B-Trees (cont); Hashing

CSE 332 Spring 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Annie Mao	Howard Xiao	Lucy Jiang
Chris Choi	Jade Watkins	Nathan Lipiarski
Ethan Knutson	Khushi Chaudhari	Richard Jiang

Announcements

- Quiz 2: Sorry about question #5 – it's now extra credit
- Due dates:
  - P2 checkpoint due next week Tuesday!
  - Exs 6-7 due next week Monday!
- Administrative:
  - Course evals released today, reply by next Friday to help us improve this course
  - 1:1 appointments available! Schedule yours today!
  - P1 grades released today; regrade requests will not be entertained until Monday

Learning Objectives

- Be able to implement B+ Tree find()/contains(), add(), and remove() operations
- Understand *how* and *why* a B+ Tree is designed, using our understanding of the memory hierarchy
- Describe how hashing differs from hash tables

B-Tree Reminder: "Just Another Dictionary"

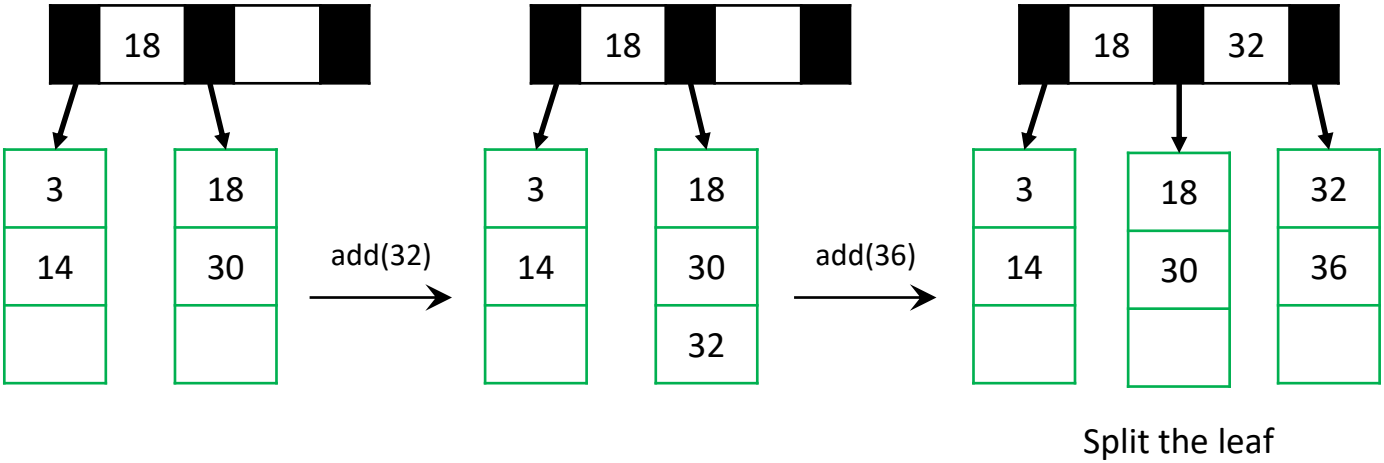
- Keep in mind how we got here:
  - Large data sets won't fit entirely in memory
  - Disk access is slooooooooooooooooowwwwwwwww
  - Design a search tree so we do one disk access per node
  - Then, our goal becomes: keep this search tree as shallow as possible
    - … which minimizes disk accesses
    - Balanced binary search tree is a good start, but we can improve $\log_2 n$ height
    - In an M-ary tree, height drops to $\log_M n$
      - Why not set M really really high? h=1 tree requires multiple disk accesses per node

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

Lecture Outline

- B-Trees
  - **Review: B+ Tree Add**
  - B+ Tree Remove
  - Wrapup
- Balanced Tree Wrapup
- Hashing
  - Designing Our Own Hash Function
  - Hashing Applications

