



CSE 332 Data Abstractions: B Trees and Hash Tables Make a Complete Breakfast

Kate Deibel
Summer 2012

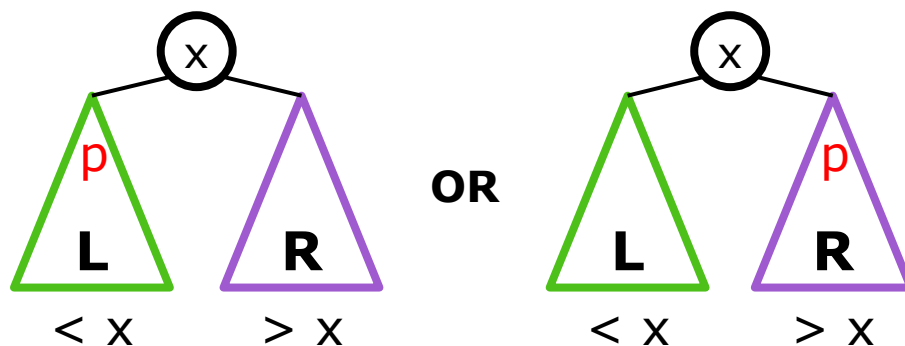
Project 2

- Big project... get started early
 - This project gives you a lot of experience implementing data structures specialized to a problem
- You can work with a partner
 - Please contact us soon with who you will be working with
 - Use the message board to find a partner
- Questions related to project are good fodder for quiz sections... so ask!

Clarifying Splay Insert

insert(x):

- Find x in splay tree
- Splays it or its parent **p** to root
- If x is in tree, stop (no duplicates)
- Else, split tree based on root **p**
 - If $r < x$, then r goes in left subtree
 - If $r > x$, then r goes in right subtree



- Join subtrees using x as root

Technically, they are called B+ trees but their name was lowered due to concerns of grade inflation

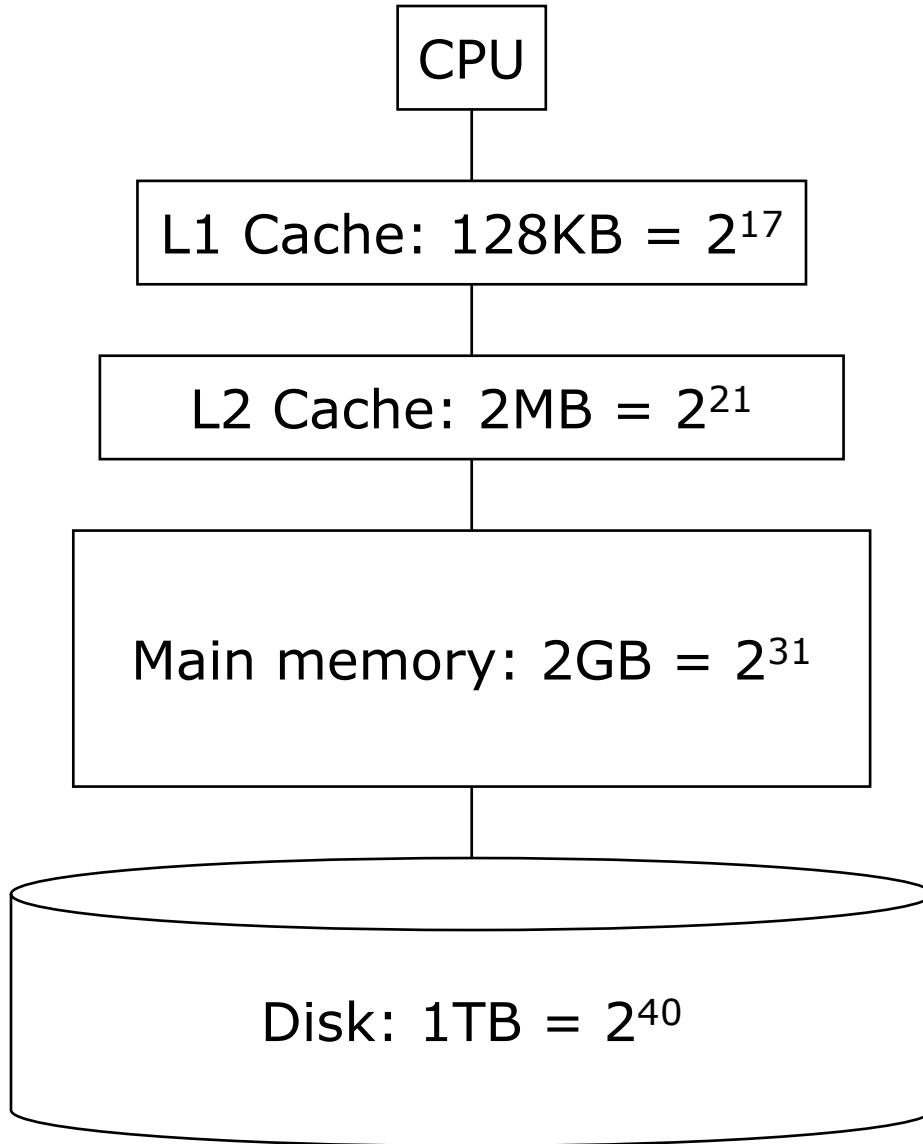
B TREES

Reality Bites

Despite our best efforts, AVL trees and splay trees can perform poorly on very large inputs

Why? It's the fault of hardware!

A Typical Memory Hierarchy



instructions (e.g., addition): $2^{30}/\text{sec}$

get data in L1: $2^{29}/\text{sec} = 2$ insns

get data in L2: $2^{25}/\text{sec} = 30$ insns

get data in main memory:
 $2^{22}/\text{sec} = 250$ insns

get data from "new place" on disk:
 $2^7/\text{sec} = 8,000,000$ insns

"streamed":
 $2^{18}/\text{sec} = 4096$ insns

Moral of The Story

It is much faster to do:

5 million arithmetic ops

2500 L2 cache accesses

400 main memory accesses

Than:

1 disk access

1 disk access

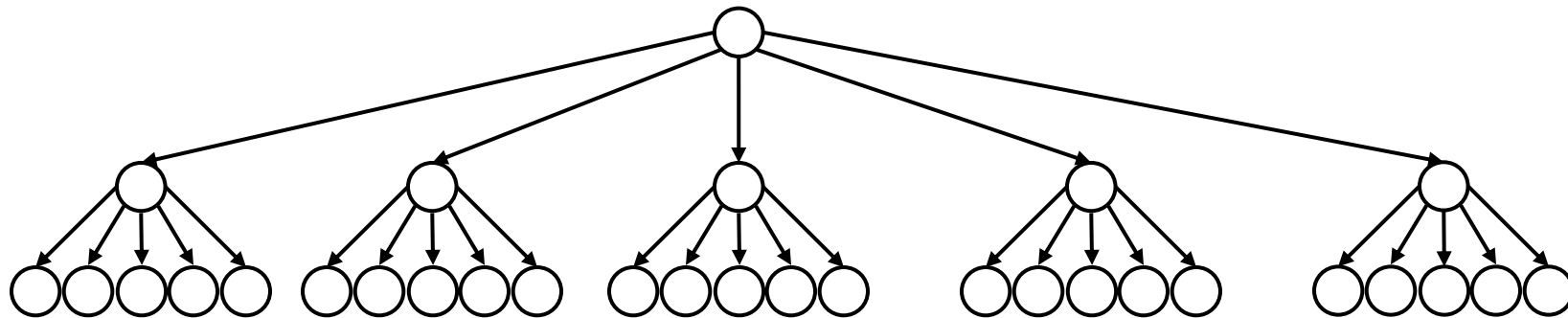
1 disk access

**Accessing the disk is
EXPENSIVE!!!**

M-ary Search Tree

Build a search tree with branching factor M:

- Have an array of sorted children (Node[])
- Choose M to fit snugly into a disk block (1 access for array)



Perfect tree of height h has $(M^{h+1}-1)/(M-1)$ nodes

hops for find: Use $\log_M n$ to calculate

- If $M=256$, that's an 8x improvement
- If $n = 2^{40}$, only 5 levels instead of 40 (5 disk accesses)

Runtime of find if balanced: $O(\log_2 M \log_M n)$

Problems with M-ary Search Trees

- What should the order property be?
- How would you rebalance (ideally without more disk accesses)?
- Any "useful" data at the internal nodes takes up disk-block space without being used by finds moving past it
- Use the branching-factor idea, but for a different kind of balanced tree
 - Not a binary search tree
 - But still logarithmic height for any $M > 2$

B+ Trees (will just say "B Trees")

Two types of nodes:

- Internal nodes and leaf nodes

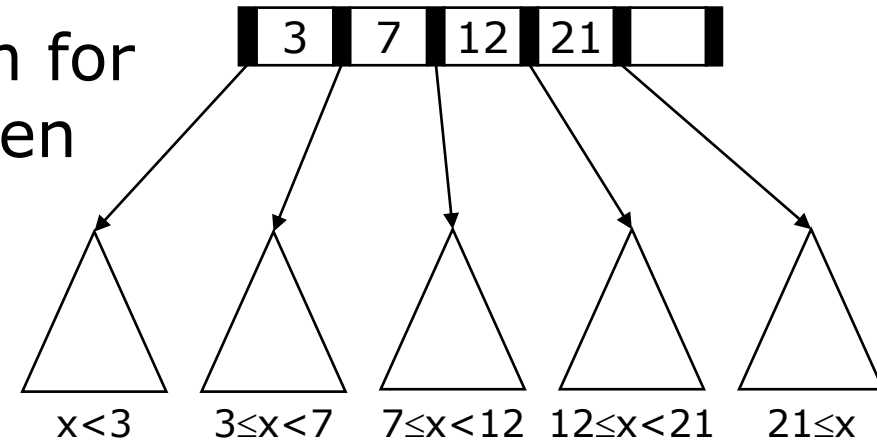
Each internal node has room for up to $M-1$ keys and M children

- All data are at the leaves!

Order property:

- Subtree between x and y
Data that is $\geq x$ and $< y$
- Notice the \geq

Leaf has up to L sorted *data* items



As usual, we will focus only on the keys in our examples

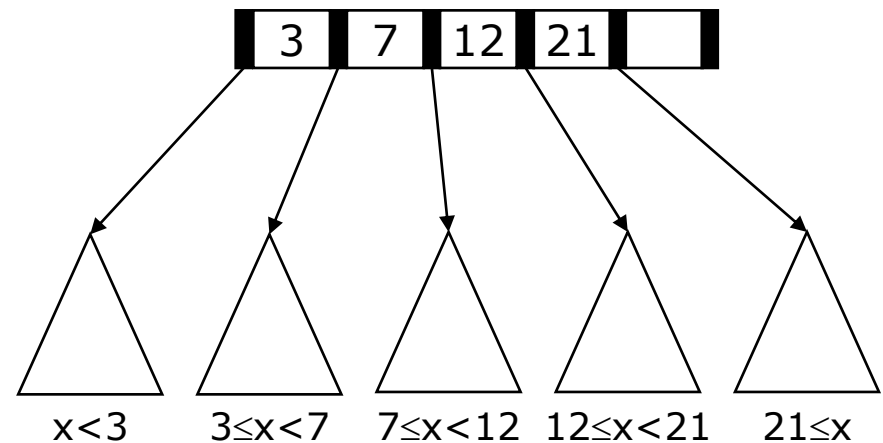
B Tree Find

We are used to data at internal nodes

But find is still an easy root-to-leaf algorithm

- At an internal node, binary search on the $M-1$ keys
- At the leaf do binary search on the $\leq L$ data items

To ensure logarithmic running time, we need to guarantee balance!



What should the balance condition be?

Structure Properties

Root (special case)

- If tree has $\leq L$ items, root is a leaf (occurs when starting up, otherwise very unusual)
- Otherwise, root has between 2 and M children

Internal Node

- Has between $\lceil M/2 \rceil$ and M children (at least half full)

Leaf Node

- All leaves at the same depth
- Has between $\lceil L/2 \rceil$ and L items (at least half full)

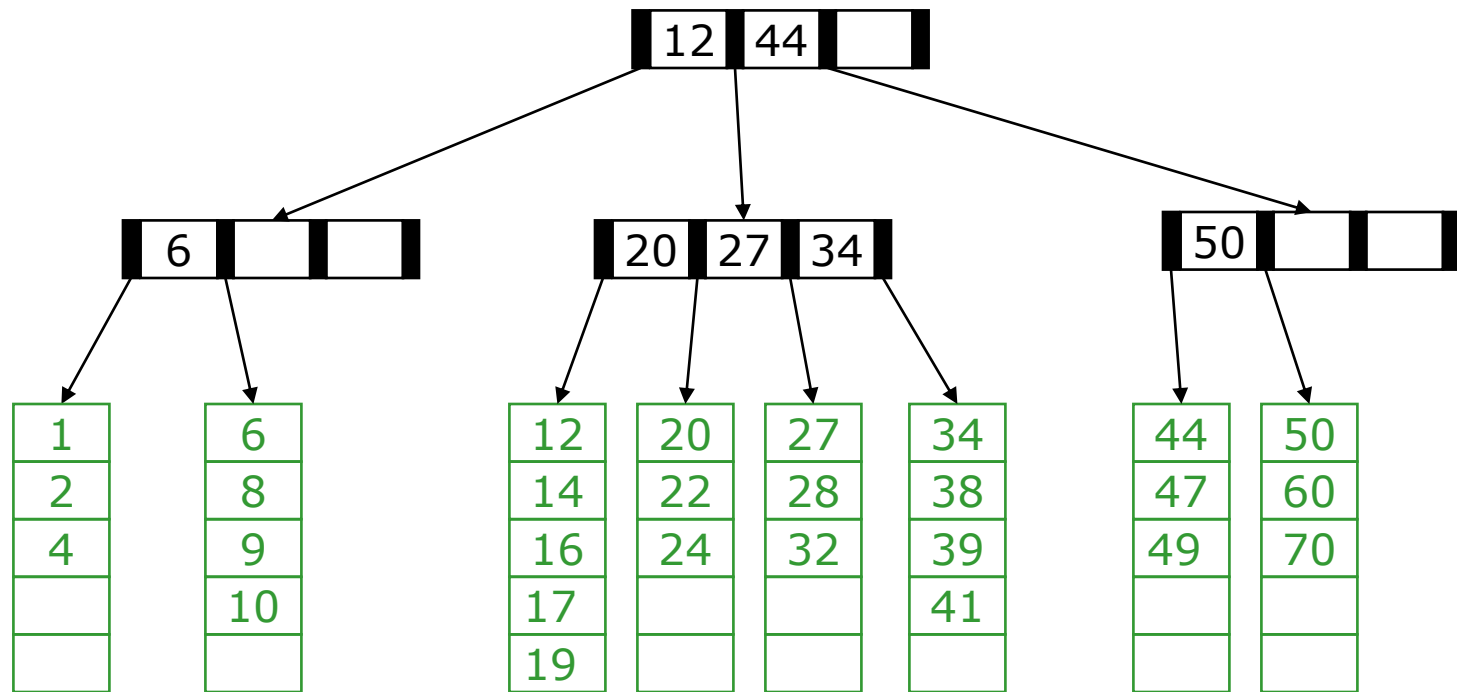
Any $M > 2$ and L will work

- Picked based on disk-block size

Example

Suppose: $M=4$ (max # children in internal node)
 $L=5$ (max # data items at leaf)

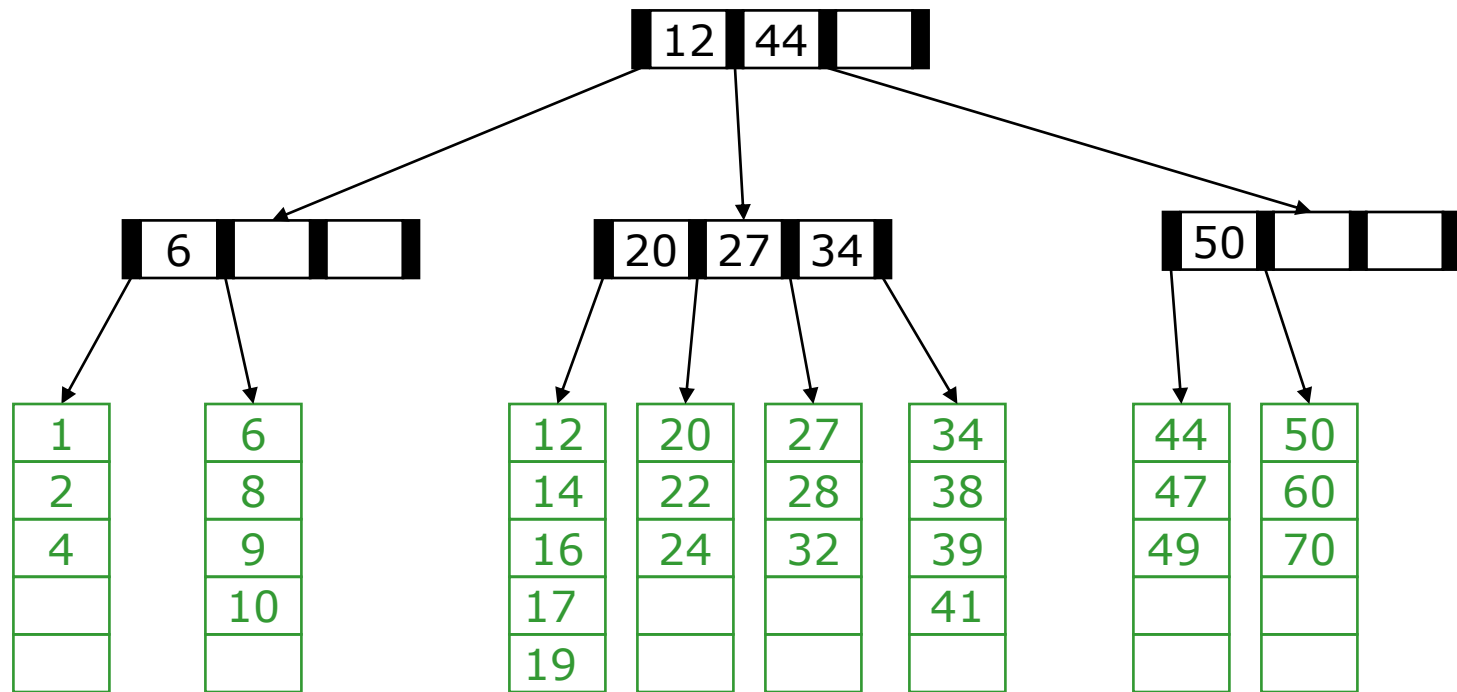
- All internal nodes have at least 2 children
- All leaves at same depth with at least 3 data items



Example

Note on notation:

- Inner nodes drawn horizontally
- Leaves drawn vertically to distinguish
- Includes all empty cells



Balanced enough

Not hard to show height h is logarithmic in number of data items n

Let $M > 2$ (if $M = 2$, then a list tree is legal \rightarrow BAD!)

