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

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

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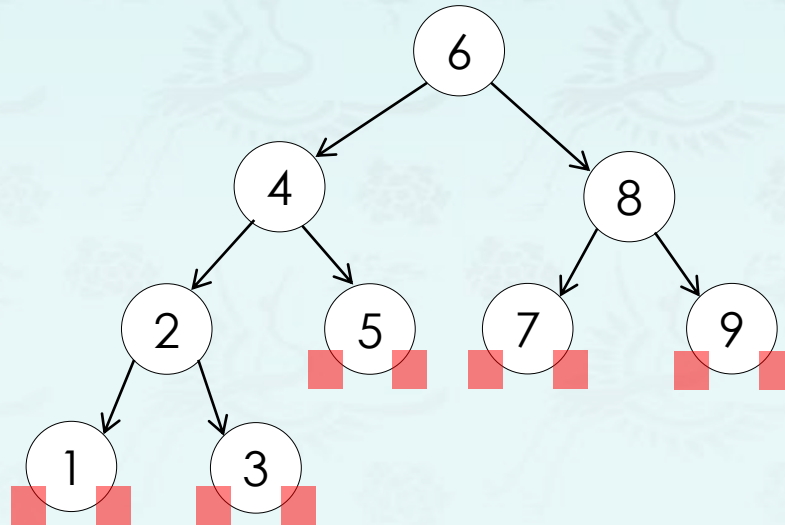
```cpp
struct TreeNode{
  int       key;
  TreeNode* left;
  TreeNode* right;

  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*;
```

null link or
empty node
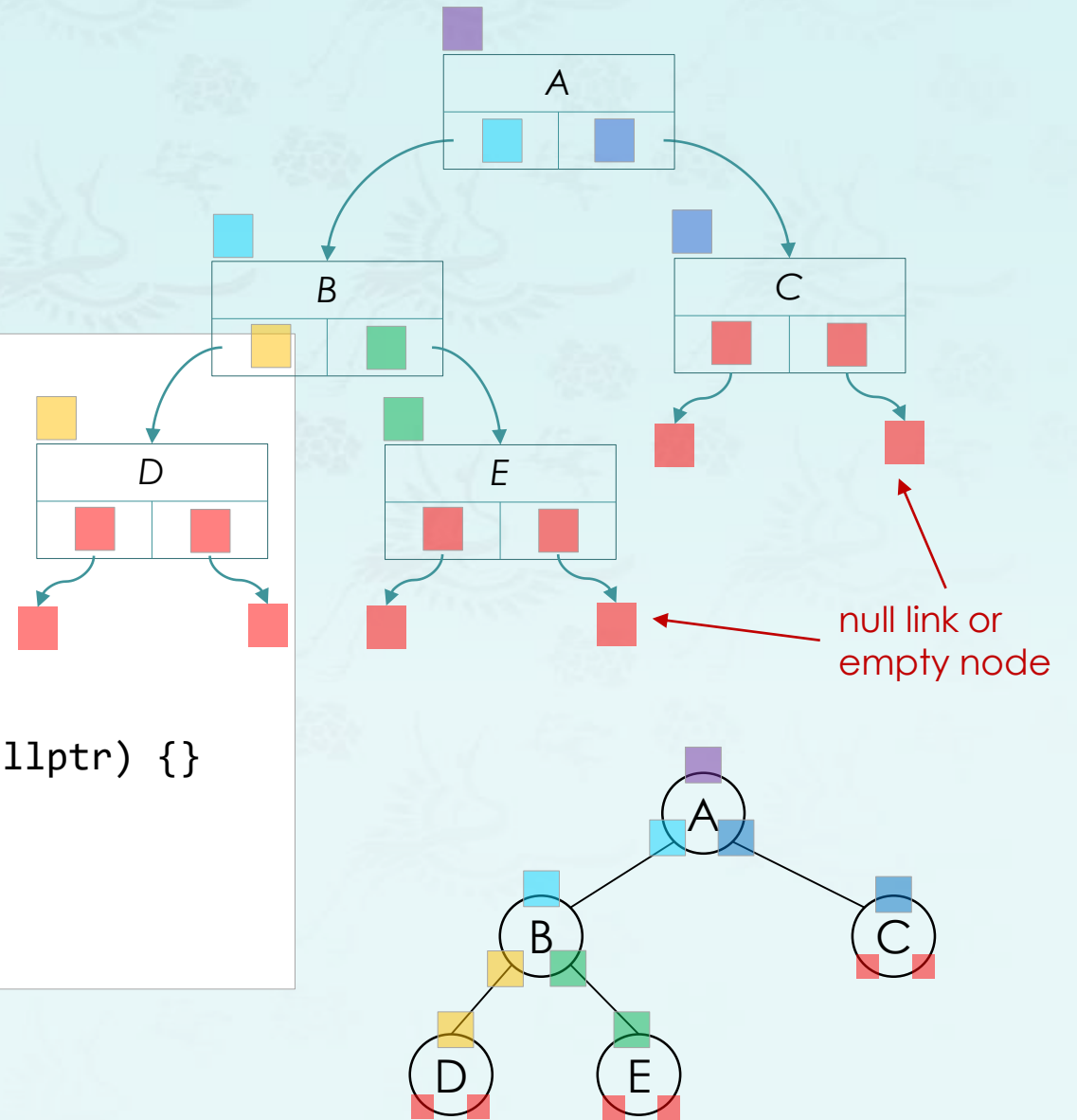
Operations: inorder()

