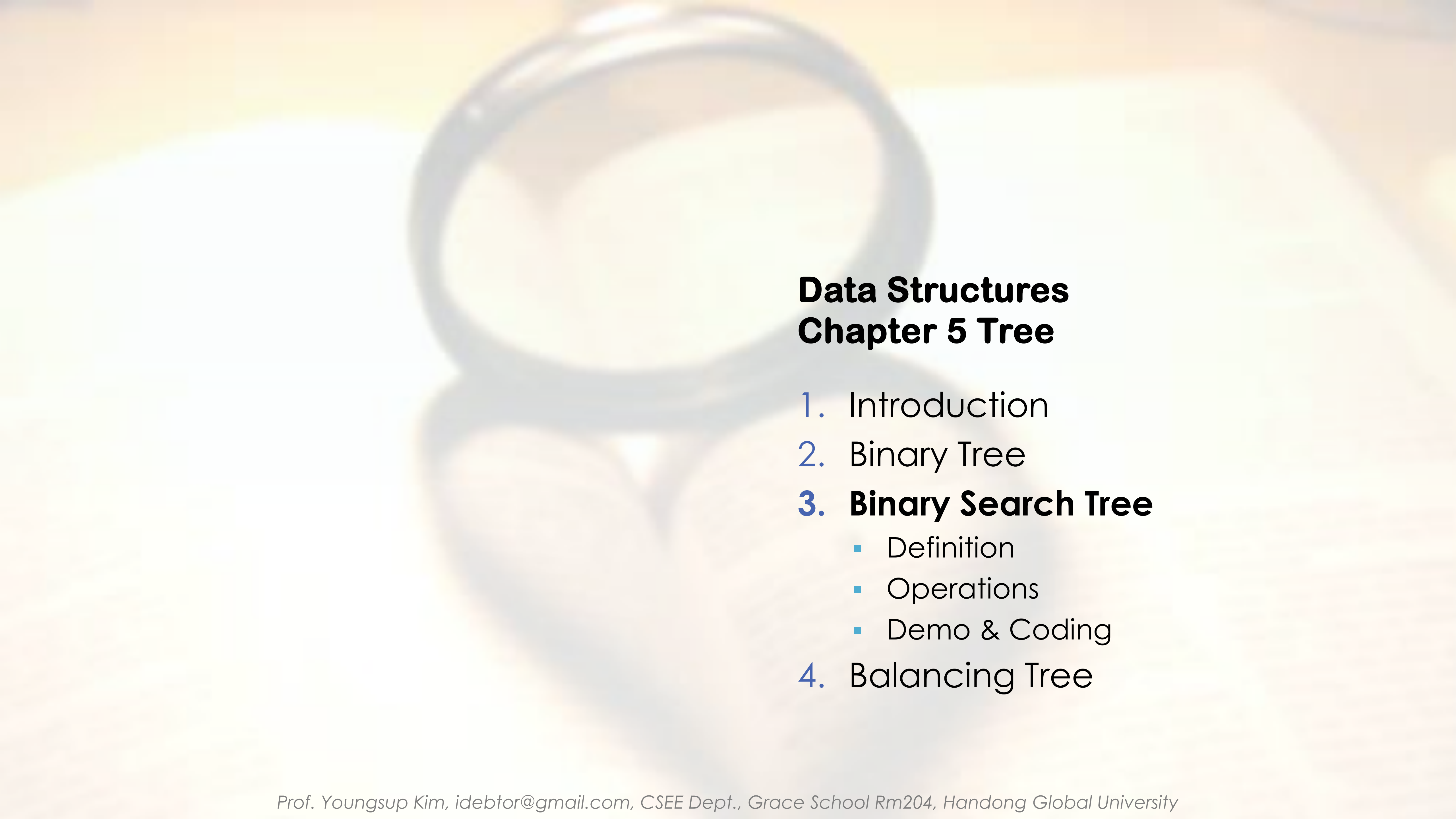
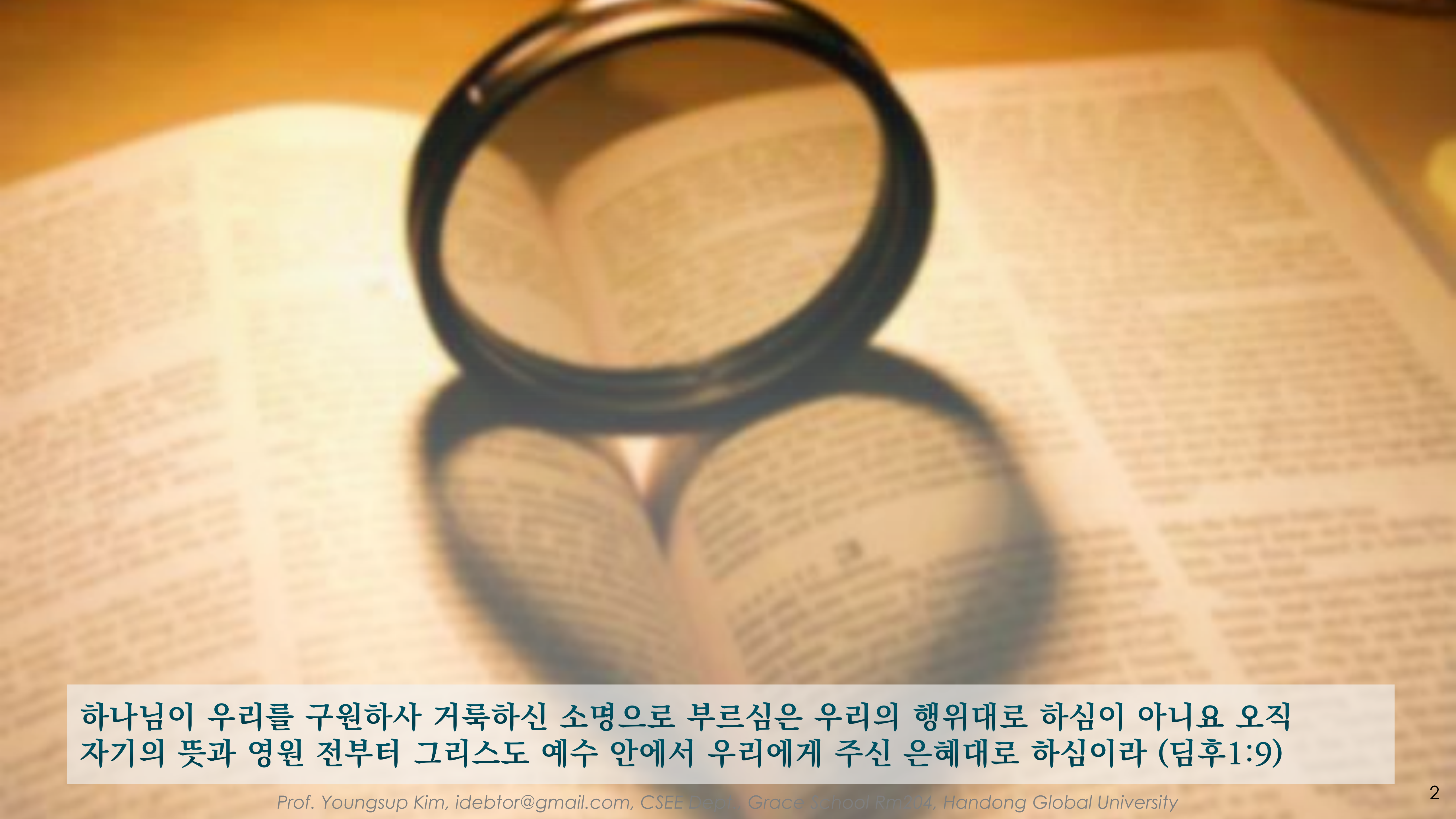
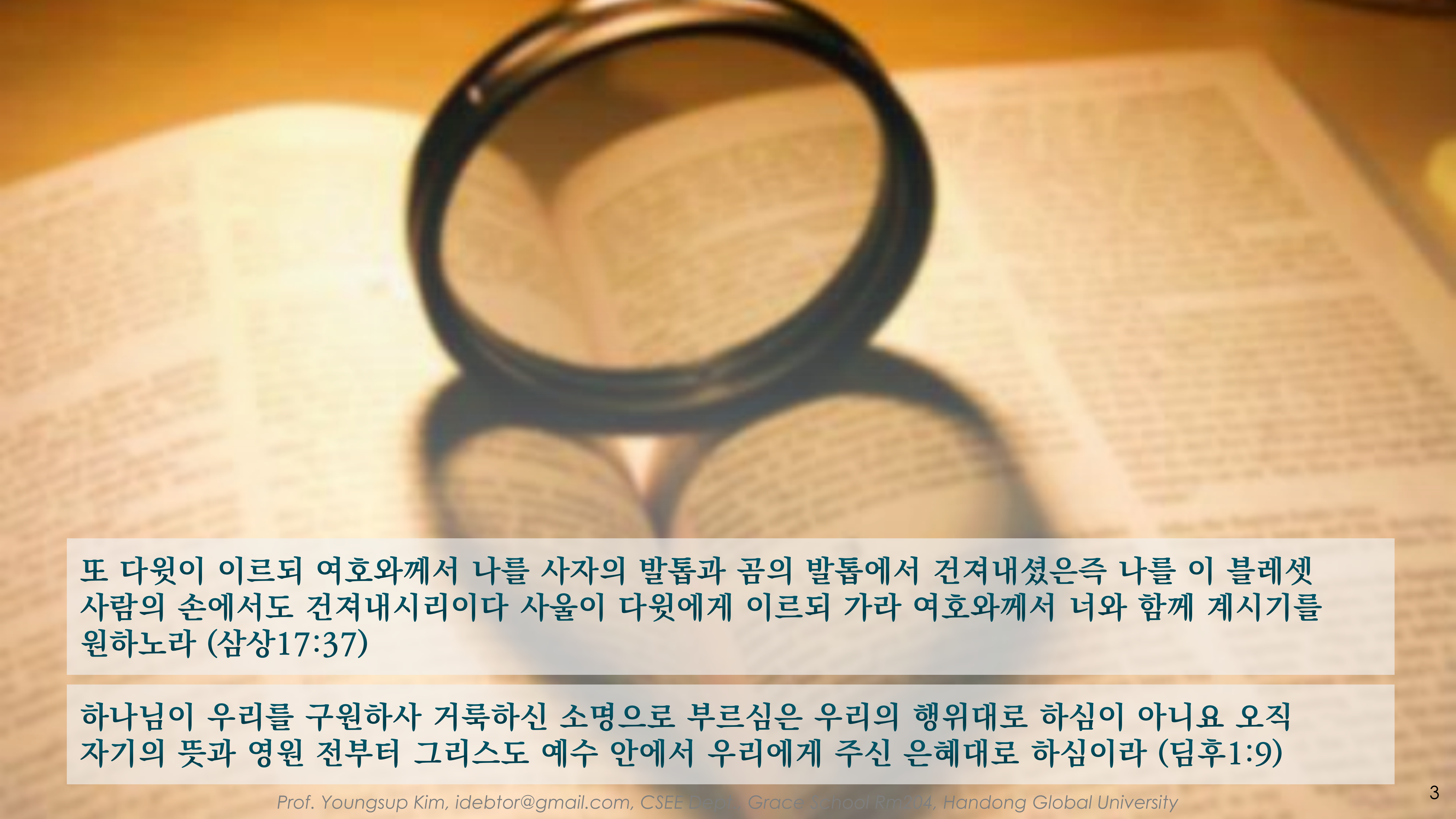


Data Structures
Chapter 5 Tree

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



하나님이 우리를 구원하사 거룩하신 소명으로 부르심은 우리의 행위대로 하심이 아니요 오직 자기의 뜻과 영원 전부터 그리스도 예수 안에서 우리에게 주신 은혜대로 하심이라 (딤후1:9)