Because all nodes are at least half full (except root may have only 2 children) and all leaves are at the same level, the minimum number of data items n for a height $h > 0$ tree is...

$$n \geq \underbrace{2 \lceil M/2 \rceil^{h-1}}_{\text{minimum number of leaves}} \cdot \underbrace{\lceil L/2 \rceil}_{\text{minimum data per leaf}}$$

Exponential in height
because $\lceil M/2 \rceil > 1$

What makes B trees so disk friendly?

Many keys stored in one **internal node**

- All brought into memory in one disk access
- But only if we pick M wisely
- Makes the binary search over $M-1$ keys worth it (insignificant compared to disk access times)

Internal nodes contain only keys

- Any **find** wants only one data item; wasteful to load unnecessary items with internal nodes
- Only bring one **leaf** of data items into memory
- Data-item size does not affect what M is

Maintaining Balance

So this seems like a great data structure

It is

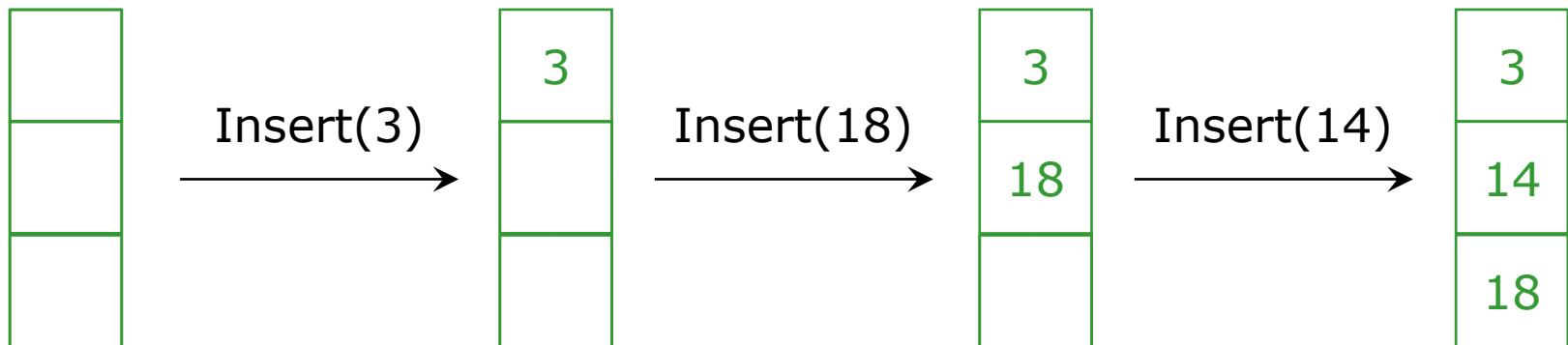
But we haven't implemented the other dictionary operations yet

- insert
- delete

As with AVL trees, the hard part is maintaining structure properties

Building a B-Tree

$$M = 3 \quad L = 3$$

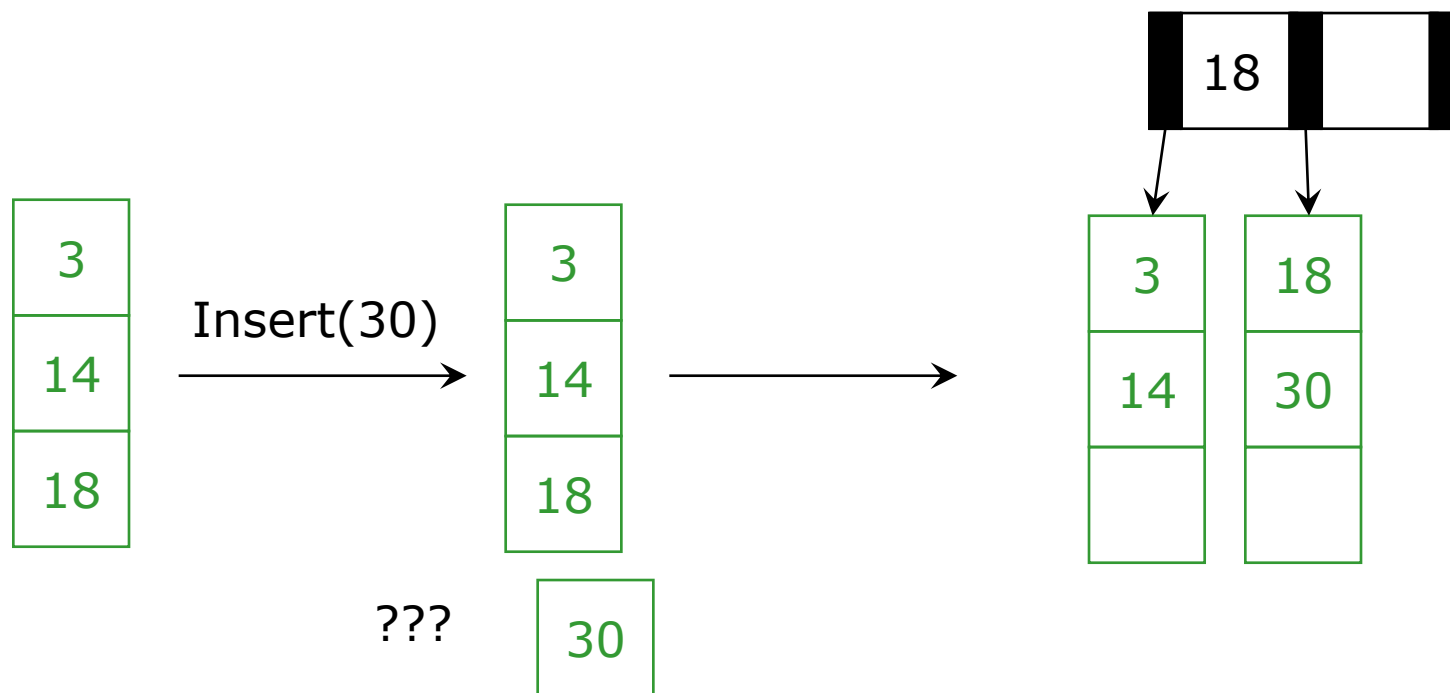


The empty B-Tree
(the **root** will be a
leaf at the beginning)

Simply need to
keep the keys
sorted in a leaf

Building a B-Tree

$$M = 3 \quad L = 3$$



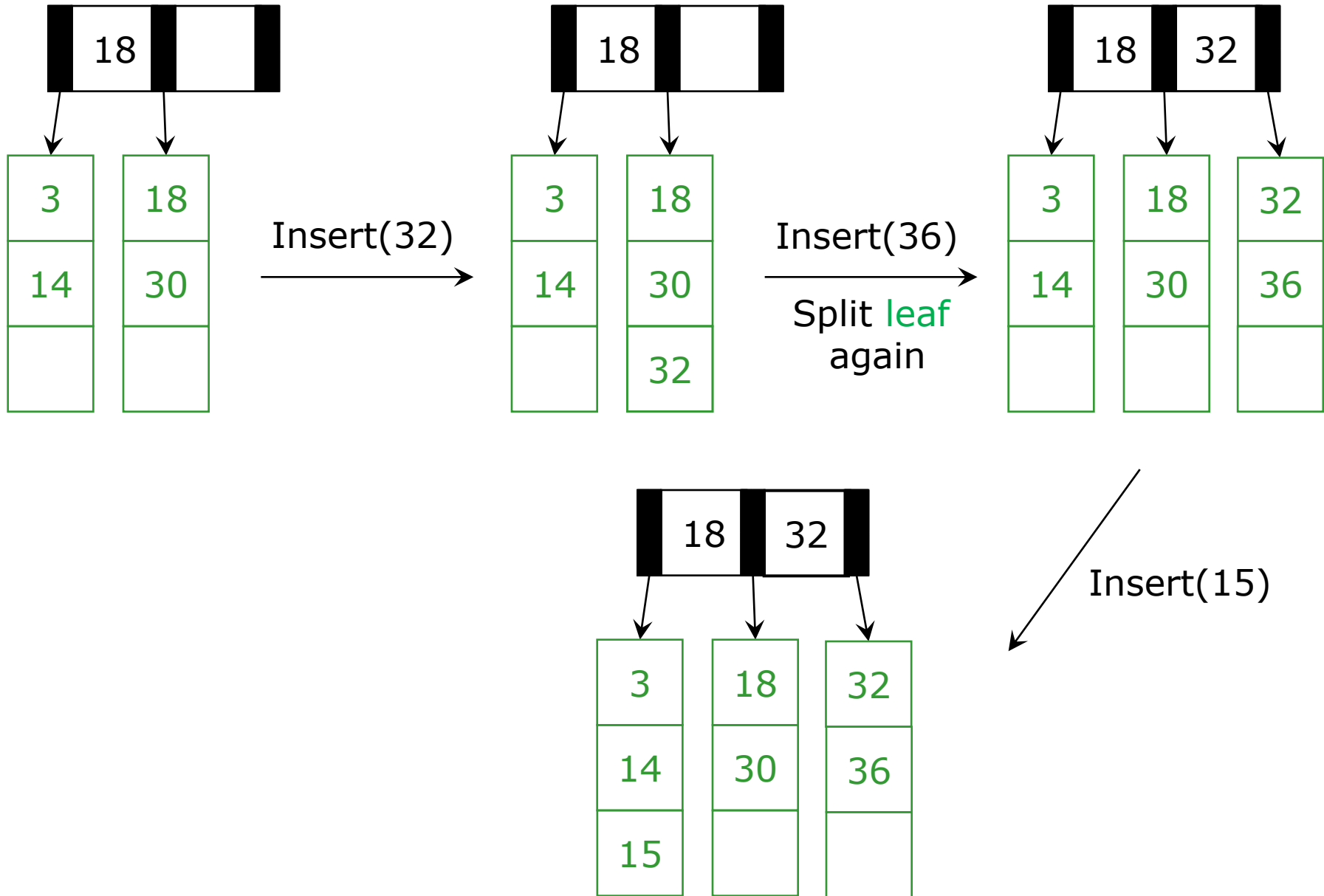
When we 'overflow' a leaf, we split it into 2 leaves

- Parent gains another child
- If there is no parent, we create one

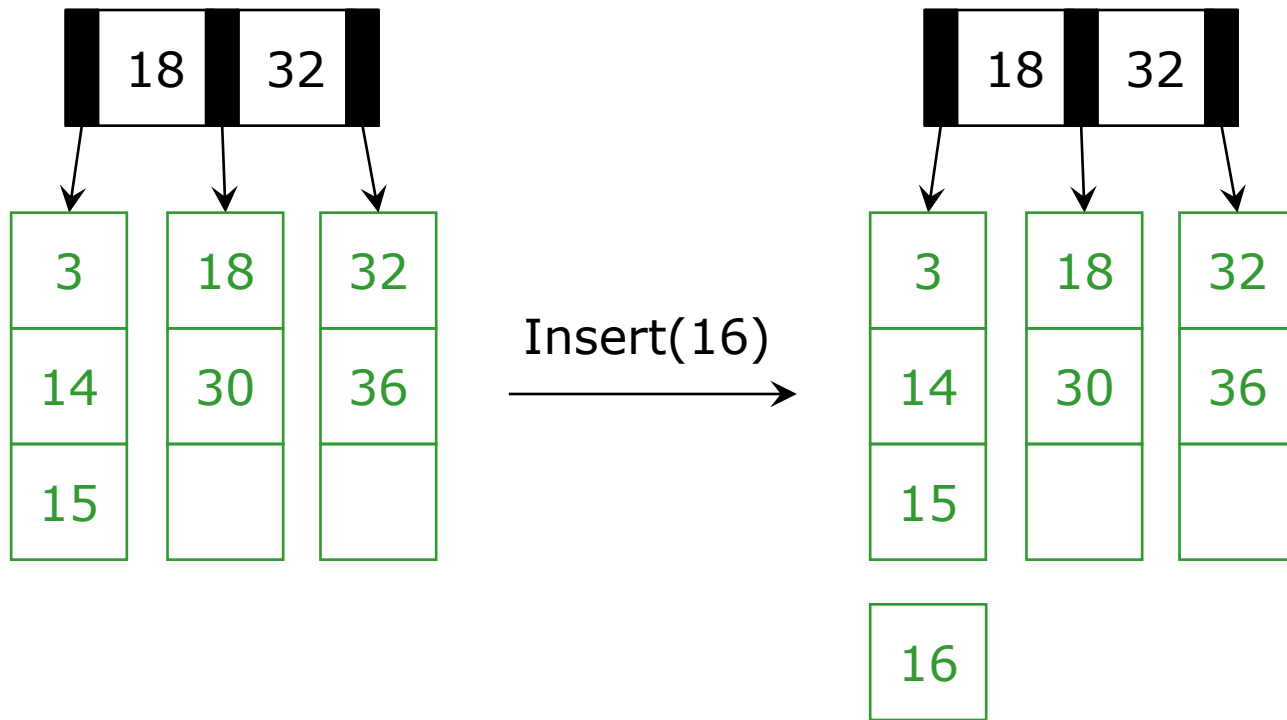
How do we pick the new key?

