Binary Search Tree

IT5003: Data Structures and Algorithms
(**AY2019/20 Semester 1**)

Lecture Overview

- Motivation
  - Table ADT
- Binary Search Tree
  - Definition
  - Major Operations

Motivation: **Why Table?**

- List ADT, Stack ADT and Queue ADT manipulate data in a collection by **index (position)** of the data
  - The position is **implicit** in the case of stack and queue
- In real life, it is more common to manipulate item based on the **value** of the data
  - e.g. "Look for the student record with matriculation number ***A0201234X*** ", "Remove the library record for the book ***Happy Coding***"
  - Value used to locate a specific record is commonly known as **key**

Table ADT: **Real Life Example**

- Phone books
  - Key = Phone#
- Street directories
  - Key = Shop name
- Dictionaries
  - Key = word
- Class schedule
  - Key = Module Code

Key	**Data**
911	Emergency Call
61345	Uncle Soo's Office
62768	Technical Service
41616	Campus Security
...	...

- Observation:
  - The key is usually **unique**
  - There can be one or more pieces of information attached with each **key**

Table ADT: Operations

- The defining operations of a Table ADT are:
  - **insert(key, data)**
    - Add a pair of **(key, data)** into the table
  - **delete(key)**
    - Delete the key and its associated data from the table
  - **data = search(key)**
    - Find key in the table and return its associated data

- Since position is not indicated explicitly:
  - The implementation is free to organize the information for best performance

Table ADT: **Our choices so far….**

- Using covered data structures to implement table, we can achieve the following efficiency:

	Unsorted Array/List	Sorted Array	Sorted LinkedList
insert	**O(1)**	O(N)	O(N)
delete	**O(N)**	**O(N)**	**O(N)**
search	O(N)	**O($\log_2$ N)**	O(N)

- Note:
  - The cost for sorted linked list includes the traversal cost

Table ADT: **Can we do better?**

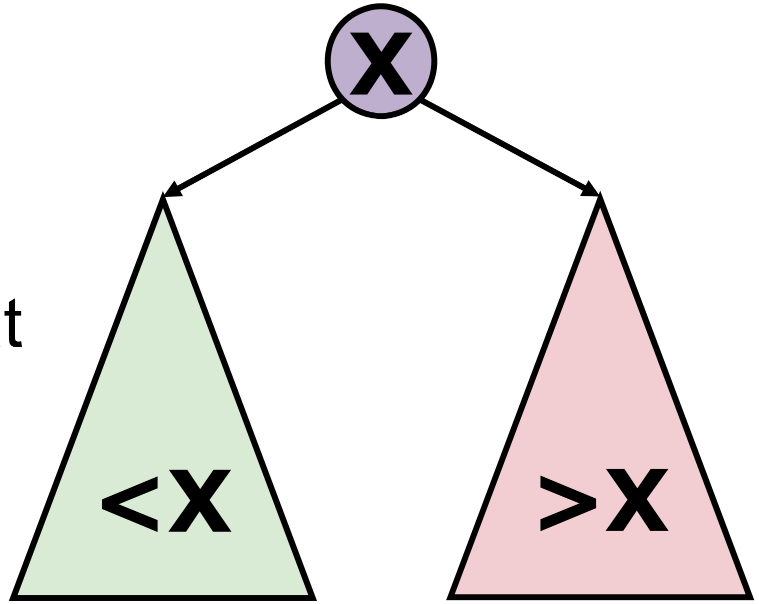
- Data structure we learned so far is not very good at table operations ☺

- Some new data structures to help:
    a. BST and AVL Tree
    b. Hashing

BINARY SEARCH TREE (BST)

Definition: **Binary Search Tree**

Binary Search Tree (BST)

