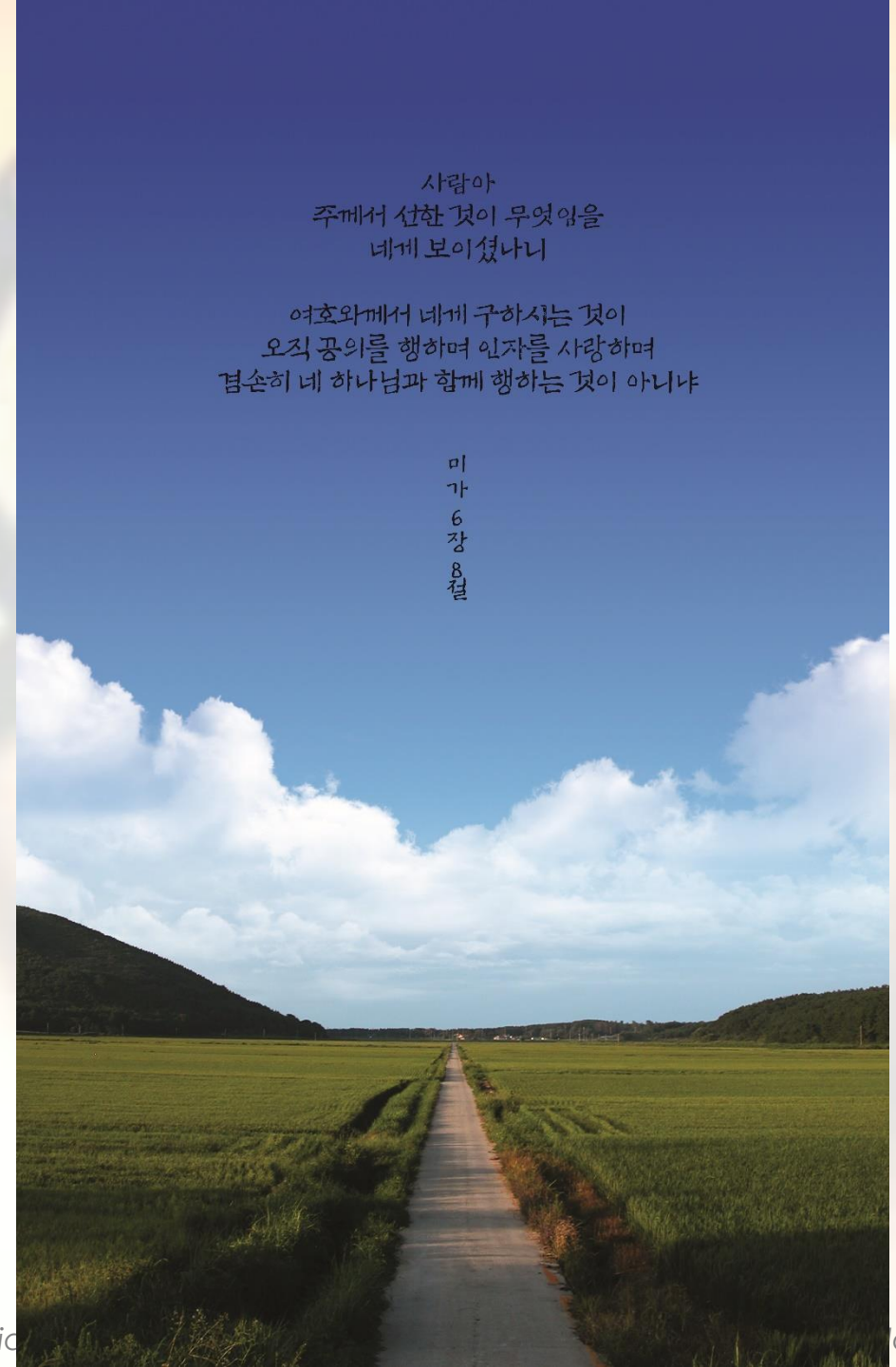
