CS250: Foundations of Computer Systems
[Data Structures for Storage]

Looking for something?
Trickle down from the root
Branching choices at the fork
As you descend

A left if it's less
A right if it's more

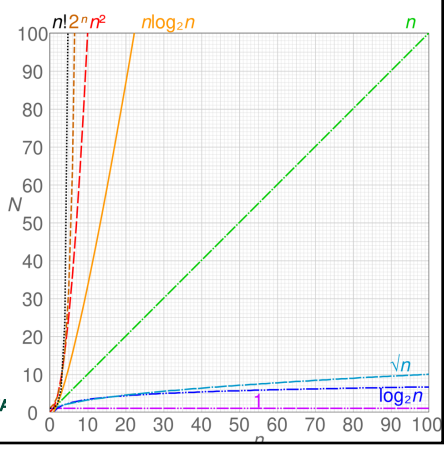
A dead end?
What you seek
Isn't here

Shrideep Pallickara
Computer Science
Colorado State University

1

Frequently asked questions from the previous class survey

- Why does printing some objects, only print their memory location? And not more details?
- Can binary search be incorrect?
- Why do companies require knowledge of data structures/algorithms/systems rather than general programming?
- Other common complexities beyond O(log n)?
  - O(1): Constant complexity
  - O(n): Linear complexity
  - O(nlogn): Loglinear complexity
  - $O(n^x)$: Polynomial complexity
  - $O(x^n)$: Exponential time
  - O(n!): Factorial complexity



2

Topics covered in this lecture

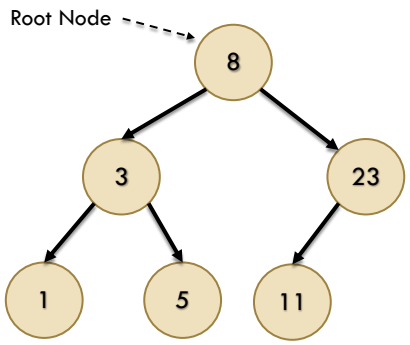
- Binary Search Trees
- B-Trees

3

4

Binary search trees

- Start at a single root node at the top of the tree
- **Branch** into multiple paths as they descend
- Allows programs to access the binary search tree through a single pointer
  - The location of its root node



5

Nodes and dispersion in memory

- A search tree's individual nodes can be **scattered** throughout memory
- Each node is only linked to its children and parents through?
  - The power and flexibility of **pointers**

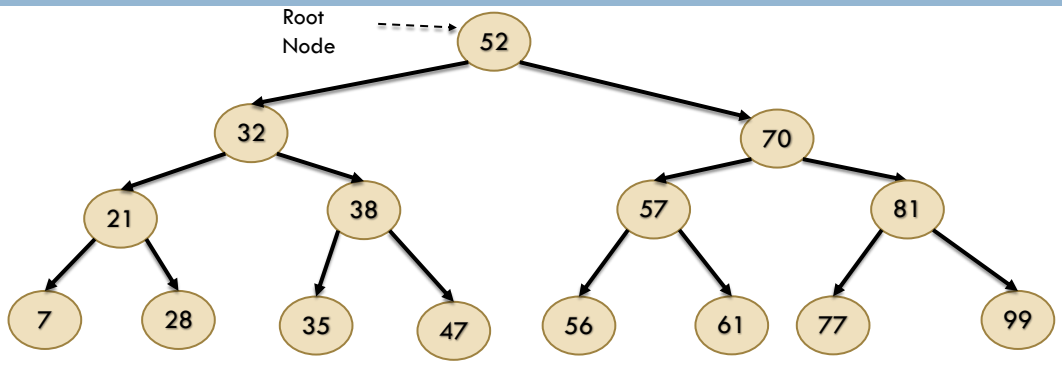
6

The power of the binary search tree stems from how values are organized within the tree.

- For any node N
  - The value of *any node* in N's **left** subtree is less than N's value
    - Values in the left node and all nodes below it are less than the value of the current node
  - The value of *any node* in N's **right** subtree is greater than N's value
    - Values in the right node and all nodes below it are greater than the value of the current node
- The rule defines the tree's structure below that node
  - Partitions the subtree into two subsets

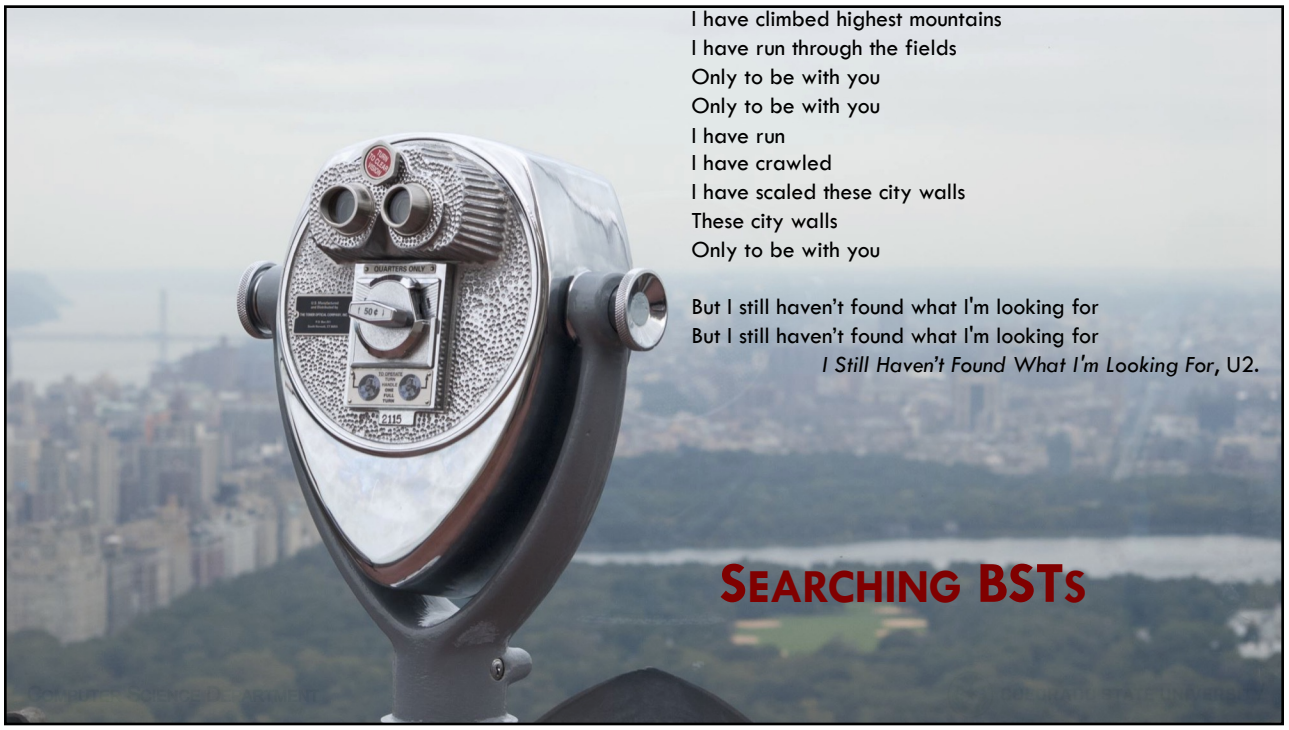
** Useful Rule: The number of right-handed people is greater than the number who are left-handed.

