Tree

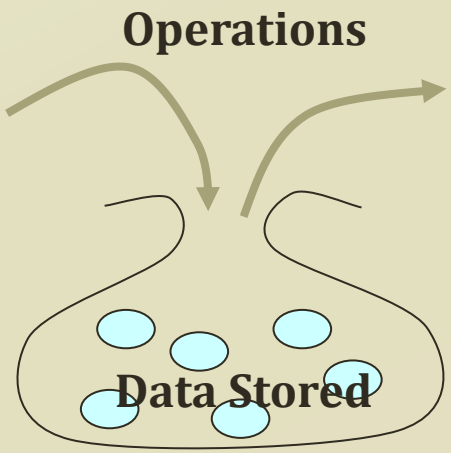
Il-Chul Moon
Dept. of Industrial and Systems Engineering
KAIST
icmoon@kaist.ac.kr

Weekly Objectives

- This week, we study the tree data structure. Particularly, we will focus on the structure and the operation of the binary search tree.
- Objectives are
  - Memorizing the definitions, the terminologies and the characteristics of trees
  - Understanding the structures of trees
  - Understanding the structure and the operations of a binary search tree
    - Insert, search, delete operations
    - Tree traversing operations
      - Depth first search
        - In-order, post-order, pre-order sequences
      - Breadth first search
      - Level order search
  - Understanding the performance of binary search tree

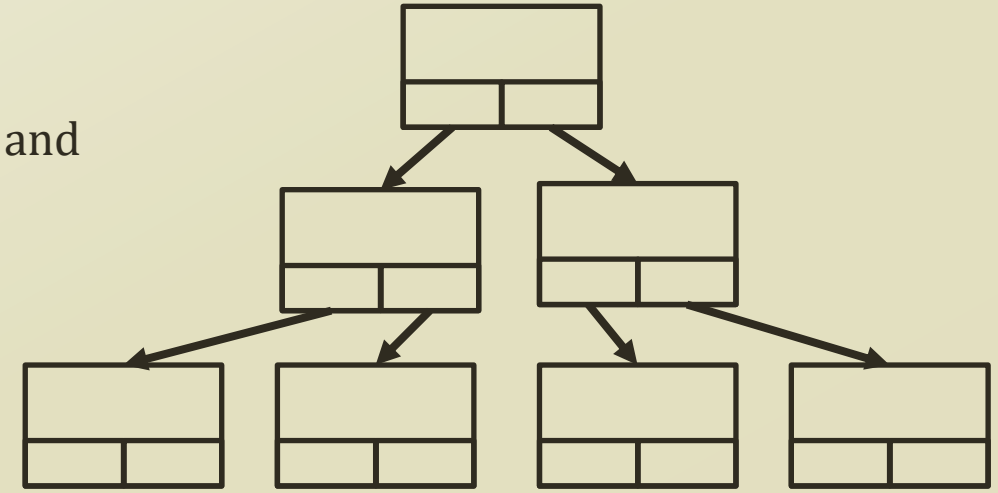
Detour: Abstract Data Types

- An abstract data type (ADT) is an abstraction of a data structure
  - An ADT specifies:
    - Data stored
    - Operations on the data
    - Error conditions associated with operations
- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell orders
  - The operations supported are
    - order buy(stock, shares, price)
    - order sell(stock, shares, price)
    - void cancel(order)
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order



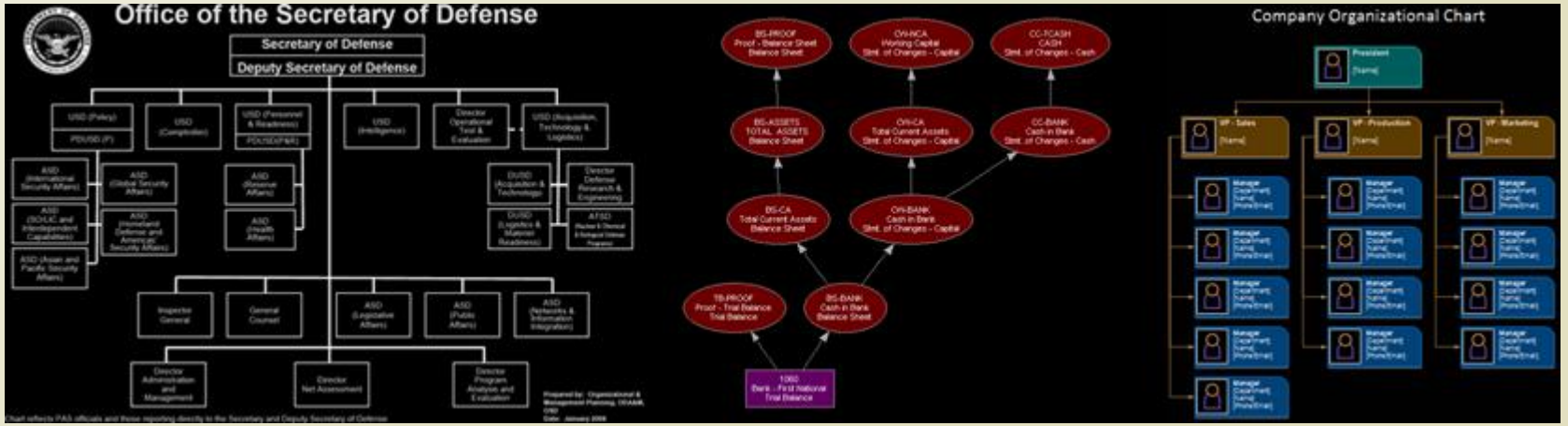
Tree as an abstract data type

- Tree structure
  - Abstract data type
  - Data stored
    - As a tree structure
  - Operations
    - Ordinary data structure operations just as linked lists
      - Insert
      - Delete
      - Search
    - Special searching approaches for trees and networks
      - Traverse



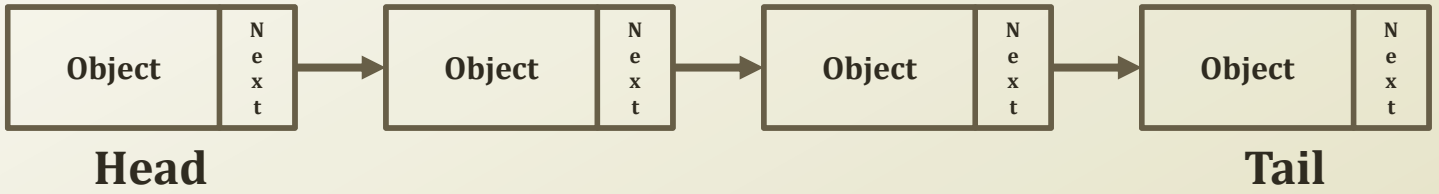
Why do we use trees?

- Because the structure of trees is a good analogy to the various real world structures
  - Corporate structures
  - Group bank accounts
  - Command and control structures
- Why is the structure one of the most favorite structures?
  - A clear approach of *Divide and Conquer*