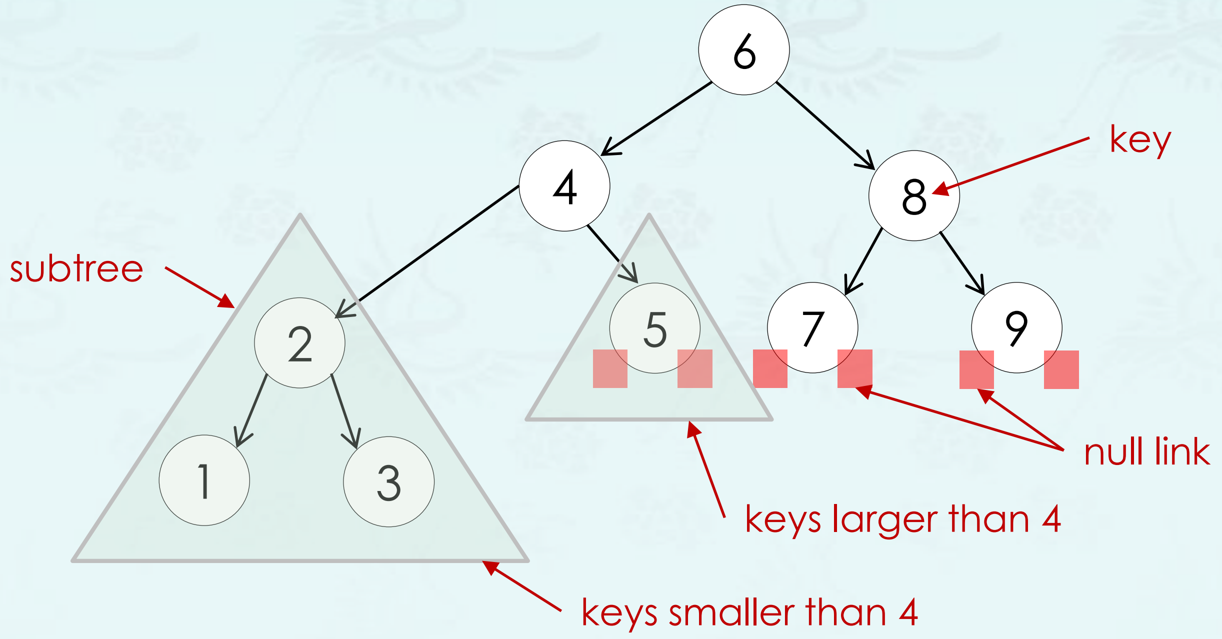


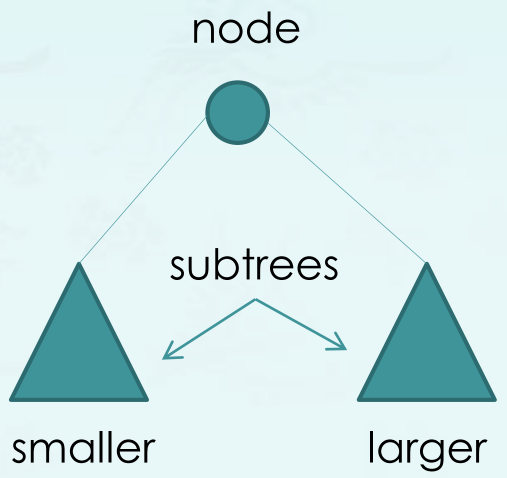
또 다윗이 이르되 여호와께서 나를 사자의 발톱과 곰의 발톱에서 건져내셨은즉 나를 이 블레셋 사람의 손에서도 건져내시리이다 사울이 다윗에게 이르되 가라 여호와께서 너와 함께 계시기를 원하노라 (삼상17:37)

하나님이 우리를 구원하사 거룩하신 소명으로 부르심은 우리의 행위대로 하심이 아니요 오직 자기의 뜻과 영원 전부터 그리스도 예수 안에서 우리에게 주신 은혜대로 하심이라 (딤후1:9)

Binary Search Trees: Definition

- **A binary tree (BT)** is a tree data structure in which each node has **at most two children**, which are referred to as the left child and the right child.
- **A binary search tree (BST)** is an ordered or sorted binary tree.
  - Each node in a binary search tree 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.)
  - Equal keys are ruled out.

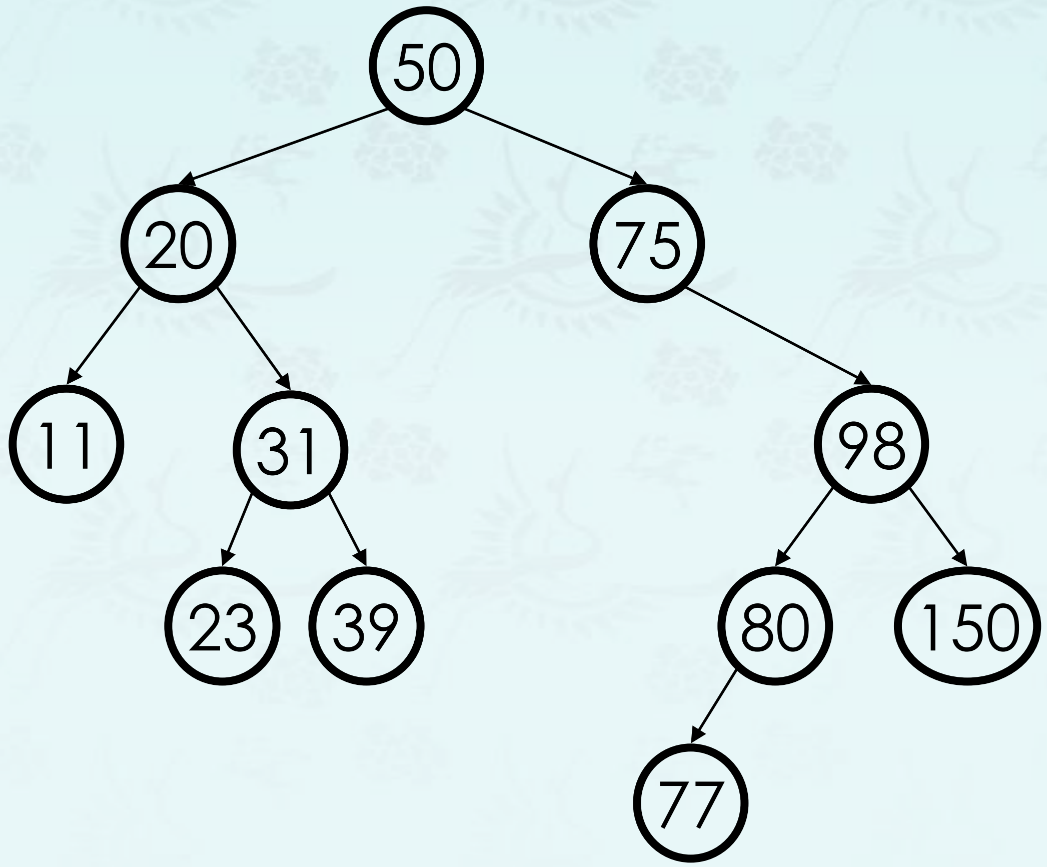


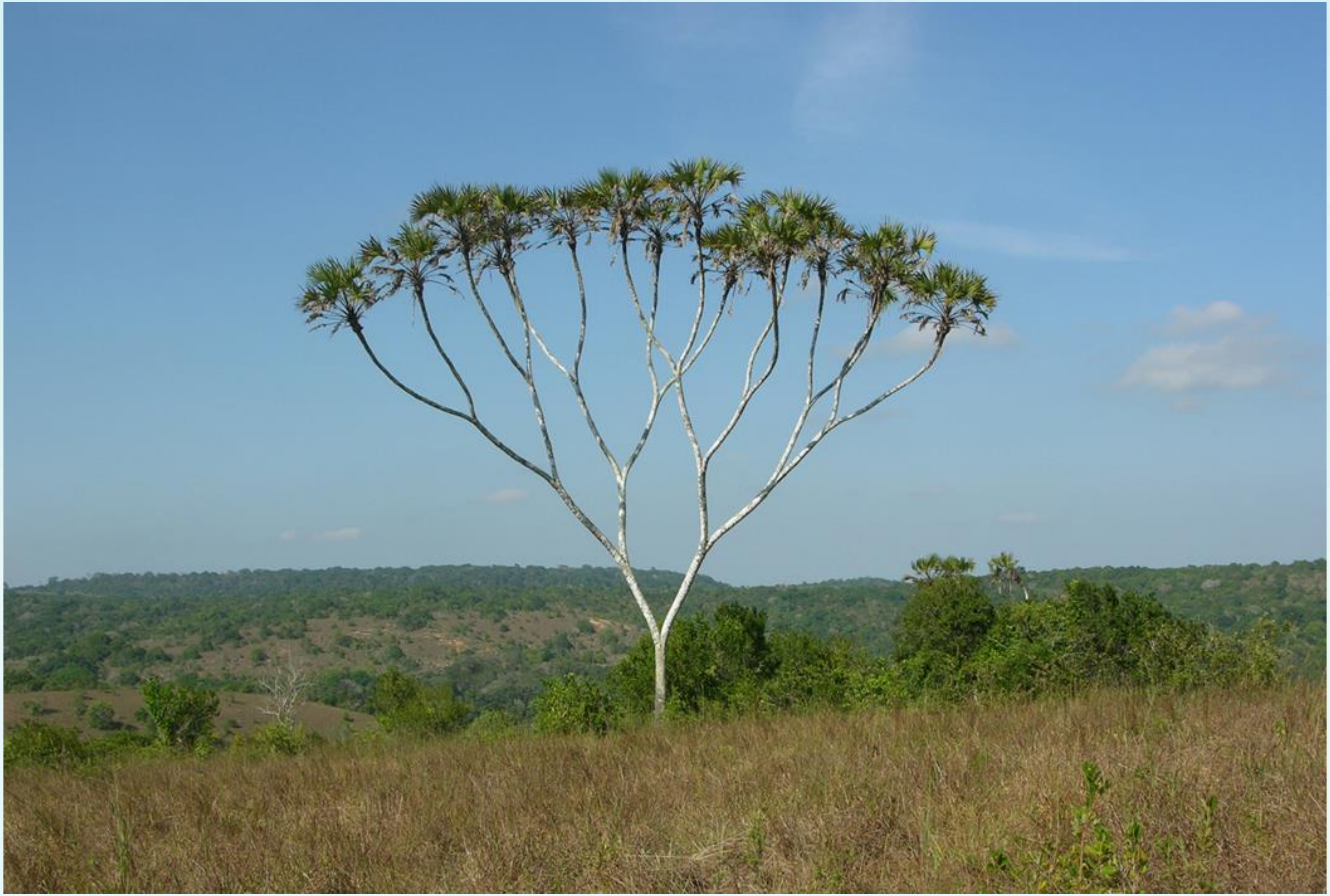


Binary Search Trees

- **Operations: insert(grow)**
  - Draw what a binary search tree would look like if the following values were added to an initially empty tree in this order:

50
20
75
98
80
31
150
39
23
11
77





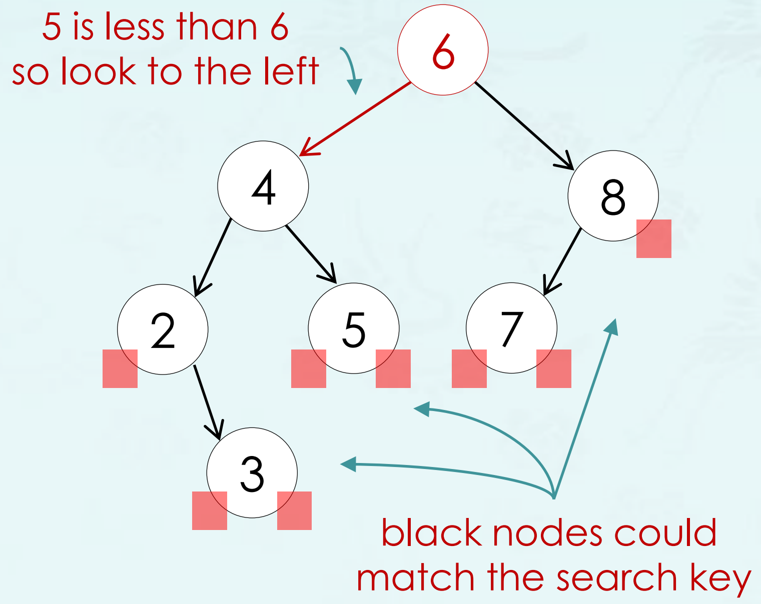
Binary Search Trees

- **Operations:**
  - Query – search(contains, find), size, height, min/max, successor, predecessor, distance
  - grow – insert
  - trim – delete
  - BT to BST conversion
  - **LCA** – Lowest Common Ancestor

Binary Search Tree: Operations

- **Search:** A recursive algorithm to search for a key in a BST follows immediately from the recursive structure: If the tree is empty, we have a search miss; if the search key is equal to the key at the root, we have a search hit. Otherwise, we search (recursively) in the appropriate subtree.
  - **`find(root, key)`** or **`contains(root, key)`**

