04-630
Data Structures and Algorithms for Engineers

Lecture 11: Binary Search Trees

George Okeyo/Theophilus A. Benson

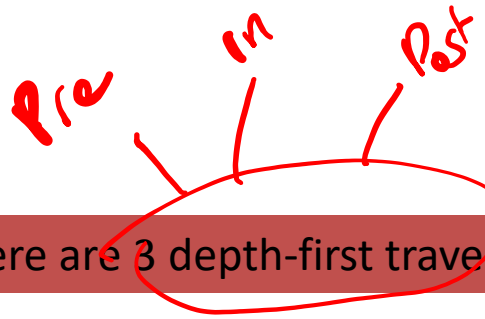
Adopted and Adapted from Material by:

David Vernon: vernon@cmu.edu ; www.vernon.eu

Lecture 7b

Trees

- Types of trees
- Binary Tree ADT
- Tree traversal
- Binary Search Tree
- Optimal Code Trees
- Huffman's Algorithm
- Height Balanced Trees
  - AVL Trees
  - Red-Black Trees



There are 3 depth-first traversals

A binary tree which when traversed **in-order,** the keys are sorted fashion

Relevance to LeetCoding:

Use a property to perform fast operations:

- Kth Smallest element in Binary Search Tree

How it works:

- Merge two Binary Trees
- Flatten Tree to a linked list
- Return *order traversal

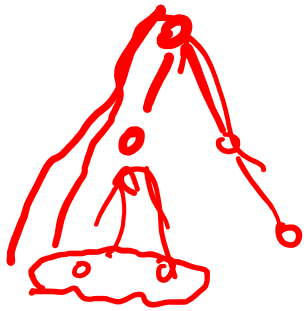
Announcements!

- L33T Coding Mini-workshop
  - 2 google Engineers: week of March 4 or March 11
  - Email me thoughts/opinions

- Teaching remotely today

Tree Traversal

Tree Traversals



- To perform a traversal of a data structure, we use a method of visiting every node in some predetermined order

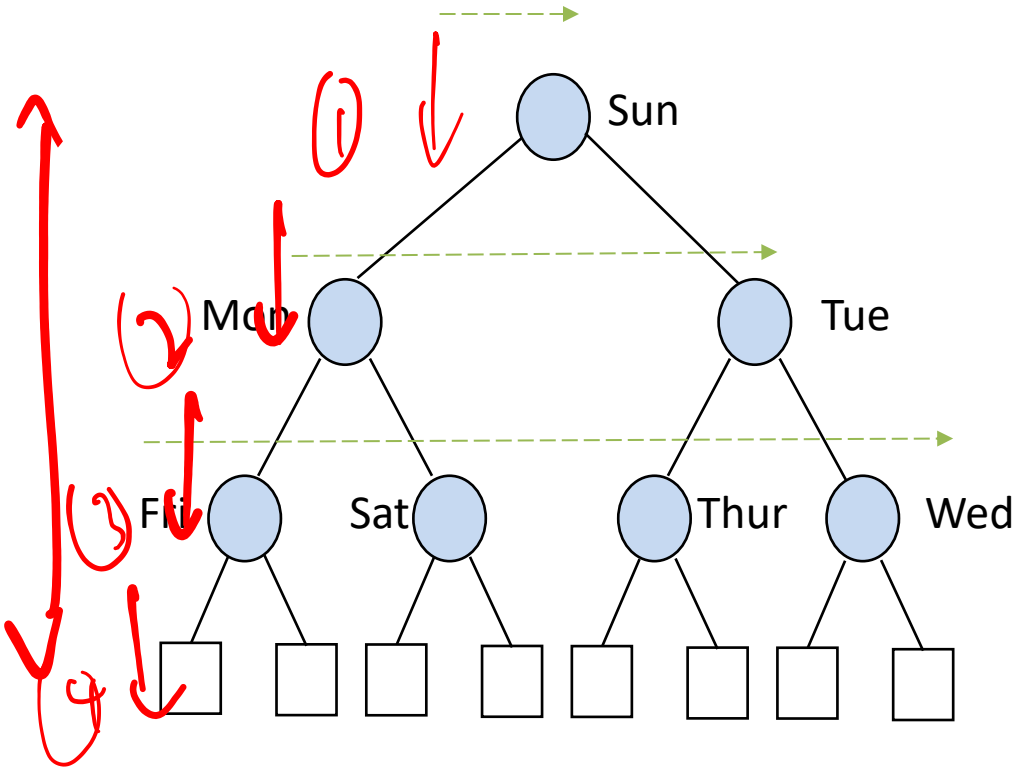
- Traversals can be used

  - to test data structures for equality
  - to display a data structure
  - to construct a data structure of a given size
  - to copy a data structure

apply traversal "thoo" to a tree

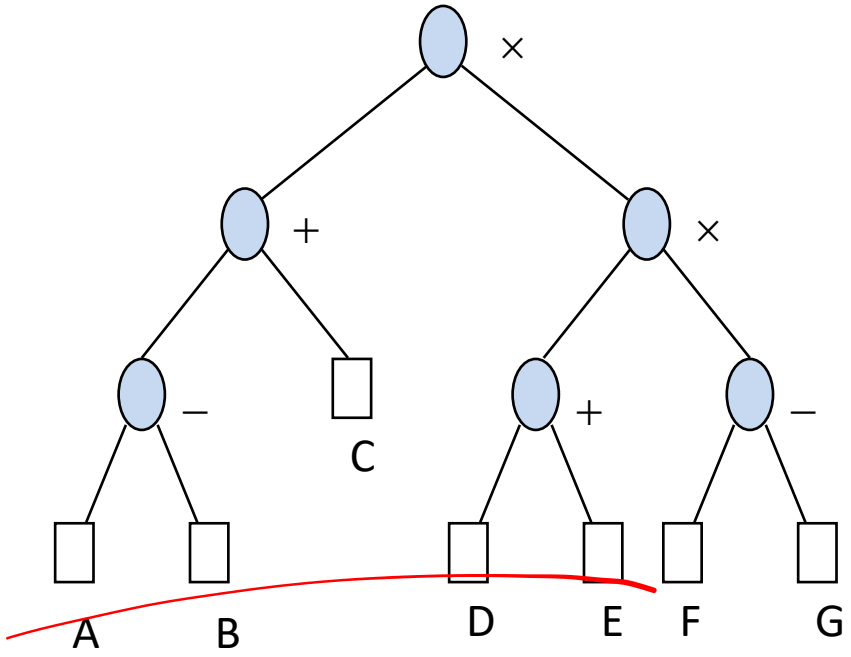
Breadth-First traversal

- The traversal happens one level at a time.
- You traverse all children **at one level** before proceeding **to the grandchildren at the next level**.



Depth-First Traversals

- Consider a binary tree.
- There are 3 depth-first traversals
  - **Pre-order traversal**: root then children(left, right)
  - **Post-order traversal**: children(left, right) then root.
  - **In-order traversal**: left child, root, right child
- For example, consider the expression tree:



