2021-2022

Data Structures and Algorithms (II) – Trees

Dr. Dapeng Dong

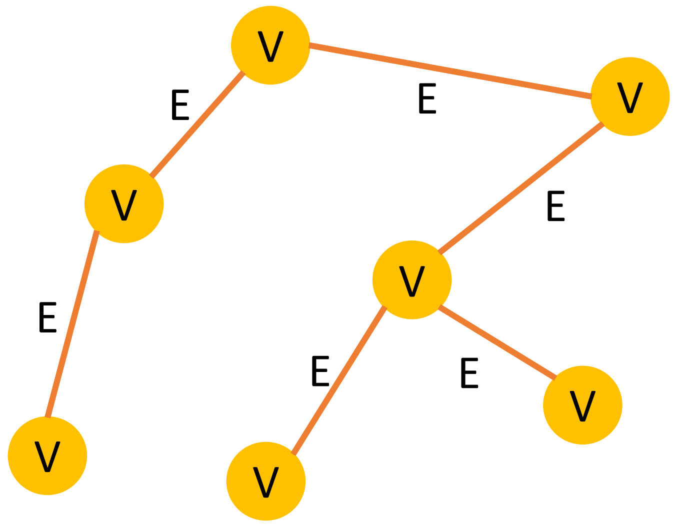
Objectives

- to know the concepts, terminologies and mathematical properties of trees
- to be able to describe the differences between various types of trees and their implementations
- to be able to analyze the performance of operations on trees
- to know how trees are used in real-world applications

Properties of Trees

V: Vertex or Node
E: Edge

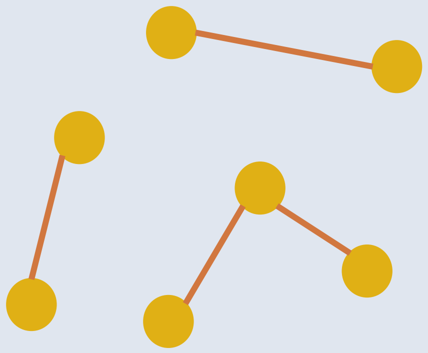
A *tree* or *free tree* is a connected, acyclic, undirected graph.



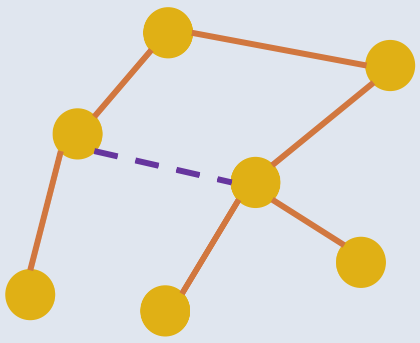
Tree

Theorem:
Let $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ be an undirected graph. The following statements are equivalent.

- Any two vertices in $\boldsymbol{G}$ are connected by a unique simple path.
- $\boldsymbol{G}$ is connected, but if any edge is removed from $\boldsymbol{E}$ (the set of edges in the tree), the resulting graph is disconnected.
- $\boldsymbol{G}$ is connected, and $|\boldsymbol{E}| = |\boldsymbol{V}| - \mathbf{1}$.
- $\boldsymbol{G}$ is acyclic, and $|\boldsymbol{E}| = |\boldsymbol{V}| - \mathbf{1}$.
- $\boldsymbol{G}$ is acyclic, but if any edge is added to $\boldsymbol{E}$, the resulting graph contains a cycle.

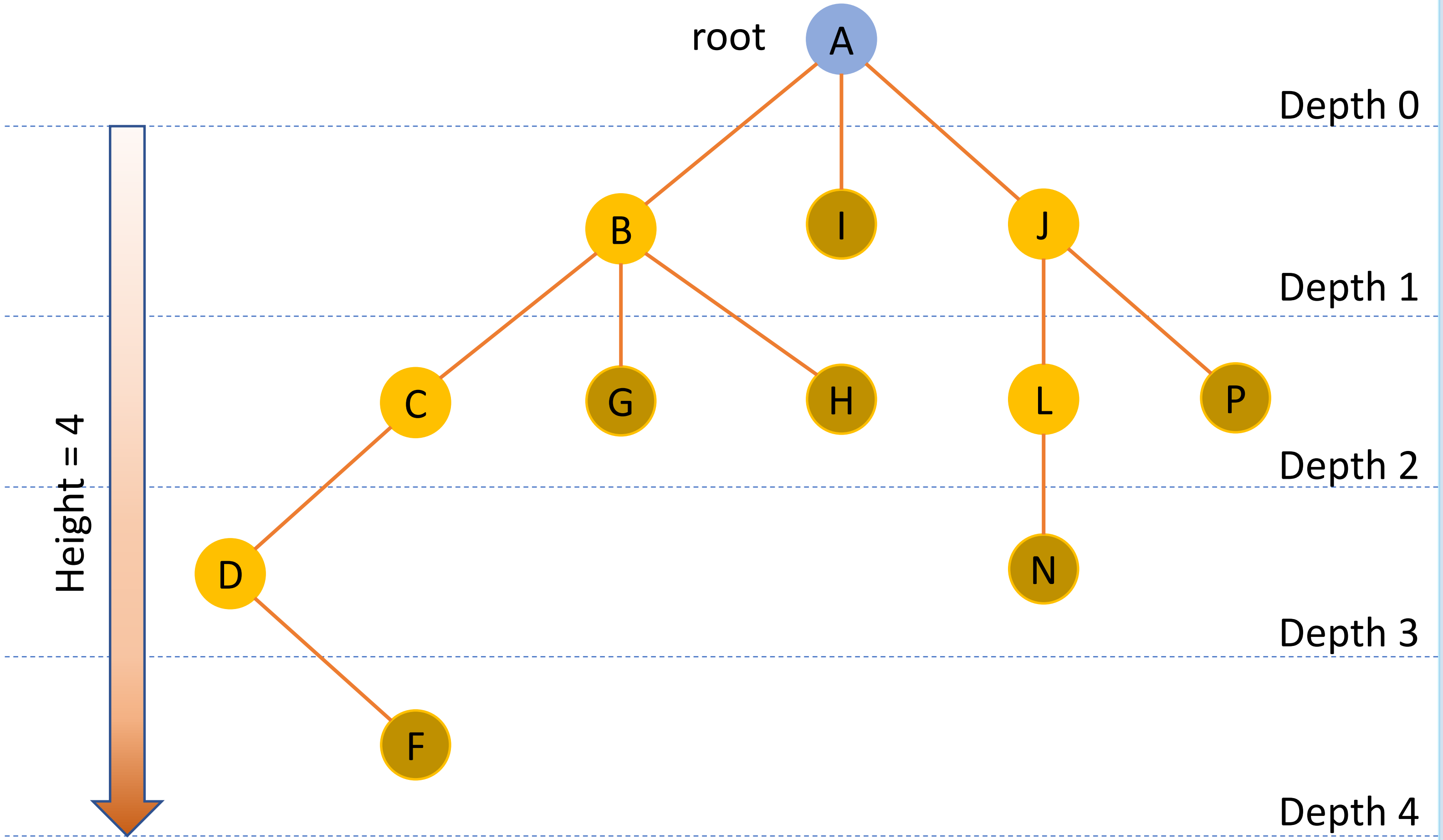


Forest
(It is not connected; thus, it is not a tree)



It is connected, but contains a loop; thus, it is neither a tree nor a forest.

Rooted Trees (1)

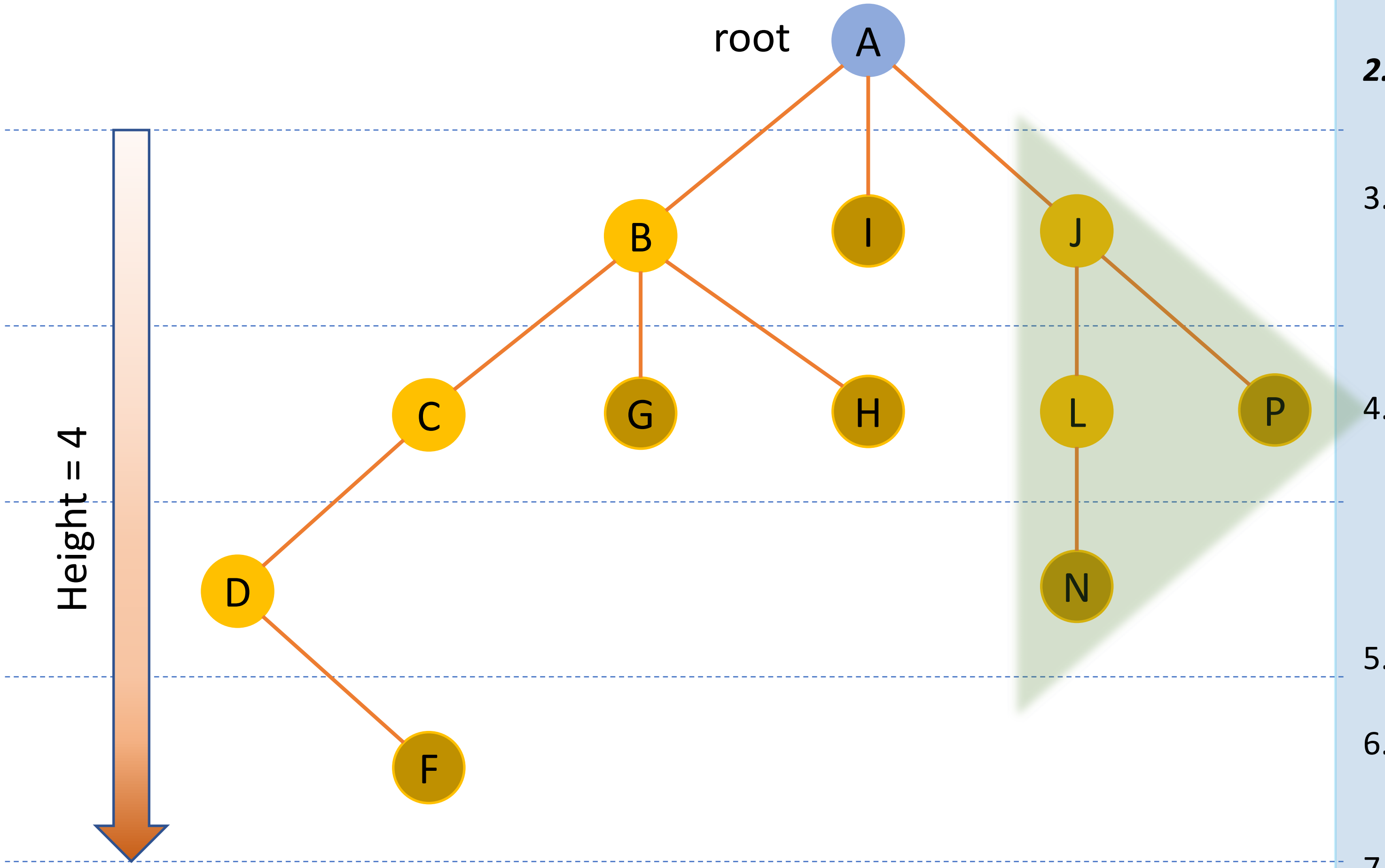


A ***rooted tree*** is a tree in which one of the vertices is distinguished from the others.

Terminologies

1. ***Root*** or ***Root Node*** of the tree: { ***A*** }
2. ***Leaf*** or ***External Node***: a node without children, { ***F, G, H, I, N, P*** }
3. ***Internal Node***: a non-leaf node, { ***A, B, C, D, J, L*** }
4. ***Depth*** of a node: the length of the simple path from the root to the node. E.g., the depth of ***J*** is 1.
5. ***Height*** of a tree: the largest depth of any node in the tree.
6. ***Degree*** of a node: the number of children of the node in a **rooted tree**. E.g., the degree of ***B*** is 3; the degree of ***L*** is 1.

Rooted Trees (2)



Terminologies

1. ***Ancestor*** of a node: any node on the unique simple path from the *root node* to a designated node is called an *ancestor* of the node. E.g., the simple path from the root node ***A*** to node ***N*** is (***A*** → ***J*** → ***L*** → ***N***), thus the nodes { ***A***, ***J***, ***L*** } are the ancestors of ***N***.

2. ***Descendant*** of a node: if a node ***x*** is an ancestor of a node ***y***, then ***y*** is a descendant of ***x***. E.g., ***D*** is a descendant of ***C***, ***B*** and ***A***.

3. Every node is an ancestor and a descendant of itself. If an ancestor of a node is not itself, then it is called a ***proper ancestor***. E.g., ***C*** is a ***proper ancestor*** of ***D***, vice versa, ***D*** is a ***proper descendant*** of ***C***.

4. If the last edge on the simple path from the root node of a rooted tree to a node ***x*** is (***x***, ***y***), then ***y*** is the ***parent*** of ***x***, and ***x*** is a child of ***y***. E.g., in a simple path from the root to node ***C***, i.e., (***A*** → ***B*** → ***C***), the last edge is (***B***, ***C***), thus ***B*** is the ***parent*** of ***C***, and ***C*** is a ***child*** of ***B***.

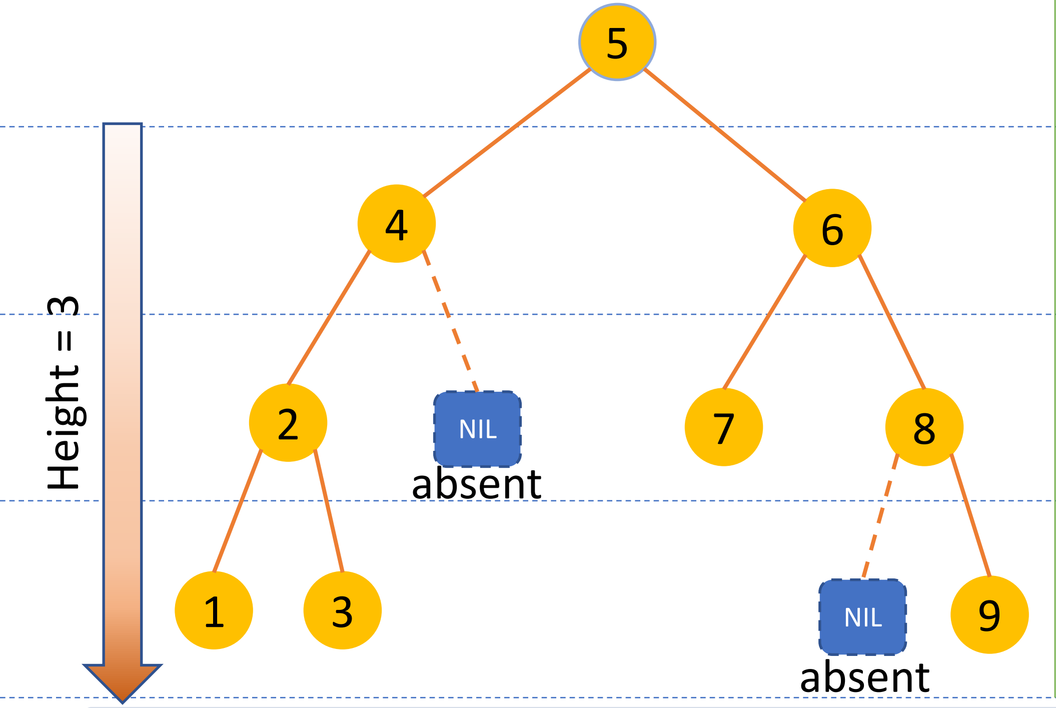
5. The root is the only node without a parent.

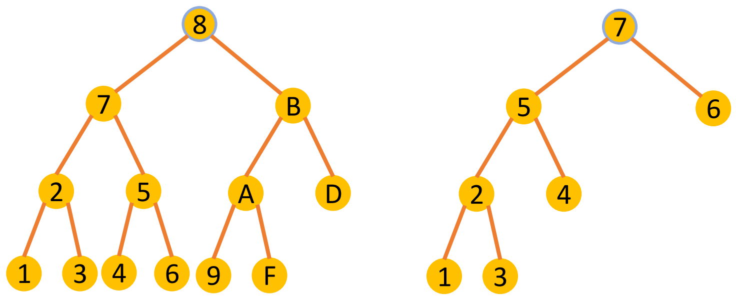
6. If nodes have the same parent, they are ***siblings***. E.g., Node ***L*** and ***P*** have the same parent ***J***, then ***L*** and ***P*** are siblings.