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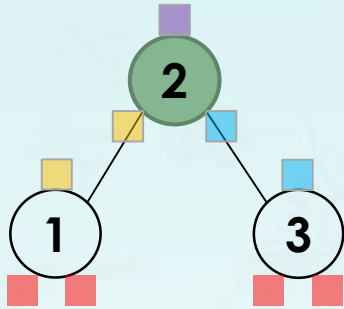
```cpp
// pinorder() and its result
vec.clear();
inorder(root, vec);
cout << "inorder: ";
for (auto i : vec)
  cout << i << " ";
cout endl;
```

Binary tree traversals

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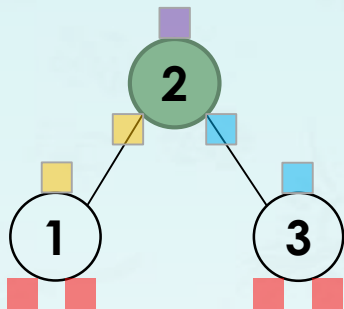 - Step 3 – Recursively traverse right subtree.

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

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

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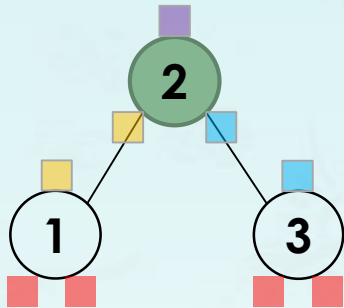
```
void inorder(tree root) {
    if (root == nullptr) return;

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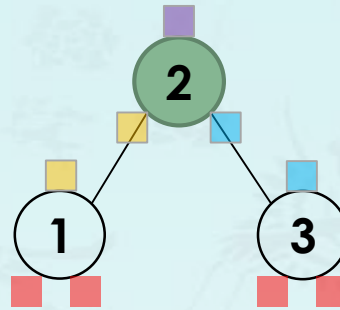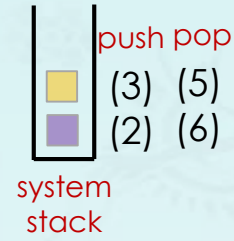
```
int main() {
    tree l = new TreeNode(1);
    tree r = new TreeNode(3);
    tree root = new TreeNode(2, l, r);
    inorder(root);
}
```

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

push pop

(3) (5)
(2) (6)
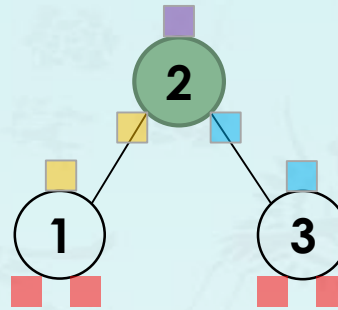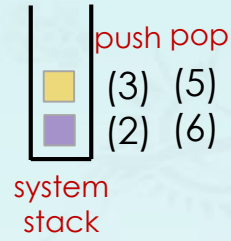
system
stack

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 →

push pop
(3) (5)
(2) (6)

system stack

2

1   3

(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. Stock pops. ▢
(6) The purple pops from stack & continues where it left. ▢
▢ `cout << root->key;` ⇒ 2   ← how many calls in stack?
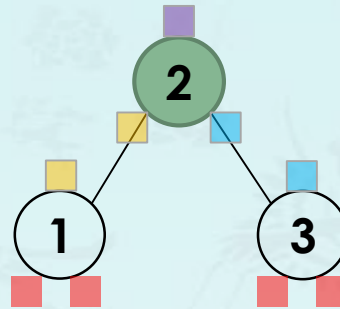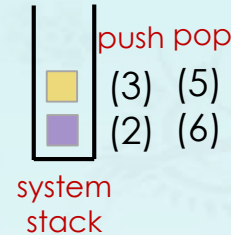
❖ FC stands for function call.

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 →

push pop