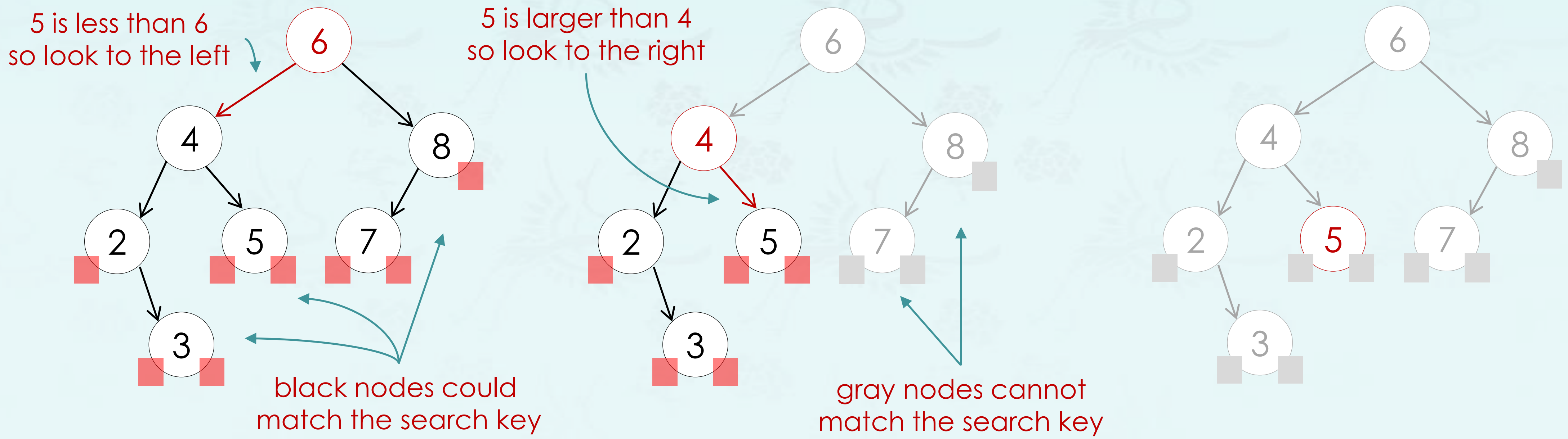
successful search for 5



Binary Search Tree: Operations

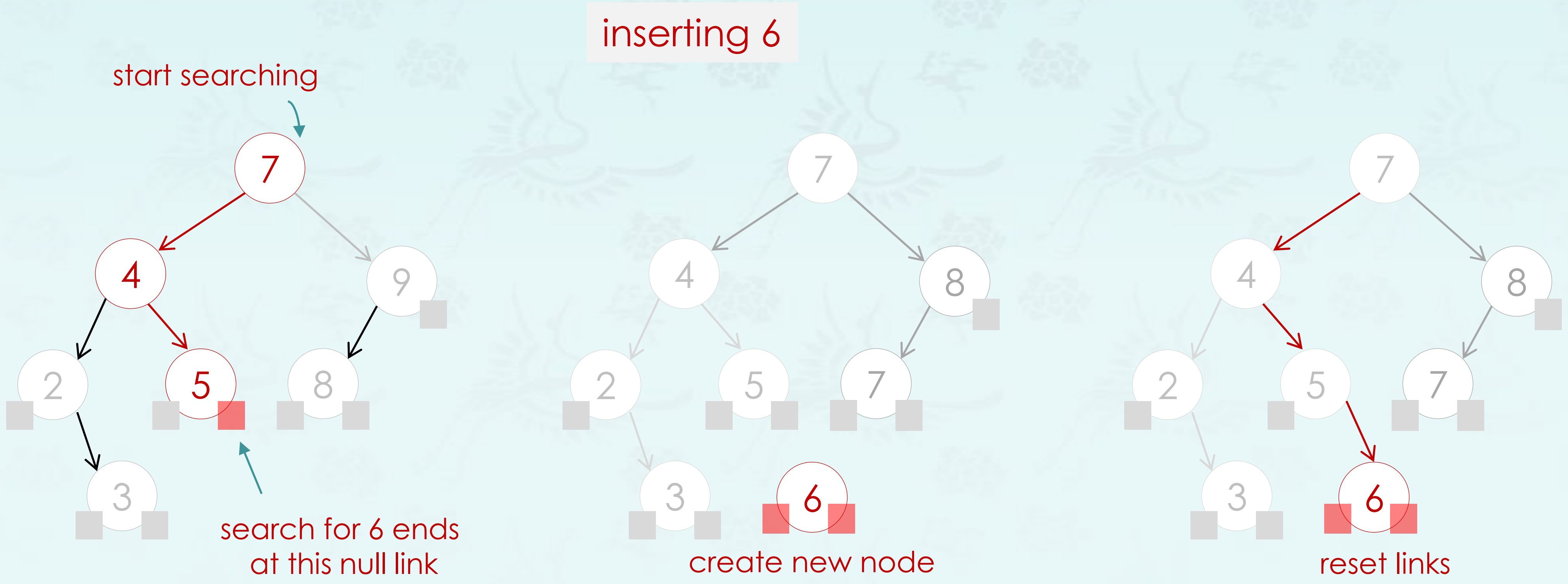
- **Search:** A recursive algorithm to search for a key in a BST follows immediately from the recursive structure: If the tree is empty, we have a search miss; if the search key is equal to the key at the root, we have a search hit. Otherwise, we search (recursively) in the appropriate subtree.
  - **`find(root, key)`** or **`contains(root, key)`**

successful search for 5



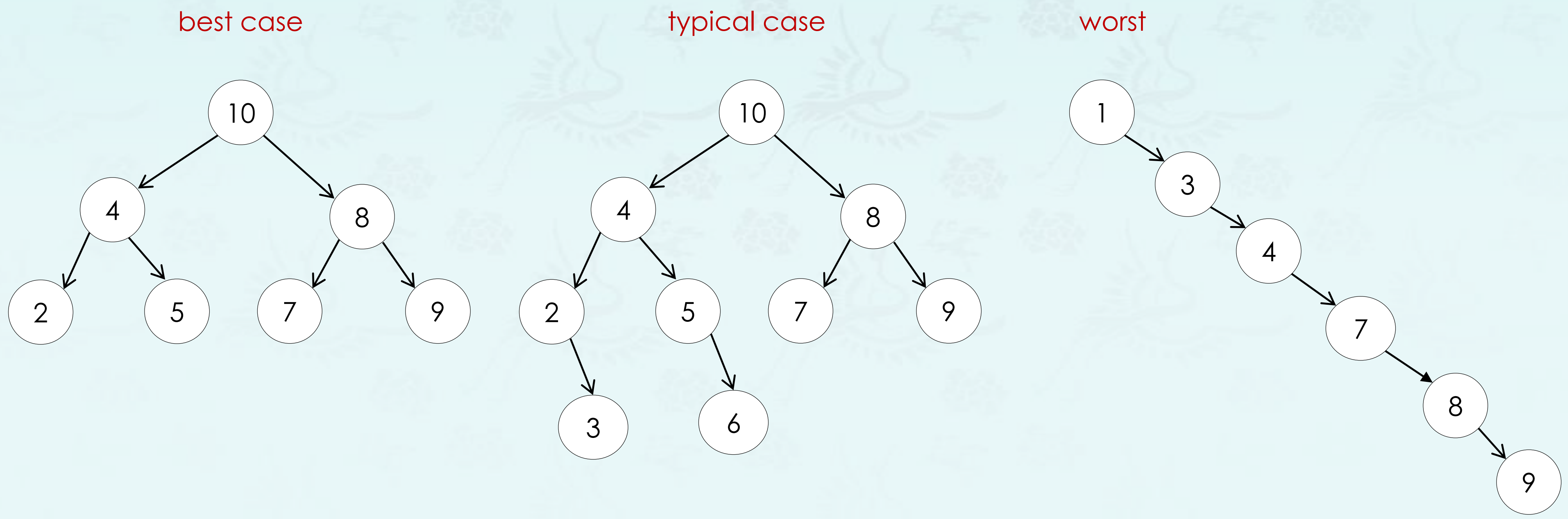
Binary Search Tree: Operations

- **Insert:** Insert is very similar to search. Indeed, a search for a key not in the tree ends at a null link, and all that we need to do is replace that link with a new node containing the key. .
  - `grow(root, 6)`



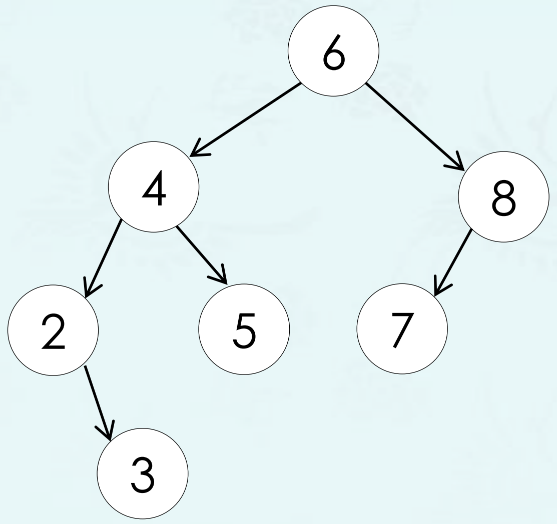
Binary Search Trees: Operations

- **Analysis:** The running times of algorithms on binary search trees depend on the shapes of the trees, which, in turn, depends on the order in which keys are inserted.
  - It is reasonable, for many applications, to use the following simple model: We assume that the keys are (uniformly) random, or, equivalently, that they are inserted in random order.
  - Insertion and search misses in a BST built from N random keys requires O(h), where h is the height of a BST. The typical case is ~ 1.39 $\log_2$ N compares on the average.



Operations: Search (or contains, find …)

```
bool containsIteration(tree node, int key) {
    if (node == nullptr) return false;
    while (node) {
        if (key == node->key) return true;
        if (key  < node->key)
            node = node->left;
        else
            node = node->right;
    }
    return false;
}
```



Operations: Search (or contains, find …)

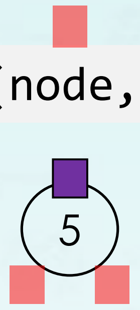
```
bool containsIteration(tree node, int key) {
    if (node == nullptr) return false;
    while (node) {
        if (key == node->key) return true;
        if (key  < node->key)
            node = node->left;
        else
            node = node->right;
    }
    return false;
}
```

```
bool contains(tree node, int key) {
    if (node == nullptr)   return false;
    if (key  == node->key) return true;

    if (key < node->key)
        return contains(node->left, key);

    return contains(node->right, key);
}
```

Operations: Insert (or grow)

- grow(node, k) - Insert a node with k

```
grow(node,key);
```

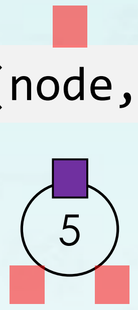


```
int main() {
  tree root = nullptr;
  int key = 5;
  ...
  grow(root, key);
  cout << root->key << endl;
  ...
} // with bugs
```

Operations: Insert (or grow)

- grow(node, k) - Insert a node with k

```
grow(node,key);
```



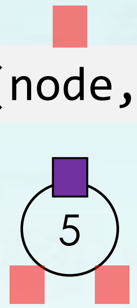
```
int main() {
  tree root = nullptr;
  int key = 5;
  ...
  root = grow(root, key);
  cout << root->key << endl;
  ...
}
```

Operations: Insert (or grow)

- grow(node, k) - Insert a node with k

```
tree grow(tree node, int key) {
  if (node == nullptr) return new tree(key);
  if (key < node->key)
    grow(node->left, key);
  else if (key > node->key)
    grow(node->right, key);
  return node;
} // with bugs
```

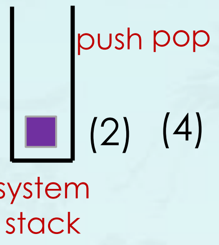
```
grow(node,key);
```



```
int main() {
  tree root = nullptr;
  int key = 5;
  ...
  grow(root, key);
  cout << root->key << endl;
  ...
} // with bugs
```

Operations: Insert (or grow)

- grow(node, k) - Insert a node with k



🟪 push before call

🟪 pop after call

(1) (2)

```
tree grow(tree node, int key) {
  if (node == nullptr) return new tree(key);
  if (key < node->key)
    grow(node->left, key);
  else if (key > node->key)
    grow(node->right, key);
  return node;
} // with bugs
```

(3)

(4)

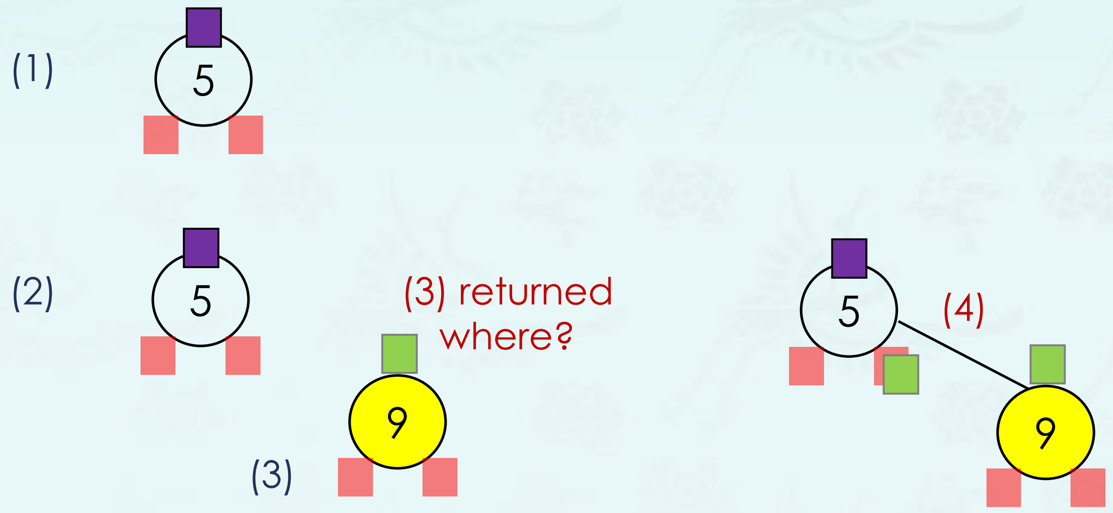
grow() is done.
new node is returned
but it is not saved

(4)

```
node->right = grow(node->right, key);
```

```
int main() {
  tree root = nullptr;
  int key = 5;
  ...
  root = grow(root, key);
  cout << root->key << endl;
  root = grow(root, 9);
  ...
}
```

```
grow(node,key);
```



Operations: Insert (or grow)

