



## Data Structures

### Chapter 5 Tree

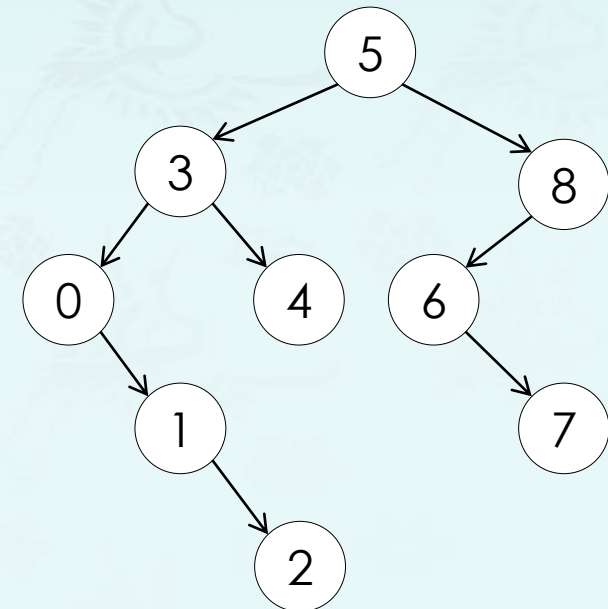
1. Introduction
2. Binary Tree
- 3. Binary Search Tree**
  - Introduction
  - Operations
  - **Demo & Coding**
4. Balancing Tree

## Minimum, Maximum:

- `Minimum()` and `maximum()` returns the node with min or max key.
  - Note that the entire tree does not need to be searched.
  - The minimum key is always located at the left most node, the maximum at the right most node.
  - Complexity of algorithm to find the maximum or minimum will be  $O(\log N)$  in almost balanced binary tree. If tree is skewed, then we have worst case complexity of  $O(N)$ .

```
tree minimum(tree node) { // returns left-most node key
            
}
```

```
tree maximum(tree node) { // returns right-most node key
            
}
```

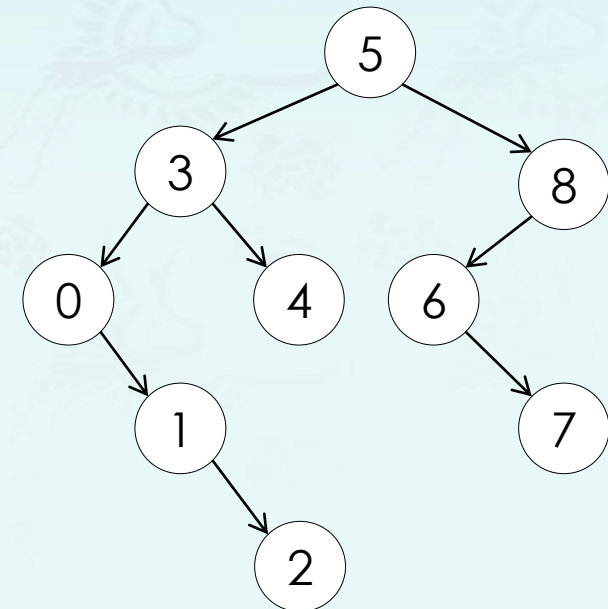


## Minimum, Maximum:

- `Minimum()` and `maximum()` returns the node with min or max key.
  - Note that the entire tree does not need to be searched.
  - The minimum key is always located at the left most node, the maximum at the right most node.
  - Complexity of algorithm to find the maximum or minimum will be  $O(\log N)$  in almost balanced binary tree. If tree is skewed, then we have worst case complexity of  $O(N)$ .

```
tree minimum(tree node) { // returns left-most node key
    if (node->left == nullptr) return node;
    return minimum(node->left);
}
```

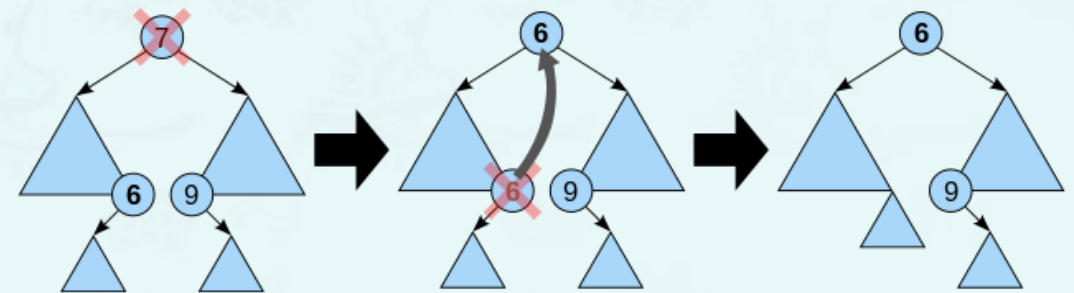
```
tree maximum(tree node) { // returns right-most node key
    if (node->right == nullptr) return node;
    return maximum(node->right);
}
```



## pred(), succ() – predecessor, successor:

- Successor
  - If the given node has a right subtree then by the BST property the next larger key must be in the right subtree. Since all keys in a right subtree are larger than the key of the given node, the successor must be the smallest of all those keys in the right subtree.
- Predecessor
  - If the given node has a left subtree then by the BST property the next smaller key must be in the left subtree. Since all keys in a left subtree are smaller than the key of the given node, the predecessor must be the largest of all those keys in the left subtree.
- Complexity of algorithm
  - $O(\log N)$  in almost balanced binary tree. If tree is skewed, then we have worst case complexity of  $O(N)$ .

