Data Structures
Chapter 5 Tree

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

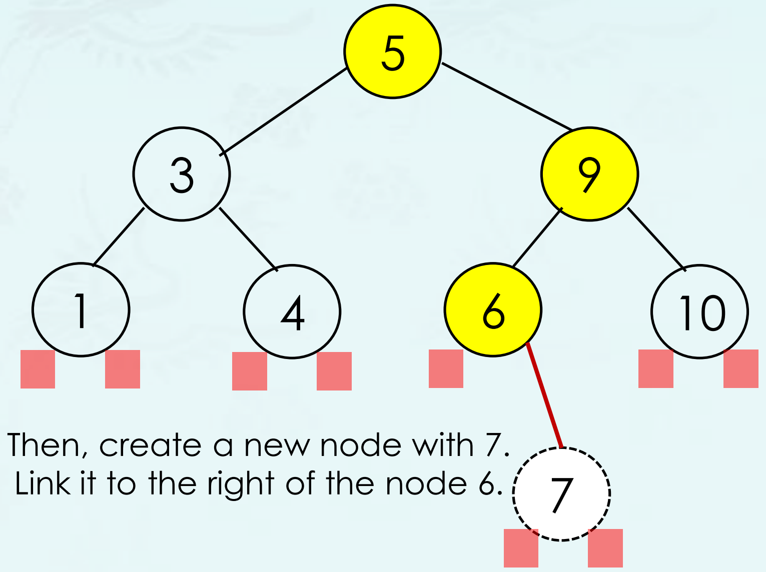
Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204, Handong Global University

Operations: Insert (or grow)

- grow(node, k) - Insert a node with k
  - **Step 1**: If the tree is empty, return a new node(k).
  - **Step 2**: Pretending to search for k in BST, until locating a nullptr.
  - **Step 3**: create a new node(k) and link it.

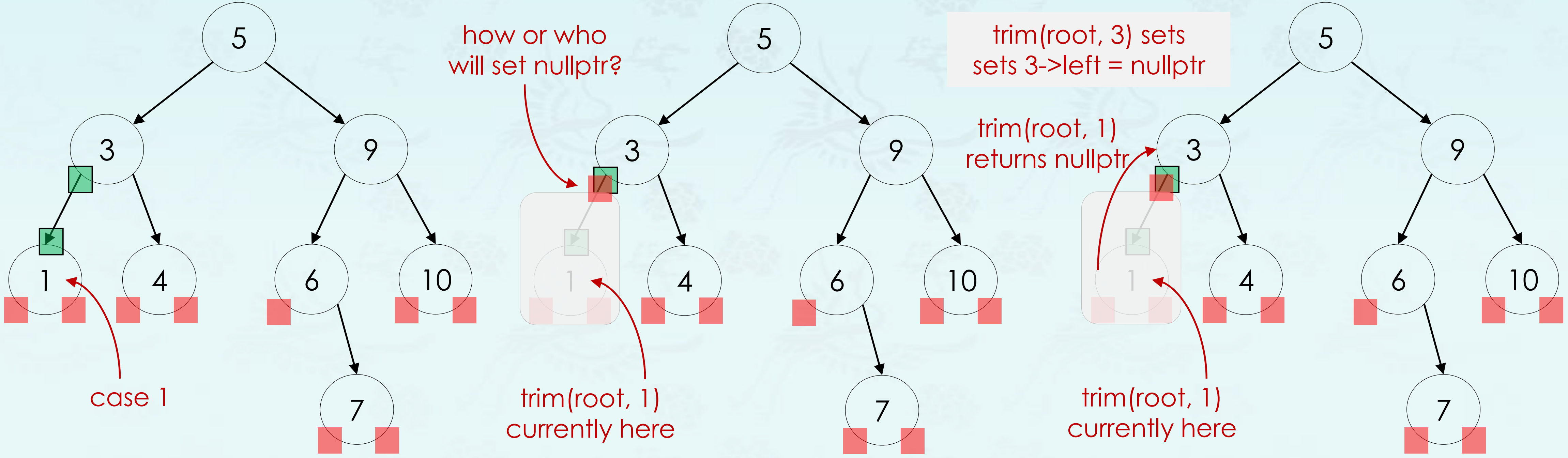
```
tree grow(tree node, int key) {
  if (node == nullptr)
    return new tree(key);

  if (key < node->key)
    node->left = grow(node->left, key);
  else if (key > node->key)
    node->right = grow(node->right, key);
  return node;
}
```

- Q1: Do you see the difference between the binary tree and binary search tree in this operation?
- Q2: To complete inserting **7**, how many times was **`grow()`** called?
- Q3: How many times "**`if (key < node->key) …`** " called during this process?
- Q4: At the end of this whole process, which **`return`** will be executed and what is the key value of the node?