- grow(node, k) - Insert a node with k

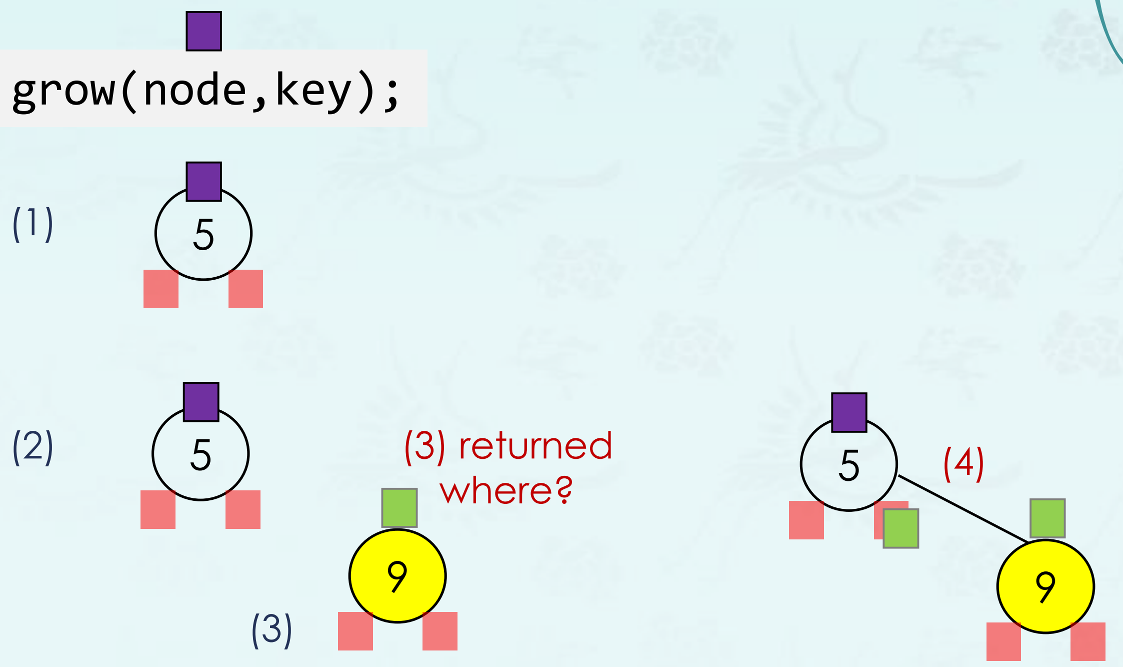
```
tree grow(tree node, int key) {
  if (node == nullptr) return new tree(key);
  if (key < node->key)
    node->left  = grow(node->left, key);
  else if (key > node->key)
    node->right = grow(node->right  , key);
  return node;
}
```

(4)

```
node->right = grow(node->right, key);
```

```
int main() {
  tree root = nullptr;
  int key = 5;
  ...
  root = grow(root, key);
  cout << root->key << endl;
  root = grow(root, 9);
  ...
}
```

```
grow(node,key);
```

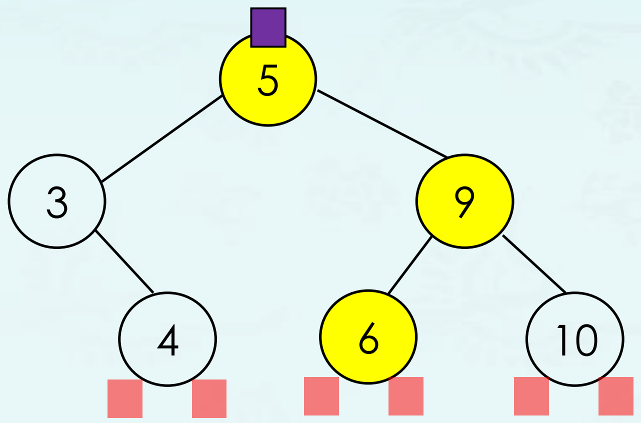


Operations: Insert (or grow)

- 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;
}
```

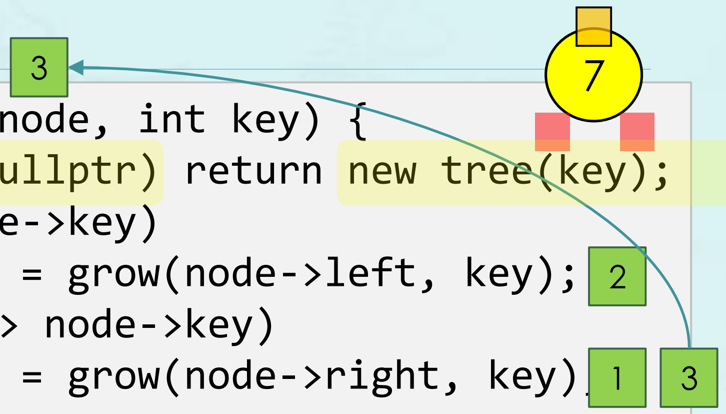
```
grow(node,7)
```



Operations: Insert (or grow)

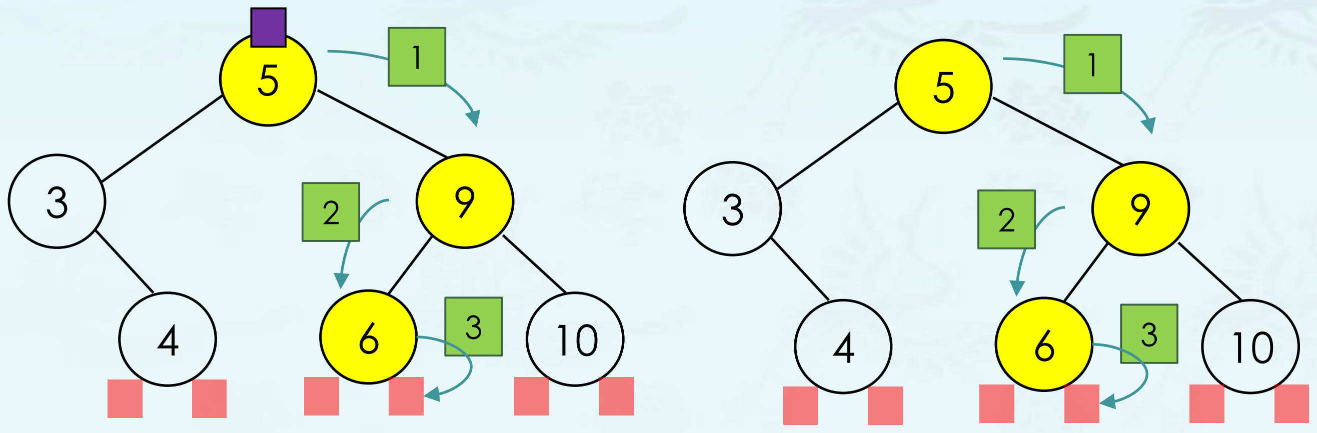
- grow(node, k) - Insert a node with k
  - **Step 1**: If the tree is empty, return a new node(k).
  - **Step 2**: Pretending to search for k in BST, until locating a nullptr.
  - **Step 3**: create a new node(k) and link it.

```
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;
}
```



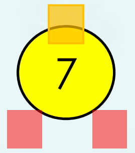
```
grow(node,7)
```

The highlight nodes are compared with key 7.



after node(6) & key(7) compared, it calls **grow(nullptr, 7)** 3

🟨 = new tree(7)

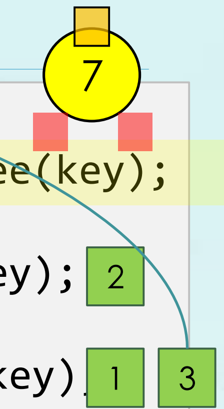


Where does it return to?
6 and 7 are **not** linked.

Operations: Insert (or grow)

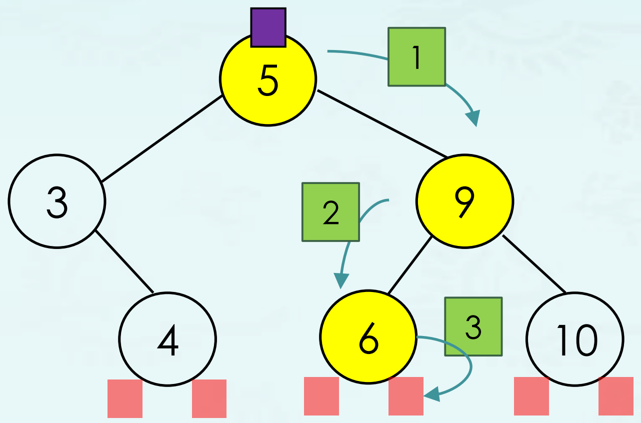
- grow(node, k) - Insert a node with k
  - **Step 1**: If the tree is empty, return a new node(k).
  - **Step 2**: Pretending to search for k in BST, until locating a nullptr.
  - **Step 3**: create a new node(k) and link it.

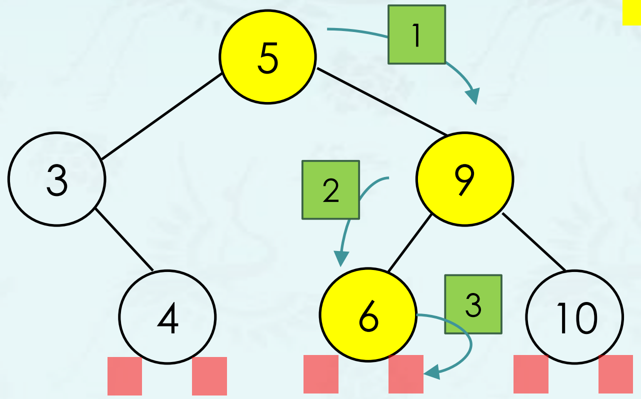
```
tree grow(tree node, int key) {
  if (node == nullptr) return new tree(key);
  if (key < node->key)
    node->left  = grow(node->left, key);
  else if (key > node->key)
    node->right = grow(node->right, key);
  return node;
}
```



```
grow(node,7)
```

The highlight nodes are compared with key 7.

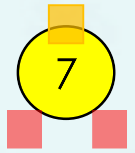




Where does it return to?

after node(6) & key(7) compared, it calls **grow(nullptr, 7)** 3

🟨 = new tree(7)

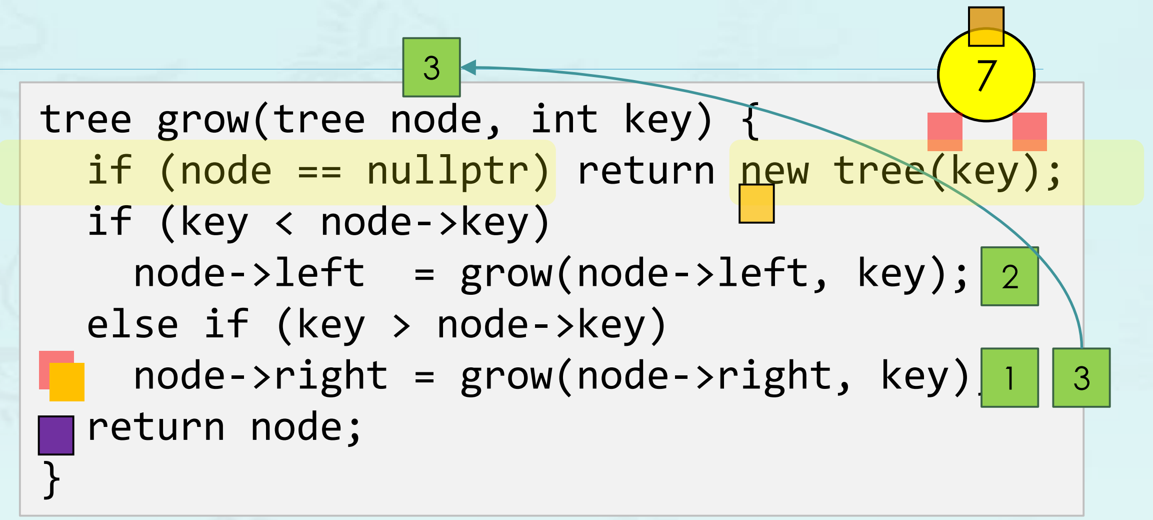


Where does it return to?
6 and 7 are **not** linked.

Operations: Insert (or grow)

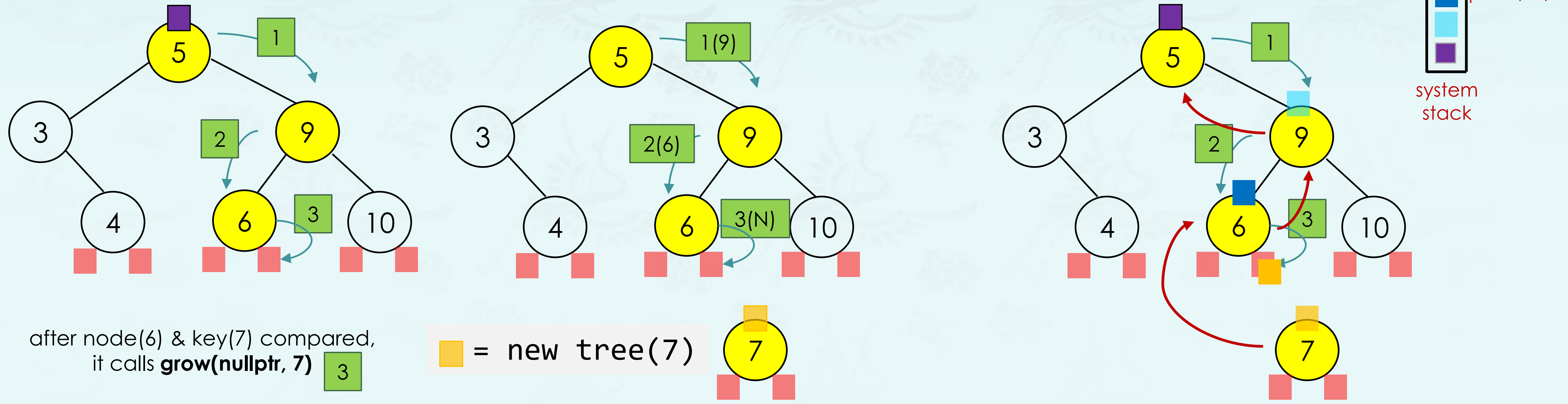
- grow(node, k) - Insert a node with k
  - **Step 1**: If the tree is empty, return a new node(k).
  - **Step 2**: Pretending to search for k in BST, until locating a nullptr.
  - **Step 3**: create a new node(k) and link it.