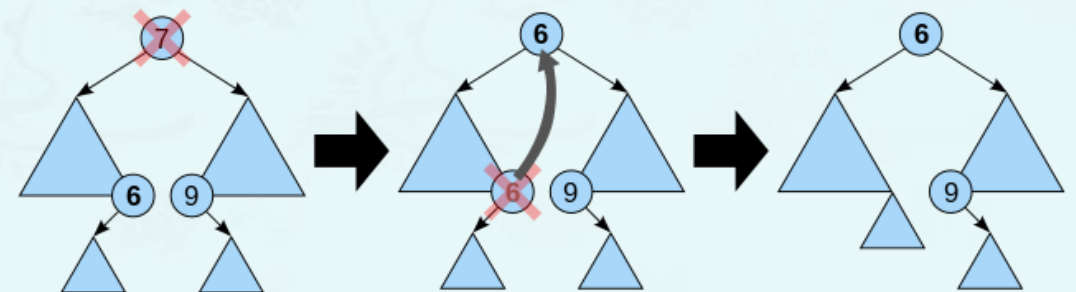
```
tree successor(tree node) {  
    if (node != nullptr && node->right != nullptr)  
          
    return nullptr;  
}
```



## pred(), succ() – predecessor, successor:

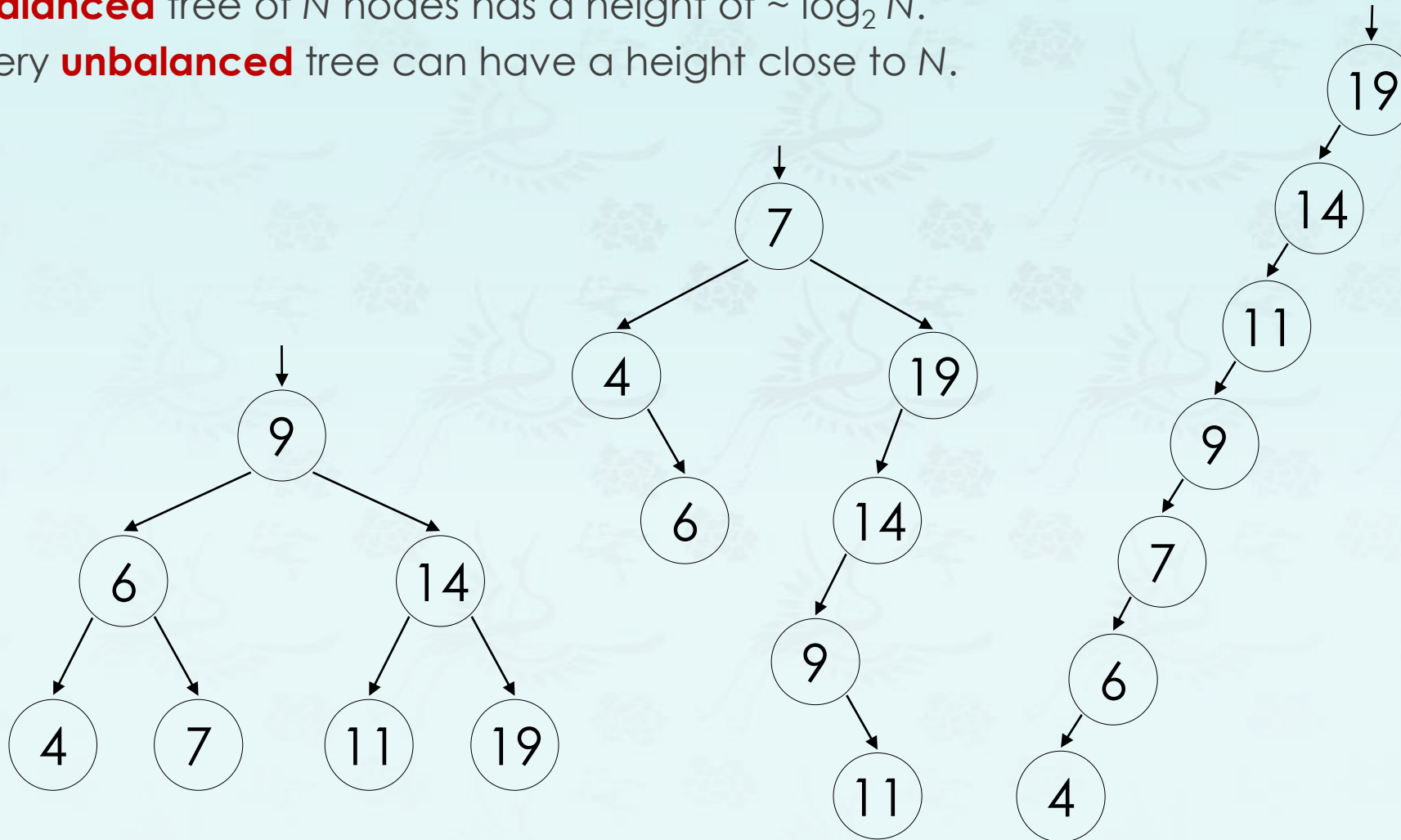
- Successor
  - If the given node has a right subtree then by the BST property the next larger key must be in the right subtree. Since all keys in a right subtree are larger than the key of the given node, the successor must be the smallest of all those keys in the right subtree.
- Predecessor
  - If the given node has a left subtree then by the BST property the next smaller key must be in the left subtree. Since all keys in a left subtree are smaller than the key of the given node, the predecessor must be the largest of all those keys in the left subtree.
- Complexity of algorithm
  - $O(\log N)$  in almost balanced binary tree. If tree is skewed, then we have worst case complexity of  $O(N)$ .

```
tree successor(tree node) {  
    if (node != nullptr && node->right != nullptr)  
        return minimum(node->right);  
    return nullptr;  
}
```



