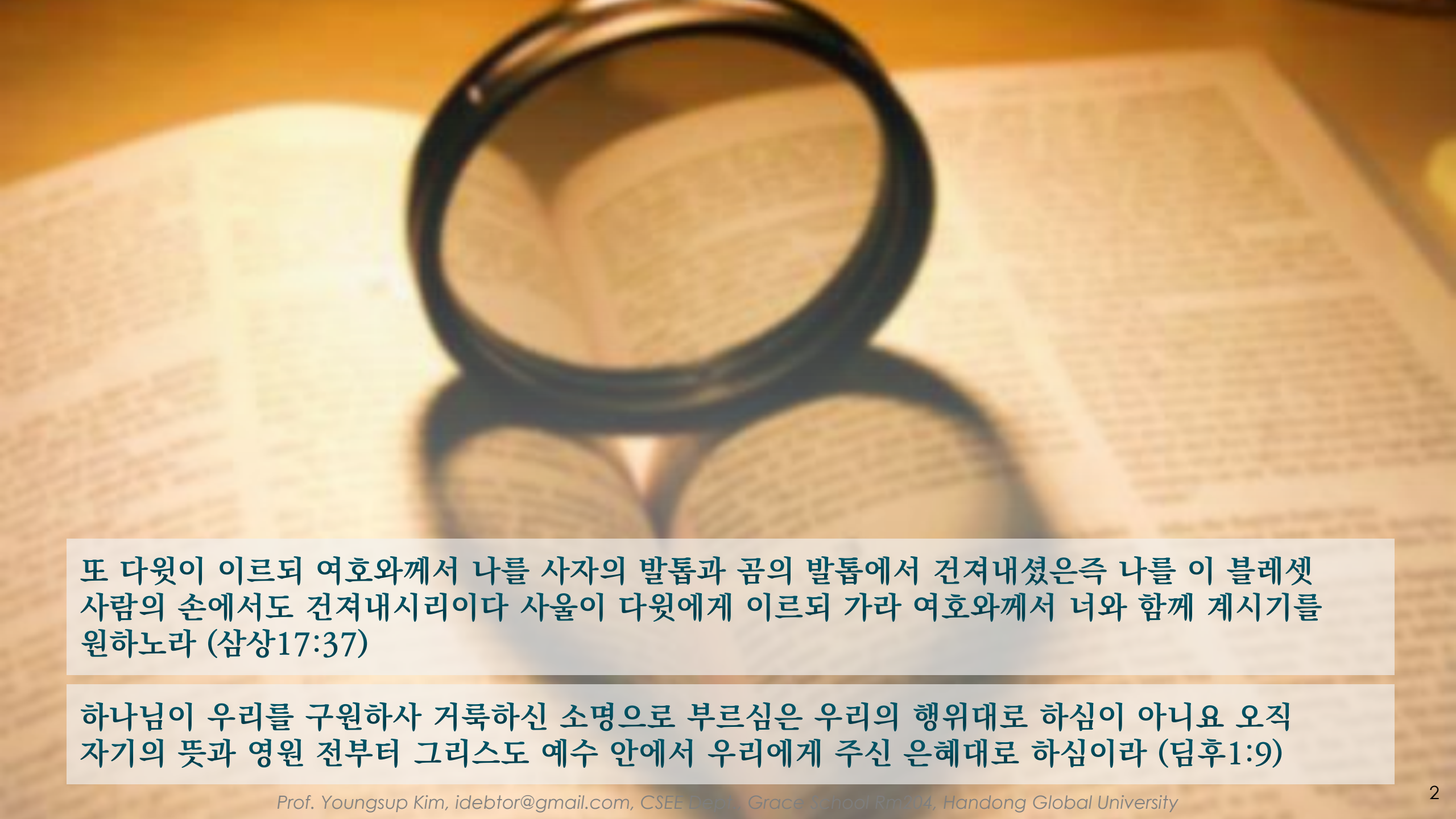
Data Structures
Chapter 5 Tree

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

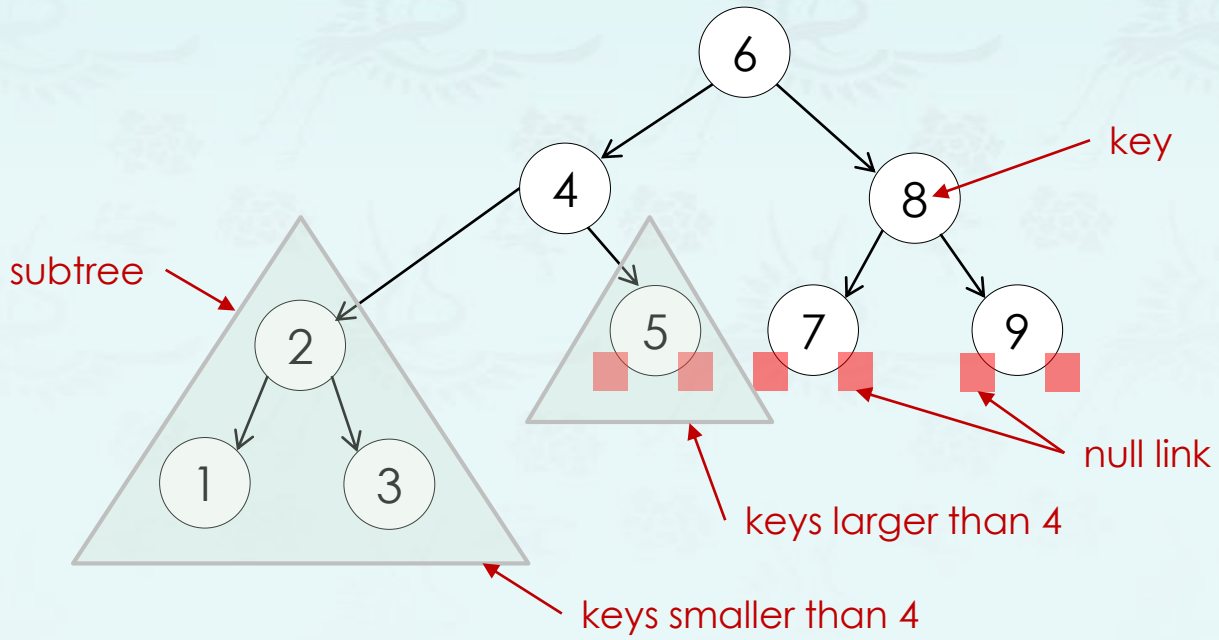


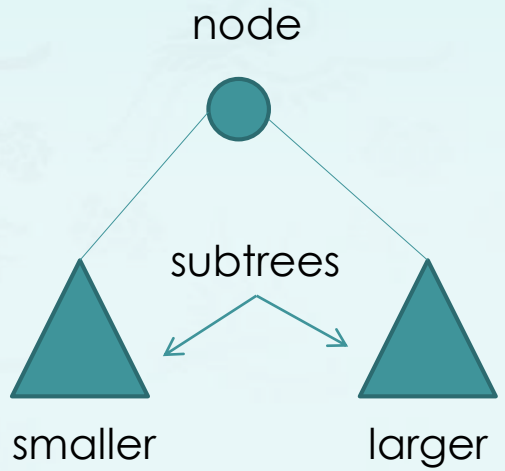
또 다윗이 이르되 여호와께서 나를 사자의 발톱과 곰의 발톱에서 건져내셨은즉 나를 이 블레셋 사람의 손에서도 건져내시리이다 사울이 다윗에게 이르되 가라 여호와께서 너와 함께 계시기를 원하노라 (삼상17:37)

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.