- Smallest element in right subtree

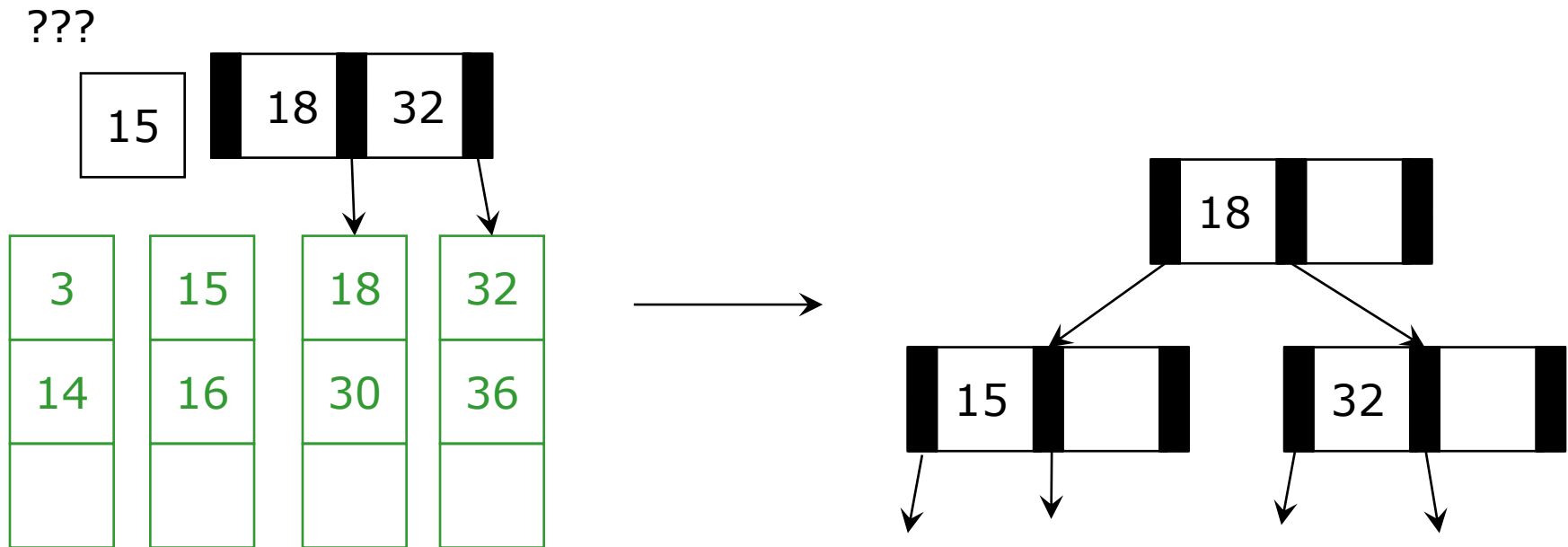
$$M = 3 \quad L = 3$$



$$M = 3 \quad L = 3$$

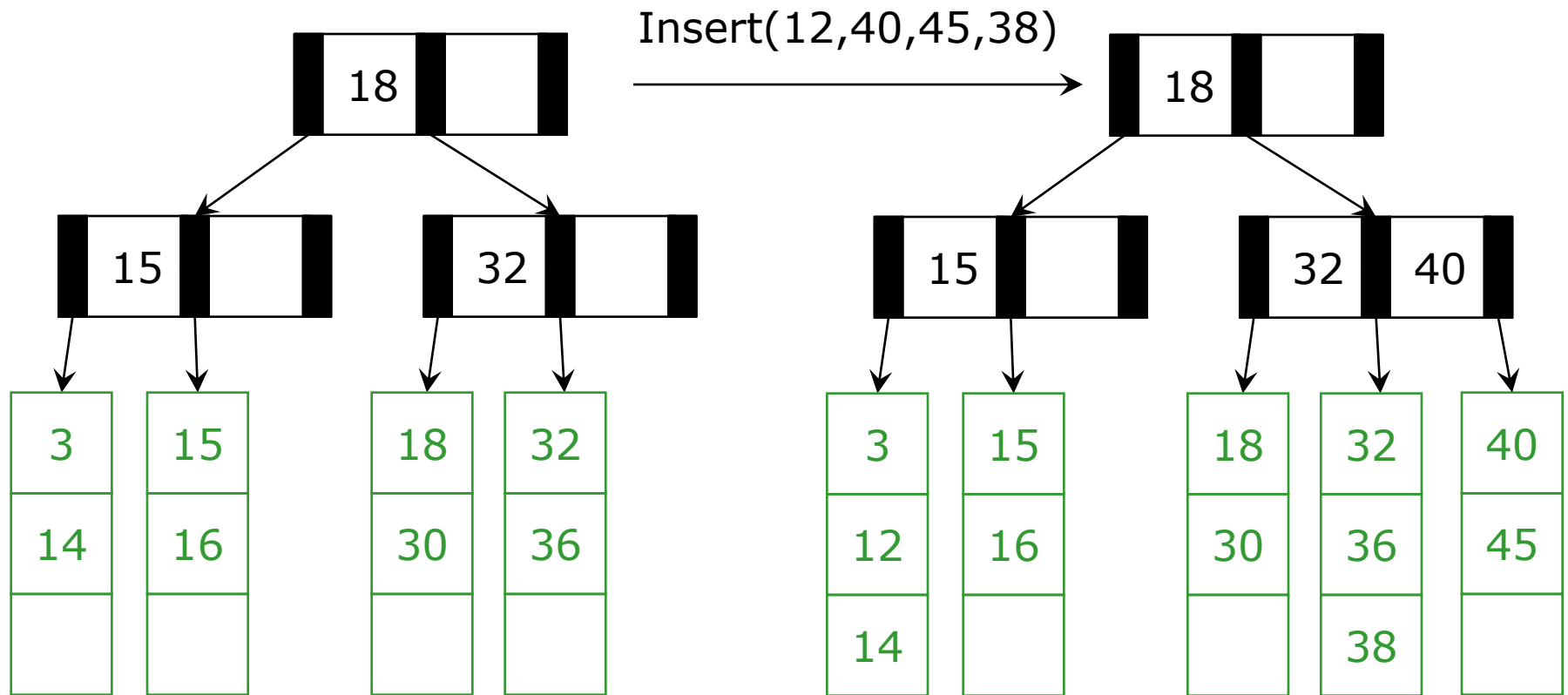


$$M = 3 \quad L = 3$$



Split the internal node
(in this case, the **root**)

$$M = 3 \quad L = 3$$



Given the **leaves** and the structure of the tree, we can always fill in internal node keys using the rule:

What is the smallest value in my right branch?

Insertion Algorithm

1. Insert the data in its leaf in sorted order
2. If the leaf now has $L+1$ items, overflow!
 - a. Split the leaf into two nodes:
 - Original leaf with $\lceil (L+1)/2 \rceil$ smaller items
 - New leaf with $\lfloor (L+1)/2 \rfloor = \lceil L/2 \rceil$ larger items
 - b. Attach the new child to the parent
 - Adding new key to parent in sorted order
3. If Step 2 caused the parent to have $M+1$ children, overflow the parent!

Insertion Algorithm (cont)

4. If an internal node (parent) has $M+1$ kids
 - a. Split the node into two nodes
 - Original node with $\lceil (M+1)/2 \rceil$ smaller items
 - New node with $\lfloor (M+1)/2 \rfloor = \lceil M/2 \rceil$ larger items
 - b. Attach the new child to the parent
 - Adding new key to parent in sorted order

Step 4 could make the parent overflow too

- Repeat up the tree until a node does not overflow
- If the root overflows, make a new root with two children. This is the only the tree height inceases

Worst-Case Efficiency of Insert

Find correct leaf:	$O(\log_2 M \log_M n)$
Insert in leaf:	$O(L)$
Split leaf:	$O(L)$
Split parents all the way to root:	$O(M \log_M n)$
Total	$O(L + M \log_M n)$

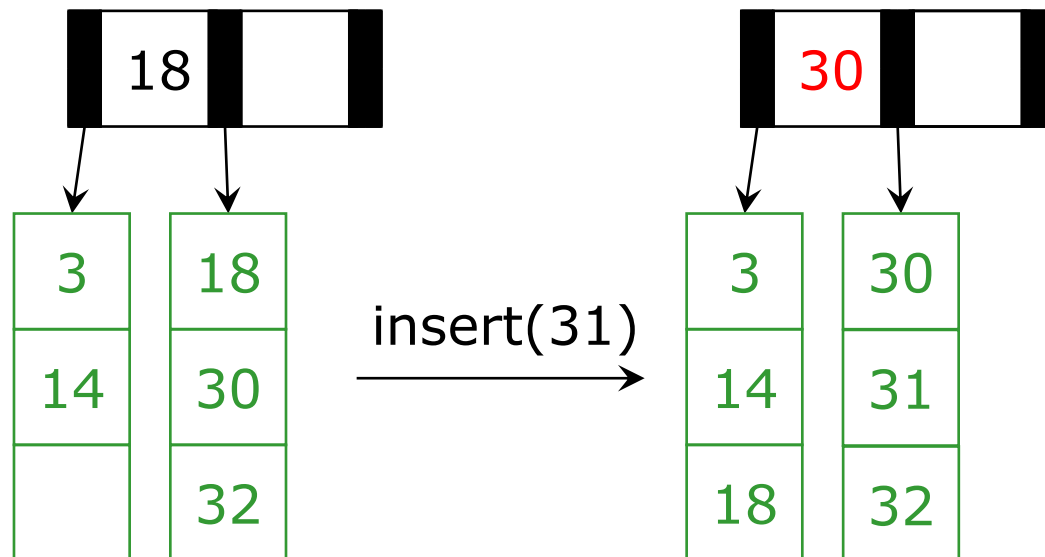
But it's not that bad:

- Splits are rare (only if a node is FULL)
- M and L are likely to be large
- After a split, nodes will be half empty
- Splitting the **root** is thus extremely rare
- Reducing disk accesses is name of the game:
inserts are thus $O(\log_M n)$ on average

Adoption for Insert

We can sometimes avoid splitting via a process called adoption

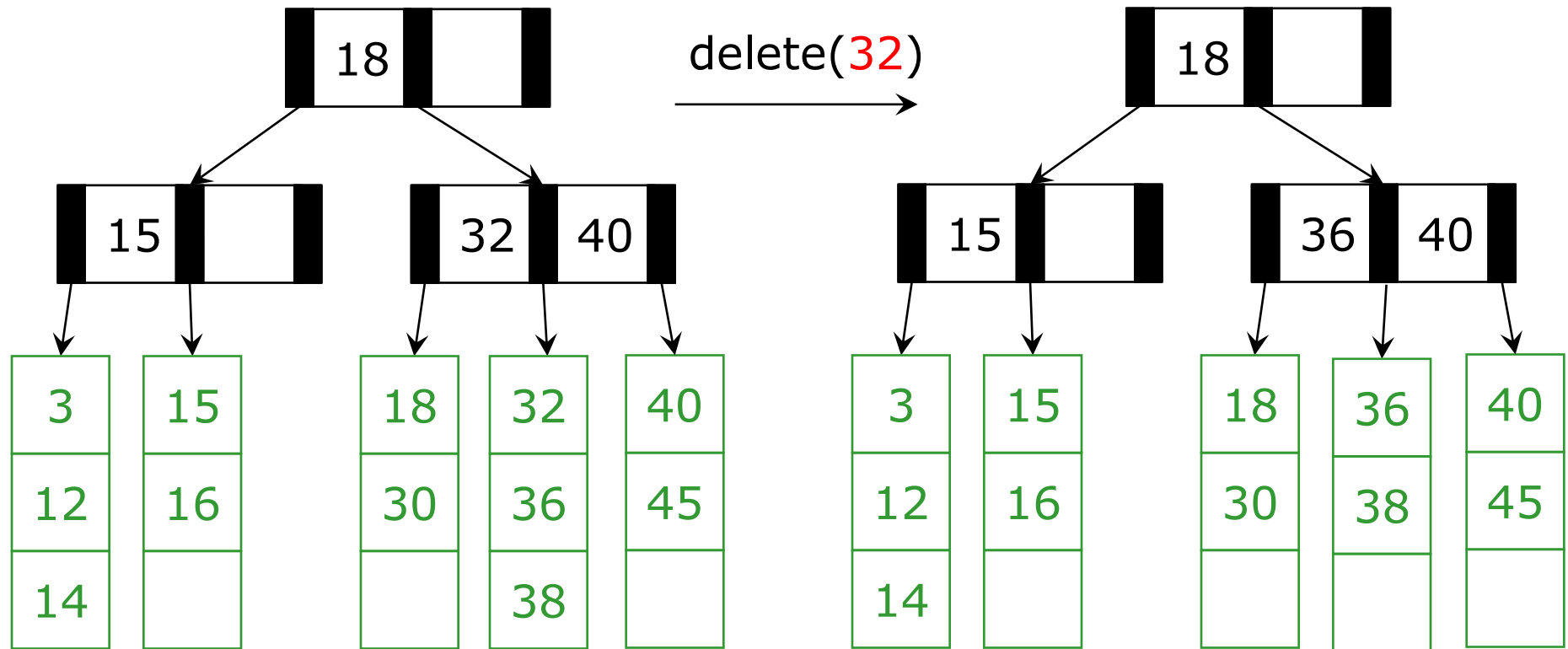
Example:



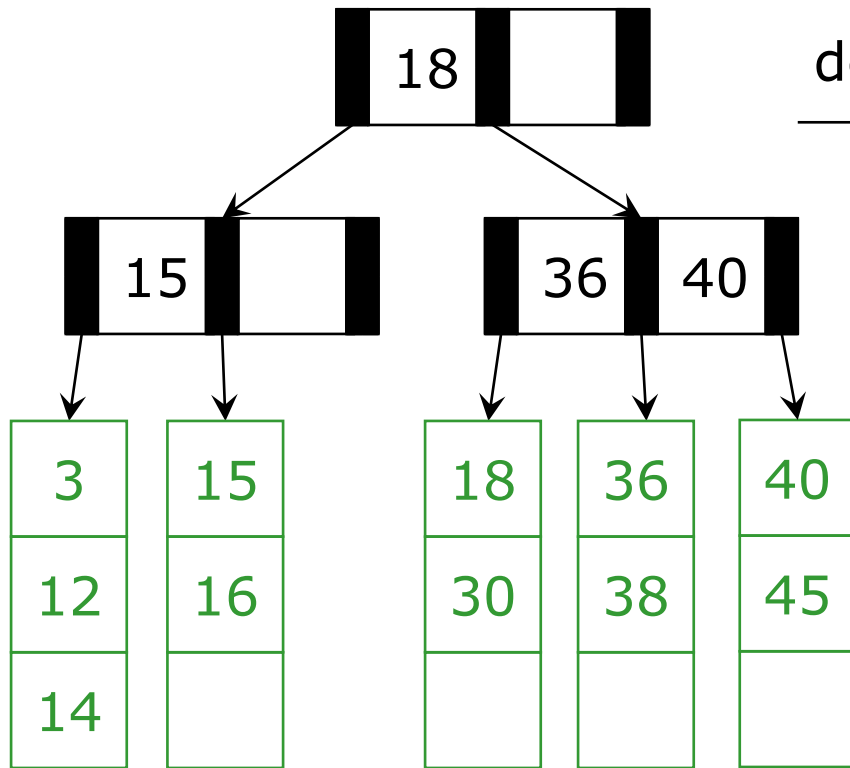
- Notice correction by changing parent keys
- Implementation not necessary for efficiency
- But introduced as it leads to how deletion works

Deletion

$$M = 3 \quad L = 3$$

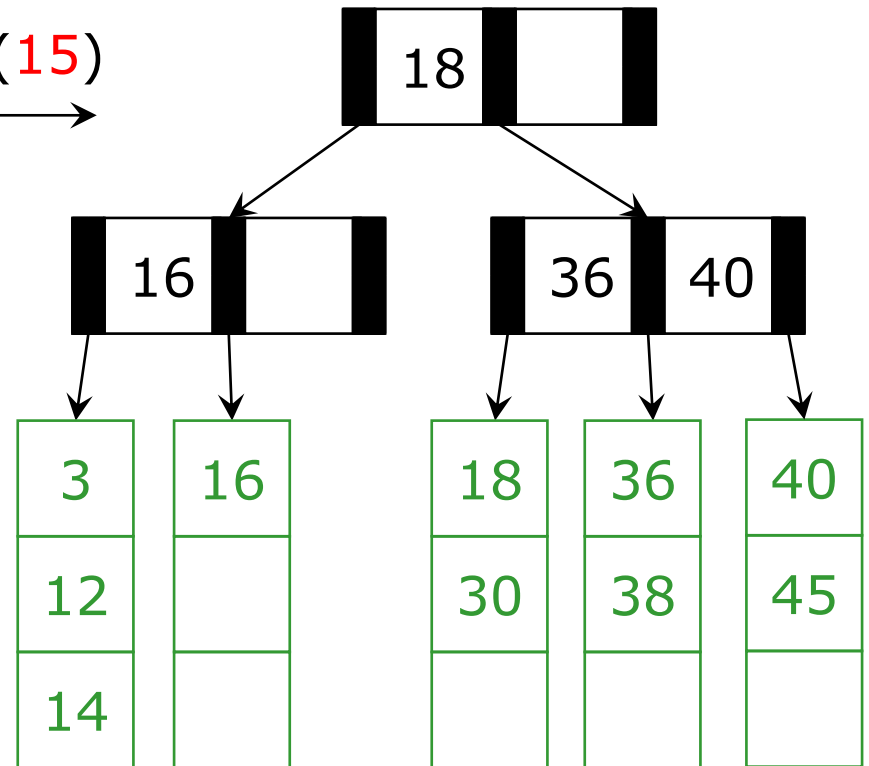


$$M = 3 \quad L = 3$$



Are we okay?