7. A ***subtree*** rooted at a node is the tree induced by descendants of the node, rooted at the node. E.g., a subtree rooted at the node ***J*** contains Nodes { ***L***, ***N***, ***P*** }

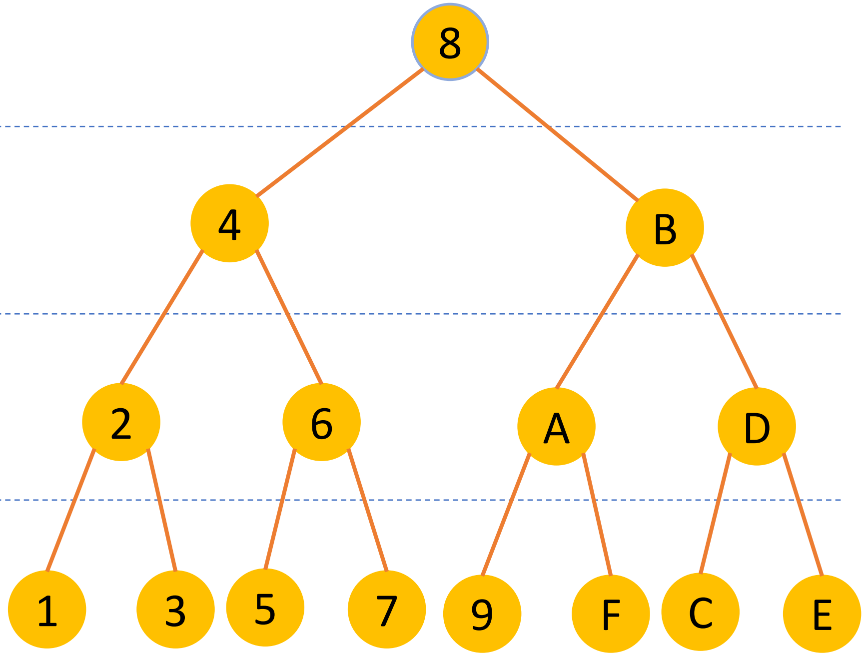
Binary Trees

A Binary Tree





Full Binary Trees



Terminologies

1. ***Empty Tree*** or ***Null Tree:*** A binary tree that contains no nodes.

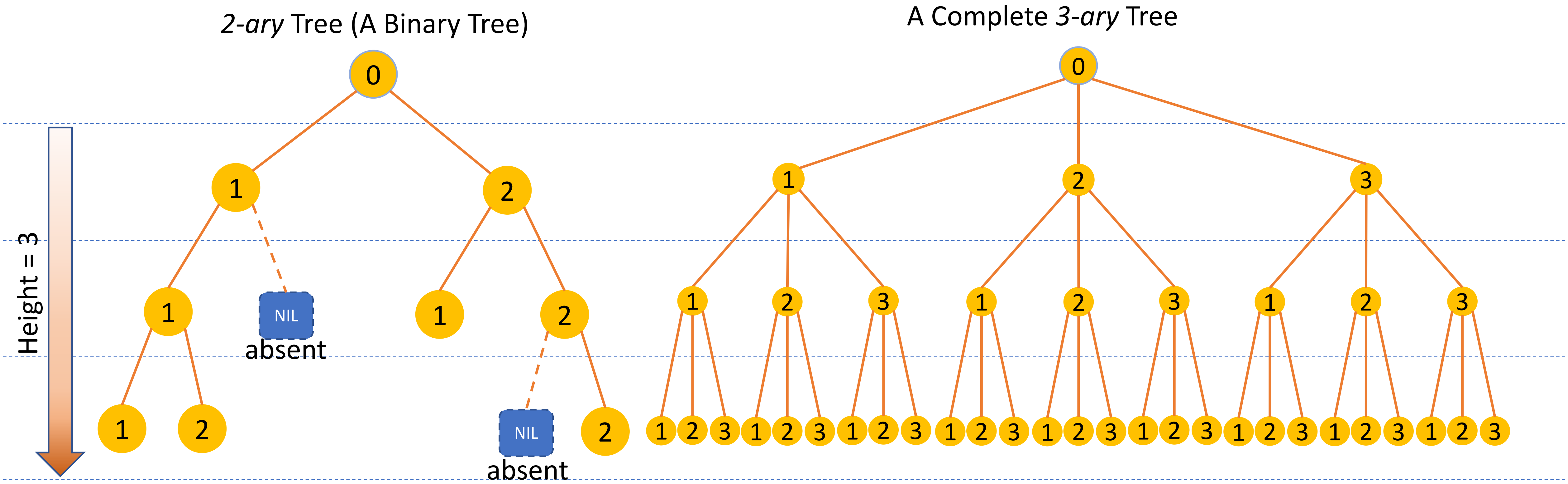
2. If a subtree is the *null tree* (represented using '**NIL**'), then the child is ***absent*** or ***missing***. E.g., the *right child* of Node 4 is *absent*; the *left child* of Node 8 is *absent*.

3. ***Full Binary Tree***: each node is either a leaf or has degree exactly 2.

A ***binary tree*** is a structure defined on a finite set of nodes that either contain no nodes or is composed of three disjoint sets of nodes: a root node, a binary tree called its left subtree, and a binary tree called its right subtree.

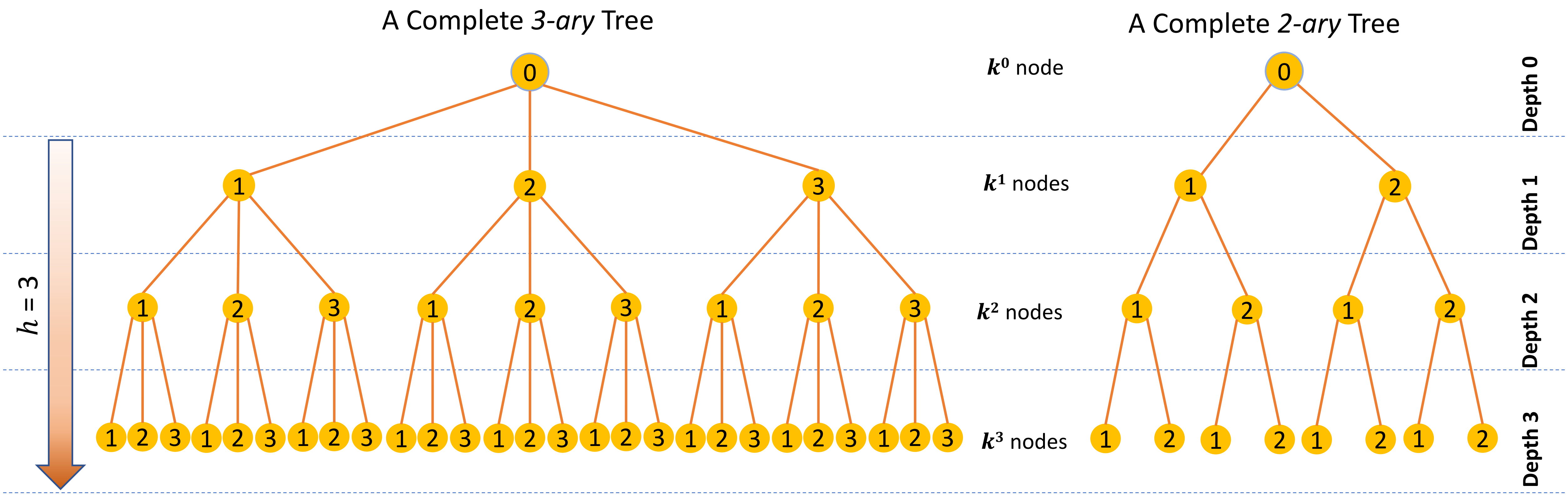
Positional Trees and *k-ary* Trees

A complete *k-ary* tree: all leaves have the same depth, and all internal nodes have degree $\boldsymbol{k}$.



- In a ***positional tree***, the children of a node are labeled with distinct positive integers.
- A ***k-ary*** tree is a positional tree in which for every node, all children with labels greater than $\boldsymbol{k}$ are missing

Properties of Complete k-*ary* Trees



Number of internal nodes of a complete ternary tree:

$$\frac{1 - 3^3}{1 - 3} = 13$$

The number of internal nodes of a complete k-*ary* tree:

1

$$1 + k + k^2 + \ldots + k^{h-1} = \sum_{i=0}^{h-1} k^i = \frac{1 - k^h}{1 - k}$$

Number of internal nodes for a complete binary tree:

$$\frac{1 - 2^3}{1 - 2} = 7$$

Q: How do we deal with duplicated keys?

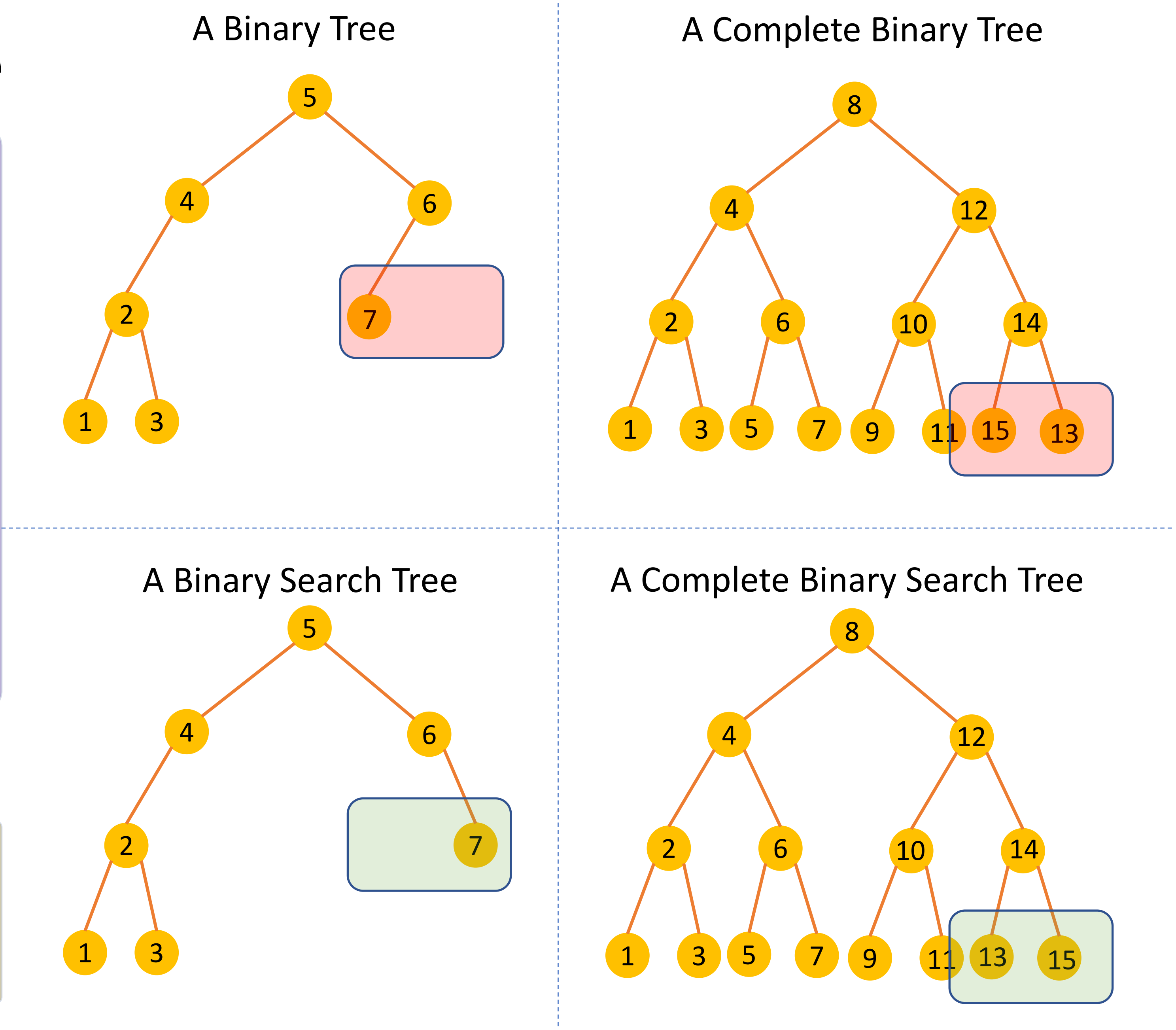
Binary Search Tree

"A *binary search tree* (BST) is a binary tree where each node has a Comparable key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left subtree and smaller than the keys in all nodes in that node's right subtree."

-- Sedgewick, R., & Wayne, K. (2011). Algorithms. Addison-wesley professional.

Property of BST:

Let $\boldsymbol{x}$ be a node in a BST. If $\boldsymbol{y}$ is a node in the left subtree of $\boldsymbol{x}$, then $\boldsymbol{y.key} \leq \boldsymbol{x.key}$. If $\boldsymbol{y}$ is a node in the right subtree of $\boldsymbol{x}$, then $\boldsymbol{y.key} \geq \boldsymbol{x.key}$.