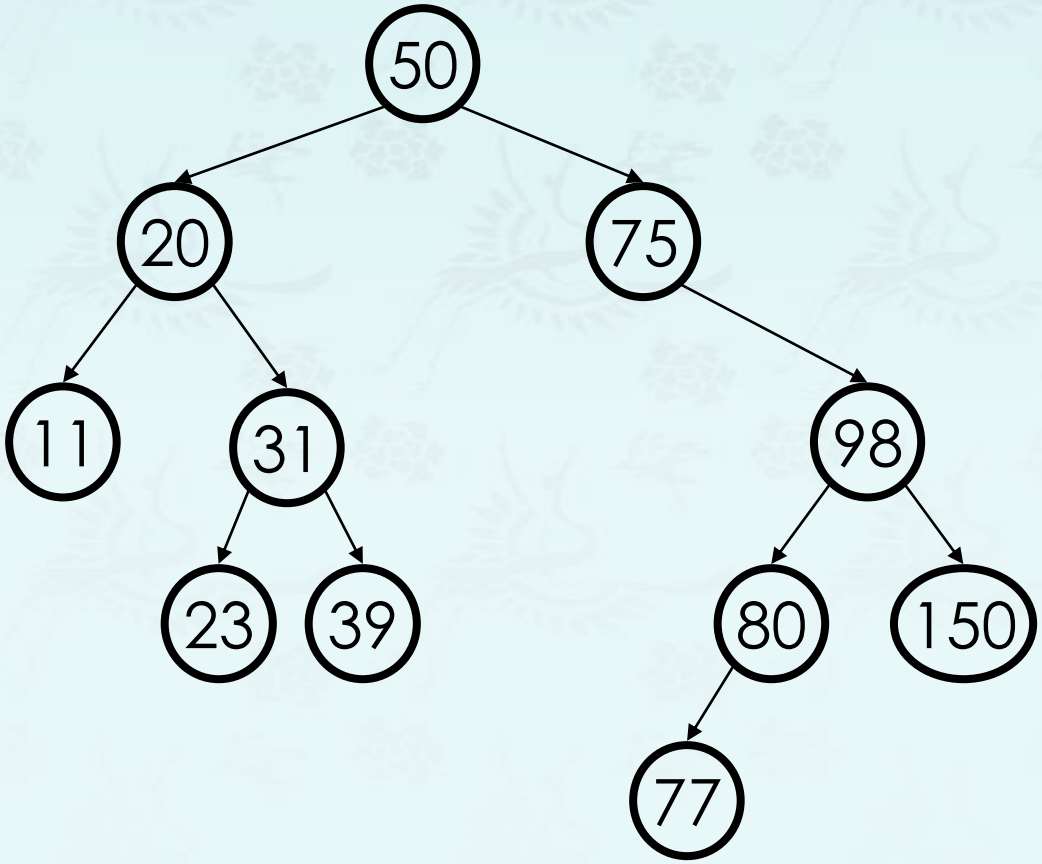




Binary Search Trees

- **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



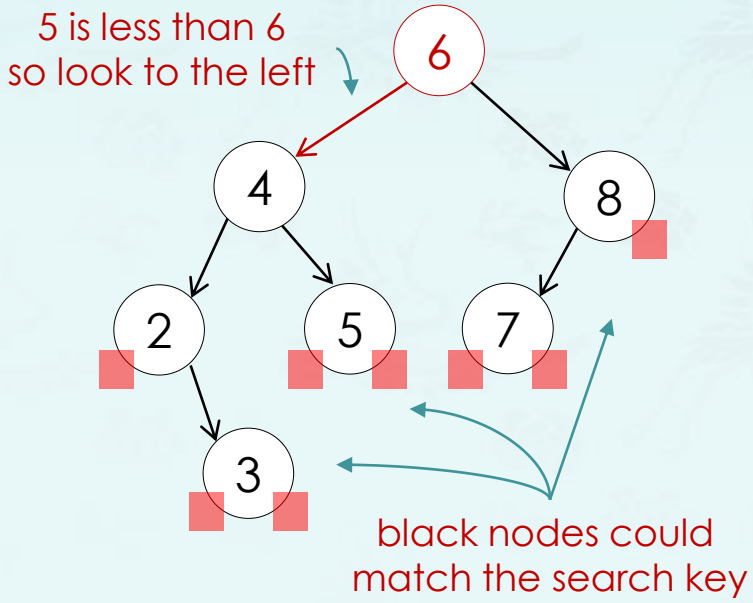
Binary Search Trees

- **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

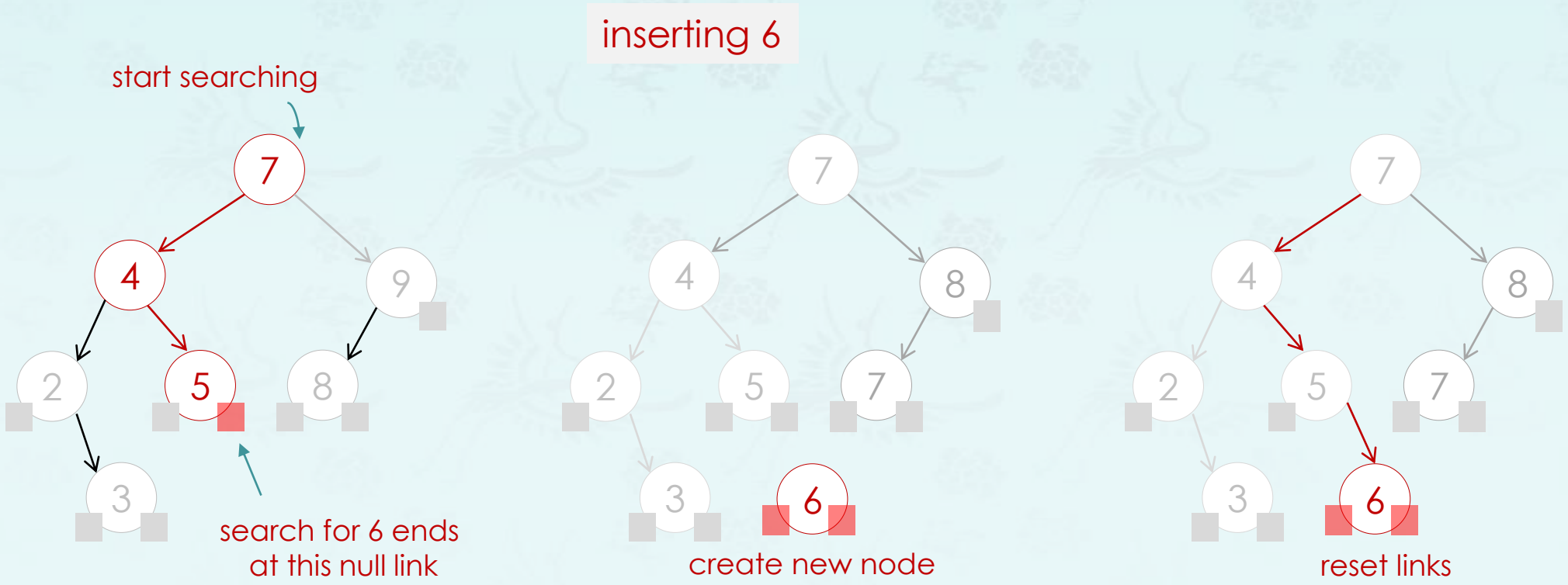


Binary Search Tree: Operations

- **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)`**

Binary Search Tree: Operations

