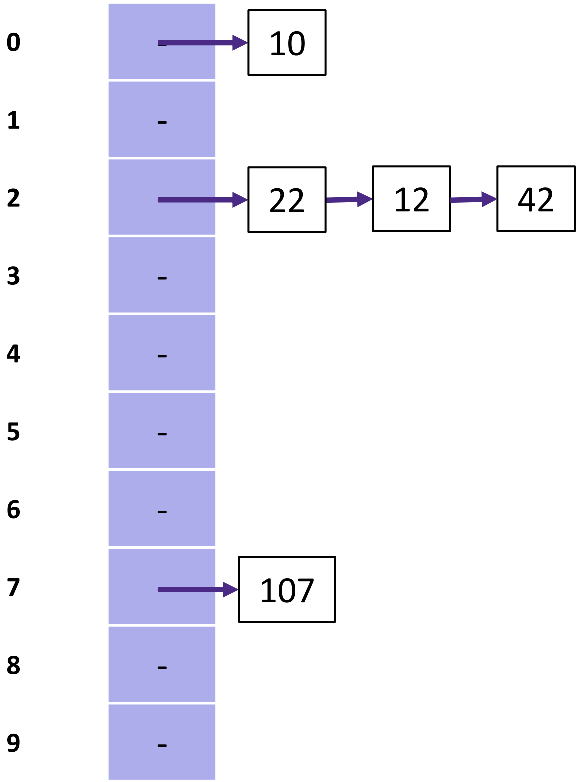



```
HashTable h;
h.add("cat", 100);
h.add("bee", 50);
h.add("dog", 200);
```

```
hashFunction("cat") == 2;
2 % 5 == 2
hashFunction("bee") == 2525393088;
2525393088 % 5 == 3
hashFunction("dog") == 9752423;
9752423 % 5 == 3
```

Separate Chaining: Add Example

- Add 10, 22, 107, 12, 42
  - Let hashFunction(x) = x
  - Let `TableSize` = 10

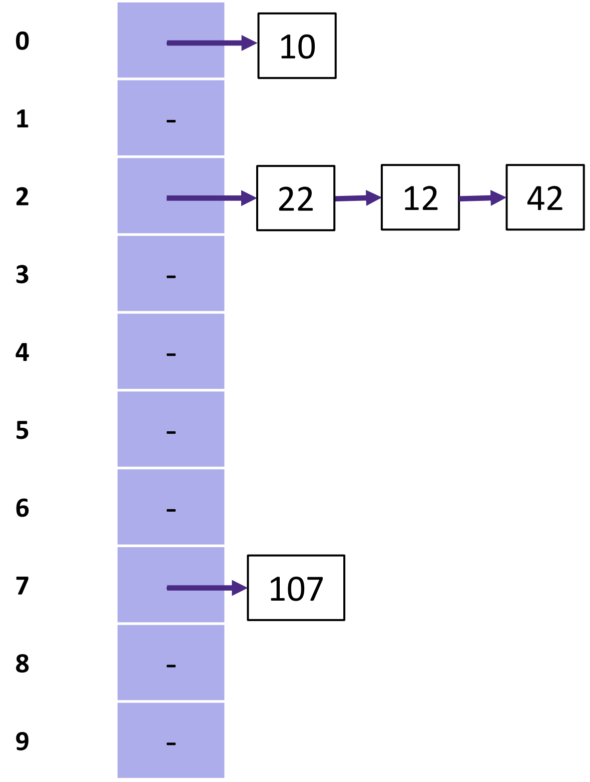


Separate Chaining: Find

- Simple – It's the first part of add!

Separate Chaining: Remove

- Not too bad!
  - Find in table
  - Delete from bucket
- Example: remove 12
- What are the runtimes of these operations (add, find, remove)?