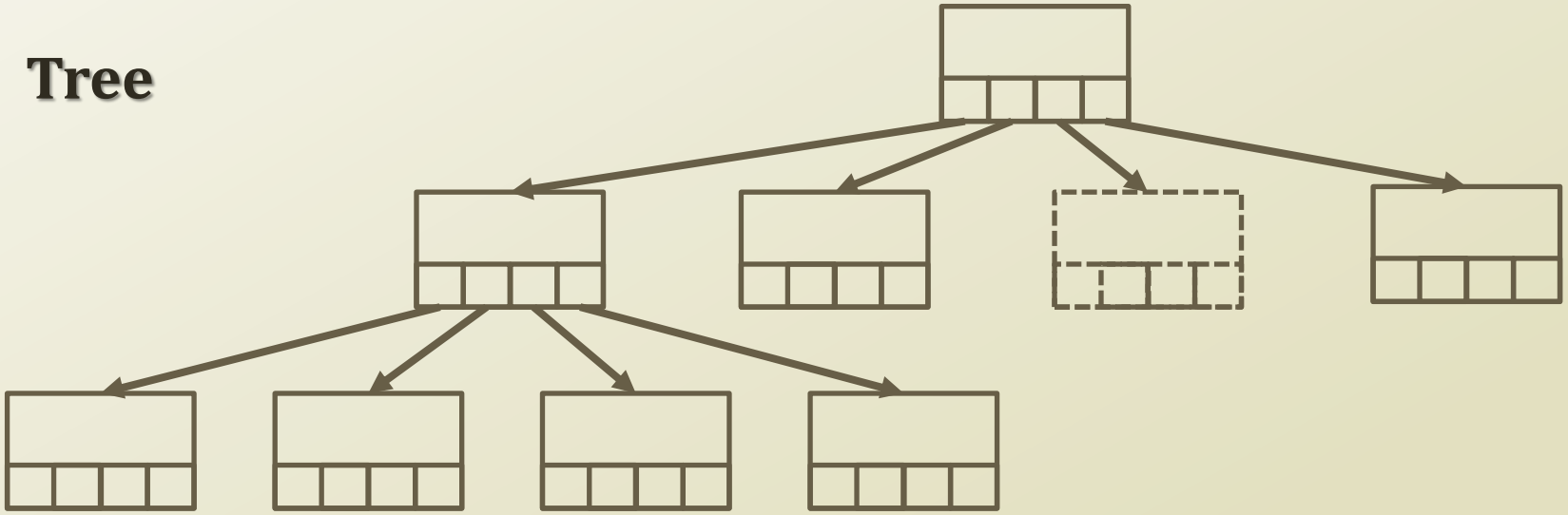


Structure of stored data

Linked List

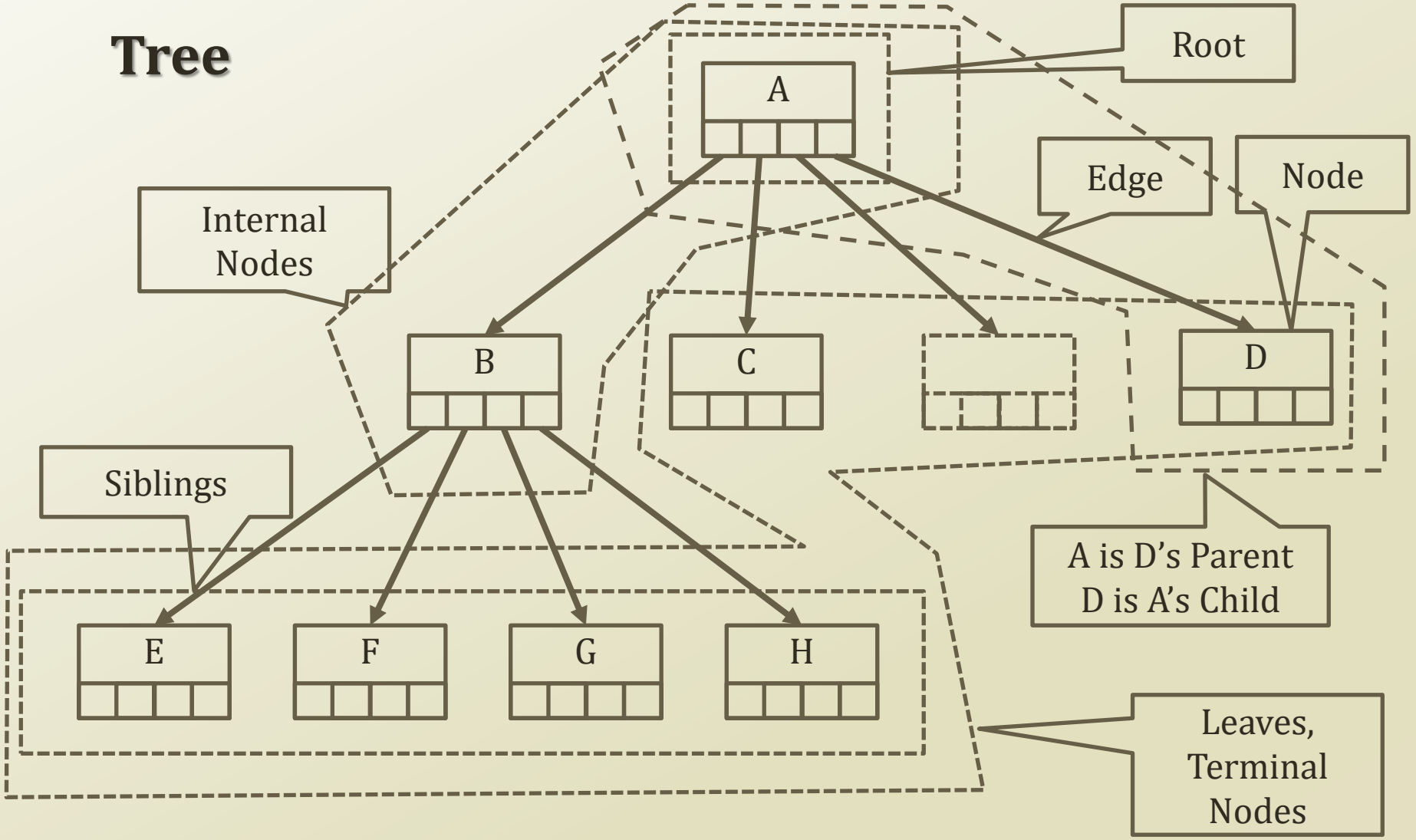


Tree

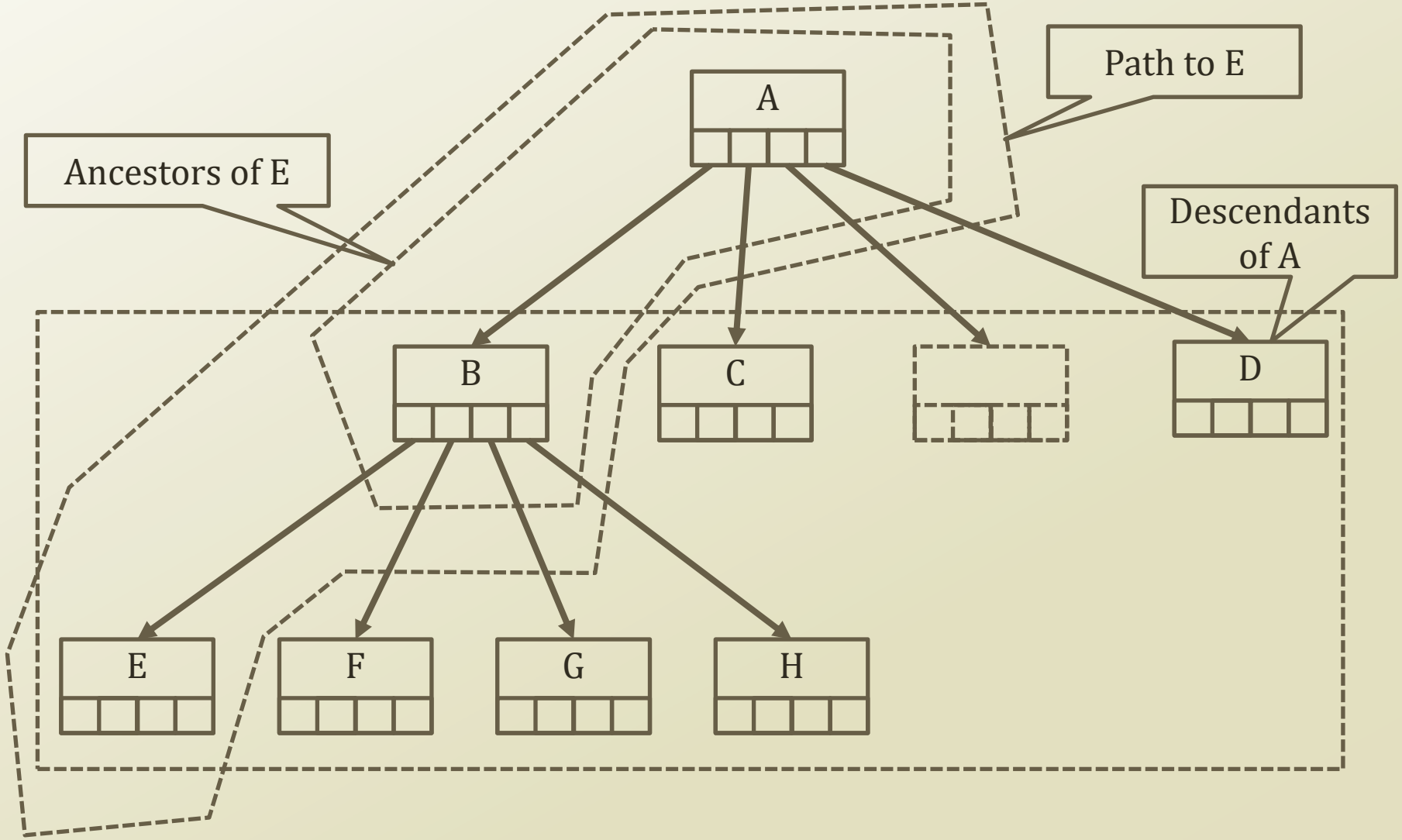


- Nothing but with multiple ***'next'***s
  - Each node has multiple next nodes
  - Particularly, this structure maintains the next nodes as an array or variables