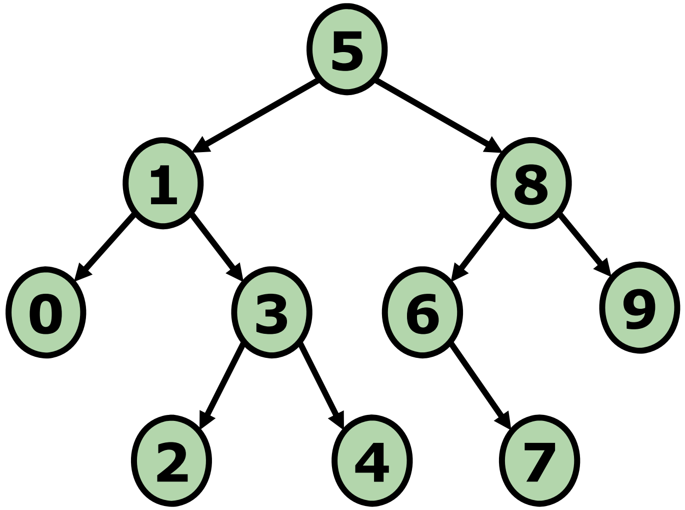


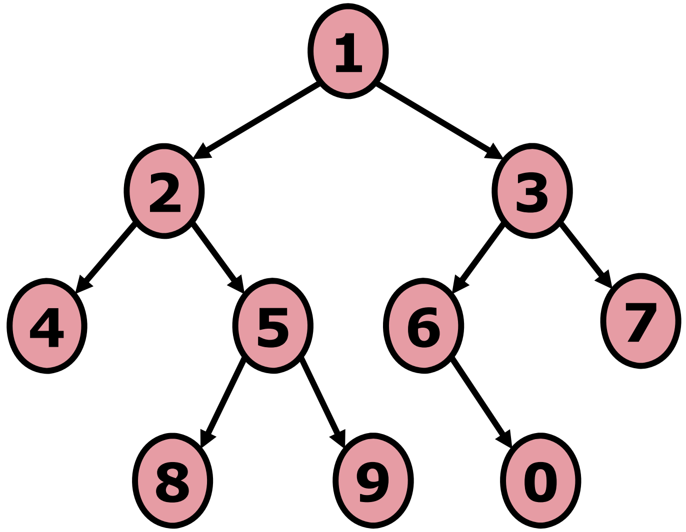
- is a **binary tree** that is
  1) Empty **OR**
  2) A node with key **k** such that
     - Nodes in left subtree have **keys < k**
     - Nodes in right subtree have **keys > k**

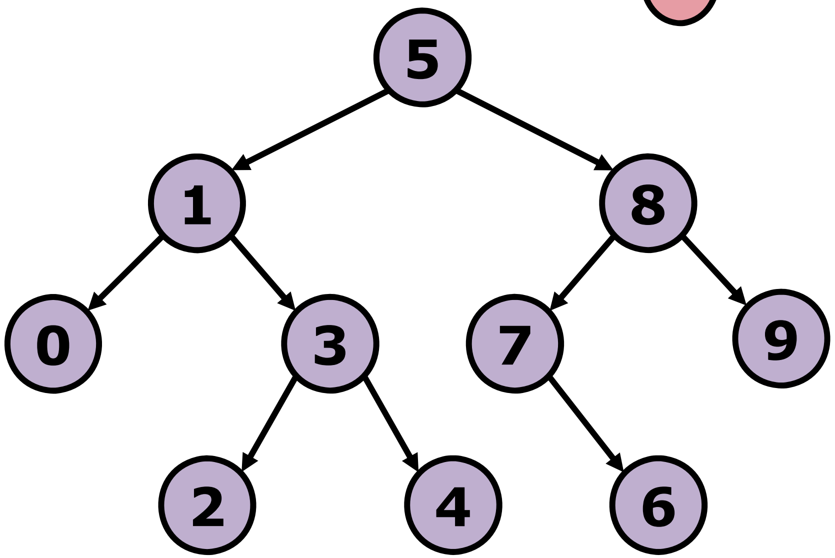
- BST is a variant of binary tree

  ➔ all binary tree definitions/operations are applicable, e.g. height, size, complete, etc…