Depth-First Traversals

- Inorder traversal

  left → root → right

  A – B + C x D + E x F – G

- Postorder traversal

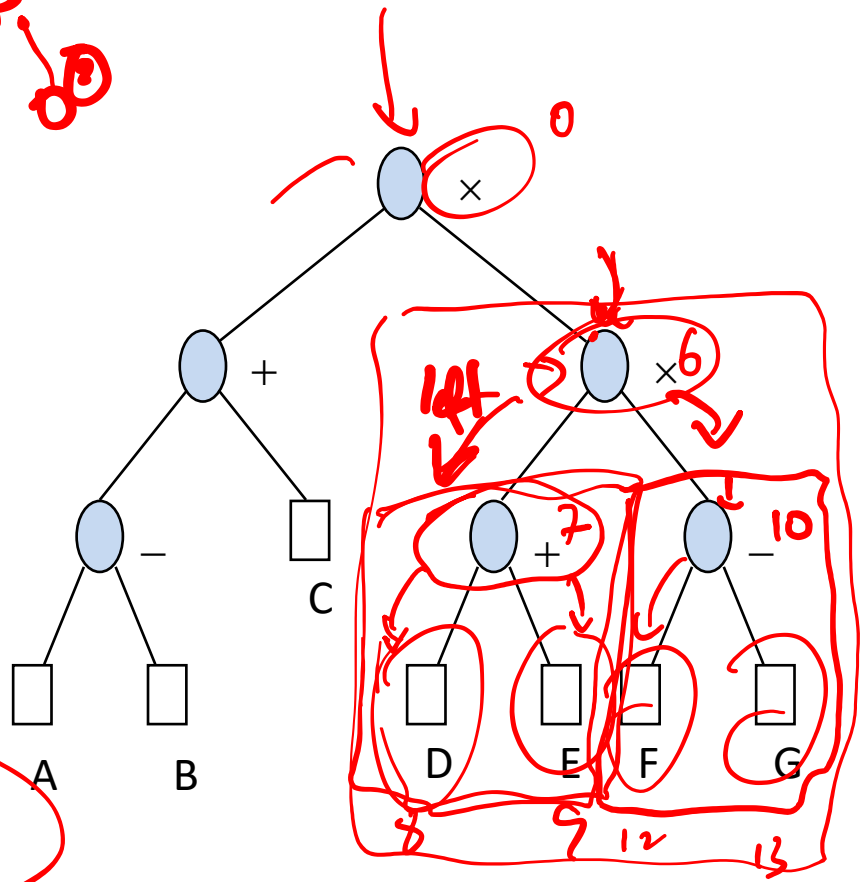
  left → right → root

  A B – C + D E + F G – x x

- Preorder traversal

  root → left → right

  x + – A B C x + D E – F G



Depth-First Traversals

- Inorder traversal

  A – B + C x D + E x F – G

This is the infix expression corresponding to the expression tree

- Postorder traversal

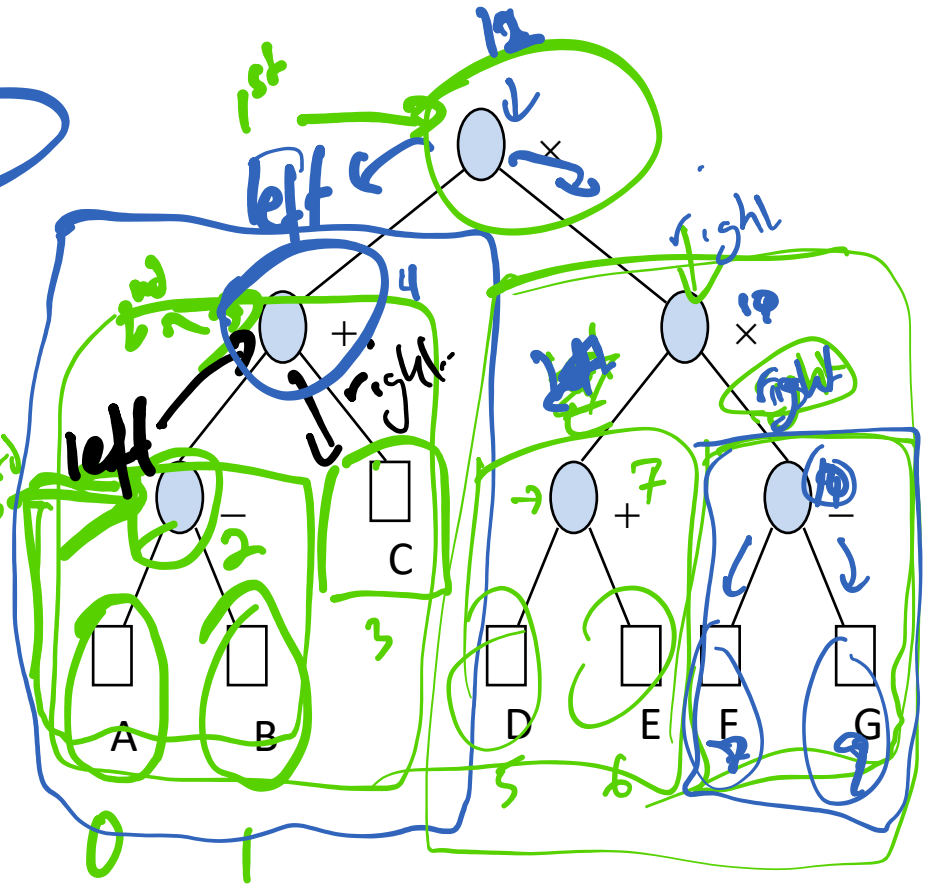
  A B – C + D E + F G – x x

Postorder traversal gives a postfix expression

- Preorder traversal

  x + –A B C x + D E – F G

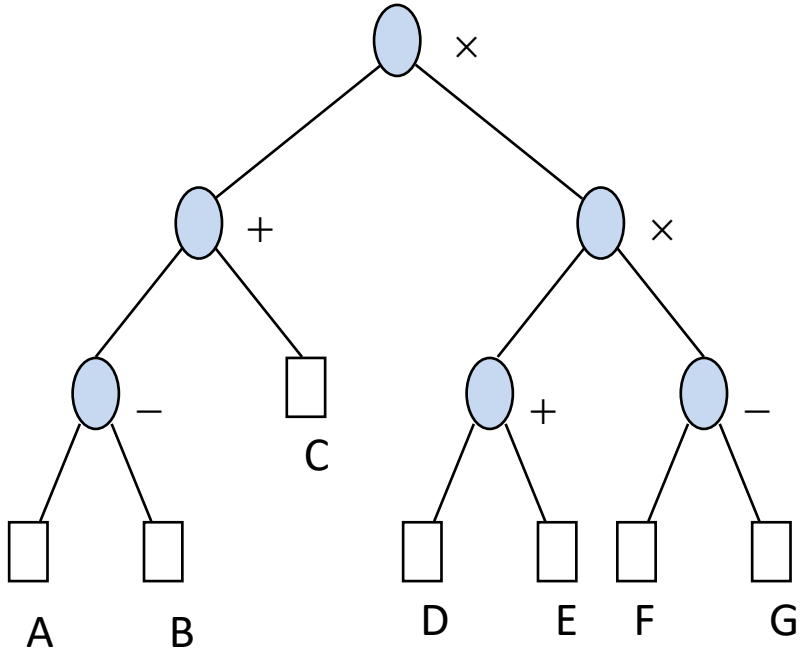
Preorder traversal gives a prefix expression