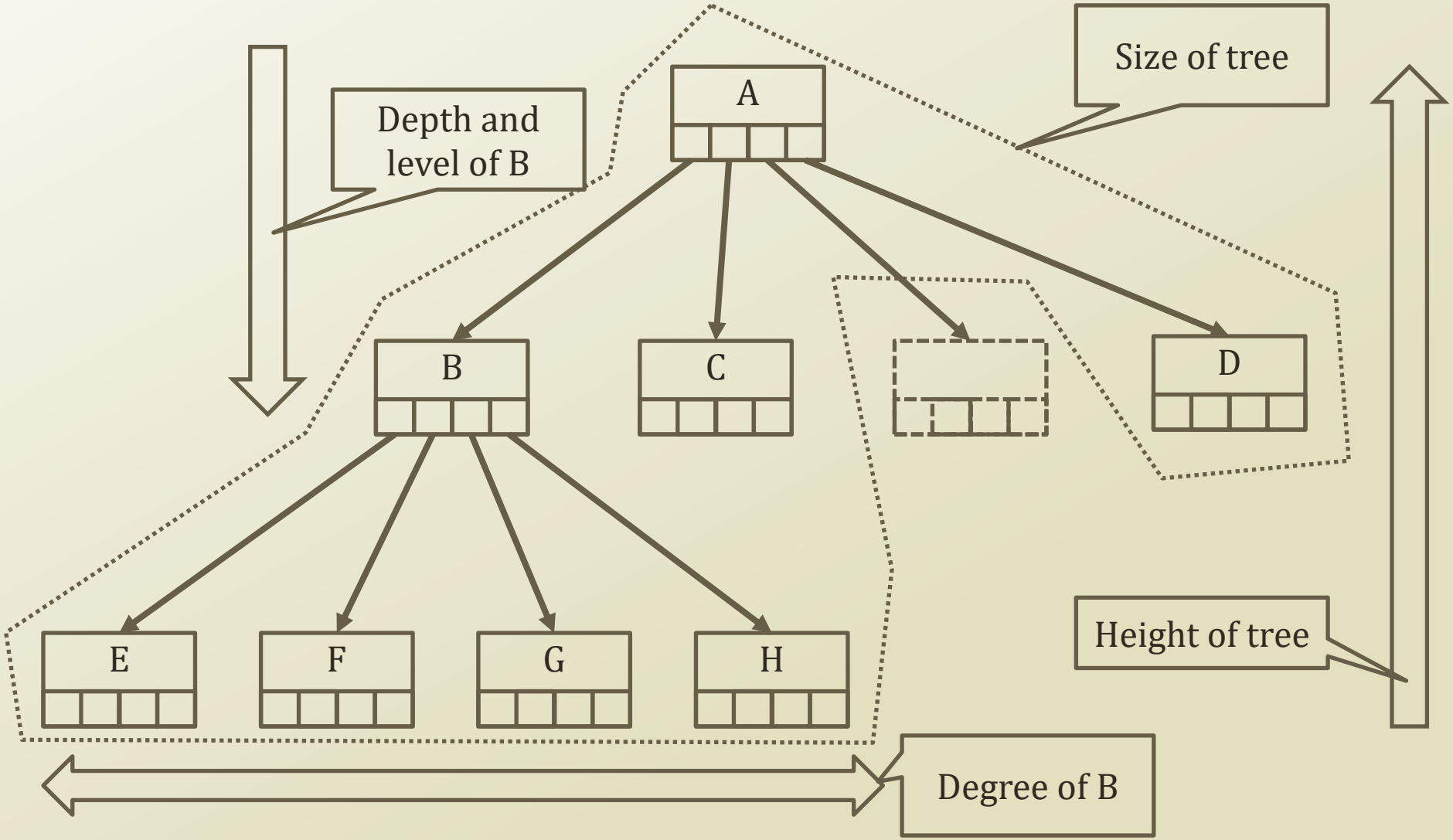
Terminologies of tree structure (1)



Terminologies of tree structure (2)

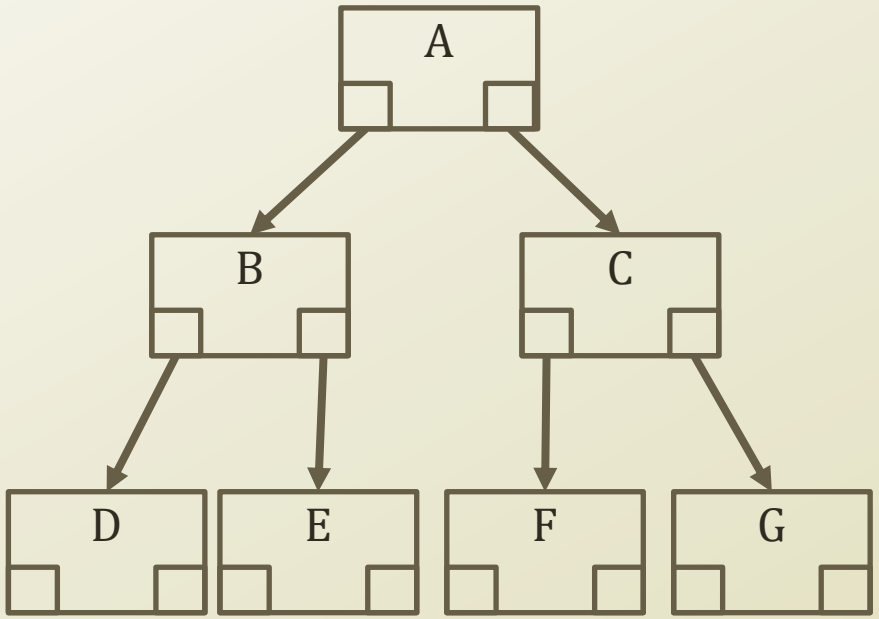


Terminologies of tree structure (3)

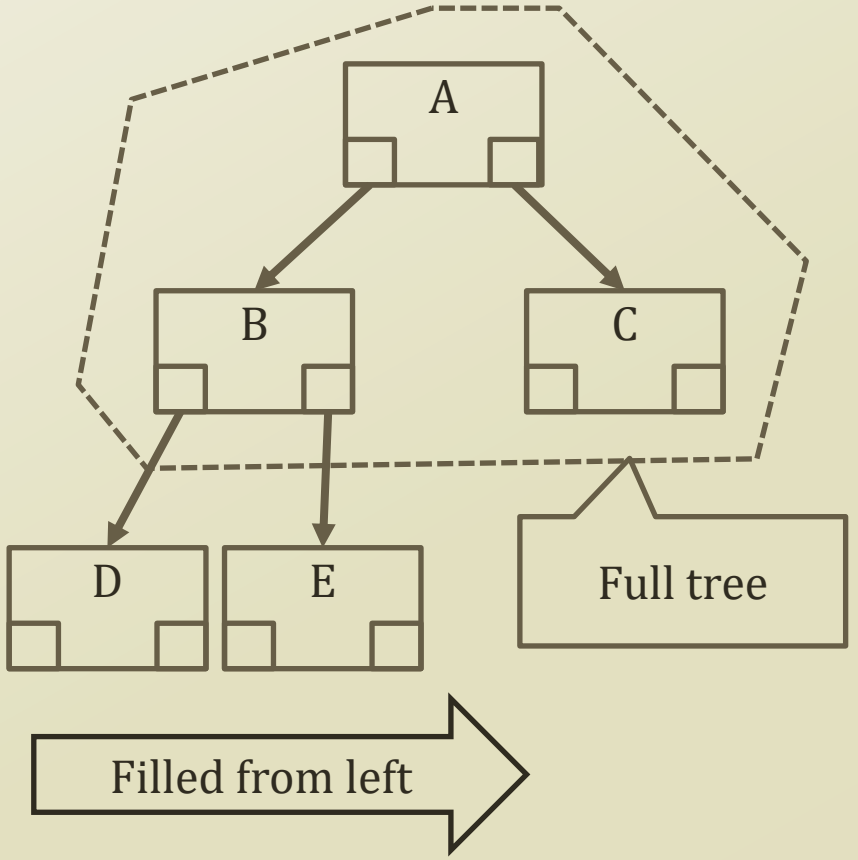


Terminologies of tree structure (4)