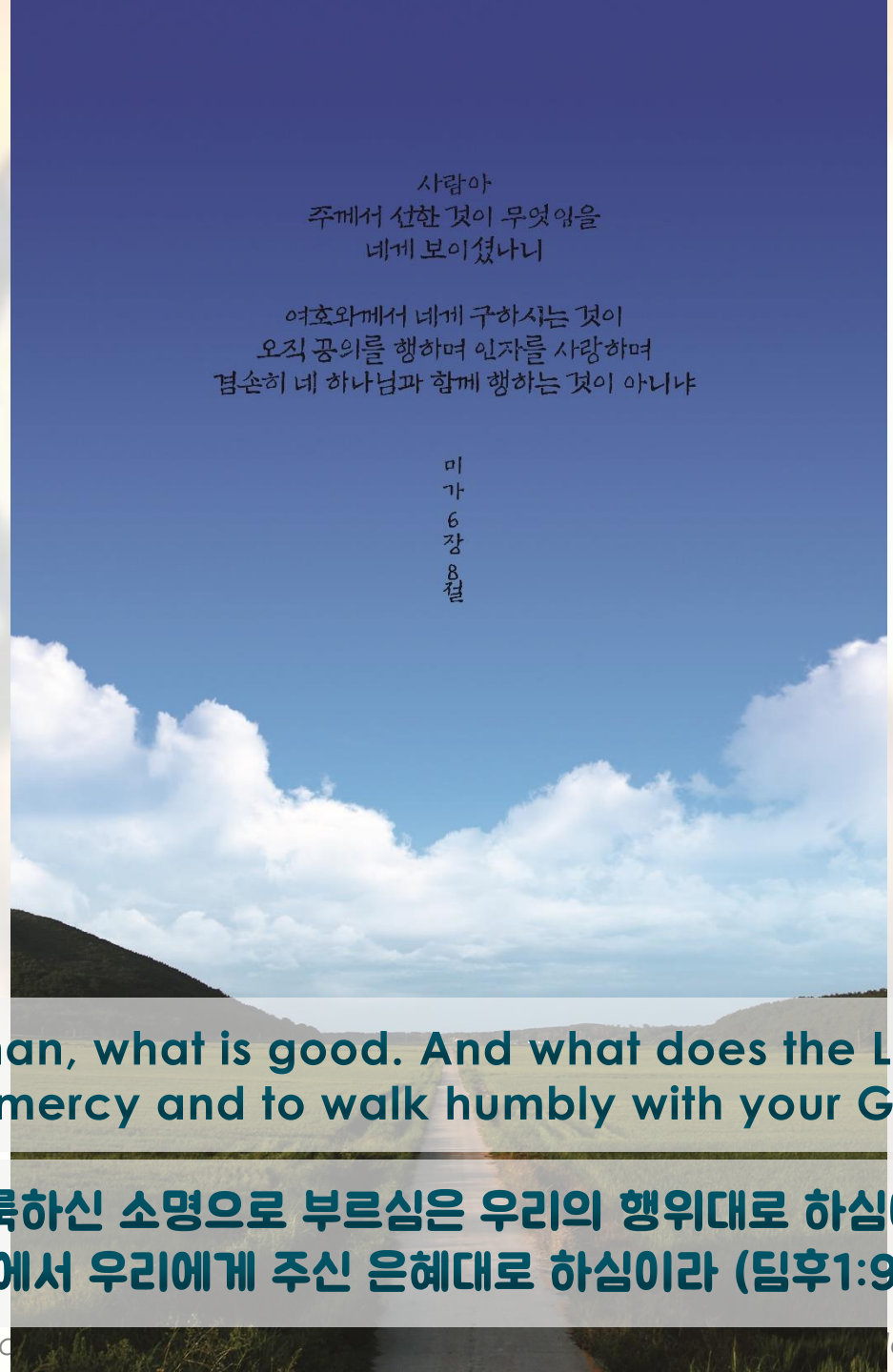
Data Structures
Chapter 5 Tree

