
MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS390 Fundamental Programming Practices (FPP)
Professor Paul Corazza

Lecture 10: Binary Search Trees

Wholeness of the Lesson

Binary Search Trees arose as a natural solution to the need for incorporating efficient insertion and deletion capabilities of linked lists with the support provided for fast sorting and binary search of sorted elements available in arrays and array lists. Expansion from a linear structure to a two-dimensional structure makes a solution of this kind possible. Likewise, any problem becomes easier to solve if one can transcend the boundaries of the problem.

A List Wish-List

1. For many purposes, keeping data stored in memory in a sorted order is a way to optimize a variety of searches on the data.

   A typical problem one might try to solve when it is possible to keep data sorted is:

   Find all Employees having a salary between $50,000 and $75,000.

2. If Employees have been maintained in sorted order – sorted by salary – in some kind of list, then to solve the problem, we find the first Employee in the list with salary no less than 50,000 and also find the first Employee with salary bigger than 75,000. In this way we can specify the desired range of Employees.

3. How to implement this strategy: *maintain sorted order to optimize searches?*

   - If we are using an ArrayList, then maintaining the list in sorted order is expensive because each time we add a new element, it must be inserted into the correct spot, and this requires array copy routines

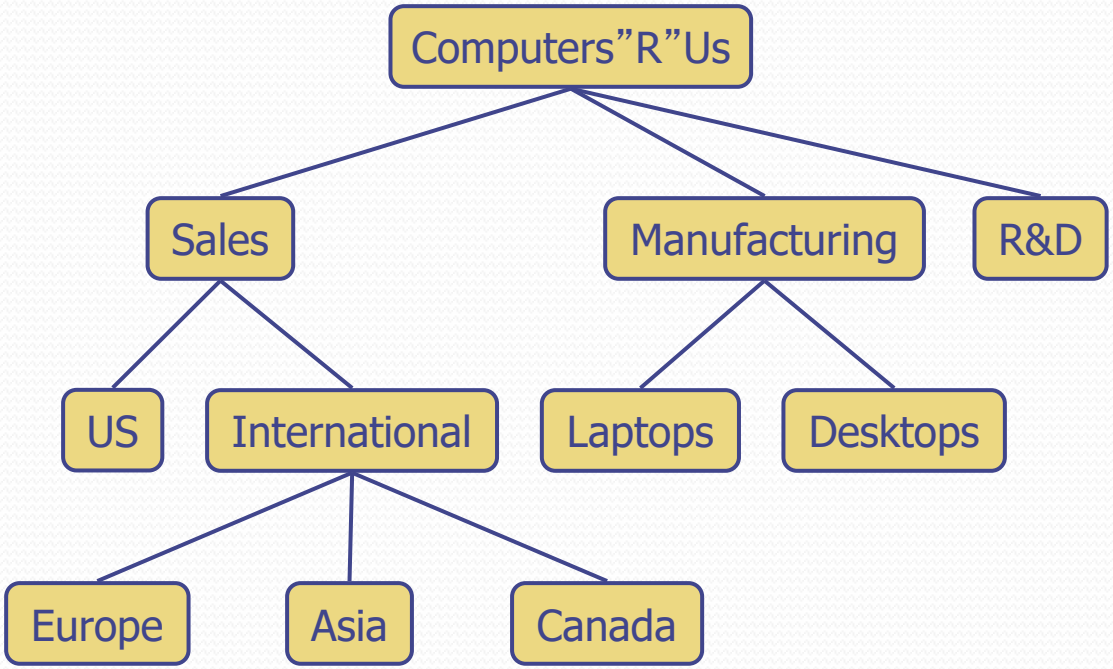
   - If we are using a LinkedList, it is easy to maintain sorted order since insertions are efficient, but searches are not efficient.

4. *The Need.* We need a data structure that performs insertions efficiently in order to maintain sorted order (like a linked list) but that also performs finds efficiently (binary search is highly efficient in an array list when elements are sorted)

5. *Solution:* Binary Search Trees

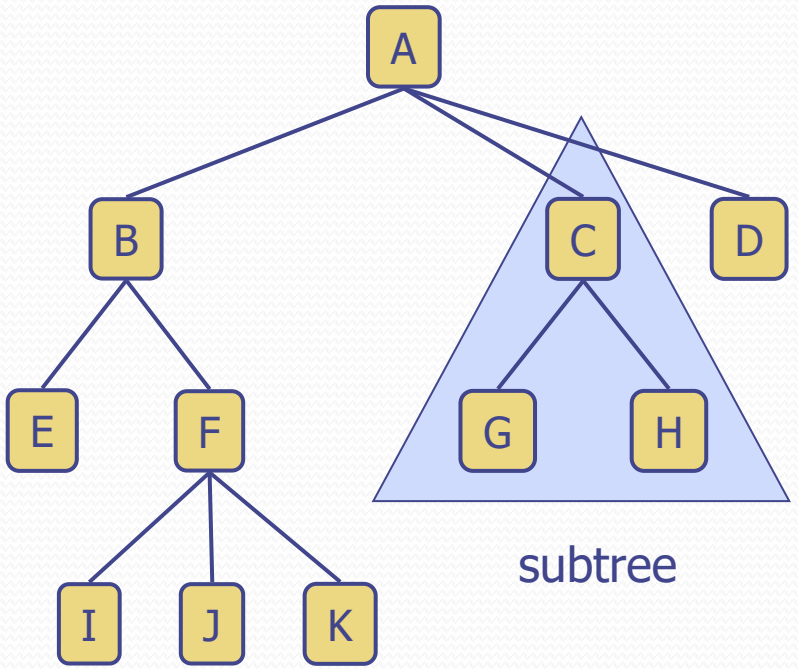
Trees

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems



Tree Terminology

- ***Root***: node without parent (e.g. A in the figure)
- ***Ancestors of a node***: parent, grandparent, grand-grandparent, etc.
  (ancestors of K: F, B, A)
- ***Descendant of a node***: child, grandchild, grand-grandchild, etc.
  (descendants of B : E, F, I, J, K)
- ***Subtree***: tree consisting of a node and its descendants