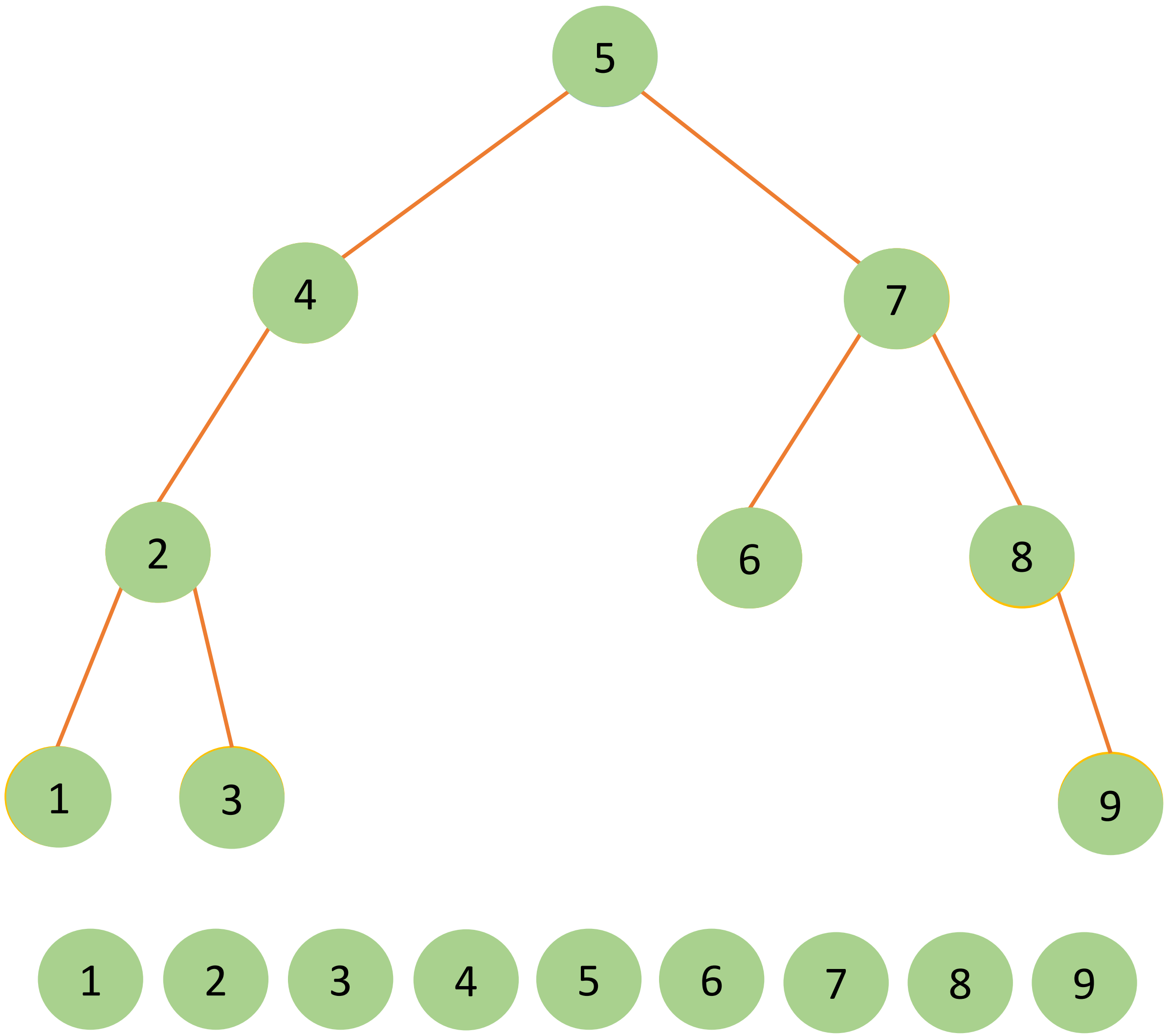
BST Operations -- Tree Traversal

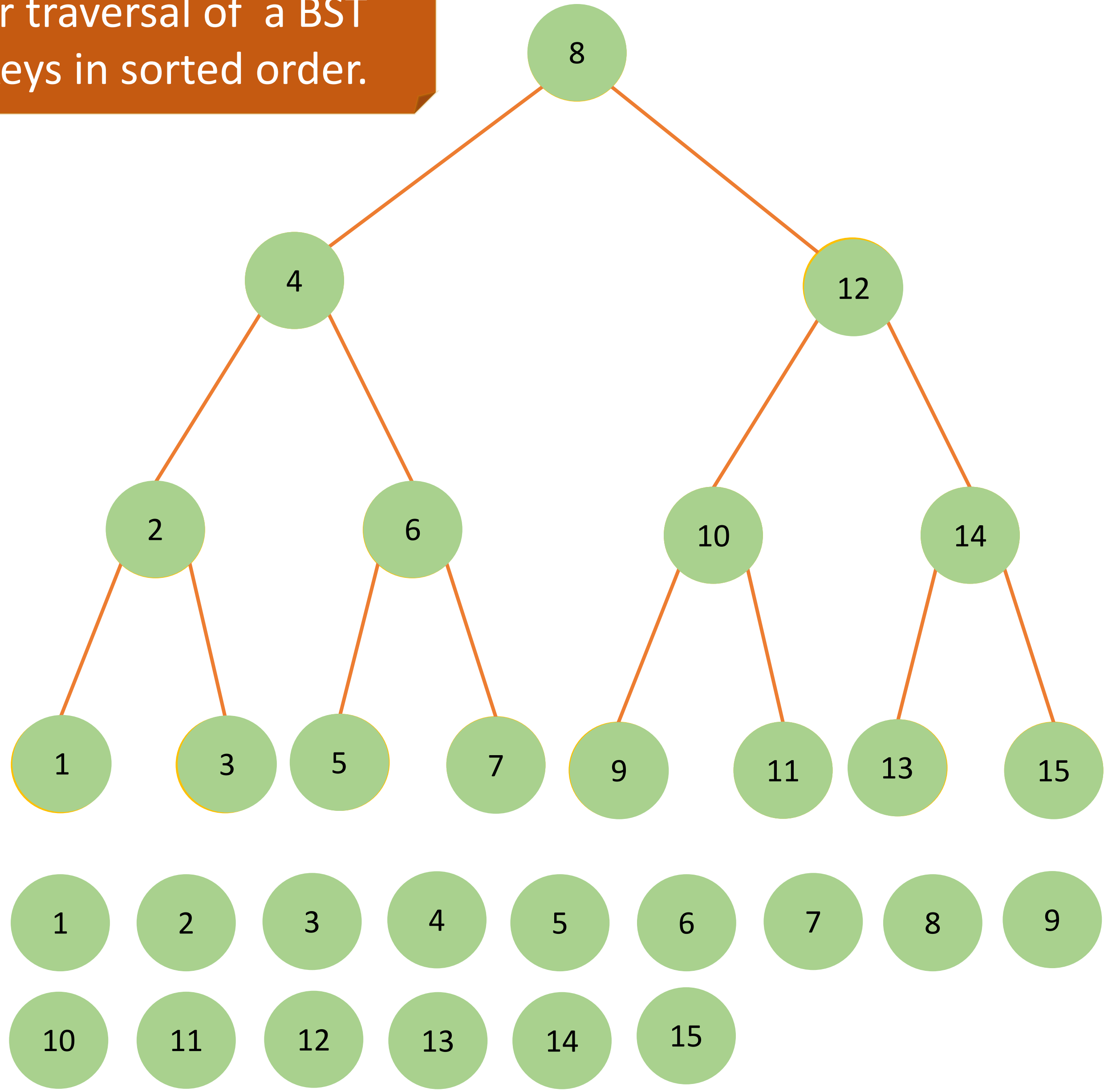
- Depth-first Search (DFS) 深度优先搜索
  - In-order
    - Prints the root of a subtree between printing the values in its left subtree and printing those in its right subtree.
  - Pre-order
    - Prints the roots before the values in either subtree.
  - Post-order
    - Prints the root after the values in its subtrees.
- Breadth-first Search (BFS) 广度优先搜索
  - Prints all the keys in order on the current level before moving to the next depth (aka, level-order search)

Left → Root → Right

In-order Traversal

In-order traversal of a BST prints keys in sorted order.





Representation of Nodes and Trees

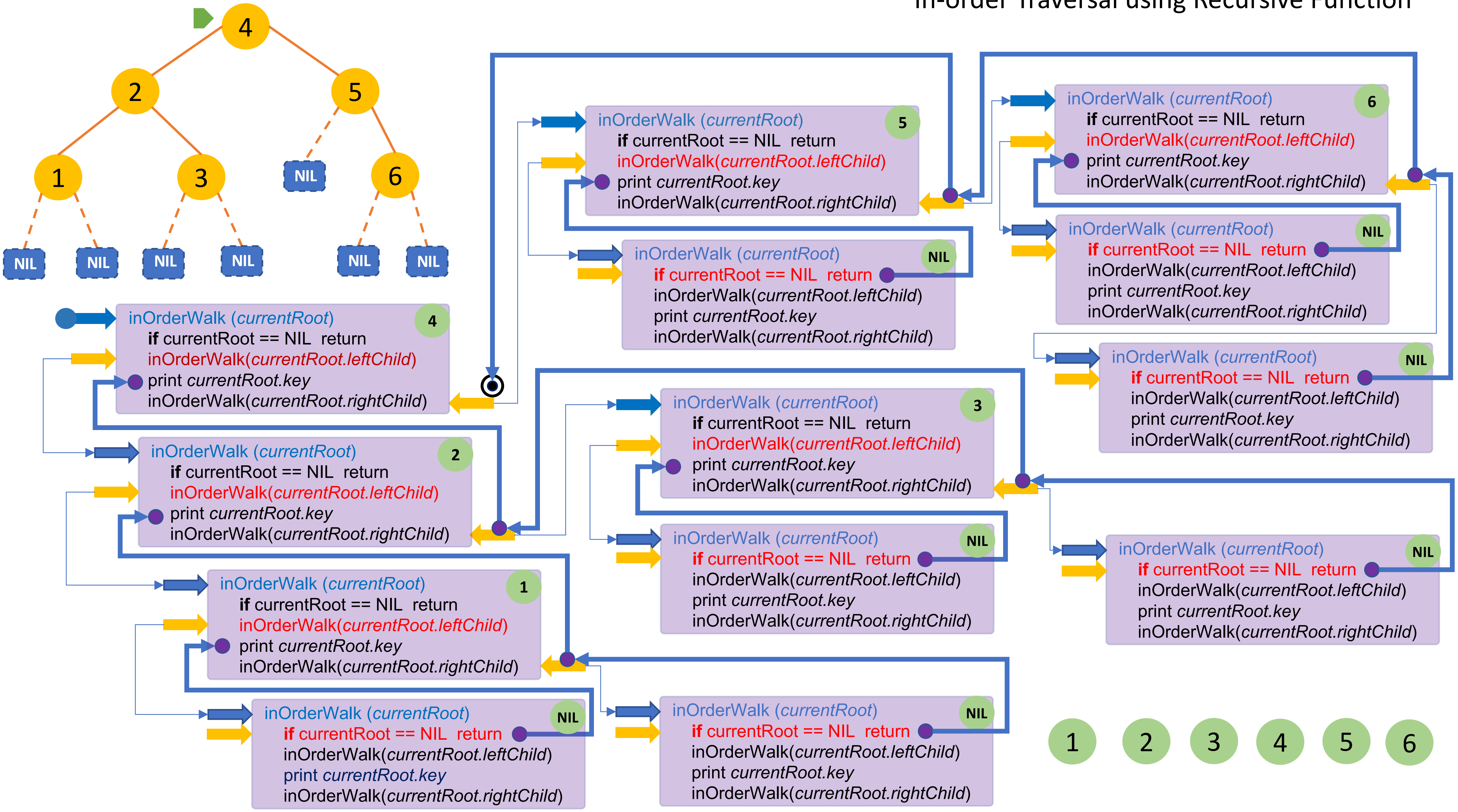
```java
public class BinarySearchTree<K extends Comparable<K>, V> {

        private Node root;

        private class Node {
            private K key;
            private V value;
            private Node leftChild, rightChild;

            public Node(K key, V value) {
                this.key = key;
                this.value = value;
            }
        }

        public void insert(K key, V value) { … }
        public void delete(K key) { … }
        public void max( ) { … }
        public void min( ) { … }
        public void successor(K key) { … }
        public void predecessor(K key) { … }
        public void search(K key) { … }
        public void inOrderWalk( ) { … }
        …
}
```

A BST consists of nodes with each having a generic type of comparable key and a generic type of value.

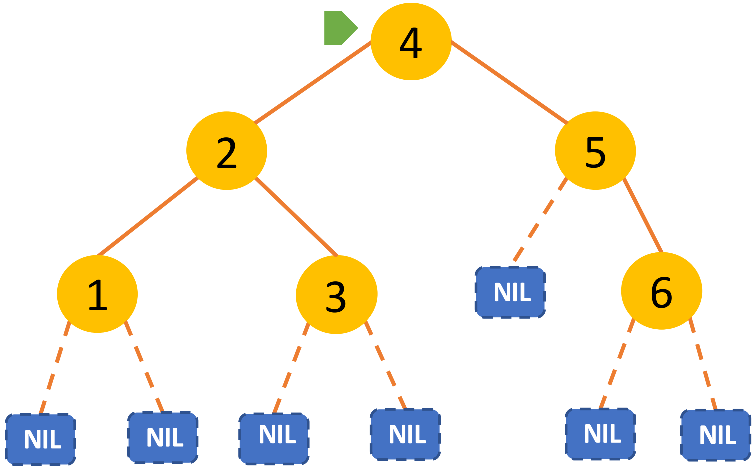
The representation of Node. Each node has a key, an associated value, a left child and a right child.

A set of operations defined on the tree.

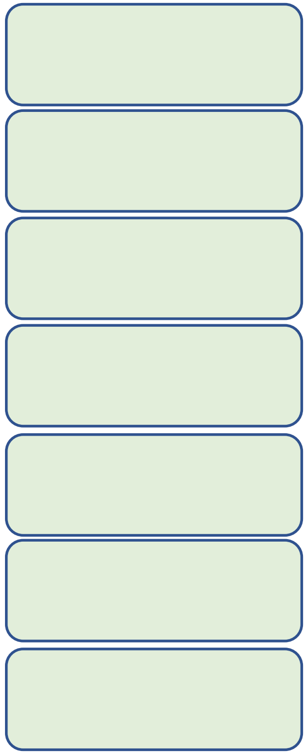
In-order Traversal using Recursive Function



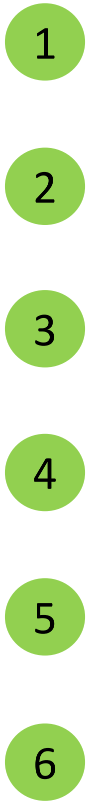
In-order Implementation (Iterative)




```
inOrderWalk (currentRoot)
    S = ∅
    while S ≠ ∅ or currentRoot ≠ NIL
        if currentRoot ≠ NIL
            PUSH(S, currentRoot)
            currentRoot = currentRoot.leftChild
        else
            currentRoot = POP(S)
            print currentRoot.key
            currentRoot = currentRoot.rightChild
```



Stack



Root → Left → Right

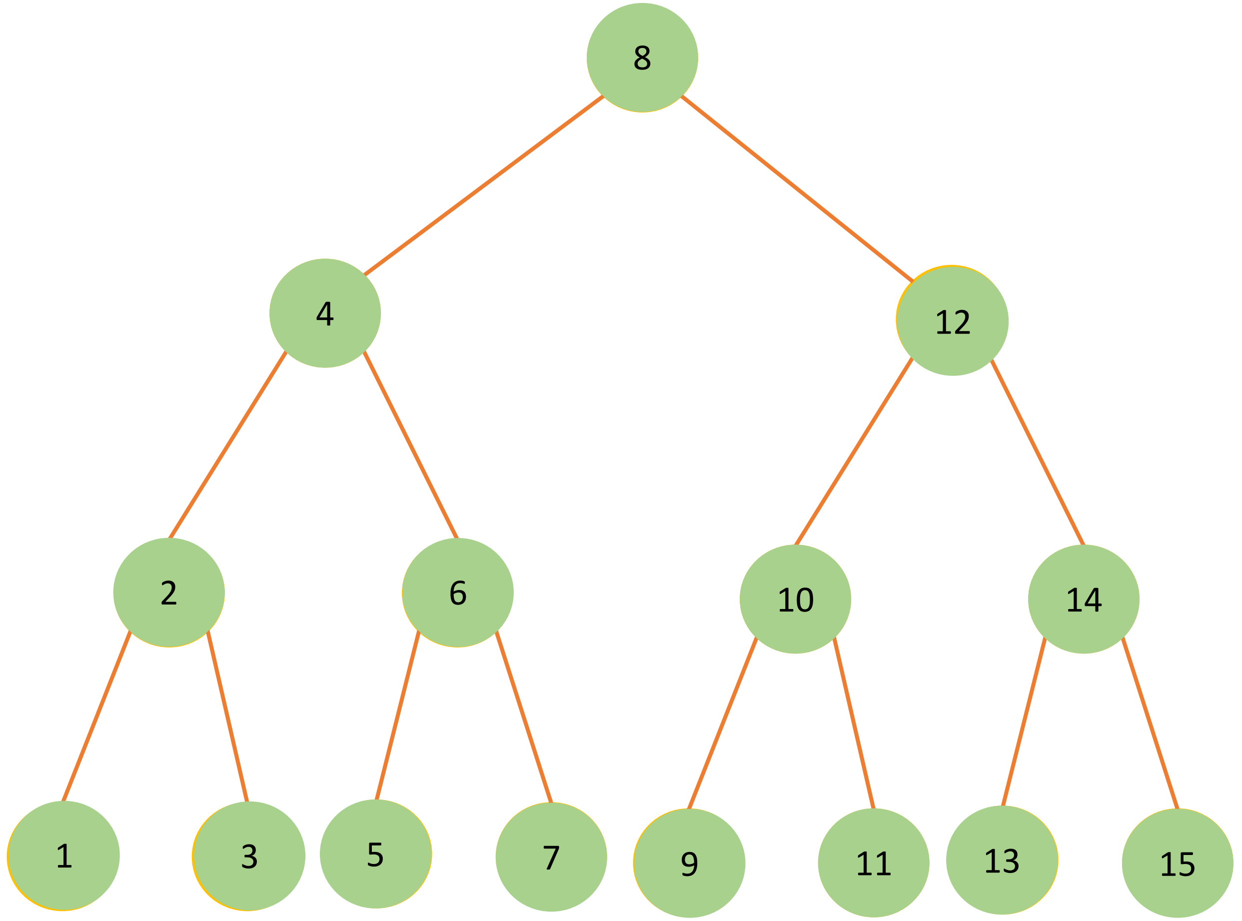
Pre-order Traversal

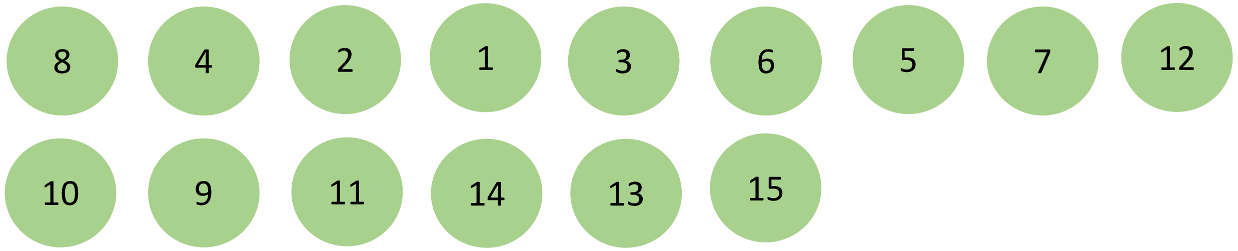
Recursive

```
preOrderWalk(currentRoot)
    if currentRoot == NIL
        return
    print currentRoot.key
    preOrderWalk(currentRoot.leftChild)
    preOrderWalk(currentRoot.rightChild)
```