# Binary Search Trees: Observations

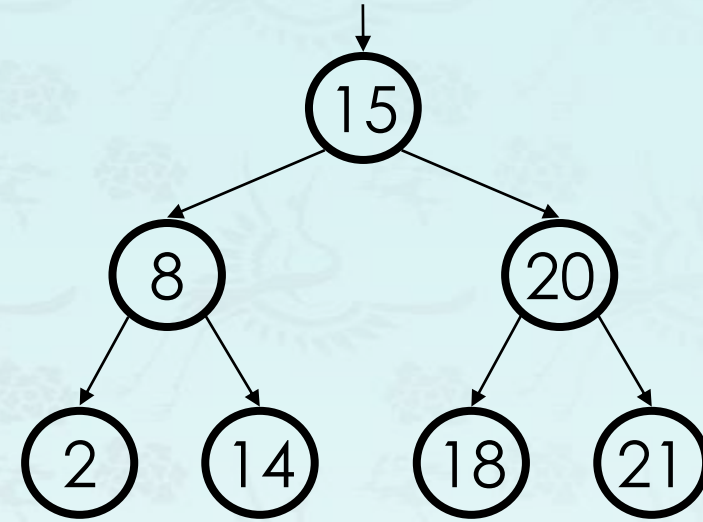
- What do you see in the following BSTs?
  - A **balanced** tree of  $N$  nodes has a height of  $\sim \log_2 N$ .
  - A very **unbalanced** tree can have a height close to  $N$ .





# Binary Search Trees: Observations

- For binary tree of height  $h$ :
  - max # of leaves:  $2^h$
  - max # of nodes:  $2^{h+1} - 1$
  - min # of leaves: 1
  - min # of nodes:  $h + 1$
- The shallower the BST the better.
  - Average case height is  $O(\log N)$
  - Worst case height is  $O(N)$
  - Simple cases such as adding  $(1, 2, 3, \dots, N)$ , or the opposite order, lead to the worst case scenario: height  $O(N)$ .



# Binary Search Trees: Observations

- Q: If you have a sorted sequence, and we want to design a data structure for it. Which one are you going to use an array or BST? and why?

Time Complexity	
BST	$O(h)$
Array	$O(\log n)$

- Q: When searching, we're traversing a path (since we're always moving to one of the children); since the length of the longest path is the height  $h$  of the binary search tree, then finding an element takes  $O(h)$ .
  - Since  $h = \log n$  (where  $n$  is the number of elements), then it's good! – right?
  - No, of course, it is wrong! Why?

A: The nodes could be arranged in linear sequence in BST, so the *height*  $h$  could be  $n$ . In worst case, it is  $O(n)$  instead of  $O(h)$ .



## Operations: growN() & trimN() for testing

---

- It performs a user specified number of insertion(or grow) or deletion(or trim) of nodes in the tree.
- The function **growN()** inserts a user specified number N of nodes in the tree.
  - If it is an empty tree, the value of keys to add ranges from 0 to N-1.
  - If there are some existing nodes in the tree, the value of keys to add ranges from  $\text{max} + 1$  to  $\text{max} + 1 + N$ , where max is the maximum value of keys in the tree.
- This function growN() is provided for your reference^^.

## Operations: growN() & trimN() for testing

---

- It performs a user specified number of insertion(or grow) or deletion(or trim) of nodes in the tree.
- The function **growN()** inserts a user specified number N of nodes in the tree.
  - If it is an empty tree, the value of keys to add ranges from 0 to N-1.
  - If there are some existing nodes in the tree, the value of keys to add ranges from  $\text{max} + 1$  to  $\text{max} + 1 + N$ , where max is the maximum value of keys in the tree.
- This function growN() is provided for your reference^^.
- If the function is called with AVLtree = true, nodes are added using BST grow() function first. Then reconstruct the BST tree into an AVL tree using reconstruct() function which is much faster.

## Operations: growN() & trimN() for testing

---

- The function **trimN()** deletes N number of nodes in the tree.
  - The nodes to trim are **randomly** selected from the tree.
  - If N is less than the tree size (which is not N), you just trim N nodes.
  - If the N is larger than the tree size, set it to the tree size.
  - At any case, you should trim all nodes one by one, but randomly.
  - With an AVL tree, reconstruct it **after** trimming N nodes from BST.

## Operations: growN() & trimN() for testing

---

- The function **trimN()** deletes N number of nodes in the tree.
  - The nodes to trim are **randomly** selected from the tree.
  - If N is less than the tree size (which is not N), you just trim N nodes.
  - If the N is larger than the tree size, set it to the tree size.
  - At any case, you should trim all nodes one by one, but randomly.
  - With an AVL tree, reconstruct it **after** trimming N nodes from BST.

Step 1: Get a list (vector) of all keys from the tree first.

Get the size of the tree using the size().