Values in a BST are ordered by the binary search property [1/2]



Values in a BST are ordered by the binary search property [2/2]

- This ordering of nodes might not seem like a lot of structure
  - Recall the power we got from using a similar property within binary search
- The binary search tree property is effectively keeping the data within the tree sorted with respect to its position in the tree
- As we will see, this allows us to not only efficiently **find values** in the tree but also efficiently **add and remove nodes**



Searching Binary Search Trees

- We search the BST by **walking down** from the root node
- At each step, we determine whether to explore the left or right subtree
  - By comparing the value at the current node with the target value
    - **Left**: If the target value is ***Less*** than the current value
    - **Right**: If the target value is *greate**R*** than the current value
- The search ends when either the target value is found, or it reaches a node with no children in the correct direction
  - In the latter case, we can assert that the target value is not in the tree

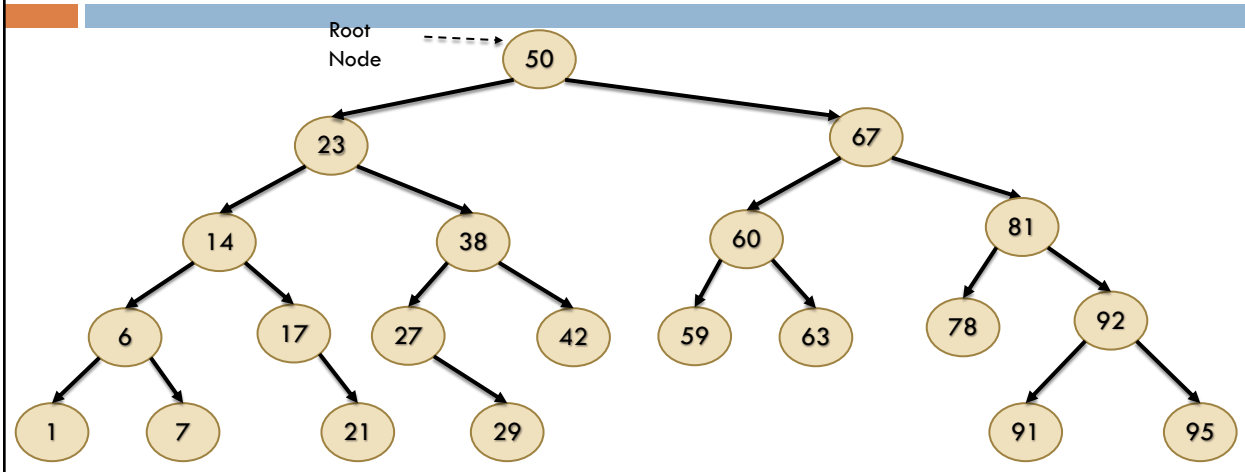
Searching Binary Search Trees: An analogy

- The node's value thus serves the same function as those helpful signs in hotels
- Signs that tell us rooms 500–519 are to the left
  - And rooms 520–545 are to the right
- With one quick check:
  - We can make the appropriate turn
    - And ignore the rooms in the other direction

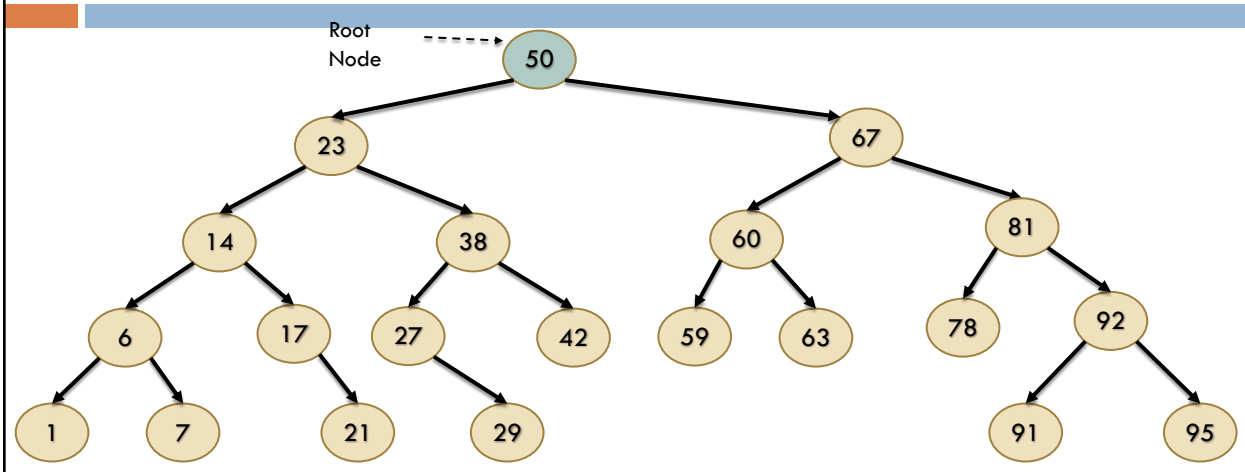
The recursive algorithm to find a value

```
FindValue(TreeNode: current, Type: target):
1 IF current == null:
      return null
2 IF current.value == target:
      return current
3 IF target < current.value AND current.left != null:
      return FindValue(current.left, target)
4 IF target > current.value AND current.right != null:
      return FindValue(current.right, target)
5 return null
```