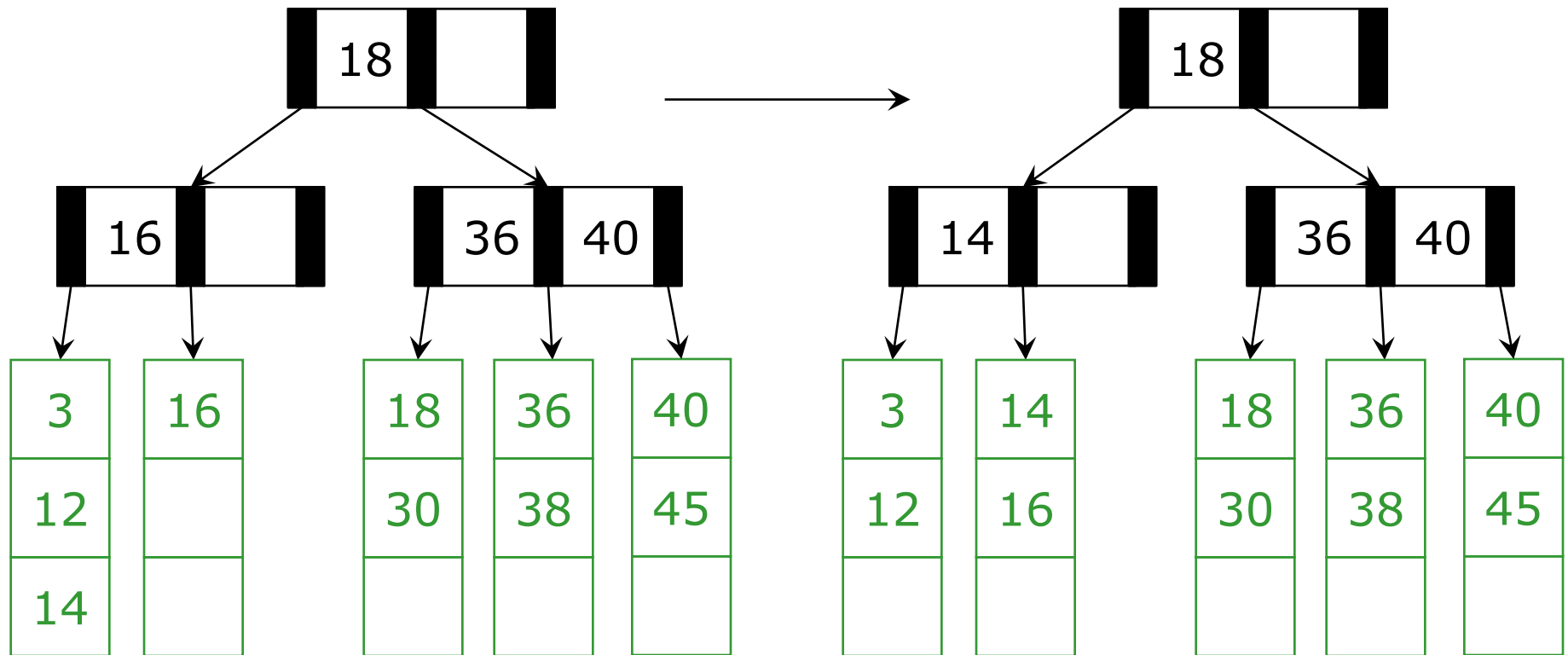
delete(15)



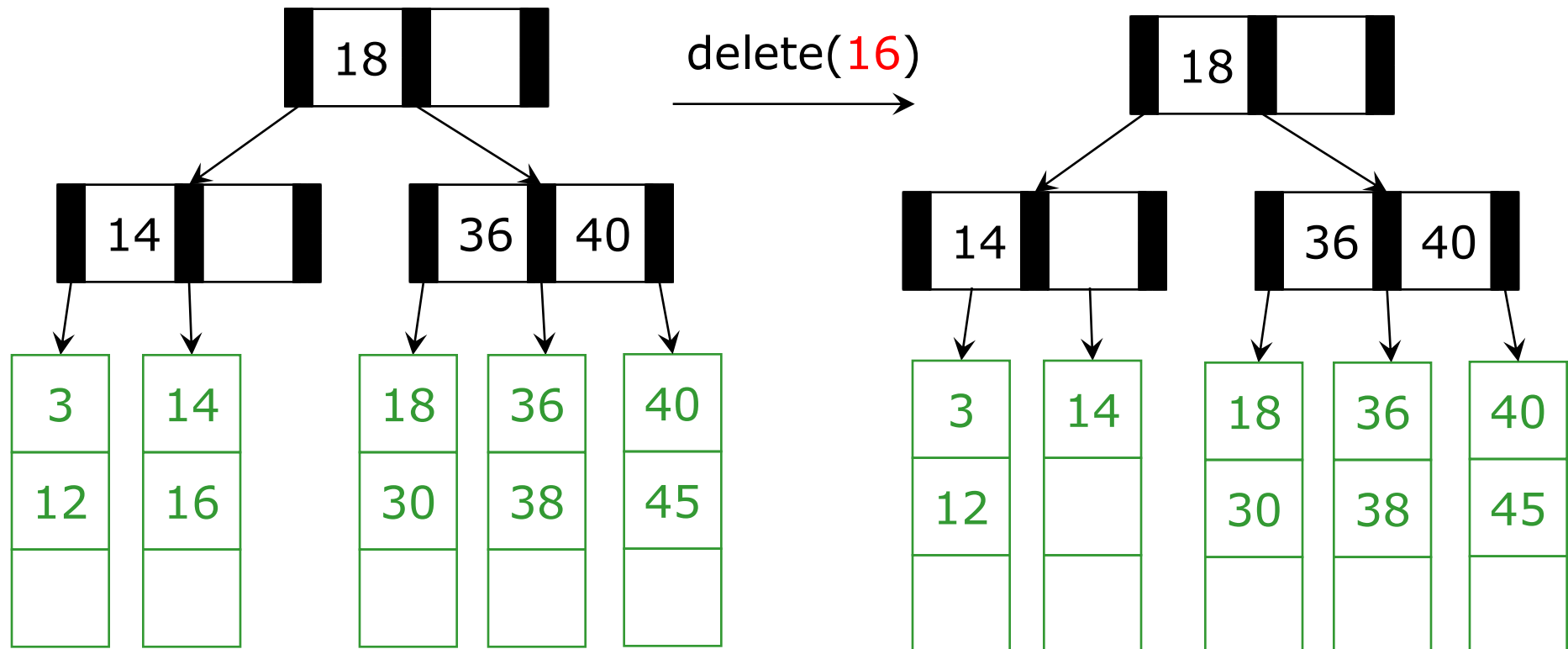
Dang, not half full

Are you using that 14?
Can I borrow it?

$$M = 3 \quad L = 3$$



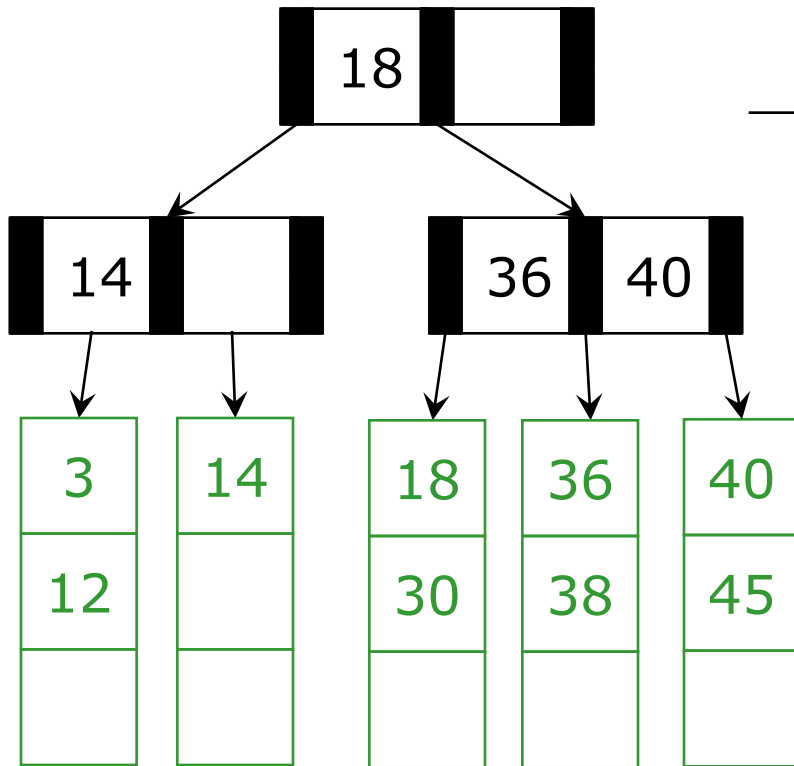
$$M = 3 \quad L = 3$$



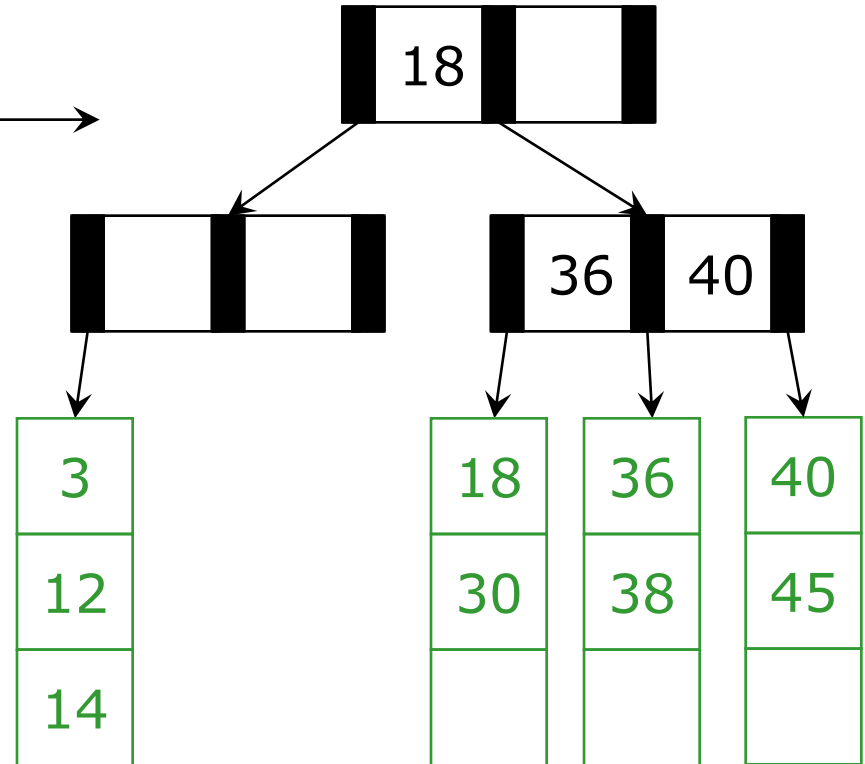
Are you using that 12? Yes

Are you using that 18? Yes

$$M = 3 \quad L = 3$$



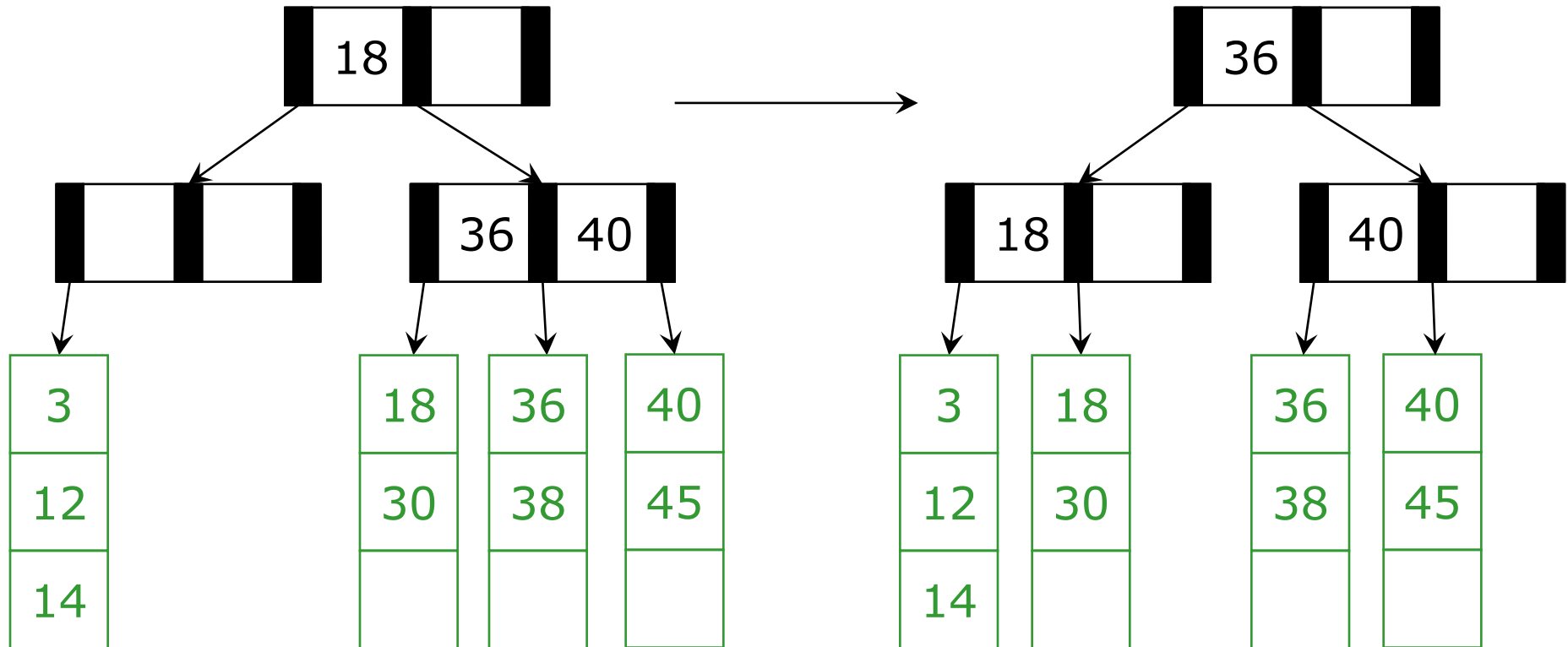
Well, let's just consolidate our leaves since we have the room



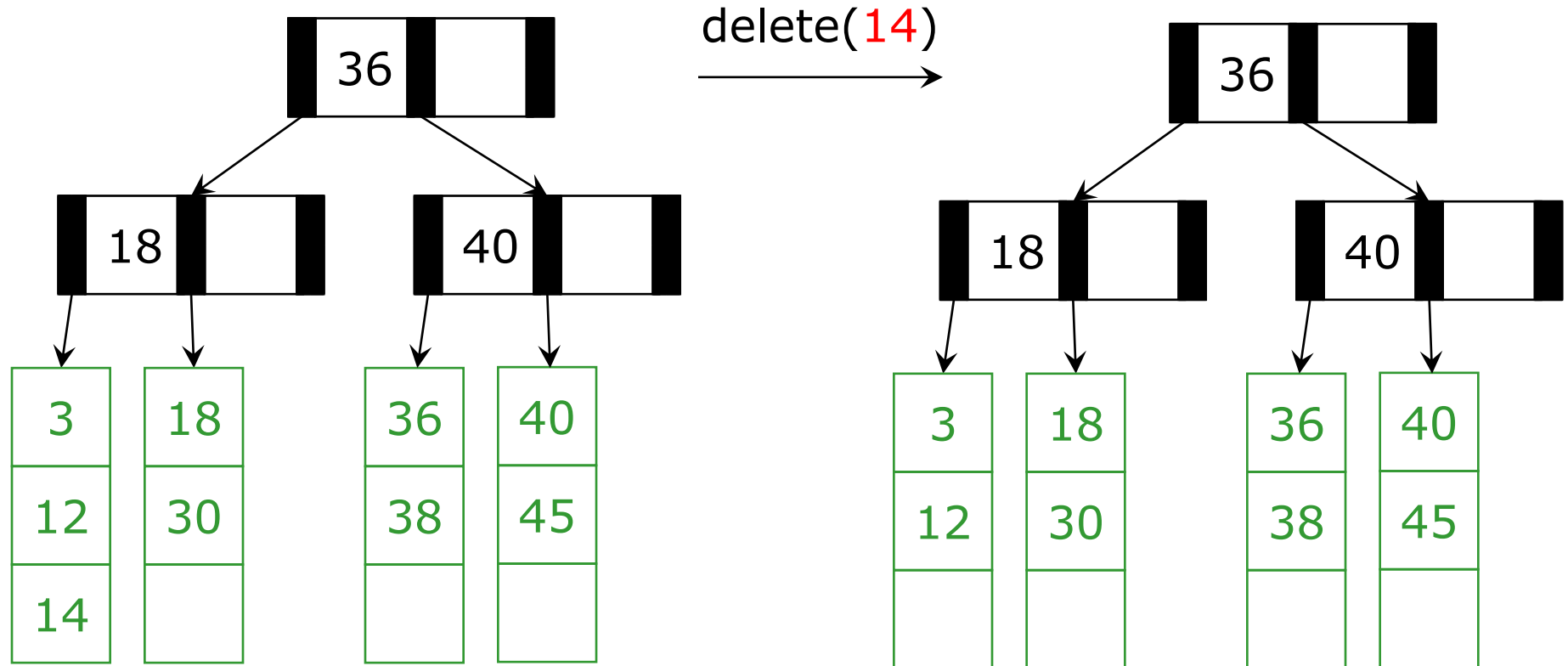
Oops. Not enough leaves

Are you using that 18/30?