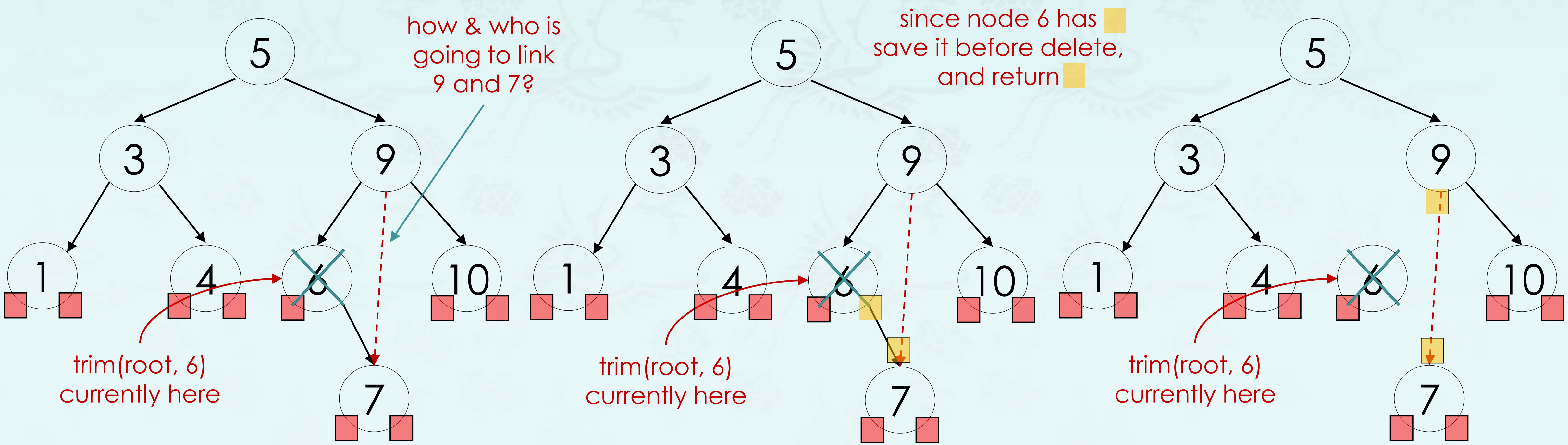
Operations: delete (or trim)

- When we delete a node, **three possibilities** arise depending on how many children the node to be deleted has:
  - **Case 1:** No child – Simply delete a leaf itself from the tree and return a null.
  - **Case 2:** Only one child – before deleting itself and save the link, then pass over the link.



Operations: delete (or trim)

- When we delete a node, **three possibilities** arise depending on how many children the node to be deleted has:
  - **Case 1:** No child – Simply delete a leaf itself from the tree and return a null.
  - **Case 2:** Only one child – before deleting itself and save the link, then pass over the link.



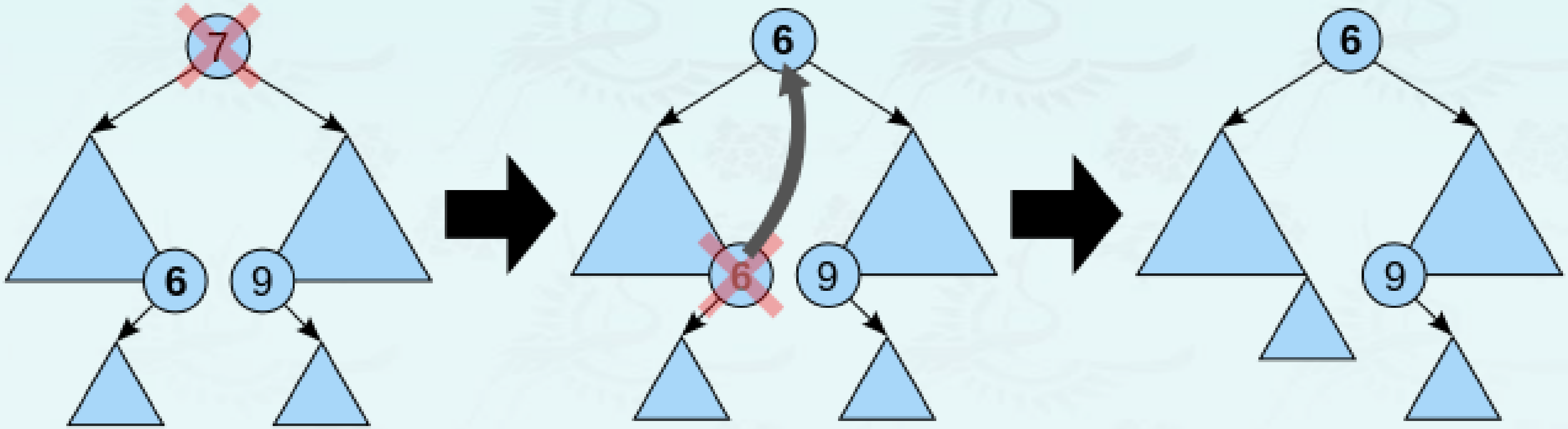
Operations: delete (or trim)