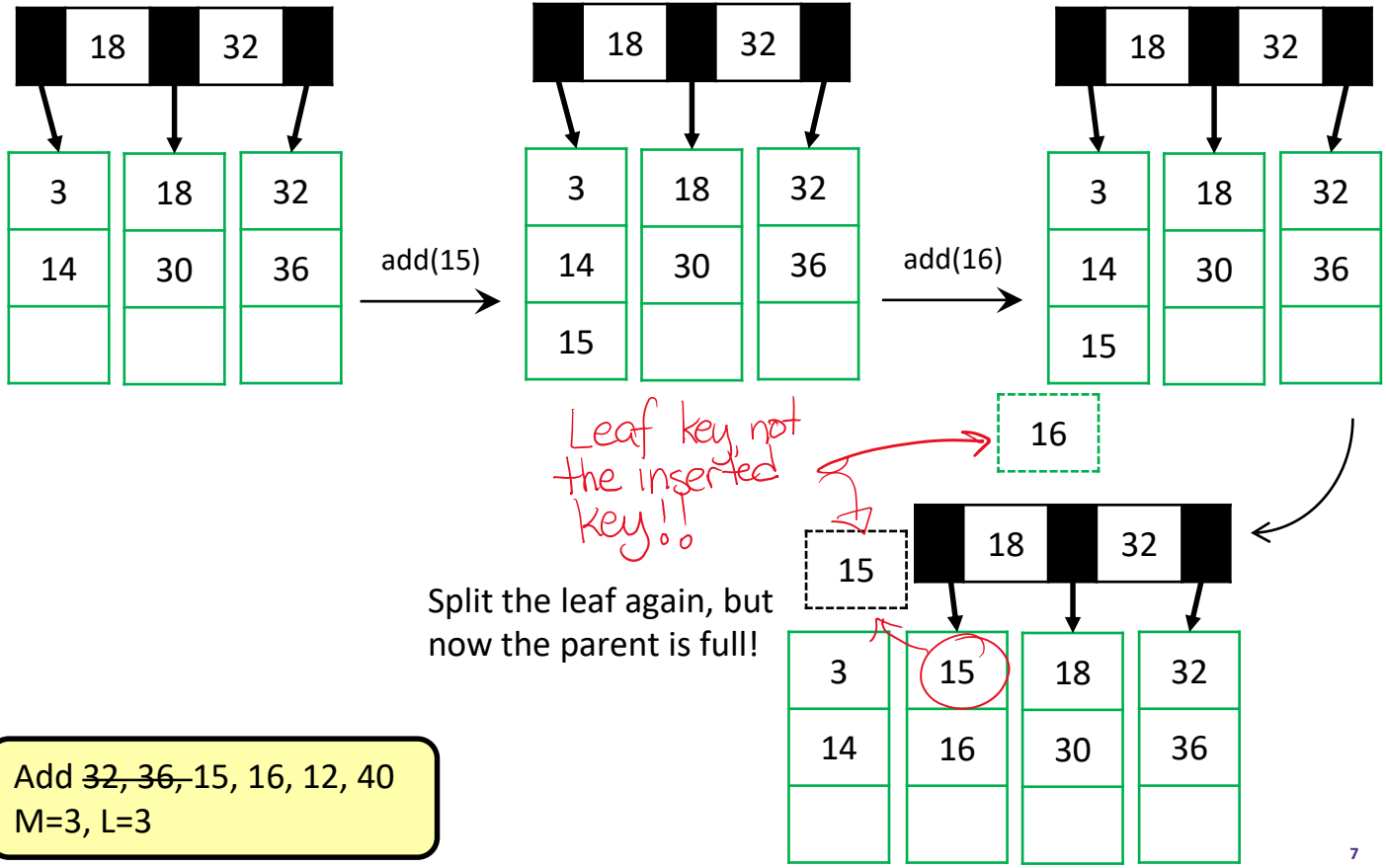
Add Example (1 of 4)



Split the leaf

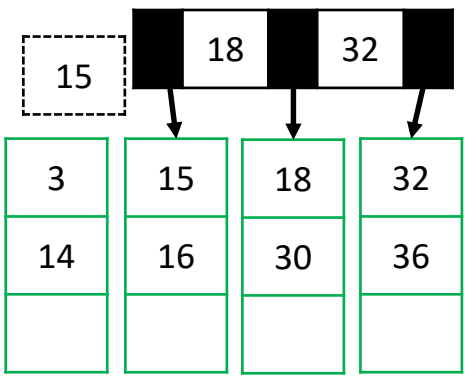
Add 32, 36, 15, 16, 12, 40
M=3, L=3

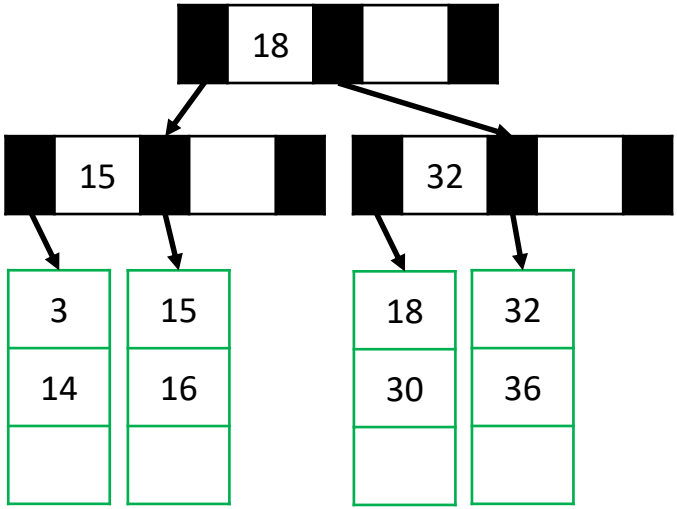
Add Example (2 of 4)



Add ~~32, 36,~~ 15, 16, 12, 40
M=3, L=3

Add Example (3 of 4)