Check: **Which Binary Tree is BST?**

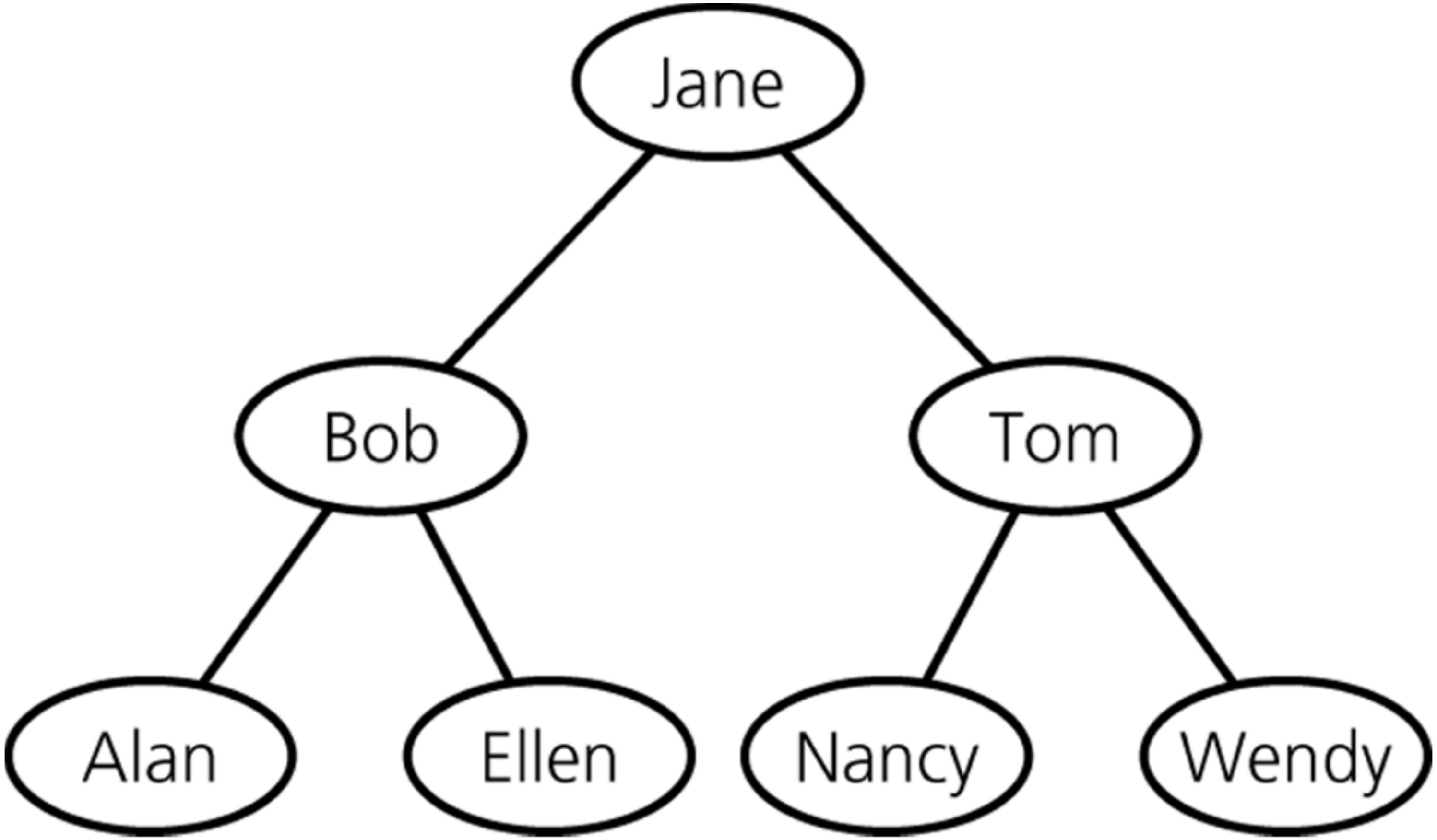






BST: **Other Data Type**

BST of names (i.e. strings):



Oh, they are mostly recursive ☺

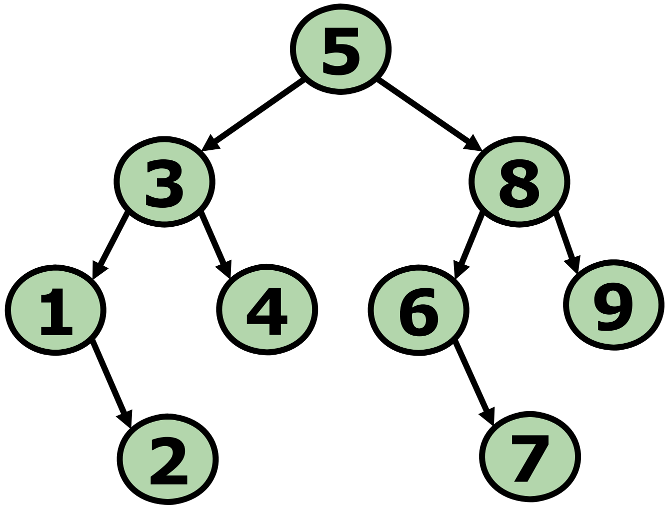
BST OPERATIONS

General Guideline

- Most of the BST operations are recursive
  - Iterative version can be written but usually harder to understand
- We use reference/pointer-based pseudo-code
  - Array based code is similar!
- Each `TreeNode` contains:
  - **Key** and **Data**
  - Only **Key** is used for manipulation
  - We use T➔key and T➔data in the pseudo-code

Finding Minimum Element

```
def findMin( T ):
    if T is empty:
        return None #or error

    if T➔leftT is empty:
        return T

    return findMin( T➔leftT )
```