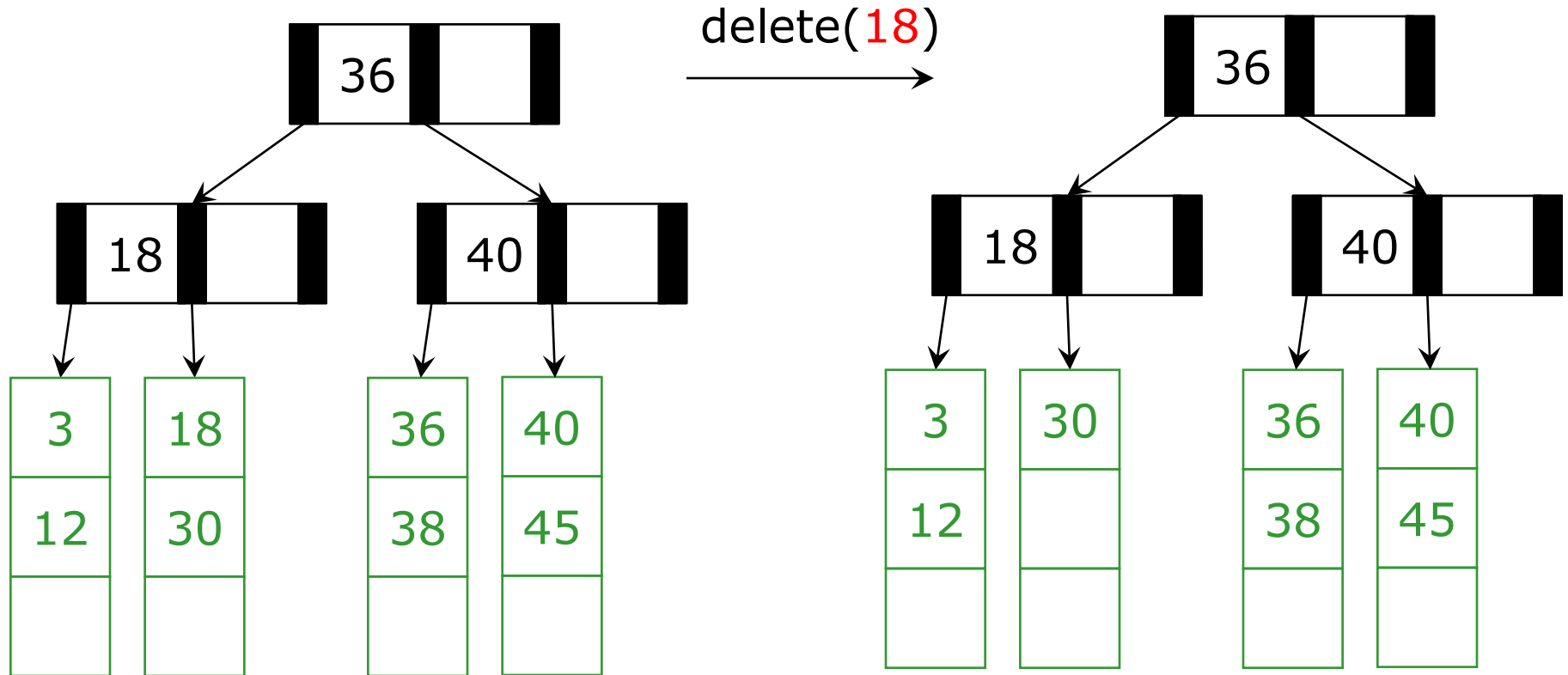
$$M = 3 \quad L = 3$$



$$M = 3 \quad L = 3$$

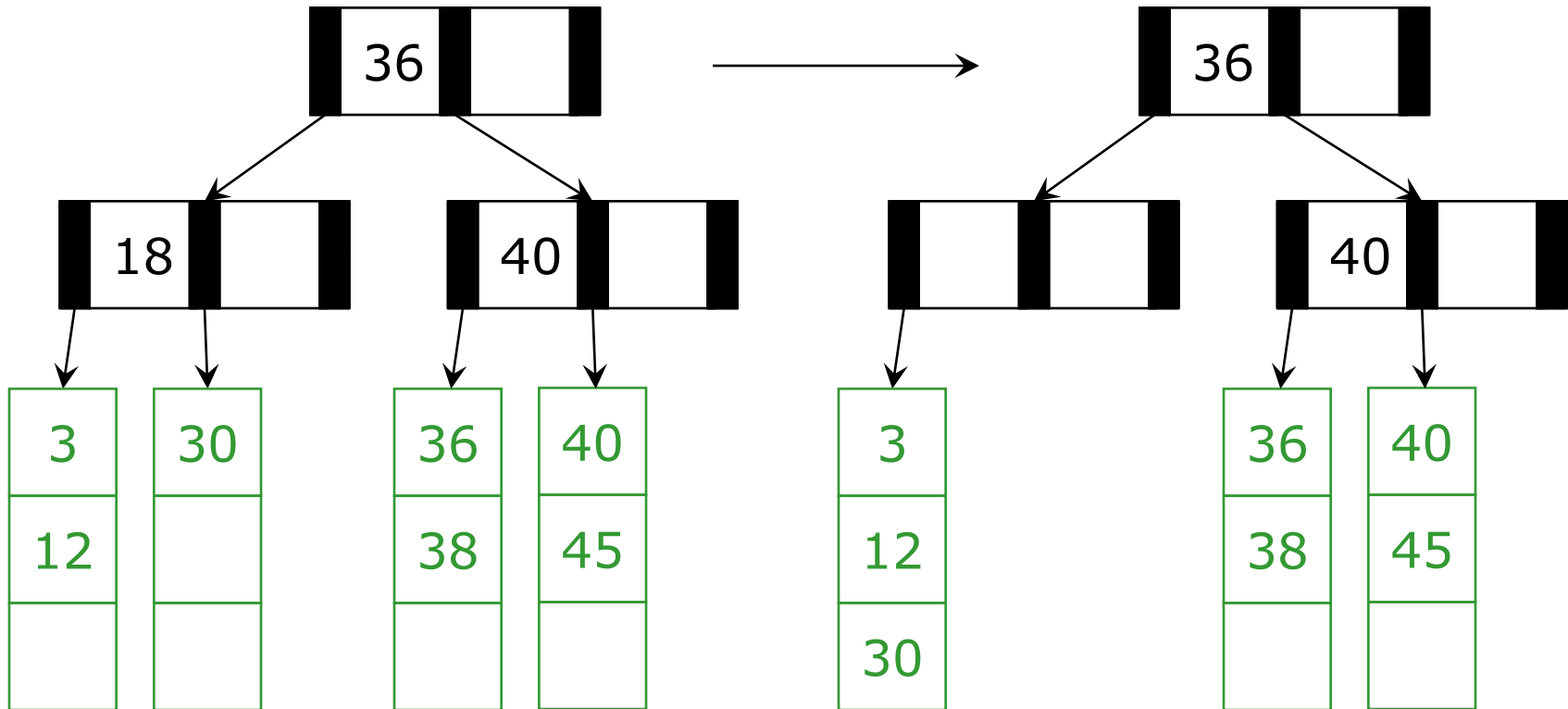


$$M = 3 \quad L = 3$$



Oops. Not enough in leaf

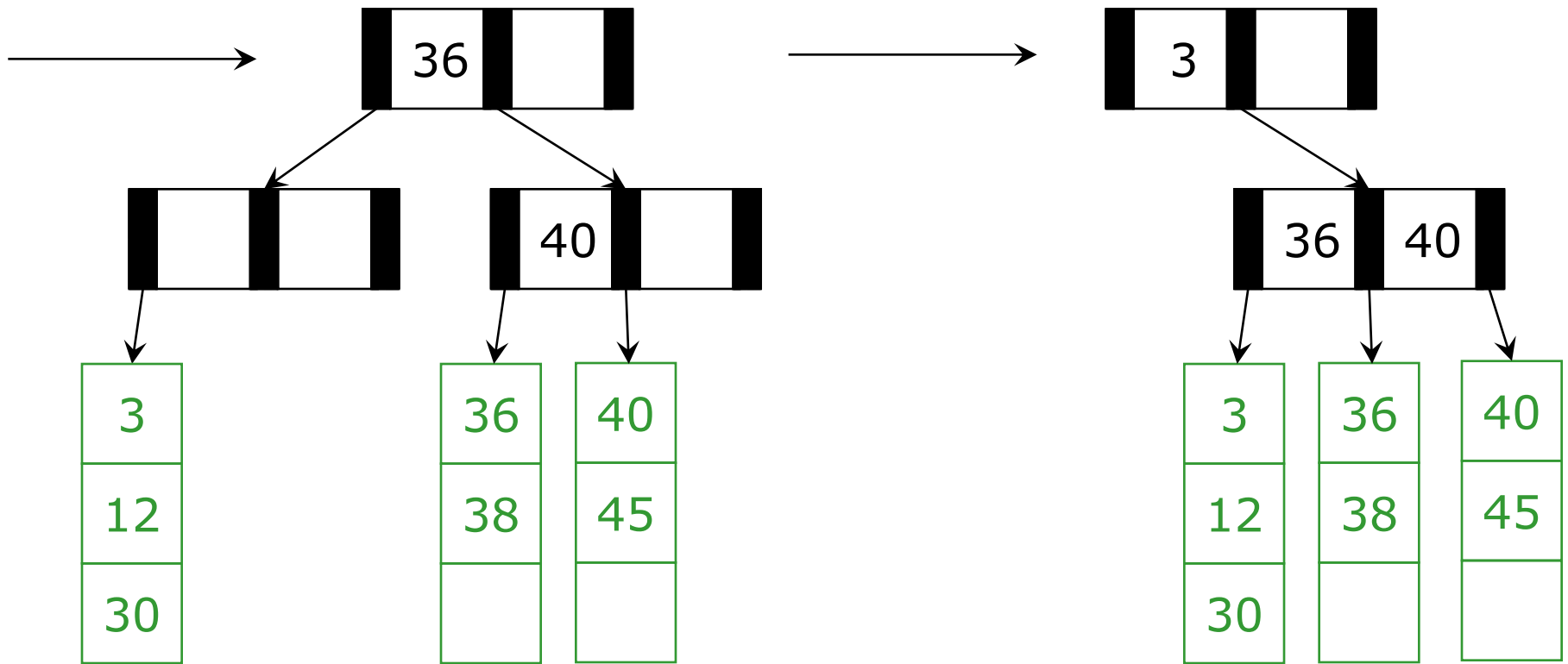
$$M = 3 \quad L = 3$$



We will borrow as before

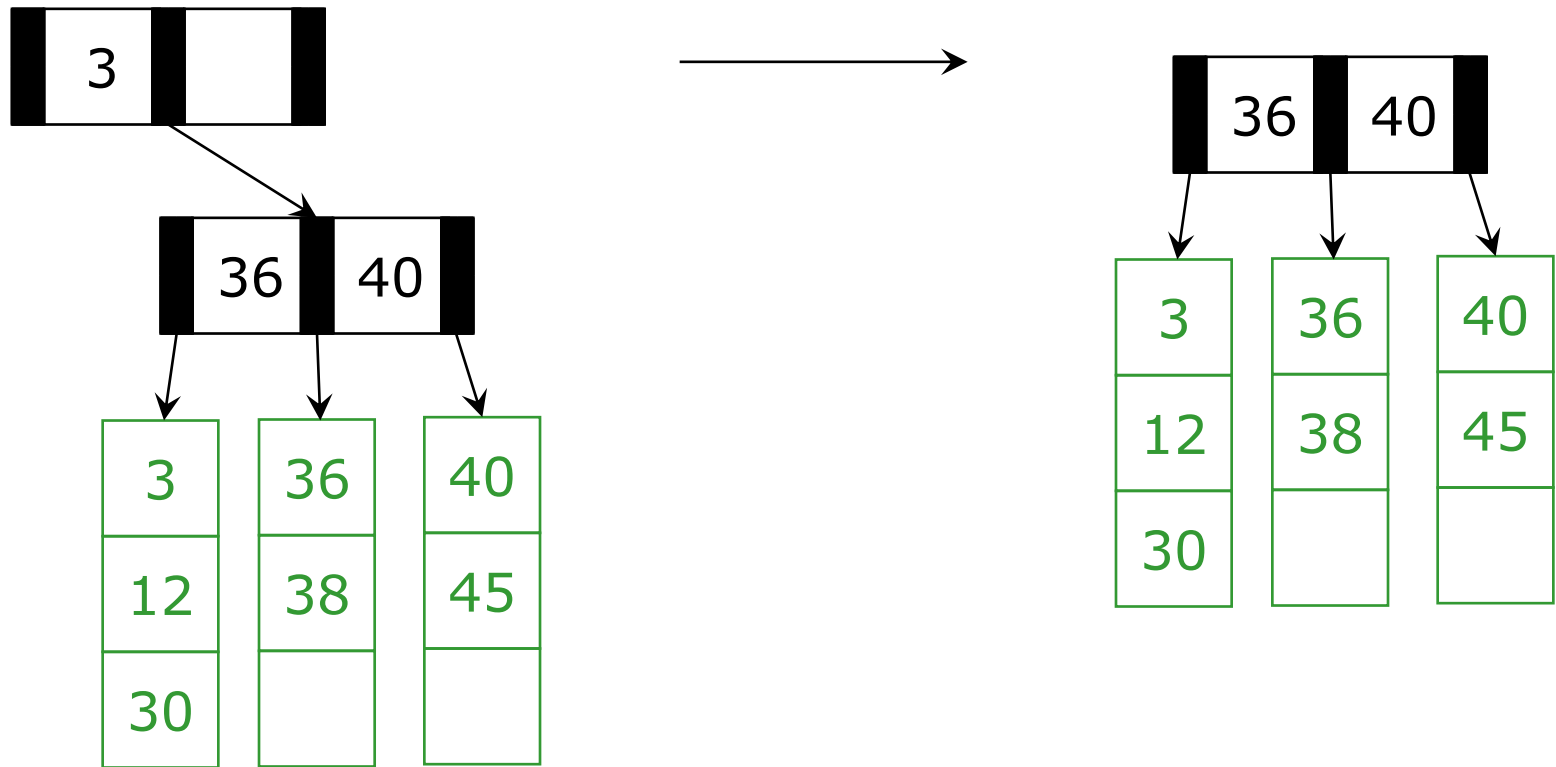
Oh no. Not enough leaves
and we cannot borrow!

$$M = 3 \quad L = 3$$



We have to move up a node and collapse into a new root.

$$M = 3 \quad L = 3$$



Huh, the root is pretty small. Let's reduce the tree's height.

Deletion Algorithm

1. Remove the data 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 node with neighbor
 - Guaranteed to have a legal number of items $\lfloor L/2 \rfloor + \lceil L/2 \rceil = L$
 - Parent now has one less node
1. If Step 2 caused parent to have $\lceil M/2 \rceil - 1$ children, underflow!

Deletion Algorithm

4. If an internal node has $\lceil M/2 \rceil - 1$ children
 - If a neighbor has $>\lceil M/2 \rceil$ items, adopt and update parent
 - Else merge node with neighbor
 - Guaranteed to have a legal number of items
 - Parent now has one less node, may need to continue underflowing up the tree

Fine if we merge all the way up to the root

- If the root went from 2 children to 1, delete the root and make child the root
- This is the only case that decreases tree height

Worst-Case Efficiency of Delete

Find correct leaf:	$O(\log_2 M \log_M n)$
Insert in leaf:	$O(L)$
Split leaf:	$O(L)$
Split parents all the way to root:	$O(M \log_M n)$
Total	$O(L + M \log_M n)$

But it's not that bad:

- Merges are not that common
- After a merge, a node will be over half full
- Reducing disk accesses is name of the game:
deletions are thus $O(\log_M n)$ on average

Implementing B Trees in Java?

Assuming our goal is efficient number of disk accesses, Java was not designed for this

This is not a programming languages course

Still, it is worthwhile to know enough about "how Java works" and why this is probably a bad idea for B trees

The key issue is extra levels of indirection...