Iterative

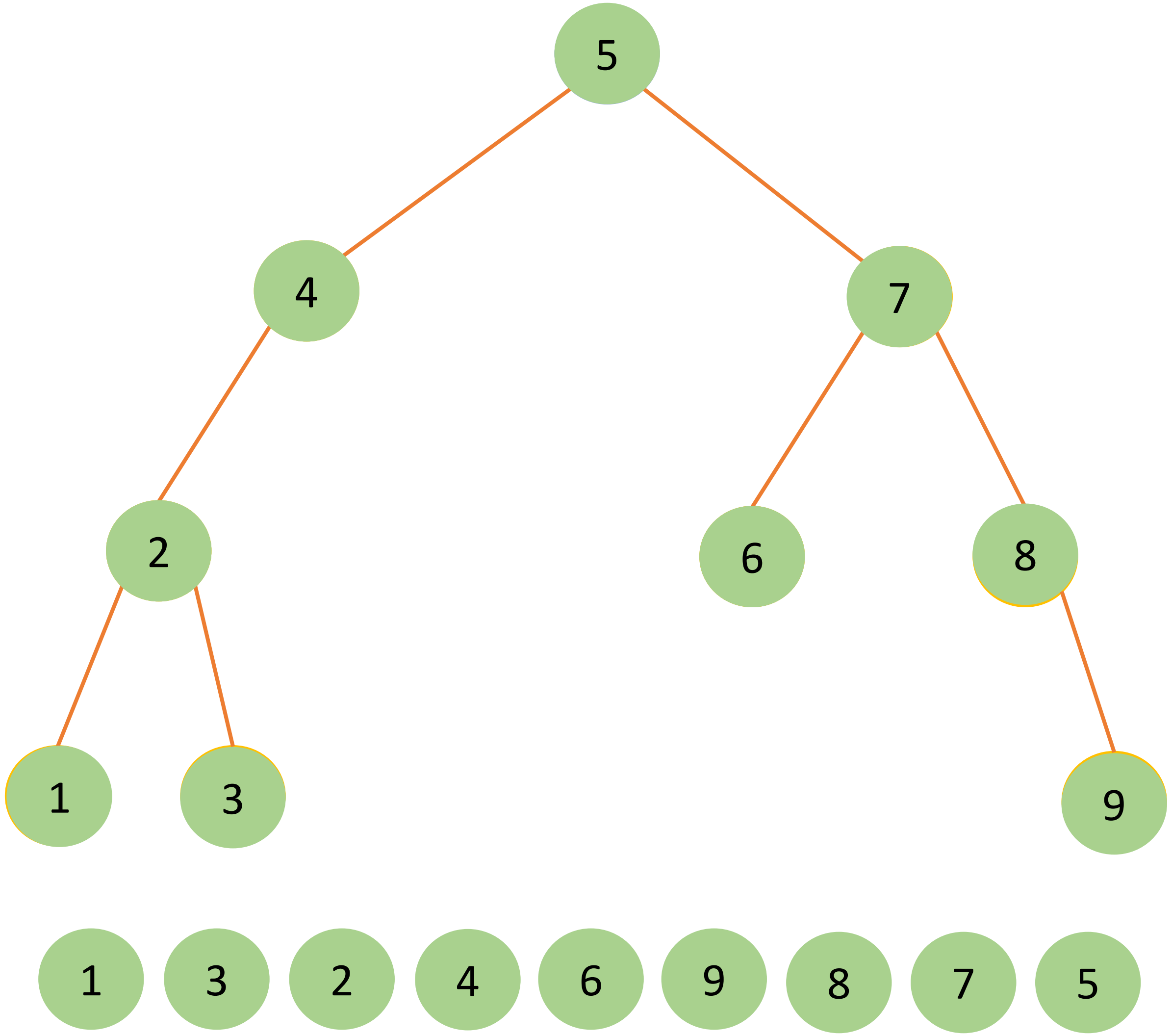
```
preOrderWalk (currentRoot)
    if currentNode == NIL    return
    S = ∅
    PUSH(S, currentRoot)
    while S ≠ ∅
        visitingNode = POP(S)
        print visitingNode.key
        if visitingNode.rightChild ≠ NIL
            PUSH(S, visitingNode.rightChild)
        if visitingNode.leftChild ≠ NIL
            PUSH(S, visitingNode.leftChild)
```





Left → Right → Root

Post-order Traversal



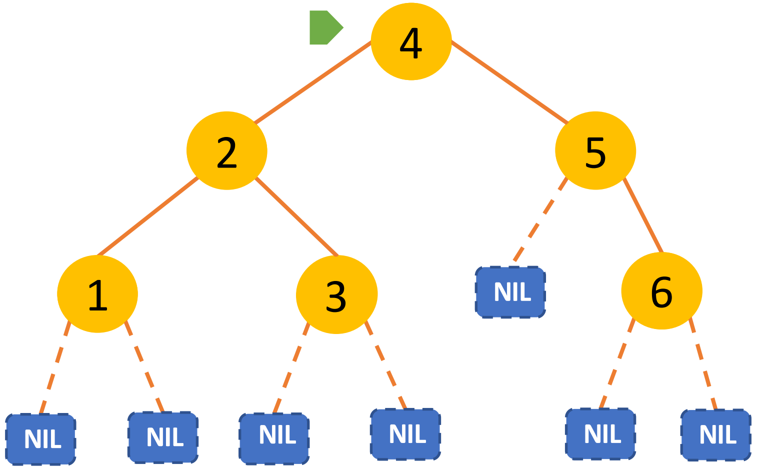
Recursive

```
postOrderWalk(currentRoot)
    if currentRoot == NIL
        return
    postOrderWalk(currentRoot.leftChild)
    postOrderWalk(currentRoot.rightChild)
    print currentRoot.key
```

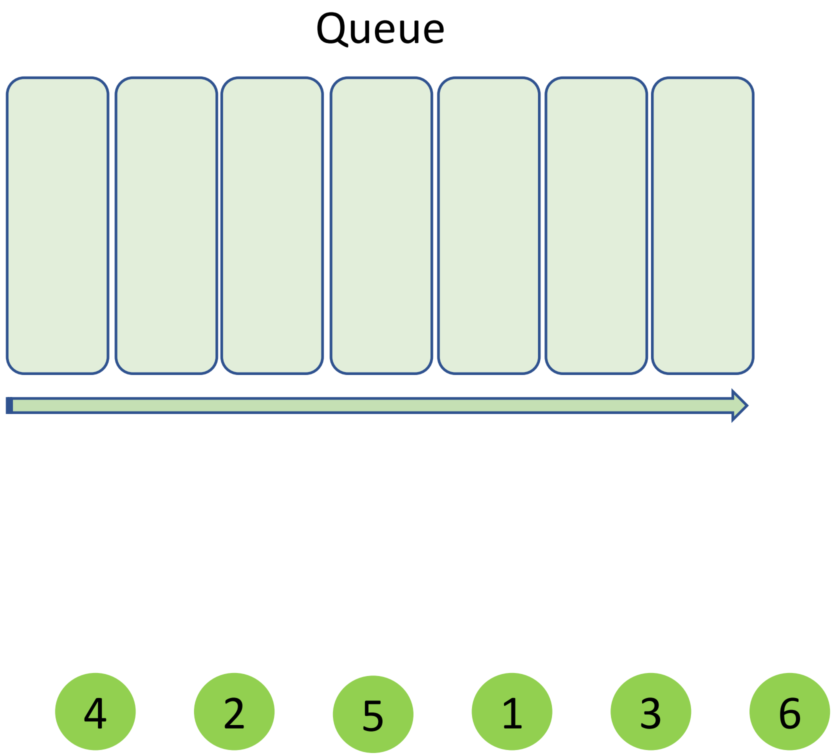
Iterative

```
postOrderWalk (currentRoot)
    S = ∅
    lastVisitedNode = NIL
    while S ≠ ∅ or currentRoot ≠ NIL
        if currentRoot ≠ NIL
            PUSH(S, currentRoot)
            currentRoot = currentRoot.leftChild
        else
            visitingNode = PEEK(S)
            if visitingNode.rightChild ≠ NIL and
                    lastVisitedNode ≠ visitingNode.rightChild
                currentRoot = visitingNode.rightChild
            else
                print visitingNode.key
                lastVisitedNode = POP(S)
```

Level-order Traversal




```
levelOrderWalk (currentRoot)
    Q = ∅
    ENQUEUE(Q, currentRoot)
    while Q ≠ ∅
        currentNode = DEQUEUE(Q)
        print currentNode.key
        if currentNode.leftChild ≠ NIL
            ENQUEUE(Q, currentNode.leftChild )
        if currentNode.rightChild ≠ NIL
            ENQUEUE(Q, currentNode.rightChild )
```

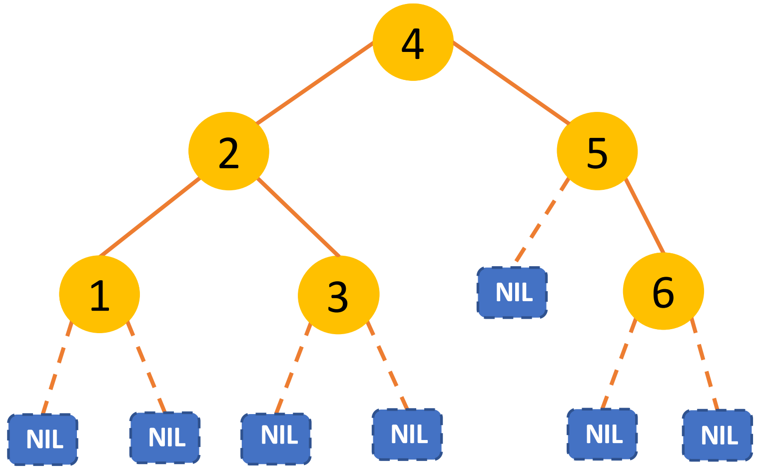


Level-order traversal prints the structure of a tree from top to bottom.

Analysis

inOrderWalk (currentRoot)	*Cost*	*Times*
S = ∅	c_1	1
while S ≠ ∅ or currentRoot ≠ NIL	c_2	$n + d$
if currentRoot ≠ NIL	c_3	$n + d - 1$
PUSH(S, currentRoot)	$O(1)$	n
currentRoot = currentRoot.leftChild	c_4	n
else		
currentRoot = POP(S)	$O(1)$	$d - 1$
print currentRoot.key	c_5	$d - 1$
currentRoot = currentRoot.rightChild	c_6	$d - 1$

If we consider NIL nodes as the leaf nodes, in a complete 2-ary tree, the number of internal nodes $n = 2^h - 1$ and the number of leaf nodes $d = 2^h$, thus $d = n + 1$.



$$\begin{aligned} T(n) &= c_1 + c_2(n + n + 1) + c_3(n + n) + n + c_4 n + n + c_5 n + c_6 n \\ &= (2c_2 + 2c_3 + 2 + c_4 + c_5 + c_6)n + c_1 + c_2 \end{aligned}$$

Definition

$$\Theta\left(g(n)\right) = \{ f(n) : \exists \text{ positive constant } c_1, c_2, \text{and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n), \forall n \ge n_0 \}$$

$$T(n) = \Theta\ (n)$$

Analysis – Substitution Method for Solving Recurrence Relation (1)

代换法求解递归关系

```
inOrderWalk (currentRoot)
    if currentRoot == NIL  return
    inOrderWalk(currentRoot.leftChild)
    print currentRoot.key
    inOrderWalk(currentRoot.rightChild)
```

1

Let $\boldsymbol{T(n)}$ denote the time taken by the recursive function, *inOrderWalk*, on the root of an $\boldsymbol{n}$-node subtree.

2 Asymptotic lower bound

Since the traversal of a tree needs to visit all nodes of the tree, thus $\boldsymbol{T(n) = \Omega(n)}$

3

Let the left subtree of $\boldsymbol{T}$ having $\boldsymbol{i}$ nodes and the right subtree having $\boldsymbol{n - i - 1}$ nodes, thus the time to perform the *inOrderWalk* is bounded by $\boldsymbol{T(n) \leq T(i) + T(n - i - 1) + d}$, for some constant $\boldsymbol{d > 0}$ that denotes an upper bound on the time to execute the body of the function.

4

Show $\boldsymbol{T(n) = O(n)}$, by proving that $\boldsymbol{T(n) \leq (c + d)n + c}$.

$$\boldsymbol{O\left(g(n)\right) = \{ f(n) : \exists \text{ positive constant } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n), \forall\, n \geq n_0 \}}$$

Analysis – Substitution Method for Solving Recurrence Relation (1)

5

$$
\begin{aligned}
\boldsymbol{T(n)} &\le \boldsymbol{T(i) + T(n - i - 1) + d} \\
&= (c+d)i + c + (c+d)(n-i-1) + c + d \\
&= (c+d)i + c + (c+d)n - (c+d)i - c - d + c + d \\
&= (c+d)n + c \qquad \blacksquare
\end{aligned}
$$

6 Removing the constants

$$\boldsymbol{T(n) = O(n)}$$

Theorem:
For any two functions $\boldsymbol{f(n)}$ and $\boldsymbol{g(n)}$, we have $\boldsymbol{f(n) = \Theta(g(n))}$ iff $\boldsymbol{f(n) = O(g(n))}$ and $\boldsymbol{f(n) = \Omega(g(n))}$.

7

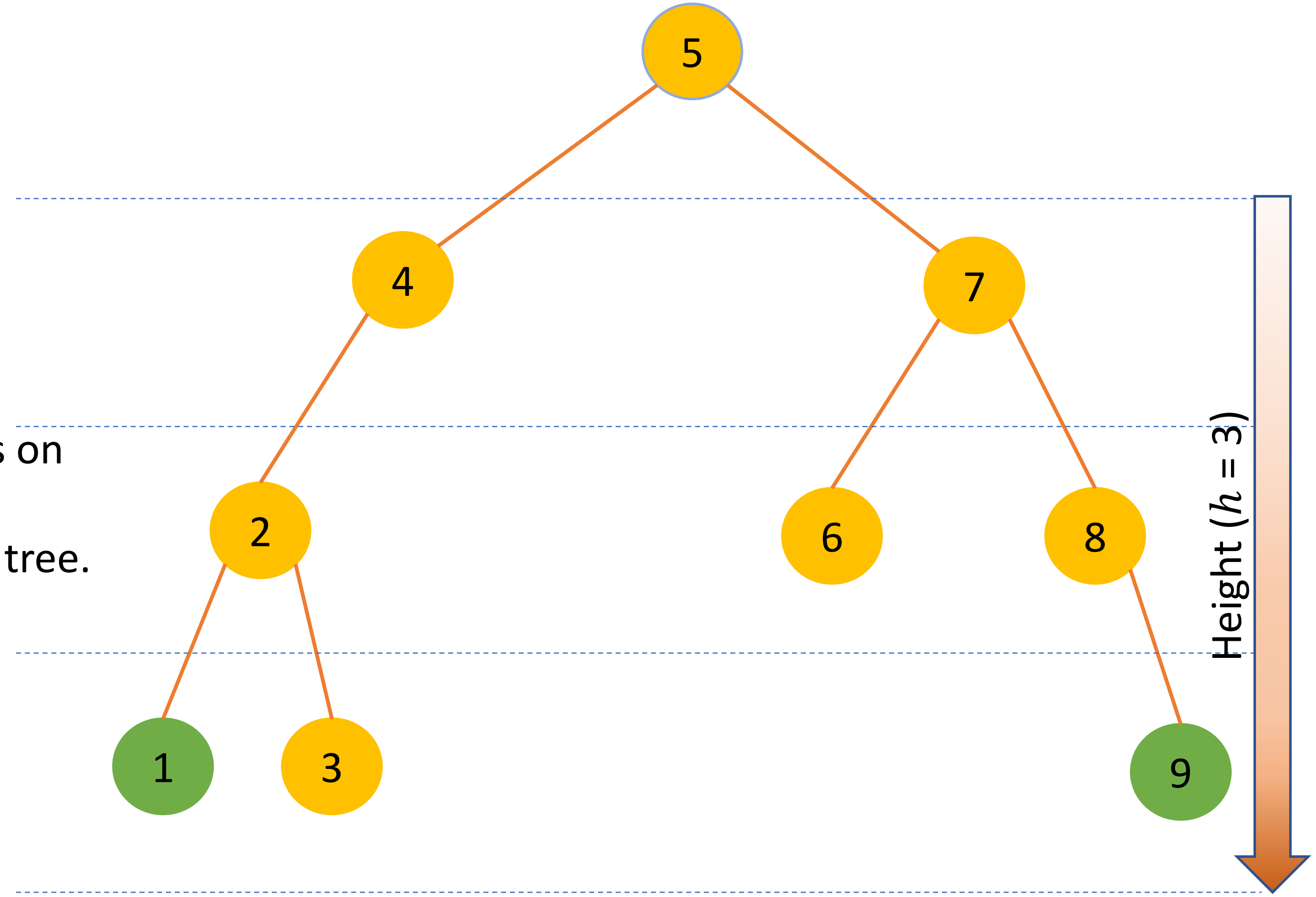
$$\boldsymbol{T(n) = \Theta(n)}$$

Find Minimum and Maximum

findMin(*currentRoot*)
    while *currentRoot.left* ≠ NIL
        currentRoot = *currentRoot.leftChild*
    return *currentRoot*

In a BST, the key with the minimum value is always on the left-most of the tree and the key with the maximum value is always on the right-most of the tree.