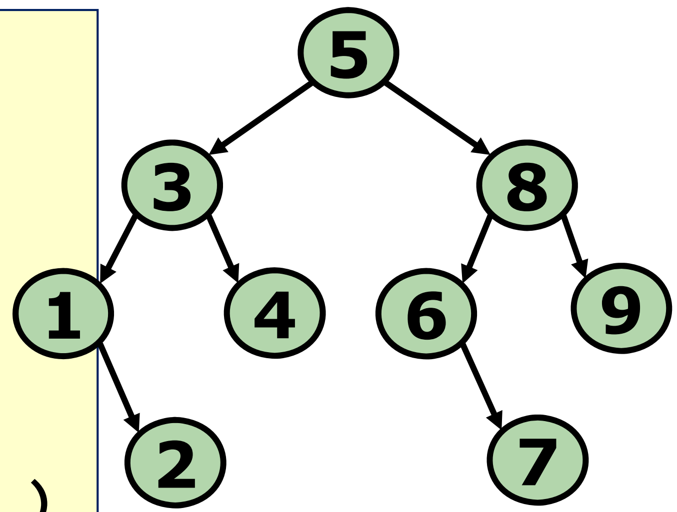
Q: How to find maximum values?

Q: How to find top-k (or bottom-k) values?

e.g. find top-3 values

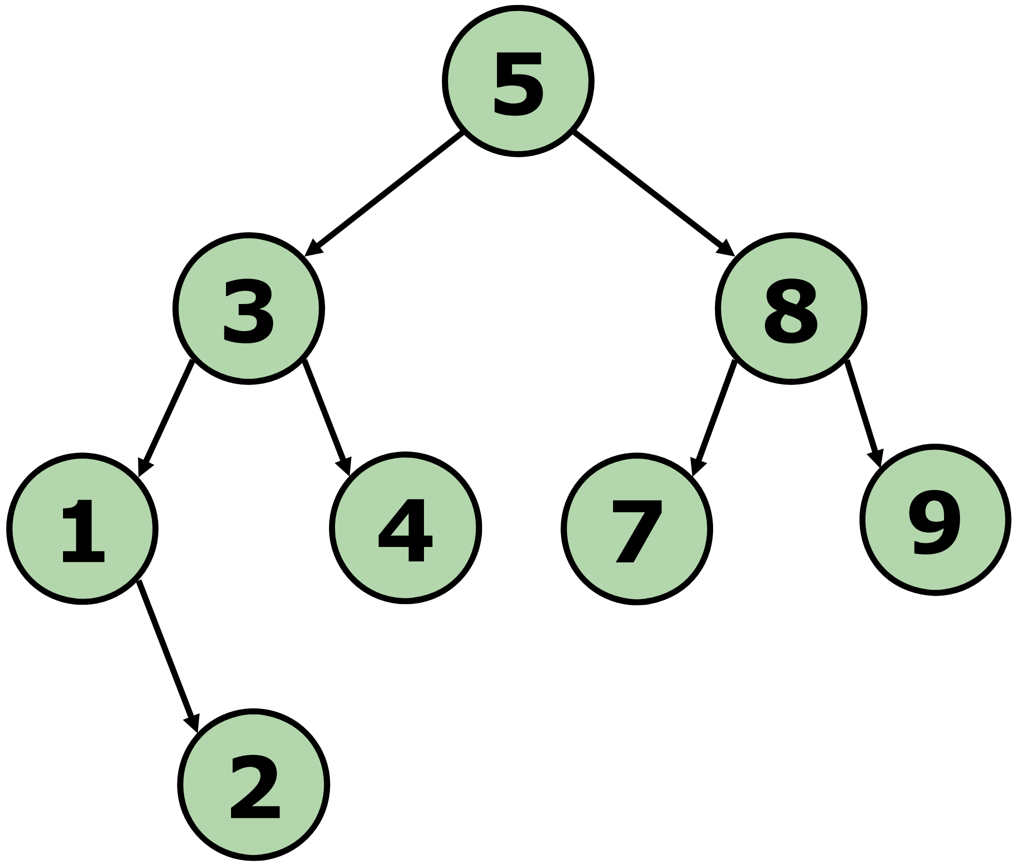
Search for a Key

```
def search( T, key ):
    if T is empty:
        cannot find key!

    if T➔key == key:
        return T➔data
    elif T➔key < key:
        return search( T➔rightT, key )
    else:
        return search( T➔leftT, key )
```