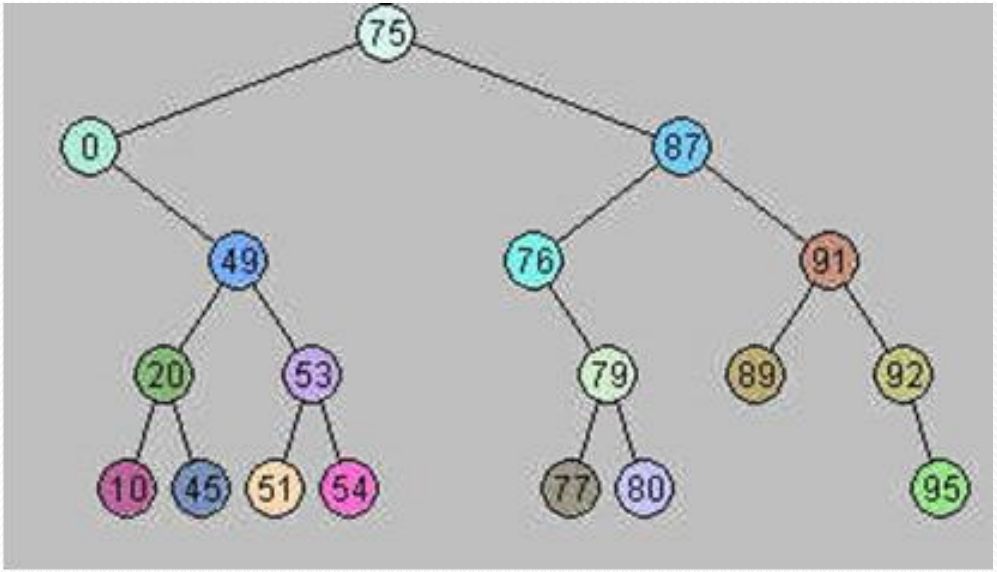
Binary Search Trees

- A *binary tree* is a tree in which each non-null node has a reference to a left and right child node (though one or both may be null).
- A *binary search tree* (BST) is a binary tree in which the BST Rule is satisfied:

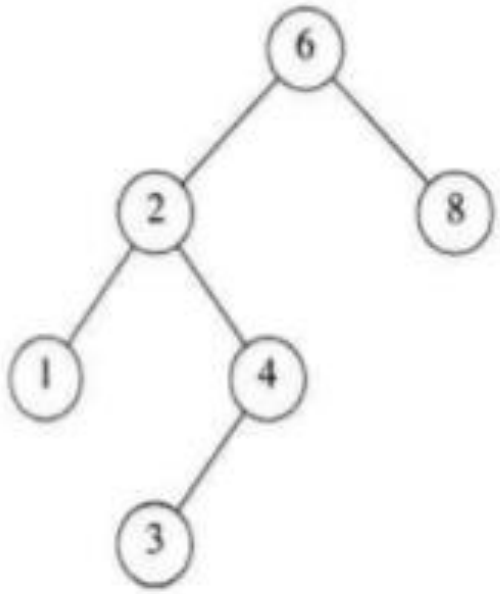
  BST Rule:

  At each node n, every value in the left subtree of n is less than the value at n, and every value in the right subtree of n is greater than the value at N.

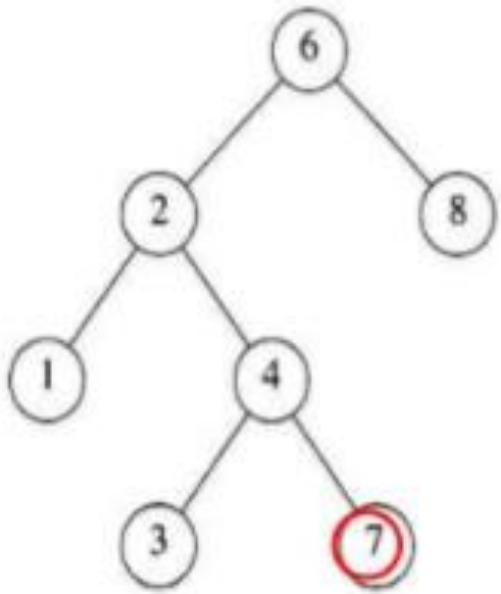
  We assume at first that all values are of type Integer.



Simple Examples



A binary search tree



Not a binary search tree

What BSTs Accomplish

- When implemented properly, BSTs perform insertions and deletions faster than can be done on Linked Lists and performs any `find` with as much efficiency as the binary search on a sorted array.

- In addition, because of the BST Rule, the BST keeps all data in sorted order, and the algorithm (known as "in-order traversal") for displaying all data in its sorted order is very efficient.