Naïve Approach

Even if we assume data items have int keys, you cannot get the data representation you want for "really big data"

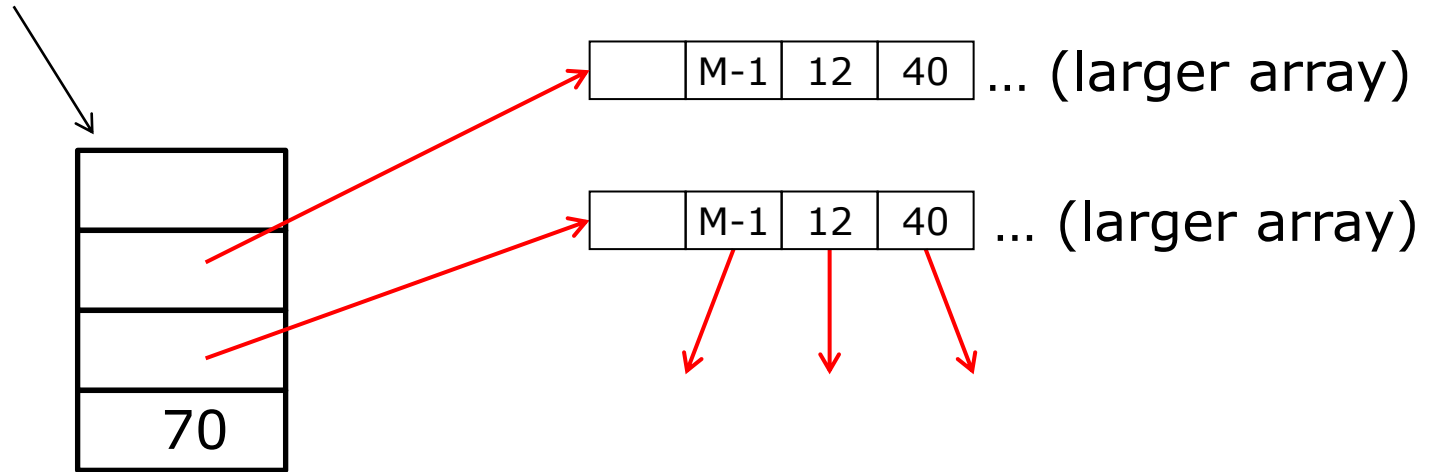
```
interface Keyed<E> {
    int key(E);
}

class BTreeNode<E> implements Keyed<E> {
    static final int M = 128;
    int[] keys = new int[M-1];
    BTreeNode<E>[] children = new BTreeNode[M];
    int numChildren = 0;
    ...
}

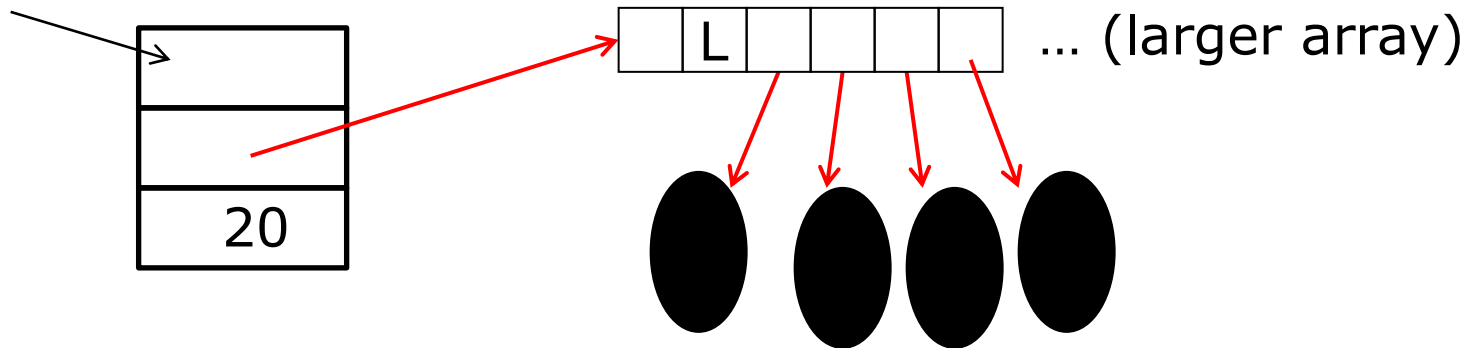
class BTreeLeaf<E> {
    static final int L = 32;
    E[] data = (E[])new Object[L];
    int numItems = 0;
    ...
}
```

What that looks like

BTreeNode (3 objects with "header words")



BTreeLeaf (data objects not in contiguous memory)



The moral

The point of B trees is to keep related data in contiguous memory

All the red references on the previous slide are inappropriate

- As minor point, beware the extra "header words"

But that is "the best you can do" in Java

- Again, the advantage is generic, reusable code
- But for your performance-critical web-index, not the way to implement your B-Tree for terabytes of data

Other languages better support "flattening objects into arrays"

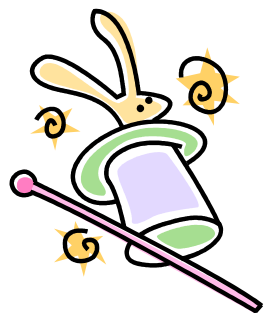
The national data structure of the Netherlands

HASH TABLES

Where We Are With Dictionaries

For dictionary with n key/value pairs

	insert	find	delete
Unsorted linked-list	$O(1)$	$O(n)$	$O(1)$
Unsorted array	$O(1)$	$O(n)$	$O(1)$
Sorted linked list	$O(n)$	$O(n)$	$O(1)$
Sorted array	$O(n)$	$O(\log n)$	$O(n)$
Balanced tree	$O(\log n)$	$O(\log n)$	$O(\log n)$



Hash Table

$O(1)$

$O(1)$

$O(1)$

"A magical array"

Wait...

Balanced trees give $O(\log n)$ worst-case
Hash tables give $O(1)$ on average

Constant time is better!

**So why did we learn about
balanced trees?**

Challenge of Hash Tables

Hashing is difficult to achieve

- A hash function must be fast to calculate
- Average $O(1)$ requires minimal collisions

Hash tables are slow for some operations as compared to balanced trees

- FindMin, FindMax, Predecessor, and Successor go from $O(\log n)$ to $O(n)$
- printSorted goes from $O(n)$ to $O(n \log n)$

Moral

If you need to frequently use operations based on sort order,

Then you may prefer a balanced BST instead of a hash table

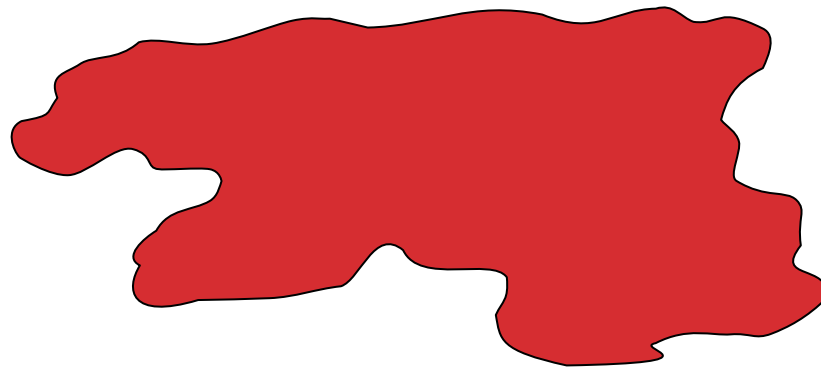
If the emphasis is on fast lookups,

Then a hash table is probably better

Hash Tables

A hash table is an array of some fixed size

Basic idea:



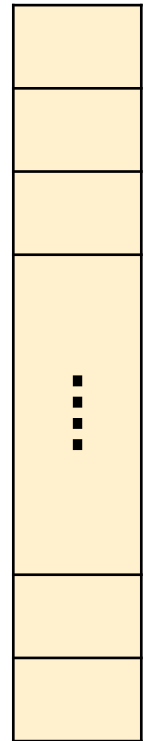
key space (e.g., integers, strings)

hash function:
 $\text{index} = h(\text{key})$



hash table

0



size - 1

The goal:

Aim for constant-time find, insert, and delete "on average" under reasonable assumptions

Hash Tables

Basic Structure

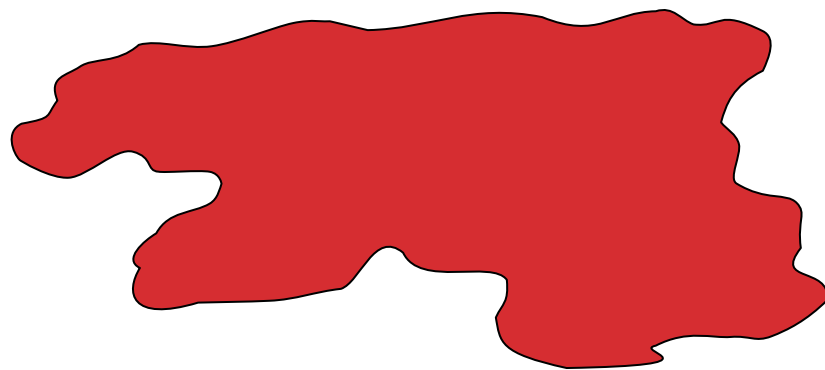
- m possible keys (m typically large, even infinite)
- Table is expected to have only n items
- n is much less than m (often written $n \ll m$)

Many dictionaries have this property

- Compiler:
All possible identifiers allowed by the language
vs. those used in some file of one program
- Database:
All possible student names vs. students enrolled
- Artificial Intelligence:
All possible chess-board configurations vs.
those considered by the current player

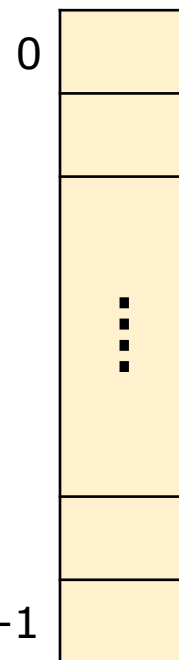
An Ideal Hash Functions

- Is fast to compute
- Rarely hashes two keys to the same index
 - Known as *collisions*
 - Zero collisions often impossible in theory but reasonably achievable in practice



key space (e.g., integers, strings)

hash function:
index = h(key)

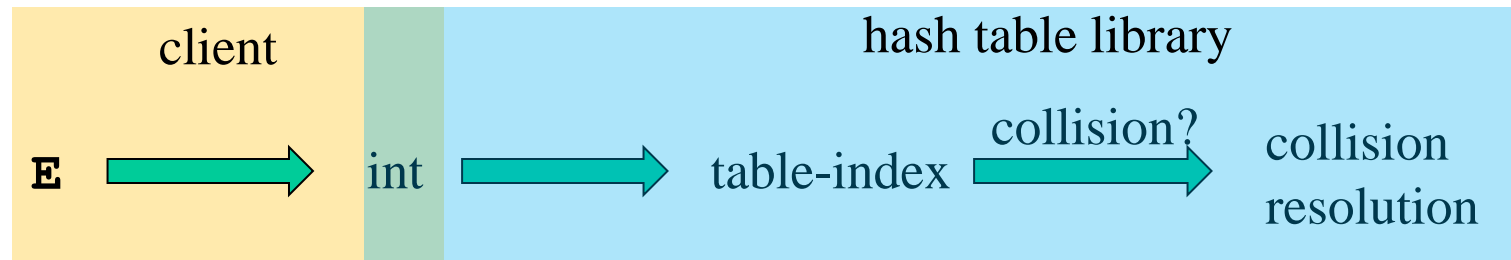


Who Hashes What

For a hash table to be generic (store elements of type E), we need E to be:

- Comparable: order any two E (for all dictionaries)
- Hashable: convert any E to an int

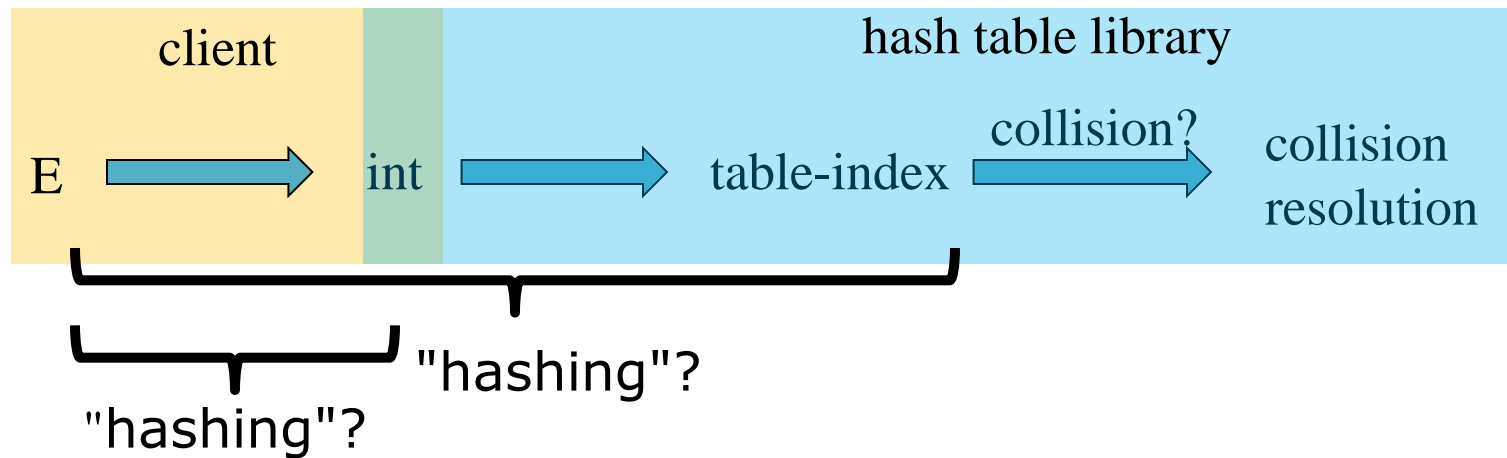
When hash tables are a reusable library, the division of responsibility involves two roles:



We will learn both roles, but most programmers "in the real world" spend more time as clients while understanding the library

More on Roles

Some ambiguity in terminology as to which parts are "hashing"



Our view is that both are important

More on Roles

Both roles must both contribute to minimizing collisions (heuristically)

Client should aim for different ints for the expected item keys

- Do not "waste" any part of E or the int's 32 bits

Library should aim for putting "similar" ints in different indices


- conversion to index is almost always "mod table-size"
- using prime numbers for table-size is common

What to Hash?

We will focus on two most common things to hash: **ints** and **strings**

If you have objects with several fields, it is usually best to hash most of the "identifying fields" to avoid collisions:

```
class Person {  
    String firstName, middleName, lastName;  
    Date birthDate;  
    ...  
}
```



use these four values

An inherent trade-off:

hashing-time vs. collision-avoidance

Hashing Integers

key space = integers

Simple hash function:

$$h(\text{key}) = \text{key} \% \text{TableSize}$$

- Client: $f(x) = x$
- Library: $g(x) = f(x) \% \text{TableSize}$
- Fairly fast and natural

Example:

- TableSize = 10
- Insert keys 7, 18, 41, 34, 10

0	
1	10
2	41
3	
4	
5	34
6	
7	
8	7
9	18

Collision Avoidance

With $(x \% \text{TableSize})$, number of collisions depends on

- the ints inserted
- TableSize

Larger table-size tends to help, but not always

- Example: 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 strategies do better with prime size

More Arguments for a Prime Size

If TableSize is 60 and...

- Lots of data items are multiples of 2, wasting 50% of table
- Lots of data items are multiples of 5, wasting 80% of table
- Lots of data items are multiples of 10, wasting 90% of table

If TableSize is 61...

- Collisions can still happen but 2, 4, 6, 8, ... will fill in table
- Collisions can still happen, but 5, 10, 15, ... will fill in table
- Collisions can still happen but 10, 20, 30, ... will fill in table

A Tidbit from Number Theory

If x and y are "co-prime" ($\gcd(x, y) = 1$),
then $(a * x) \% y = (b * x) \% y$
if and only if $a \% y = b \% y$

Hashing non-integer keys

If keys are not ints, the client must provide a means to convert the key to an int

Programming Trade-off:

- Calculation speed
- Avoiding distinct keys hashing to same ints

Hashing Strings

Key space $K = s_0s_1s_2\cdots s_{k-1}$
where s_i are chars: $s_i \in [0, 256]$

Some choices: Which ones best avoid collisions?

$$h(K) = (s_0) \% \text{TableSize}$$

$$h(K) = \left(\sum_{i=0}^{k-1} s_i \right) \% \text{TableSize}$$

$$h(K) = \left(\sum_{i=0}^{k-1} s_i \cdot 37^i \right) \% \text{TableSize}$$

Combining Hash Functions

A few rules of thumb / tricks:

1. Use all 32 bits (be careful with negative numbers)
2. Use different overlapping bits for different parts of the hash
 - This is why a factor of 37^i works better than 256^i
 - Example: "abcde" and "ebcda"
3. When smashing two hashes into one hash, use bitwise-xor
 - bitwise-and produces too many 0 bits
 - bitwise-or produces too many 1 bits
4. Rely on expertise of others; consult books and other resources for standard hashing functions
5. Advanced: If keys are known ahead of time, a *perfect hash* can be calculated

A Final Tidbit about Hash Functions

Hash functions are typically *one-way functions*:

- Calculating $h(x) = y$ is easy/straightforward
- Calculating $h^{-1}(y) = x$ is difficult/impossible

This complexity of calculating the inverse of a hash function is very useful in security/encryption

- Generating signatures of messages
- You might recognize some names:
SHA-1, MD4, MD5, etc.

Calling a State Farm agent is not an option...