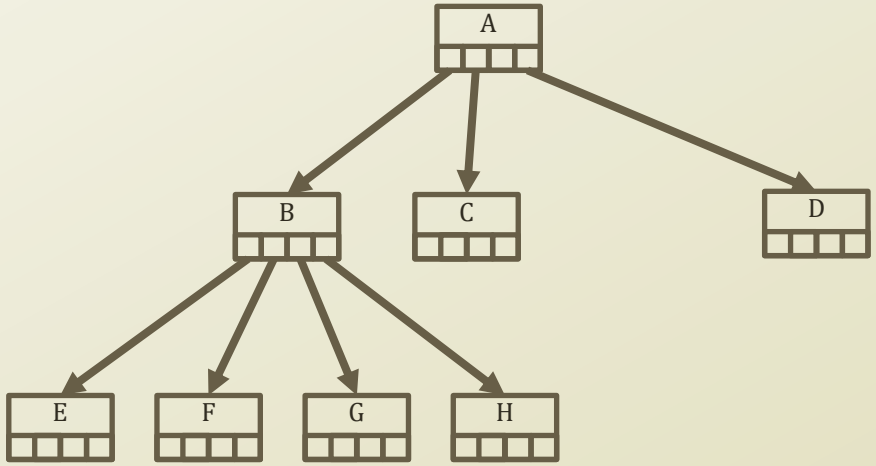
Full Tree



Complete Tree



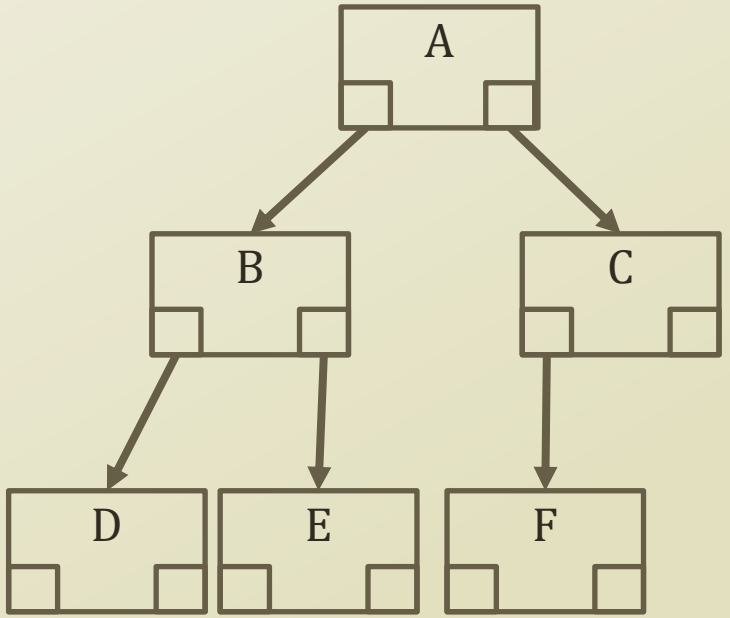
Characteristics of trees



- (Num. of edges) =
  (Num. of nodes) – 1
- Depth of root
  - 0
- (Maximum num. of nodes at level *i* with degree *d*)
  $= d^i$
- (Maximum num. of leaves with height *h* and degree *d*)
  $= d^h$
- (Maximum size of a tree with height *h* and degree *d*)
  $= 1+d+d^2+\ldots+d^h = \frac{d^{h+1}-1}{d-1}$
- (Height of a **complete** tree with size *s* and degree *d*)
  $=\lceil \log_d(s(d-1)+1) \rceil - 1$

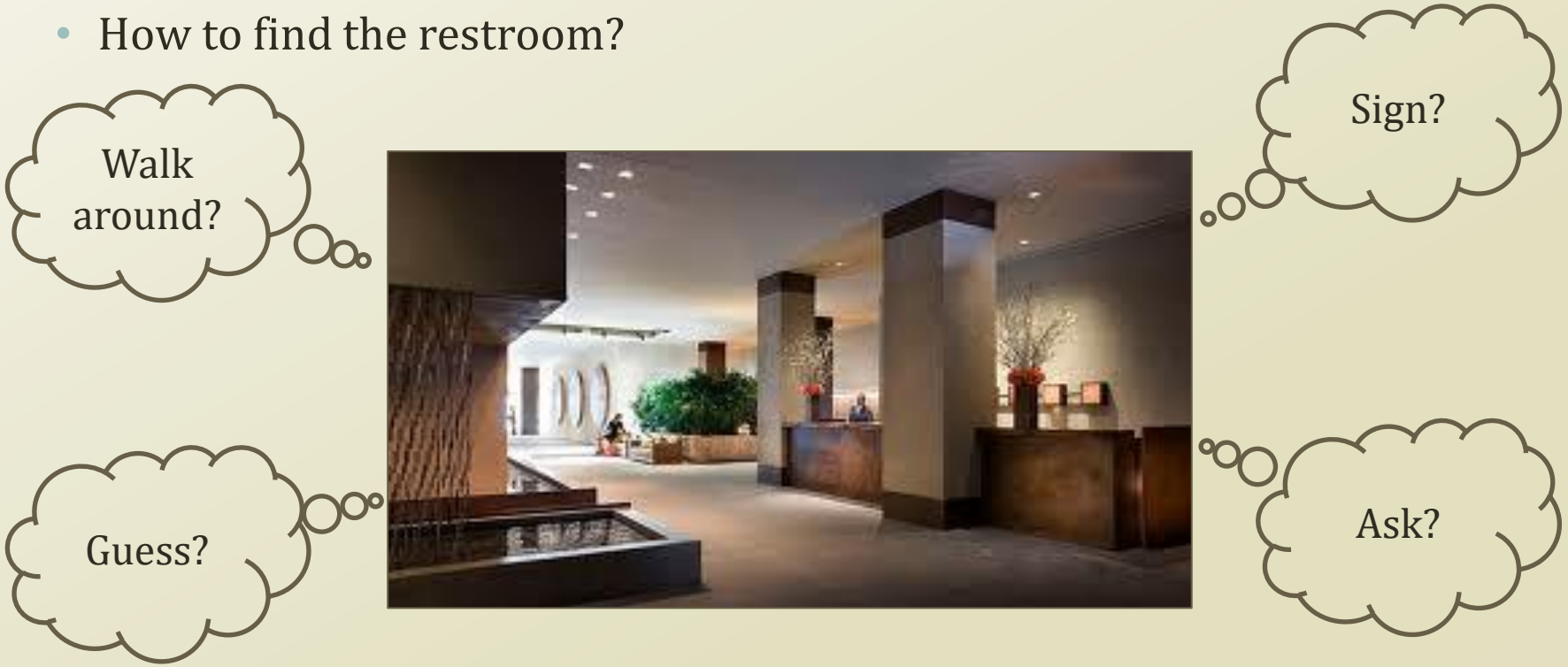
Binary search tree: a simple structure

- Binary tree
  - Tree with degree 2
- Binary search tree
  - Tree with degree 2
  - Tree designed for a fast search of stored data
  - So far, what we have studied the definitions and the characteristics of stored data
  - Now, this is related to the operations
  - **How to perform a faster search?**
- Do you remember what I discussed in the lecture 0?



Detour: Intuitive Analogy – Finding Restroom in Building

- You enter a building to use a restroom
  - This is your fist time in the building.
  - How to find the restroom?

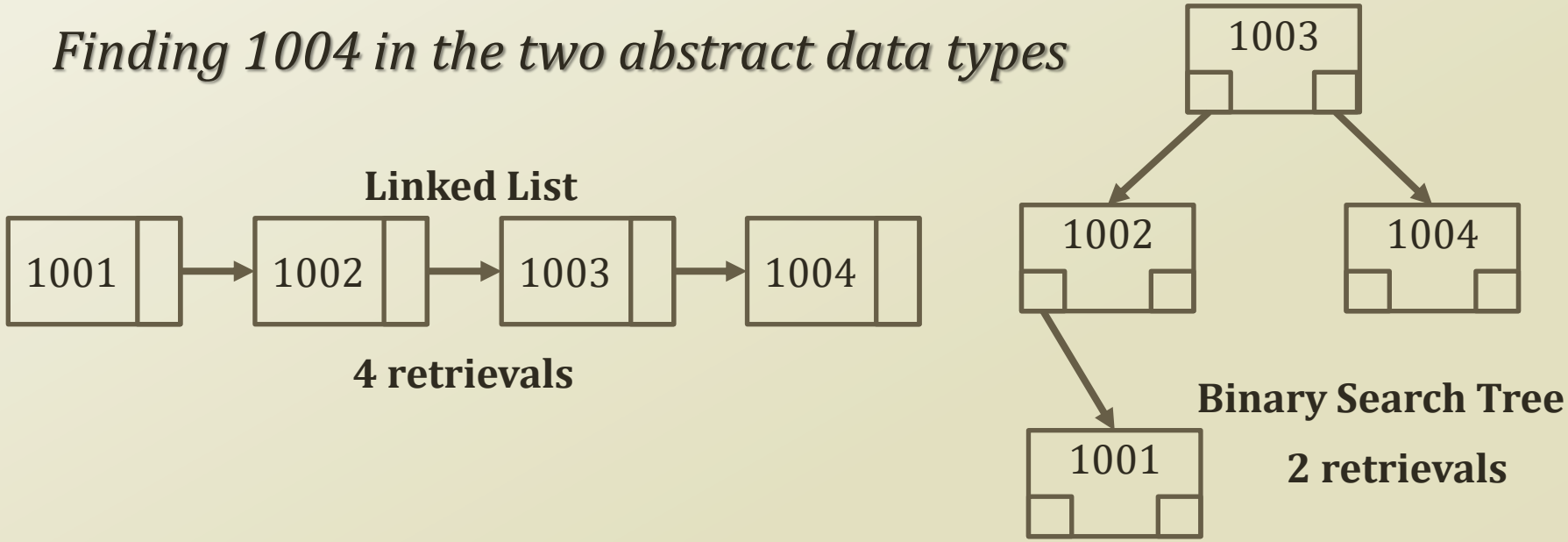


A scenario of using binary search tree

Bank Account Management System

Account #	Name	Amout	Type
1001	Smith	100,000	Simple Interest
1002	Koh	50,000	Compound Interest
1003	Moon	10,000	Simple Interest
1004	Kim	30,000	No Interest

