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

Looking for something?

Trickle down from the root
Branching choices at the fork

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

Computer Science Department
Colorado State University

1

Frequently asked questions from the previous class survey

- Why don't we use Binary Search trees *everywhere*?

Colorado State University
Professor: Shrideep Pallickara
Computer Science Department
Data Structures for Storage
L25.2

2

Topics covered in this lecture

- Binary Search Trees
- B-Trees

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT DATA STRUCTURES FOR STORAGE L25.3

3

4

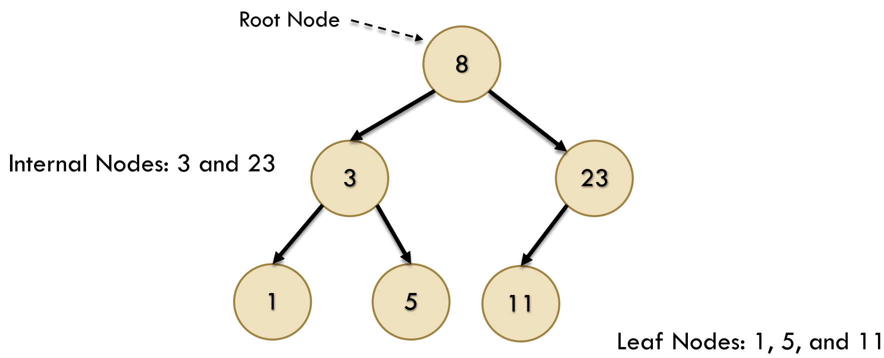
Binary Search Tree (BST)

- Binary search trees use the concepts underpinning the binary search algorithm to create a **dynamic data structure**
  - The key word here is dynamic
- Unlike sorted arrays, binary search trees support the efficient **addition and removal** of elements in addition to searches
  - Making them the perfect blend of the algorithmic efficiency of binary search and the adaptability of dynamic data structures

Tree Structures

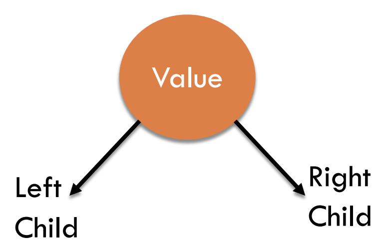
- Trees are **hierarchical** data structures
- Comprise **branching chains** of nodes
- Natural *extension* of linked lists
  - In the case of BSTs, each tree node is permitted two next pointers
    - Point to subsequent nodes in disjoint lists

Example BST



BST Nodes

- A node contains a **value** (of a given type)
  - Plus, **up to two** pointers to lower nodes in the tree
- Nodes with at least one child?
  - Internal nodes
- Nodes without any children?
  - **Leaf** nodes



Tree nodes may contain other information

- Often include a **pointer back to the node's parent**, for instance
  - Allows **bottom-up traversals** of the tree
    - In addition to the typical top down
  - Comes in handy when we consider removing nodes

The TreeNode data structure

```
TreeNode {
    Type: value
    TreeNode: left
    TreeNode: right
    TreeNode: parent
}
```

We might also want to store auxiliary data …

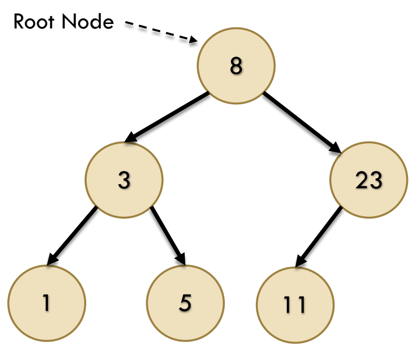
- Storing and searching for individual values are useful
- However, **using these values as keys** for looking up more detailed information greatly extends the power of the data structure
  - For e.g., coffee names as the node's values, allowing us to efficiently look up records for any coffee
    - Our auxiliary data would be a detailed record of that coffee

BST and auxiliary data

- The tree node data structure can either store this auxiliary data
  - Directly
  - Include a pointer to a composite data structure
    - Located somewhere else in memory