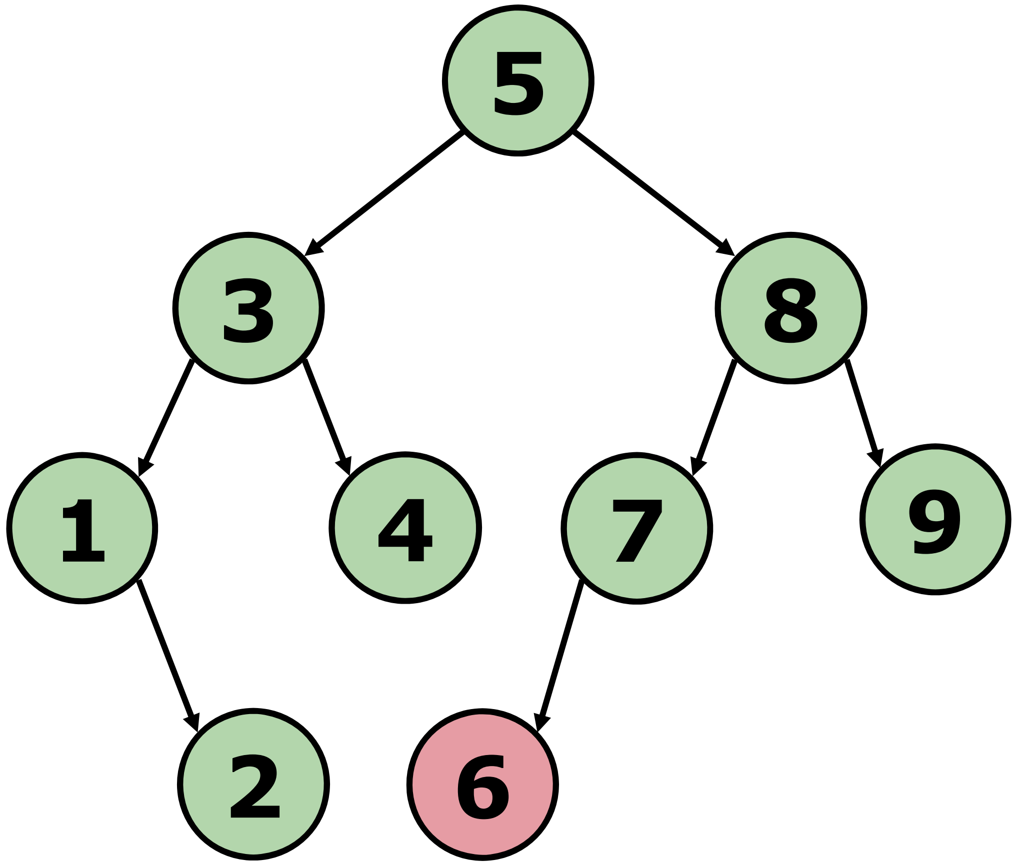


- **Question:**
  - See the similarity with **binary search**?
  - What is the efficiency of BST search then?

Insertion Check: **How to Insert 6?**

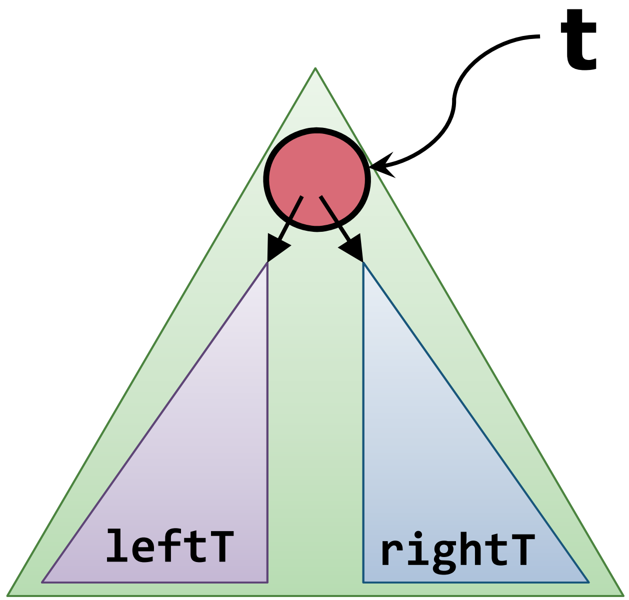


Insertion Check: **How to Insert 6?**



Insertion: Idea

- Behaviour of insertion:
  a. Takes key and data as parameter
  b. Returns the ***modified binary search tree*** after insertion
- Base case:
  a. Insert into empty BST
  b. Return BST with one node
- General cases:
  - Compare **key** with **x**
  - **key < x**: insert into **leftT**
  - **key > x**: insert into **rightT**