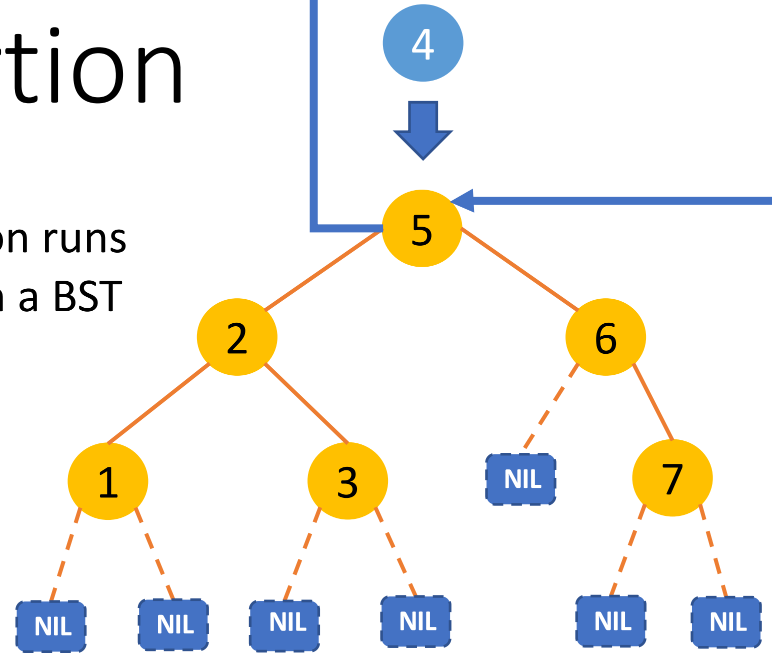
findMax(*currentRoot*)
    while *currentRoot.rightChild* ≠ NIL
        currentRoot = *currentRoot.rightChild*
    return *currentRoot*



Both *findMin( )* and *findMax( )* run in $\boldsymbol{O(h)}$ time on a tree of height $\boldsymbol{h}$ since the sequence of nodes encountered forms a simple path downward from the root.

Insertion

INSERT operation runs in $O(h)$ time on a BST of height h.



5

```
insert(currentRoot, node)
    if currentRoot == NIL
        return node
    if node.key < currentRoot.key
        currentRoot.leftChild = insert(currentRoot.leftChild, node)
    else if node.key > currentRoot.key
        currentRoot.rightChild = insert(currentRoot.rightChild, node)
    else
        currentRoot.value = node.value
    return currentRoot
```

2

```
insert(currentRoot, node)
    if currentRoot == NIL
        return node
    if node.key < currentRoot.key
        currentRoot.leftChild = insert(currentRoot.leftChild, node)
    else if node.key > currentRoot.key
        currentRoot.rightChild = insert(currentRoot.rightChild, node)
    else
        currentRoot.value = node.value
    return currentRoot
```

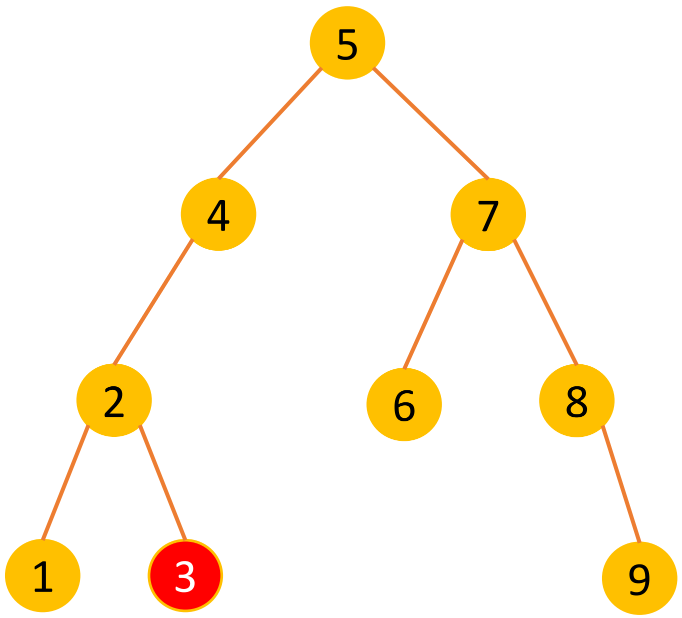
3

```
insert(currentRoot, node)
    if currentRoot == NIL
        return node
    if node.key < currentRoot.key
        currentRoot.leftChild = insert(currentRoot.leftChild, node)
    else if node.key > currentRoot.key
        currentRoot.rightChild = insert(currentRoot.rightChild, node)
    else
        currentRoot.value = node.value
    return currentRoot
```

NIL

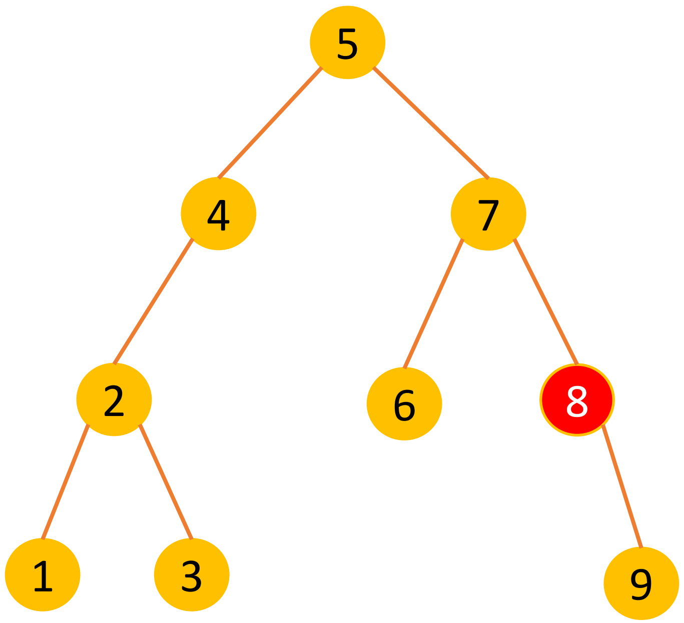
```
insert(currentRoot, node)
    if currentRoot == NIL
        return node
    if node.key < currentRoot.key
        currentRoot.leftChild = insert(currentRoot.leftChild, node)
    else if node.key > currentRoot.key
        currentRoot.rightChild = insert(currentRoot.rightChild, node)
    else
        currentRoot.value = node.value
    return currentRoot
```

Deletion (1)



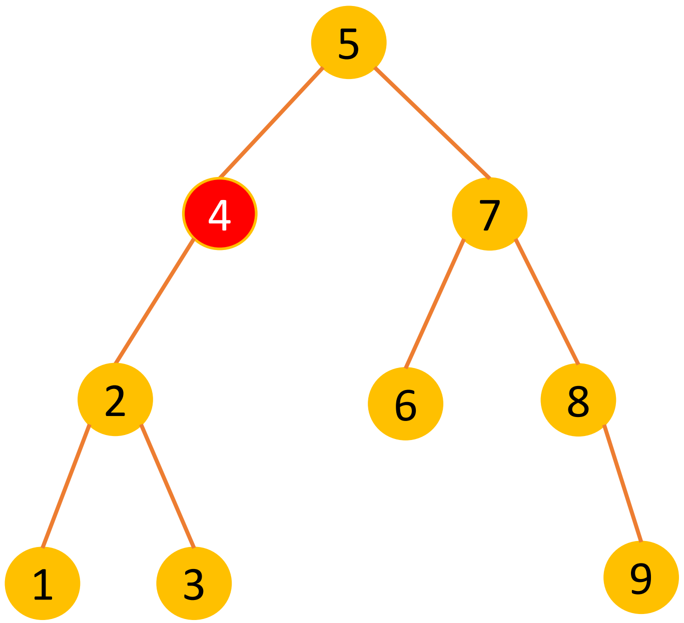
Scenario 1: deleting a leaf node

Solution: Remove the node from the tree immediately



Scenario 2: deleting a node that has no left child

Solution: Remove the node and upgrade its right child



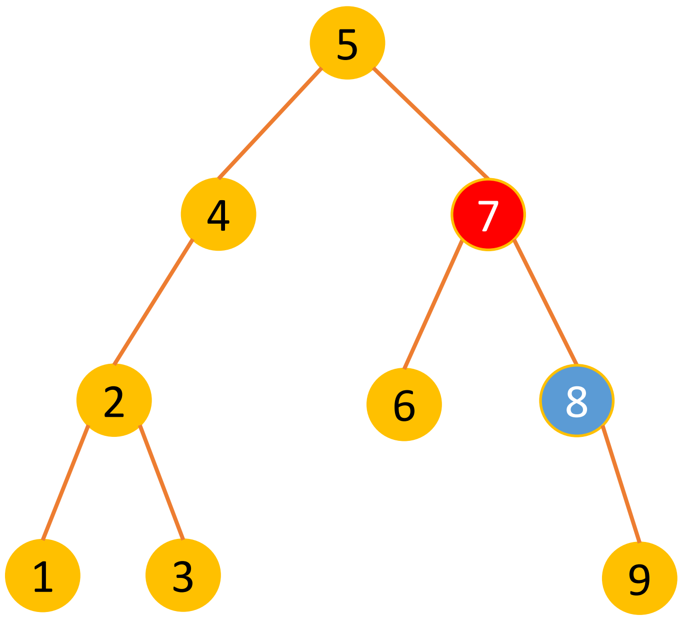
Scenario 3: deleting a node that has no right child

Solution: Remove the node and upgrade its left child

Deletion (2)

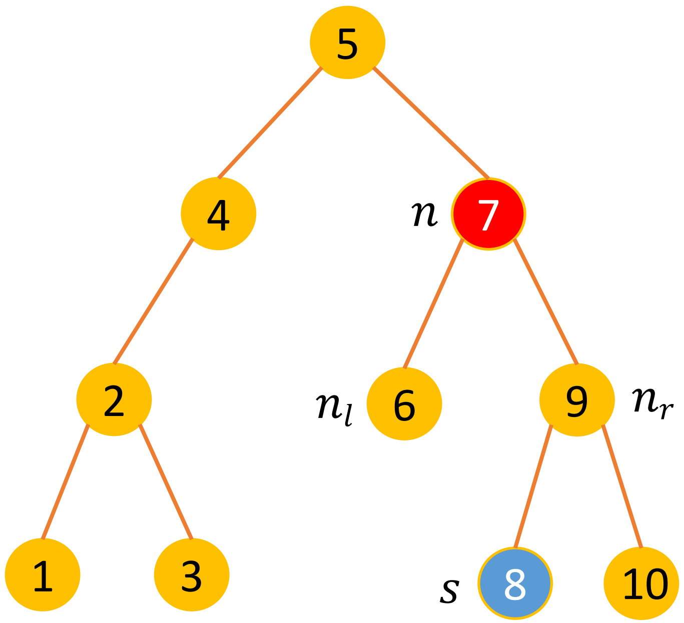
Successor: If all keys are distinct, the *successor* of a node x is the node with the smallest key greater than $x.key$.

Node 8 is the successor of Node 7



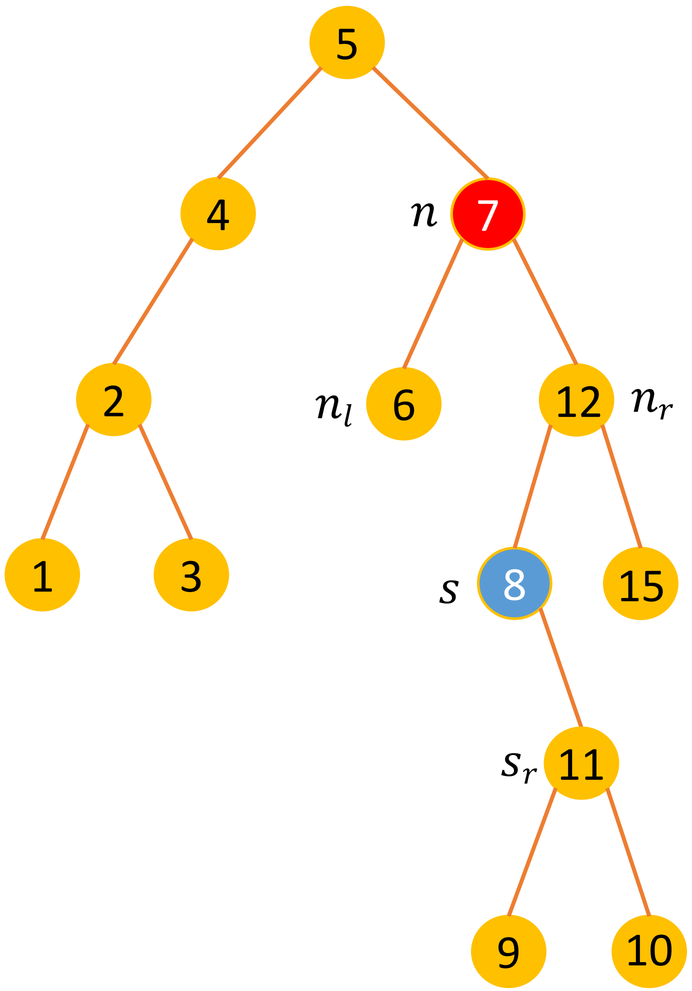
Scenario 4: deleting a node with two children; its right child is its ***successor***

Solution: Remove the node, and upgrade its successor



Scenario 5: deleting a node with two children; its right child is NOT its successor; its successor is a leaf node

Solution: set $\boldsymbol{s}$ to be the parent of $\boldsymbol{n_r}$, use $\boldsymbol{s}$ to replace $\boldsymbol{n}$, set $\boldsymbol{n_l}$ to be the left child of $\boldsymbol{s}$



Scenario 6: deleting a node with two children; its right child is NOT its successor; its successor is NOT a leaf node

Solution: set $\boldsymbol{s}$ to be the parent of $\boldsymbol{n_r}$, use $\boldsymbol{s}$ to replace $\boldsymbol{n}$, set $\boldsymbol{n_l}$ to be the left child of $\boldsymbol{s}$, set $\boldsymbol{s_r}$ to be the left child of $\boldsymbol{n_r}$

Deletion Pseudocode

DELETE operation runs in $\boldsymbol{O}(\boldsymbol{h})$ time on a BST of height $\boldsymbol{h}$.