- When we delete a node, **three possibilities** arise depending on how many children the node to be deleted has:
  - **Case 1:** No child – Simply delete a leaf itself from the tree and return a null.
  - **Case 2:** Only one child – before deleting itself and save the link, then pass over the link.
  - **Case 3: Two children**
    - Call the node to be deleted N. Do not delete N.
    - Instead, choose either its in-order **successor** node or its in-order **predecessor** node, R.
    - Then, recursively call delete on R until reaching one of the first two cases.
    - If you choose in-order **successor** of a node, as right subtree is not NULL, then its in-order **successor** is node when least value in its right subtree, which will have at a maximum of 1 subtree, so deleting it would fall in one of first two cases.

Operations: delete (or trim)

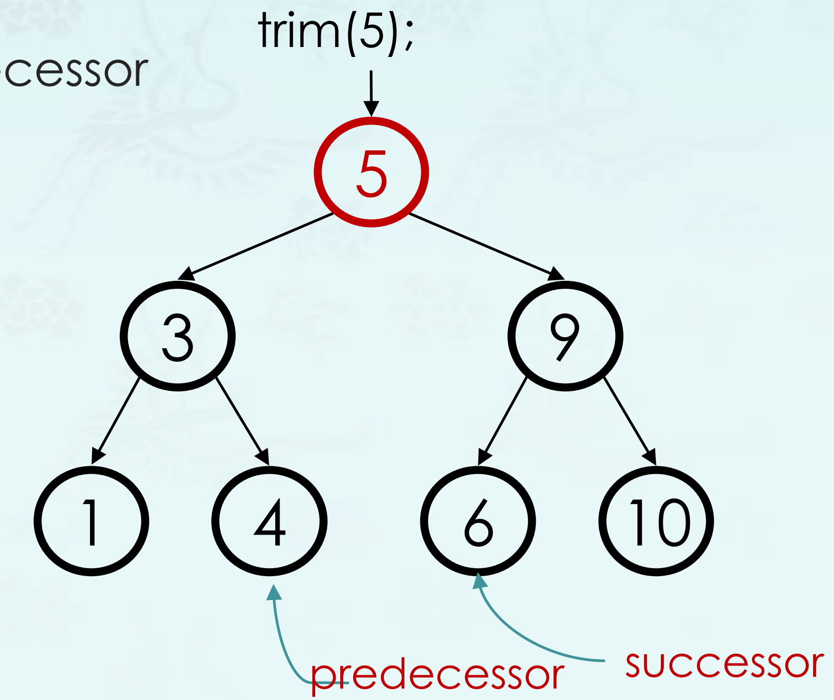
- Case 3: **Two children**
  1. The rightmost node in the left subtree, the inorder **predecessor 6**, is identified.
  2. Its value is copied into the node being trimmed.
  3. The inorder **predecessor** can then be trimmed because it has at most one child.

- NOTE: The same method works symmetrically using the inorder **successor** labelled **9**.



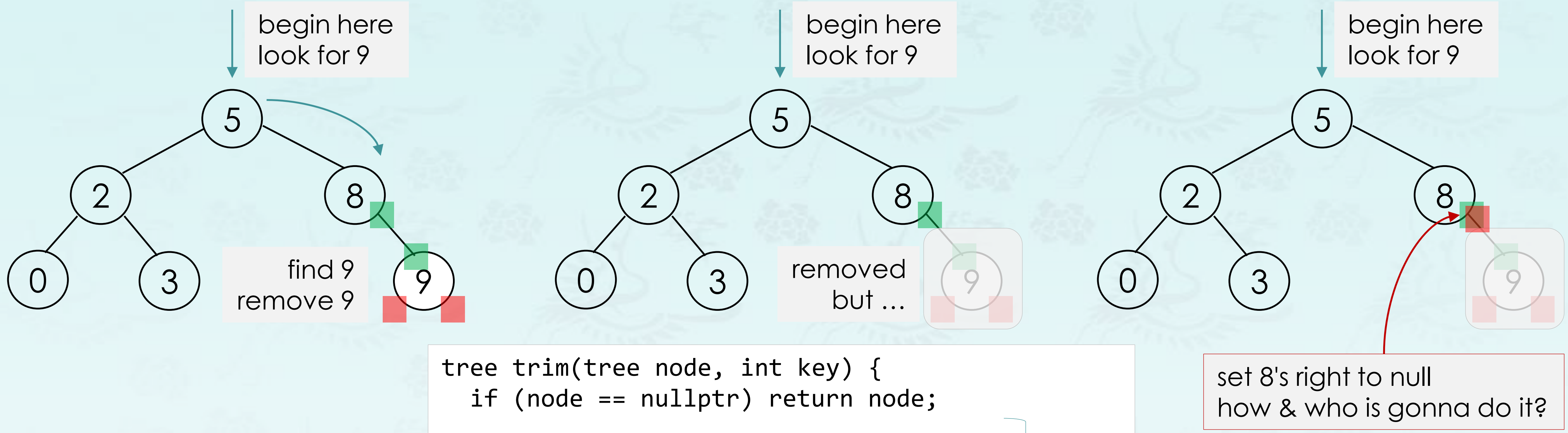
Operations: delete (or trim)