```
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;
}
```



`grow(node,7)`

The highlight nodes are compared with key 7.



after node(6) & key(7) compared, it calls **grow(nullptr, 7)** 3

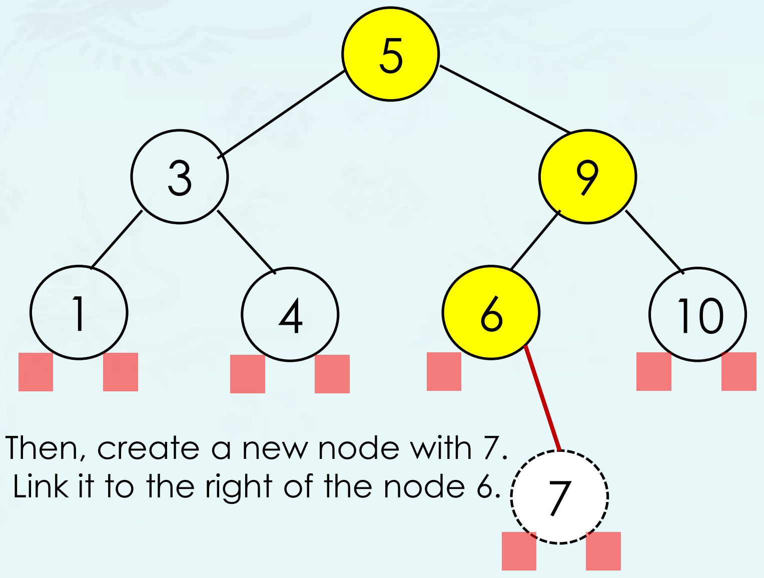
= new tree(7)

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?


5
3
9
1
4
6
10
7
Then, create a new node with 7.
Link it to the right of the node 6.

Data Structures
Chapter 5 Tree

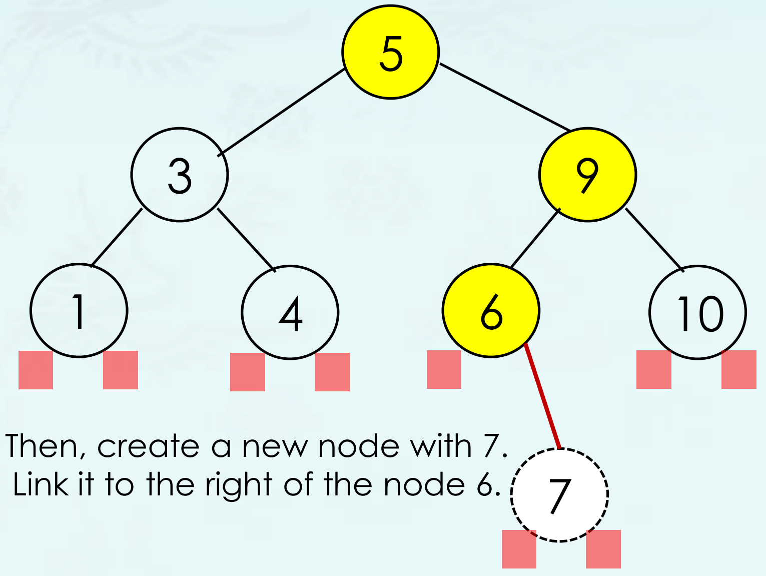
1. Introduction
2. Binary Tree
3. **Binary Search Tree**
   - Definition
   - Operations
   - Demo & Coding
4. Balancing Tree

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.

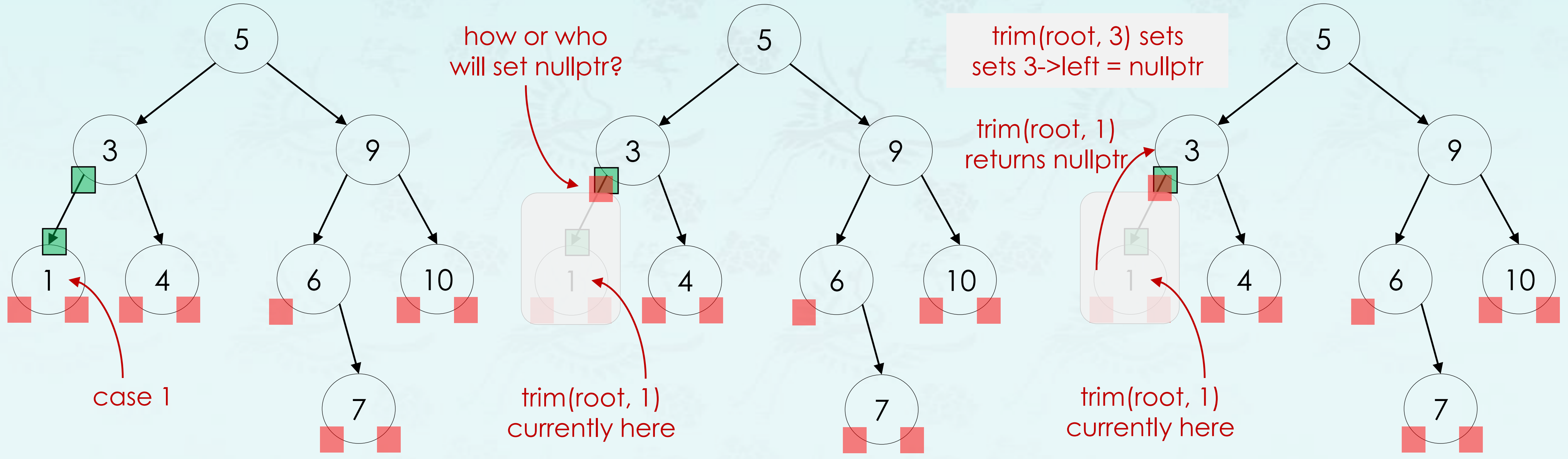
```
tree grow(tree node, int key) {
  if (node == nullptr)
    return new tree(key);

  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?


5
3
9
1
4
6
10
Then, create a new node with 7.
Link it to the right of the node 6.
7

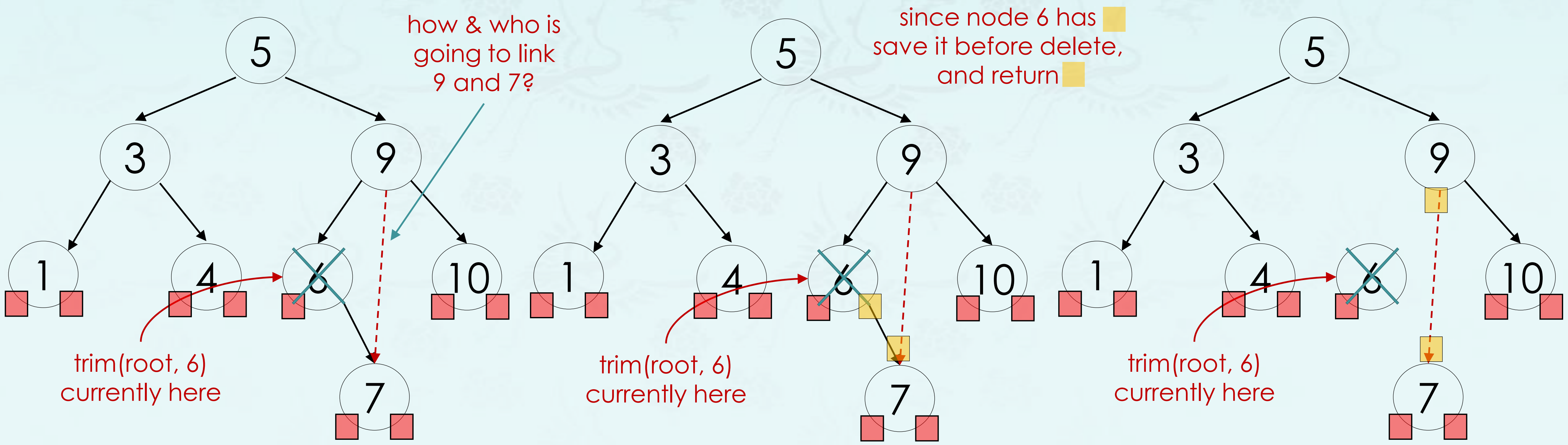
Operations: delete (or trim)

- 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.


how & who is going to link 9 and 7?
since node 6 has
save it before delete,
and return
trim(root, 6) currently here
trim(root, 6) currently here
trim(root, 6) currently here
5
3
9
1
4
6
10
7

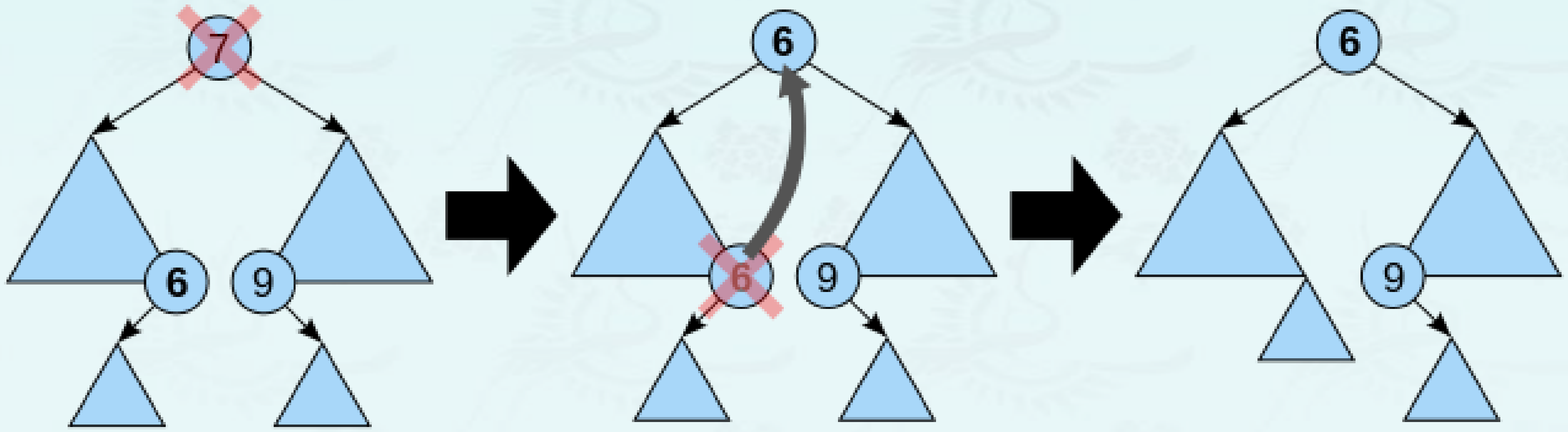
Operations: delete (or trim)

- 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)

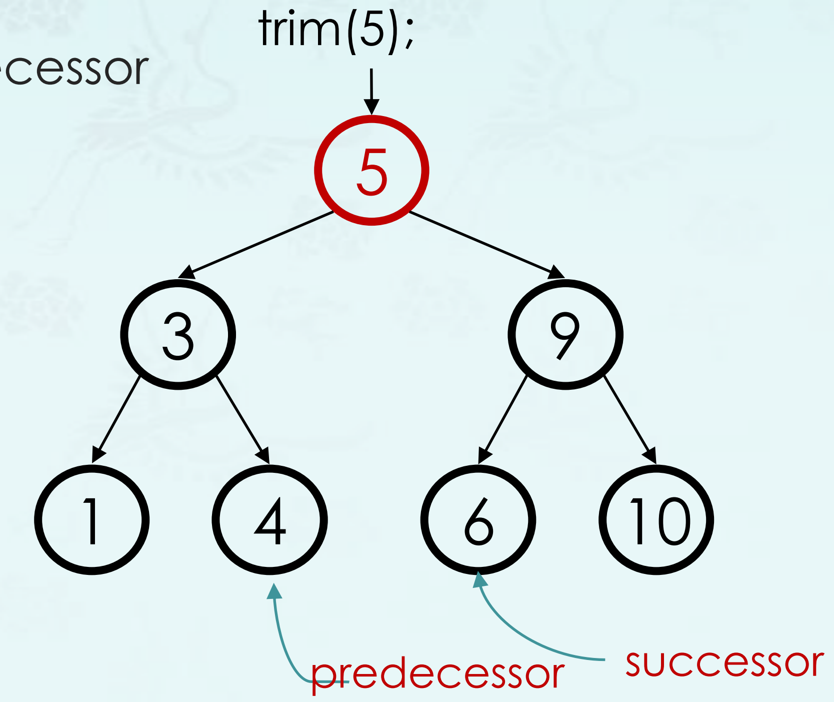
- Case 3: **Two children**
  1. The rightmost node in the left subtree, the inorder **predecessor 6**, is identified.
  2. Its value is copied into the node being trimmed.
  3. The inorder **predecessor** can then be trimmed because it has at most one child.