Depth-First Traversals

Recursive definition of inorder traversal

A – B + C x D + E x F – G



Depth-First Traversals

Recursive definition of inorder traversal

Given a binary tree T

if T is empty

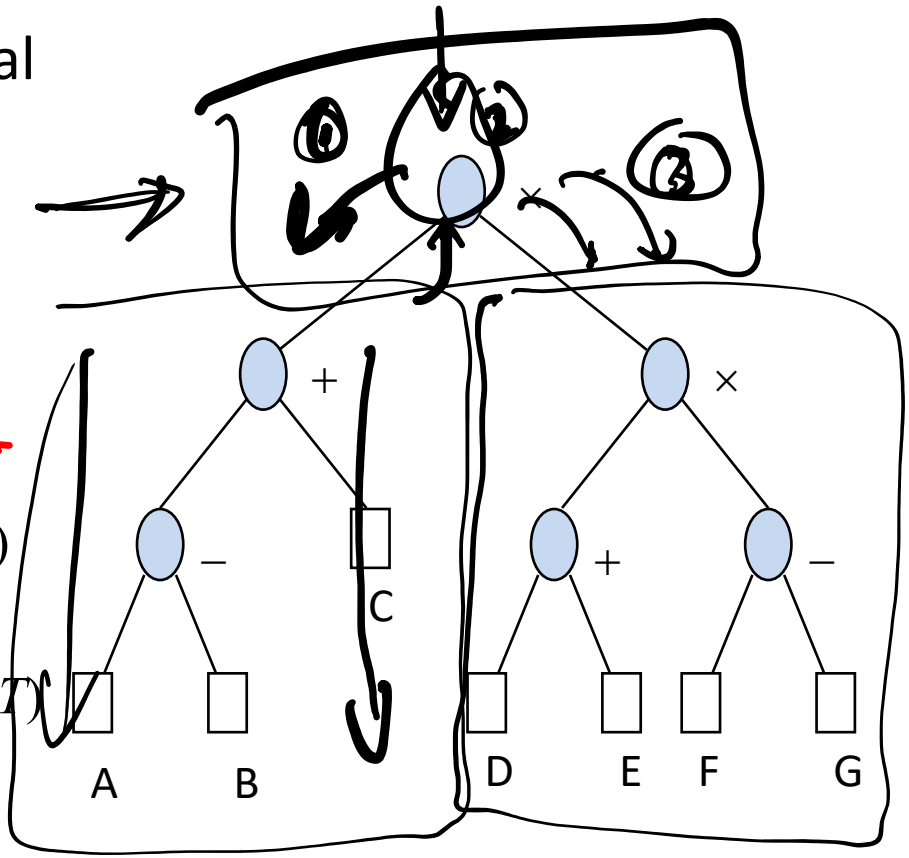
visit the external node

otherwise

perform an inorder traversal of $Left(T)$

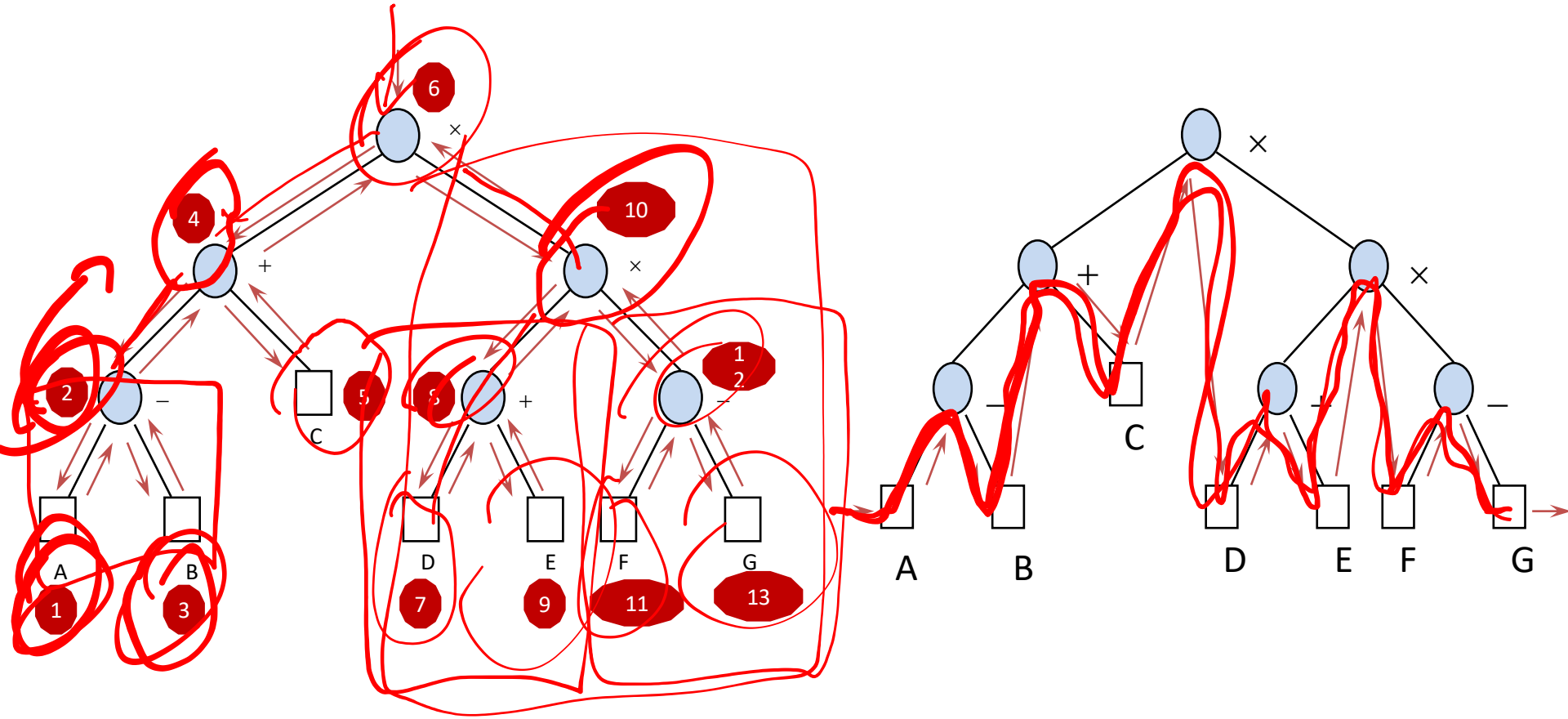
visit the root of T

perform an inorder traversal of $Right(T)$



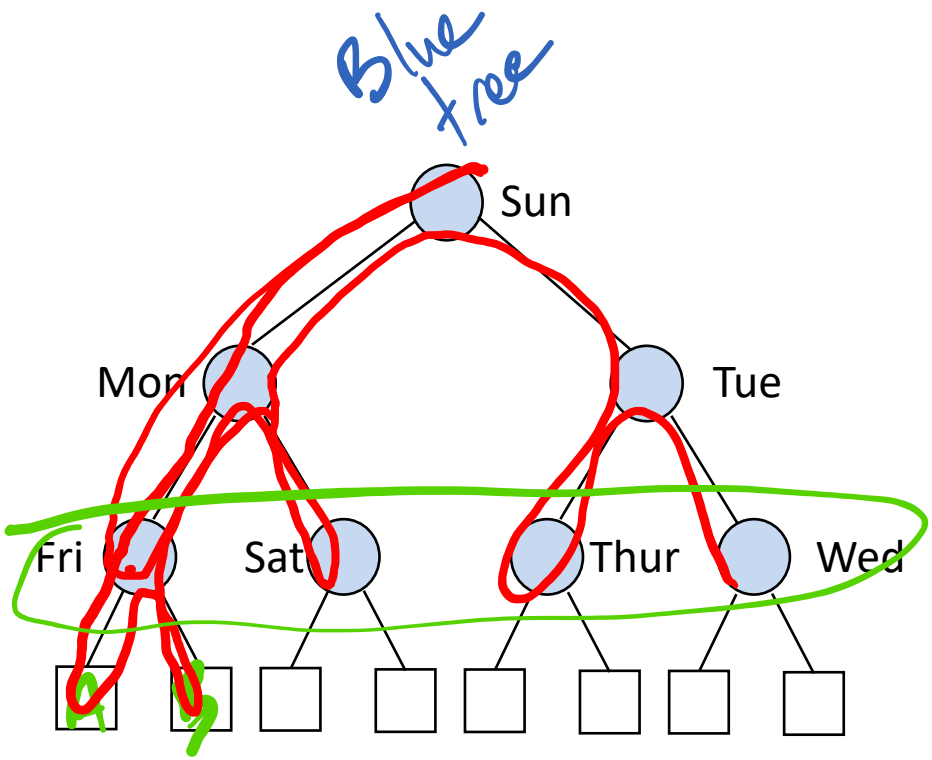
This is Recursive!!!!

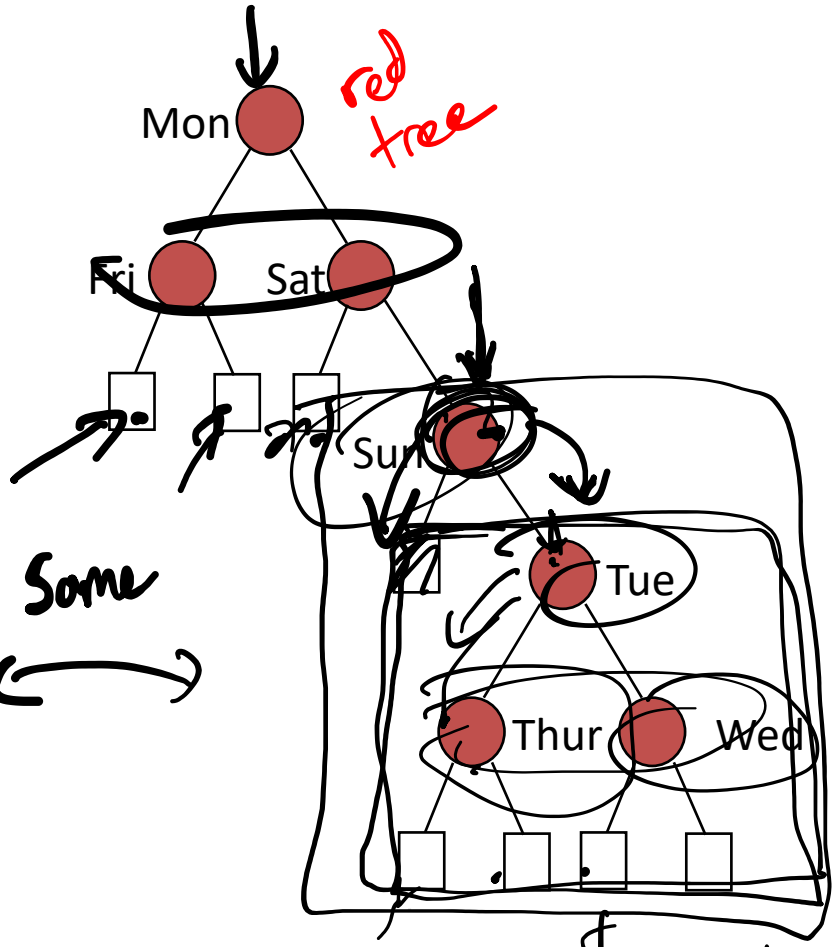
Example: Inorder Traversal



Exercise:

What is the Inorder Traversal for these Trees?