Moves down to the node that is to be deleted

Deals with the scenario 4, 5, 6

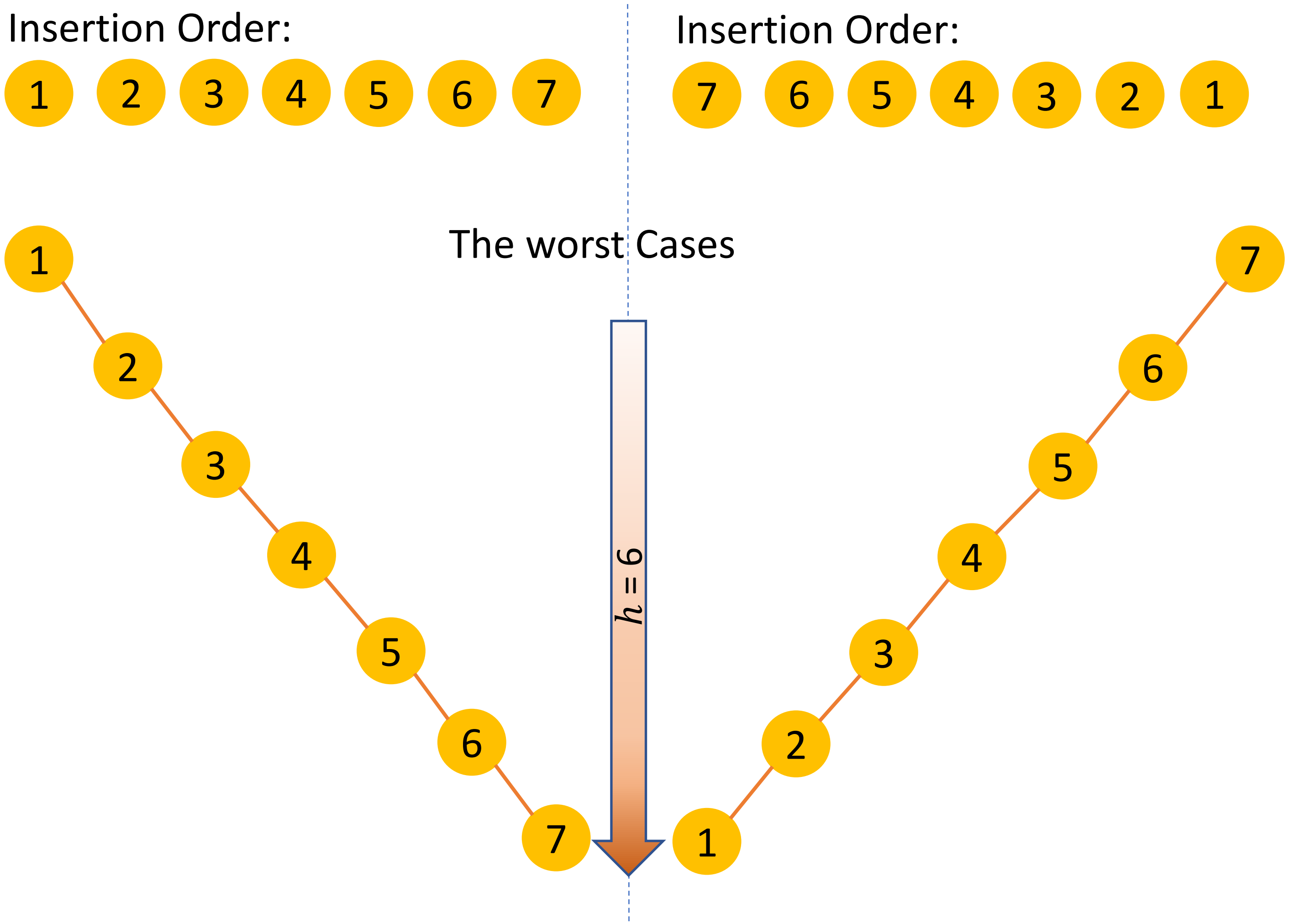
```
delete(currentRoot, key)
    if currentRoot == NIL
        return currentRoot
    if key < currentRoot.key
        currentRoot.leftChild = delete(currentRoot.leftChild, key)
    else if key > currentRoot.key
        currentRoot.rightChild = delete(currentRoot.rightChild, key)
    else if currentRoot.leftChild ≠ NIL and currentRoot.rightChild ≠ NIL
        currentRoot.key = findMin(currentRoot.rightChild).key
        currentRoot.rightChild = delete(currentRoot.rightChild, currentRoot.key)
    else
        currentRoot = (currentRoot.leftChild != null) ? currentRoot.leftChild : currentRoot.rightChild

    return currentRoot
```

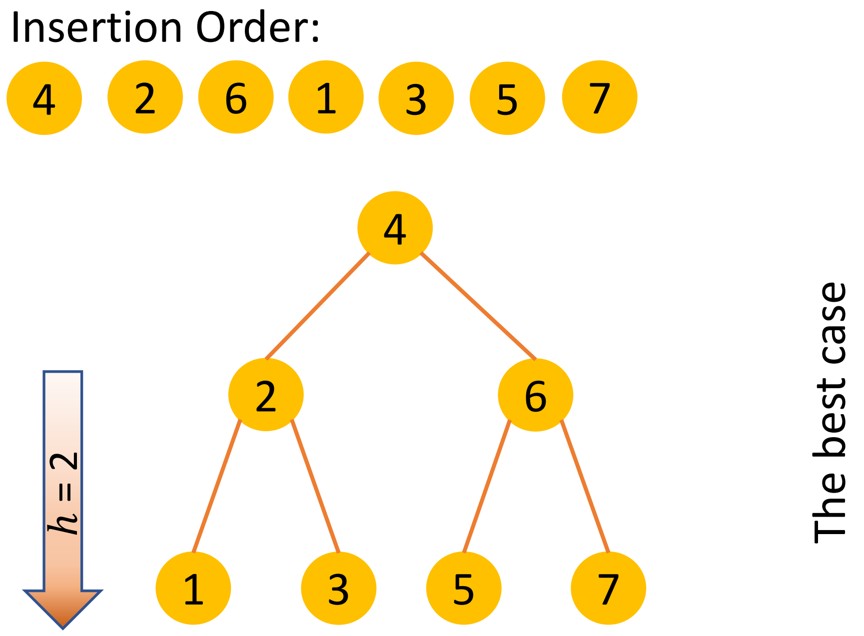
Recursive Base Case

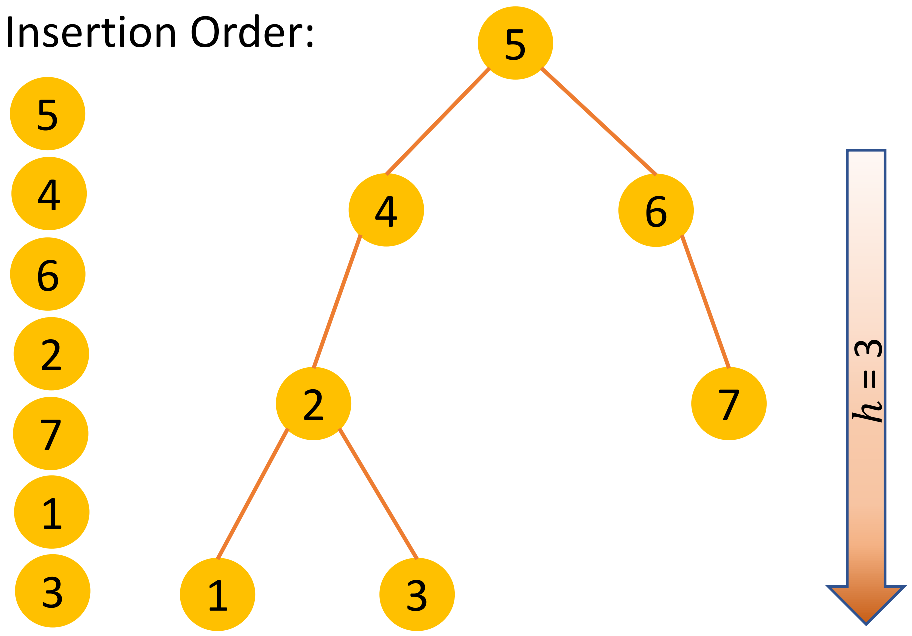
Replace the node to be deleted by its successor

Deals with the scenario 1, 2, 3

Randomly Built BSTs







The height h of a BST varies depending on the order of the nodes inserted.

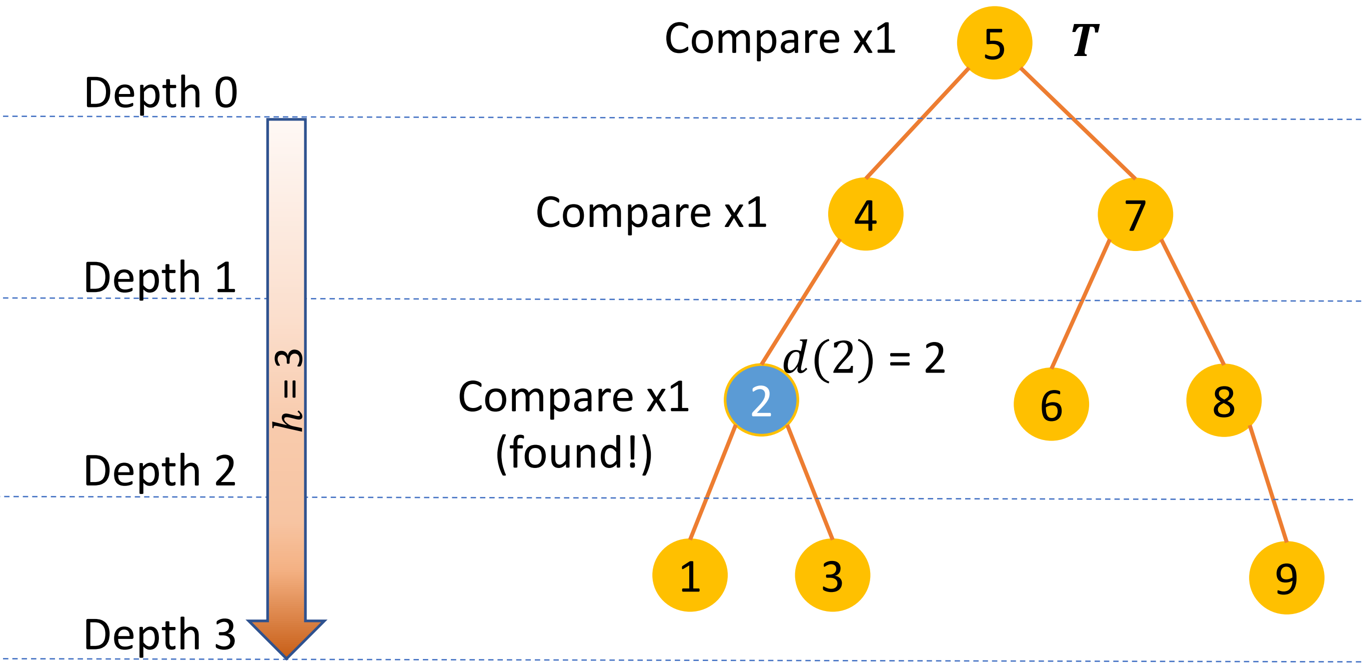
Average Search Hit in a Randomly Built BST (1)

1

Given a BST tree $\boldsymbol{T}$ with $\boldsymbol{N}$ nodes $\{\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots\ \boldsymbol{x_n}\}$, the number of comparisons needed for a search hit on node $\boldsymbol{i}$ is its depth $\boldsymbol{d(x_i) + 1}$.

The total *internal path length* of the BST $\boldsymbol{D(N)} = \sum_{i=1}^{N} \boldsymbol{d(x_i)}$

Internal Path Length: The sum of the depths of all nodes in a tree.



$\boldsymbol{N = 9}$

The Internal Path Length for this example tree $\boldsymbol{T}$

$$
\begin{aligned}
D(9) &= d(1) + d(3) + d(9) + d(2) + d(6) + d(8) + d(4) + d(7) + d(5) \\
&= 3 + 3 + 3 + 2 + 2 + 2 + 1 + 1 + 0 \\
&= 17
\end{aligned}
$$

The depth for the root node of a tree is always $\mathbf{0}$.

The total numbers of comparison needed for examining all nodes is: $\boldsymbol{D(N) + N}$. For this particular configuration of T of N nodes, $D(9) + 9 = 26$,

$D_{avg} = \frac{D(N) + N}{N} = \frac{26}{9} \approx 2.89.$

Average Search Hit in a Randomly Built BST (2)

2 Assuming all permutations of trees with $\boldsymbol{N}$ nodes are equally likely, the expected internal path length is,

$$E[D(N)] = \sum \frac{1}{N!} \sum_{i=1}^{N} d(x_i)$$

3 In a BST $\boldsymbol{T}$, let the left subtree of $\boldsymbol{T}$ having $\boldsymbol{i}$ nodes and the right subtree having $\boldsymbol{N} - \boldsymbol{i} - \boldsymbol{1}$ nodes, forming the recurrence relation,

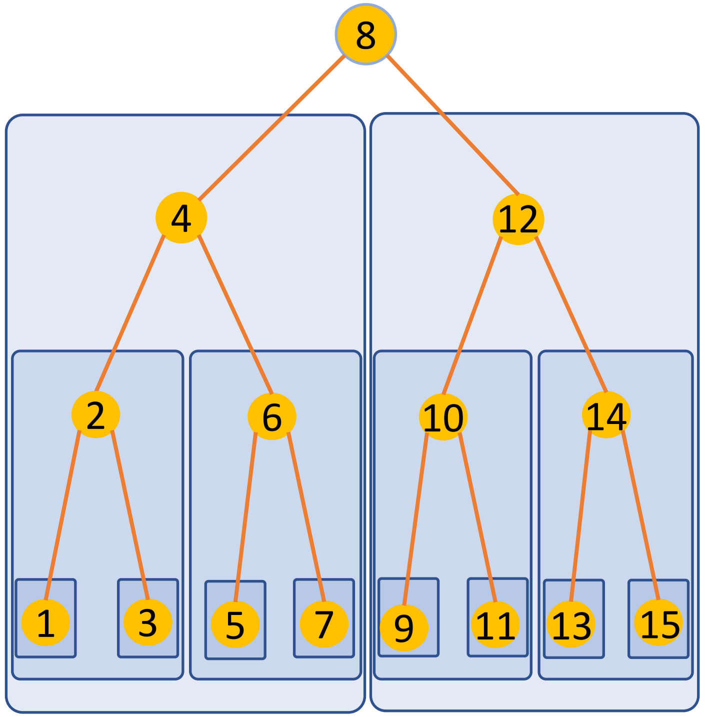
$$D(N) = [D(i) + i] + [D(N - i - 1) + N - i - 1] + 1$$

$$= D(i) + D(N - i - 1) + N$$

4 Randomly built BSTs

The number of nodes on the left subtree and the right subtree depends on the relative *rank* of the root of the subtree $\boldsymbol{R_{N,i}}$

$\boldsymbol{R_{N,i}}$ is equally likely to be any node in a randomly built BST $\boldsymbol{T}$, thus $\boldsymbol{Pr}(\boldsymbol{R_{N,i}}) = \frac{\boldsymbol{1}}{\boldsymbol{N}}$



Rank: the rank of a node is its position in a sorted list of the nodes

Average Search Hit in a Randomly Built BST (3)

$$D(N) = D(i) + D(N - i - 1) + N$$

5 The expected $D(N)$ given i nodes on left and $(N - i - 1)$ nodes on the right is given by,

$$E[D(N|i)] = \sum_{i=0}^{N-1} Pr(R_{N,i})\,[D(i) + D(N - i - 1) + N]$$

$$= \sum_{i=0}^{N-1} \frac{1}{N}[D(i) + D(N - i - 1) + N]$$

$$= \frac{1}{N}\sum_{i=0}^{N-1} D(i) + \frac{1}{N}\sum_{i=0}^{N-1} D(N - i - 1) + \sum_{i=0}^{N-1} \frac{1}{N} N$$

$$= \frac{1}{N}\sum_{i=0}^{N-1} D(i) + \frac{1}{N}\sum_{i=0}^{N-1} D(N - i - 1) + N$$

$$= \frac{2}{N}\sum_{i=0}^{N-1} D(i) + N$$

5.1

$$\sum_{i=0}^{N-1} D(i) = D(0) + D(1) + \ldots + D(N-2) + D(N-1)$$

Reversed order

5.2

$$\sum_{i=0}^{N-1} D(N - i - 1) = D(N-1) + D(N-2) + \ldots + D(1) + D(0)$$

Average Search Hit in a Randomly Built BST (4)

Simplify the notation

5

$$E[D(N|i)] = \frac{2}{N}\sum_{i=0}^{N-1} D(i) + N$$

6 Multiplying by N,

$$ND(N) = 2\sum_{i=0}^{N-1} D(i) + N^2$$

7 Substituting $N-1$ for N,

$$(N-1)D(N-1) = 2\sum_{i=0}^{N-2} D(i) + (N-1)^2$$

8 Subtracting the two equations,

$$ND(N) - (N-1)D(N-1) = \underline{2\sum_{i=0}^{N-1} D(i)} + N^2 - \underline{2\sum_{i=0}^{N-2} D(i)} - (N-1)^2$$

$$\underline{2\sum_{i=0}^{N-1} D(i) - 2\sum_{i=0}^{N-2} D(i) = 2D(N-1)}$$

9 Rearranging,

$$ND(N) = (N+1)D(N-1) + (2N-1)$$

10 Dividing by $N(N+1)$,

$$\frac{D(N)}{N+1} = \frac{D(N-1)}{N} + \frac{2N-1}{N(N+1)}$$

Average Search Hit in a Randomly Built BST (5)