Finding 1004 in the two abstract data types



Implementation of tree node

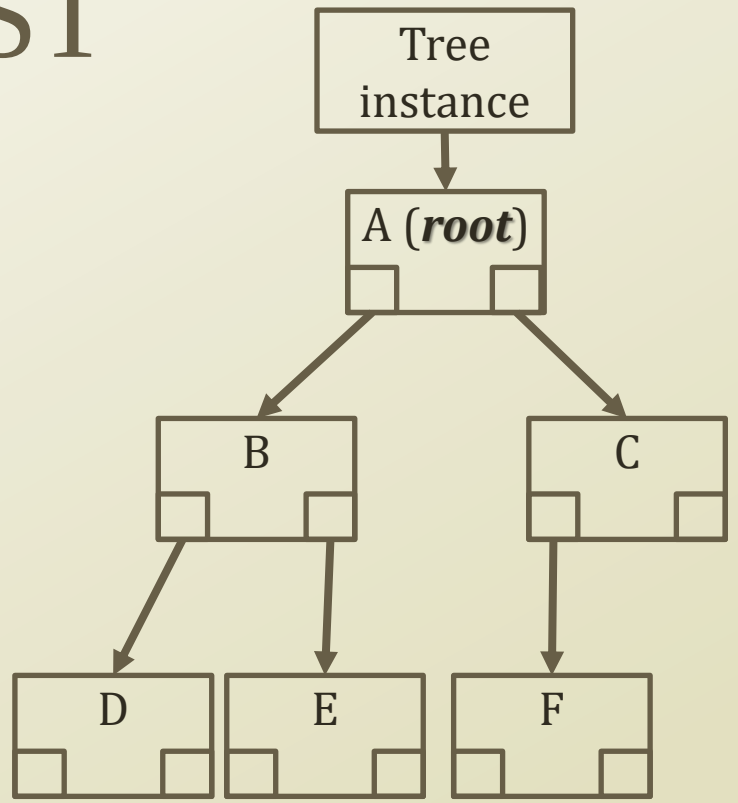
- Has three references
  - Left hand side (LHS)
  - Right hand side (RHS)
  - Its own value
  - Its parent node
  - Not implemented here, but
    - LHS stores
      - Values have lower than its own value
    - RHS stores
      - Values have higher than its own value
    - Just as we all know that the department stores do not have a restroom on the first floor
- Other than four references,
  - Simple get/set methods
    - What are the get/set methods?
      - Coming from encapsulation

```
class TreeNode:
    nodeLHS = None
    nodeRHS = None
    nodeParent = None
    value = None

    def __init__(self, value, nodeParent):
        self.value = value
        self.nodeParent = nodeParent
    def getLHS(self):
        return self.nodeLHS
    def getRHS(self):
        return self.nodeRHS
    def getValue(self):
        return self.value
    def getParent(self):
        return self.nodeParent
    def setLHS(self, LHS):
        self.nodeLHS = LHS
    def setRHS(self, RHS):
        self.nodeRHS = RHS
    def setValue(self, value):
        self.value = value
    def setParent(self, nodeParent):
        self.nodeParent = nodeParent
```

Four references

Implementation of BST

```
class BinarySearchTree:
    root = None

    def __init__(self):
        pass

    def insert(self, value, node = None):...

    def search(self, value, node = None):...

    def delete(self, value, node = None):...

    def findMax(self, node = None):...

    def findMin(self, node = None):...

    def traverseLevelOrder(self):...

    def traverseInOrder(self, node = None):...

    def traversePreOrder(self, node = None):...

    def traversePostOrder(self, node = None):...
```



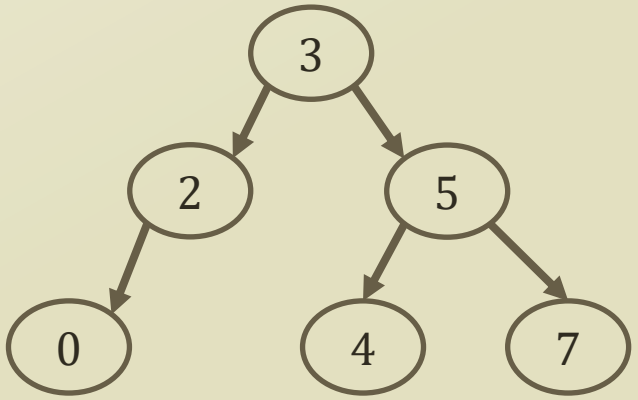
- BST handles the data stored through its root
  - Root has its own value
  - Tree instance access to the root
  - Only through the root, the tree instances access to the descendant nodes of the root

Insert operation of binary search tree

- Insertion operation
  - Retrieve the current node value
    - If the value is equal to the value to insert
      - Return already there!
    - If the value is smaller than the value to insert
      - If there is a node in the right hand-side (RHS), then move to the RHS node (***Recursion***)
      - If there is no node in RHS, create a RHS node with the value to insert
    - If the value is larger than the value to insert
      - If there is a node in the left hand-side (LHS), then move to the LHS node (***Recursion***)
      - If there is no node in LHS, create a LHS node with the value to insert

```
def insert(self, value, node = None):
    if node is None:
        node = self.root
    if self.root is None:
        self.root = TreeNode(value, None)
        return
    if value == node.getValue():
        return
    if value > node.getValue():
        if node.getRHS() is None:
            node.setRHS(TreeNode(value, node))
        else:
            self.insert(value, node.getRHS())
    if value < node.getValue():
        if node.getLHS() is None:
            node.setLHS(TreeNode(value, node))
        else:
            self.insert(value, node.getLHS())
    return
```

Insert numbers: 3, 2, 0, 5, 7, 4…..

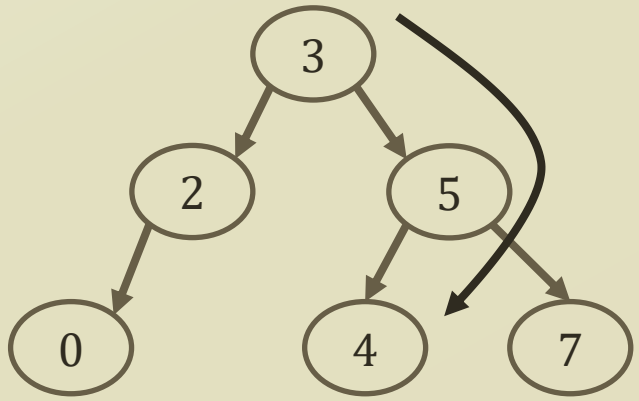


Search operation of binary search tree

- Search operation
  - Retrieve the current node value
    - If the value is equal to the value to search
      - Return **TRUE**
    - If the value is smaller than the value to search
      - If there is a node in the right hand-side (RHS), then move to the RHS node (***Recursion***)
      - If there is no node in RHS, return **FALSE**
    - If the value is larger than the value to search
      - If there is a node in the left hand-side (LHS), then move to the LHS node (***Recursion***)
      - If there is no node in LHS, return **FALSE**

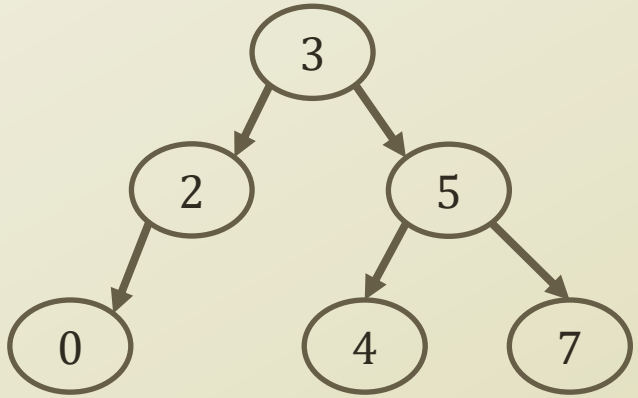
```
def search(self, value, node = None):
    if node is None:
        node = self.root
    if value == node.getValue():
        return True
    if value > node.getValue():
        if node.getRHS() is None:
            return False
        else:
            return self.search(value, node.getRHS())
    if value < node.getValue():
        if node.getLHS() is None:
            return False
        else:
            return self.search(value, node.getLHS())
```