- Case 3: **Two children**
  - Idea: Replace the trimmed node with a value guaranteed to be between two child subtrees
- Options:
  - predecessor from left subtree: maximum(node->left)
  - successor from right subtree: minimum(node->right)
  - These are the easy cases of predecessor/successor
  - Now trim the original node containing successor or predecessor
  - It becomes leaf or one child case – easy cases of trim!



Operations: delete (or trim)

- **Example:** Case 1: No child – a leaf node deletion




```
...
int key = 9;
root = trim(root, key);

...
```

```
tree trim(tree node, int key) {
  if (node == nullptr) return node;
  ...
  else if (key > node->key)
    node->right = trim(node->right, key);
  ...
  else   // found
    ...  // two children case
    ...  // one left/right child case
    ...  // no child case
  return node;
}
```

search

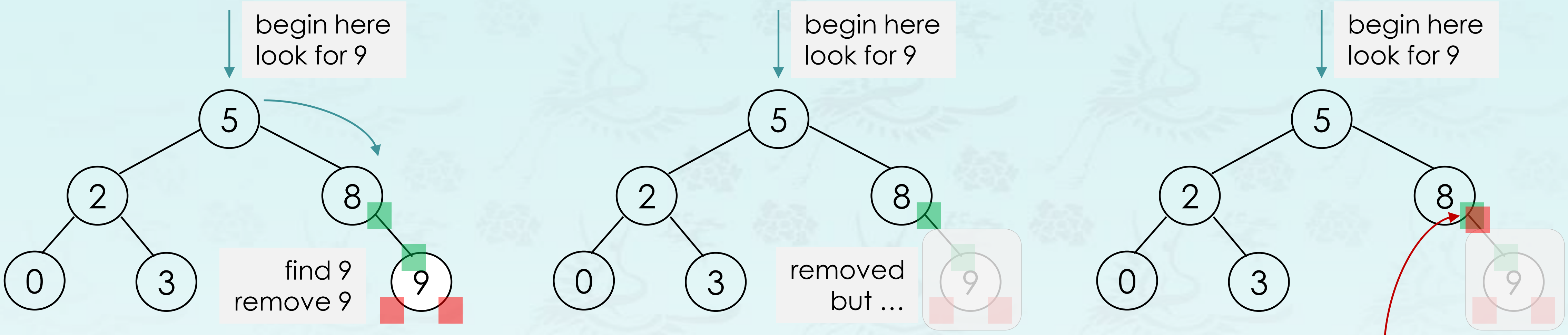
trim

```
... // no child case
  delete node;

...
```

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204, Handong Global University

Operations: delete (or trim)

- **Example:** Case 1: No child – a leaf node deletion



set 8's right to null
how & who is gonna do it?

```
...
int key = 9;
root = trim(root, key);

...
```

```
tree trim(tree node, int key) {
  if (node == nullptr) return node;
  ...
  else if (key > node->key)
    node->right = trim(node->right, key);
  ...
  else   // found
    ...  // two children case
    ...  // one left/right child case
    ...  // no child case
  return node;
}
```