1. introduction
2. Binary tree
   - Definition and Properties
   - Traversal
   - **Coding**
3. Binary search tree
4. Tree balancing

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


사람아
주께서 선한 것이 무엇임을
네게 보이셨나니
여호와께서 네게 구하시는 것이
오직 공의를 행하며 인자를 사랑하며
겸손히 네 하나님과 함께 행하는 것이 아니냐
미
가
6
장
8
절

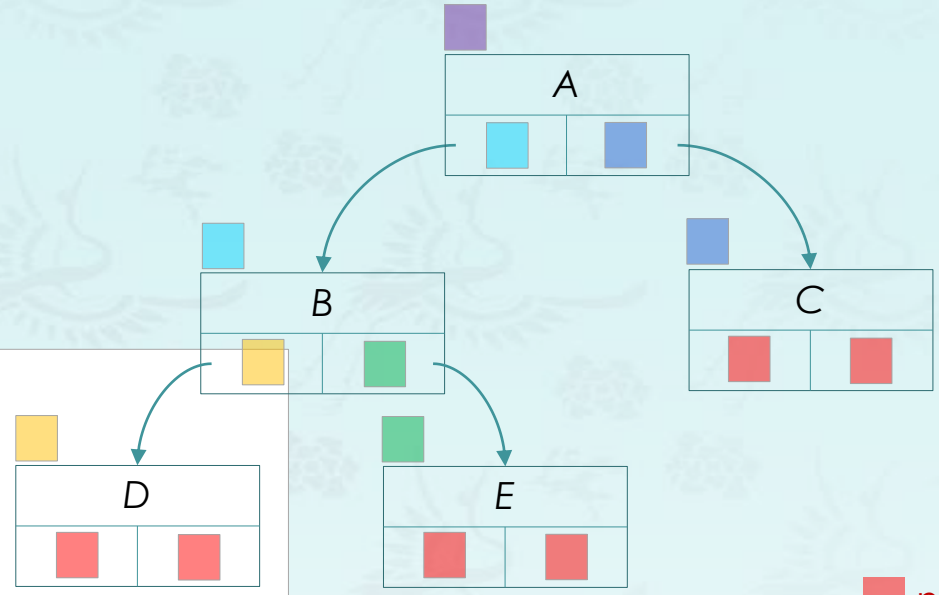


He has showed you, O man, what is good. And what does the LORD require of you? To act justly and to love mercy and to walk humbly with your God. Micah 6:8

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

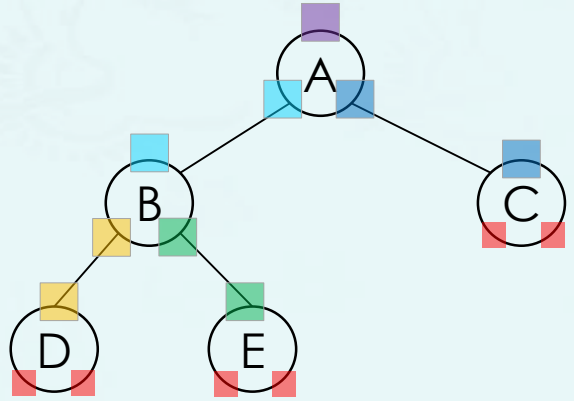
Recursion & Tree Structure

```
struct TreeNode{
    int       key;
    TreeNode* left;
    TreeNode* right;
};
using tree = TreeNode*;
```



null link or empty node

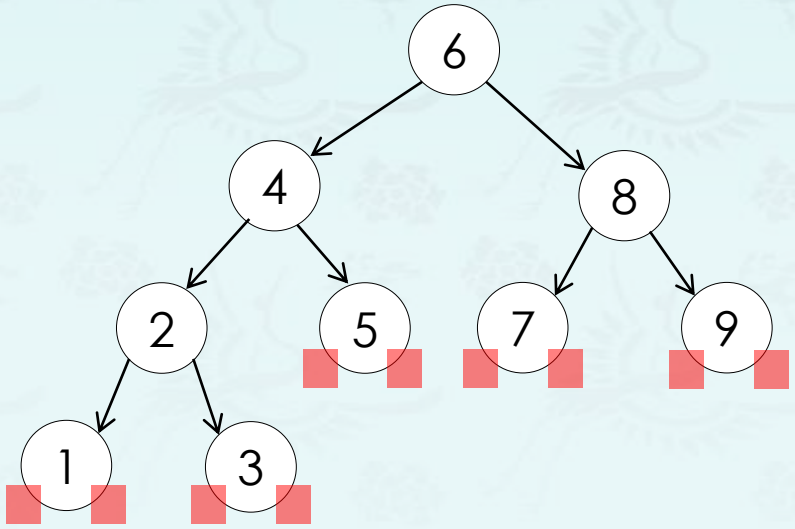
```
struct TreeNode{
  int       key;
  TreeNode* left;
  TreeNode* right;

  TreeNode(int k, TreeNode* l, TreeNode* r) {
    key = k; left = l; right = r;
  }
  TreeNode(int k) : key(k), left(nullptr), right(nullptr) {}

  ~TreeNode(){}
};
using tree = TreeNode*;
```