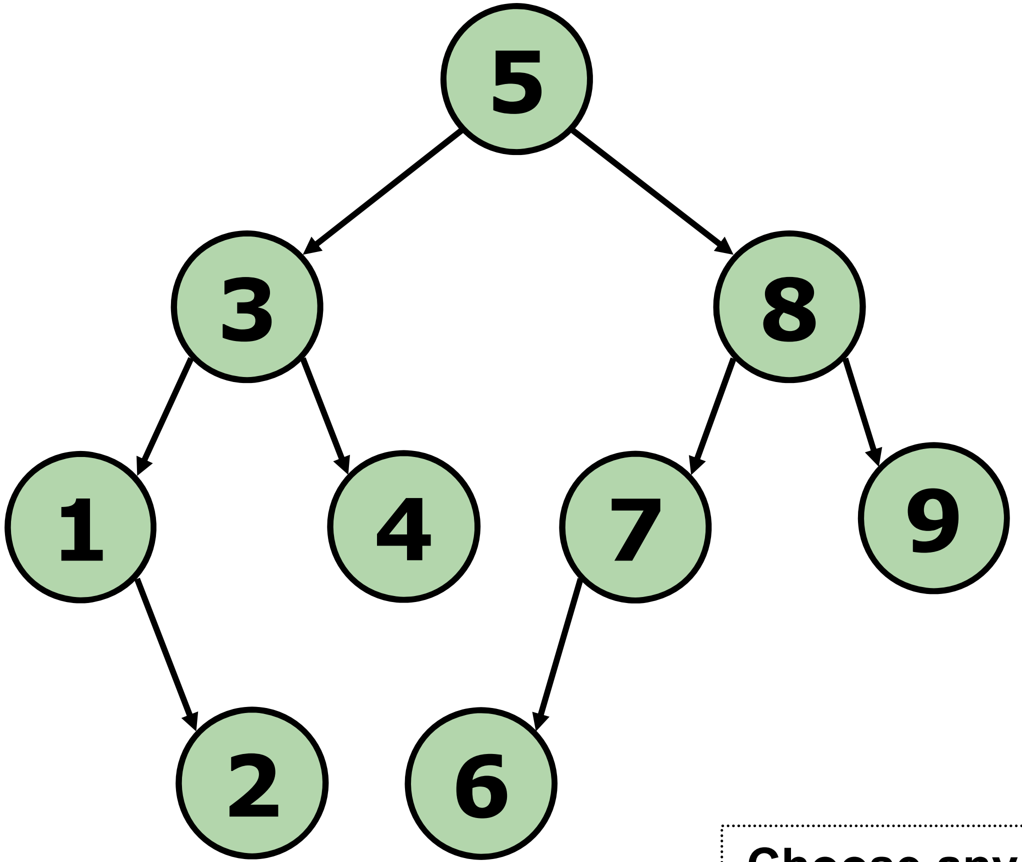
Insertion: Pseudo-Code

```
def insert( T, key, data ):
    if T is empty:
         return TreeNode( key, data )

    if T➔key == key:
        Duplicate Key Error!

    elif T➔key < key:
        T➔rightT = insert( T➔rightT, key, data )

    else:
        T➔leftT  = insert( T➔leftT, key, data )

    return T
```

Create a new TreeNode

Pay attention to the assignment

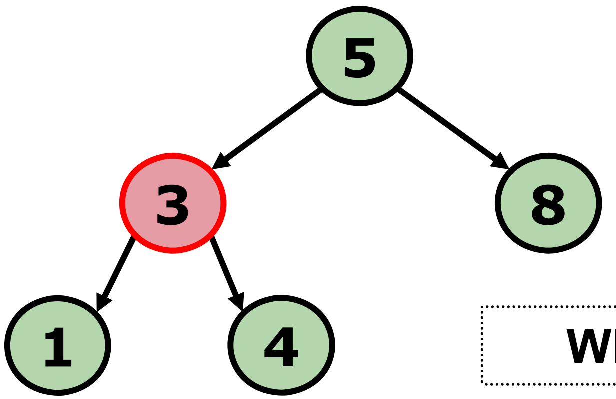
Deletion Check: **How to delete?**



Choose any node to delete, how do we make sure it is still a BST afterward?

Deletion: Idea

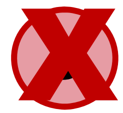
- Behaviour of deletion:
  a. Takes key as parameter
  b. Returns the ***modified binary search tree*** after deletion
- Deletion is more involving:
  - Simple when the target node is a leaf node
  - What if the target node is an internal node?



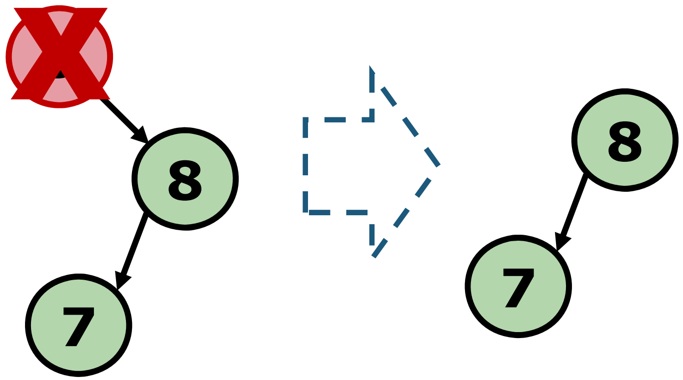
What if we delete "3"?