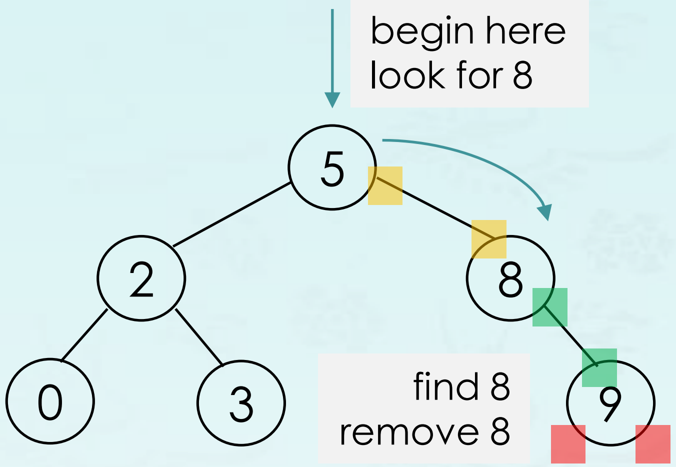
search

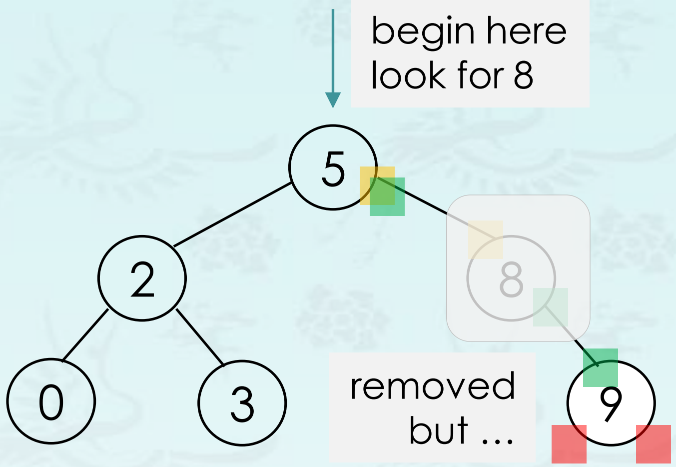
trim

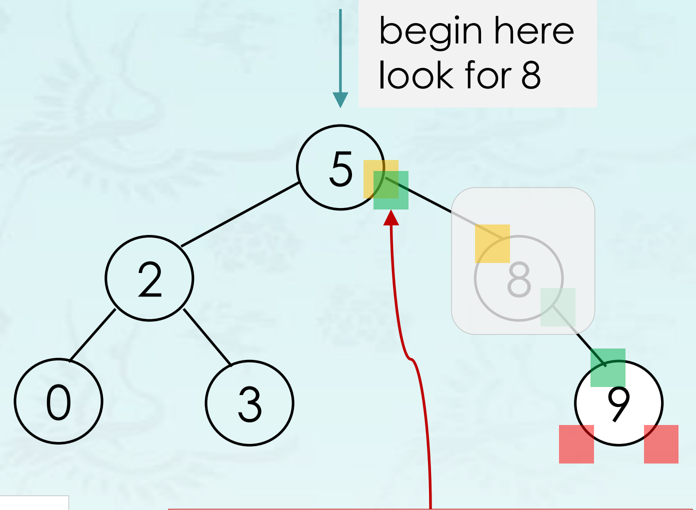
```
... // no child case
  delete node;
  return nullptr;
...
```

Operations: delete (or trim)

- **Example:** Case 2: One child – a node deletion







set 5's right to node 9
how & who is gonna do it?

```
...
int key = 8;
root = trim(root, key);

...
```

```
tree trim(tree node, int key) {
  if (node == nullptr) return node;
  ...
  else if (key > node->key)
   node->right = trim(node->right, key);
  ...
  else   // found
    ...  // two children case
    ...  // one left/right child case
    ...  // no child case
  return node;
}
```

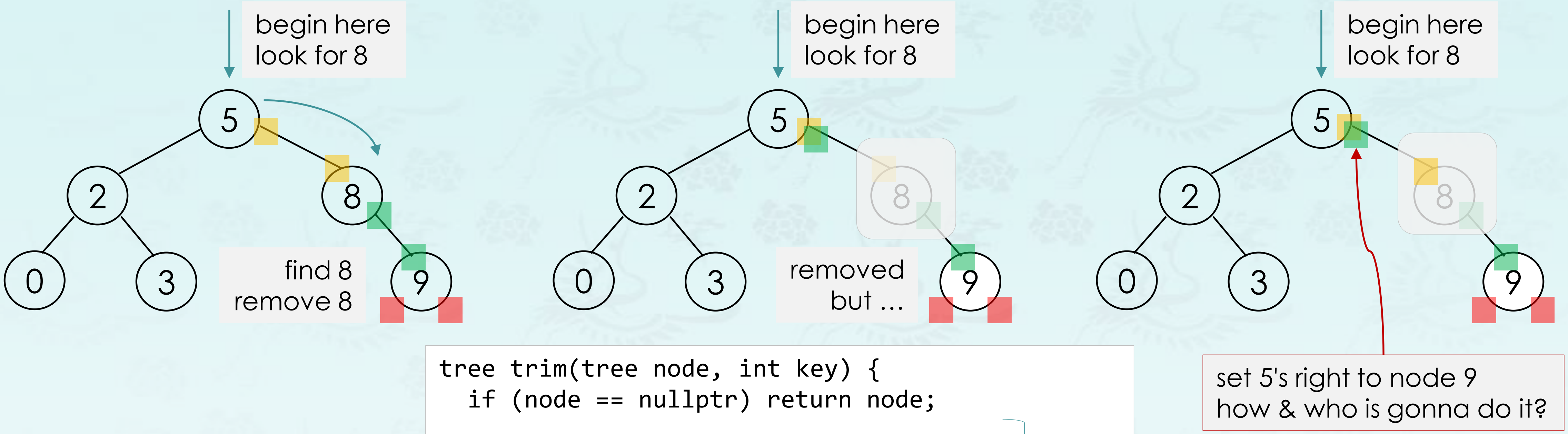
search

trim

```
... // one right child case


  delete node;

...
```

Operations: delete (or trim)