Operations: inorder()

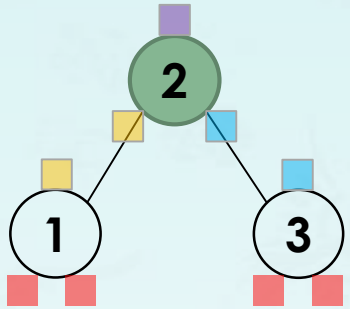
```
// Given a binary tree, its node values in inorder are passed
// back through the argument v which is passed by reference.
void inorder(tree node, vector<int>& v) {
  if (empty(node)) return;

  inorder(node->left, v);
  v.push_back(node->key);
  inorder(node->right, v);
}
```

```
// pinorder() and its result
vec.clear();
inorder(root, vec);
cout << "inorder: ";
for (auto i : vec)
  cout << i << " ";
cout endl;
```



Binary tree traversals

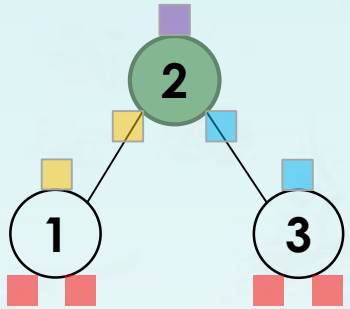
- **Example: Inorder traversal(LVR)**
  - Step 1 – Recursively traverse left subtree.
  - Step 2 – Visit root node. (print or save it.)
  - Step 3 – Recursively traverse right subtree.




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

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

Binary tree traversals

- **Example: Inorder traversal(LVR)**
  - Step 1 – Recursively traverse left subtree.
  - Step 2 – Visit root node. (print or save it.)
  - Step 3 – Recursively traverse right subtree.

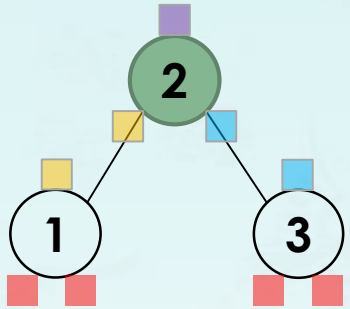



```
void inorder(tree root) {
    if (root == nullptr) return;

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