Use assert to check two sizes;

Step 2: Shuffle the vector with keys. – shuffle()

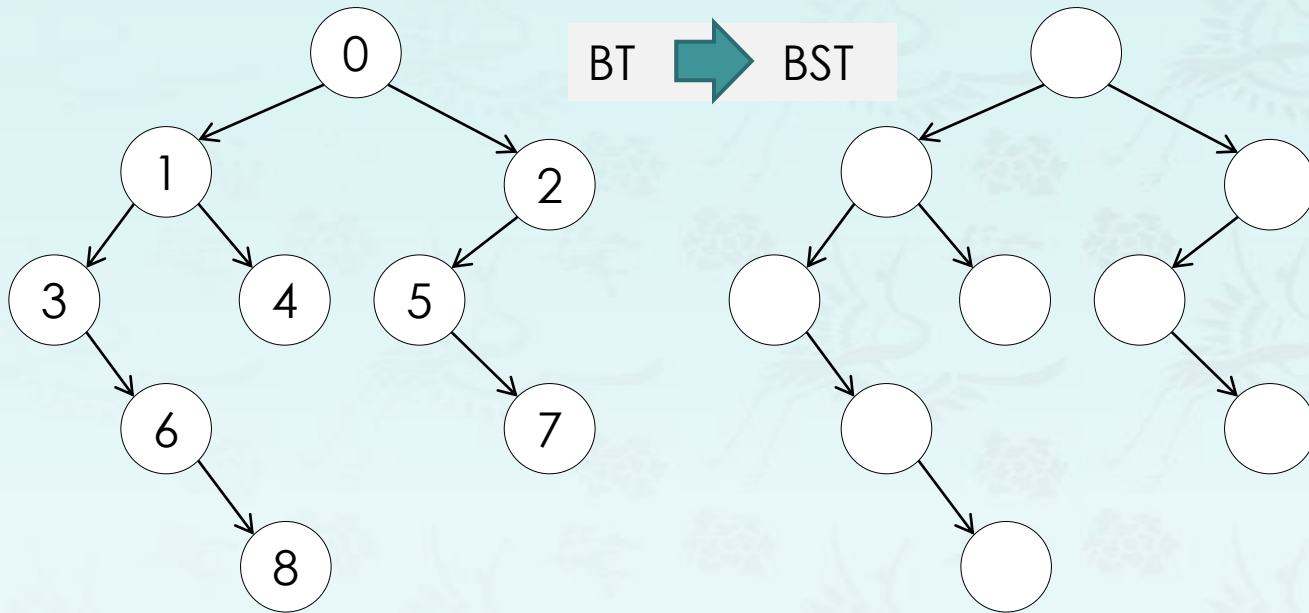
Step 3: Invoke trim() N times with a key from the vector in sequence.

Inside a for loop, trim() may return a new root of the tree.

Step 4: The function is called with AVLtree = true, then reconstruct the tree.

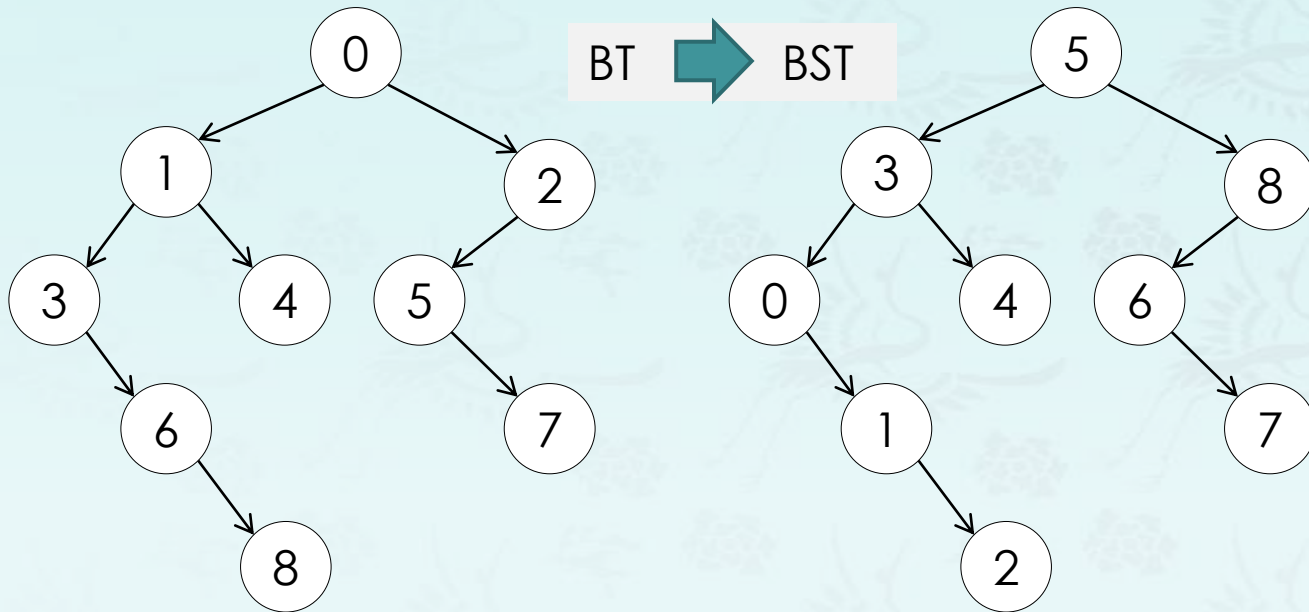
# Convert BT to BST in-place

- Convert a binary tree to a binary search tree while keeping its tree structure as it is.
- For example:



## Convert BT to BST in-place

- Convert a binary tree to a binary search tree while keeping its tree structure as it is.
- For example:





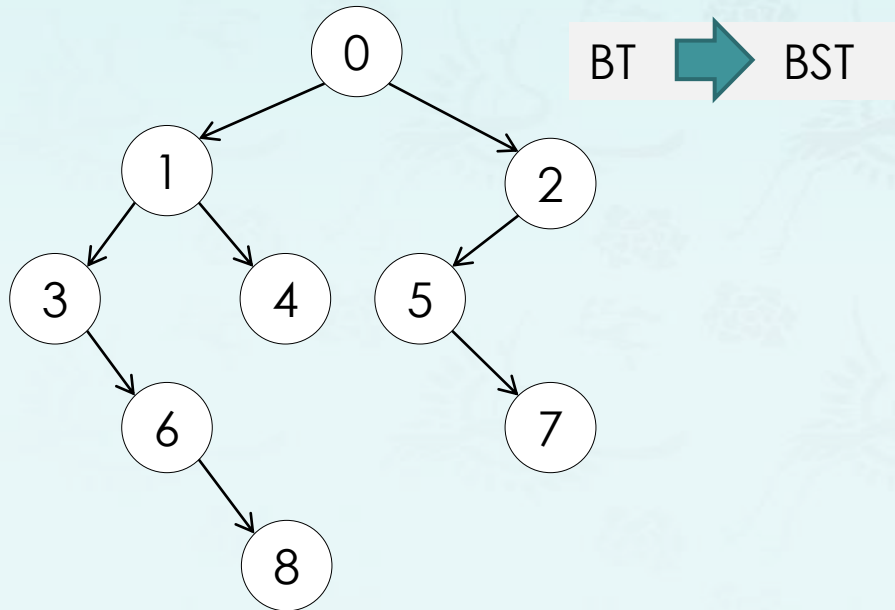
## Convert BT to BST in-place

---

- Convert a binary tree to a binary search tree while keeping its tree structure as it is.
- Algorithm:
  - **Step 1** – store keys of a binary tree into a container like **vector** or **set**. (Do not use an array.)  
*only for pedagogical reason*
  - **Step 2** – sort the keys in vector. Skip this step if set is used since it is already sorted.
  - **Step 3** – Now, do the **inorder** traversal of the tree and copy back the elements of the container into the nodes of the tree one by one.

# Convert BT to BST in-place

- Convert a binary tree to a binary search tree while keeping its tree structure as it is.
- Algorithm:
  - Step 1** – store keys of a binary tree into a container like **vector** or **set**. (**Do not use an array.**)
  - Step 2** – sort the keys in vector. Skip this step if set is used since it is already sorted.
  - Step 3** – Now, do the **inorder** traversal of the tree and copy back the elements of the container into the nodes of the tree one by one.



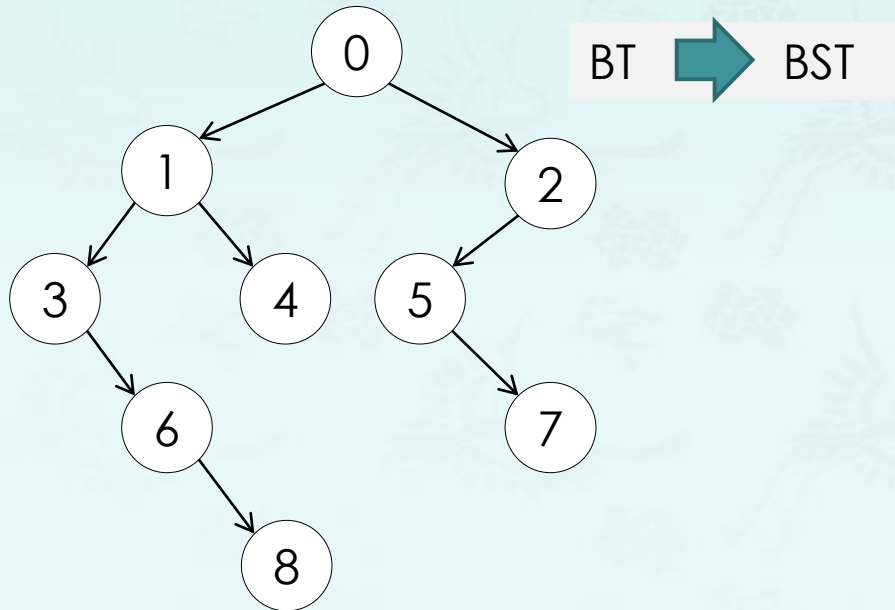
(1) Retrieve the keys from BT:  
3 6 8 1 4 0 5 7 2 // if in-order used

```
void inorder(tree root) {  
    if (root == nullptr) return;  
  
    inorder(root->left);  
    cout << root->key;  
    inorder(root->right);  
}
```

**L**  
**V**  
**R**

# Convert BT to BST in-place

- Convert a binary tree to a binary search tree while keeping its tree structure as it is.
- Algorithm:
  - Step 1** – store keys of a binary tree into a container like **vector** or **set**. (**Do not use an array.**)
  - Step 2** – sort the keys in vector. Skip this step if set is used since it is already sorted.
  - Step 3** – Now, do the **inorder** traversal of the tree and copy back the elements of the container into the nodes of the tree one by one.