- **Example:** Case 2: One child – a node deletion




```
...
int key = 8;
root = trim(root, key);

...
```

```
tree trim(tree node, int key) {
  if (node == nullptr) return node;
  ...
  else if (key > node->key)
    node->right = trim(node->right, key);
  ...
  else   // found
    ...  // two children case
    ...  // one left/right child case
    ...  // no child case
  return node;
}
```

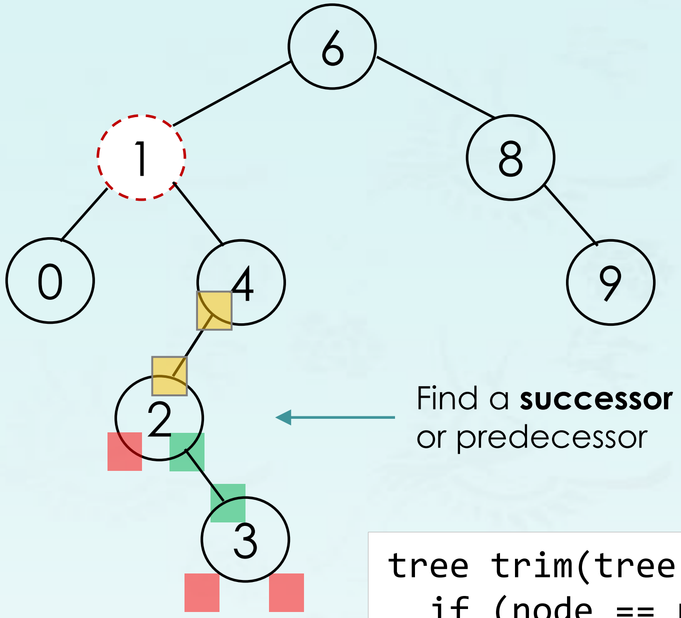
search

trim

```
... // one right child case
  tree temp = node;
  node = node->right;
  delete temp;
  return node;
...
```

Operations: delete (or trim)

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



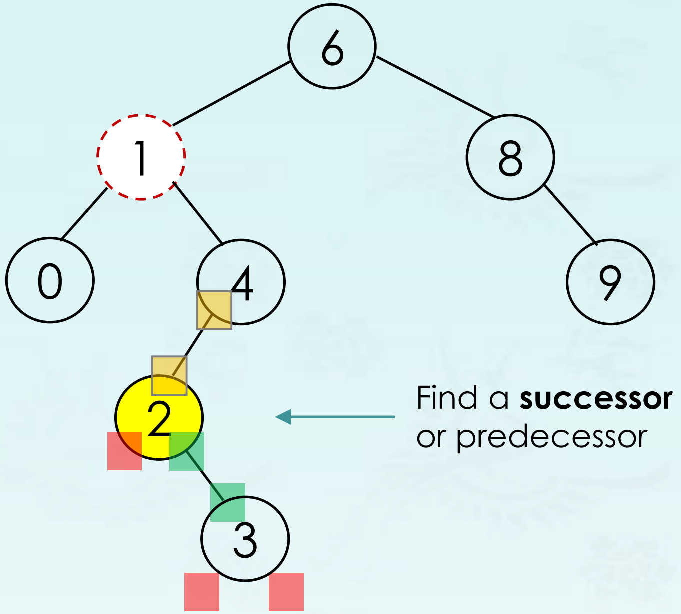

```
1. find the node 1 to delete
2. if (found)
     if (two children case),
```

```
tree trim(tree node, int key) {
  if (node == nullptr) return node;
  ...
  else if (key > node->key)
    node->right = trim(node->right, key);
  ...
  else   // found
    ...  // two children case
    ...  // one left/right child case
    ...  // no child case
  return node;
}
```

```
// two children case
if (height(node->left) <= height(node->right)) {
  // get the successor
  // copy the successor's key to this node
  // trim the successor starting at node->right
}
else {
  // get the predeccessor
  // copy the predeccessor's key to this node
  // trim the predeccessor strating at node->left
}
```

Operations: delete (or trim)