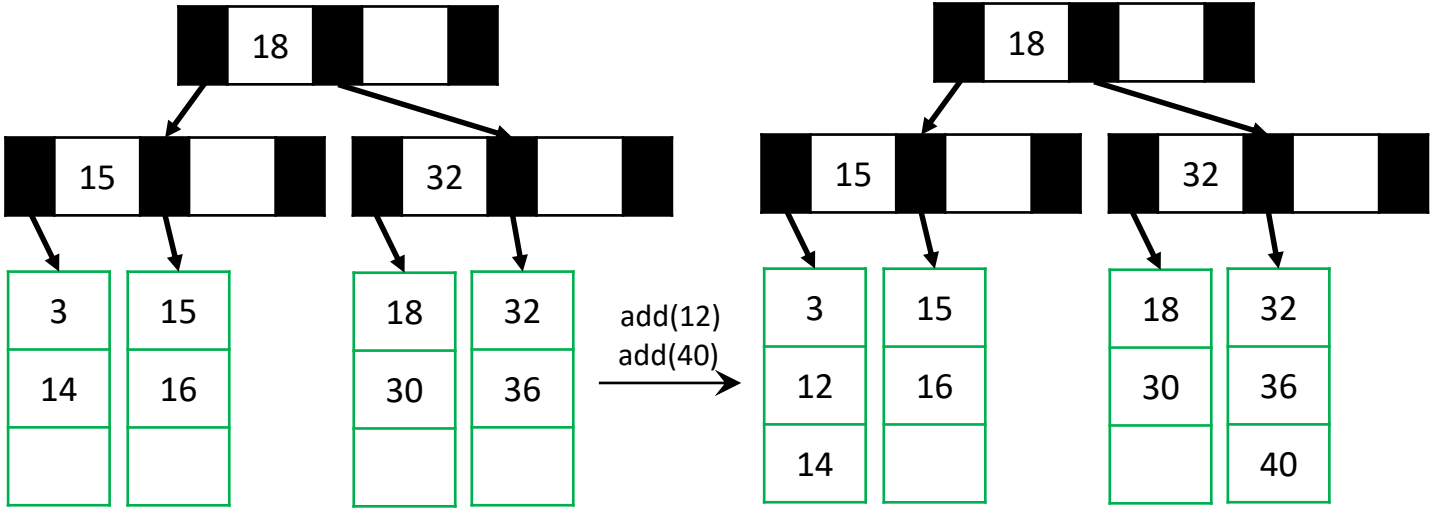




Split the parent (in this case, the root)

Add ~~32, 36, 15,~~ 16, 12, 40
M=3, L=3

Add Example (4 of 4)



Add ~~32, 36, 15, 16,~~ 12, 40
M=3, L=3

B+ Tree Add Algorithm (1 of 2)

1. Add the item 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 $\lceil (L+1)/2 \rceil$ key-smaller items
     - New **leaf** with $\lfloor (L+1)/2 \rfloor = \lceil L/2 \rceil$ key-larger items
   - Attach the new **leaf** to its parent
     - Add a new key (smallest key in new leaf) to parent in key-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 $\lceil (M+1)/2 \rceil$ key-smaller keys
     - New **node** with $\lfloor (M+1)/2 \rfloor = \lceil M/2 \rceil$ key-larger keys
   - Attach the new **internal node** to its parent
     - Add a new key (smallest key in new node) to parent in key-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)

per node binary search

- Find correct **leaf**: $O(\log_2 M \log_M n)$ — num nodes in path
- Add item to **leaf**: $O(L)$
  - Why? Shift items in leaf
- Possibly split **leaf**: $O(L)$
  - Why? $L/2$ copies
- Possibly split **parent nodes** up to **root**: $O(M \log_M n)$
  - Why? $M/2$ copies × $\log_M n$ parents
- Total: $O(L + M \log_M n)$

B+ Tree Add: Efficiency (2 of 2)

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

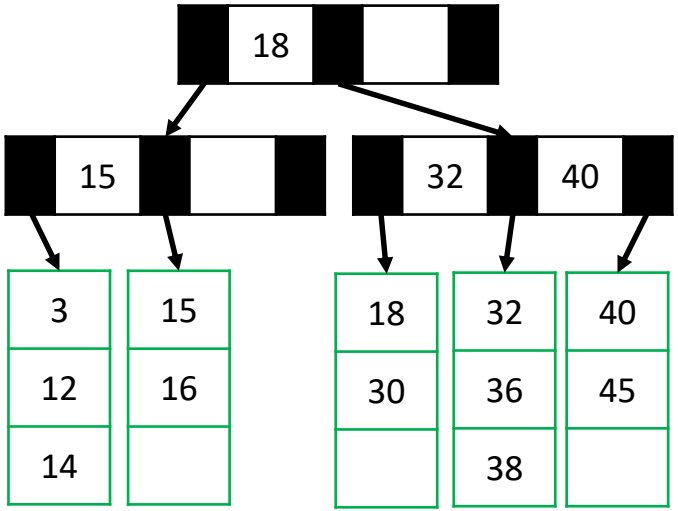
- But the worst-case isn't that common!
  - Splits are uncommon
    - Only required when a node is *full*
    - M and L are likely large and, after a split, nodes will be half empty
  - Splitting the **root** is extremely rare
  - Remember that our goal is minimizing disk accesses! Disk accesses are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