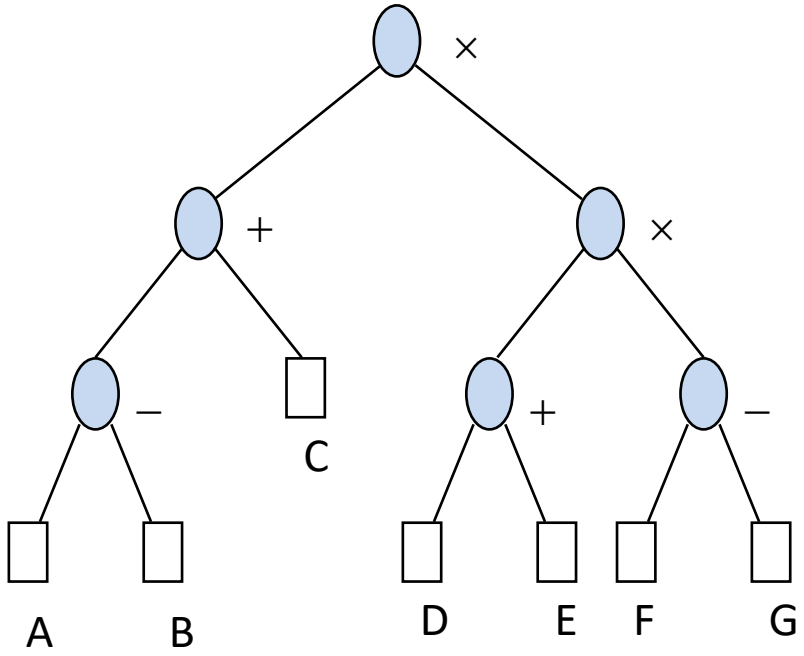




Depth-First Traversals

- Recursive definition of postorder traversal

A B – C + D E + F G – x x



Depth-First Traversals

- Recursive definition of postorder traversal

Given a binary tree T

if T is empty

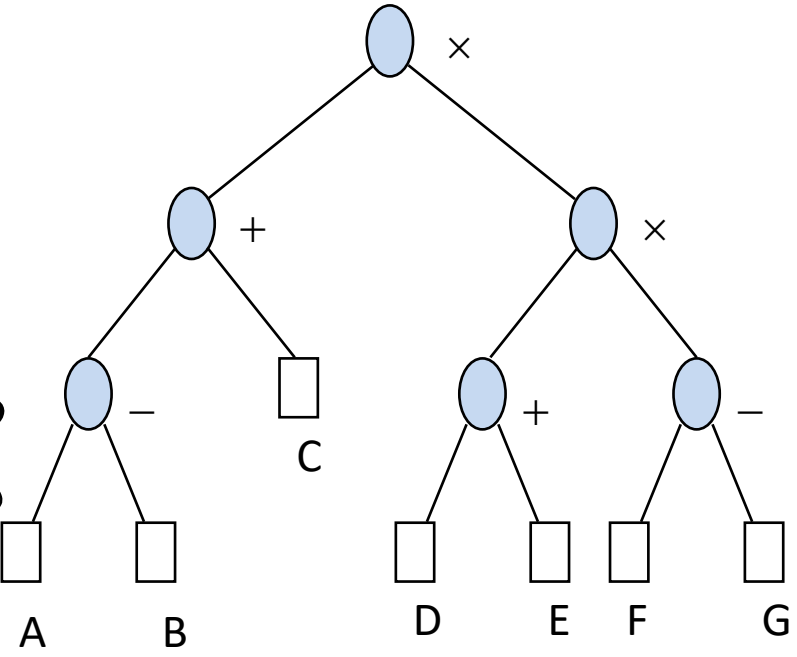
visit the external node

otherwise

perform an postorder traversal of $Left(T)$

perform an postorder traversal of $Right(T)$

visit the root of T

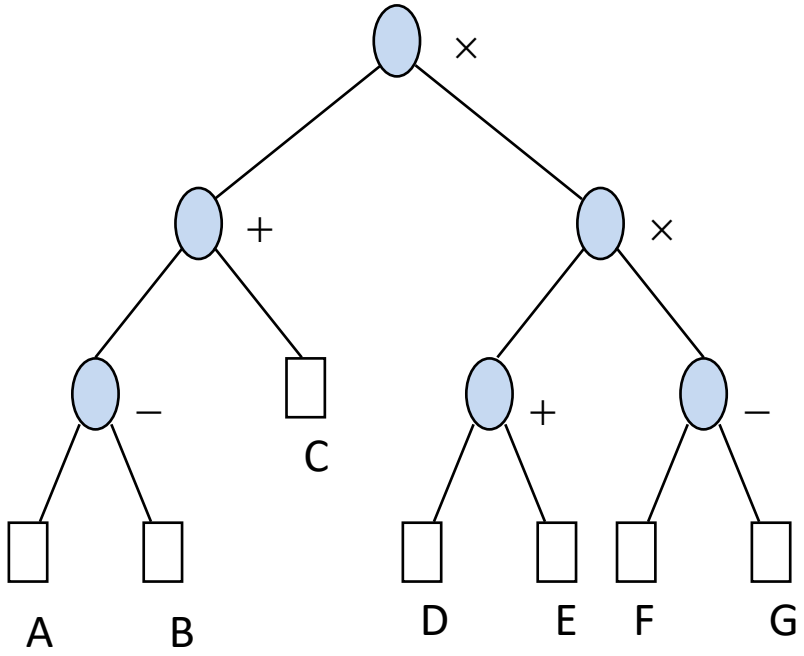


A B – C + D E + F G – x x

Depth-First Traversals

- Recursive definition of preorder traversal

x + –A B C x + D E – F G



Depth-First Traversals

- Recursive definition of preorder traversal

Given a binary tree T

if T is empty

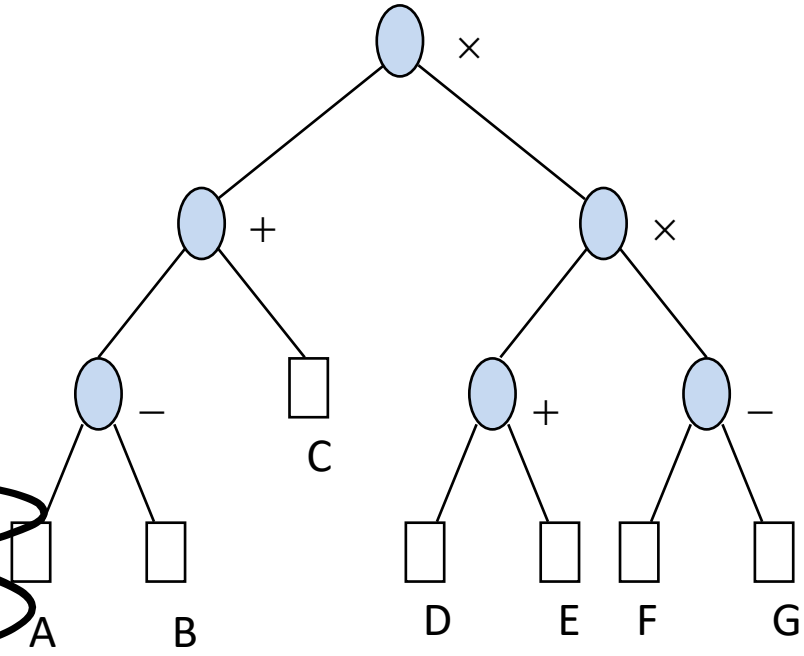
visit the external node

otherwise

visit the root of T

perform an preorder traversal of $Left(T)$

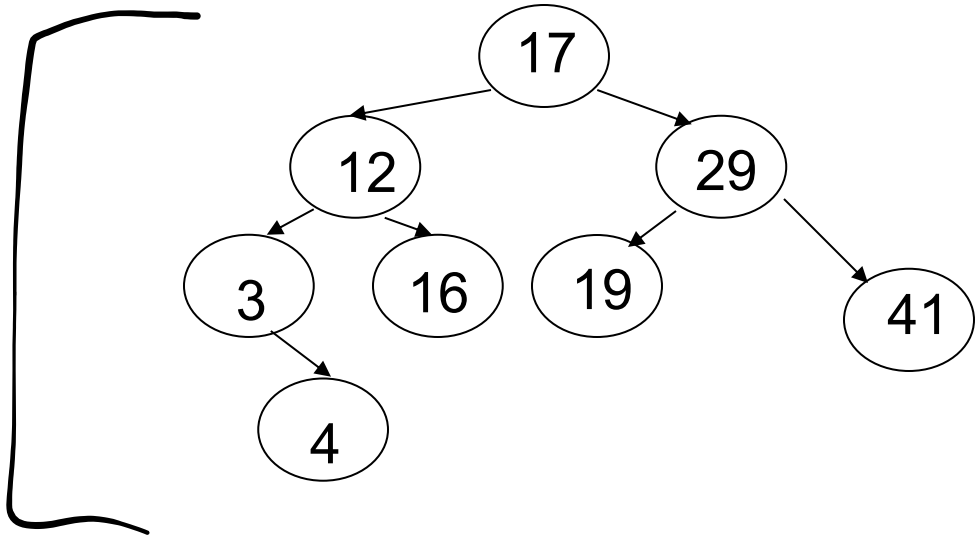
perform an preorder traversal of $Right(T)$



x + –A B C x + D E – F G