Binary Search Tree Operations

Demo code: `lesson10.bst`

- The fundamental operations on a BST are:

```
public boolean find(Integer val)
public void insert(Integer val)
public boolean remove(Integer val)
public void print() (in-order traversal)
```

- Binary search trees are typically implemented using *binary nodes*:

```
class Node {
    private Integer data; // The data in the node
    private Node left; // Left child
    private Node right; // Right child

    Node(Integer theElement) {
        this(theElement, null, null);
    }
    Node(Integer element, Node left, Node right) {
        this.data = element;
        this.left = left;
        this.right = right;
    }
    public String toString() {
        return data.toString();
    }
}
```

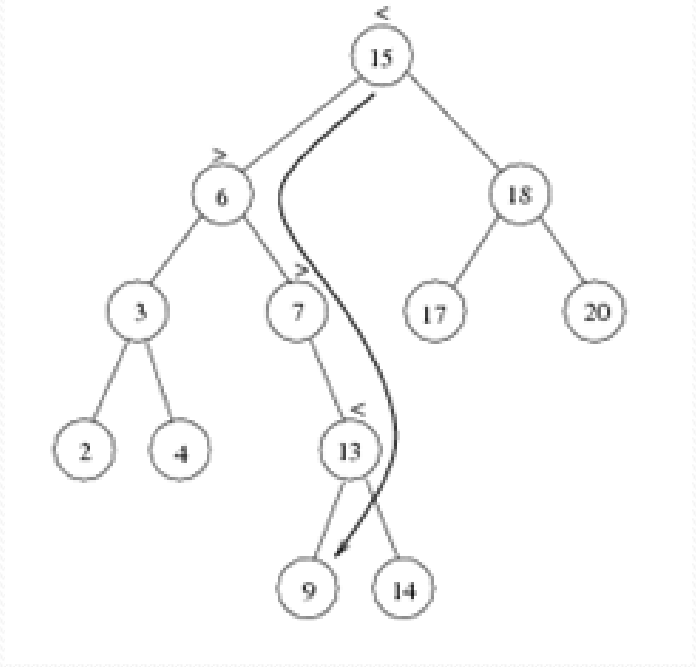
Searching a BST

Searching a BST is an easy generalization of Binary Search on a sorted array. "Divide and Conquer" strategy is expanded to the two-dimensional context of a BST in this case.

Recursive algorithm for finding an Integer x

The recursive *find* operation for BSTs is in reality the binary search algorithm in the context of BSTs.

1. If the root is null, return false
2. If the value in the root equals x, return true
3. If x is less than the value in the root, return the result of searching the left subtree
4. If x is greater than the value in the root, return the result of searching the right subtree



Example: search for 9

Building Binary Search Trees (Client Perspective)

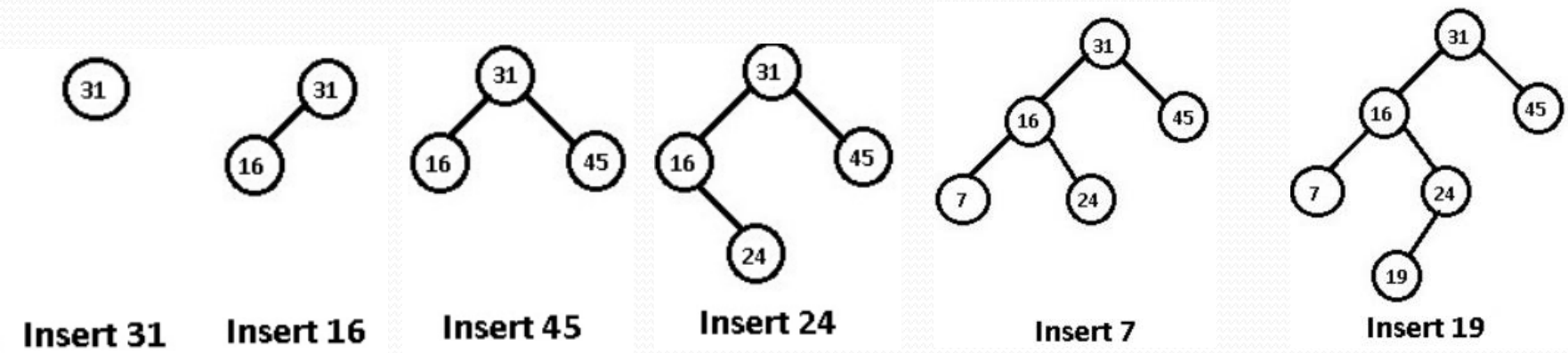
- BSTs are typically built from an *insertion sequence*, starting with an empty tree.
- We use the insertion algorithm (next slide) to insert the elements one by one to the BST.
- Insertion into a BST uses the same Divide and Conquer strategy to locate the *insertion point* in the BST.