```
int main() {

}
```

Binary tree traversals

- **Example: Inorder traversal(LVR)**
  - Step 1 – Recursively traverse left subtree.
  - Step 2 – Visit root node. (print or save it.)
  - Step 3 – Recursively traverse right subtree.




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

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

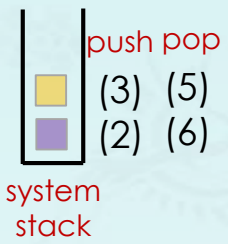
```
int main() {
    tree l = new TreeNode(1);
    tree r = new TreeNode(3);
    tree root = new TreeNode(2, l, r);
    inorder(root);
}
```

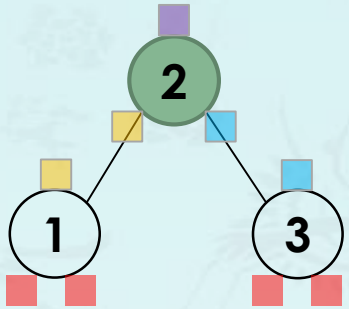
Binary tree traversals - Inorder traversal(LVR)

(1) (2) (3) (5)

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

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

push
pop
push
pop



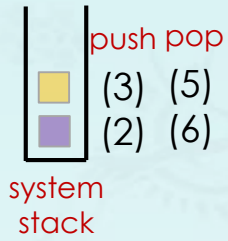


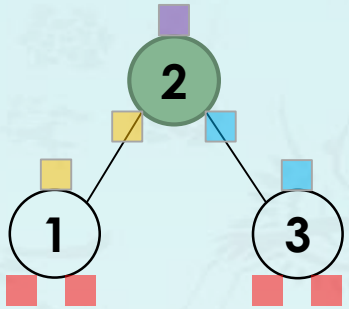
Binary tree traversals - Inorder traversal(LVR)

(1) (2) (3) (5)

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

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

push
pop
push
pop





(1) The initial call inorder(purple, key=2) is made.
(2) The purple calls **inorder( )** with left (or orange).
The purple FC (the initial call) stops and stacked.
(3) The orange calls **inorder( )** with left (or pink)
The orange FC stops and stacked.
The pink FC returns immediately since it is null.
The pink FC is over 1st time. Then, stack pops.
(4) The orange pops from stack & continues where it left.
`cout << root->key;` ⇨ 1
(5) The orange calls **inorder( )** with right(or pink).
The orange FC stops and stacked. ← how many calls in stack?
The pink FC returns immediately since it is null.
The orange FC finishes its job here. Stack pops.
(6) The purple pops from stack & continues where it left.
`cout << root->key;` ⇨ 2 ← how many calls in stack?

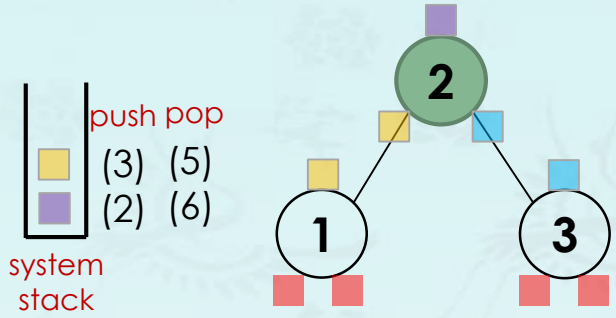
❖ FC stands for function call.

Binary tree traversals - Inorder traversal(LVR)

(1) (2) (3) (5)

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

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

push
pop
push
pop



(7) (8) (10)

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

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

(1) The initial call inorder(purple, key=2) is made.
(2) The purple calls **inorder( )** with left (or orange).
The purple FC (the initial call) stops and stacked.
(3) The orange calls **inorder( )** with left (or pink)
The orange FC stops and stacked.
The pink FC returns immediately since it is null.
The pink FC is over 1st time. Then, stack pops.
(4) The orange pops from stack & continues where it left.
`cout << root->key;` ⇨ 1