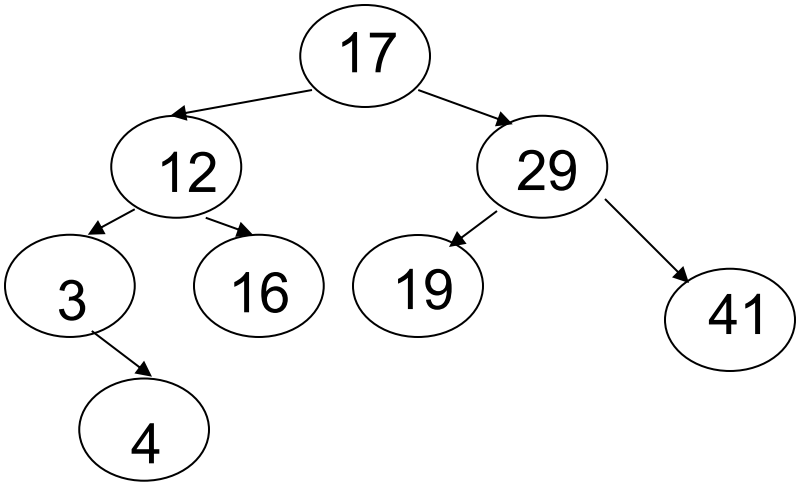
Exercise:

- Show the output of traversal using in-order, pre-order, and post-order traversal.



Exercise

- Show the output of traversal using in-order, pre-order, and post-order traversal.



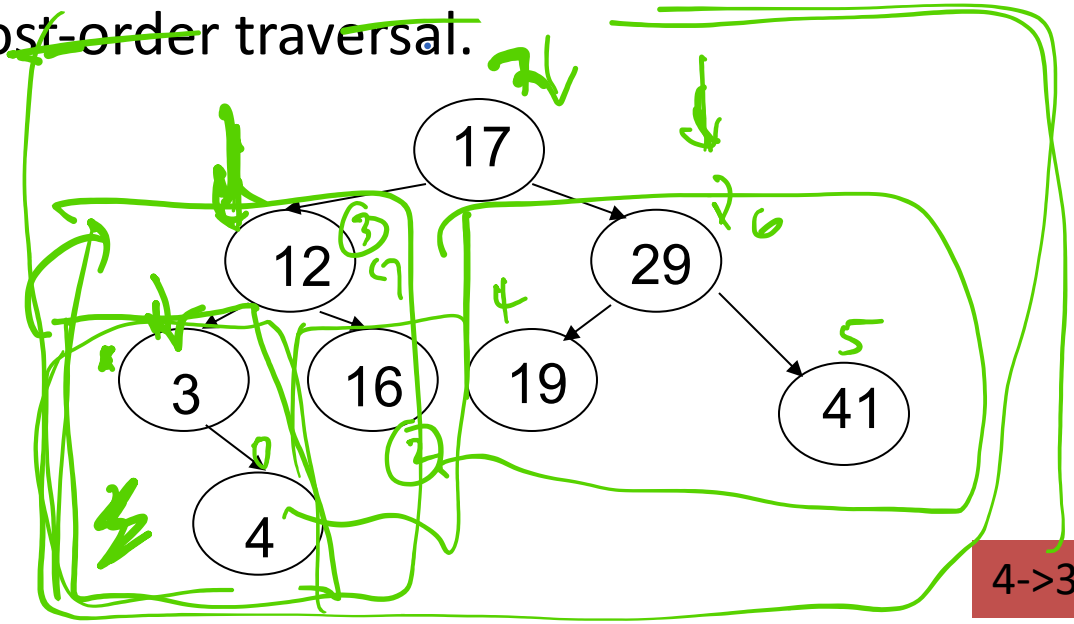
In Order
left -> root -> right

Pre Order
root -> left -> right

Post Order
left -> right -> root

Exercise

- Show the output of traversal using in-order, pre-order, and post-order traversal.



4->3->16->12->19->41->29->17

In Order
left -> root -> right

3->4->12->16->17->19->29->41

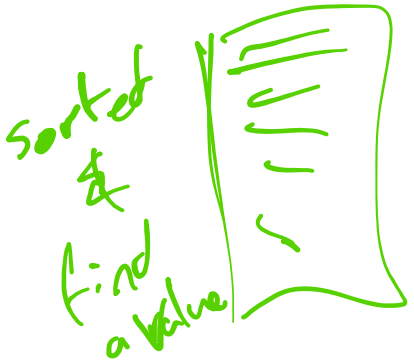
Pre Order
root -> left -> right

17->12->3->4->16->29->19->41

Post Order
left -> right -> root

Binary Search Tree

Applications of BST (Binary Search Tree)



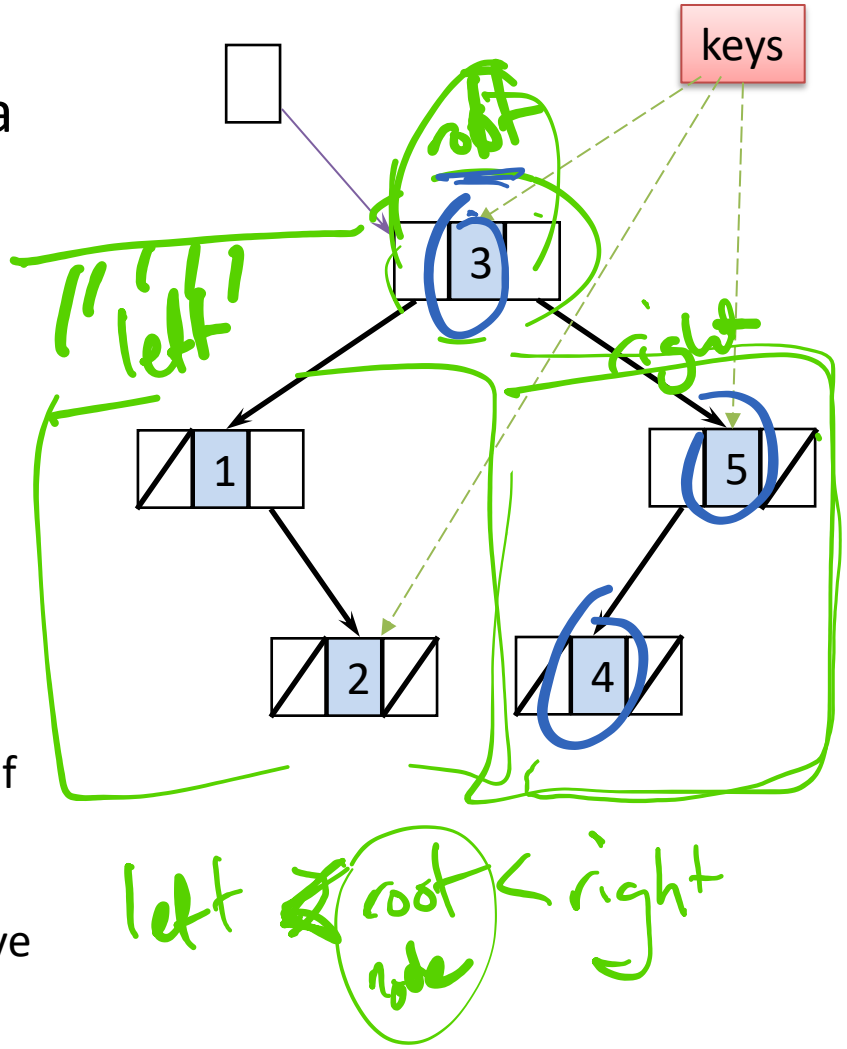
- Implementation of searching algorithms
- Implementation of sorting algorithms:
  - Elements are added and traversed using in-order traversal.
- Indexing and multi-level indexing
- etc.