Insertion into a BST (recursive)

Recursive algorithm for insertion of Integer x

1. If the root is null, create a new root having value x
2. Otherwise, if x is less than the value in the root, insert x into the left subtree; if x is bigger than the value in the root, insert x into the right subtree

Insert these numbers in an initially empty BST: 31, 16, 45, 25, 7, 19



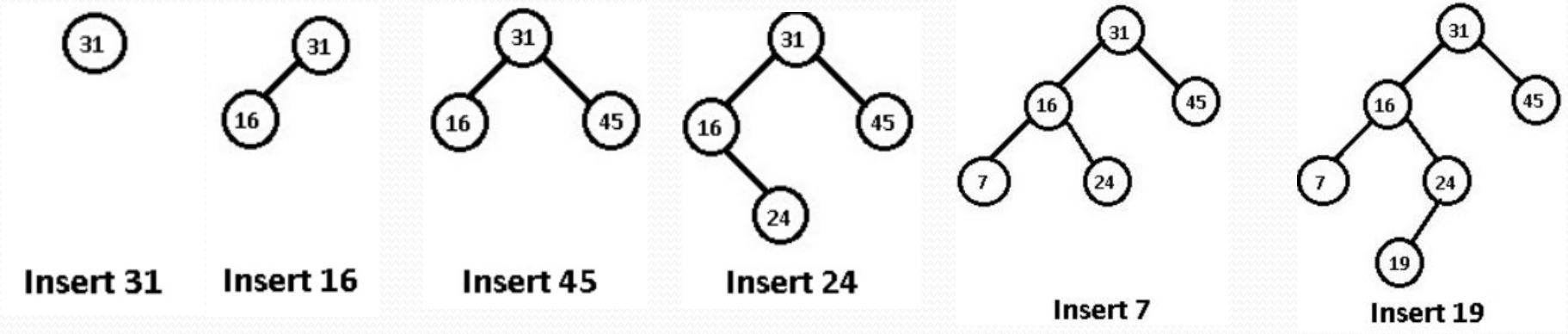
Insertion into a BST (iterative)

See Demo `lesson10.bst.MyBST.java`

Note: The demo provides an implementation of the insertion algorithm that does not support insertion of duplicates. Handling of duplicate values is discussed later in this lesson.

Example

Insert these numbers in an initially empty BST: 31, 16, 45, 24, 7, 19

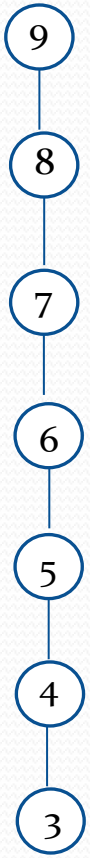


Exercise 10.1

Build the BST obtained from the following insertion sequence: 9, 8, 7, 6, 5, 4, 3. What does your BST look like?

Solution

No more efficient than a linked list!



BST In-Order Traversal

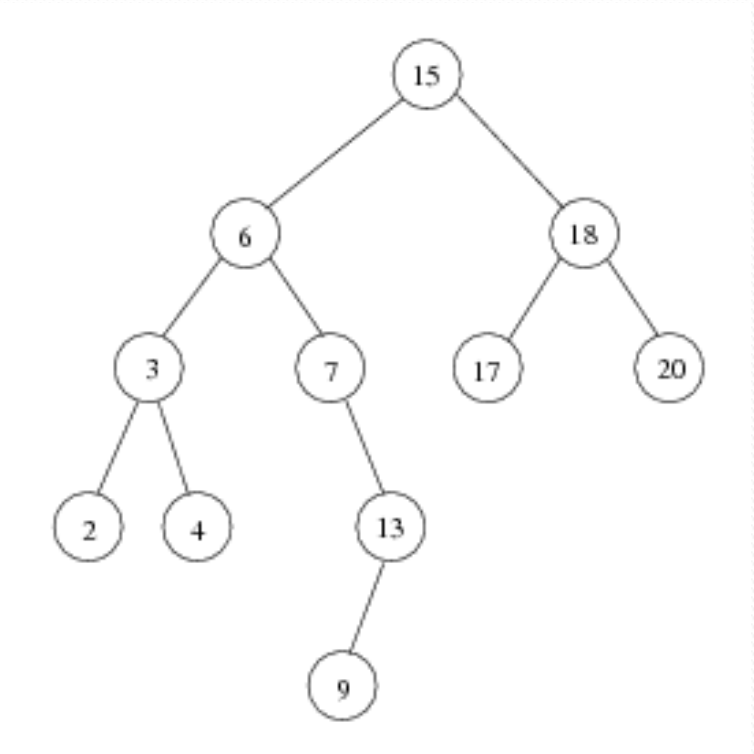
BST In-Order Traversal visits every node of a BST "in order" – smaller values are visited before larger values.

The In-Order traversal strategy is shown in the following recursive print algorithm. The procedure outputs values of every node *in sorted order*

1. If the root is null, return
2. Print the left subtree of the root, in sorted order
3. Print the root
4. Print the right subtree of the root, in sorted order