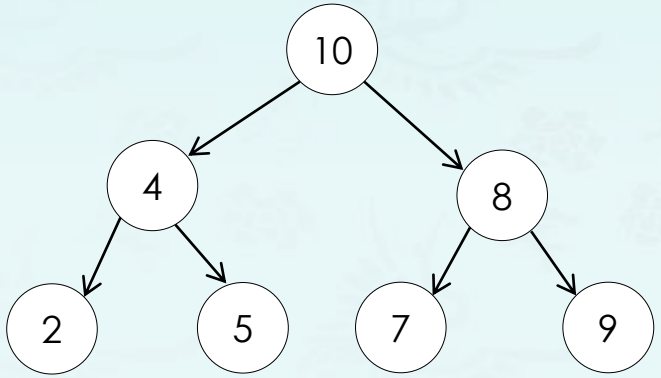
- **Insert:** Insert is very similar to search. Indeed, a search for a key not in the tree ends at a null link, and all that we need to do is replace that link with a new node containing the key. .
  - `grow(root, 6)`



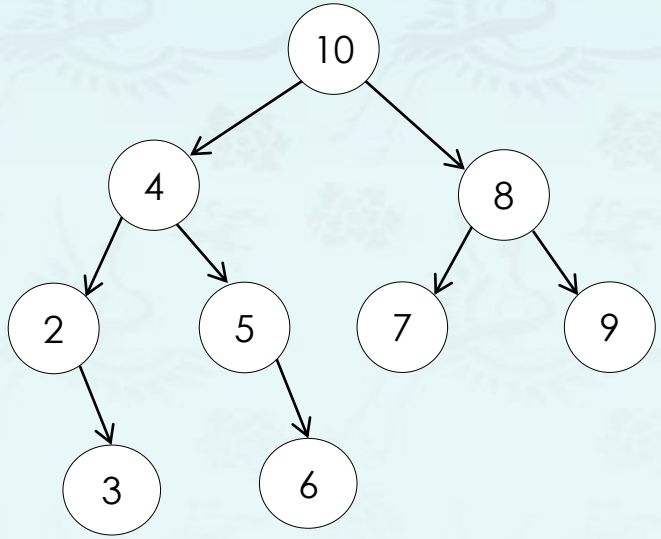
Binary Search Trees: Operations

- **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.