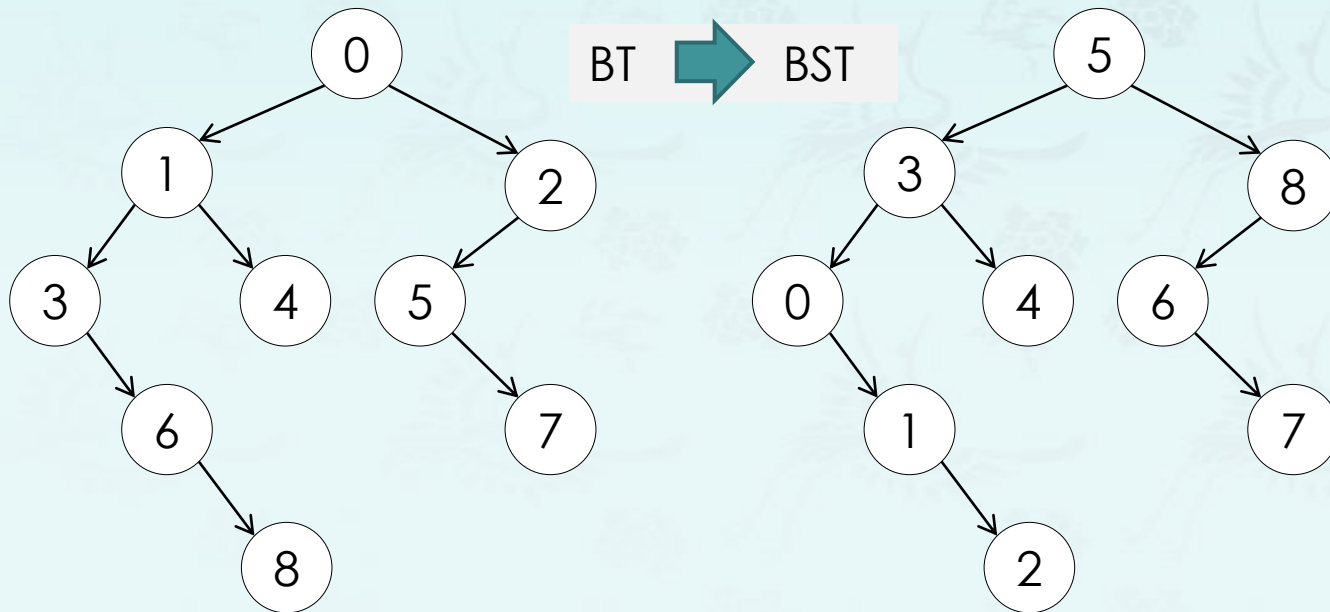
(1) Retrieve the keys from BT:  
3 6 8 1 4 0 5 7 2 // if in-order used  
(2) Sort keys in the container:  
0 1 2 3 4 5 6 7 8

```
void inorder(tree root) {  
    if (root == nullptr) return;  
  
    inorder(root->left);  
    cout << root->key;  
    inorder(root->right);  
}
```

**L**  
**V**  
**R**

# Convert BT to BST in-place

- Convert a binary tree to a binary search tree while keeping its tree structure as it is.
- Algorithm:
  - Step 1** – store keys of a binary tree into a container like **vector** or **set**. (**Do not use an array.**)
  - Step 2** – sort the keys in vector. Skip this step if set is used since it is already sorted.
  - Step 3** – Now, do the **inorder** traversal of the tree and copy back the elements of the container into the nodes of the tree one by one.



- (1) Retrieve the keys from BT:  
3 6 8 1 4 0 5 7 2 // if in-order used
- (2) Sort keys in the container:  
0 1 2 3 4 5 6 7 8
- (3) Replace keys in BT with sorted keys while in-order traversal.

```
void inorder(tree root) {
    if (root == nullptr) return;

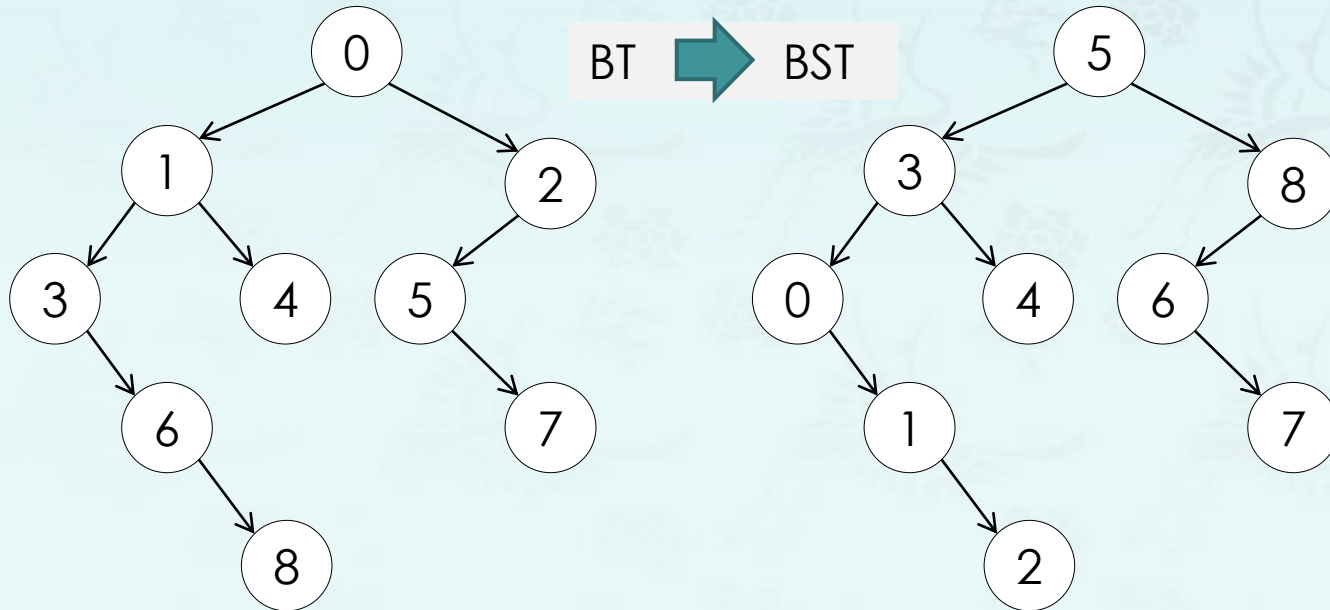
    inorder(root->left);
    cout << root->key;
    inorder(root->right);
}
```

**L**  
**V**  
**R**

# Convert BT to BST in-place

- Convert a binary tree to a binary search tree while keeping its tree structure as it is.