Suppose we used this strategy to search for 63

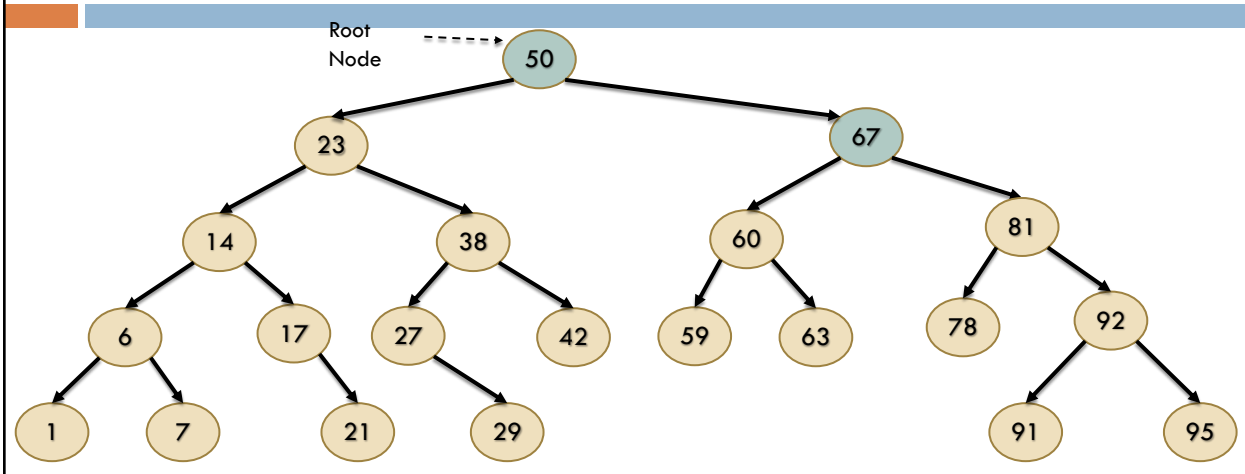


Suppose we used this strategy to search for 63

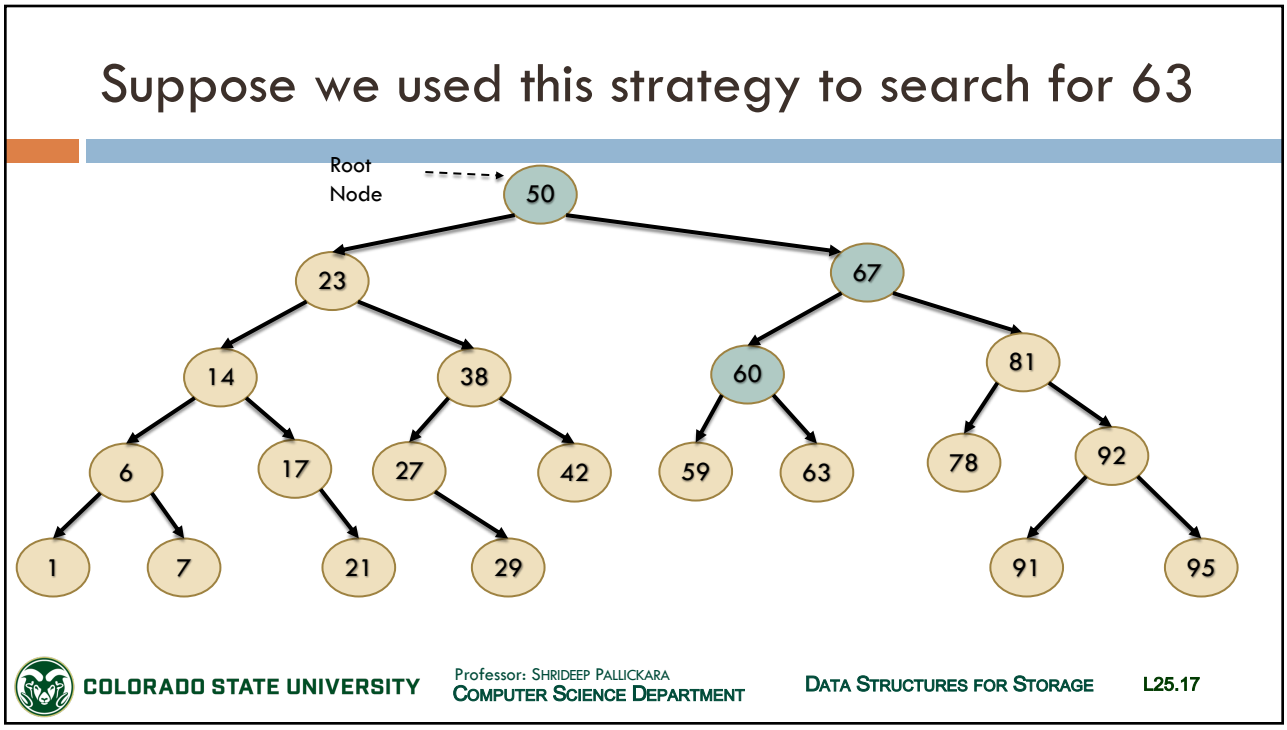


15

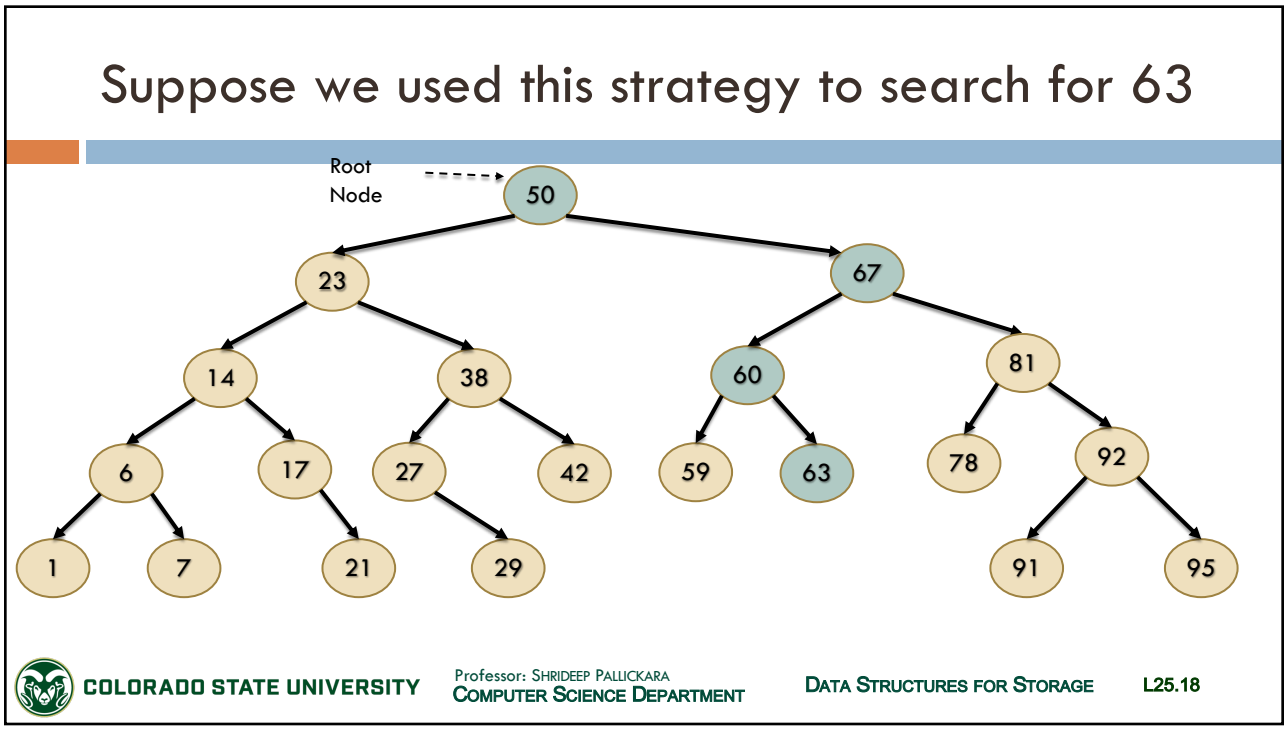
Suppose we used this strategy to search for 63