Database

"Index"

Binary Search Trees

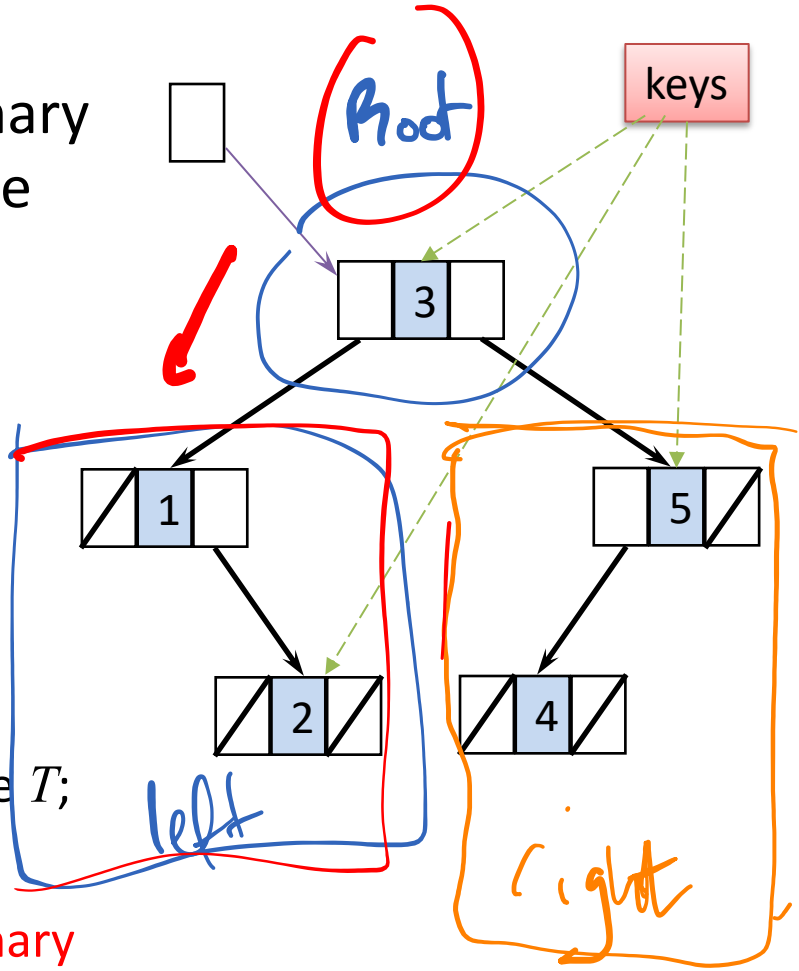
- A Binary Search Tree (BST) is a special type of binary tree
  - it represents information in an ordered format
  - A binary tree is a binary search tree if for every node w,
    - all keys in the left subtree of w have values less than the key of w
    - all keys in the right subtree have values greater than key of w.



Binary Search Trees

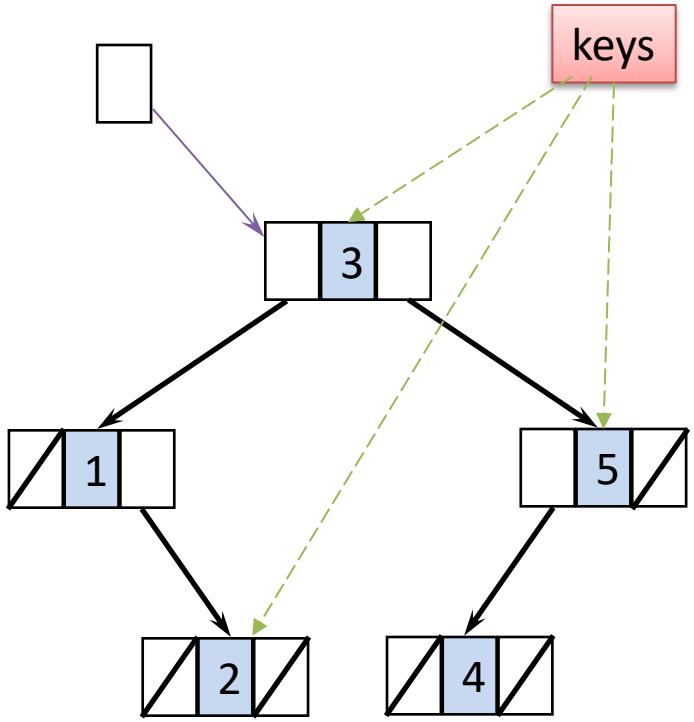
Definition: A binary search tree T is a binary tree; either it is empty or each node in the tree contains an identifier and:

- all keys in the left subtree of T are less (numerically or alphabetically) than the identifier in the root node T;
- all identifiers in the right subtree of T are greater than the identifier in the root node T;
- The left and right subtrees of T are also binary search trees.



Binary Search Trees

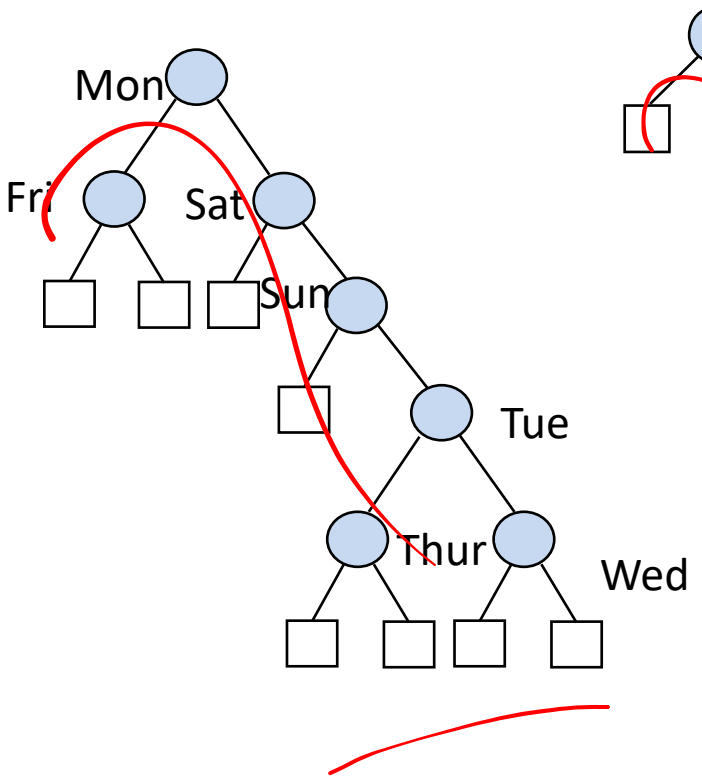
- The main point to notice about such a tree is that, if traversed inorder, the keys of the tree (*i.e.,* its data elements) will be encountered in a sorted fashion