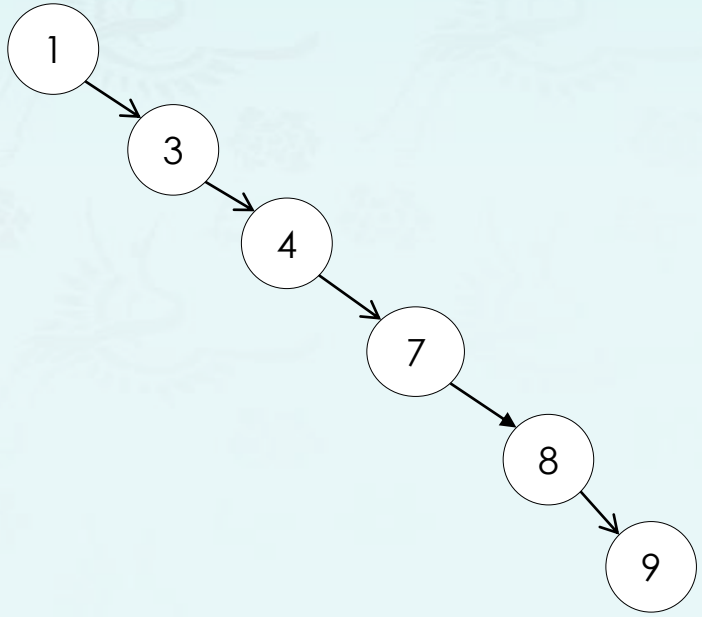
best case



typical case

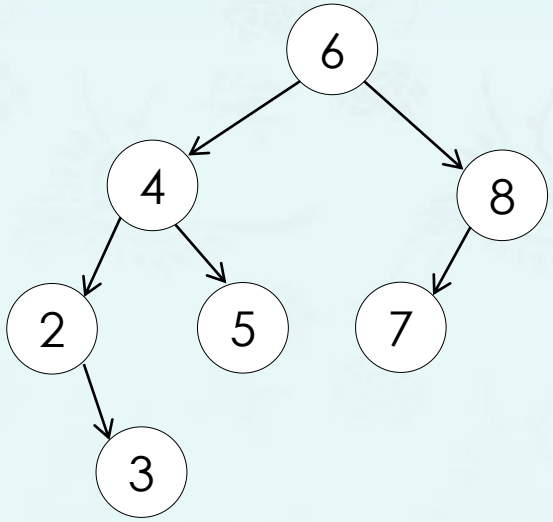


worst



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 …)

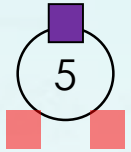
```
bool containsIteration(tree node, int key) {
    if (node == nullptr) return false;
    while (node) {
        if (key == node->key) return true;
        if (key  < node->key)
            node = node->left;
        else
            node = node->right;
    }
    return false;
}
```

```
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);
```

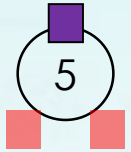



```
int main() {
  tree root = nullptr;
  int key = 5;
  ...
  grow(root, key);
  cout << root->key << endl;
  ...
} // with bugs
```

Operations: Insert (or grow)

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

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



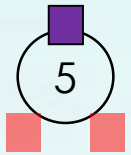

```
int main() {
  tree root = nullptr;
  int key = 5;
  ...
  root = grow(root, key);
  cout << root->key << endl;
  ...
}
```

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
```

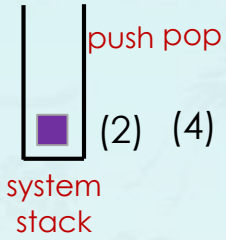
```
grow(node,key);
```