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



Nodes and dispersion in memory

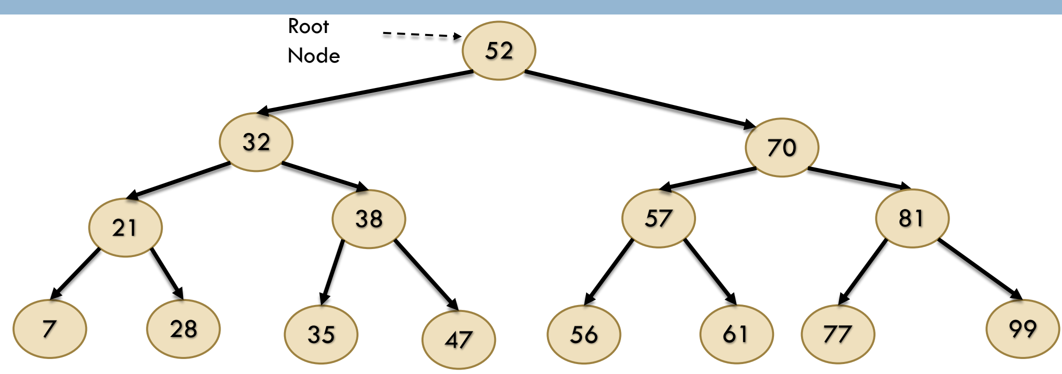
- A search tree's individual nodes can be **scattered** throughout memory
- Each node is only linked to its children and parents through?
  - The power and flexibility of **pointers**

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**

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT DATA STRUCTURES FOR STORAGE L25.17

17

18

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 *greater* 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

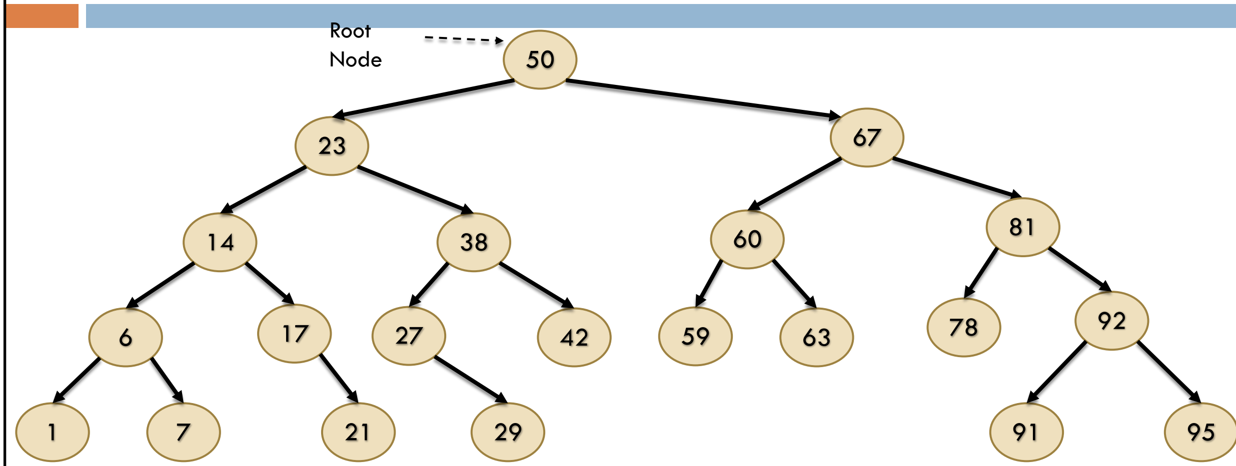
Searching Binary Search Trees: An analogy