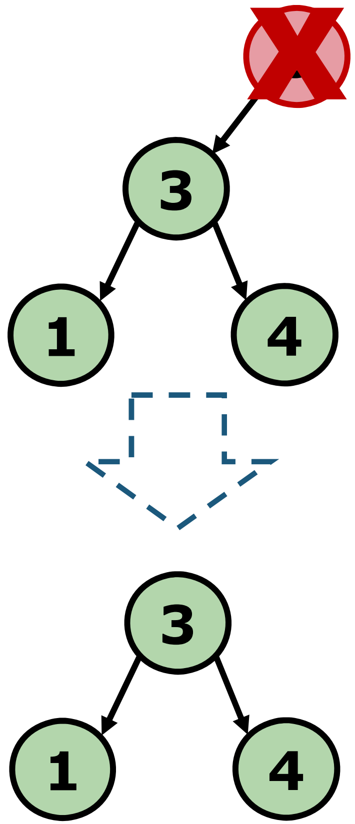
Deletion: 3 Different Cases (1/2)

- **Case 1**: Leaf node
  - return empty tree



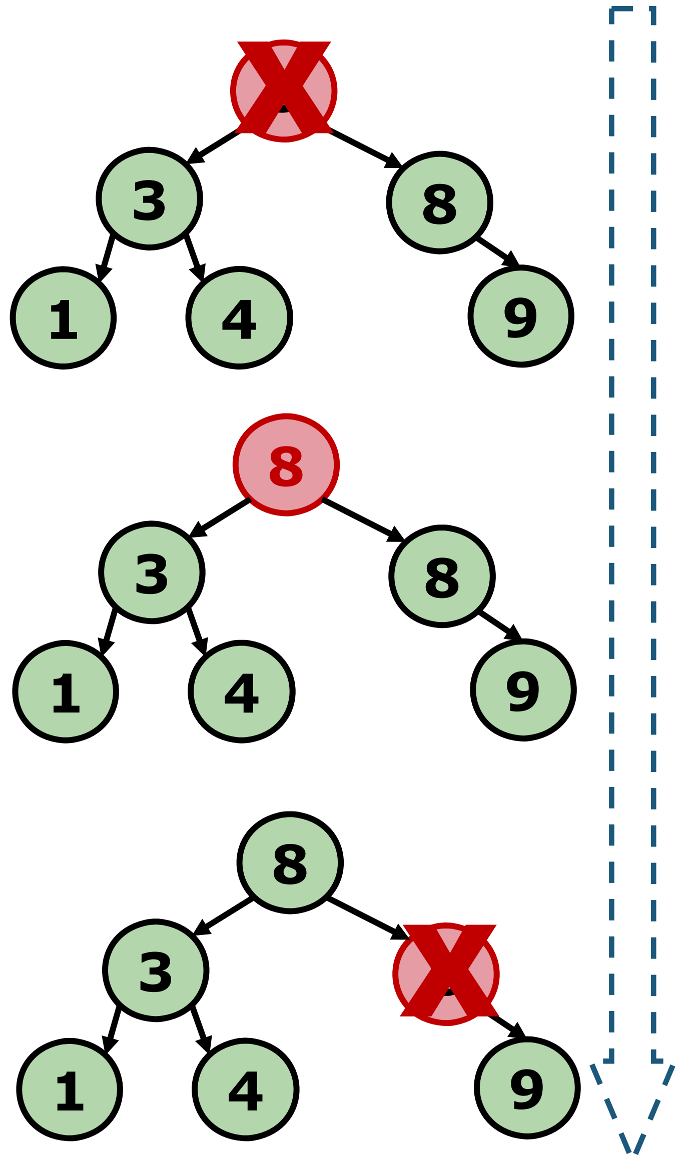
- **Case 2**: node with 1 child
  - return the child as result





Deletion: 3 Different Cases (2/2)

- **Case 3**: node with 2 children
  1. Get immediate successor **S**
     - key that is immediately after in the sorted sequence
  2. Replace target key with **S**
  3. Delete **S** from child tree



- **Question:**
  - What if "8" has two children?

Deletion: Pseudo-Code

```
def delete( T, key ):
    if T is empty:
        Cannot Find Key Error!
    if T➔key < key:
        T➔rightT = delete( T➔rightT, key )
    elif T➔key > key :
        T➔leftT  = delete( T➔leftT, key )
    else:
        if T has no child:
            return Empty Tree
        elif T has left child ONLY:
            return T➔leftT
        elif T has right child ONLY:
            return T➔right
        else:
            successor = findMin( T➔rightT)
            T➔key = successor➔key
            T➔data = successor➔data
            T➔rightT = delete( T➔rightT, successor➔key )
        return T
```