```
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

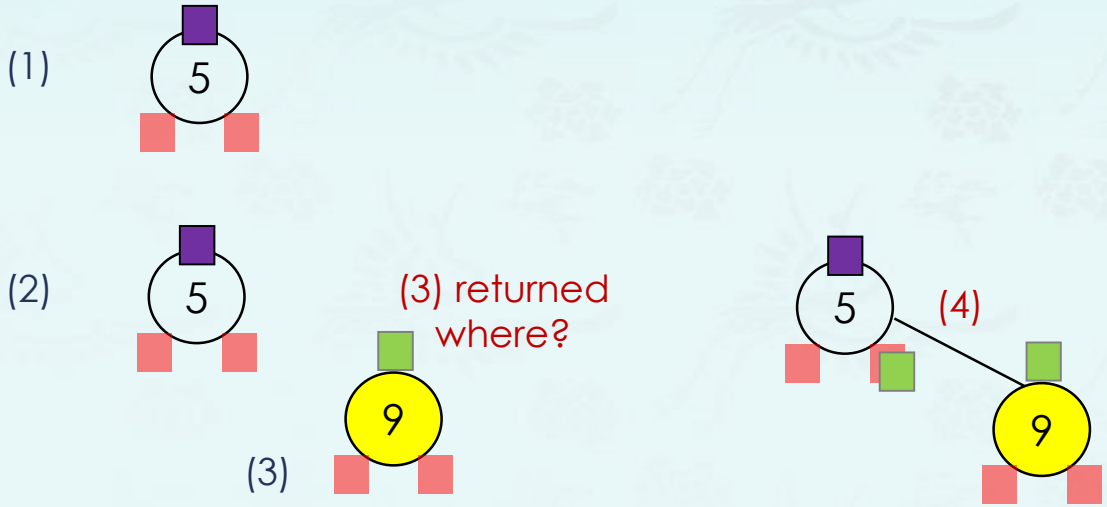
pop after call

```
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
```

(1) (2) (3) (4)

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

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



(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);
  ...
}
```

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;
}
```

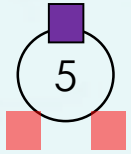
(4) `node->right = grow(node->right, key);`

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

`grow(node,key);`

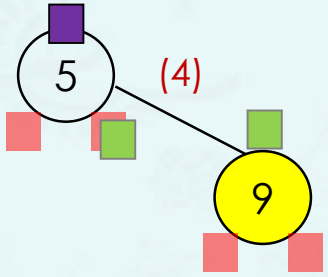
(1)

(2)



(3) returned where?

(3)

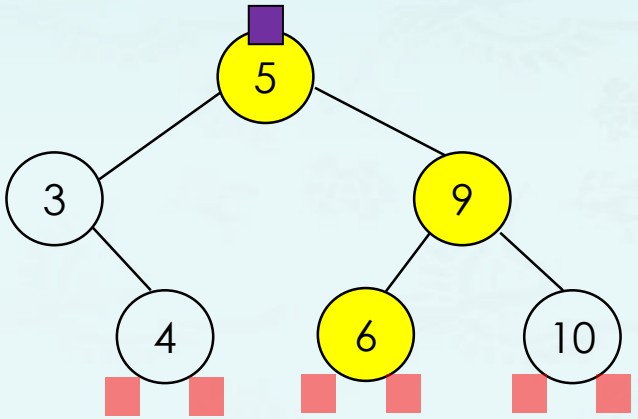


Operations: Insert (or grow)