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



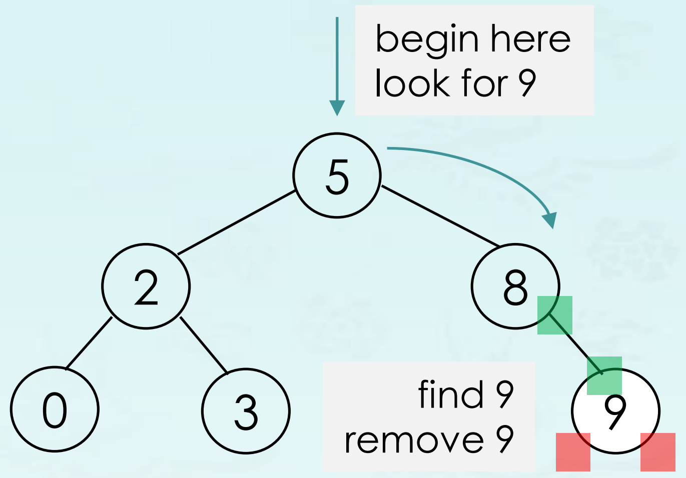
Operations: delete (or trim)

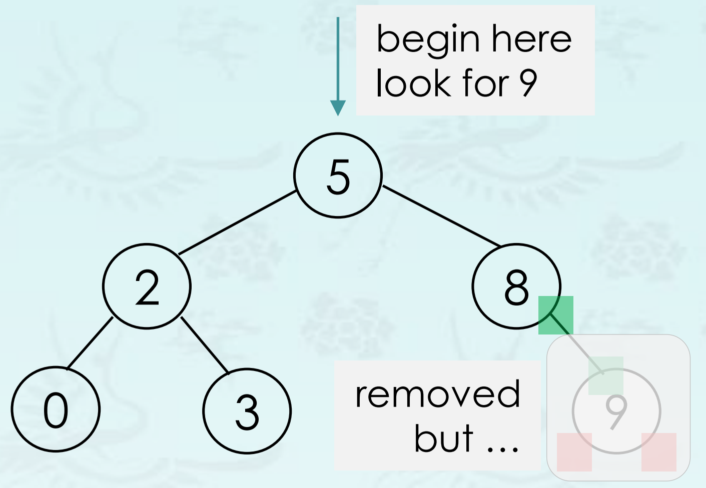
- Case 3: **Two children**
  - Idea: Replace the trimmed node with a value guaranteed to be between two child subtrees
- Options:
  - predecessor from left subtree: maximum(node->left)
  - successor from right subtree: minimum(node->right)
  - These are the easy cases of predecessor/successor
  - Now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



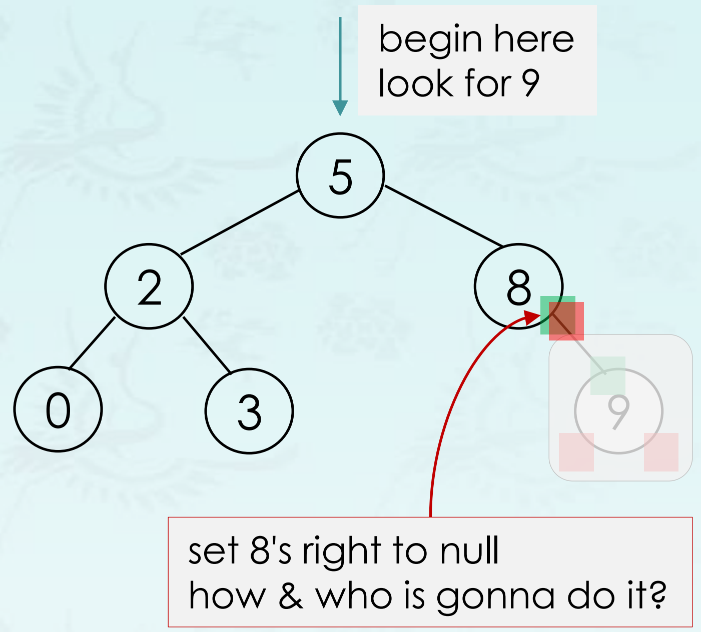
Operations: delete (or trim)

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





```
...
delete node;
...
```

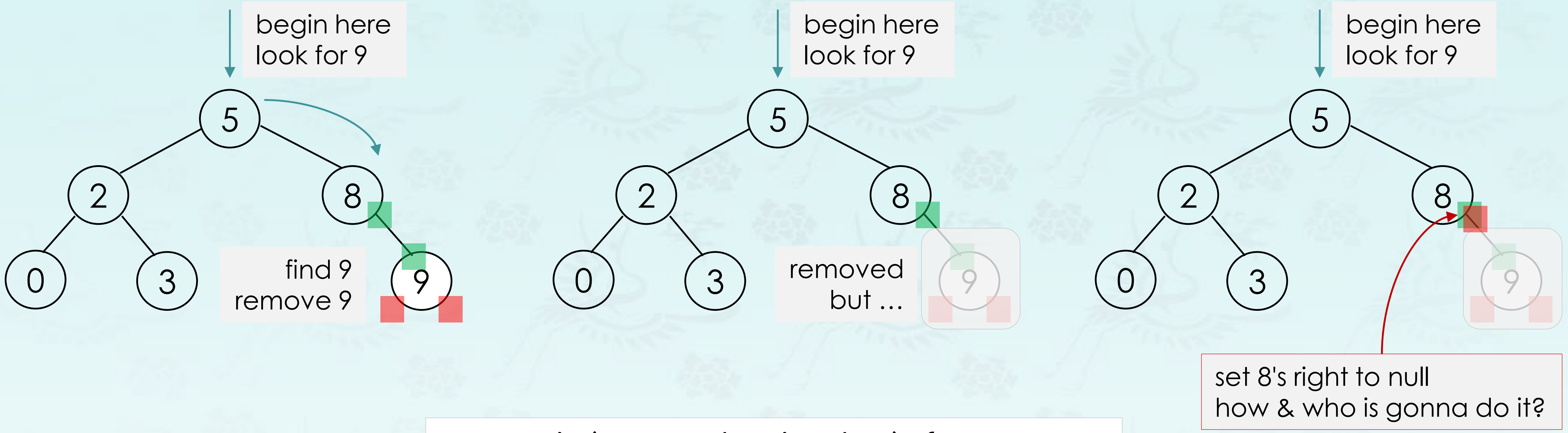


```
...
int key = 9;
root = trim(root, key);

...
```

```
tree trim(tree node, int key) {
  if (node == nullptr) return node;
  ...
  return node;
}
```

Operations: delete (or trim)

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



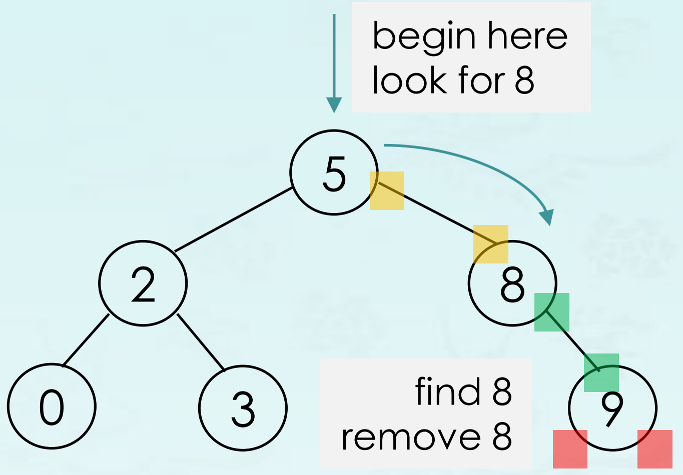
```
...
int key = 9;
root = trim(root, key);

...
```

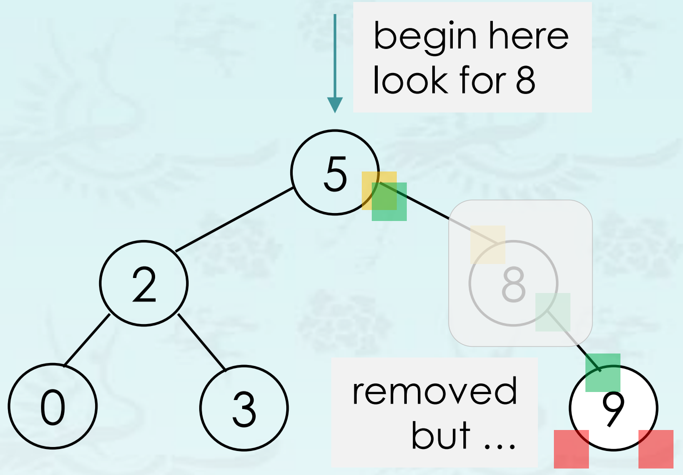
```
tree trim(tree node, int key) {
  if (node == nullptr) return node;
  ...
  else if (key > node->key)
    node->right = trim(node->right, key);
  ...
  return node;
}
```

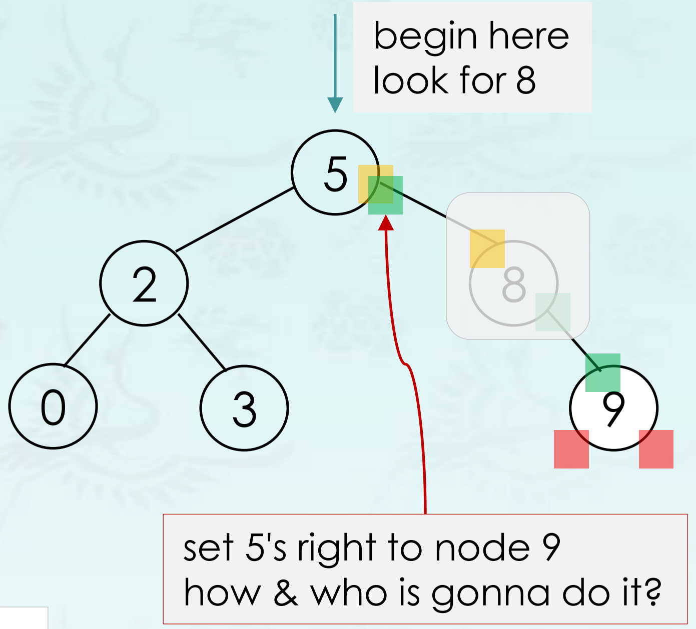
```
... // no child case
  delete node;
  return nullptr;
...
```

Operations: delete (or trim)

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







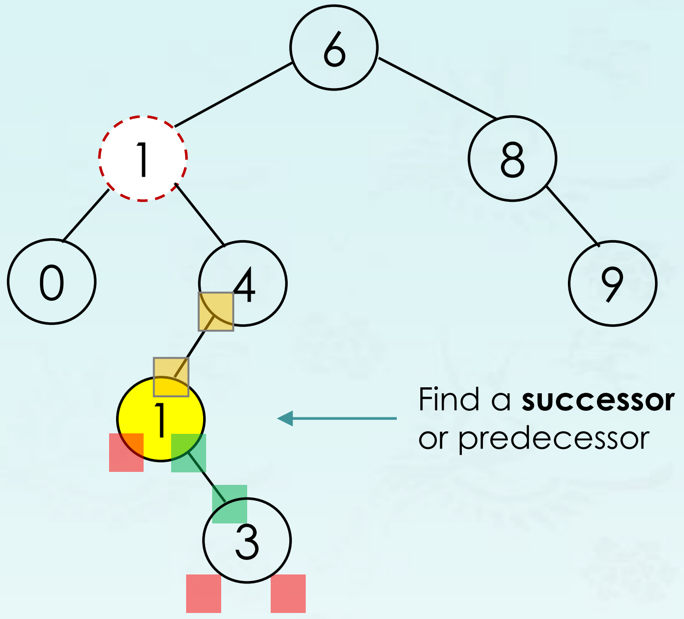
```
...
int key = 8;
root = trim(root, key);

...
```

```
tree trim(tree node, int key) {
  if (node == nullptr) return node;
  ...
  else if (key > node->key)
    node->right = trim(node->right, key);
  ...
  return node;
}
```

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

Operations: delete (or trim)

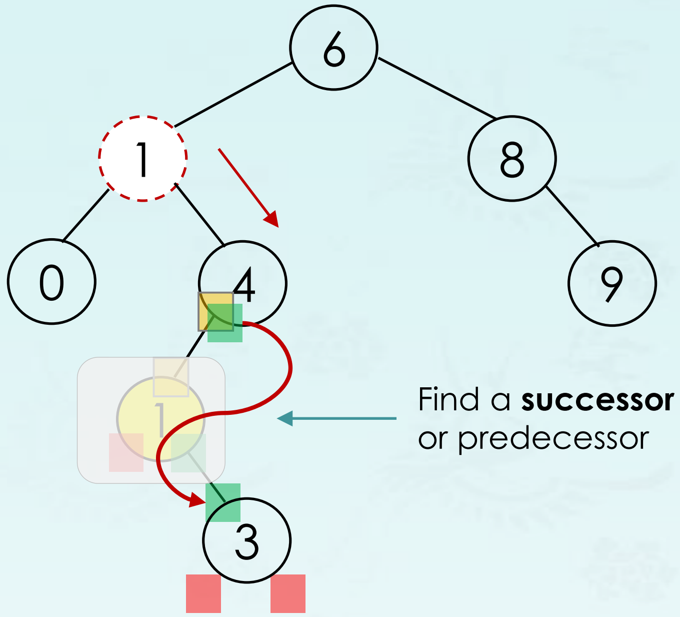
- **Example:** Case 3: Two children



```
1. find the node 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)
```

Some thoughts:

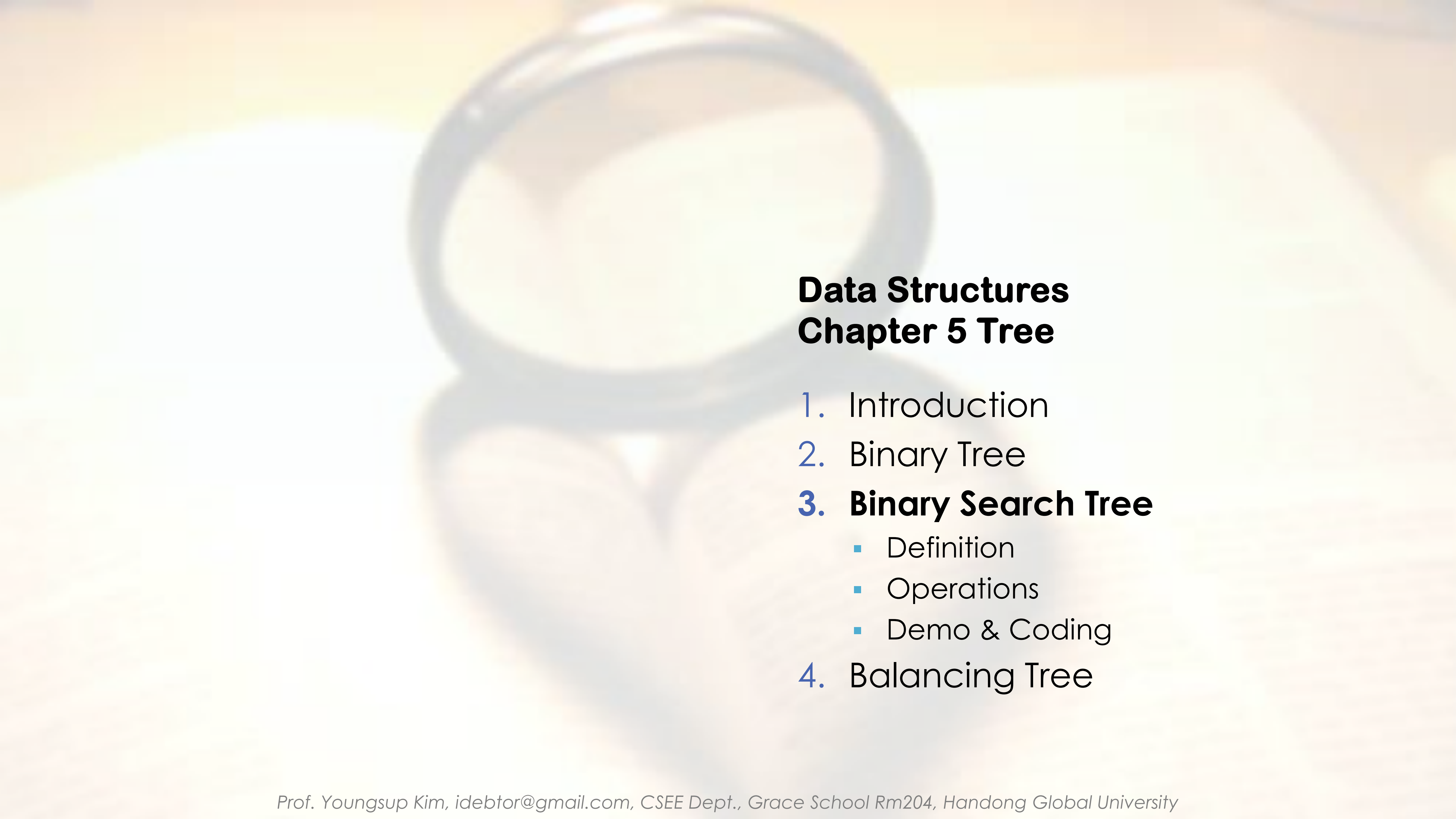
- Step 2 Get the heights of two subtree first.
  - If right subtree height is larger, then use the successor. Otherwise use the predecessor to shorten the tree height.
- Step 4 simply uses the code for one-child case deletion.

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

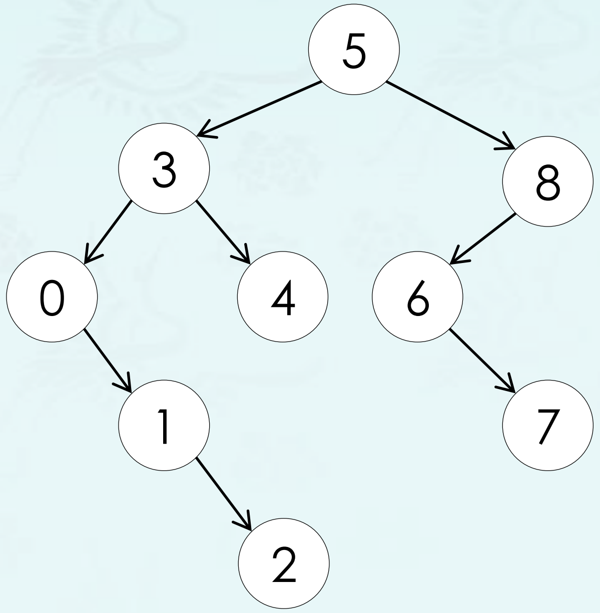
Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204, Handong Global University

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);
}
```

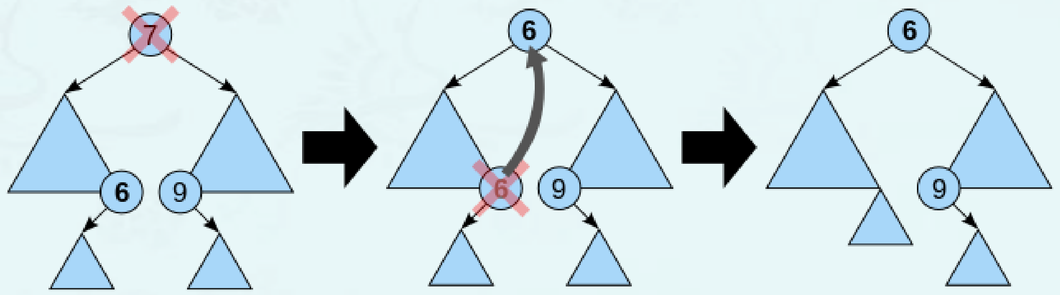
```
tree maximum(tree node) {  // returns right-most node key


}
```



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

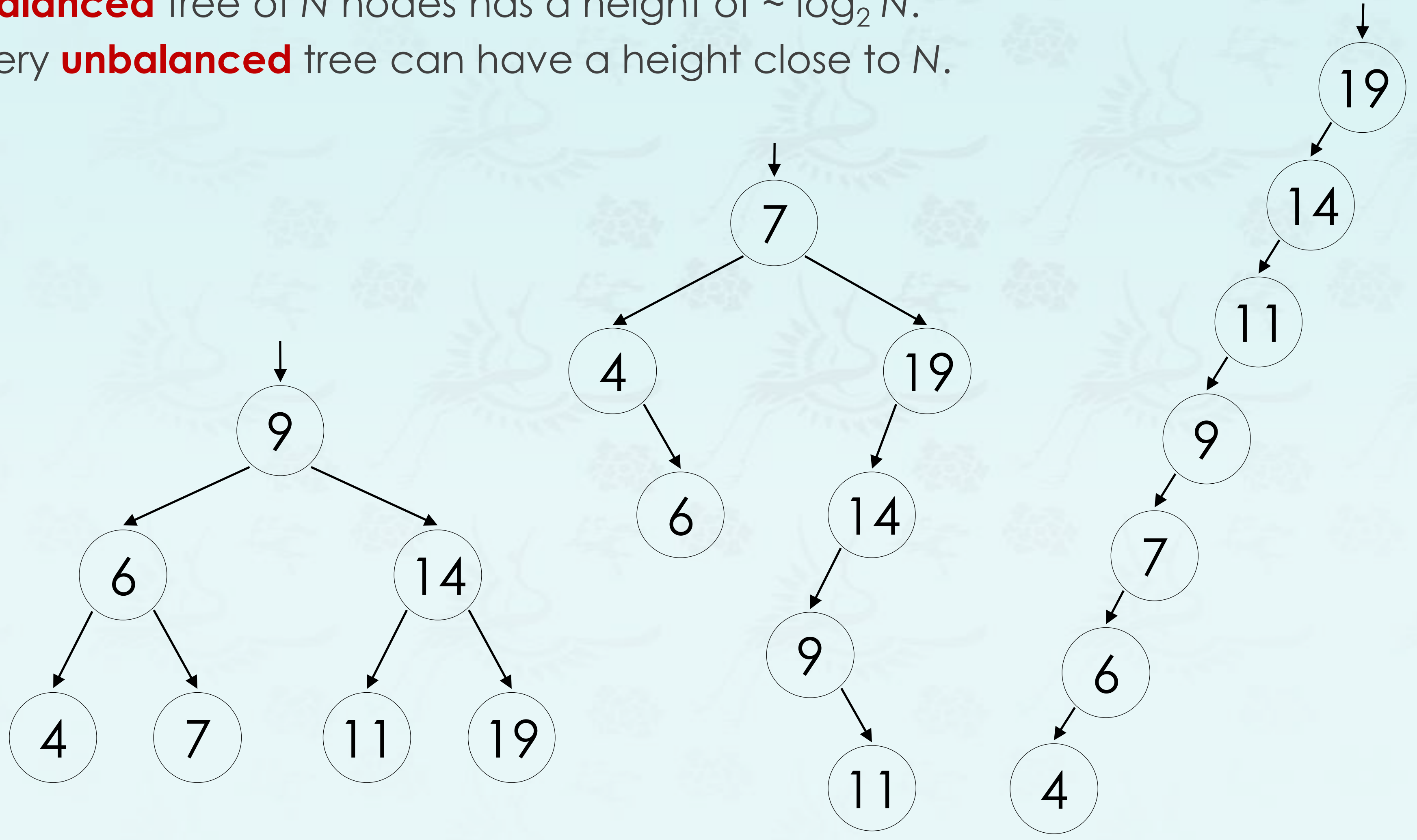
- Successor
  - If the given node has a right subtree then by the BST property the next larger key must be in the right subtree. Since all keys in a right subtree are larger than the key of the given node, the successor must be the smallest of all those keys in the right subtree.
- Predecessor
  - If 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 successor 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 root) {
  if (node != nullptr && node->right != nullptr)
    return minimum(node->right);
  return nullptr;
}
```



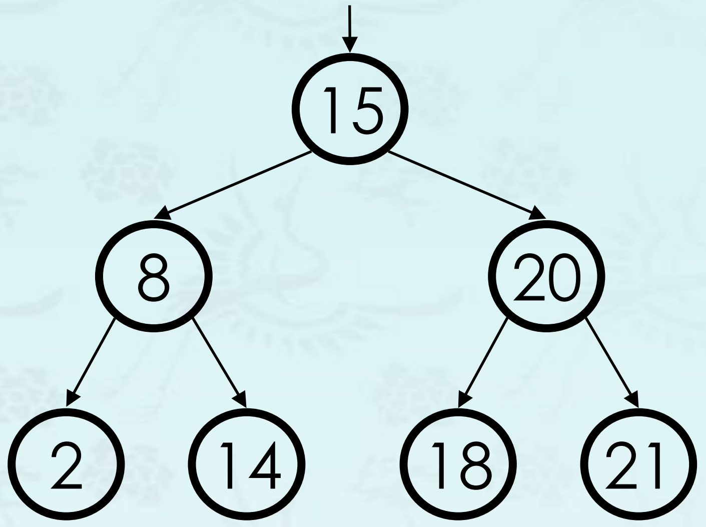
Binary Search Trees: Observations

- What do you see in the following BSTs?
  - A **balanced** tree of *N* nodes has a height of ~ $\log_2 N$.
  - 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, ..., *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 max + 1 to max + 1 + N, where max is the maximum value of keys in the tree.
- This function growN() is provided for your reference^^.

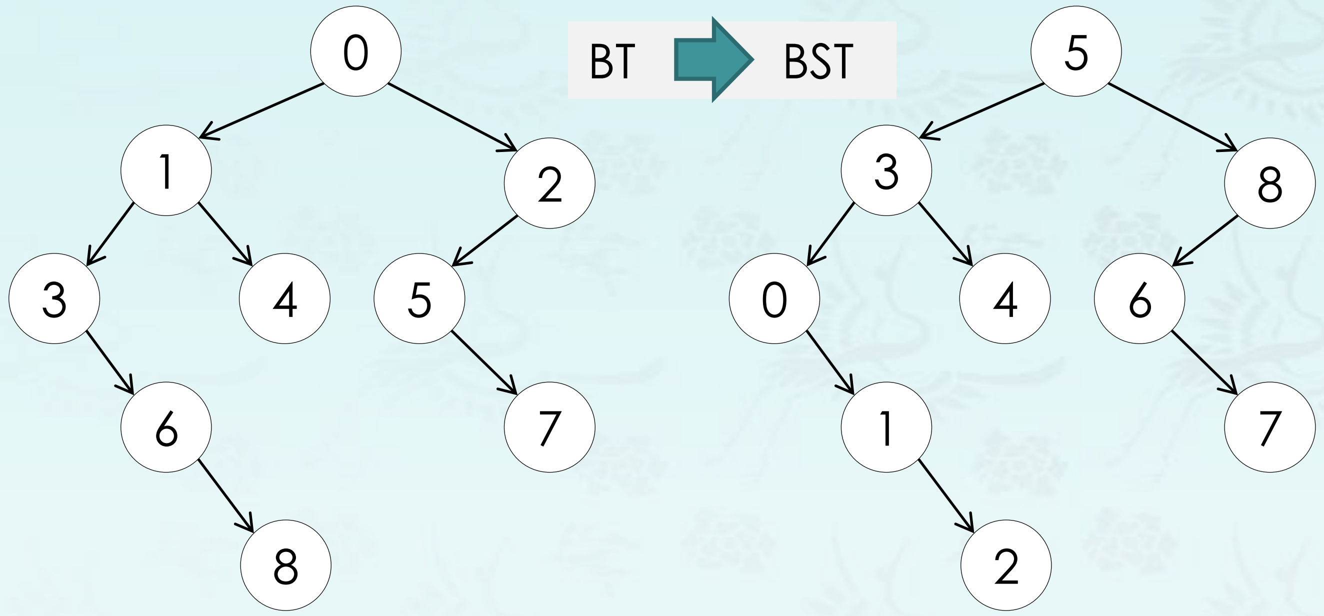
Operations: growN() & trimN() for testing

- 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.