- 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

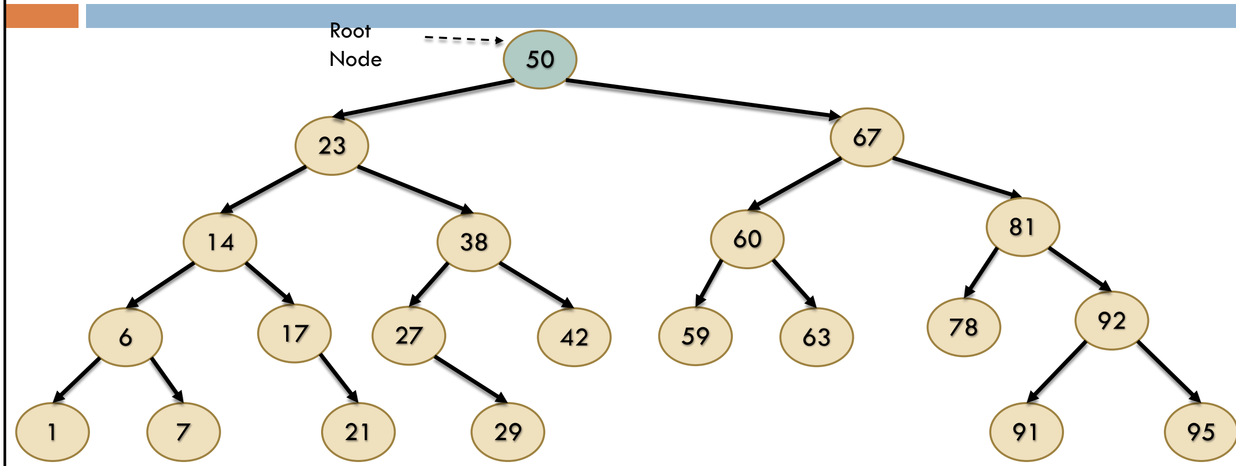
The recursive algorithm to find a value

```
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

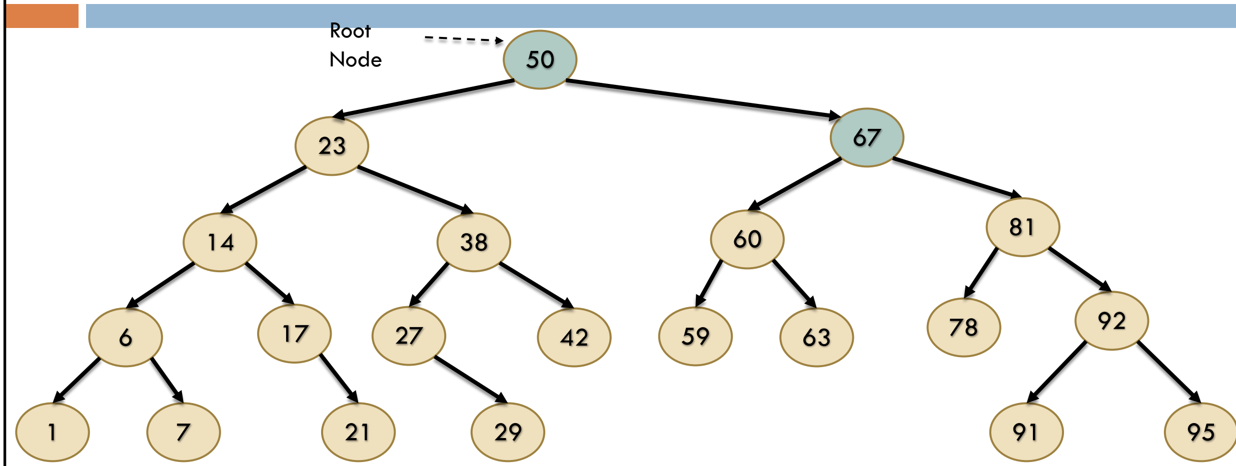


Suppose we used this strategy to search for 63



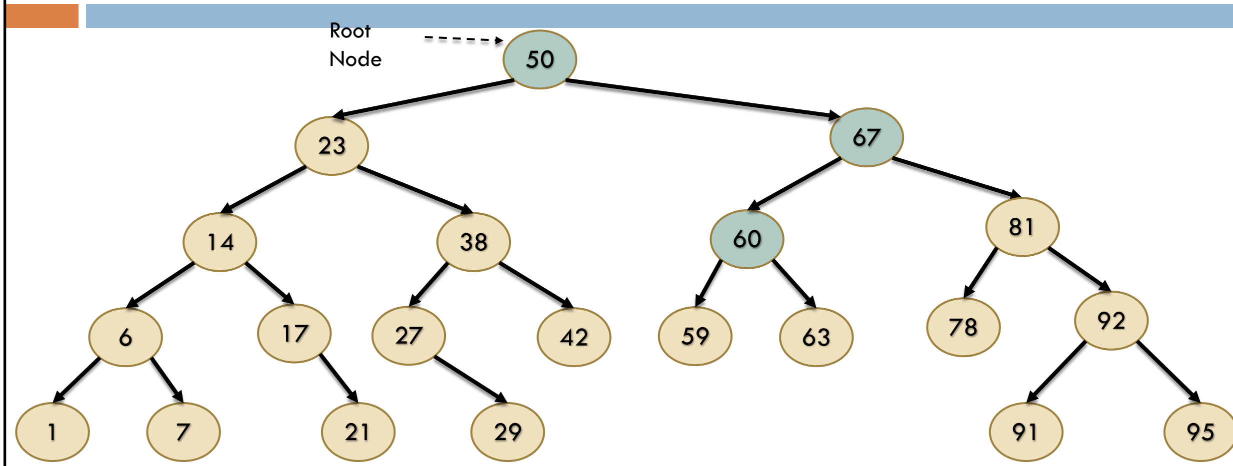
23

Suppose we used this strategy to search for 63