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

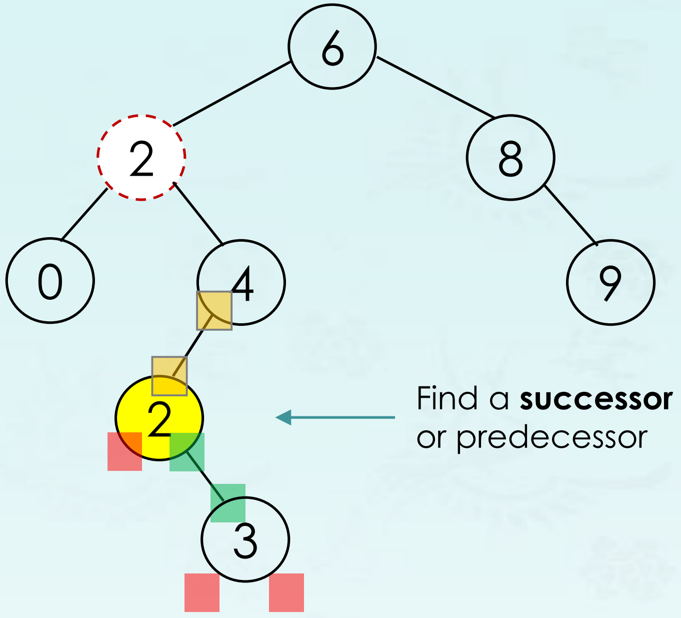



```
1. find the node 1 to delete
2. if (found)
     if (two children case),
       find 1's successor's key = 2
```

```
// two children case
if (height(node->left) <= height(node->right)) {
  // get the successor
  // copy the successor's key to this node
  // trim the successor starting at node->right
}
else {
  // get the predeccessor
  // copy the predeccessor's key to this node
  // trim the predeccessor strating at node->left
}
```

Operations: delete (or trim)

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

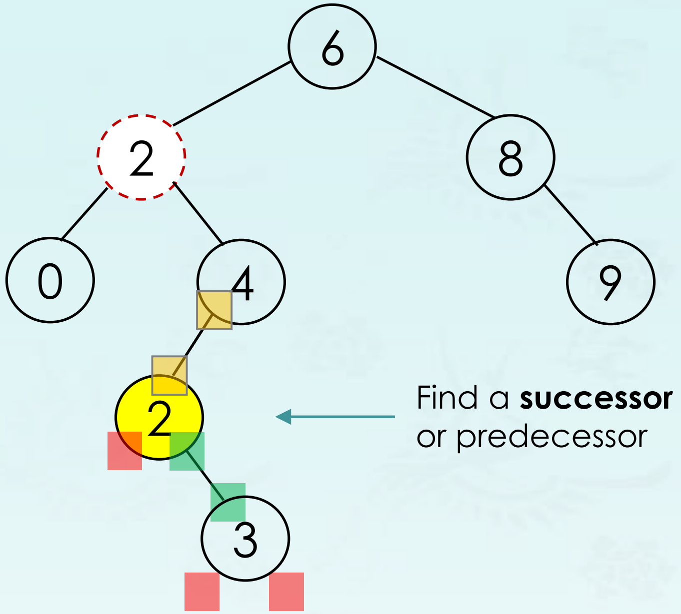



```
1. find the node 1 to delete
2. if (found)
     if (two children case),
       find 1's successor's key = 2
       replace 1 with 2
```

```
// two children case
if (height(node->left) <= height(node->right)) {
  // get the successor
  // copy the successor's key to this node
  // trim the successor starting at node->right
}
else {
  // get the predeccessor
  // copy the predeccessor's key to this node
  // trim the predeccessor strating at node->left
}
```

Operations: delete (or trim)

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

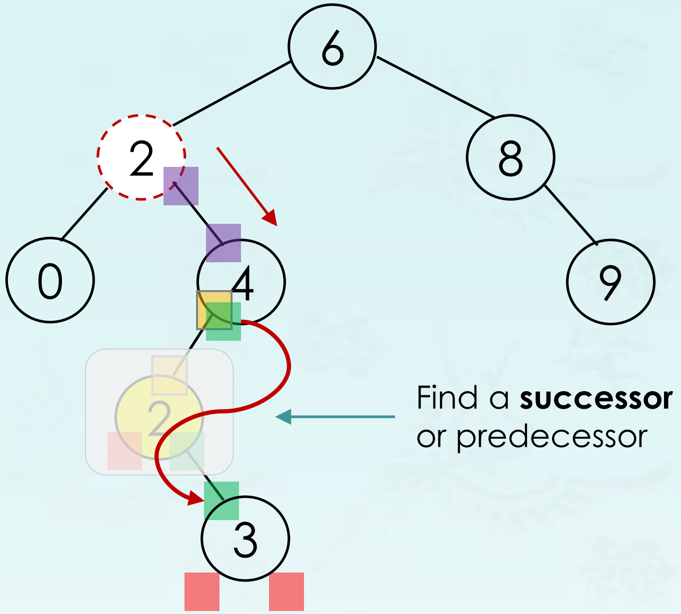



```
1. find the node 1 to delete
2. if (found)
     if (two children case),
       find 1's successor's key = 2
       replace 1 with 2
       trim 2 starting at node->right or 4
```

```
// two children case
if (height(node->left) <= height(node->right)) {
  // get the successor
  // copy the successor's key to this node
  // trim the successor starting at node->right
}
else {
  // get the predeccessor
  // copy the predeccessor's key to this node
  // trim the predeccessor strating at node->left
}
```