Lecture Outline

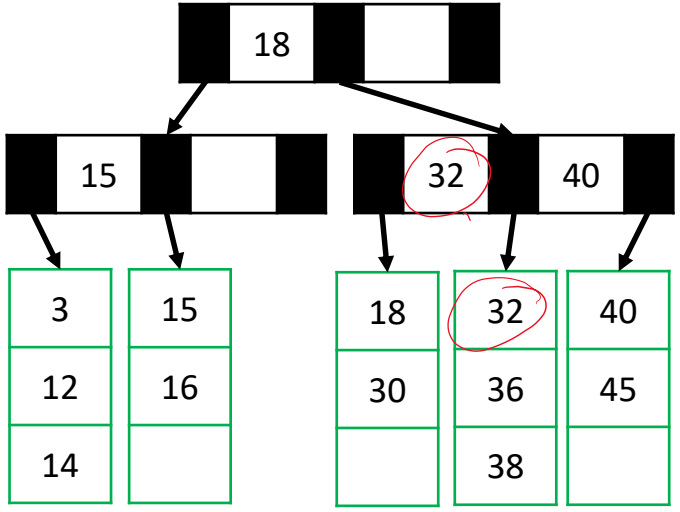
- B-Trees
  - Review: B+ Tree Add
  - **B+ Tree Remove**
  - Wrapup
- Balanced Tree Wrapup
- Hashing
  - Designing Our Own Hash Function
  - Hashing Applications

Remove Example:

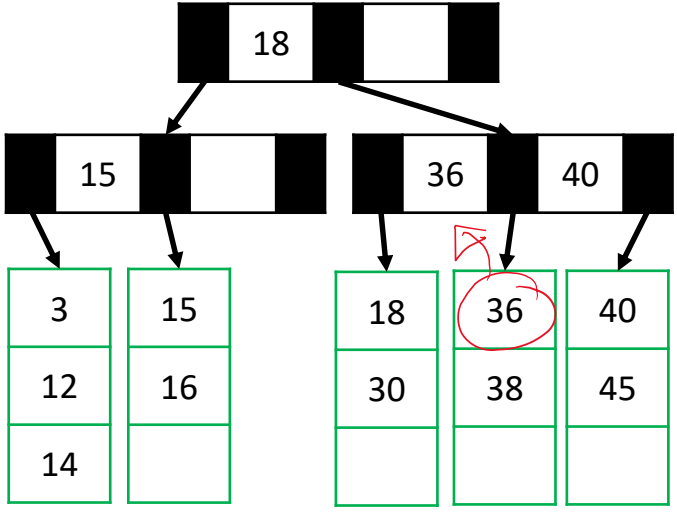
- Remove 32, 15, 16, 14, 18
- M=3, L=3
  - Min #children = 2
  - Min #items = 2



Remove Example: Answer (1 of 8)

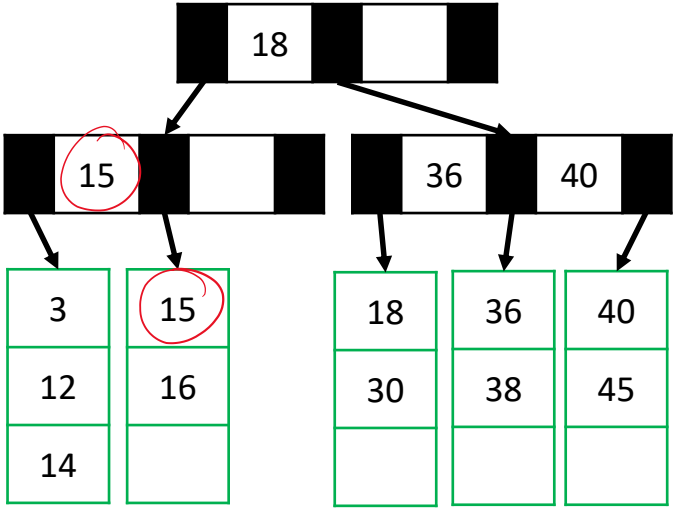


remove(32)



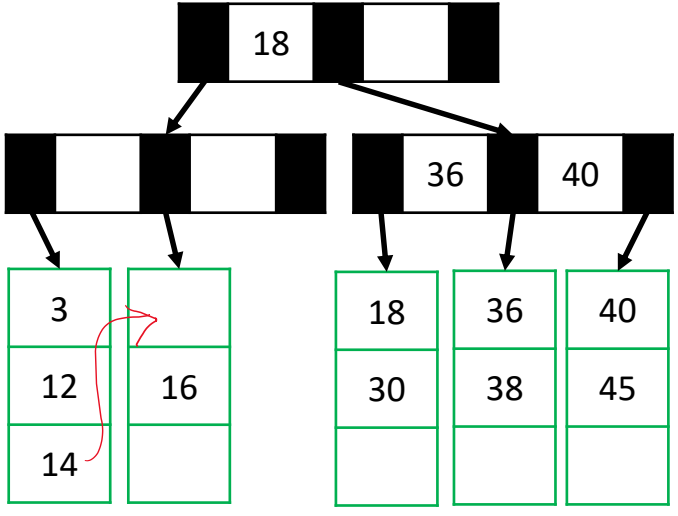
Remove 32, 15, 16, 14, 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (2 of 8)