Find 4 in the BST

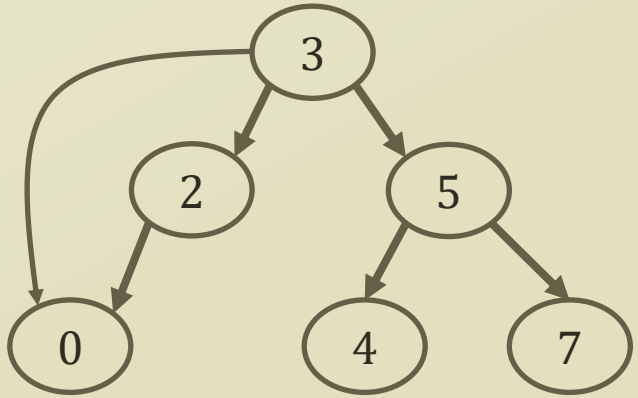


Delete operation of binary search tree (1)

- First, you need to find the node to delete through recursions
- Three deletion cases
  - Case 1: deleting a node with no children
    - Just remove the node by modifying its parent
  - Case 2: deleting a node with one child
    - Replace the node with the child
  - Case 3: deleting a node with two children
    - Find either
      - A maximum in the LHS or A minimum in the RHS
      - Substitute the node to delete with the found value
      - Delete the found node in the LHS or the RHS

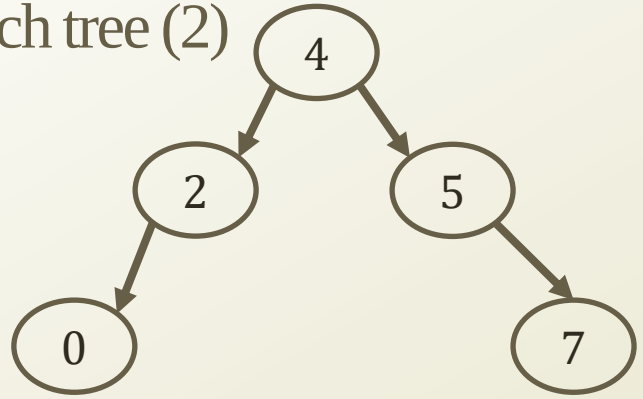


Case 1: Just Remove '0'

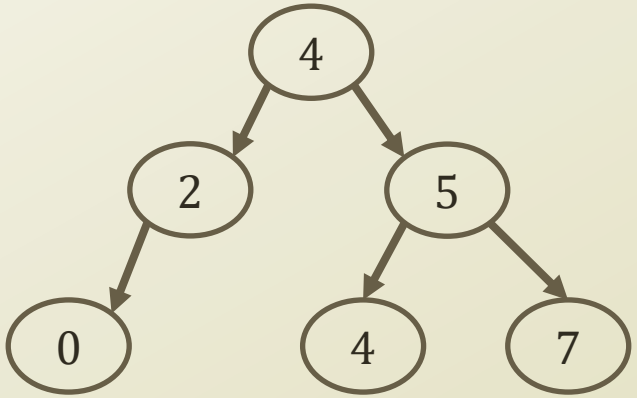


Case 2: Replace '2' with '0'

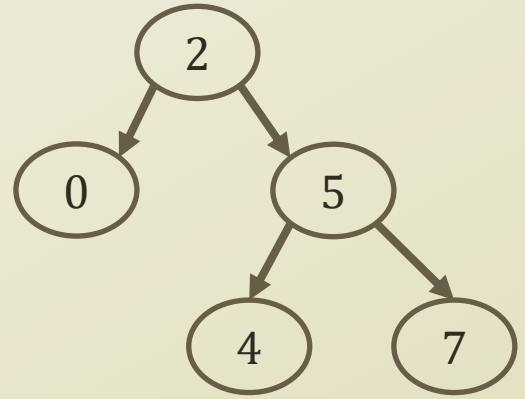
Delete operation of binary search tree (2)



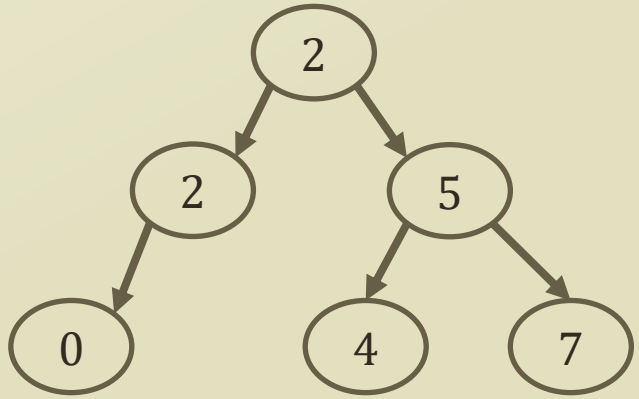
Delete '4' in RHS




```
def delete(self, value, node = None):
    if node is None:
        node = self.root
    if node.getValue() < value:
        return self.delete(value, node.getRHS())
    if node.getValue() > value:
        return self.delete(value, node.getLHS())
    if node.getValue() == value:
        if node.getLHS() is not None and node.getRHS() is not None:
            nodeMin = self.findMin( node.getRHS() )
            node.setValue(nodeMin.getValue())
            self.delete(nodeMin.getValue(), node.getRHS())
            return
        parent = node.getParent()
        if node.getLHS() is not None:
            if node == self.root:
                self.root = node.getLHS()
            elif parent.getLHS() == node:
                parent.setLHS(node.getLHS())
                node.getLHS().setParent(parent)
            else:
                parent.setRHS(node.getLHS())
                node.getLHS().setParent(parent)
            return
        if node.getRHS() is not None:
            if node == self.root:
                self.root = node.getRHS()
            elif parent.getLHS() == node:
                parent.setLHS(node.getRHS())
                node.getRHS().setParent(parent)
            else:
                parent.setRHS(node.getRHS())
                node.getRHS().setParent(parent)
            return
        if node == self.root:
            self.root = None
        elif parent.getLHS() == node:
            parent.setLHS(None)
        else:
            parent.setRHS(None)
        return
```

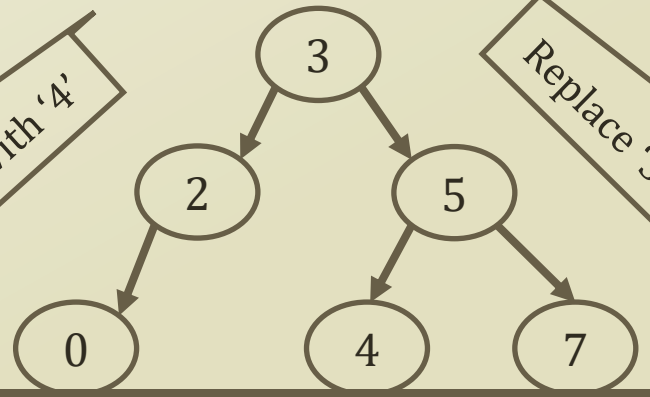


Delete '2' in LHS



Replace '3' with '4'

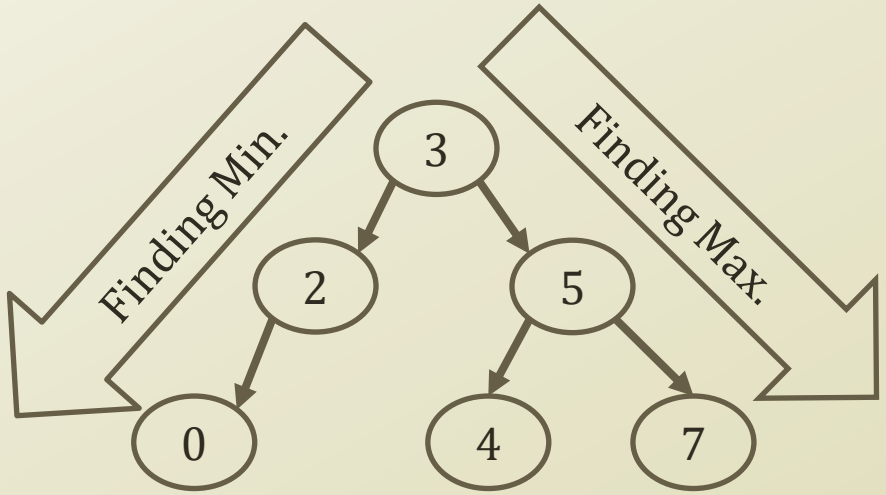
Replace '3' with '2'