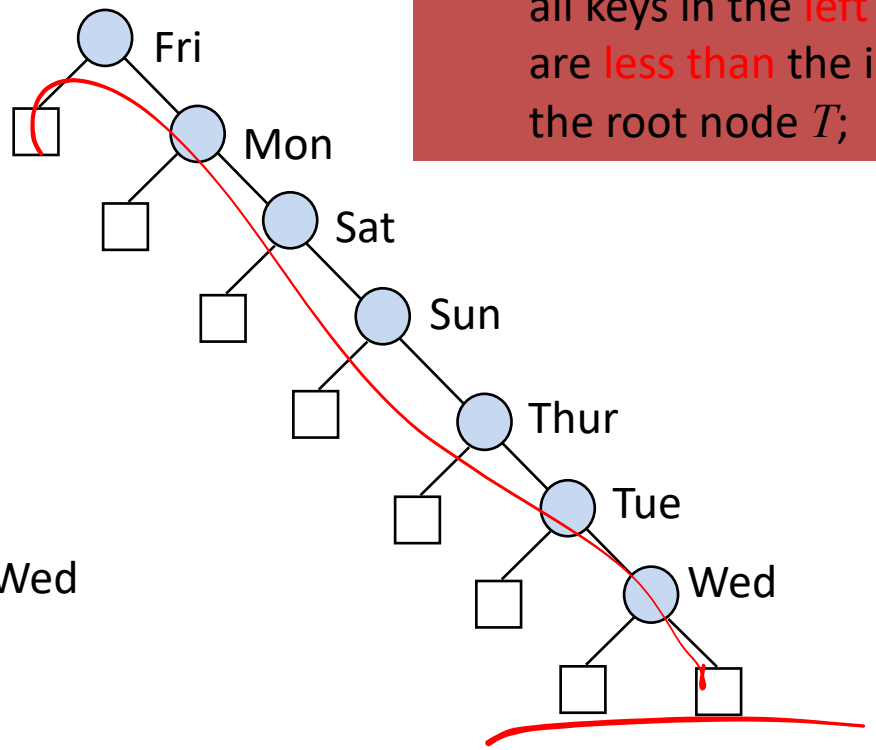


- Furthermore, efficient searching is possible using the *binary search technique*
  - search time is $O(log_2 n)$

Binary Search Trees

It should be noted that several binary search trees are possible for a given data set, *e.g.,* consider the following tree:



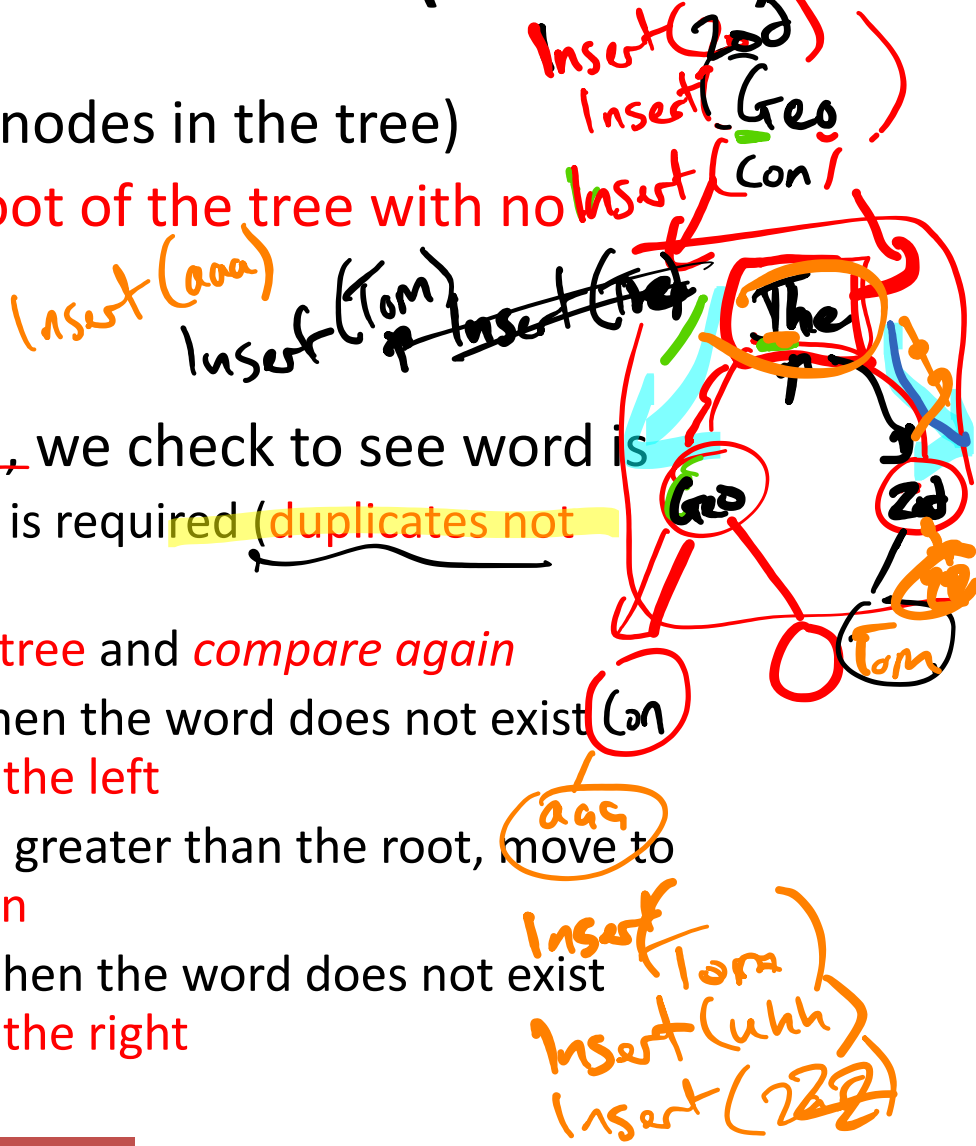


all keys in the left subtree of T are less than the identifier in the root node T;

Construct a binary search tree (of words)

Initially, the tree is null (, i.e., no nodes in the tree)

- The first word is inserted as root of the tree with no children



On insertion of the second word, we check to see word is

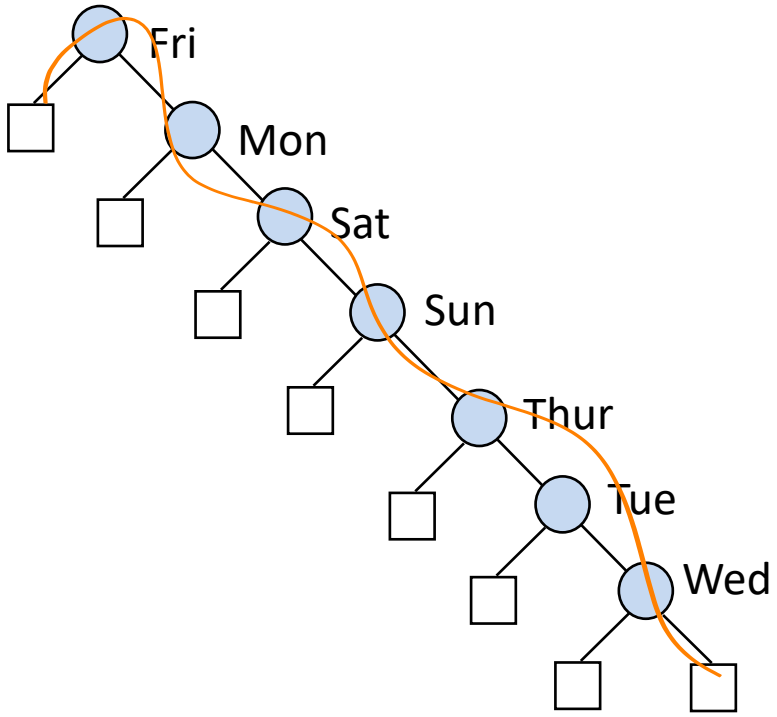
- the same as root, no further action is required (duplicates not allowed)
- less than root, move to the left subtree and *compare again*
- If the left subtree does not exist, then the word does not exist and it is inserted as a new node on the left
- If, on the other hand, the word was greater than the root, move to the right subtree and compare again
- If the right subtree does not exist, then the word does not exist and it is inserted as a new node on the right

Root == key in root

Binary Search Trees

- The point here is that the structure of the tree depends on the order in which the data is inserted in the list

- If the words are entered in sorted order, then the tree will degenerate to a 1-D list



BST Operations

- *Insert*: E x BST $\rightarrow$ BST :