COLLISION RESOLUTION

Collision Resolution

Collision:

When two keys map to the same location in the hash table

We try to avoid it, but the number of keys always exceeds the table size

Ergo, hash tables generally must support some form of **collision resolution**

Flavors of Collision Resolution

Separate Chaining

Open Addressing

- Linear Probing
- Quadratic Probing
- Double Hashing

Terminology Warning

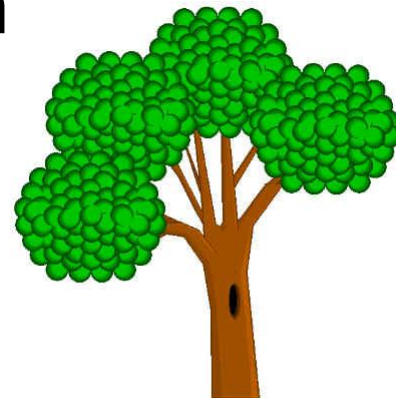
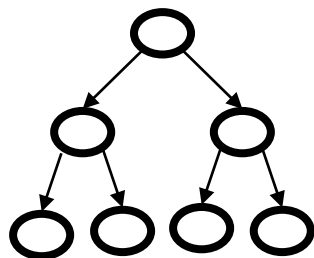
We and the book use the terms

- "chaining" or "separate chaining"
- "open addressing"

Very confusingly, others use the terms

- "open hashing" for "chaining"
- "closed hashing" for "open addressing"

We also do trees upside-down



Separate Chaining

0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	/
9	/

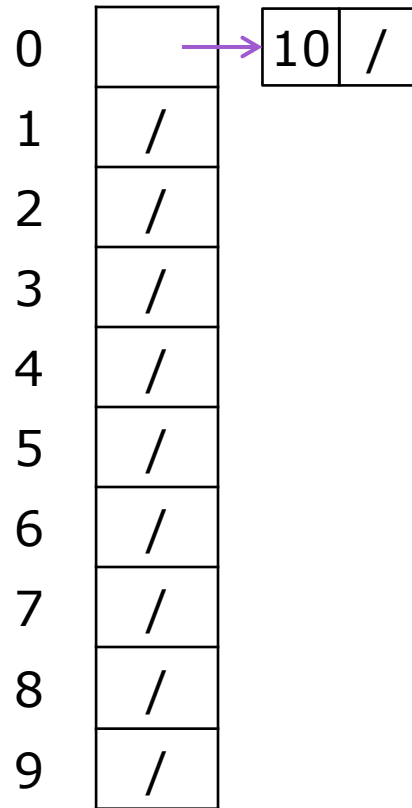
All keys that map to the same table location are kept in a linked list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

insert 10, 22, 86, 12, 42
with $h(x) = x \% 10$

Separate Chaining



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

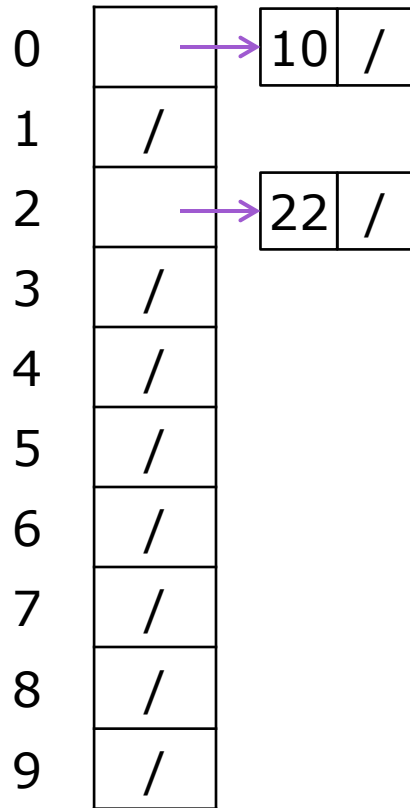
As easy as it sounds

Example:

insert 10, 22, 86, 12, 42
with $h(x) = x \% 10$

Separate Chaining

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



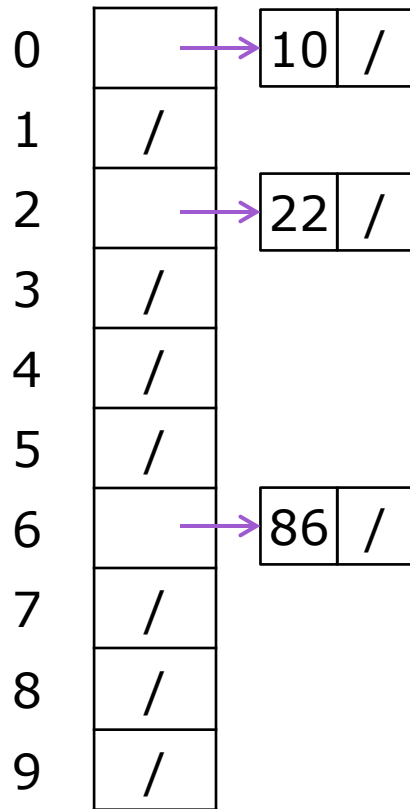
As easy as it sounds

Example:

insert 10, 22, 86, 12, 42
with $h(x) = x \% 10$

Separate Chaining

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



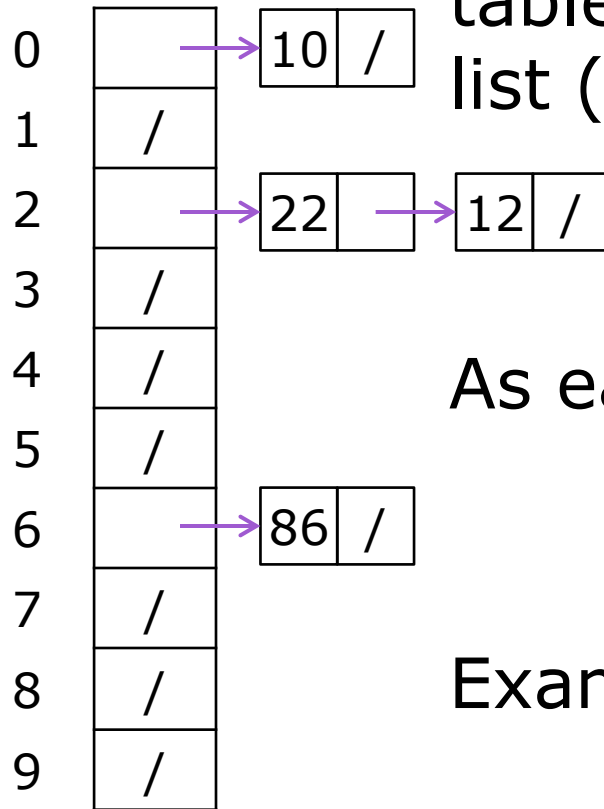
As easy as it sounds

Example:

insert 10, 22, 86, 12, 42
with $h(x) = x \% 10$

Separate Chaining

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



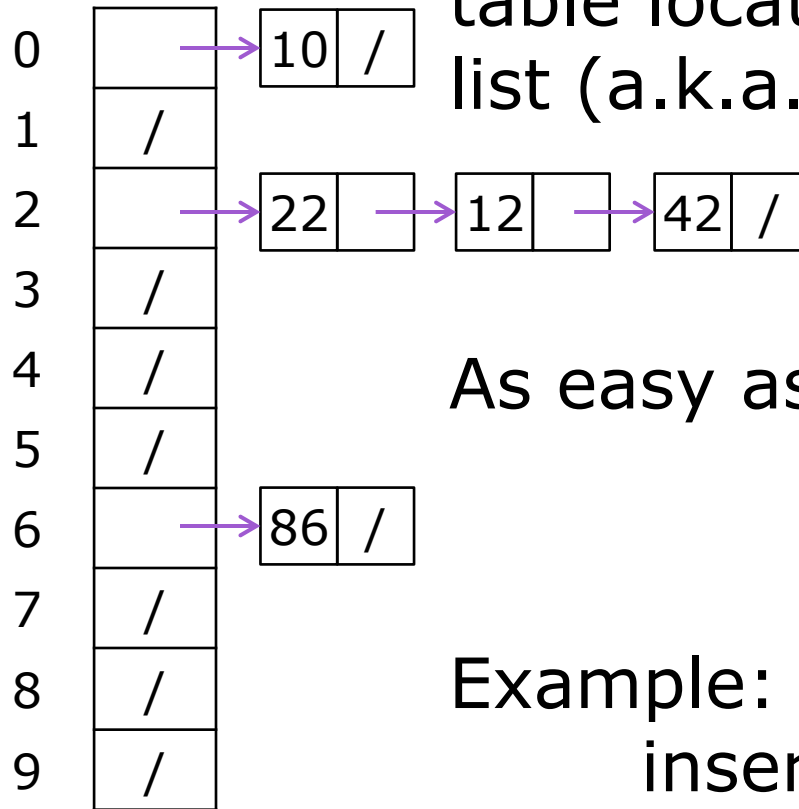
As easy as it sounds

Example:

insert 10, 22, 86, 12, 42
with $h(x) = x \% 10$

Separate Chaining

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



As easy as it sounds

Example:

insert 10, 22, 86, 12, 42
with $h(x) = x \% 10$

Thoughts on Separate Chaining

Worst-case time for find?

- Linear
- But only with really bad luck or bad hash function
- Not worth avoiding (e.g., with balanced trees at each bucket)
 - Keep small number of items in each bucket
 - Overhead of tree balancing not worthwhile for small n

Beyond asymptotic complexity, some "data-structure engineering" can improve constant factors

- Linked list, array, or a hybrid
- Insert at end or beginning of list
- Splay-like: Always move item to front of list

Rigorous Separate Chaining Analysis

The **load factor**, λ , of a hash table is calculated as

$$\lambda = \frac{n}{TableSize}$$

where n is the number of items currently in the table

Under chaining, the average number of elements per bucket is ____

So if some inserts are followed by random finds, then on average:

- Each unsuccessful find compares against ____ items
- Each successful find compares against ____ items

How big should TableSize be??

Rigorous Separate Chaining Analysis

The **load factor**, λ , of a hash table is calculated as

$$\lambda = \frac{n}{TableSize}$$

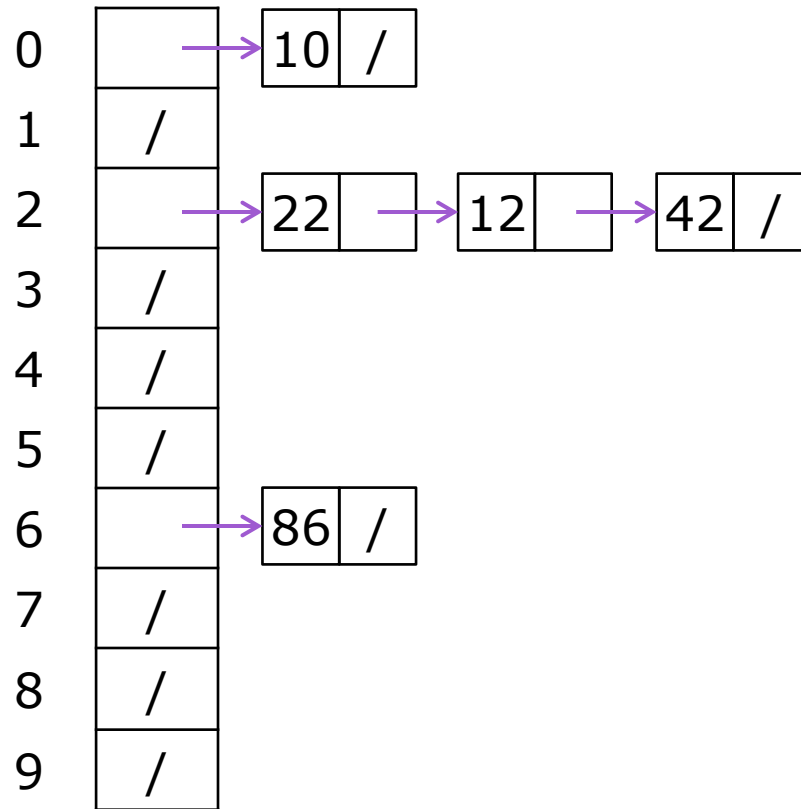
where n is the number of items currently in the table

Under chaining, the average number of elements per bucket is λ

So if some inserts are followed by random finds, then on average:

- Each unsuccessful find compares against λ items
- Each successful find compares against λ items
- If λ is low, find and insert likely to be $O(1)$
- We like to keep λ around 1 for separate chaining

Separate Chaining Deletion



Not too bad and quite easy

- Find in table
- Delete from bucket

Similar run-time as insert

- Sensitive to underlying bucket structure

Open Addressing: Linear Probing

Separate chaining does not use all the space in the table. Why not use it?

- Store directly in the array cell (no linked list or buckets)

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

How to deal with collisions?

If $h(\text{key})$ is already full,

try $(h(\text{key}) + 1) \% \text{TableSize}$. If full,

try $(h(\text{key}) + 2) \% \text{TableSize}$. If full,

try $(h(\text{key}) + 3) \% \text{TableSize}$. If full...

Example: insert 38, 19, 8, 79, 10

Open Addressing: Linear Probing

0	
1	
2	
3	
4	
5	
6	
7	
8	38
9	

Separate chaining does not use all the space in the table. Why not use it?

- Store directly in the array cell (no linked list or buckets)

How to deal with collisions?

If $h(\text{key})$ is already full,

try $(h(\text{key}) + 1) \% \text{TableSize}$. If full,

try $(h(\text{key}) + 2) \% \text{TableSize}$. If full,

try $(h(\text{key}) + 3) \% \text{TableSize}$. If full...

Example: insert 38, 19, 8, 79, 10

Open Addressing: Linear Probing

Separate chaining does not use all the space in the table. Why not use it?

- Store directly in the array cell (no linked list or buckets)

0	
1	
2	
3	
4	
5	
6	
7	
8	38
9	19

How to deal with collisions?

If $h(\text{key})$ is already full,

try $(h(\text{key}) + 1) \% \text{TableSize}$. If full,

try $(h(\text{key}) + 2) \% \text{TableSize}$. If full,

try $(h(\text{key}) + 3) \% \text{TableSize}$. If full...

Example: insert 38, 19, 8, 79, 10

Open Addressing: Linear Probing

Separate chaining does not use all the space in the table. Why not use it?

- Store directly in the array cell (no linked list or buckets)

0	8
1	
2	
3	
4	
5	
6	
7	
8	38
9	19

How to deal with collisions?

If $h(\text{key})$ is already full,

try $(h(\text{key}) + 1) \% \text{TableSize}$. If full,

try $(h(\text{key}) + 2) \% \text{TableSize}$. If full,

try $(h(\text{key}) + 3) \% \text{TableSize}$. If full...

Example: insert 38, 19, 8, 79, 10

Open Addressing: Linear Probing

0	8
1	79
2	
3	
4	
5	
6	
7	
8	38
9	19

Separate chaining does not use all the space in the table. Why not use it?

- Store directly in the array cell (no linked list or buckets)

How to deal with collisions?

If $h(\text{key})$ is already full,

try $(h(\text{key}) + 1) \% \text{TableSize}$. If full,

try $(h(\text{key}) + 2) \% \text{TableSize}$. If full,

try $(h(\text{key}) + 3) \% \text{TableSize}$. If full...

Example: insert 38, 19, 8, 79, 10

Open Addressing: Linear Probing

Separate chaining does not use all the space in the table. Why not use it?

- Store directly in the array cell (no linked list or buckets)

0	8
1	79
2	10
3	
4	
5	
6	
7	
8	38
9	19

How to deal with collisions?

If $h(\text{key})$ is already full,

try $(h(\text{key}) + 1) \% \text{TableSize}$. If full,

try $(h(\text{key}) + 2) \% \text{TableSize}$. If full,

try $(h(\text{key}) + 3) \% \text{TableSize}$. If full...

Example: insert 38, 19, 8, 79, 10

Open Addressing

This is one example of open addressing

Open addressing means resolving collisions by trying a sequence of other positions in the table

Trying the next spot is called probing

- We just did linear probing
 $h(\text{key}) + i) \% \text{TableSize}$
- In general have some probe function f and use
 $h(\text{key}) + f(i) \% \text{TableSize}$

Open addressing does poorly with high load factor λ

- So we want larger tables
- Too many probes means we lose our $O(1)$

Open Addressing: Other Operations

insert finds an open table position using a probe function

What about **find**?

- Must use same probe function to "retrace the trail" for the data
- Unsuccessful search when reach empty position

What about **delete**?

- Must use "lazy" deletion. Why?