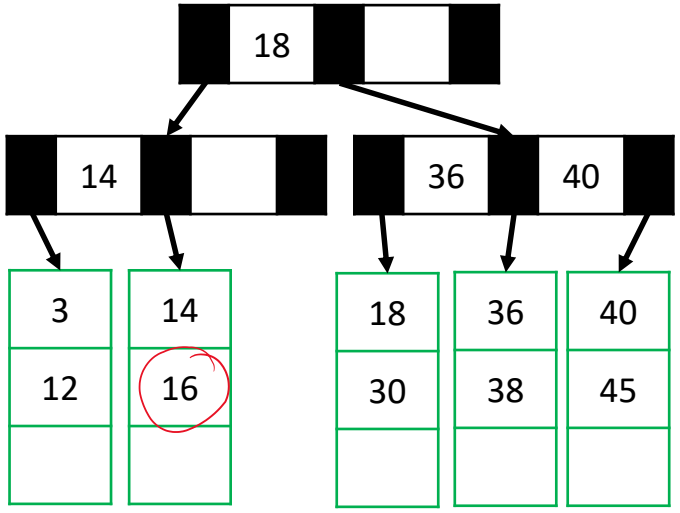
remove(15)

Adopt an item from a neighbor **leaf**



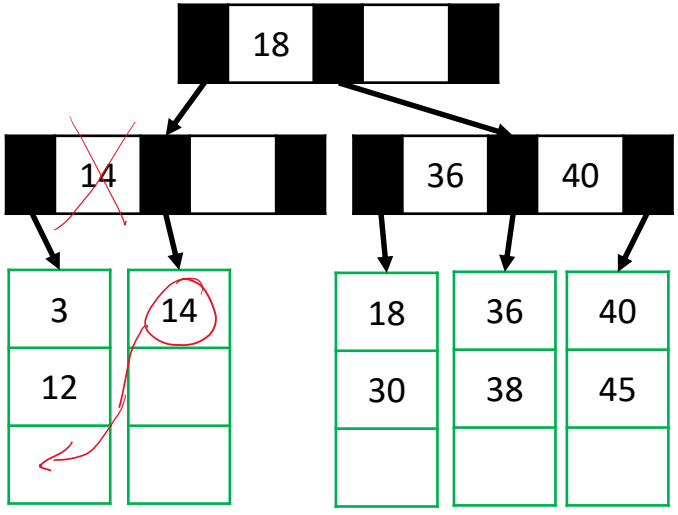
Remove ~~32,~~ 15, 16, 14, 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (3 of 8)



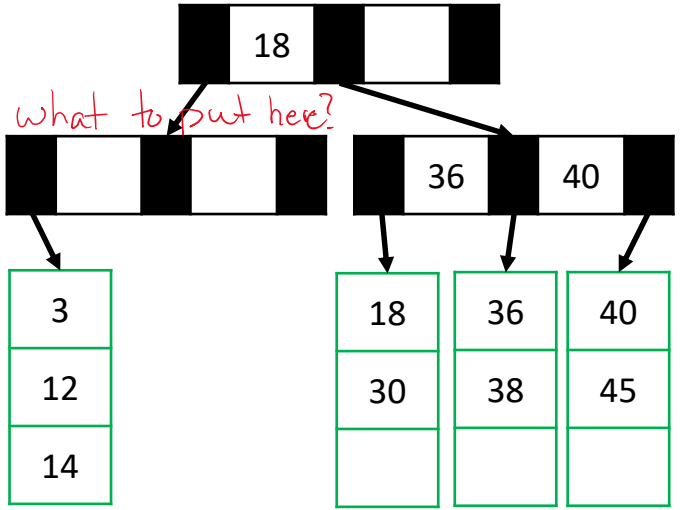
remove(16)

Merge with a neighbor **leaf**

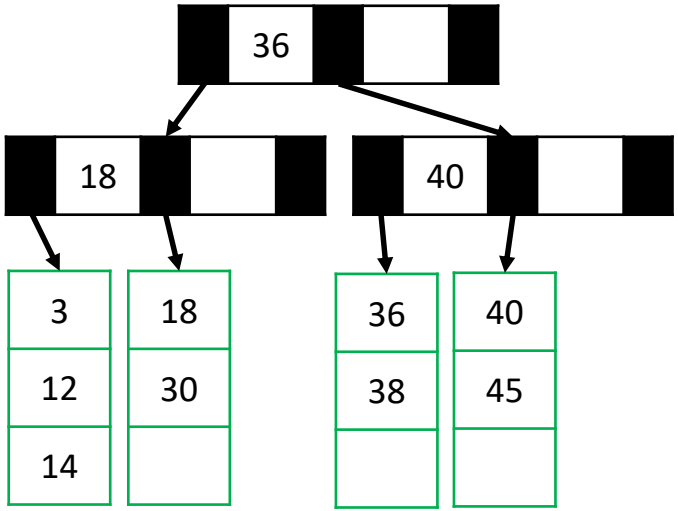


Remove ~~32, 15,~~ 16, 14, 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (4 of 8)