(5) The orange calls **inorder( )** with right(or pink).
The orange FC stops and stacked.
The pink FC returns immediately since it is null.
The orange FC finishes its job here. Stack pops.
(6) The purple pops from stack & continues where it left.
`cout << root->key;` ⇨ 2

(7) The purple call **inorder( )** with right(or blue).
The purple FC stops and stacked.
(8) The blue calls **inorder( )** with left (or pink)
The blue FC stops and stacked.
The pink FC returns immediately since it is null.
The pink FC is over. Then, stack pops.
(9) The blue pops from stack & continues where it left.
`cout << root->key;` ⇨ 3
(10) The blue calls **inorder( )** with right(or pink).
The blue FC stops and stacked.
The pink FC returns immediately since it is null.
The blue FC finishes its job here. Stack pops.
(11) The purple pops from stack & continues where it left.
The purple finishes its job and returns to the caller(main).

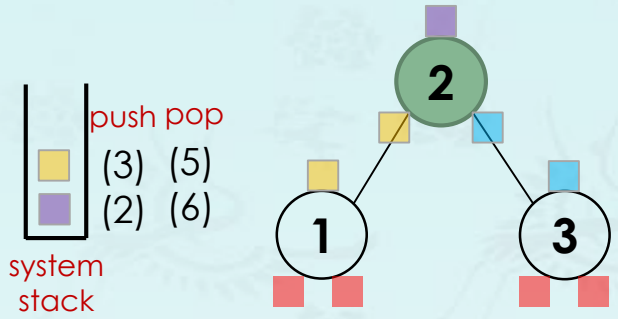
❖ FC stands for function call.

Binary tree traversals - Inorder traversal(LVR)

(1) (2) (3) (5)

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

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

push
pop
push
pop



(7) (8) (10)

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

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

(1) The initial call inorder(purple, key=2) is made.
(2) The purple calls **inorder( )** with left (or orange).
The purple FC (the initial call) stops and stacked.
(3) The orange c
The orange
The pink FC
The pink FC
(4) The orange
cout << roo
(5) The orange
The orange FC stops and stacked.
The pink FC returns immediately since it is null.
The orange FC finishes its job here. Stack pops.
(6) The purple pops from stack & continues where it left.
cout << root->key; ⇨ 2

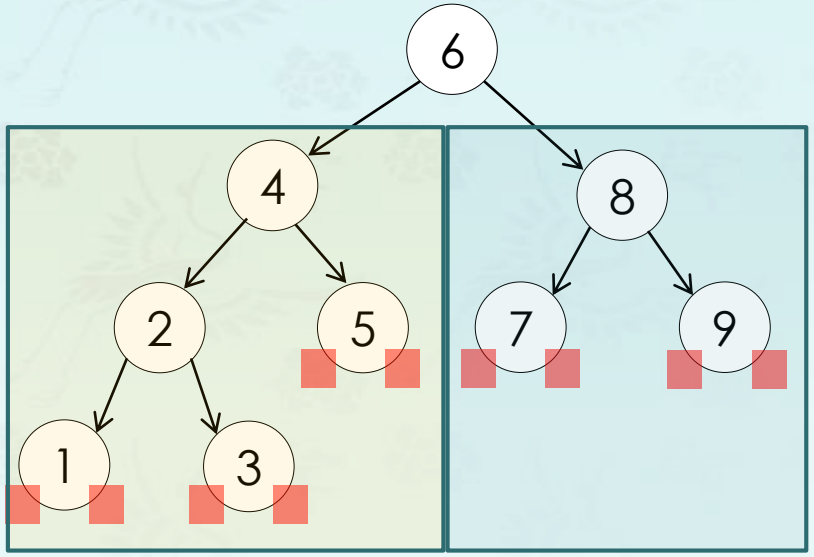
- **The final output:**
- **The number of times of inorder() invoked:**
- **The number of times of the first line return executed:**
- **The number of times of the hidden return executed:**
- **List root's keys passed as an argument and its count:**

(7) The purple call **inorder( )** with right(or blue).
The purple FC stops and stacked.
The blue FC stops and stacked.
The pink FC returns immediately since it is null.
The blue FC finishes its job here. Stack pops.
(11) The purple pops from stack & continues where it left.
The purple finishes its job and returns to the caller(main).

❖ FC stands for function call.

Operations: size()

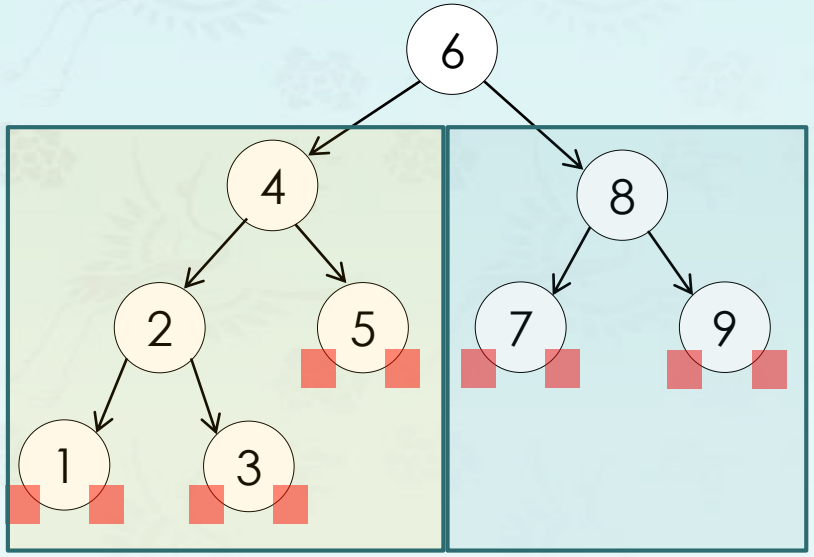
```
// returns the number of nodes in the binary tree
int size(tree node) {
  if (empty(node)) return 0;
  return size(node->left) + size(node->right) + 1;
}
```



Operations: size()