16


Suppose we used this strategy to search for 63
Root Node
50
23
67
14
38
60
81
6
17
27
42
59
63
78
92
1
7
21
29
91
95
COLORADO STATE UNIVERSITY
Professor: SHRIDEEP PALLICKARA
COMPUTER SCIENCE DEPARTMENT
DATA STRUCTURES FOR STORAGE
L25.17

17


Suppose we used this strategy to search for 63
Root Node
50
23
67
14
38
60
81
6
17
27
42
59
63
78
92
1
7
21
29
91
95
COLORADO STATE UNIVERSITY
Professor: SHRIDEEP PALLICKARA
COMPUTER SCIENCE DEPARTMENT
DATA STRUCTURES FOR STORAGE
L25.18

18

The iterative approach to binary search

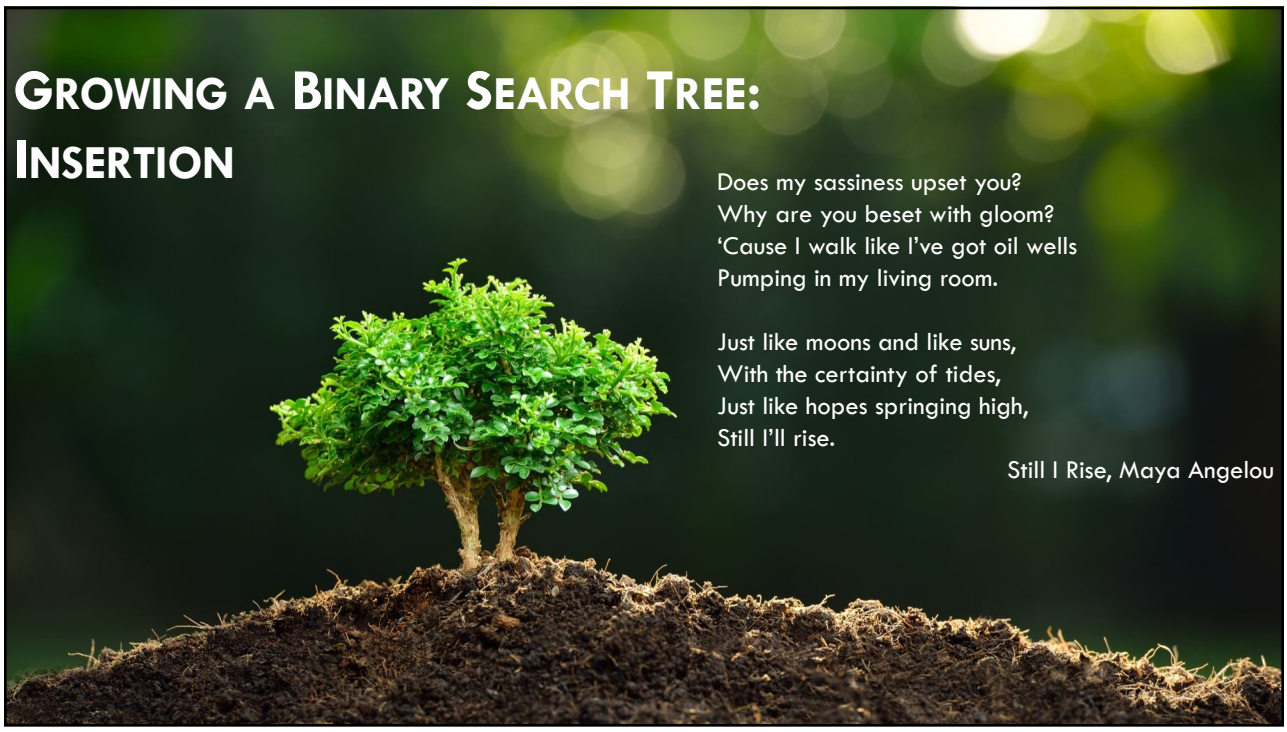
FindValueItr(TreeNode: root, Type: target):

1. TreeNode: current = root
2. WHILE current != null AND current.value != target:
3. IF target < current.value:
   current = current.left
   ELSE:
   current = current.right
4. return current

Simplifying the logic for using binary search trees

- We can wrap the entire tree in a **thin** data structure that contains the root node:

```
BinarySearchTree {
    TreeNode: root
}
```



21

Adding a node to a BST [1/2]