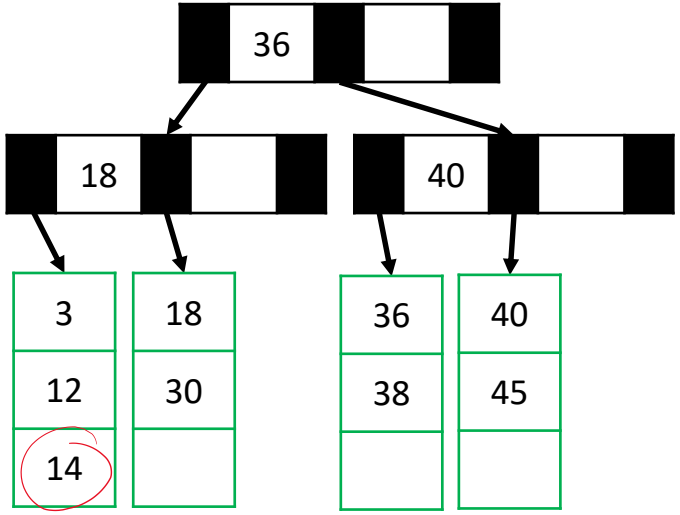


Adopt from a neighbor **node**

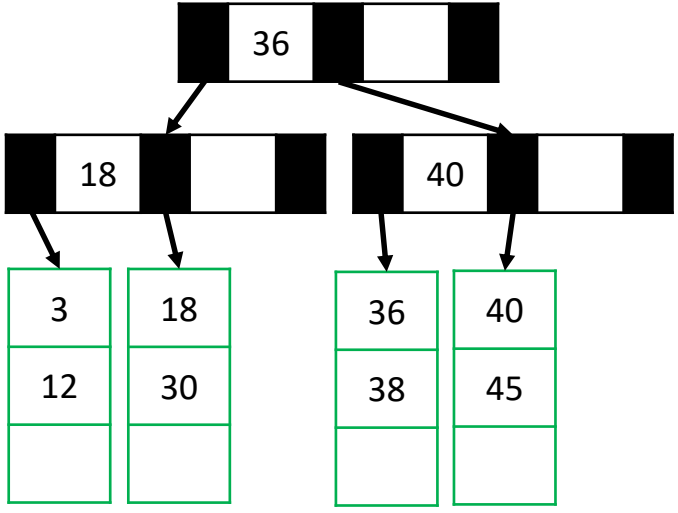


Remove ~~32, 15,~~ 16, 14, 18
M=3, L=~~3~~; min children=2, min items=2

Remove Example: Answer (5 of 8)

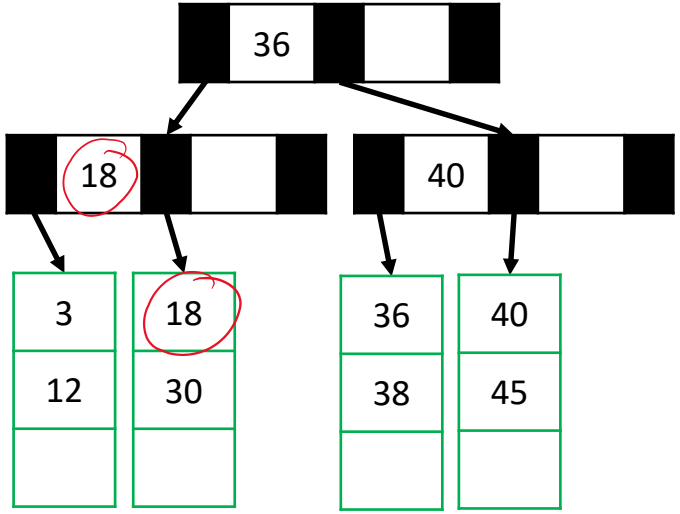


remove(14)



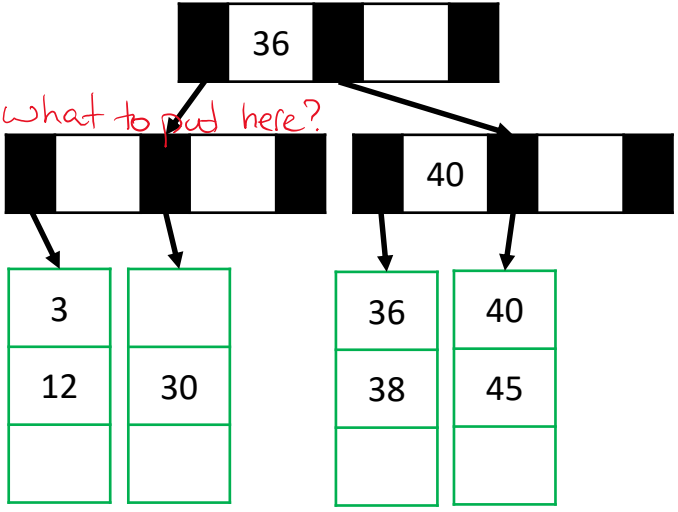
Remove ~~32, 15, 16,~~ 14, 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (6 of 8)



