

# Binary Search Trees & Forests

Joseph Chuang-Chieh Lin (林莊傑)

Department of Computer Science & Engineering,  
National Taiwan Ocean University

Fall 2024



# Outline

- 1 Binary Search Trees
- 2 Forests

# Outline

1 Binary Search Trees

2 Forests

# Binary Search Tree (BST) (1/2)

For searching, insertions and deletions...

Binary search tree provides a better performance than any of the data structures studied so far.



## Binary Search Tree (BST) (2/2)

### BST

A binary search tree (BST) is a binary tree which may be empty.

If it is not empty, then it satisfies the following properties:

- Each node has **exactly one key** and the **keys in the tree are distinct**.
- The keys (if any) in the **left** subtree are **smaller** than the key in the root.
- The keys (if any) in the **right** subtree are **larger** than the key in the root.
- The left and right subtrees are also binary search tree.



## Binary Search Tree (BST) (2/2)

### BST

A binary search tree (BST) is a binary tree which may be empty.

If it is not empty, then it satisfies the following properties:

- Each node has **exactly one key** and the **keys in the tree are distinct**.
- The keys (if any) in the **left** subtree are **smaller** than the key in the root.
- The keys (if any) in the **right** subtree are **larger** than the key in the root.
- The left and right subtrees are also binary search tree.
  - a flavor of recursion?

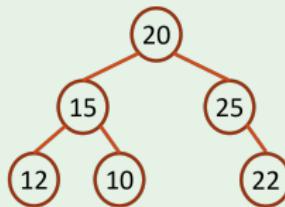


# Examples of BST

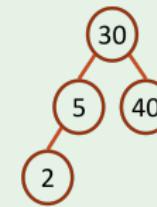
- Which one of the following trees is BST? Which one is NOT?

# Examples of BST

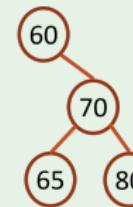
- Which one of the following trees is BST? Which one is NOT?



(a)



(b)



(c)



(d)

## Recursive Search of a BST

```
element* search(treePointer root, int key) {  
    /* return a pointer to the node that contains key,  
       if there is no such node, return NULL. */  
    if (!root) return NULL;  
    if (k == root->data.key) return &(root->data);  
    if (k < root->data.key)  
        return search(root->leftChild, k);  
    return search(root->rightChild, k);  
}
```

- The time complexity of the search function is  $O(h)$ , where  $h$  is the height of the binary search tree.

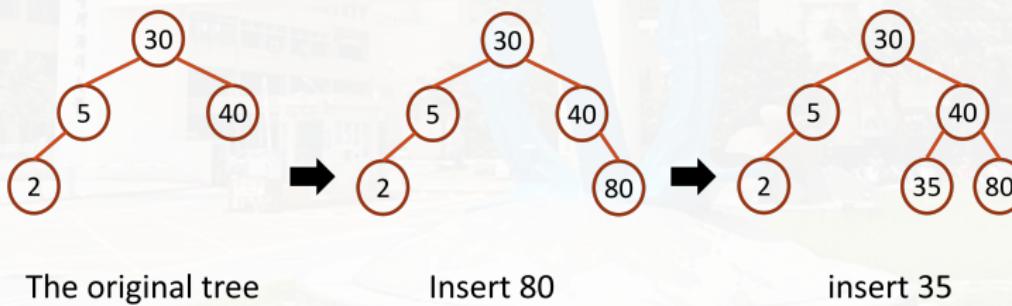
## Iterative Search of a BST

```
element* iterSearch(treePointer tree, int k) {
    /* return a pointer to the node that contains key,
       if there is no such node, return NULL. */
    while (tree) {
        if (k == tree->data.key) return &(tree->data);
        if (k < tree->data.key)
            tree = tree->leftChild;
        else
            tree = tree->rightChild;
    }
    return NULL;
}
```

- The time complexity of the search function is  $O(h)$ , where  $h$  is the height of the binary search tree.

# Inserting into a BST

- Verify that the key is different from those existing elements.
- If the search is **unsuccessful**, insert the element at the point **the search terminated**.



# A Modified Searching Function

```
modifiedSearch(treePointer *node, int k)
```

- If the BST is empty, then return NULL.
- If the key  $k$  exists in the BST, return NULL.
- Otherwise, return the pointer of the last node in the BST.

# Inserting a Dictionary Pair into a BST

```
void insert(treePointer *node, int k, iType theItem) {  
    /* If k is in the tree pointed at by "node", do nothing;  
       otherwise, add a new node with data = (k, theItem) */  
    treePointer ptr, temp = modifiedSearch(*node, k);  
    if (temp || !(*node)) { /* k is not in the tree */  
        ptr = (treePointer)malloc(sizeof(node));  
        ptr->data.key = k;  
        ptr->data.item = theItem;  
        ptr->leftChild = ptr->rightChild = NULL;  
        if (*node) /* insert as child of temp */  
            if (k < temp->data.key)  
                temp->leftChild = ptr;  
            else  
                temp->rightChild = ptr;  
        else // the case of empty tree  
            *node = ptr;  
    }  
}
```

# Deletion from a BST

- Case 1: **leaf**
  - delete the node and set the pointer from the parent node to NULL.
- Case 2: **having only one child**:
  - delete the node and change the pointer from the parent node to the single-child node.
- Case 3: **having two children**:
  - replaced by the **largest** element in its **left** subtree, or
  - replaced by the **smallest** element in its **right** subtree.