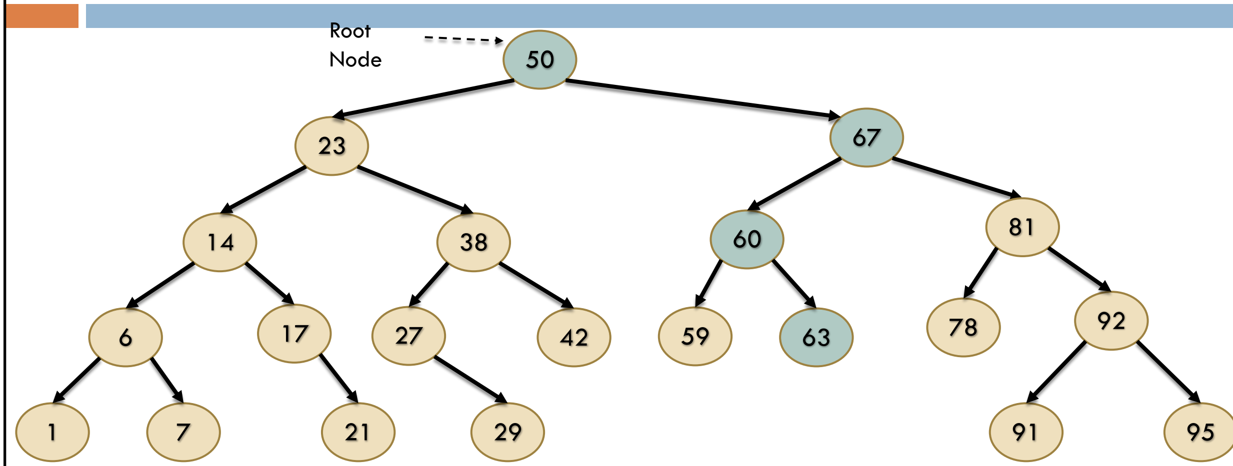
24

Suppose we used this strategy to search for 63



25

Suppose we used this strategy to search for 63



26

The iterative approach to binary search

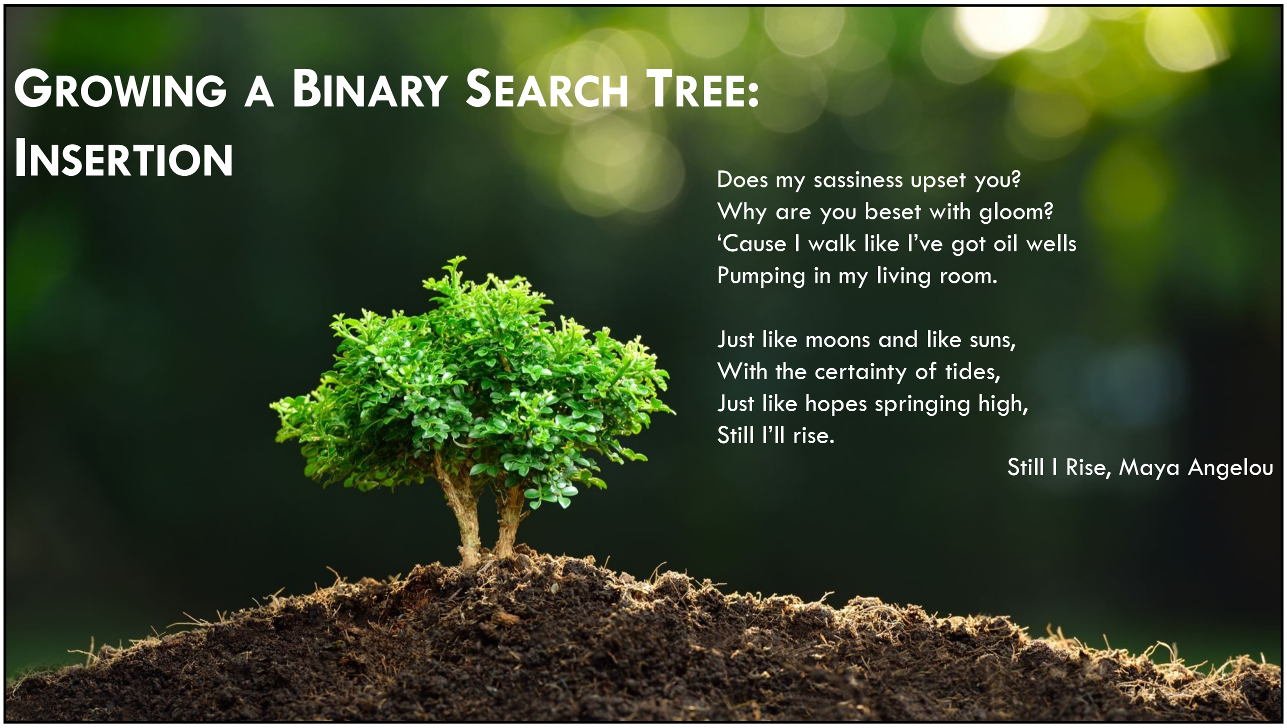
FindValueItr(TreeNode: root, Type: target):

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
}
```