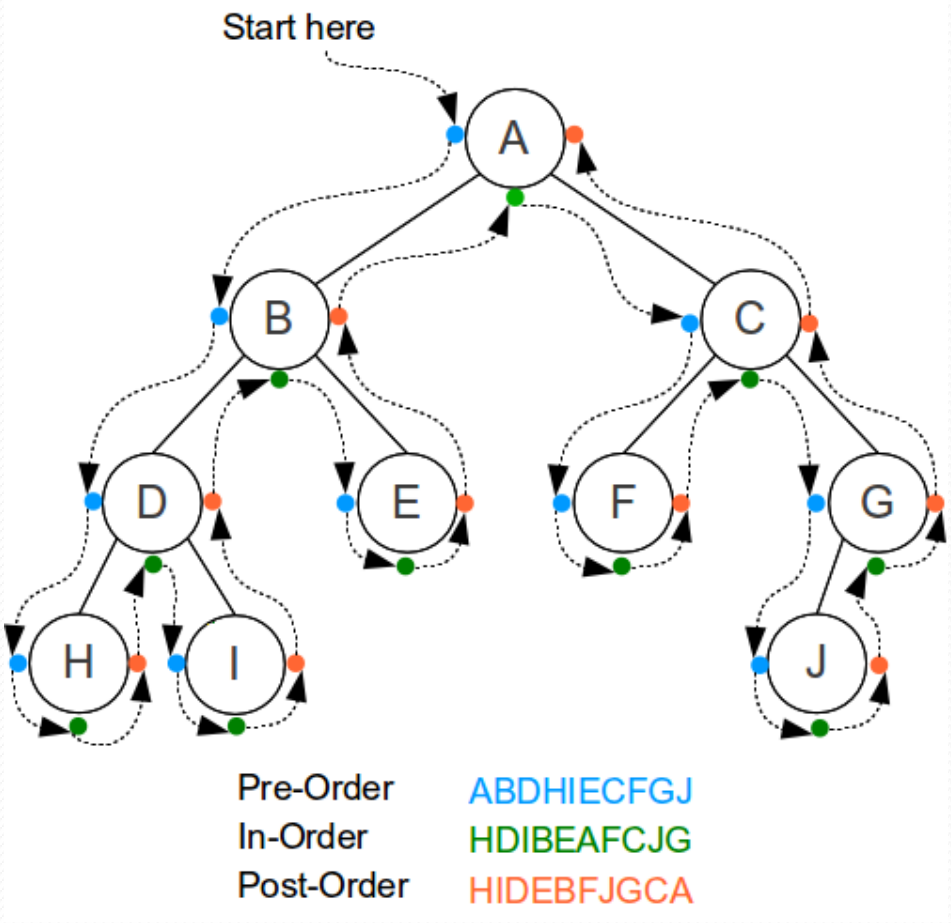
BST In-Order Traversal

- Example:



In-Order: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

BST Traversal

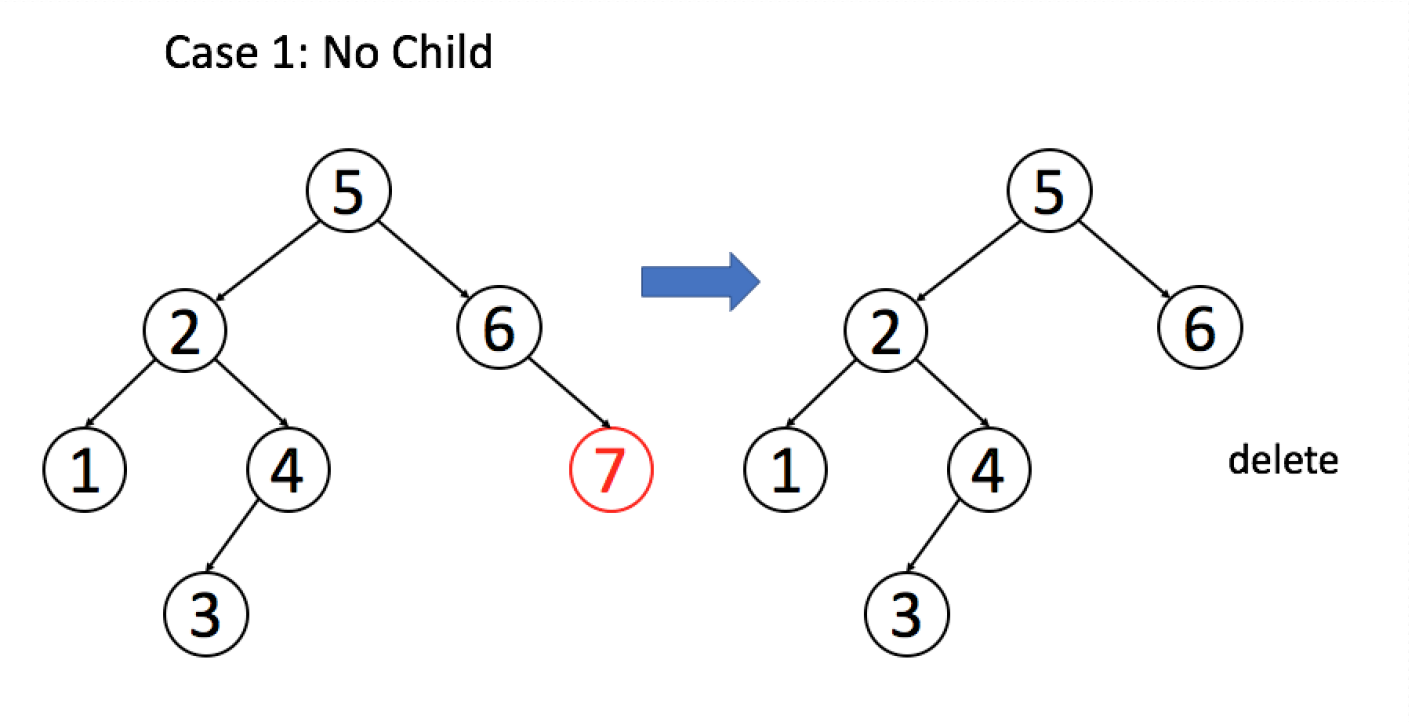


Deletions in a BST

Algorithm to remove Integer x

- Case I: Node to remove is a leaf node

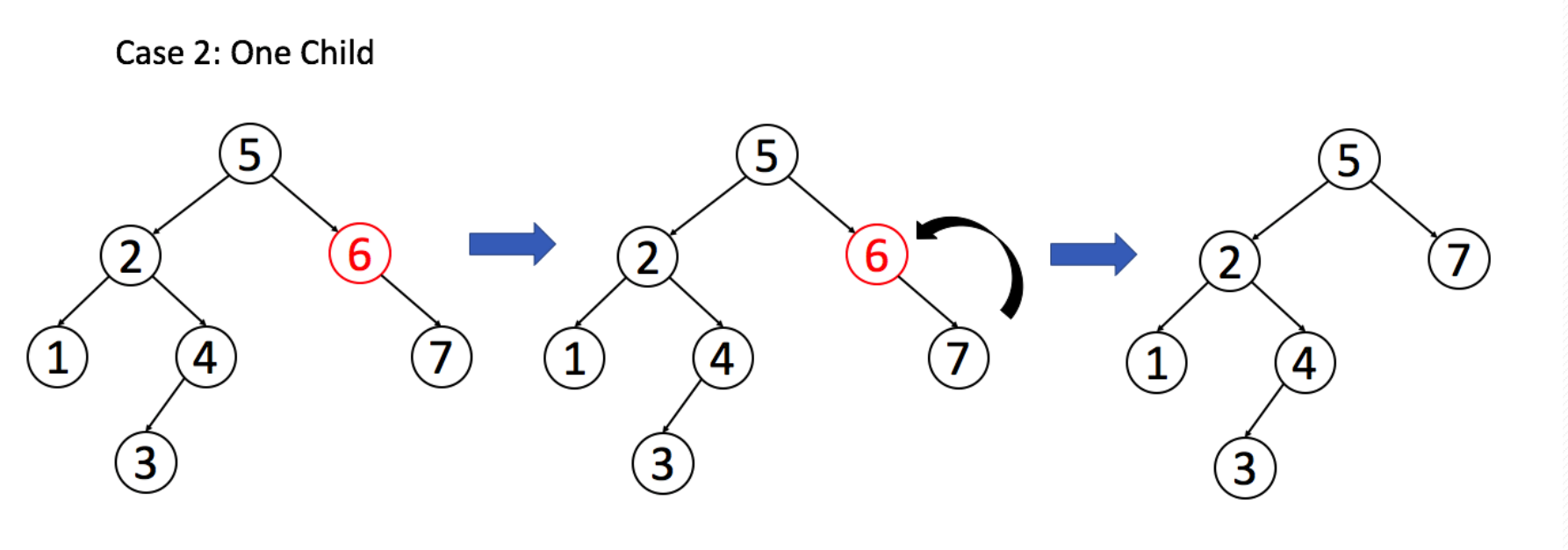
  Find it and set reference to this node to null



Deletions in a BST

- Case II: Node to remove has one child

  Create new link from parent to child, and set node to be removed to null



Deletions in a BST - continued

Algorithm to remove Integer x

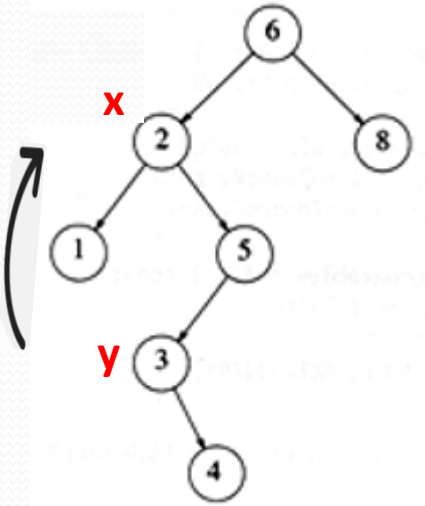
- Case III: Node to remove has two children

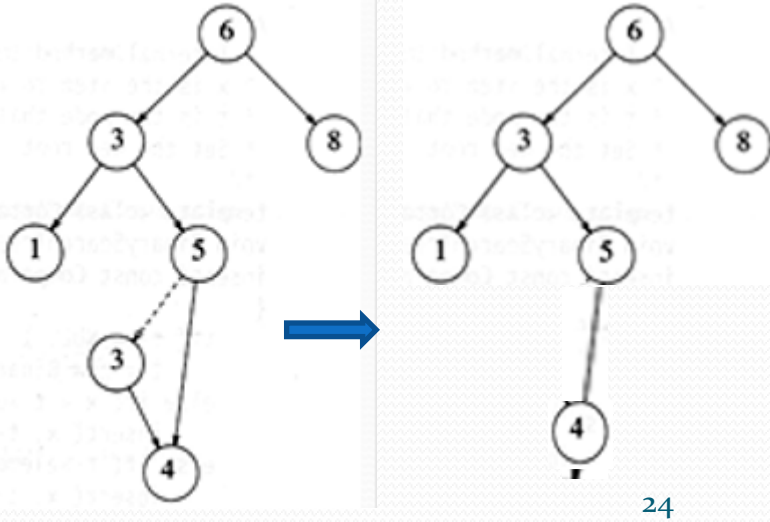
Successor is not a leaf

- Find smallest node (inorder successor) in right subtree

- Replace x with y in node to be removed.