Case 3: Replace '3' with 'X'

Minimum and maximum in binary search tree

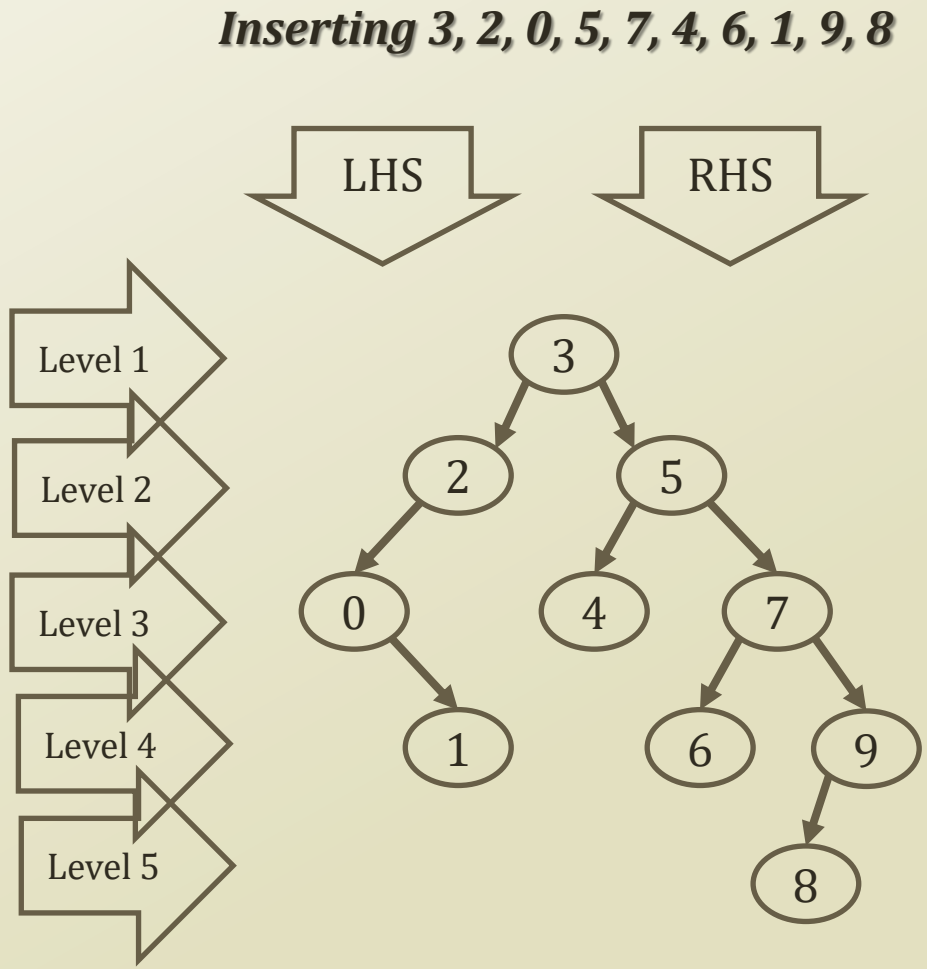
- Finding minimum in a BST
  - Just keep following the LHS
    - Because this will always result in the smaller value than the value of the current node
    - When you can't any LHS, then the value of the current node is the smallest
- Finding maximum in a BST
  - Just keep following the RHS
    - Because this will always result in the larger value than the value of the current node
    - When you can't any RHS, then the value of the current node is the largest




```
def findMax(self, node = None):
    if node is None:
        node = self.root
    if node.getRHS() is None:
        return node
    return self.findMax(node.getRHS())

def findMin(self, node = None):
    if node is None:
        node = self.root
    if node.getLHS() is None:
        return node
    return self.findMin(node.getLHS())
```

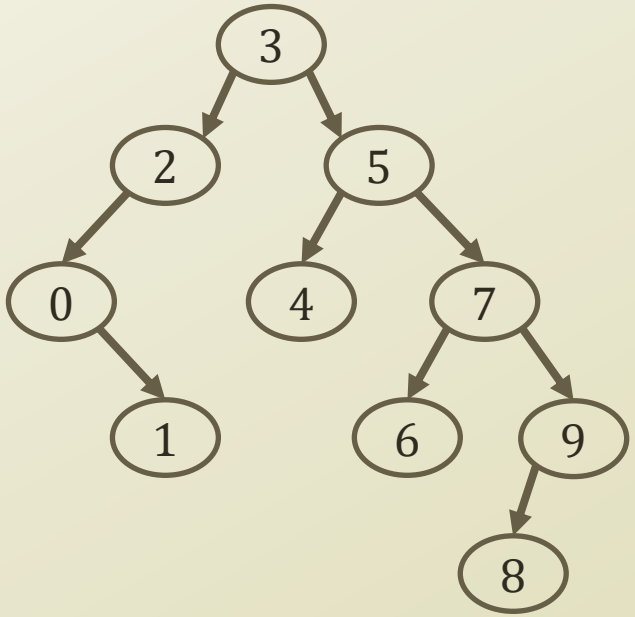
Tree traversing

- Tree
  - Complicated than a list
  - Multiple ways to show the entire dataset
    - If it were a list
      - Just show the values from the beginning to the end
    - Since this is a BST
      - You have to choose what to show at a time
        - The value in LHS
        - The value in RHS
        - The value that you have
- Hence there are multiple traversing approaches



Depth first traverse

- Pre-order traverse
  - Order: Current, LHS, RHS in ***Recursion***
  - 3, 2, 0, 1, 5, 4, 7, 6, 9, 8
- In-order traverse
  - Order: LHS, Current, RHS in ***Recursion***
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- Post-order traverse
  - Order: LHS, RHS, Current in ***Recursion***
  - 1, 0, 2, 4, 6, 8, 9, 7, 5, 3

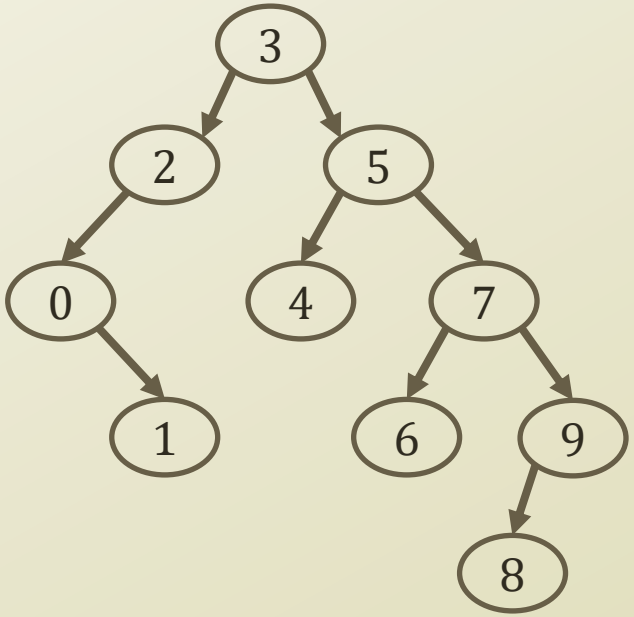



```
def traverseInOrder(self, node = None):
    if node is None:
        node = self.root
    ret = []
    if node.getLHS() is not None:
        ret = ret + self.traverseInOrder(node.getLHS())
    ret.append( node.getValue() )
    if node.getRHS() is not None:
        ret = ret + self.traverseInOrder(node.getRHS())
    return ret

def traversePreOrder(self, node = None):
    if node is None:
        node = self.root
    ret = []
    ret.append( node.getValue() )
    if node.getLHS() is not None:
        ret = ret + self.traversePreOrder(node.getLHS())
    if node.getRHS() is not None:
        ret = ret + self.traversePreOrder(node.getRHS())
    return ret

def traversePostOrder(self, node = None):
    if node is None:
        node = self.root
    ret = []
    if node.getLHS() is not None:
        ret = ret + self.traversePostOrder(node.getLHS())
    if node.getRHS() is not None:
        ret = ret + self.traversePostOrder(node.getRHS())
    ret.append( node.getValue() )
    return ret
```

Breadth first traverse

- Queue-based level-order traverse
  - 3, 2, 5, 0, 4, 7, 1, 6, 9, 8
  - Enqueue the root
    - While until queue is empty
      - Current = Dequeue one element
      - Print current
      - If Current's LHS exist
        - Enqueue current.LHS
      - If Current's RHS exist
        - Enqueue current.RHS