- 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)
```



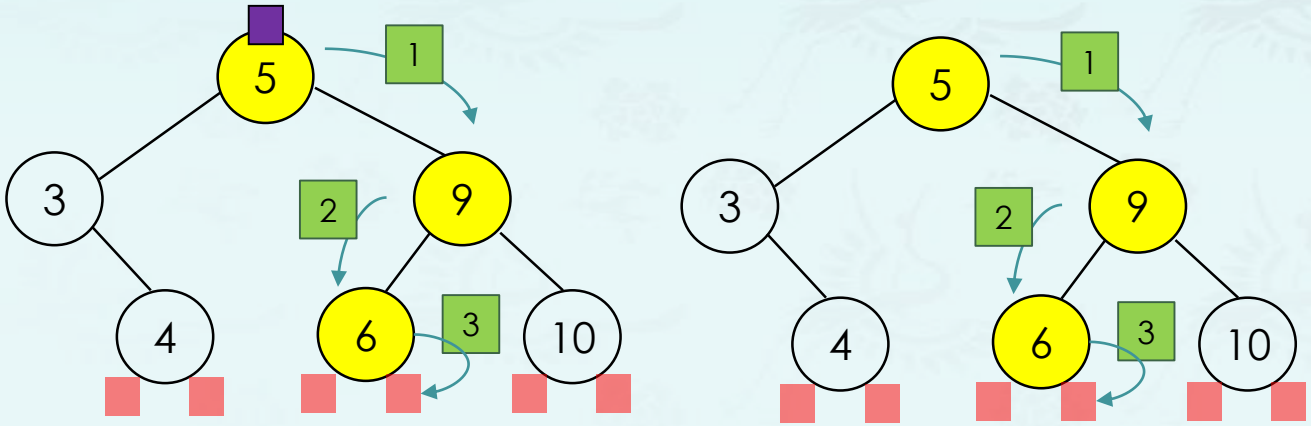
Operations: Insert (or grow)

- 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.

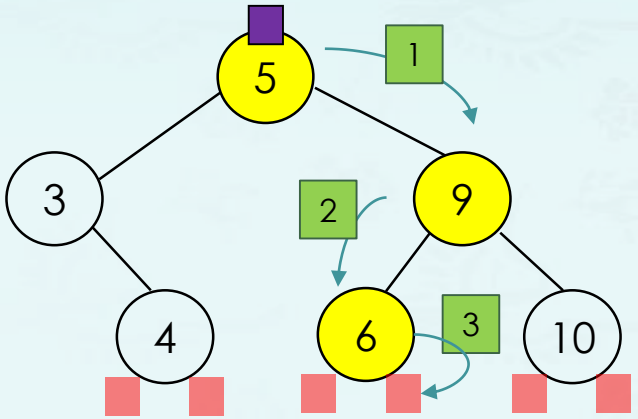
Operations: Insert (or grow)

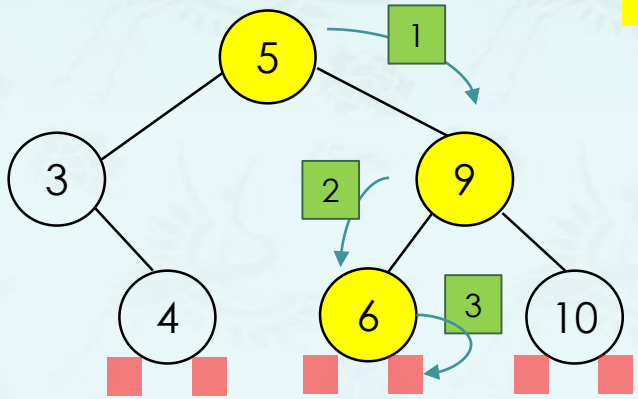
- grow(node, k) - Insert a node with k
  - **Step 1**: If the tree is empty, return a new node(k).
  - **Step 2**: Pretending to search for k in BST, until locating a nullptr.
  - **Step 3**: create a new node(k) and link it.

```
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.

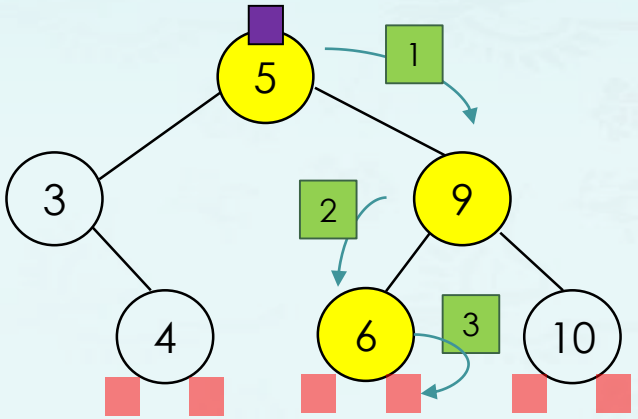
Operations: Insert (or grow)