- Delete node that used to store y – this is done as in one of the two cases above

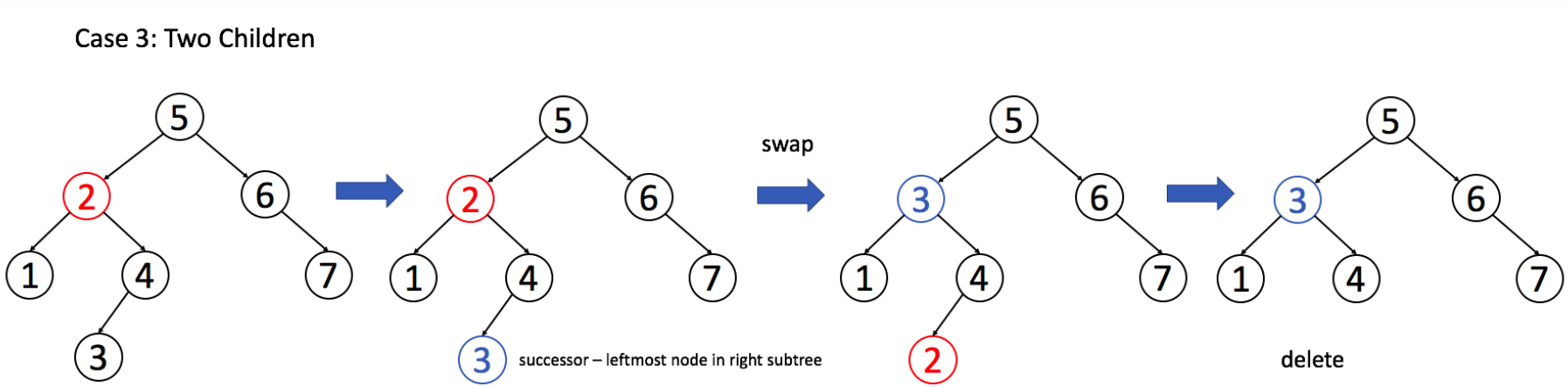




Deletions in a BST - continued

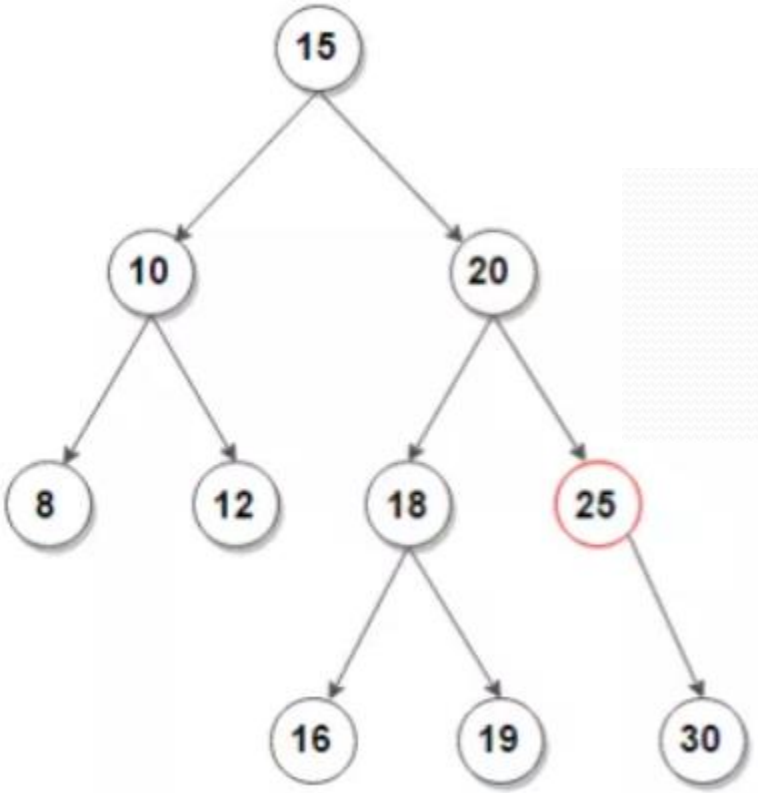
- Case III: Node to remove has two children