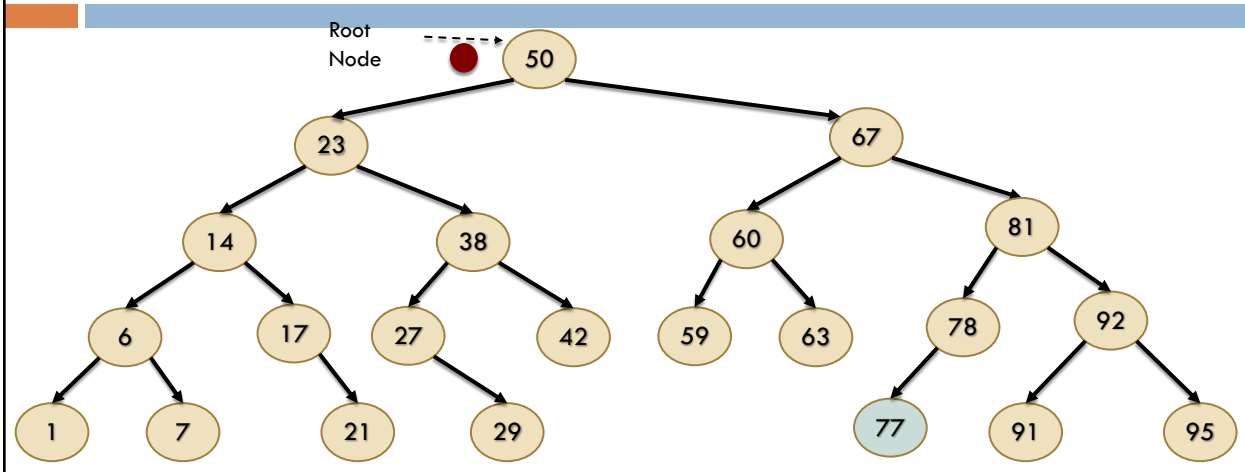
- We use the same basic algorithm to add values to a binary search tree as we do to search it
- Start at the root node, progress down the tree **as if searching for the new value**, and
  - Terminate once we hit a dead end:
    - Either a leaf node or an internal node with a single child

22

Adding a node to a BST [2/2]

- The primary **difference** between our search and insertion algorithms comes **after we hit the dead end**
- The insertion algorithm creates a new node as a child of the current node:
  - A left-hand child if the new value is less than that of the current node
  - A right-hand child if the new value is greater than that of the current node

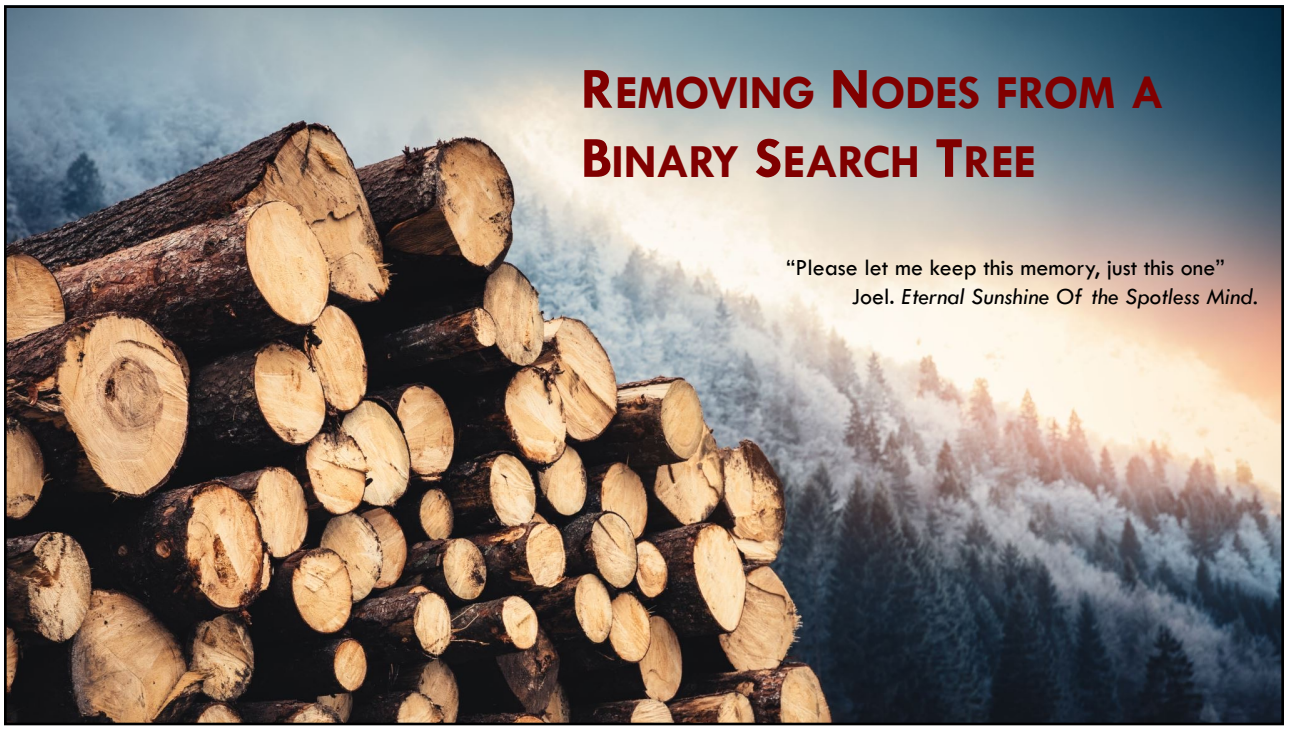
For example, if we want to add 77



Cost of inserting a node?

- Proportional to the **depth of the branch** along which we insert the new node

25

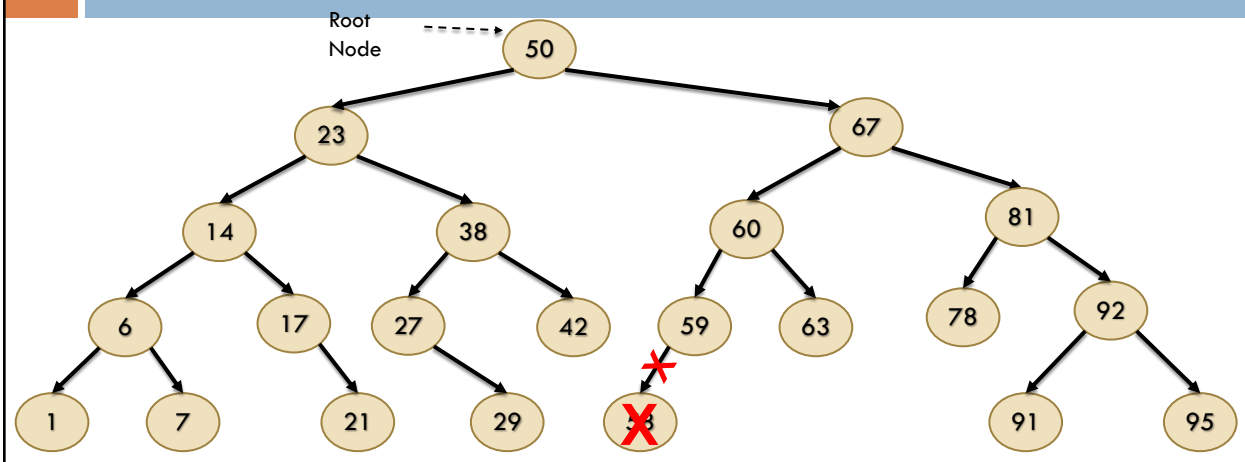


26

Removing Nodes is more complicated than adding them

- There are three cases of node removals to consider:
  1. Removing a leaf node (with no children)
  2. Removing an internal node with a single child
  3. Removing an internal node with two children