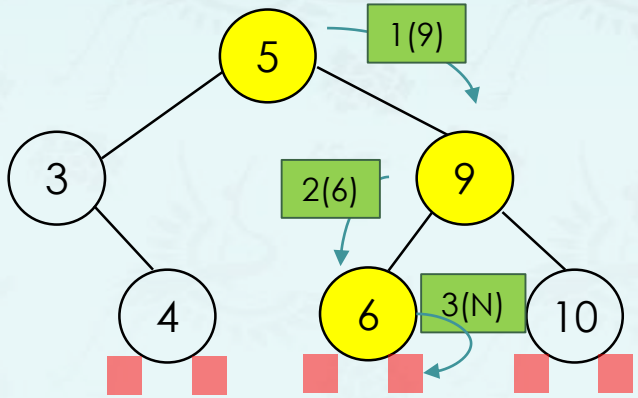
- grow(node, k) - Insert a node with k
  - **Step 1**: If the tree is empty, return a new node(k).
  - **Step 2**: Pretending to search for k in BST, until locating a nullptr.
  - **Step 3**: create a new node(k) and link it.

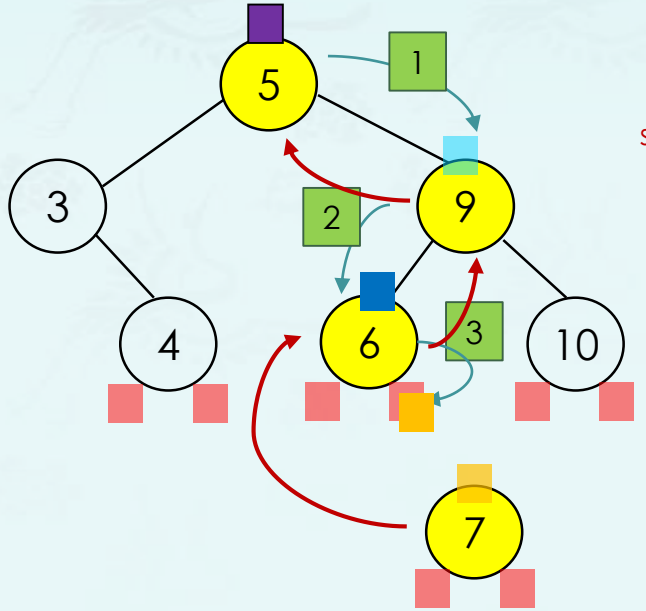
```
tree grow(tree node, int key) {
  if (node == nullptr) return new tree(key);
  if (key < node->key)
    node->left  = grow(node->left, key);
  else if (key > node->key)
    node->right = grow(node->right, key);
  return node;
}
```

`grow(node,7)`

The highlight nodes are compared with key 7.







push pop

system stack

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

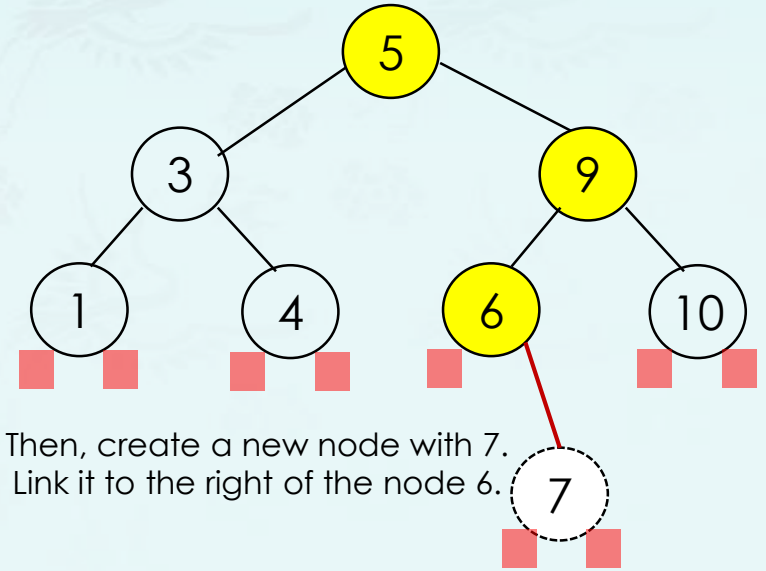
= new tree(7)

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?



Data Structures
Chapter 5 Tree

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