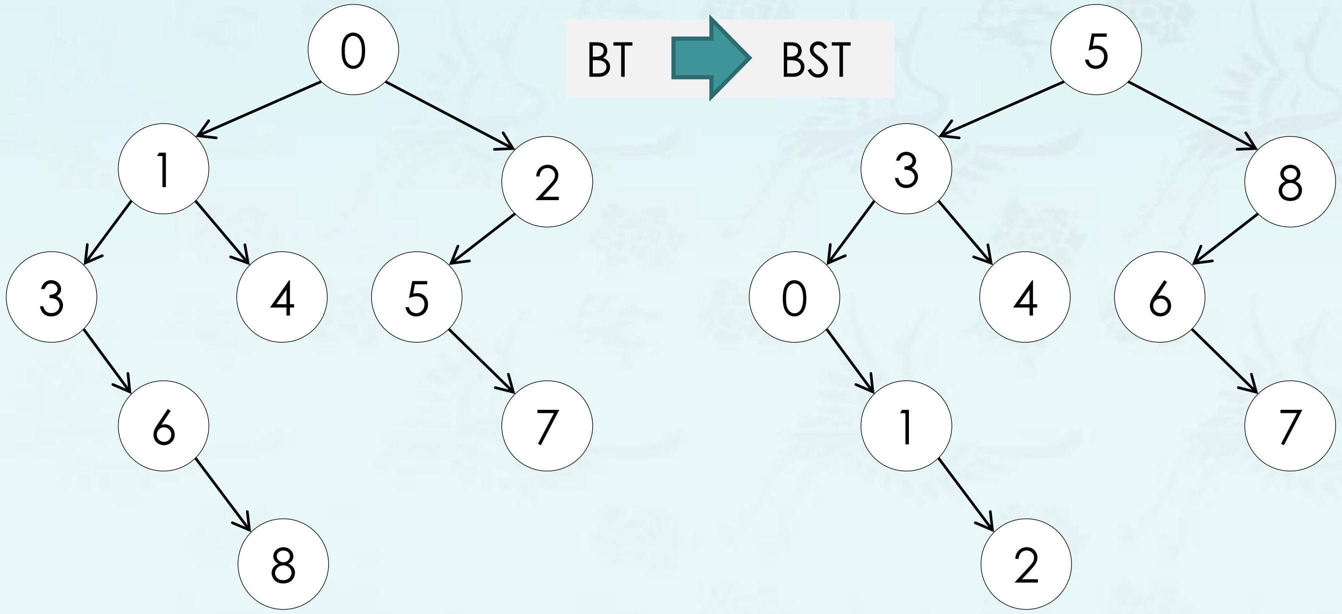
Convert BT to BST in-place

- 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.
- 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.

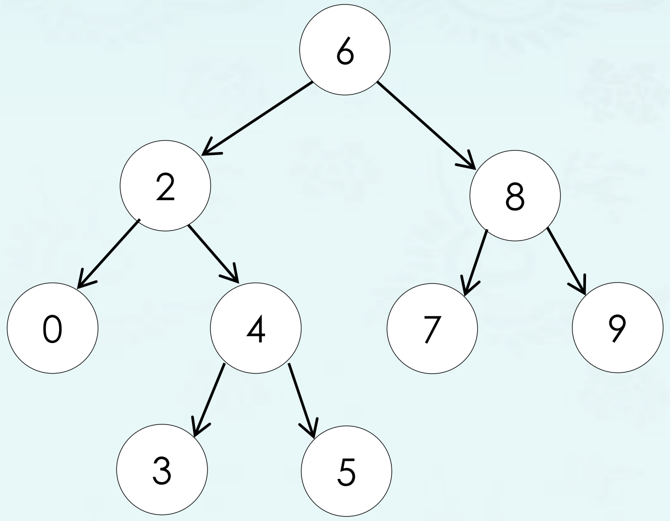


(1) Retrieve the keys from BT:
    3 6 8 1 4 0 5 7 2   // if in-order used
(2) Sort keys in the container:
    0 1 2 3 4 5 6 7 8
(3) Replace keys in BT with sorted keys while in-order traversal.

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

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

Operations: LCA in BST

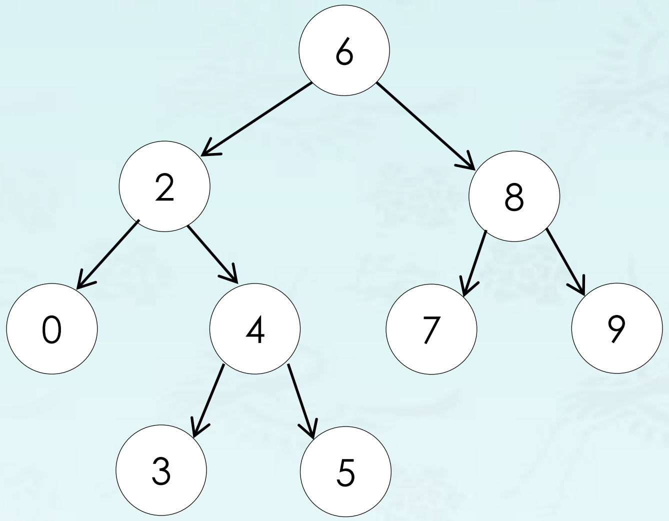
- Find the lowest common ancestor(LCA) of two given nodes, given in BST.
  - The LCA is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself)."
  - In BST, all of the nodes' values will be unique.
    Two nodes given, p and q, are different and both values will exist in the BST.



```
For example:
2, 8 -> 6
2, 5 -> 2
9, 5 -> 6
8, 7 -> 8
0, 5 -> 2
```

Operations: LCA(**iteration**) in BST

- **Intuition (Iteration)**: Traverse down the tree iteratively to **find the split point**. The point from where p and q won't be part of the same subtree or when one is the parent of the other.

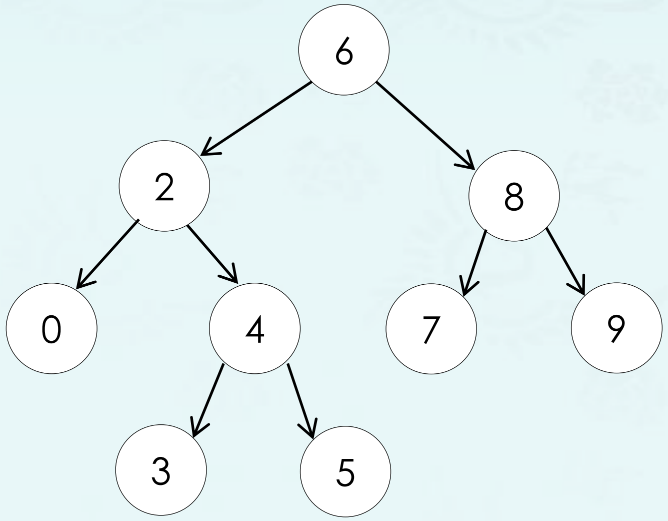


```
For example:
2, 5 -> 2
9, 7 -> 6
0, 4 -> 8
0, 5 -> 2
2, 7 -> 6
```

```
tree LCAiteration(tree root, tree p, tree q) {
  while (node != nullptr) {
     // your code here
     if (both p & q > root)
        node move to right to search
     else if (both q & q < root)
        node moves to left to search
     else
        LCA found
  }
  return node;
} // iteration solution
```

Operations: LCA(**recursion**) in BST

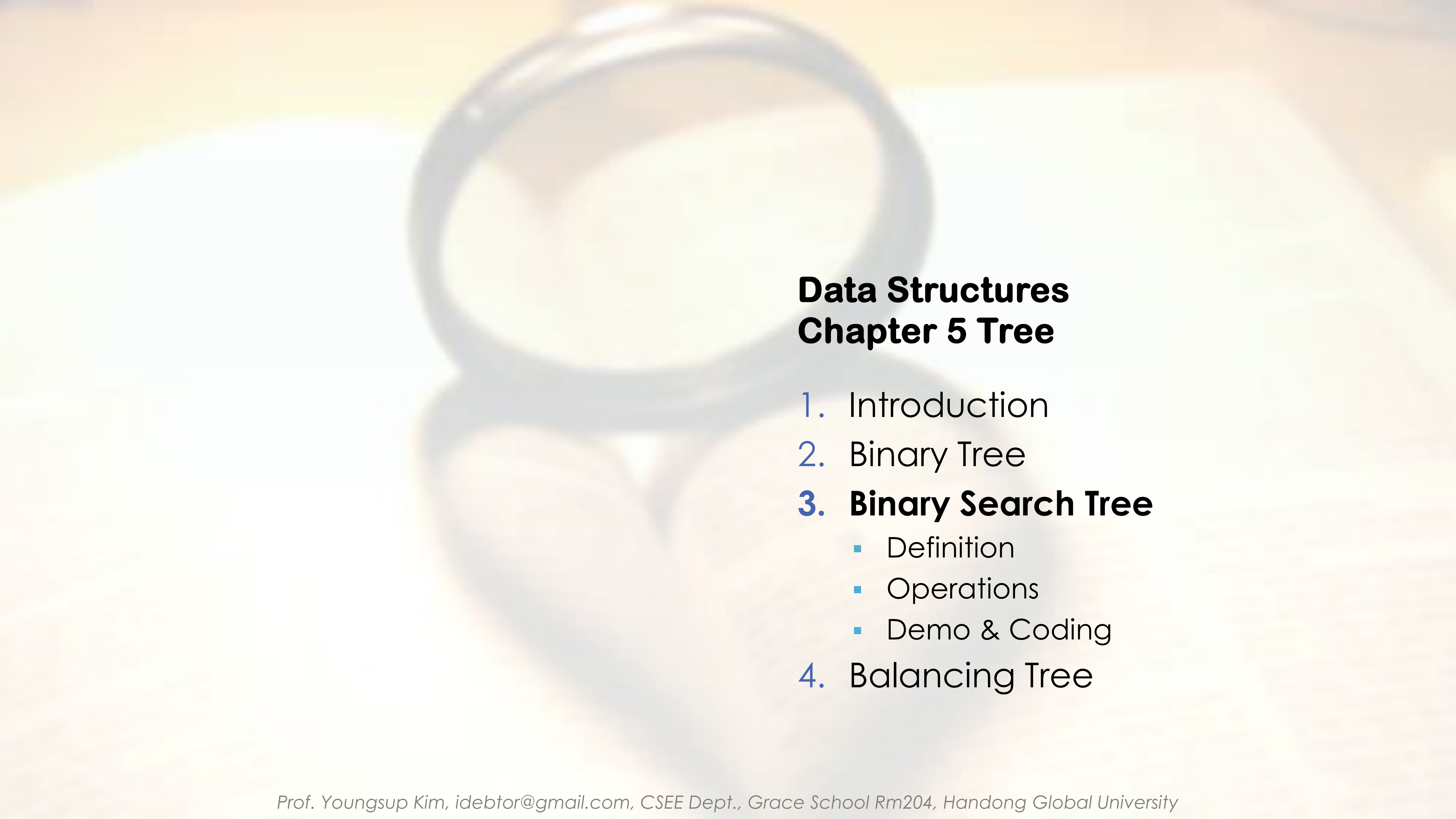
- **Algorithm: (Recursion)**
  1. Start traversing the tree from the root node.
  2. If both the nodes p and q are in the right subtree, then continue the search with right subtree starting step 1.
  3. If both the nodes p and q are in the left subtree, then continue the search with left subtree starting step 1.
  4. If both step 2 and step 3 are **not true**, this means we have **found** the node which is common to node p's and q's subtrees. Hence we return this common node as the LCA.



```
tree LCA(tree root, tree p, tree q) {


  // your code here


} // recursive solution
```

Operations: LCA in BST

- Recursion Algorithm
  - Time Complexity: $O(N)$, where N is the number of nodes in the BST. In the worst case we might be visiting all the nodes of the BST.
  - Space Complexity: $O(N)$. This is because the maximum amount of space utilized by the recursion stack would be N since the height of a skewed BST could be N.
- Iteration Algorithm
  - Time Complexity : O(N), where N is the number of nodes in the BST. In the worst case we might be visiting all the nodes of the BST.
  - Space Complexity : O(1).



Data Structures
Chapter 5 Tree

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

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