29

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

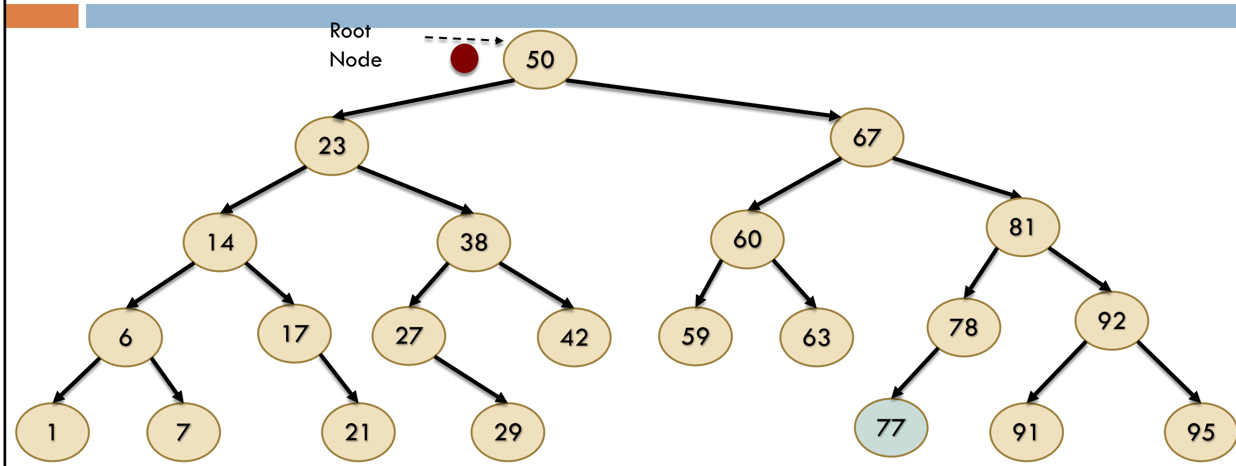
COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT DATA STRUCTURES FOR STORAGE L25.30