```
// returns the number of nodes in the binary tree
int size(tree node) {
  if (empty(node)) return 0;
  return size(node->left) + size(node->right) + 1;
}
```

- Q1. What is the total number of the function calls to complete with the tree traversal and how many returns from each side of the root 6? _____ _____ _____

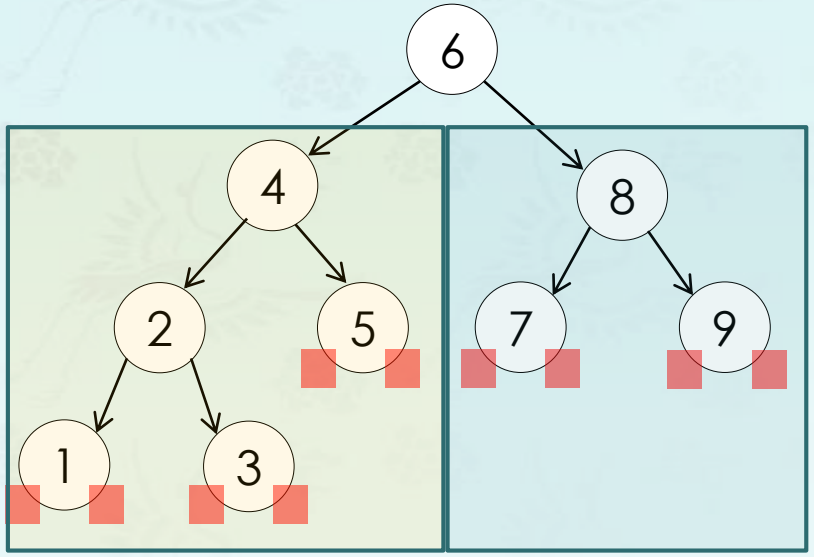