10.1 Telescoping to give,

$$\frac{D(N)}{N+1} = \frac{D(N-1)}{N} + \frac{2N-1}{N(N+1)}$$

$$\frac{D(N-1)}{N} = \frac{D(N-2)}{N-1} + \frac{2(N-1)-1}{(N-1)N}$$

$$\frac{D(N-2)}{N-1} = \frac{D(N-3)}{N-2} + \frac{2(N-2)-1}{(N-2)(N-1)}$$

… …

$$\frac{D(2)}{3} = \frac{D(1)}{2} + \frac{2*2-1}{2*(2+1)}$$

$$\frac{D(1)}{2} = \frac{D(0)}{1} + \frac{2*1-1}{1*(1+1)}$$

$D(0) = 0$, $D(1) = 1$

11 Collecting terms (when the tree T is empty, i.e., $D(0) = 0$),

$$\frac{D(N)}{N+1} = \frac{D(0)}{1} + \frac{2*1-1}{1*(1+1)} + \frac{2*2-1}{2*(2+1)} + \dots + \frac{2(N-2)-1}{(N-2)(N-1)} + \frac{2(N-1)-1}{(N-1)N} + \frac{2N-1}{N(N+1)}$$

$$= \sum_{i=1}^{N} \frac{2i-1}{i(i+1)} = 2\sum_{i=1}^{N} \frac{1}{i+1} - \underline{\sum_{i=1}^{N} \frac{1}{i(i+1)}}$$

11.1 Telescoping to give,

$$\underline{\sum_{i=1}^{N} \frac{1}{i(i+1)} = \sum_{i=1}^{N} \frac{1}{i} - \frac{1}{i+1}}$$

$$= \frac{1}{1} - \frac{1}{1+1} + \frac{1}{2} - \frac{1}{2+1} + \frac{1}{3} - \frac{1}{3+1} + \dots + \frac{1}{N-1} - \frac{1}{N} + \frac{1}{N} - \frac{1}{N+1}$$

$$= 1 - \frac{1}{N+1}$$

$$= \frac{N}{N+1}$$

Average Search Hit in a Randomly Built BST (6)

12

$$\frac{D(N)}{N+1} = 2\sum_{i=1}^{N}\frac{1}{i+1} - \frac{N}{N+1}$$

A n^{th} harmonic number,

$$H_n = \sum_{i=1}^{n}\frac{1}{i} \approx \int_1^n \frac{1}{x}dx = \ln(n)$$

12.1 Compare the first term with the n^{th} harmonic number

$$H_n = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-1} + \frac{1}{n}$$

$$\sum_{i=1}^{n}\frac{1}{i+1} = \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-1} + \frac{1}{n} + \frac{1}{n+1}$$

$$\sum_{i=1}^{N}\frac{1}{i+1} = Hn - 1 + \frac{1}{N+1}$$

$$\sum_{i=1}^{N}\frac{1}{i+1} \approx \ln(N) - 1 + \frac{1}{N+1}$$

13

$$\frac{D(N)}{N+1} = 2\ln(N) - 2 + \frac{2}{N+1} - \frac{N}{N+1} = 2\ln(N) - \frac{3N}{N+1}$$

$$\Rightarrow D(N) = 2(N+1)\ln(N) - 3N$$

Average Search Hit in a Randomly Built BST (7)

$$\boldsymbol{D(N) = 2(N+1)\ln(N) - 3N}$$

14

$$\boldsymbol{D(N) \approx 1.39\,(N+1)}\log_2 \boldsymbol{N\; - 3N}$$

15 Average number of comparisons for a node search in a randomly build BST of N nodes is given by,

$$\boldsymbol{\frac{D(N)}{N} \approx 1.39\,\frac{N+1}{N}}\log_2 \boldsymbol{N\; - 3}$$

16 Ignoring the constants

The average-case search time is $\boldsymbol{\Theta}\,(\log_2 \boldsymbol{N})$.

$$\boldsymbol{\Theta\,(g(n)) = \{\, f(n) : \exists\ positive\ constant\ c_1, c_2, and\ n_0\ such\ that\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n), \forall\, n \ge n_0 \,\}}$$

$$\boldsymbol{n_0 = 4}$$

B

$$\boldsymbol{\log_b a = \frac{\log_c a}{\log_c b}} \qquad \ln(N) = \frac{\log_2 N}{\log_2 e}$$

$$\log_2 \boldsymbol{e} = \frac{\ln \boldsymbol{e}}{\ln \boldsymbol{2}} \qquad \ln \boldsymbol{e = 1}$$

$$\ln(N) = \ln \boldsymbol{2}\;\log_2 N$$
$$= 0.693 \log_2 N$$

$$\ln \boldsymbol{2} \approx 0.693$$

C

$$\lim_{n\to\infty} \frac{N+1}{N} = 1$$

Applications of Binary Tree – Expression Trees (1)

- An algebraic expression can be expressed in ***infix***, ***prefix*** and ***postfix*** notations, e.g.,
  - Infix Notation: **a + b * (c + d) – f**
    - *Operand Operator Operand*
  - Prefix Notation (Polish Notation): **– + a * b + c d f**
    - *Operator Operand Operand*
  - Postfix Notation (Reverse Polish Notation): **a b c d + * + f –**
    - *Operand Operand Operator*

In the prefix and postfix notations, the expression is unambiguous without parentheses.

Infix	**a + b * (c + d) – f**
Fully parenthesize	**((a + (b * (c + d))) – f)**
Move operators to the right	**((a (b (c d) +) *) + f) –**
Remove the parentheses	**a b c d + * + f –** Postfix

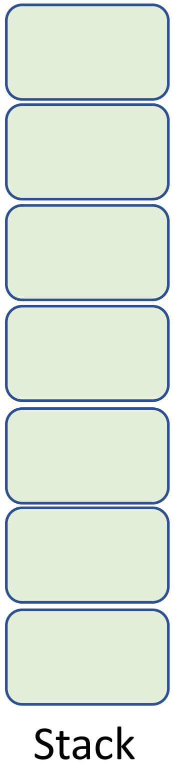
Infix	**a + b * (c + d) – f**
Fully parenthesize	**((a + (b * (c + d))) – f)**
Move operators to the left	**– (+ (a * (b + (c d))) f)**
Remove the parentheses	**– + a * b + c d f** Prefix

A *binary tree* can be used to represent an algebraic expression that involves the binary arithmetic operators. The leaves of an expression tree are operands and internal nodes are operators.

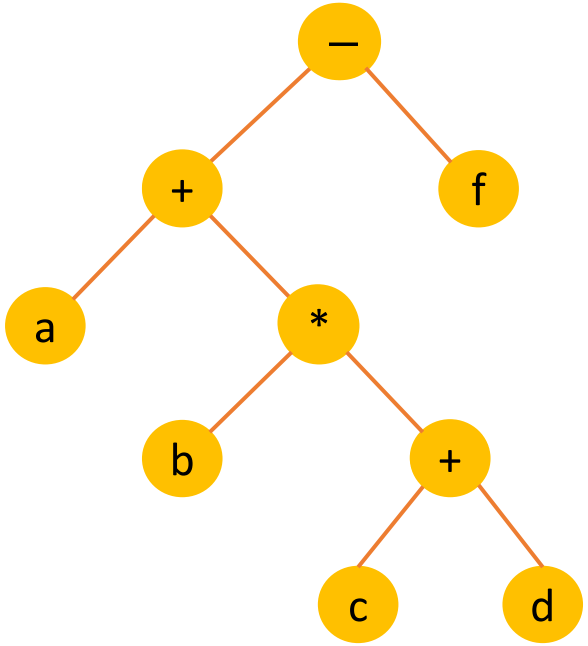
Applications of Binary Tree – Expression Trees (2)

Postfix: a b c d + * + f –

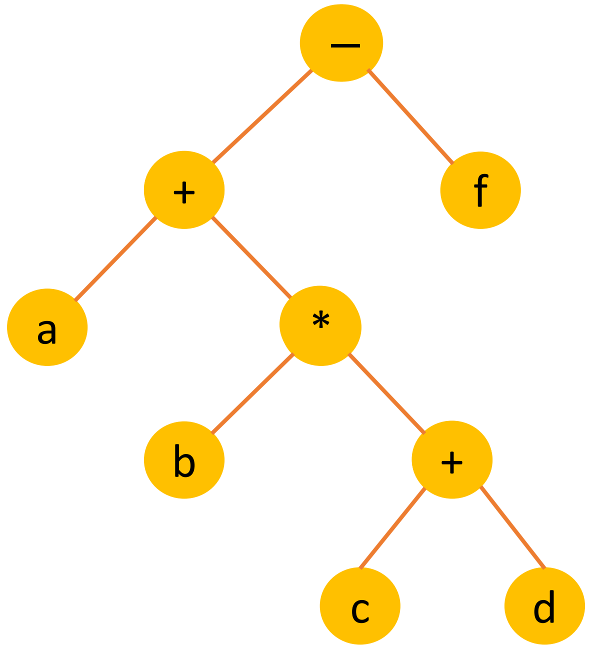
```
postfixBT (exp)
    S = Ø
    Node opd1, opd2, currentRoot
    for i to LEN(exp)
        currentRoot = new Node(exp[ i ])
        if not operator
            PUSH(S, currentRoot)
        else
            opd1 = POP(S)
            opd2 = POP(S)
            currentRoot.left = opd2
            currentRoot.right = opd1
            PUSH(S, CurrentRoot)
    POP(S)
```



- Infix Notation: **a + b * (c + d) – f**
- Prefix Notation: **– + a * b + c d f**
- Postfix Notation: **a b c d + * + f –**



Applications of Binary Tree – Expression Trees (3)



- Infix Notation: **a + b * (c + d) – f**
  - **Inorder traversal**
- Prefix Notation: **– + a * b + c d f**
  - **Preorder traversal**
- Postfix Notation: **a b c d + * + f –**
  - **Postorder traversal**

Applications of Binary Tree – Huffman Coding(1)

A message from Claude Shannon:

The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point.

The information represented in ASCII code stored in computer:

01010100 01101000 01100101 00100000 01100110 01110101 01101110 01100100 01100001 01101101 01100101 01101110 01110100 01100001 01101100 00100000 01110000 01110010 01101111 01100010 01101100 01100101 01101101 00100000 01101111 01100110 00100000 01100011 01101111 01101101 01101101 01110101 01101110 01101001 01100011 01100001 01110100 01101001 01101111 01101110 00100000 01101001 01110011 00100000 01110100 01101000 01100001 01110100 00100000 01101111 01100110 00100000 01110010 01100101 01110000 01110010 01101111 01100100 01110101 01100011 01101001 01101110 01100111 00100000 01100001 01110100 00100000 01101111 01101110 01100101 00100000 01110000 01101111 01101001 01101110 01110100 00100000 01100101 01101001 01110100 01101000 01100101 01110010 00100000 01100101 01111000 01100001 01100011 01110100 01101100 01111001 00100000 01101111 01110010 00100000 01100001 01110000 01110000 01110010 01101111 01111000 01101001 01101101 01100001 01110100 01100101 01101100 01111001 00100000 01100001 00100000 01101101 01100101 01110011 01110011 01100001 01100111 01100101 00100000 01110011 01100101 01101100 01100101 01100011 01110100 01100101 01100100 00100000 01100001 01110100 00100000 01100001 01101110 01101111 01110100 01101000 01100101 01110010 00100000 01110000 01101111 01101001 01101110 01110100 00101110

The message contains **145** characters and represented using (145 * 8) = **1160** bits in computer.

ASCII Code: American Standard Code for Information Interchange

Source of the message: Shannon, C. E. (1948). A mathematical theory of communication. *The Bell system technical journal*, *27*(3), 379-423.

Applications of Binary Tree – Huffman Coding(2)

> **The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point.**

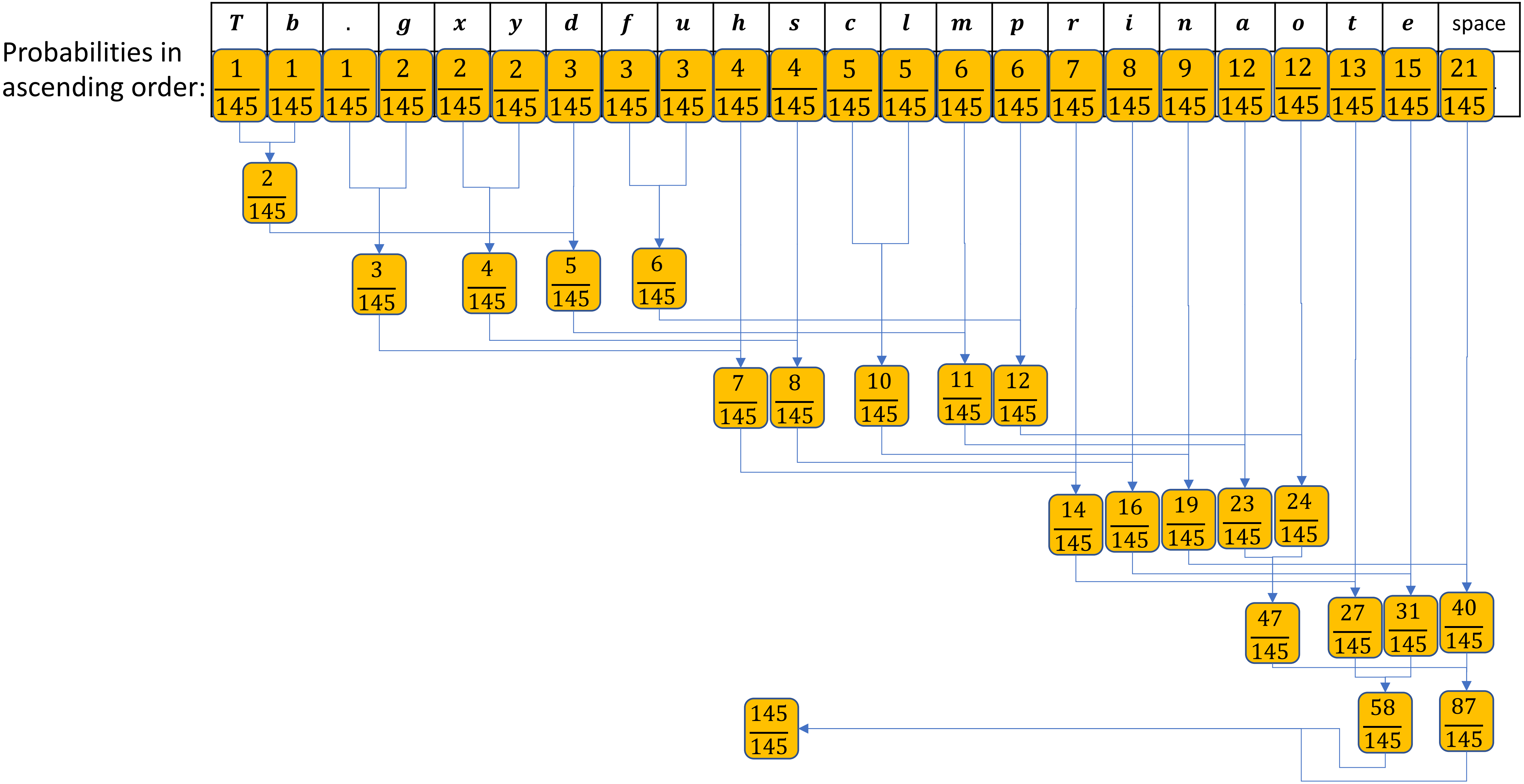
According to Shannon's Information Entropy: $\boldsymbol{H(X) = -\sum_{i=1}^{n} p(x_i)\ \log_2 p(x_i)}$

Alphabets(X):	T	a	b	c	d	e	f	g	h	i	l	m	n	o	p	r	s	t	u	x	y	.	space
Frequency:	1	12	1	5	3	15	3	2	4	8	5	6	9	12	6	7	4	13	3	2	2	1	21
Probability:	$\frac{1}{145}$	$\frac{12}{145}$	$\frac{1}{145}$	$\frac{5}{145}$	$\frac{3}{145}$	$\frac{15}{145}$	$\frac{3}{145}$	$\frac{2}{145}$	$\frac{4}{145}$	$\frac{8}{145}$	$\frac{5}{145}$	$\frac{6}{145}$	$\frac{9}{145}$	$\frac{12}{145}$	$\frac{6}{145}$	$\frac{7}{145}$	$\frac{4}{145}$	$\frac{13}{145}$	$\frac{3}{145}$	$\frac{2}{145}$	$\frac{2}{145}$	$\frac{1}{145}$	$\frac{21}{145}$

$$\boldsymbol{H(X) = -\sum_{i=1}^{23} p(x_i)\ \log_2 p(x_i)} = -\left(\frac{1}{145}\log_2\frac{1}{145} + \frac{12}{145}\log_2\frac{12}{145} + \frac{1}{145}\log_2\frac{1}{145} + \ \dots + \frac{1}{145}\log_2\frac{1}{145} + \frac{21}{145}\log_2\frac{21}{145}\right) \approx 4.09\ bits$$

The same message can be presented using 4.09 * 145 = **593.05** bits in theory.