30

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

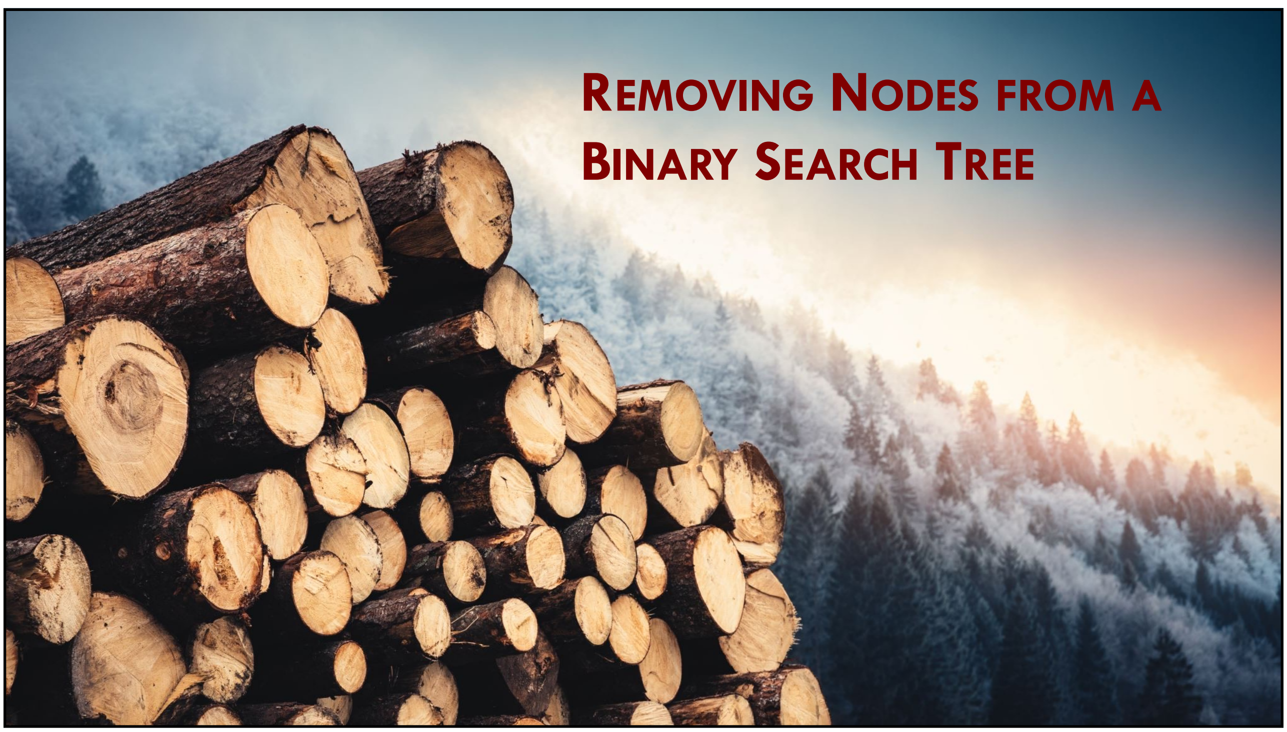
For example, if we want to add 77



Cost of inserting a node?