# Illustration (Case 1 & 2)

Case 1:

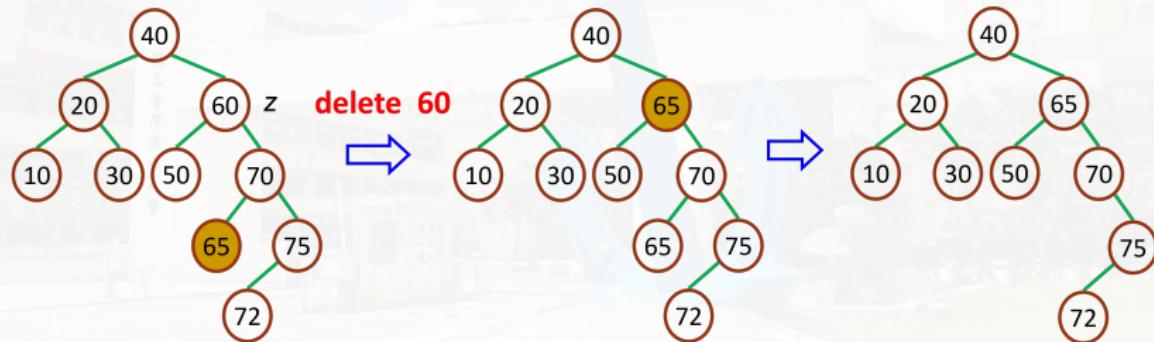


Case 2:



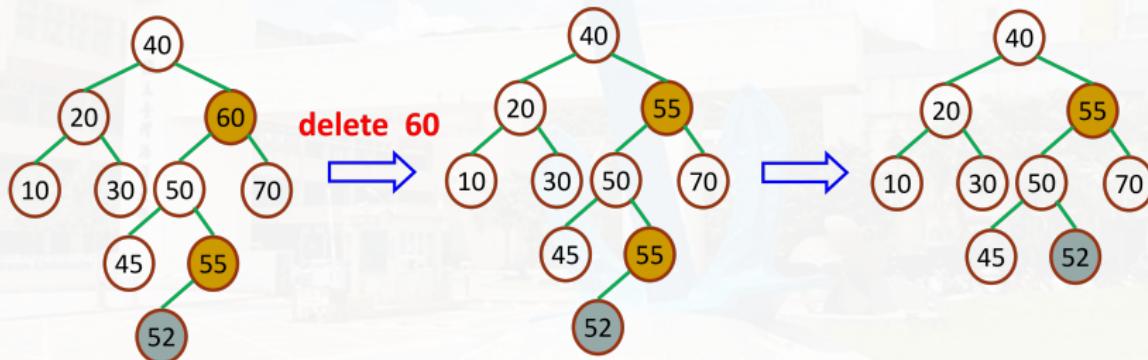
# Illustration (Case 3)

Case 3:



# Illustration (Case 3)

Case 3:



# Time Complexity Analysis of Deleting a Node in a BST

- The case: Deleting a nonleaf node that has two children.
- We can verify (Exercise) that, in both ways, it is originally in a node with a degree of at most one.
  - Check the largest and smallest elements in a subtree.
- The time complexity for case 3 is  $O(h)$  ( $h$ : the height of the BST).
- A deletion can be performed in  $O(h)$  time.

# Outline

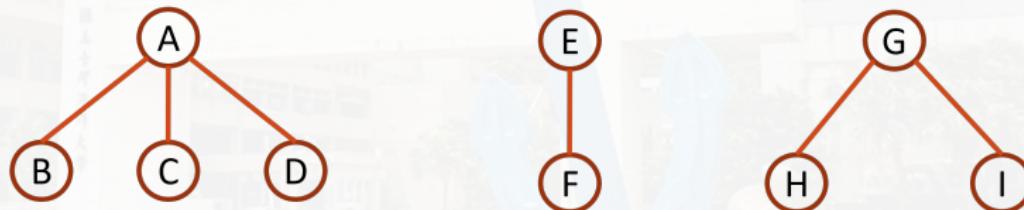
1 Binary Search Trees

2 Forests

# Forest

## Forest

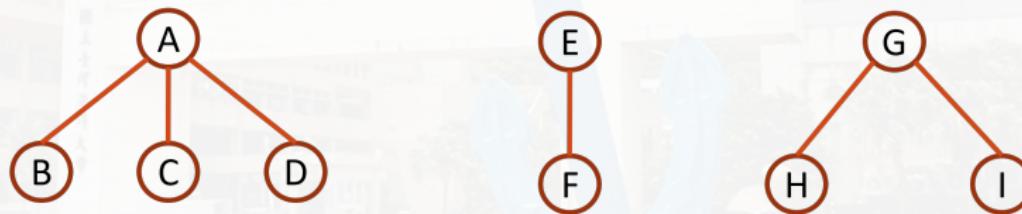
A forest is a set of  $n \geq 0$  disjoint trees.



# Forest

## Forest

A forest is a set of  $n \geq 0$  disjoint trees.



## Question

Is a binary tree a forest?

# Rule of Transforming a Forest into a Binary Tree

- If  $T_1, T_2, \dots, T_n$  is a forest of disjoint trees  $T_1, T_2, \dots, T_n$ , then the binary tree corresponding to this forest, denoted by  $B(T_1, T_2, \dots, T_n)$ ,
  - is empty if  $n = 0$ .
  - has root equal to  $\text{root}(T_1)$ ;
  - has left subtree equal to  $B(T_{11}, T_{12}, \dots, T_{1m})$ , where  $T_{11}, T_{12}, \dots, T_{1m}$  are the subtrees of  $\text{root}(T_1)$ ; and
  - has right subtree  $B(T_2, \dots, T_n)$ .

# Binary Tree Representation of a Forest



# Discussions