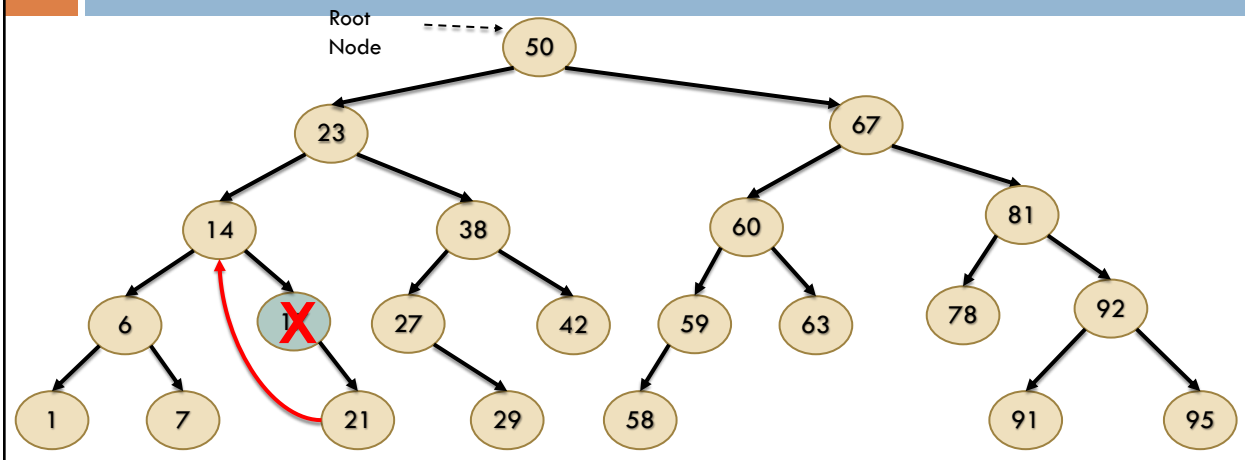
Remove a leaf node: Delete that node & update its parent's child pointer to reflect that it doesn't exist



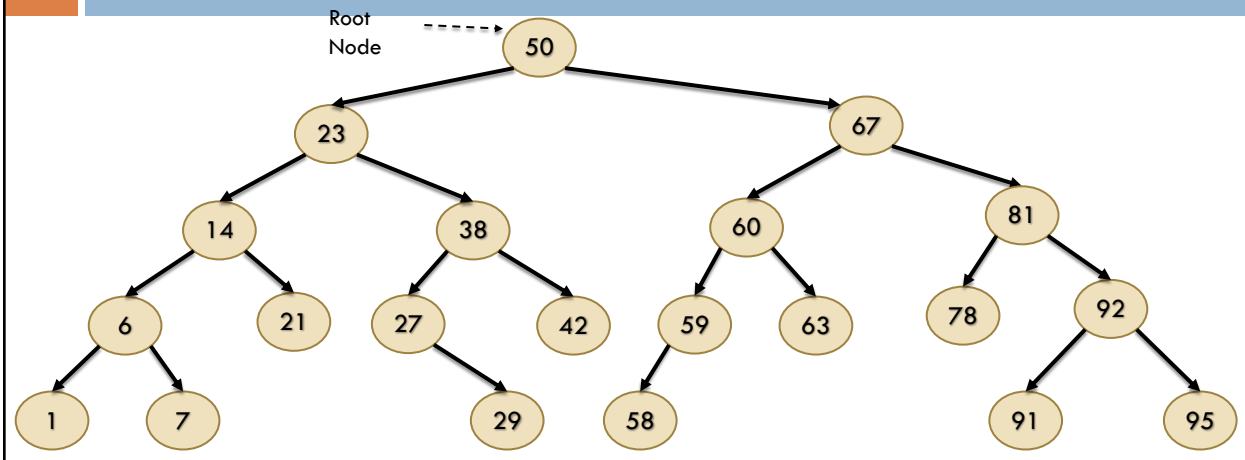
Removal of a leaf node

- Might make the parent node into a leaf
- The usefulness of parent pointers
  - Allows us to follow the parent pointer back to that node's parent, and set the corresponding child pointer to null
  - Storing this single piece of additional data makes deletion much simpler

Removal: If target node has a single child, **promote that child** to be child of deleted node's parent



Removal: If target node has a single child, **promote that child** to be child of deleted node's parent



31

What about deleting a node with two children [1/2]

- The complexity ramps up substantially
- No longer sufficient to just
  - Delete the node (leaf)
  - Shift a single child up (node with single child)