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

33

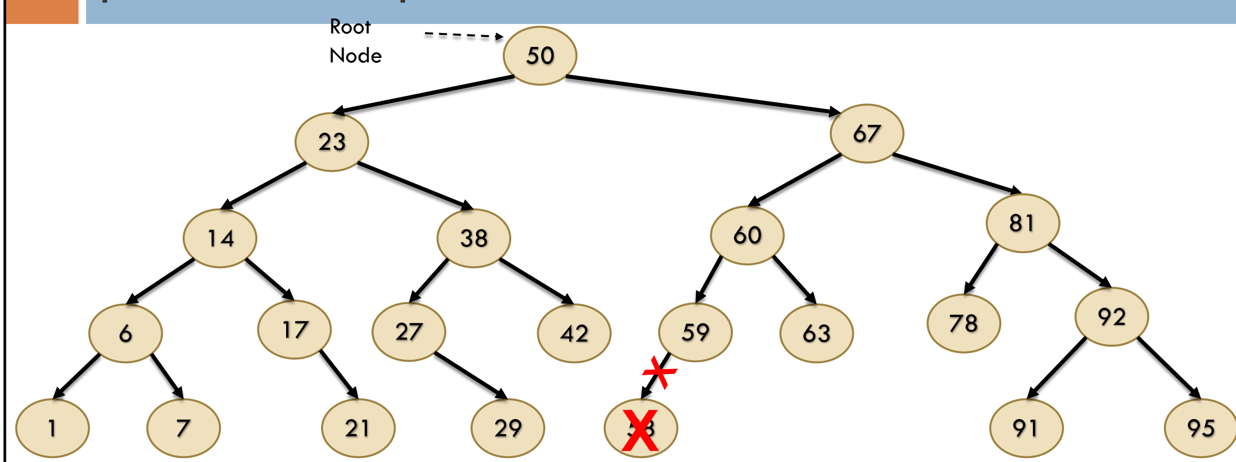


34

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

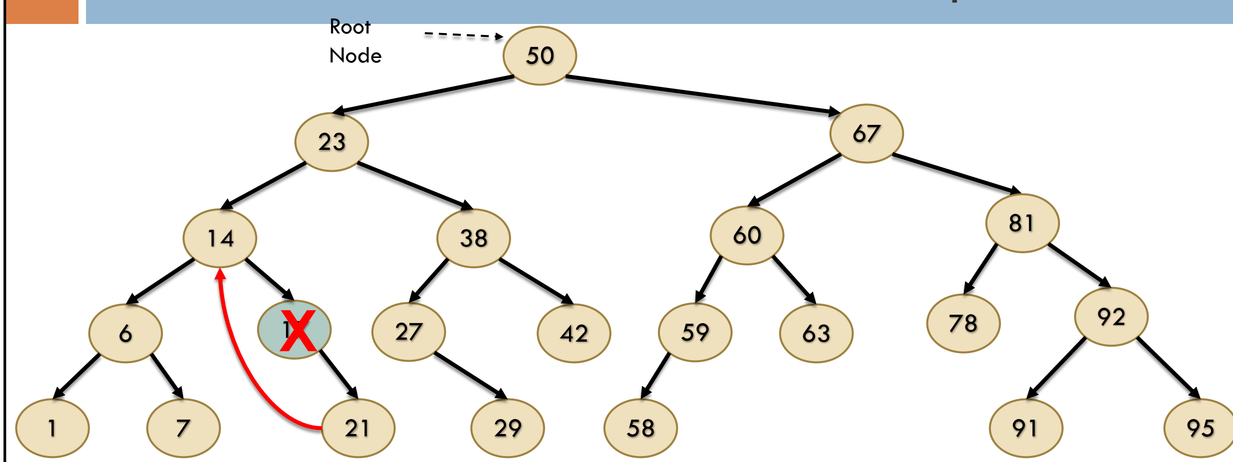
Remove a leaf node: Delete that node & update its parent's child pointer to reflect that it doesn't exist



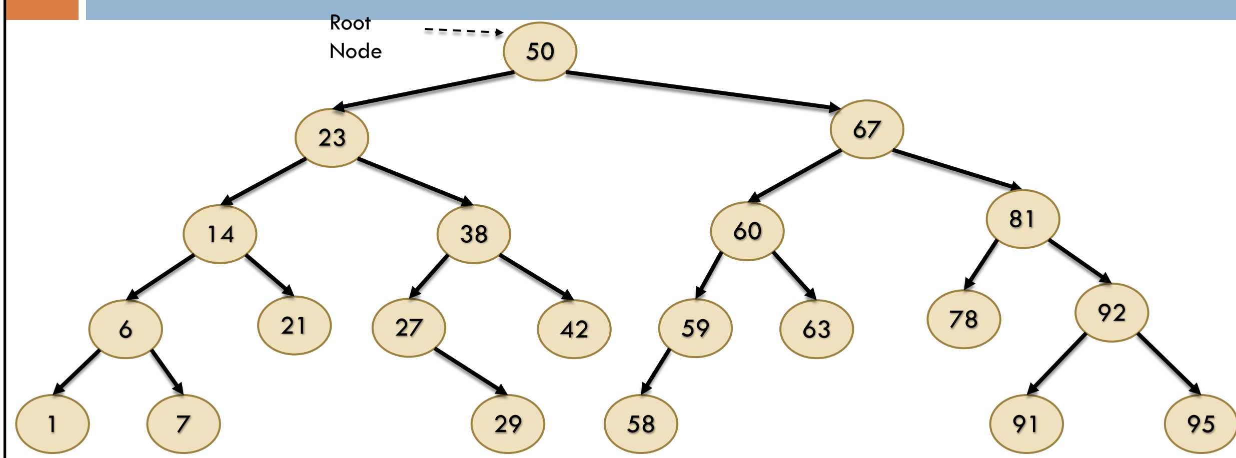
Removal of a leaf node

- 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



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)

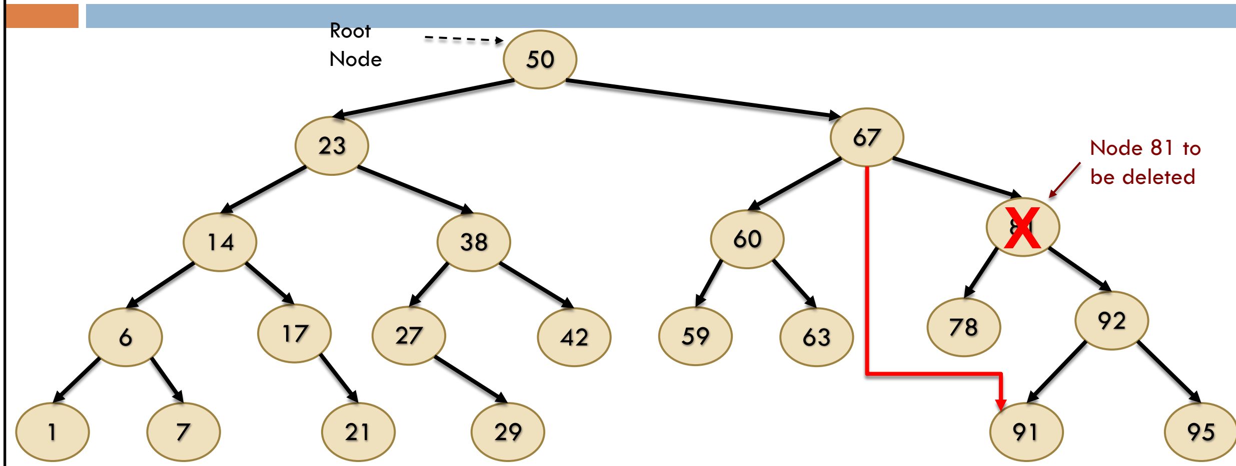
What about deleting a node with two children [2/2]