Current	Queue
	3
3	2, 5
2	5, 0
5	0, 4, 7
0	4, 7, 1
4	7, 1
7	1, 6, 9
1	6, 9
6	9
9	8
8	

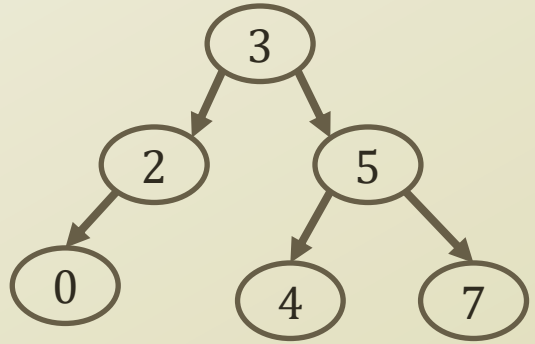



```
def traverseLevelOrder(self):
    ret = []
    Q = Queue()
    Q.enqueue(self.root)
    while not Q.isEmpty():
        node = Q.dequeue()
        if node is None:
            continue
        ret.append(node.getValue())
        if node.getLHS() is not None:
            Q.enqueue(node.getLHS())
        if node.getRHS() is not None:
            Q.enqueue(node.getRHS())
    return ret
```

Performance of binary search tree

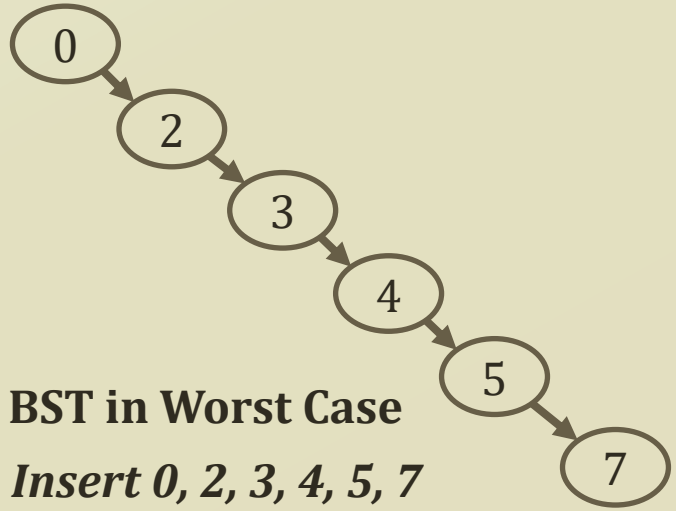
Coming from divide and conquer

	Linked List	BST in Average	BST in Worst Case
Search	O(n)	O(log n)	O(n)
Insert after search	O(1)	O(1)	O(1)
Delete after search	O(1)	O(1)	O(1)
Traverse	O(n)	O(n)	O(n)



BST in Average

Insert 3, 2, 0, 5, 4, 7



BST in Worst Case

Insert 0, 2, 3, 4, 5, 7

Complete Implementation of BST (1)

```
from src.edu.kaist.seslab.ie362.week3.Queue import Queue

class TreeNode:
    nodeLHS = None
    nodeRHS = None
    nodeParent = None
    value = None

    def __init__(self, value, nodeParent):
        self.value = value
        self.nodeParent = nodeParent
    def getLHS(self):
        return self.nodeLHS
    def getRHS(self):
        return self.nodeRHS
    def getValue(self):
        return self.value
    def getParent(self):
        return self.nodeParent
    def setLHS(self, LHS):
        self.nodeLHS = LHS
    def setRHS(self, RHS):
        self.nodeRHS = RHS
    def setValue(self, value):
        self.value = value
    def setParent(self, nodeParent):
        self.nodeParent = nodeParent

class BinarySearchTree:
    root = None

    def __init__(self):
        pass

    def insert(self, value, node = None):
        if node is None:
            node = self.root
        if self.root is None:
            self.root = TreeNode(value, None)
            return
        if value == node.getValue():
            return
        if value > node.getValue():
            if node.getRHS() is None:
                node.setRHS(TreeNode(value, node))
            else:
                self.insert(value, node.getRHS())
        if value < node.getValue():
            if node.getLHS() is None:
                node.setLHS(TreeNode(value, node))
            else:
                self.insert(value, node.getLHS())
        return

    def search(self, value, node = None):
        if node is None:
            node = self.root
        if value == node.getValue():
            return True
        if value > node.getValue():
            if node.getRHS() is None:
                return False
            else:
                return self.search(value, node.getRHS())
        if value < node.getValue():
            if node.getLHS() is None:
                return False
            else:
                return self.search(value, node.getLHS())

    def delete(self, value, node = None):
        if node is None:
            node = self.root
        if node.getValue() < value:
            return self.delete(value, node.getRHS())
        if node.getValue() > value:
            return self.delete(value, node.getLHS())
        if node.getValue() == value:
            if node.getLHS() is not None and node.getRHS() is not None:
                nodeMin = self.findMin( node.getRHS() )
                node.setValue(nodeMin.getValue())
                self.delete(nodeMin.getValue(), node.getRHS())
                return
            parent = node.getParent()
            if node.getLHS() is not None:
                if node == self.root:
                    self.root = node.getLHS()
                elif parent.getLHS() == node:
                    parent.setLHS(node.getLHS())
                    node.getLHS().setParent(parent)
                else:
                    parent.setRHS(node.getLHS())
                    node.getLHS().setParent(parent)
                return
            if node.getRHS() is not None:
                if node == self.root:
                    self.root = node.getRHS()
                elif parent.getLHS() == node:
                    parent.setLHS(node.getRHS())
                    node.getRHS().setParent(parent)
                else:
                    parent.setRHS(node.getRHS())
                    node.getRHS().setParent(parent)
                return
            if node == self.root:
                self.root = None
            elif parent.getLHS() == node:
                parent.setLHS(None)
            else:
                parent.setRHS(None)
            return
```

Complete Implementation of BST (2)

```
    def findMax(self, node = None):
        if node is None:
            node = self.root
        if node.getRHS() is None:
            return node
        return self.findMax(node.getRHS())

    def findMin(self, node = None):
        if node is None:
            node = self.root
        if node.getLHS() is None:
            return node
        return self.findMin(node.getLHS())

    def traverseLevelOrder(self):
        ret = []
        Q = Queue()
        Q.enqueue(self.root)
        while not Q.isEmpty():
            node = Q.dequeue()
            if node is None:
                continue
            ret.append(node.getValue())
            if node.getLHS() is not None:
                Q.enqueue(node.getLHS())
            if node.getRHS() is not None:
                Q.enqueue(node.getRHS())
        return ret

    def traverseInOrder(self, node = None):
        if node is None:
            node = self.root
        ret = []
        if node.getLHS() is not None:
            ret = ret + self.traverseInOrder(node.getLHS())
        ret.append( node.getValue() )
        if node.getRHS() is not None:
            ret = ret + self.traverseInOrder(node.getRHS())
        return ret

    def traversePreOrder(self, node = None):
        if node is None:
            node = self.root
        ret = []
        ret.append( node.getValue() )
        if node.getLHS() is not None:
            ret = ret + self.traversePreOrder(node.getLHS())
        if node.getRHS() is not None:
            ret = ret + self.traversePreOrder(node.getRHS())
        return ret

    def traversePostOrder(self, node = None):
        if node is None:
            node = self.root
        ret = []
        if node.getLHS() is not None:
            ret = ret + self.traversePostOrder(node.getLHS())
        if node.getRHS() is not None:
            ret = ret + self.traversePostOrder(node.getRHS())
        ret.append( node.getValue() )
        return ret


tree = BinarySearchTree()
tree.insert(3)
tree.insert(2)
tree.insert(0)
tree.insert(5)
tree.insert(7)
tree.insert(4)
tree.insert(6)
tree.insert(1)
tree.insert(9)
tree.insert(8)
```

```
print(tree.traverseLevelOrder())
print(tree.traverseInOrder())
print(tree.traversePreOrder())
print(tree.traversePostOrder())

tree.delete(5)

print(tree.traverseLevelOrder())

tree.delete(1)

print(tree.traverseLevelOrder())

tree.delete(9)

print(tree.traverseLevelOrder())

tree.delete(3)

print(tree.traverseLevelOrder())
```

```
[3, 2, 5, 0, 4, 7, 1, 6, 9, 8]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[3, 2, 0, 1, 5, 4, 7, 6, 9, 8]
[1, 0, 2, 4, 6, 8, 9, 7, 5, 3]
[3, 2, 6, 0, 4, 7, 1, 9, 8]
[3, 2, 6, 0, 4, 7, 9, 8]
[3, 2, 6, 0, 4, 7, 8]
[4, 2, 6, 0, 7, 8]
```