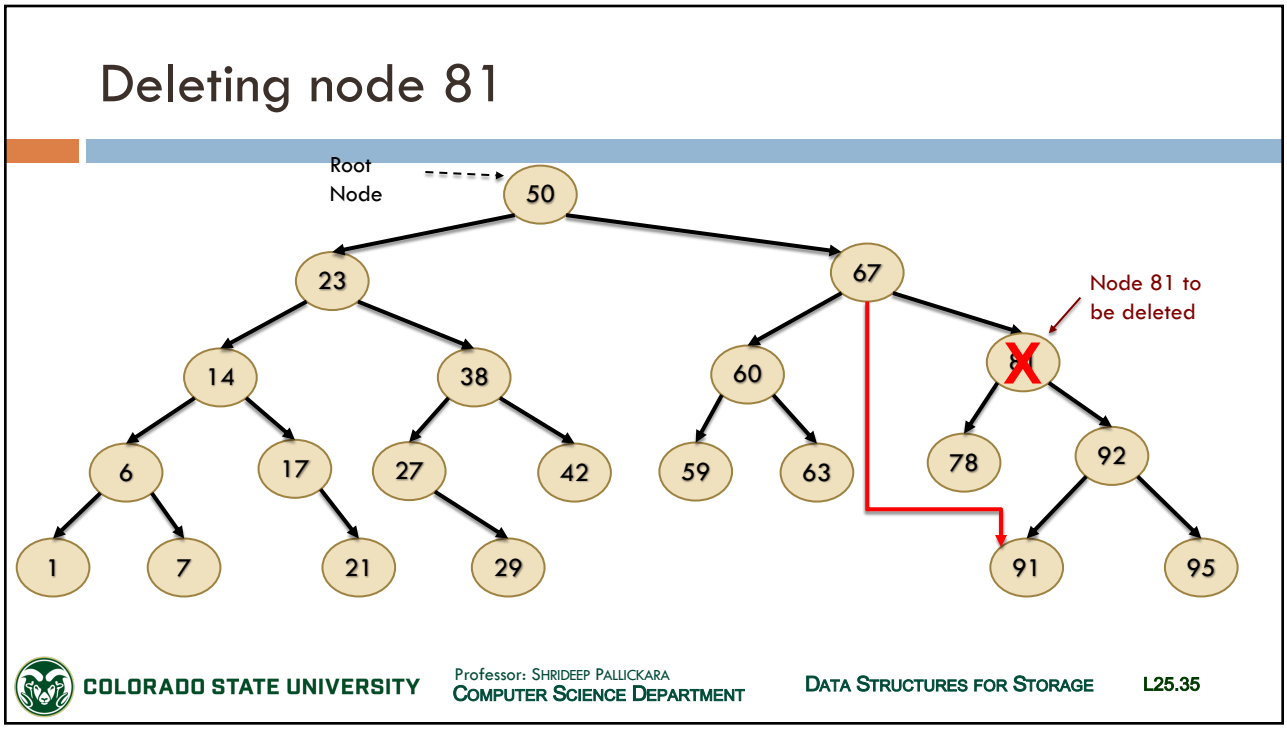
32

What about deleting a node with two children [2/2]

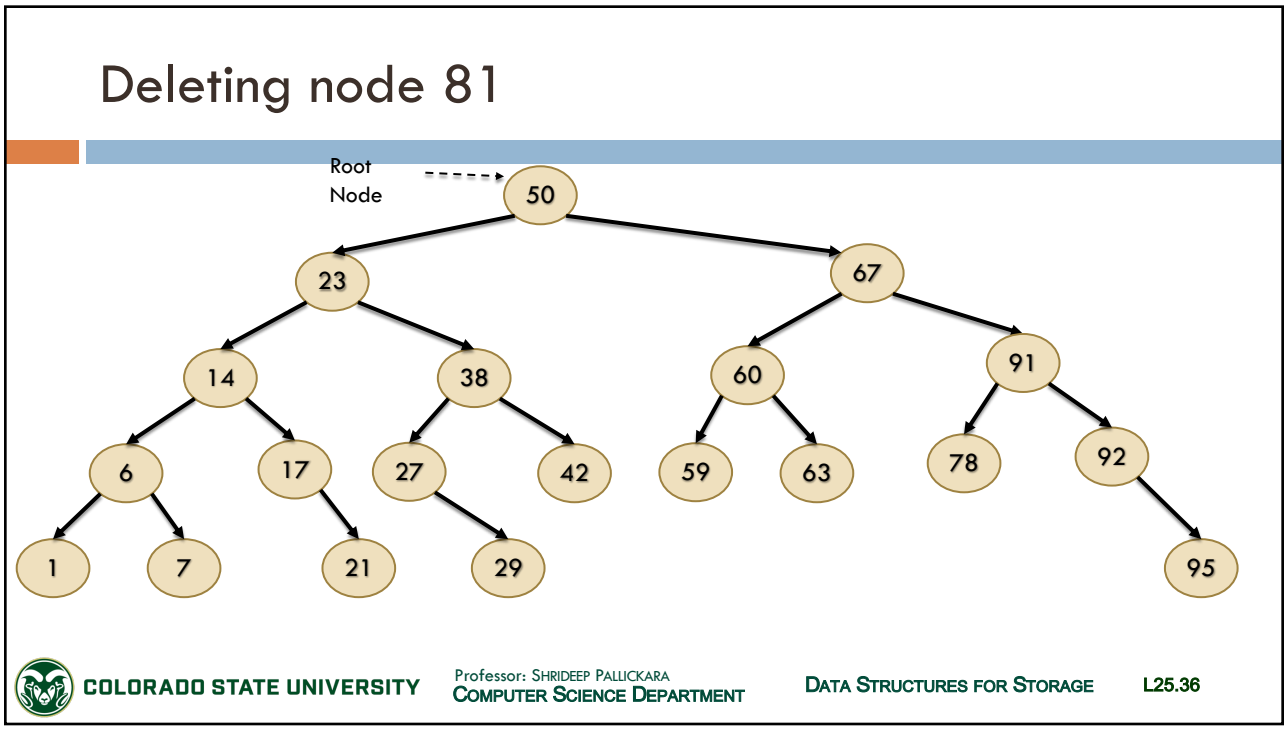
- Efficiently **find that node's successor**
- While this might seem like a daunting task, it isn't
- We can always find the successor **in the node's right-hand subtree**
  - Specifically, the successor will be the minimum (or leftmost) node in the right-hand subtree

BST: Predecessor and successor of a node

- Predecessor
  - Maximum value in its left subtree
- Successor
  - Minimum value in its right subtree


Deleting node 81
Root Node
50
23
67
14
38
60
81
6
17
27
42
59
63
78
92
1
7
21
29
91
95
Node 81 to be deleted
COLORADO STATE UNIVERSITY
Professor: SHRIDEEP PALLICKARA
COMPUTER SCIENCE DEPARTMENT
DATA STRUCTURES FOR STORAGE
L25.35

35


Deleting node 81
Root Node
50
23
67
14
38
60
91
6
17
27
42
59
63
78
92
1
7
21
29
95
COLORADO STATE UNIVERSITY
Professor: SHRIDEEP PALLICKARA
COMPUTER SCIENCE DEPARTMENT
DATA STRUCTURES FOR STORAGE
L25.36

36

Deletion of nodes with children in BST

- We looked at how you can splice out the successor
- You can also splice out the predecessor
  - This has the same complexity as splicing the successor: O(h) where h is the height of the tree
- For better empirical performance, some implementations
  - Alternate (or give equal priority to) splicing out the successor and predecessor

Predecessor → Maximum value in its left subtree
Successor → Minimum value in its right subtree

And the days, they linger on
And every night, what I'm waiting for
Is the real possibility that I may meet you in my dreams
Sometimes you're there and you're talking back to me
Come the morning I could swear you're next to me
And it's okay

Come Back; Eddie Vedder/Michael McCready; Pearl Jam

WHY DATA STRUCTURE MATTER? Using BSTs as an Exemplar

Data structures are critical because organization determines what operations are easy, hard, fast or slow

Structure	Search	Insert	Big Idea
Unsorted list	O(n)	O(1) append	Easy to add Slow to find
Sorted array	O(log n)	O(n)**	Fast search Expensive updates
BST (average)	O(log n)	O(log n)	Balances search and updates

- The **same data** can produce *very different costs* depending on how it is organized
- A data structure is a strategy for shaping work
- BSTs show us how representation and

** While the position for the new element can be found quickly using binary search in $O(\log n)$ time, the insertion itself requires shifting all subsequent elements to the right to maintain the sorted order, which takes $O(n)$ time.