- 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

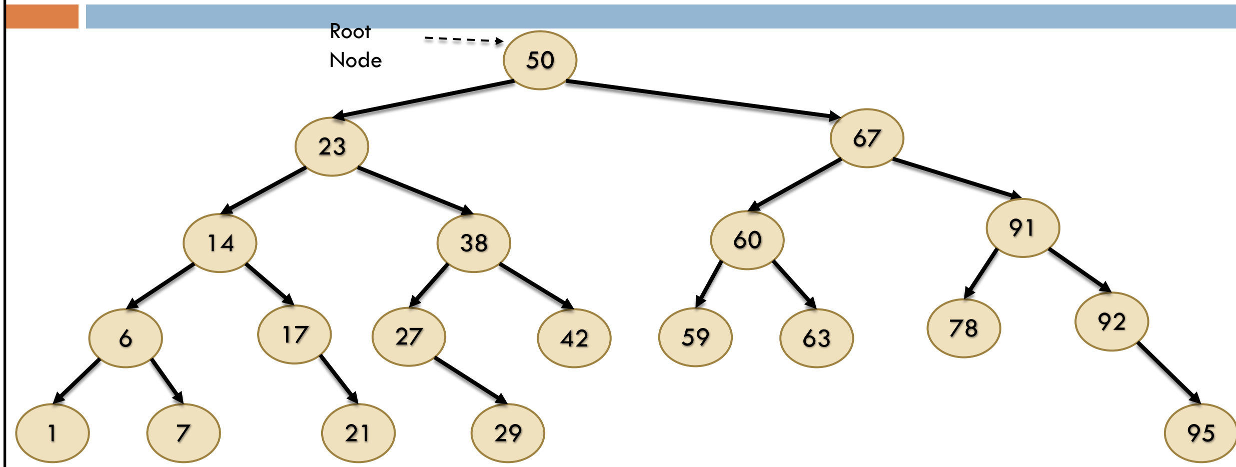
Deleting node 81



COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT DATA STRUCTURES FOR STORAGE L25.43

43

Deleting node 81



COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT DATA STRUCTURES FOR STORAGE L25.44

44

UNBALANCED BINARY SEARCH TREES

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

Albert Einstein

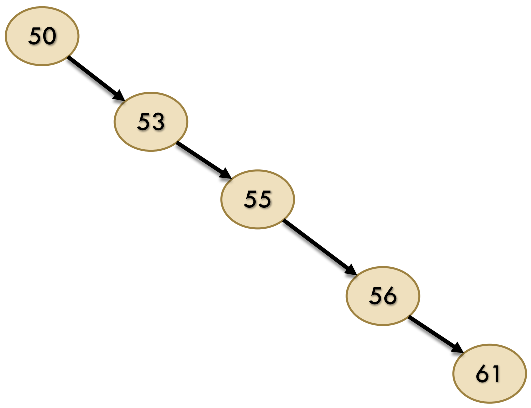
45

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

46

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

49

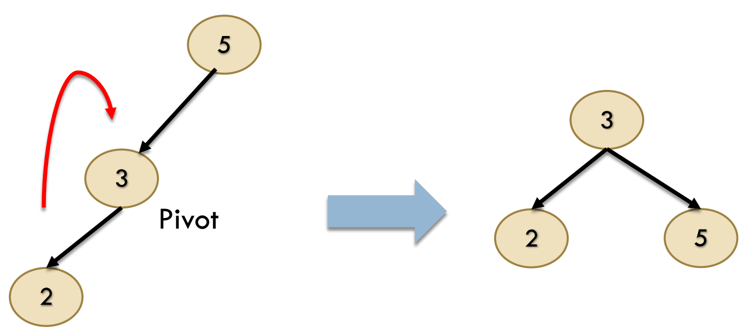
PIVOTING TO BALANCE BSTs

50

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

Balancing BSTs: Rotation



53

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

54

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]