(1) Retrieve the keys from BT:  
3 6 8 1 4 0 5 7 2 // if in-order used  
(2) Sort keys in the container:  
0 1 2 3 4 5 6 7 8  
(3) Replace keys in BT with sorted keys  
while in-order traversal.



```
void get_keys(tree root, set<int> &keys) {
    if (root == nullptr) return;
    keys.insert(root->key);
    get_keys(root->left, keys);
    get_keys(root->right, keys);
}
```

```
void put_keys(tree root,
              set<int>::iterator& it) {
}
```

```
void BTtoBST(tree root) {
    set<int> keys;
    get_keys(root, keys);
    assert(
    set<int>::iterator it = keys.begin();
    put_keys(root, it);
}
```



## Data Structures

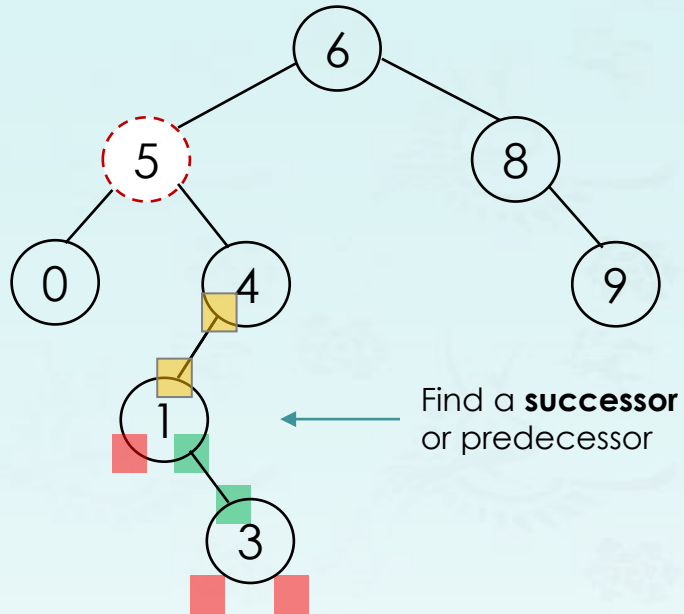
### Chapter 5 Tree

1. Introduction
2. Binary Tree
- 3. Binary Search Tree**
  - Introduction
  - Operations
  - Demo & Coding
4. Balancing Tree



# Operations: delete (or trim)

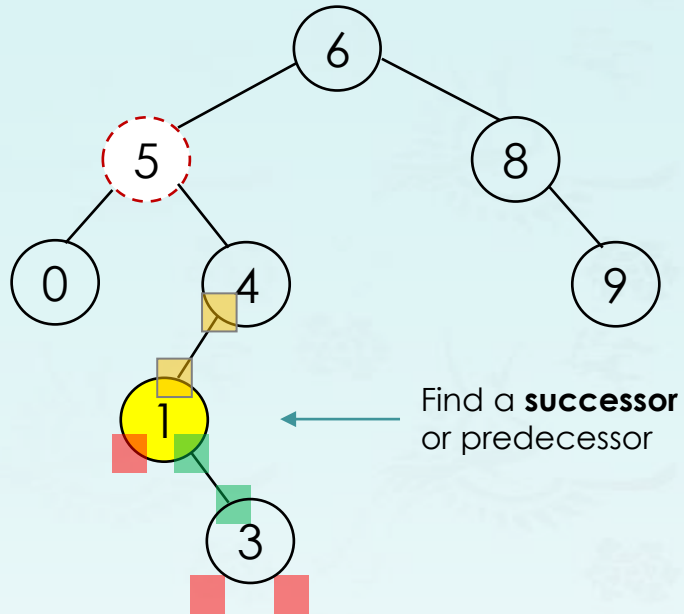
- **Example:** Case 3: Two children



1. find the node 5 to delete

# Operations: delete (or trim)

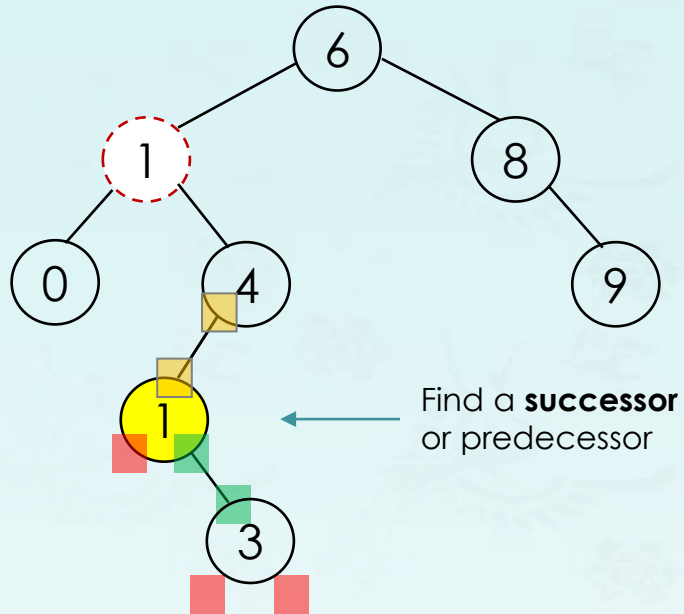
- **Example:** Case 3: Two children



1. find the node 5 to delete
2. if (two children case),  
find 5's successor's key = 1