Case 1

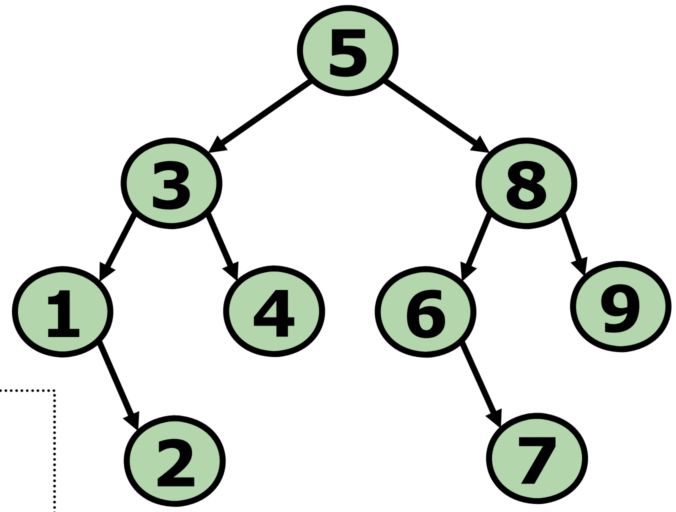
Case 2s

Case 3

BST Traversals

- As mentioned, all binary tree traversals are applicable to BST:
  - **Pre-, In-, Post- Order**
  - **Level Order**

Try it out!



- Some traversals are more useful than others:
  - **In-order traversal**: What do you get?
  - **Pre-order traversal**: Useful for saving / restoring BST

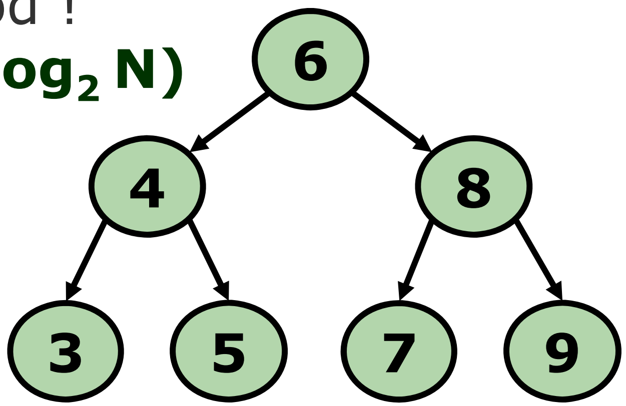
Just how good is the BST?

BST COMPLEXITY

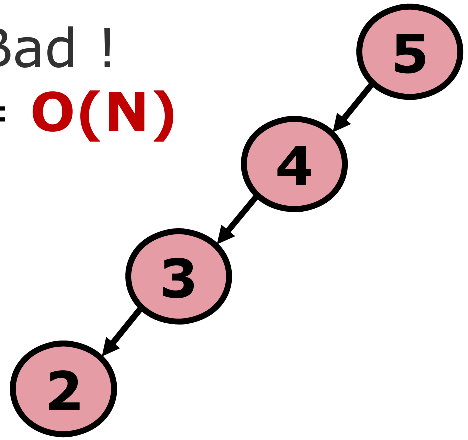
Binary Search Tree: **Analysis**

- Operations are dependent on **BST Height**
  - *`findMin`* = O(h)
  - *`search`* = O(h)
  - *`insert`* = O(h)
  - *`delete`* = O(h)
- Height can differs **greatly:**

Good !
h = $O(\log_2 N)$



Bad !
h = $O(N)$



Binary Search Tree: **Complexity**

- The BST height:
  - $O(\log_2 N)$: **Best Case**
  - $O(\log_2 N)$: **Average Case**
  - $O(N)$: **Worst Case**

- Informal argument for Average Case:
  - If we insert N numbers in **random order**, it is much more often that we can get a more balanced tree than a badly skewed one at the end
- Proving average case complexity is beyond the scope of this course ☺

BST APPLICATIONS

TreeSort: Another Sorting Algorithm

- **Key Idea:**
  1. Take the unsorted numbers and insert into a BST
  2. Perform an in-order traversal after all numbers are inserted

- **Complexity:**
  - Average case: O(n * log n)
  - Worst case: O(n^2)

BST in Storage

- Algorithms for saving a binary search tree
  1. Saving a binary search tree and then restoring it to its original shape
     a. Save the **preorder traversal** to a file
     b. When restoring the tree: just read from the file and insert

  2. Saving a binary search tree and then restoring it to a **balanced shape**
     a. Uses **inorder traversal** to save the tree to a file
     b. When restoring the tree:
        i. Read the middle item as root
        ii. Then recursively construct left / right subtrees by the left/right halves of sequence

Summary

- Binary Search Tree property
- Major operations
  - Find the minimum, search, insert, delete
  - Traversals
- Complexity of BST operations
- BST Applications

END