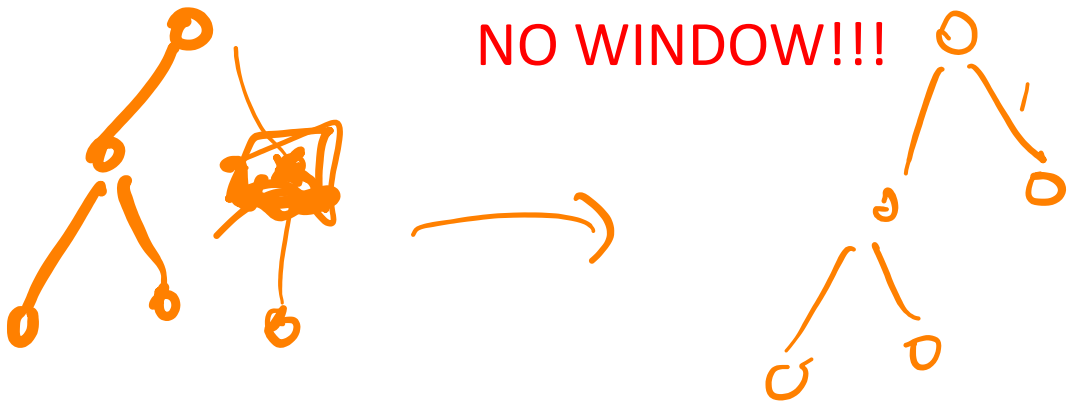
  The function value $Insert(e,T)$ returns the BST T with the element e inserted as a leaf node; if the element already exists, no action is taken

NO WINDOW!!!

BST Operations

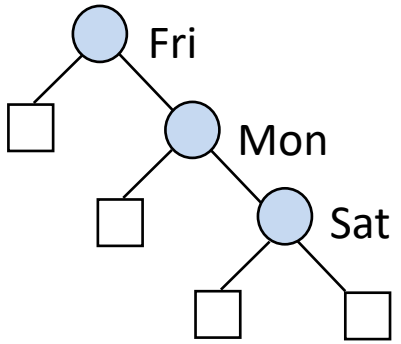
- *Delete*: E x BST $\rightarrow$ BST :

  The function value $Delete(e, T)$ returns the BST T with the element e deleted; if the element is not in the BST, no action is taken.



Implementation of *Insert*(*e*, *T*)

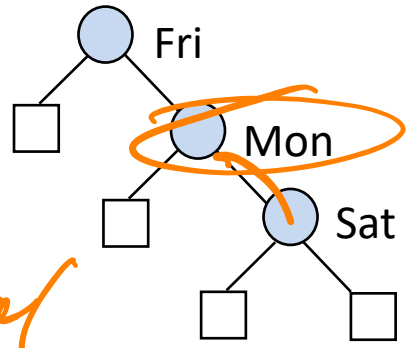
- If T is empty (i.e. T is NULL)
  - create a new node for e
  - make T point to it
- If T is not empty
  - if e < element at root of T
    - Insert e in left child of T: *Insert(e, T(1))*
  - if e > element at root of T
    - Insert e in right child of T: *Insert(e, T(2))*

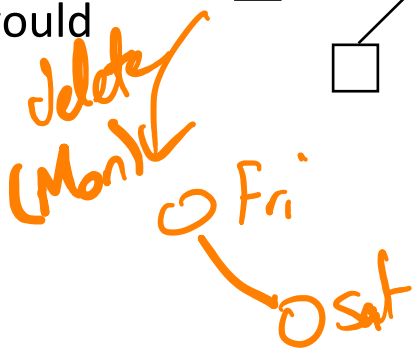


Implementation of *Delete*(*e*, *T*)

First, we must locate the element e to be deleted in the tree

- if e is at a leaf node
  - we can delete that node and be done
- if e is at an interior node at w
  - we **can't** simply delete the node at w as that would disconnect its children
- if the node at w has only one child
  - we can replace that node with its child

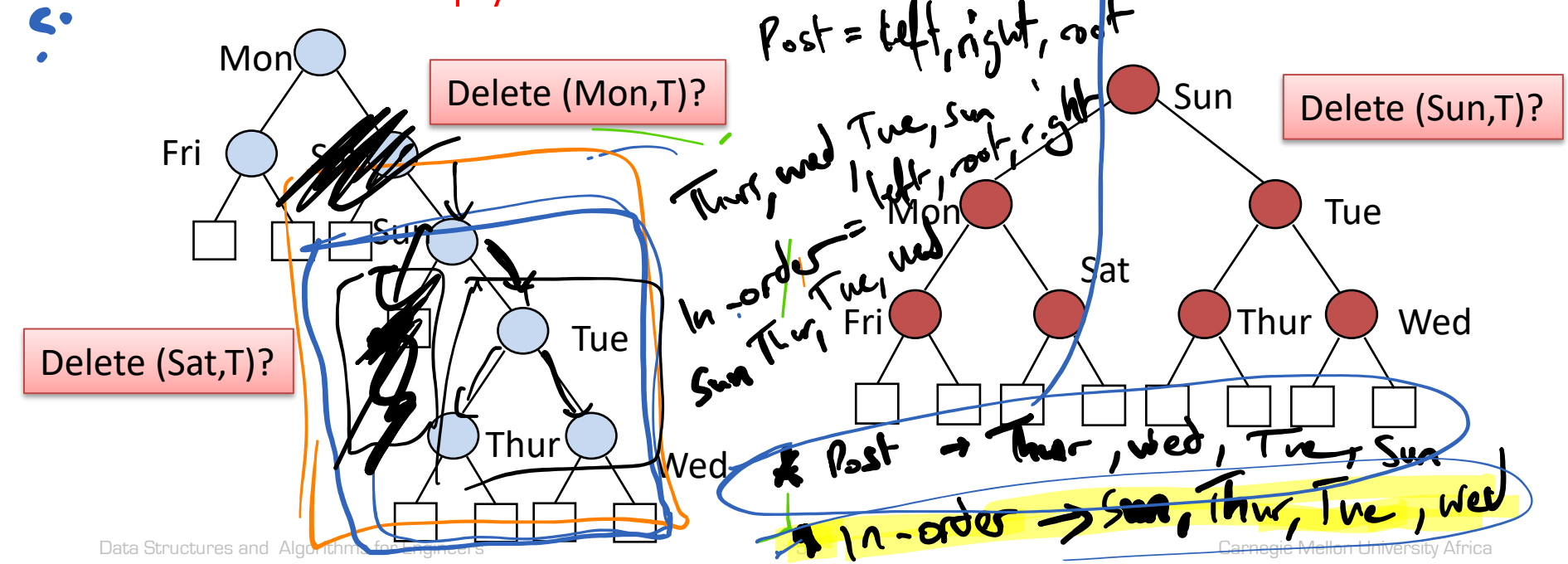




Implementation of *Delete*(*e*, *T*)

– if the node at w has two children

- we replace the node at w with the lowest-valued element among the descendents of its right child
- this is the left-most node of the right tree
- It is useful to have a function DeleteMin() which removes the smallest element from a non-empty tree and returns the value of the element removed



Today!

- Tree Traversal
  - In-order, Pre-order, post-order

In Order
left -> root -> right

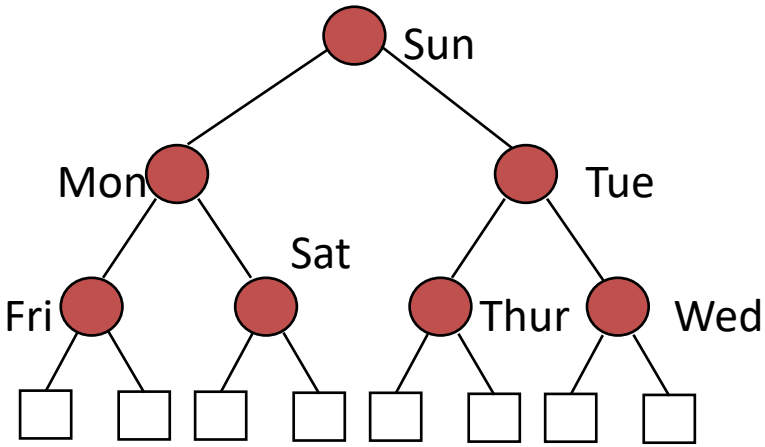
Pre Order
root -> left -> right

Post Order
left -> right -> root

Binary Search Tree

- A binary tree which when traversed in-order, the keys are sorted fashion
- Applications of BST
- Insert/Deletes to a BST

Delete (Sun,T)?



To delete a node, T, with two children

Replace T with *left-most node in right sub-tree!"N*

Next Class

Height-Balanced Trees: AVL Trees