# Operations: delete (or trim)

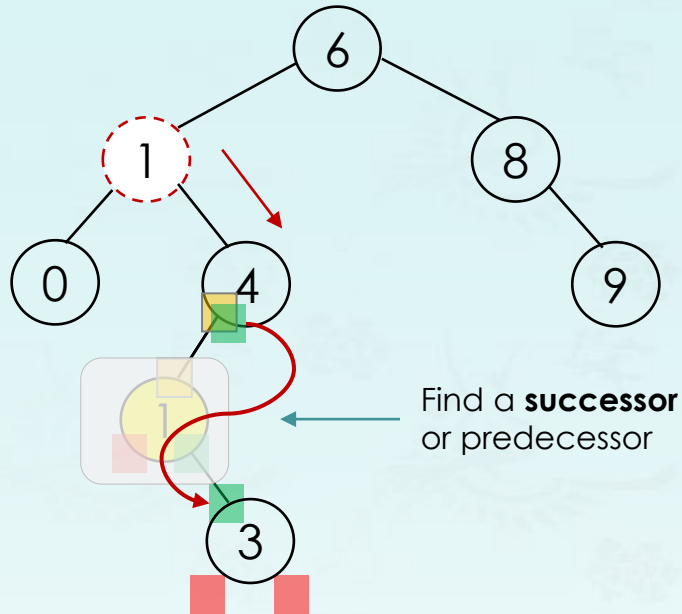
- **Example:** Case 3: Two children



1. find the node 5 to delete
2. if (two children case),  
find 5's successor's key = 1
3. replace 5 with 1

# Operations: delete (or trim)

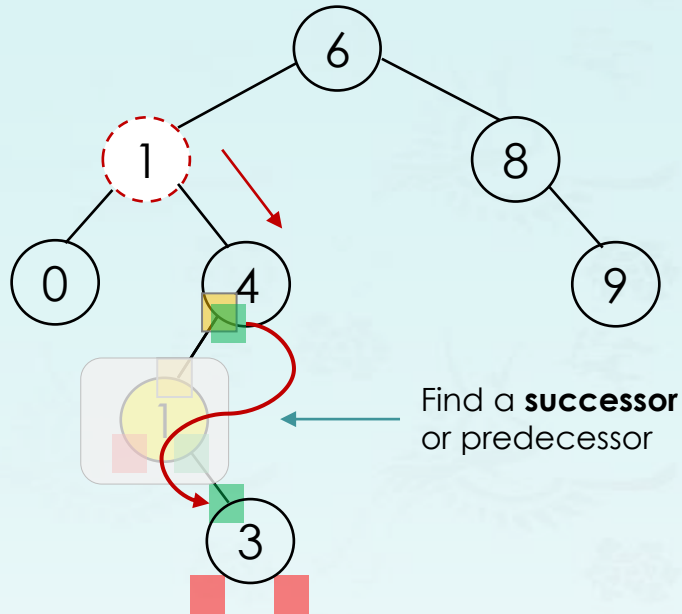
- **Example:** Case 3: Two children



1. find the node 5 to delete
2. if (two children case),  
find 5's successor's key = 1
3. replace 5 with 1
4. invoke  
`node->right = trim(node->right, 1)`

# Operations: delete (or trim)

## ■ **Example:** Case 3: Two children



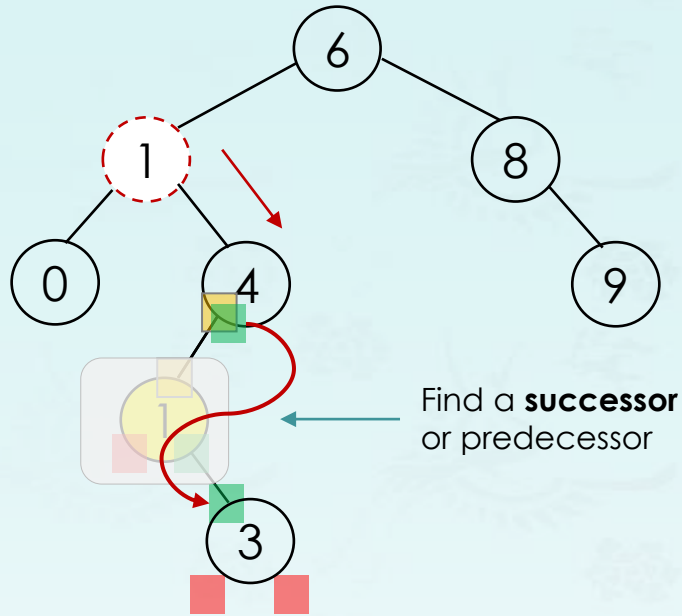
1. find the node 5 to delete
2. if (two children case),  
find 5's successor's key = 1
3. replace 5 with 1
4. invoke  
`node->right = trim(node->right, 1)`

### **Some thoughts:**

- Step 2 Get the heights of two subtree first.
  - If right subtree height is larger, then use the successor.  
Otherwise use the predecessor to shorten the tree height.
- Step 4 simply uses the code for one-child case deletion.

# Operations: delete (or trim)

## ■ **Example:** Case 3: Two children



1. find the node 5 to delete
2. if (two children case),  
find 5's successor's key = 1
3. replace 5 with 1
4. invoke  
`node->right = trim(node->right, 1)`

### **Some thoughts:**

- Step 2 Get the heights of two subtree first.
  - If right subtree height is larger, then use the successor.  
Otherwise use the predecessor to shorten the tree height.
- Step 4 simply uses the code for one-child case deletion.

### **Some questions:**

- What if successor has **two** children?
  - **Not possible !**
  - Because if it has two nodes, at least one of them is less than it, then in the process of finding successor, we won't pick it !