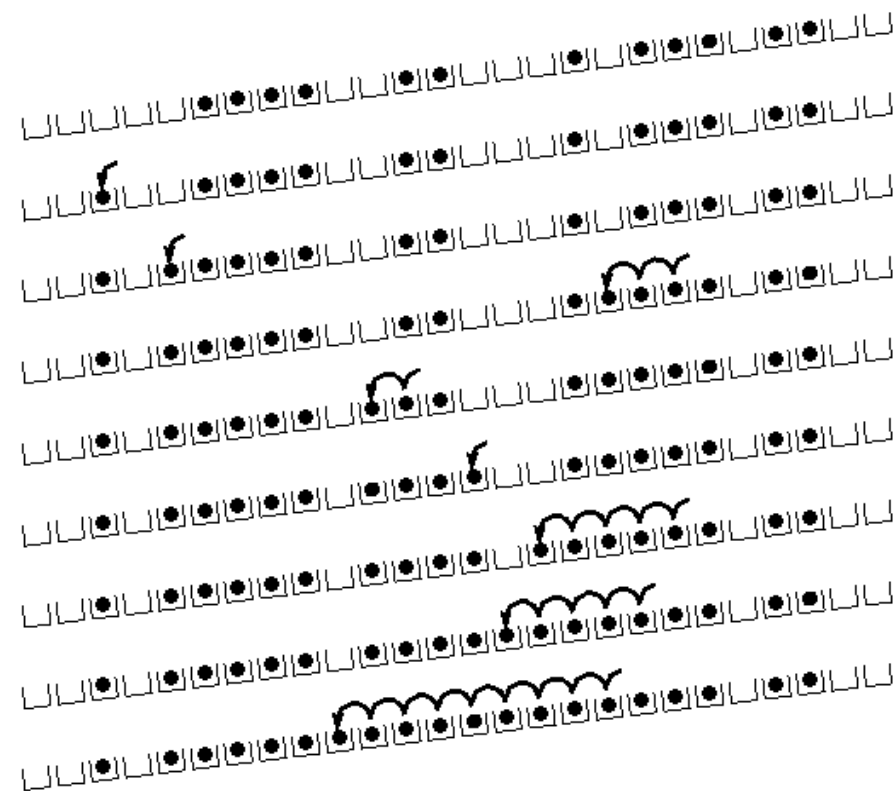
10	×	/	23	/	/	16	×	26
----	---	---	----	---	---	----	---	----

- Marker indicates "data was here, keep on probing"

Primary Clustering

It turns out linear probing is a bad idea, even though the probe function is quick to compute (which is a good thing)

- This tends to produce clusters, which lead to long probe sequences
- This is called *primary clustering*
- We saw the start of a cluster in our linear probing example



[R. Sedgewick]

Analysis of Linear Probing

Trivial fact:

For any $\lambda < 1$, linear probing will find an empty slot

- We are safe from an infinite loop unless table is full

Non-trivial facts (we won't prove these):

Average # of probes given load factor λ

- For an unsuccessful search as $\text{TableSize} \rightarrow \infty$:

$$\frac{1}{2} \left(1 + \frac{1}{(1 - \lambda)^2} \right)$$

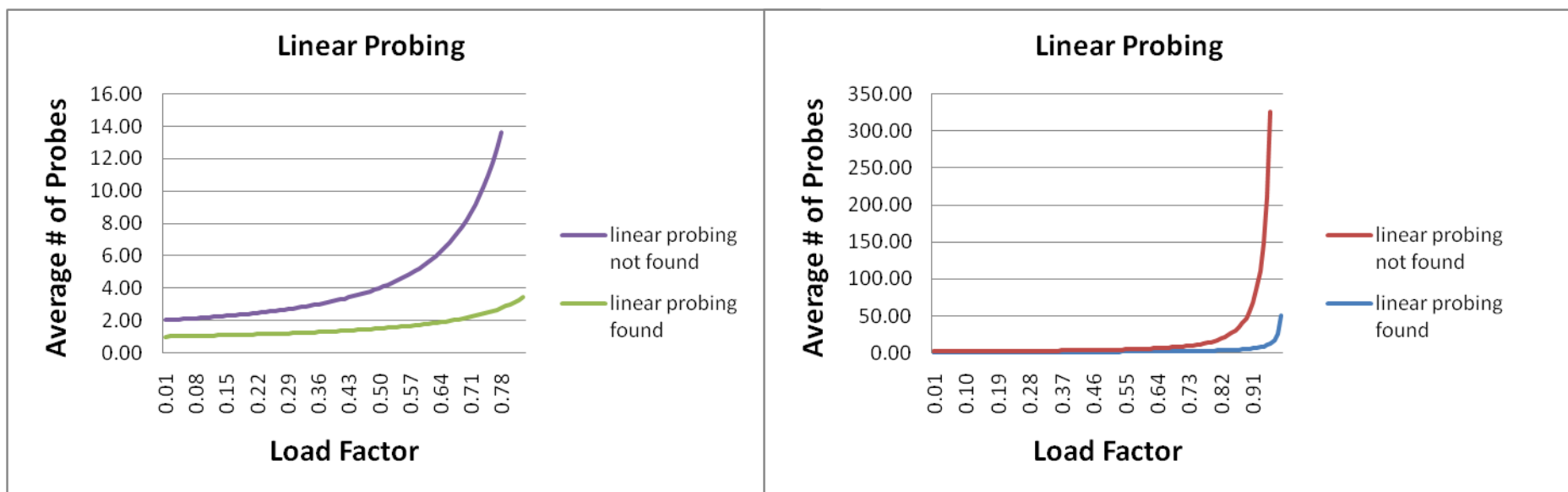
- For an successful search as $\text{TableSize} \rightarrow \infty$:

$$\frac{1}{2} \left(1 + \frac{1}{(1 - \lambda)} \right)$$

Analysis in Chart Form

Linear-probing performance degrades rapidly as the table gets full

- The Formula does assumes a "large table" but the point remains



Note that separate chaining performance is linear in λ and has no trouble with $\lambda > 1$

Open Addressing: Quadratic Probing

We can avoid primary clustering by changing the probe function from just i to $f(i)$

$$(h(\text{key}) + f(i)) \% \text{TableSize}$$

For **quadratic probing**, $f(i) = i^2$:

0th probe: $(h(\text{key}) + 0) \% \text{TableSize}$

1st probe: $(h(\text{key}) + 1) \% \text{TableSize}$

2nd probe: $(h(\text{key}) + 4) \% \text{TableSize}$

3rd probe: $(h(\text{key}) + 9) \% \text{TableSize}$

...

i^{th} probe: $(h(\text{key}) + i^2) \% \text{TableSize}$

Intuition: Probes quickly "leave the neighborhood"

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

TableSize = 10
insert(89)

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

TableSize = 10

insert(89)

insert(18)

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

TableSize = 10

insert(89)

insert(18)

insert(49)

Quadratic Probing Example

0	49
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

TableSize = 10

insert(89)

insert(18)

insert(49)

$49 \% 10 = 9$ collision!

$(49 + 1) \% 10 = 0$

insert(58)

Quadratic Probing Example

0	49
1	
2	58
3	
4	
5	
6	
7	
8	18
9	89

TableSize = 10

insert(89)

insert(18)

insert(49)

insert(58)

$58 \% 10 = 8$ collision!

$(58 + 1) \% 10 = 9$ collision!

$(58 + 4) \% 10 = 2$

insert(79)

Quadratic Probing Example

0	49
1	
2	58
3	79
4	
5	
6	
7	
8	18
9	89

TableSize = 10

insert(89)

insert(18)

insert(49)

insert(58)

insert(79)

$79 \% 10 = 9$ collision!

$(79 + 1) \% 10 = 0$ collision!

$(79 + 4) \% 10 = 3$

Another Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	

TableSize = 7

Insert:

76 ($76 \% 7 = 6$)

40 ($40 \% 7 = 5$)

48 ($48 \% 7 = 6$)

5 ($5 \% 7 = 5$)

55 ($55 \% 7 = 6$)

47 ($47 \% 7 = 5$)

Another Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	76

TableSize = 7

Insert:

76 ($76 \% 7 = 6$)

40 ($40 \% 7 = 5$)

48 ($48 \% 7 = 6$)

5 ($5 \% 7 = 5$)

55 ($55 \% 7 = 6$)

47 ($47 \% 7 = 5$)

Another Quadratic Probing Example

0	
1	
2	
3	
4	
5	40
6	76

TableSize = 7

Insert:

76 ($76 \% 7 = 6$)

40 ($40 \% 7 = 5$)

48 ($48 \% 7 = 6$)

5 ($5 \% 7 = 5$)

55 ($55 \% 7 = 6$)

47 ($47 \% 7 = 5$)

Another Quadratic Probing Example

0	48
1	
2	
3	
4	
5	40
6	76

TableSize = 7

Insert:

76 ($76 \% 7 = 6$)

40 ($40 \% 7 = 5$)

48 ($48 \% 7 = 6$)

5 ($5 \% 7 = 5$)

55 ($55 \% 7 = 6$)

47 ($47 \% 7 = 5$)

Another Quadratic Probing Example

0	48
1	
2	5
3	
4	
5	40
6	76

TableSize = 7

Insert:

76 ($76 \% 7 = 6$)

40 ($40 \% 7 = 5$)

48 ($48 \% 7 = 6$)

5 ($5 \% 7 = 5$)

55 ($55 \% 7 = 6$)

47 ($47 \% 7 = 5$)

Another Quadratic Probing Example

0	48
1	
2	5
3	55
4	
5	40
6	76

TableSize = 7

Insert:

76 ($76 \% 7 = 6$)

40 ($40 \% 7 = 5$)

48 ($48 \% 7 = 6$)

5 ($5 \% 7 = 5$)

55 ($55 \% 7 = 6$)

47 ($47 \% 7 = 5$)

Another Quadratic Probing Example

0	48
1	
2	5
3	55
4	
5	40
6	76

**Will we ever get
a 1 or 4?!?**

TableSize = 7

Insert:

76 ($76 \% 7 = 6$)

40 ($40 \% 7 = 5$)

48 ($48 \% 7 = 6$)

5 ($5 \% 7 = 5$)

55 ($55 \% 7 = 6$)

47 ($47 \% 7 = 5$)

$(47 + 1) \% 7 = 6$ collision!

$(47 + 4) \% 7 = 2$ collision!

$(47 + 9) \% 7 = 0$ collision!

$(47 + 16) \% 7 = 0$ collision!

$(47 + 25) \% 7 = 2$ collision!

Another Quadratic Probing Example

0	48
1	
2	5
3	55
4	
5	40
6	76

insert(47) will always fail here. Why?

For all n , $(5 + n^2) \% 7$ is 0, 2, 5, or 6

Proof uses induction and

$$(5 + n^2) \% 7 = (5 + (n - 7)^2) \% 7$$

In fact, for all c and k ,

$$(c + n^2) \% k = (c + (n - k)^2) \% k$$

From Bad News to Good News

After TableSize quadratic probes, we cycle through the same indices

The good news:

- For prime T and $0 \leq i, j \leq T/2$ where $i \neq j$,
 $(h(\text{key}) + i^2) \% T \neq (h(\text{key}) + j^2) \% T$
- If TableSize is prime and $\lambda < 1/2$, quadratic probing will find an empty slot in at most TableSize/2 probes
- If you keep $\lambda < 1/2$, no need to detect cycles as we just saw

Clustering Reconsidered

Quadratic probing does not suffer from primary clustering as the quadratic nature quickly escapes the neighborhood

But it is no help if keys initially hash the same index

- Any 2 keys that hash to the same value will have the same series of moves after that
- Called *secondary clustering*

We can avoid secondary clustering with a probe function that depends on the key: *double hashing*

Open Addressing: Double Hashing

Idea:

Given two good hash functions h and g , it is very unlikely that for some key, $h(\text{key}) == g(\text{key})$

Ergo, why not probe using $g(\text{key})$?

For **double hashing**, $f(i) = i \cdot g(\text{key})$:

0th probe: $(h(\text{key}) + 0 \cdot g(\text{key})) \% \text{TableSize}$

1st probe: $(h(\text{key}) + 1 \cdot g(\text{key})) \% \text{TableSize}$

2nd probe: $(h(\text{key}) + 2 \cdot g(\text{key})) \% \text{TableSize}$

...

i^{th} probe: $(h(\text{key}) + i \cdot g(\text{key})) \% \text{TableSize}$

Crucial Detail:

We must make sure that $g(\text{key})$ cannot be 0

Double Hashing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

$T = 10$ (TableSize)

Hash Functions:

$$h(\text{key}) = \text{key} \bmod T$$

$$g(\text{key}) = 1 + ((\text{key}/T) \bmod (T-1))$$

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13

28

33

147

43

Double Hashing

0	
1	
2	
3	13
4	
5	
6	
7	
8	
9	

$T = 10$ (TableSize)

Hash Functions:

$$h(\text{key}) = \text{key} \bmod T$$

$$g(\text{key}) = 1 + ((\text{key}/T) \bmod (T-1))$$

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13

28

33

147

43

Double Hashing

0	
1	
2	
3	13
4	
5	
6	
7	
8	28
9	

$T = 10$ (TableSize)

Hash Functions:

$$h(\text{key}) = \text{key} \bmod T$$

$$g(\text{key}) = 1 + ((\text{key}/T) \bmod (T-1))$$

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13

28

33

147

43

Double Hashing

0	
1	
2	
3	13
4	
5	
6	
7	33
8	28
9	

$T = 10$ (TableSize)

Hash Functions:

$$h(\text{key}) = \text{key} \bmod T$$

$$g(\text{key}) = 1 + ((\text{key}/T) \bmod (T-1))$$

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13

28

$33 \rightarrow g(33) = 1 + 3 \bmod 9 = 4$

147

43

Double Hashing

0	
1	
2	
3	13
4	
5	
6	
7	33
8	28
9	147

$T = 10$ (TableSize)

Hash Functions:

$$h(\text{key}) = \text{key} \bmod T$$

$$g(\text{key}) = 1 + ((\text{key}/T) \bmod (T-1))$$

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13

28

33

147 $\rightarrow g(147) = 1 + 14 \bmod 9 = 6$

43

Double Hashing

0	
1	
2	
3	13
4	
5	
6	
7	33
8	28
9	147

$T = 10$ (TableSize)

Hash Functions:

$$h(\text{key}) = \text{key} \bmod T$$

$$g(\text{key}) = 1 + ((\text{key}/T) \bmod (T-1))$$

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13

28

33

147 $\rightarrow g(147) = 1 + 14 \bmod 9 = 6$

43 $\rightarrow g(43) = 1 + 4 \bmod 9 = 5$

We have a problem:

$$3 + 0 = 3 \quad 3 + 5 = 8$$

$$3 + 15 = 18$$

$$3 + 10 = 13$$

$$3 + 20 = 23$$

Double Hashing Analysis

Because each probe is "jumping" by $g(\text{key})$ each time, we should ideally "leave the neighborhood" and "go different places from the same initial collision"

But, as in quadratic probing, we could still have a problem where we are not "safe" due to an infinite loop despite room in table

This cannot happen in at least one case:

For primes p and q such that $2 < q < p$

$$h(\text{key}) = \text{key} \% p$$

$$g(\text{key}) = q - (\text{key} \% q)$$

Summarizing Collision Resolution

Separate Chaining is easy

- find, delete proportional to load factor on average
- insert can be constant if just push on front of list

Open addressing uses probing, has clustering issues as it gets full but still has reasons for its use:

- Easier data representation
- Less memory allocation
- Run-time overhead for list nodes (but an array implementation could be faster)

When you make hash from hash leftovers...

REHASHING

Rehashing

As with array-based stacks/queues/lists

- If table gets too full, create a bigger table and copy everything
- Less helpful to shrink a table that is underfull

With chaining, we get to decide what "too full" means

- Keep load factor reasonable (e.g., < 1)?
- Consider average or max size of non-empty chains

For open addressing, half-full is a good rule of thumb

Rehashing

What size should we choose?

- Twice-as-big?
- Except that won't be prime!

We go twice-as-big but guarantee prime

- Implement by hard coding a list of prime numbers
- You probably will not grow more than 20-30 times and can then calculate after that if necessary

Rehashing

Can we copy all data to the same indices in the new table?

- Will not work; we calculated the index based on TableSize

Rehash Algorithm:

Go through old table

Do standard insert for each item into new table

Resize is an $O(n)$ operation,

- Iterate over old table: $O(n)$
- n inserts / calls to the hash function: $n \cdot O(1) = O(n)$

Is there some way to avoid all those hash function calls?

- Space/time tradeoff: Could store $h(\text{key})$ with each data item
- Growing the table is still $O(n)$; only helps by a constant factor

Reality is never as clean-cut as theory

IMPLEMENTING HASHING

Hashing and Comparing

Our use of int key can lead to us overlooking a critical detail

- We do perform the initial hash on E
- While chaining/probing, we compare to E which requires equality testing (compare == 0)

A hash table needs a hash function and a comparator

- In Project 2, you will use two function objects
- The Java library uses a more object-oriented approach: each object has an equals method and a hashCode method:

```
class Object {  
    boolean equals(Object o) {...}  
    int hashCode() {...}  
    ...  
}
```

Equal Objects Must Hash the Same

The Java library (and your project hash table) make a very important assumption that clients must satisfy

Object-oriented way of saying it:

If `a.equals(b)`, then we must require
`a.hashCode() == b.hashCode()`

Function object way of saying it:

If `c.compare(a,b) == 0`, then we must require
`h.hash(a) == h.hash(b)`

If you ever override equals

- You need to override hashCode also in a consistent way
- See CoreJava book, Chapter 5 for other "gotchas" with equals

Comparable/Comparator Rules

We have not emphasized important "rules" about comparison for:

- all our dictionaries
- sorting (next major topic)

Comparison must impose a consistent, total ordering:

For all a , b , and c :

- If $\text{compare}(a,b) < 0$, then $\text{compare}(b,a) > 0$
- If $\text{compare}(a,b) == 0$, then $\text{compare}(b,a) == 0$
- If $\text{compare}(a,b) < 0$ and $\text{compare}(b,c) < 0$, then $\text{compare}(a,c) < 0$

A Generally Good hashCode()

```
int result = 17; // start at a prime
```

```
foreach field f
```

```
    int fieldHashCode =
```

```
        boolean: (f ? 1: 0)
```

```
        byte, char, short, int: (int) f
```

```
        long: (int) (f ^ (f >>> 32))
```

```
        float: Float.floatToIntBits(f)
```

```
        double: Double.doubleToLongBits(f), then above
```

```
        Object: object.hashCode( )
```

```
        result = 31 * result + fieldHashCode;
```

```
return result;
```



Final Word on Hashing

The hash table is one of the most important data structures

- Efficient find, insert, and delete
- Operations based on sorted order are not so efficient
- Useful in many, many real-world applications
- Popular topic for job interview questions

Important to use a good hash function

- Good distribution of key hashes
- Not overly expensive to calculate (bit shifts good!)

Important to keep hash table at a good size

- Keep TableSize a prime number
- Set a preferable λ depending on type of hashtable