What BSTs teach about computational thinking

- **Invariant**
  - *Correctness* comes from a single rule; all keys in the left subtree are smaller; all keys in the right subtree are greater
- **Decomposition**
  - A large search space problem becomes a smaller one by focusing on just one subtree
- **Recursive reasoning**
  - The same logic applies again and again at every node
- Trade-offs
  - Performance depends on the *shape*
  - Representation and efficiency must be designed together

Same data but two very different ways to waste your time [1/2]

- Assume a balanced BST, so search cost is about $\log_2 N$; with N=1,000,000,000, that is about 30 comparisons
  - While a linear scan may require up to 1,000,000,000 comparisons
- If each comparison takes 1 millisecond
  - Linear scan worst case: 1,000,000,000 ms
  - BST worst case: about 30 ms
- Converting those:
  - Linear scan: 1,000,000 seconds ≈ 16,667 minutes ≈ 277.8 hours ≈ 11.6 days
  - BST: 0.03 seconds

Same data but two very different ways to waste your time [2/2]

- Ratio:
  - Worst-case linear scan vs BST ≈ 1,000,000,000 / 30
  - About **33 million times** more work
- With the same data and the same 1 ms per operation,
  - One organization makes search feel like two weeks,
  - The other makes it feel like a blink
- That is the real lesson:
  - Data structures are not cosmetic
  - They decide whether computation is practical or absurd

Why interviews obsess over this stuff instead of "real software engineering"

- Because frameworks come and go, but **reasoning** does not
  - They are testing whether you can think clearly when the scaffolding is taken away
- Data structures reveal **how you shape cost**
  - Companies care less that you can write code, and more that you can avoid writing code that quietly takes eleven days
- These questions **compress signals**; a BST or hash table discussion reveals
  - Precision, problem decomposition, trade-off awareness, and debugging habits
- Software engineering is broader, but harder to sample quickly
  - Interviewers would love to assess years of judgment, teamwork, and design taste
  - A whiteboard is, alas, a very small stage

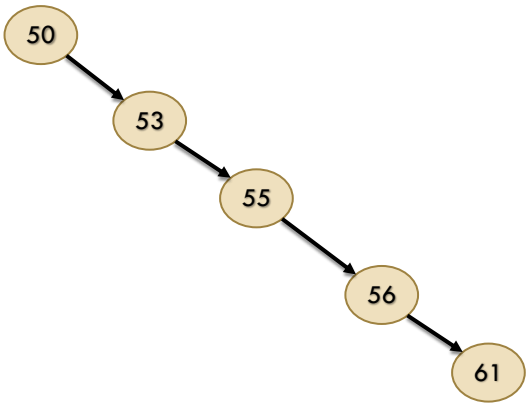
UNBALANCED BINARY SEARCH TREES

Life is like riding a bicycle. To keep your balance, you must keep moving.
Albert Einstein

The perils of unbalanced trees

- Binary search trees are still efficient as long as the trees are **mostly, if not perfectly, balanced**
- But if the tree becomes highly unbalanced?
  - Its depth could grow linearly with the number of elements
- In fact, in the *extreme case*, our splendid BST becomes nothing more than a **sorted linked list**
  - All the nodes have a single child in the same direction

Example of a BST devolving into a sorted linked list



Highly unbalanced trees can easily occur in many real-world applications

- Imagine storing coffee log in a BST indexed by timestamp
- Every time we drink a cup of coffee, we insert the relevant information into our tree
- Things go bad quickly
  - Due to the **monotonically increasing timestamps**, we insert every entry in sorted order
  - We end up creating a linked list using only the **right-hand child pointers**

Operations on an unbalanced tree can be extremely inefficient

- Consider a tree with N nodes
- If our tree is balanced?
  - Our operations take time logarithmic in N or O(log N)
- In the opposite case, where our tree is a list?
  - Our operations can take time linearly proportional to N or O(N)
- We can use red-black trees, 2-3 trees, and B-trees, to preserve balance
  - While undergoing dynamic insertions and deletions
  - The tradeoff? Increased complexity in the tree operations

PIVOTING TO BALANCE BSTS

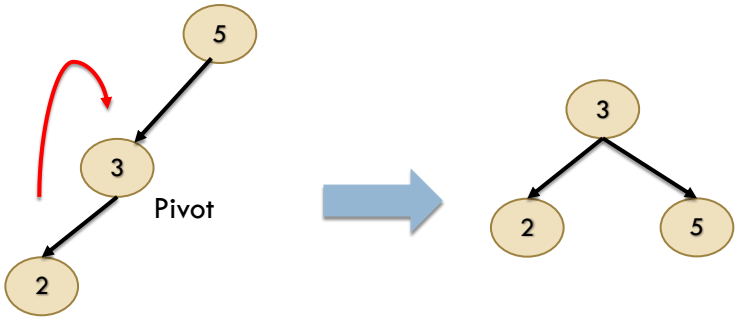
49

Balancing BSTs

- Perform a **rotation step** *after* nodes are added or removed
- If the insert operation leaves a branch unbalanced?
  - i.e., two consecutive nodes in the branch have only one child
  - **Rotate** nodes around the middle one

50

Balancing BSTs: Rotation



51

B-TREES

52

B-Trees

- B-Trees combine ideas such as:
  - Increase node fanout
  - Reduce tree height
  - Reduce the number of node pointers
  - Reduce frequency of balancing operations

The contents of this slide-set are based on the following references

- Jeremy Kubica. *Data Structures the Fun Way: An Amusing Adventure with Coffee-Filled Examples*. No Starch Press. ISBN-10. /13: 1718502605/978-1718502604. [Chapter 5]