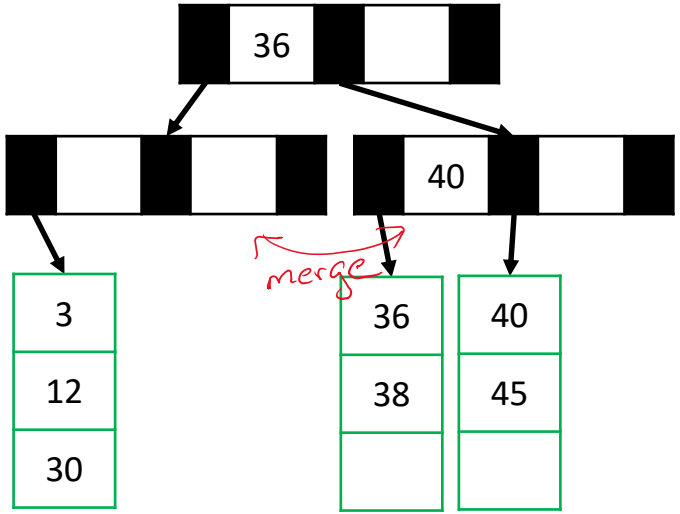
remove(18)

Merge with a neighbor **leaf**

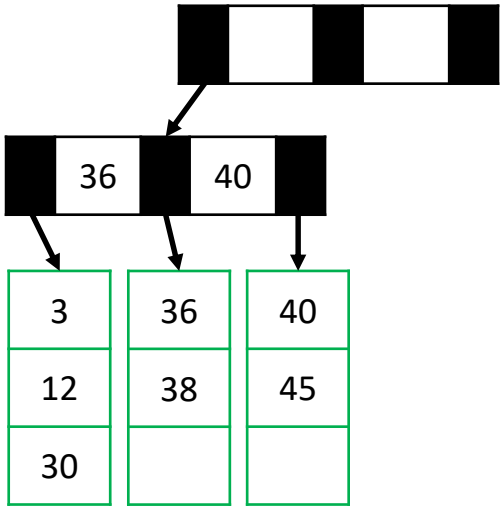


Remove ~~32, 15, 16, 14,~~ 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (7 of 8)

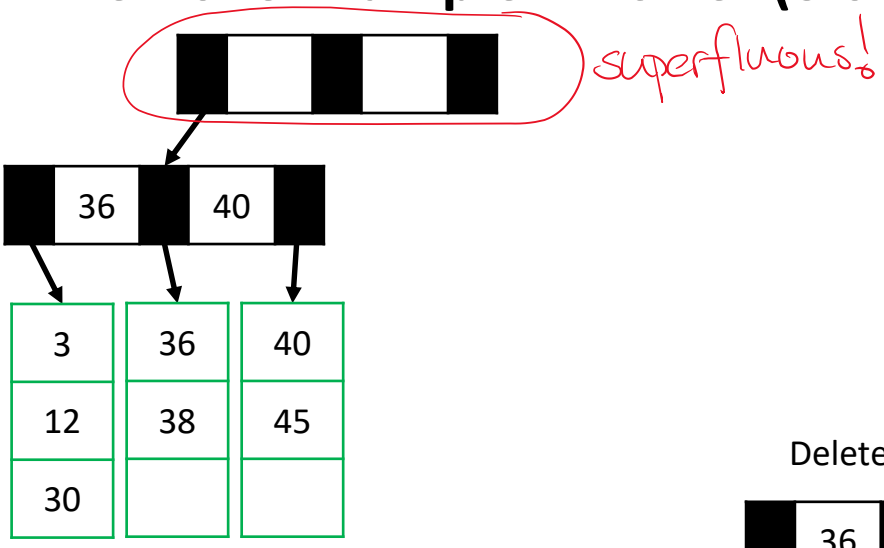


Merge with a neighbor **node**

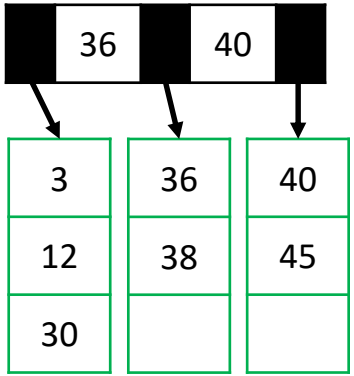


Remove ~~32, 15, 16, 14,~~ 18
M=3, L=3; min children=2, min items=2

Remove Example: Answer (8 of 8)



Delete the **root**



Remove ~~32, 15, 16, 14,~~ 18
M=3, L=3; min children=2, min items=2

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)

per node binary search

number of nodes on path to leaf (ie, height)

- Find correct **leaf**: $O(\log_2 M \log_M n)$
- Remove item from **leaf**: $O(L)$
  - Why? Shift items into correct sorted order
- Possibly adopt from or merge with neighbor **leaf**: $O(L)$
  - Why? $L/2$ copies
- Possibly adopt or merge **parent node** up to **root**: $O(M \log_M n)$
  - Why? $M/2$ copies per node $\times$ $\log_M n$ nodes on path to root (ie, tree height)
- Total: $O(L + M \log_M n)$

B+ Tree Remove: Efficiency (2 of 2)

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

- 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 $O(\log_M n)$

Lecture Outline

- B-Trees
  - Review: B+ Tree Add
  - B+ Tree Remove
  - **Wrapup**
- Balanced Tree Wrapup
- Hashing
  - Designing Our Own Hash Function
  - Hashing Applications