Successor is a leaf

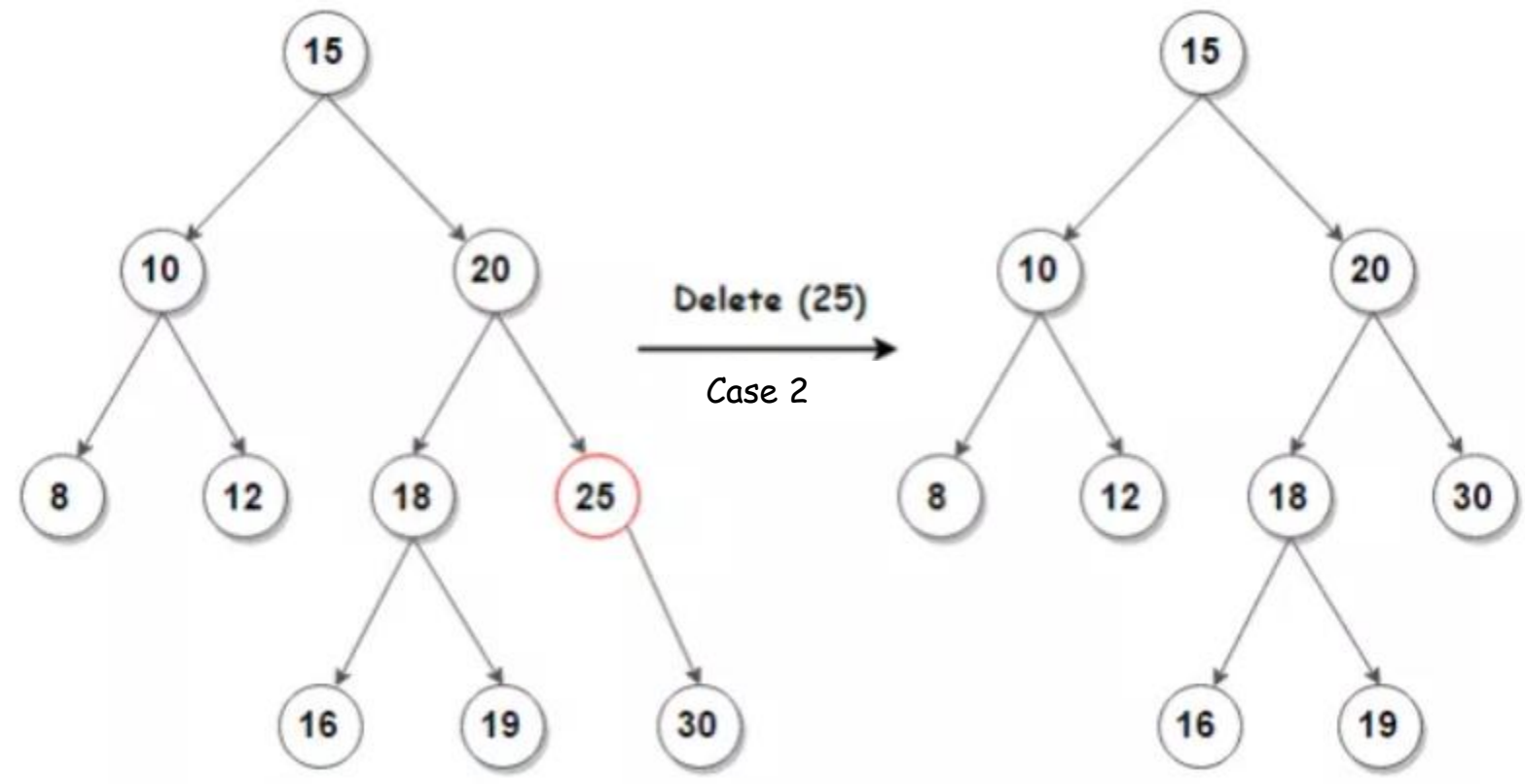


Exercise 10.2 – Removal in a BST

Remove 25 from the following BST



Solution



Handling Duplicates in a BST

- The algorithms and implementations above assume no duplicate values are being inserted into the BST.

- To handle duplicate values, store in each `Node` a `List` (instead of a value); then when a value is added to a `Node`, it is instead added to the `List`. When removal causes one of the lists to become empty, the node itself is removed.

- **Example**: Suppose we are storing `Employee`s in a BST, ordered by `name`. Our BST will be created so that the value in each node is a `List<Employee>` instance. Then, after insertions, all `Employee`s with the same `name` will be found in a single `List` located in a single `Node`.

- **Example**: Suppose we want to have a data structure that provides us with a count of how many Employees have a particular salary. To begin, we could order Employees by salary (using a `SalaryComparator`), insert into a BST using `Lists` as values in each `Node`. After all `Employees` have been inserted, we could answer the query "How many have salary 50000?" by searching for the 50,000 node and then returning the size of the list stored at that node.

Using BSTs for Sorting

- Consider the following procedure for sorting a list of Integers:
  - Insert them into a BST
  - Print the results (or modify "print" so that it puts values in a list)
- How good is this new sorting algorithm? How does it compare to `MinSort` ? The Lab asks you to implement this algorithm and compare its running time to that of `MinSort`.

BSTs in the Java Libraries

- Java uses a special kind of BST (a "balanced" tree) as a background data structure for several of their data structures – namely `TreeSet` and `TreeMap`.

- If a BST becomes unbalanced, its performance degrades dramatically; techniques have been developed to keep a tree from slipping into an unbalanced condition – the most popular such technique produces *red-black trees*. `TreeSet` and `TreeMap` are implemented using red-black trees.
- See MyJavaBST in package lesson10.bst

Main Point

A *binary search tree* (BST) is a binary tree in which the BST Rule is satisfied:

> At each node *N*, every value in the left subtree of *N* is less than the value at *N*, and every value in the right subtree of *N* is greater than the value at *N*.

BSTs provide efficient search, insert, and remove operations on orderable data. A binary search tree is an example of the principle of Diving: Because the structure is right, the basic operations are accomplished with maximum efficiency.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

Fundamental patterns of consciousness in the realm of binary trees

1. Binary search trees support insert, remove, and find operations with efficient performance, as well as efficient support for accessing elements in order, such as findMin, findMax, and finding elements in a specified range.
2. The structure of complete binary trees involves an expansion from 1 to 2 to 4 to 8, and eventually to 64 (mirroring to a large extent the significant numbers that mark the progress of unfoldment within the Ved, from A to AK to the fourfold Rishi, Devata, Chhandas, Samhita, to 8-fold prakriti through the first Richa to the 64 fundamental impulses that structure the first 192 syllables of Rk Ved).

---

3. **Transcendental Consciousness:** TC is the home of all the impulses of natural law, of creative intelligence.
4. **Wholeness moving within itself**: In Unity Consciousness, the emergence the structure of pure knowledge as appreciated as a self-referral activity of consciousness interacting with itself.