```
// returns the number of nodes in the binary tree
int size(tree node) {
  if (empty(node)) return 0;
  int left  = size(node->left);
  int right = size(node->right);
  return left + right + 1;
} // debug & trace friendly version
```

Operations: size()

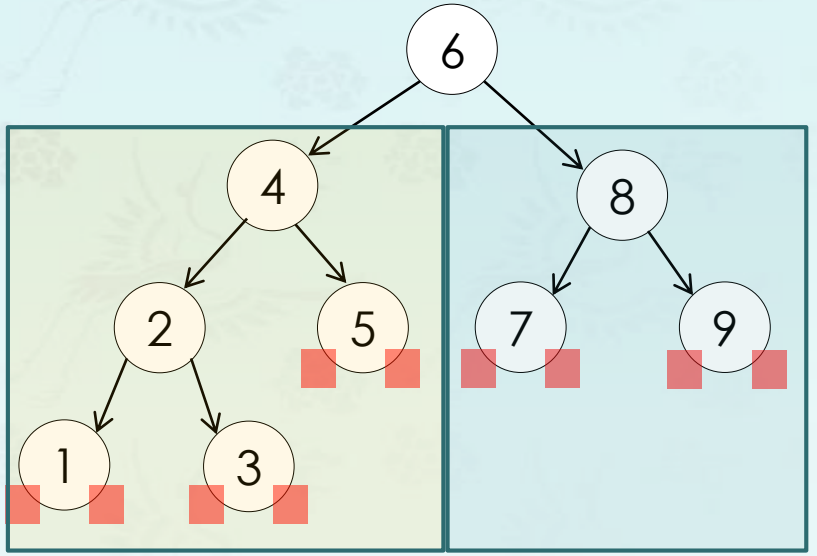
```
// returns the number of nodes in the binary tree
int size(tree node) {
  if (empty(node)) return 0;
  return size(node->left) + size(node->right) + 1;
}
```

- Q1. What is the total number of the function calls to complete with the tree traversal and how many returns from each side of the root 6? _____ _____ _____
- Q2. Which node invokes the last function call?
- Q3. Which node finishes its size function call and returns size = 1 for the first time?



Operations: height()

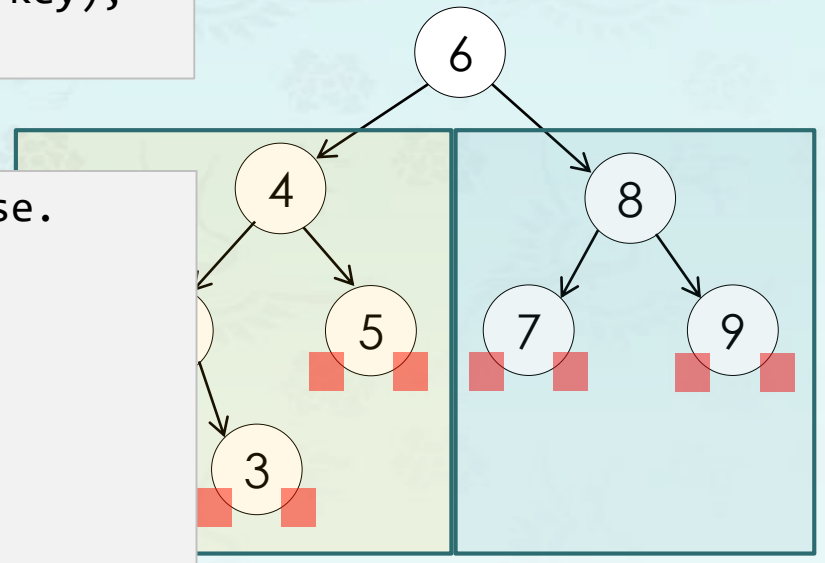
```
// returns the max depth of a tree.
// height = -1 for empty tree, 0 for root only tree
int height(tree node) {
  if (empty(node)) return -1;
  int left  = height(node->left);
  int right = height(node->right);
  return max(left, right) + 1;
}
```



- Q1. What is the total number of the function call to complete with the tree traversal?
- Q2. What is the return value of the 10th and 12th function call?
- Q3. What is the return value of the node 2?

Operations: containsBT(), findBT()

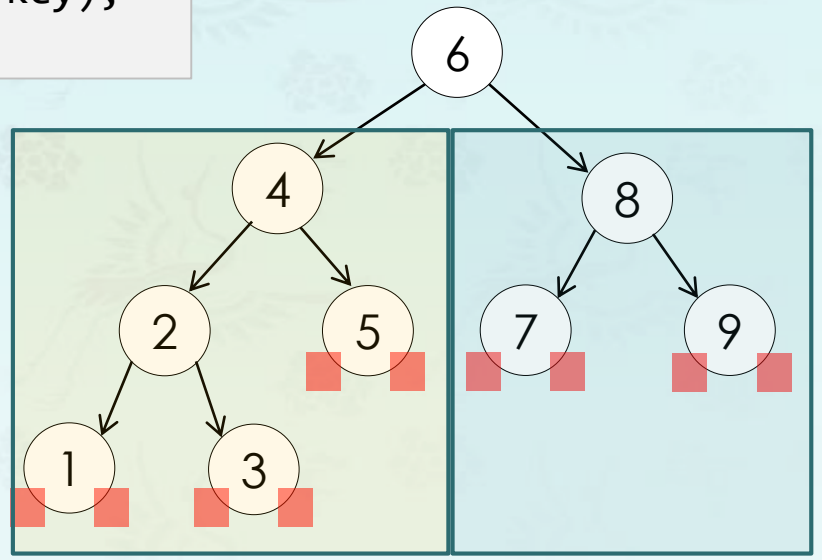
```
// returns true if key is in a given binary tree, false otherwise.
bool containsBT(tree root, int key) {
  if (empty(root)) return false;
  if (key == root->key) return true;

  return containsBT(root->left, key) || containsBT(root->right, key);
}
```




```
// returns true if key is in a given binary tree, false otherwise.
bool containsBT(tree root, int key) {
  if (empty(root)) return false;
  if (key == root->key) return true;

  if (containsBT(root->left, key)
    return true;
  if (containsBT(root->right, key)
    return true;
  return false;
} // debug & trace friendly version
```

Operations: containsBT(), findBT()

```
// returns true if key is in a given binary tree, false otherwise.
bool containsBT(tree root, int key) {
  if (empty(root)) return false;
  if (key == root->key) return true;

  return containsBT(root->left, key) || containsBT(root->right, key);
}
```



- Q1: Which node invokes **`containsBT(root->right, key)`** for the first time?
- Q2: Which node will invoke **`return false`** for the first time?
- Q3: How many function calls are made to reach the node **key=5**?
- Q4: How many function calls still remains in the system stack to finish after key=5 is found and what are they?

Data Structures
Chapter 5 Tree

1. introduction
2. Binary tree
   - Definition and Properties
   - Traversal
   - **Coding II**
3. Binary search tree
4. Tree balancing