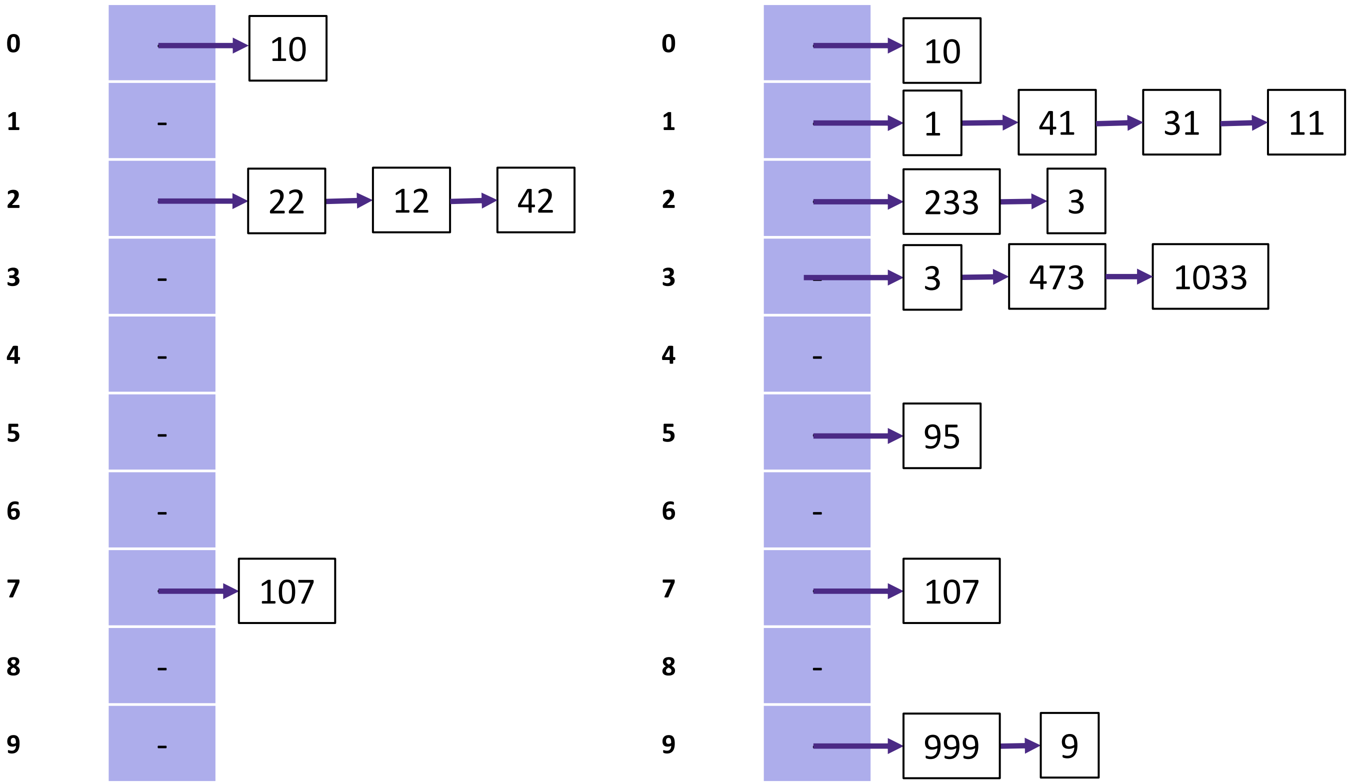
Separate Chaining Runtime: Load Factor

- The **load factor** λ, of a hash table is

$$\lambda = \frac{\text{N}}{\text{TableSize}}$$

$\leftarrow$ number of elements

Load Factor: Example



Separate Chaining Runtime: Cases

- Under separate chaining:
  - The average number of elements per bucket is:
  - If we have some *random* inserts are followed by *random* finds, then:
    - How many keys does each **unsuccessful** `find` compare against?
    - How many keys does each **successful** `find` compare against?
  - If we have a sequence of *worst-case* adds, then:
    - What is the runtime of the next `add`?
    - What is the runtime of `find`?
    - What is the runtime of the next `remove`?
- How big should TableSize be??

Separate Chaining Optimizations

- Worst-case asymptotic runtime
  - Only happens with really bad luck or bad hash function
  - Generally not worth avoiding (e.g., with balanced trees in each bucket)
    - Keep # of items in each bucket small
    - Overhead of AVL tree, etc. not worth it for small n

- Some simple modifications can improve constant factors
  - Linked list vs. array vs. a hybrid of the two
  - Move-to-front (part of Project 2)
  - Leave room for 1 element (or 2?) in the table itself, to optimize constant factors for the common case
    - A time-space trade-off…

A Time vs. Space Optimization

(only makes a difference in constant factors)