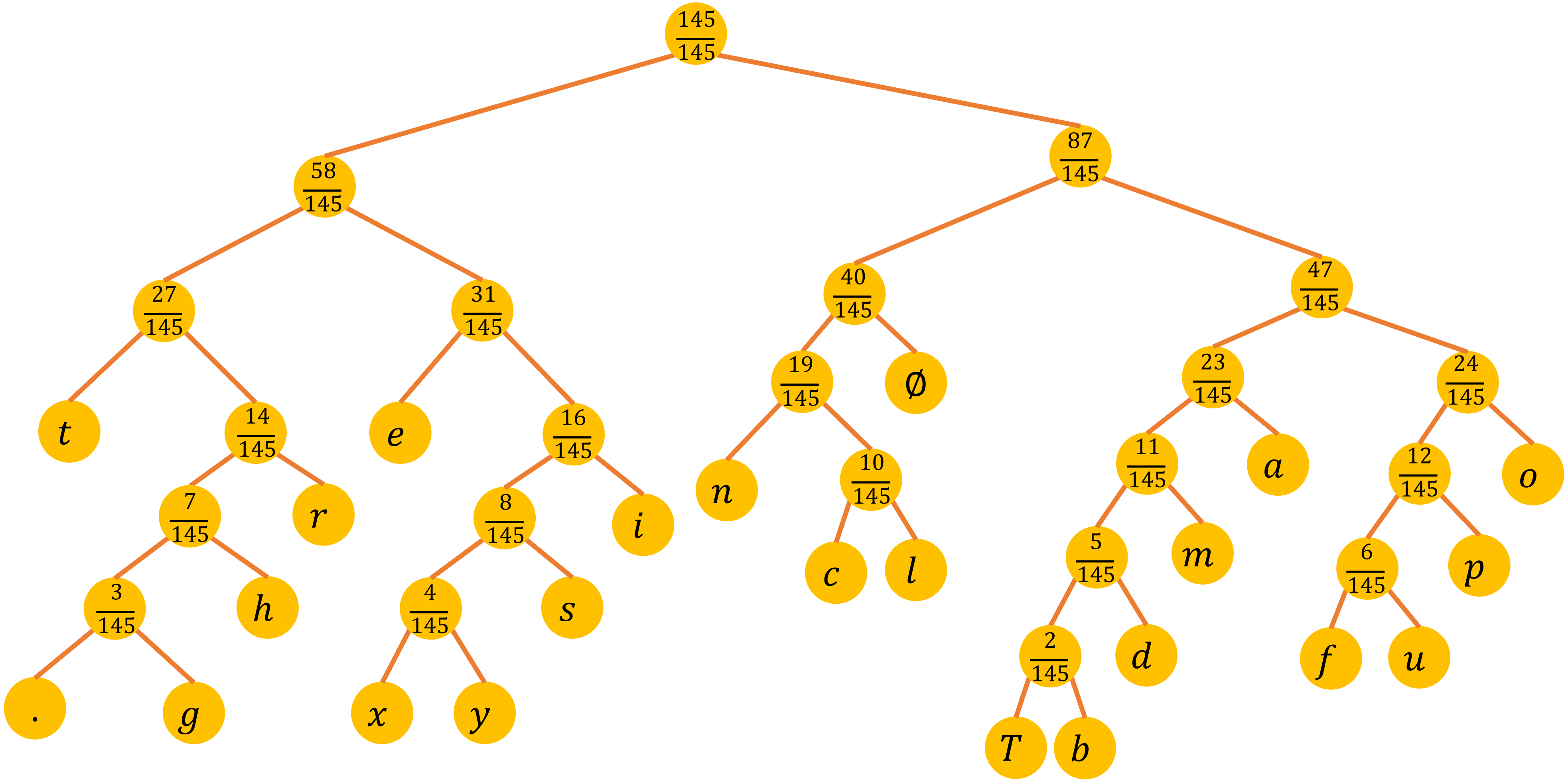
Applications of Binary Tree – Huffman Coding(3)



Applications of Binary Tree – Huffman Coding(4)

Probabilities in ascending order:

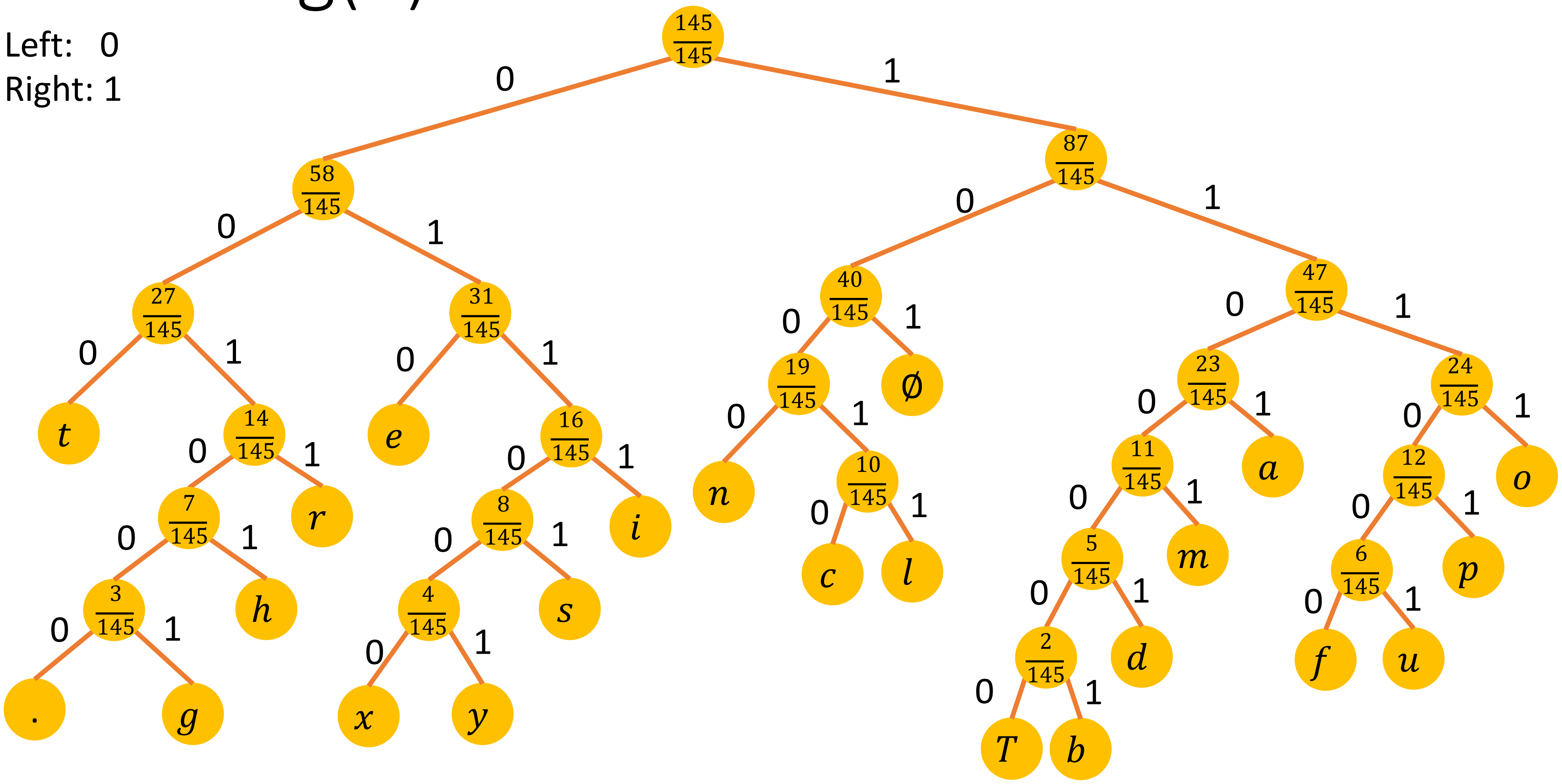
T	b	.	g	x	y	d	f	u	h	s	c	l	m	p	r	i	n	a	o	t	e	space
$\frac{1}{145}$	$\frac{1}{145}$	$\frac{1}{145}$	$\frac{2}{145}$	$\frac{2}{145}$	$\frac{2}{145}$	$\frac{3}{145}$	$\frac{3}{145}$	$\frac{3}{145}$	$\frac{4}{145}$	$\frac{4}{145}$	$\frac{5}{145}$	$\frac{5}{145}$	$\frac{6}{145}$	$\frac{6}{145}$	$\frac{7}{145}$	$\frac{8}{145}$	$\frac{9}{145}$	$\frac{12}{145}$	$\frac{12}{145}$	$\frac{13}{145}$	$\frac{15}{145}$	$\frac{21}{145}$



Huffman Code: Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, *40*(9), 1098-1101.

Applications of Binary Tree – Huffman Coding(5)

Left: 0
Right: 1



Symbol	Codeword
Space	1 0 1
e	0 1 0
t	0 0 0
a	1 1 0 1
i	0 1 1 1
n	1 0 0 0
o	1 1 1 1
r	0 0 1 1
c	1 0 0 1 0
h	0 0 1 0 1
l	1 0 0 1 1
m	1 1 0 0 1
p	1 1 1 0 1
s	0 1 1 0 1
d	1 1 0 0 0 1
f	1 1 1 0 0 0
u	1 1 1 0 0 1
x	0 1 1 0 0 0
•	0 0 1 0 0 0
g	0 0 1 0 0 1
y	0 1 1 0 0 1
T	1 1 0 0 0 0 0
b	1 1 0 0 0 0 1

Applications of Binary Tree – Huffman Coding(6)

The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point.

Compression (**599** bits)

1100000 00101 010 101 111000 111001 1000 110001 1101 11001 010 1000 000 1101 10011 101 11101
T h e sss f u n d a m e n t a l sss p

0011 1111 1100001 10011 010 11001 101 1111 111000 101 10010 1111 11001 11001 111001 1000 0111
r o b l e m sss o f sss c o m m u n i

10010 1101 000 0111 1111 1000 101 0111 01101 101 000 00101 1101 000 101 1111 111000 101 0011
c a t i o n sss i s sss t h a t sss o f sss r

010 11101 0011 1111 110001 111001 10010 0111 1000 001001 101 1101 000 101 1111 1000 010 101
e p r o d u c i n g sss a t sss o n e sss

11101 1111 0111 1000 000 101 010 0111 000 00101 010 0011 101 010 011000 1101 10010 000 10011
p o i n t sss e i t h e r sss e x a c t l

011001 101 1111 0011 101 1101 11101 11101 0011 1111 011000 0111 11001 1101 000 010 10011
y sss o r sss a p p r o x i m a t e l

011001 101 1101 101 11001 010 01101 01101 1101 001001 010 101 01101 010 10011 010 10010 000
y sss a sss m e s s a g e sss s e l e c t

010 110001 101 1101 000 101 1101 1000 1111 000 00101 010 0011 101 11101 1111 0111 1000 000
e d sss a t sss a n o t h e r sss p o i n t

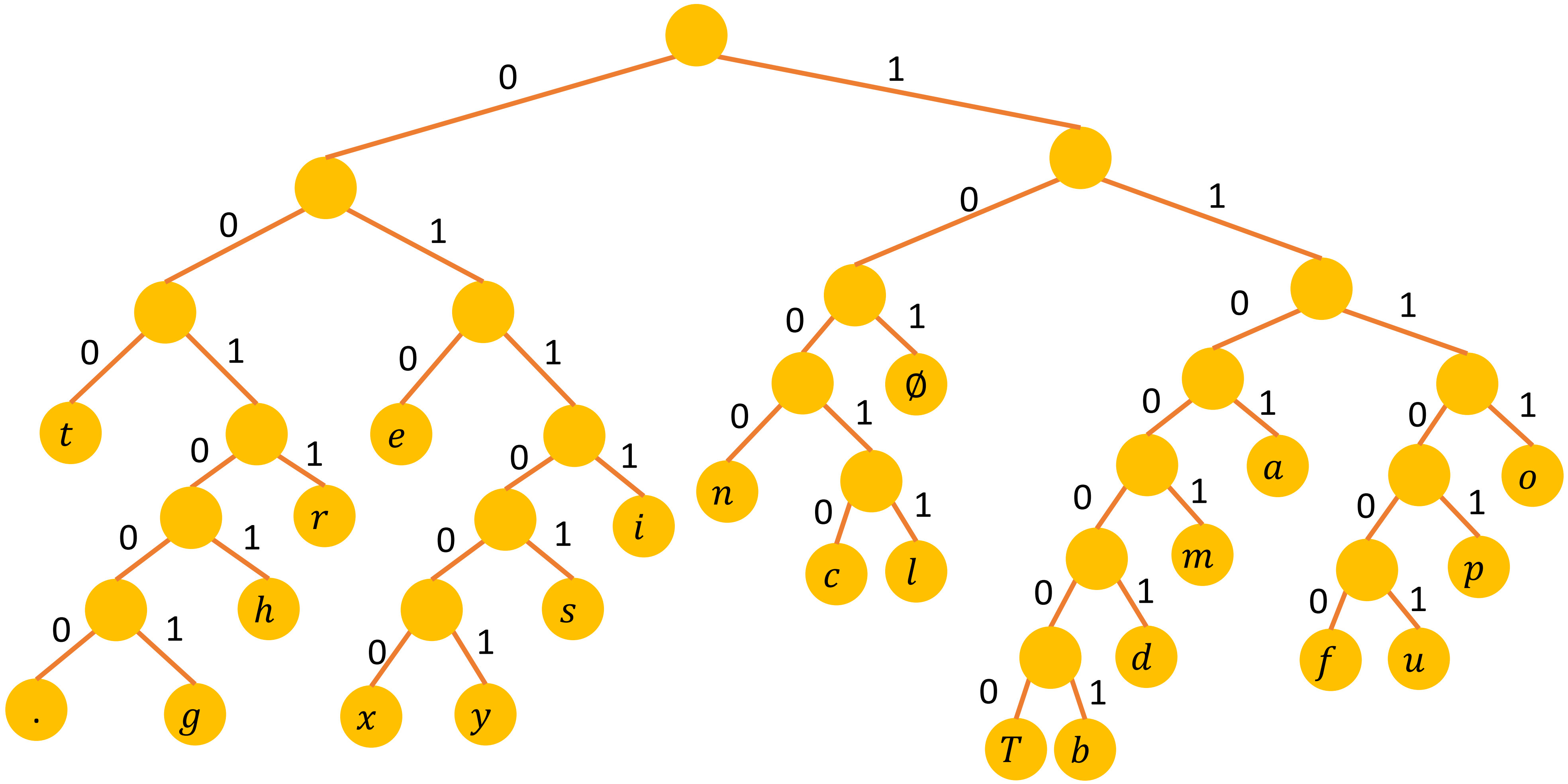
001000
.

Symbol	Codeword
Space	1 0 1
e	0 1 0
t	0 0 0
a	1 1 0 1
i	0 1 1 1
n	1 0 0 0
o	1 1 1 1
r	0 0 1 1
c	1 0 0 1 0
h	0 0 1 0 1
l	1 0 0 1 1
m	1 1 0 0 1
p	1 1 1 0 1
s	0 1 1 0 1
d	1 1 0 0 0 1
f	1 1 1 0 0 0
u	1 1 1 0 0 1
x	0 1 1 0 0 0
•	0 0 1 0 0 0
g	0 0 1 0 0 1
y	0 1 1 0 0 1
T	1 1 0 0 0 0 0
b	1 1 0 0 0 0 1

Can you decode the message starting from a randomly selected point?

Applications of Binary Tree – Huffman Coding(7)

Decompression:
Starting from the beginning of the compressed file, and then follow the binary tree to decode.



**110000000101010101111100
011100110001100011101111
001010100000011011001111
011110100111111111000011
001101011001101111111110
001011001011111110011100
111100110000111110010110
100001111111110001010111
011011010000010111010000
101111111110001010011010
111010011111111100011110
011001001111000000100110
111010001011111110000101
011110111110111110000001
010100111000000101010001
110101001100011011001000
001001101100110111111001
110111011110111110100111
111011000011111100111010
000101001101100110111101
101110010100110101101111
010010010101010110101010
001101010010000010110000
110111010001011101100011
111000000101010001110111
101111101111100000000100
0**