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*…

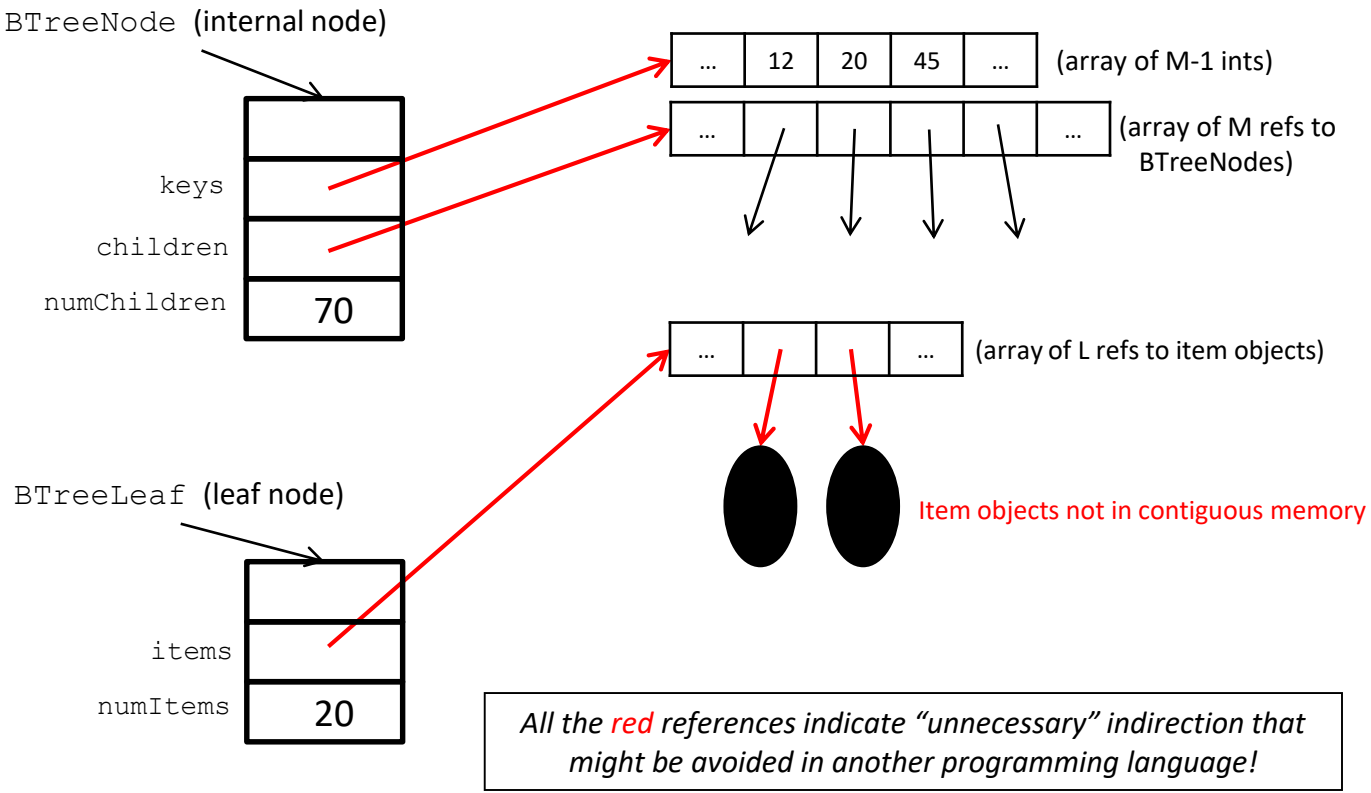
Possible Java Implementation: Code

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
  - Review: B+ Tree Add
  - B+ Tree Remove
  - Wrapup
- **Balanced Tree Wrapup**
- Hashing
  - Designing Our Own Hash Function
  - Hashing Applications

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
  - Review: B+ Tree Add
  - B+ Tree Remove
  - Wrapup
- Balanced Tree Wrapup
- Hashing
  - **Designing Our Own Hash Function**
  - Hashing Applications

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();
}
```

Deterministic?

```
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;
}
```

Efficient?

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

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("覞")
  - 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*

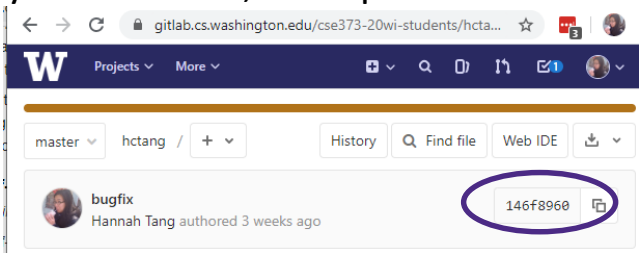
Lecture Outline

- B-Trees
  - Review: B+ Tree Add
  - B+ Tree Remove
  - Wrapup
- Balanced Tree Wrapup
- Hashing
  - Designing Our Own Hash Function
  - **Hashing Applications**

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

Poll Everywhere

pollev.com/cse332

- Can hashing be appropriately used for this application?
  - Verifying files or messages are untampered ("integrity")
  - Verifying the identity of the other party ("authentication")
  - Verifying that an entered password matches a previous password *without storing the password itself*

A. Yes / Yes / No
B. Yes / Yes / Yes
C. Yes / No / No
D. Yes / No / Yes
E. I'm not sure …