Operations: delete (or trim)

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



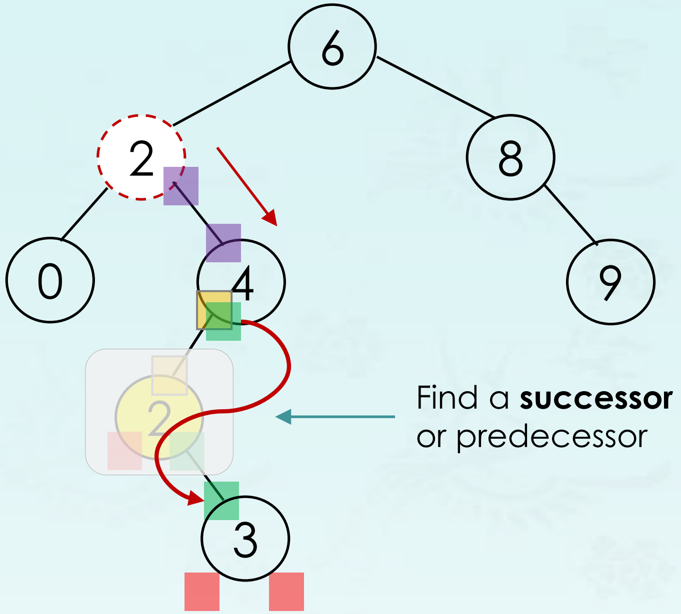

```
1. find the node 1 to delete
2. if (found)
     if (two children case),
       find 1's successor's key = 2
       replace 1 with 2
       trim 2 starting at node->right or 4
```

```
       node->right = trim(node->right, 2)
```

```
// two children case
if (height(node->left) <= height(node->right)) {
  // get the successor
  // copy the successor's key to this node
  // trim the successor starting at node->right
}
else {
  // get the predeccessor
  // copy the predeccessor's key to this node
  // trim the predeccessor strating at node->left
}
```

Operations: delete (or trim)

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



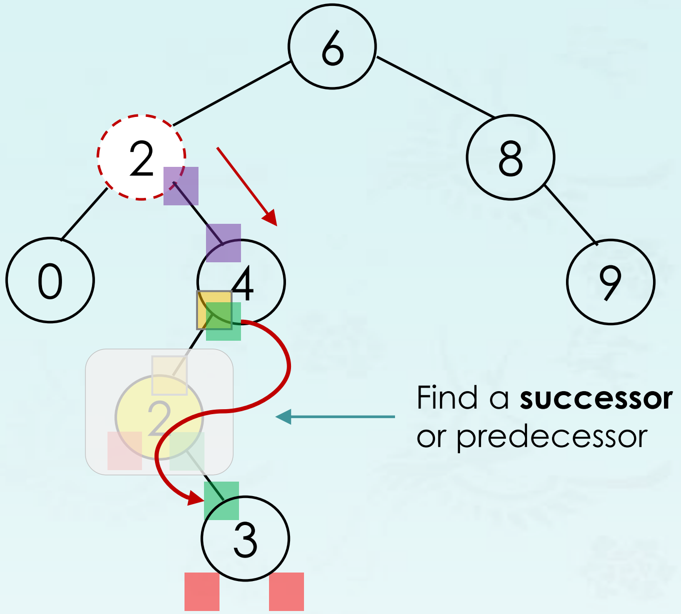

```
1. find the node 1 to delete
2. if (found)
     if (two children case),
       find 1's successor's key = 2
       replace 1 with 2
       trim 2 starting at node->right or 4
```

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 1 to delete
2. if (found)
     if (two children case),
       find 1's successor's key = 2
       replace 1 with 2
       trim 2 starting at node->right or 4
```

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 !

Binary search trees

- **More Operations:**
  - Query – search, minimum, maximum, successor, predecessor
  - Minimum, maximum
    - For min, we simply follow the left pointer until we find a nullptr node. Time complexity: O(h)
  - Search operation takes time O(h), where h is the height of a BST.

Data Structures
Chapter 5 Tree

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

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204, Handong Global University