(3) (5)
(2) (6)

system stack

2
1   3

push pop
(8) (10)
(7) (11)

system stack

(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. Stock 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. Stock 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.
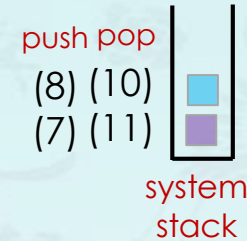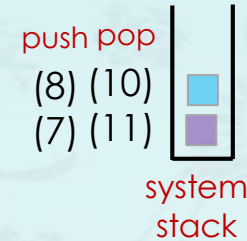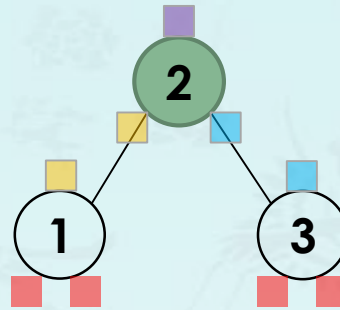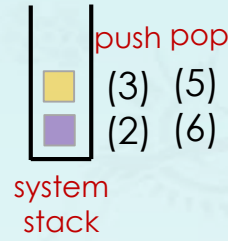
(1) (2) (3) (5)

(7) (8) (10)

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

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

push pop
(3) (5)
(2) (6)

system stack

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

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

push pop
(8) (10)
(7) (11)

system stack

2

1   **3**

(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 F...
   The pink FC r...
   The pink FC i...
(4) The orange p...
   ▢ cout << roo...
(5) The orange c...
   The orange FC stops and stacked.
   The pink FC returns immediately since it is null.
   The orange FC finishes its job here. Stock 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 call **inorder(▢)** with left (or pink).
   ...
   The blue FC stops and stacked.
   The pink FC returns immediately since it is null.
   The blue FC finishes its job here. Stock pops.
(11) The purple pops from stack & continues where it left.
   The purple finishes its job and returns to the caller(main).

- **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:**
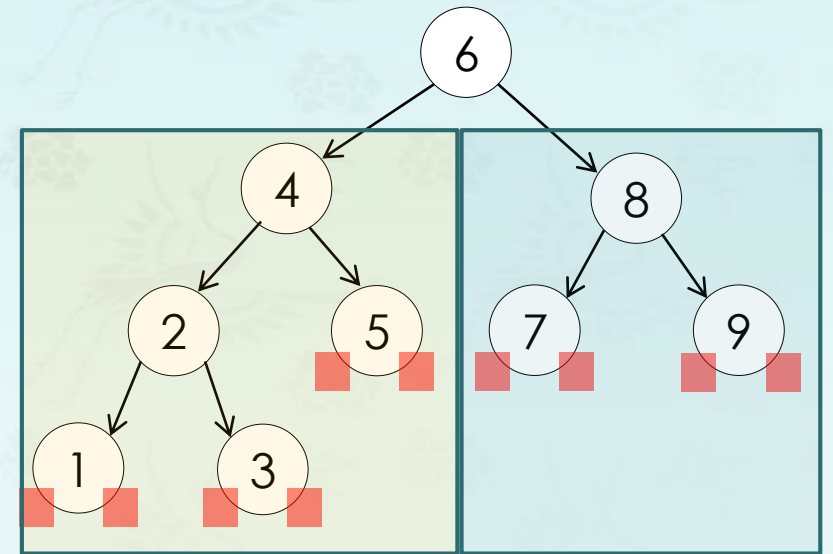- **List root's keys passed as an argument and its count:**

❖ FC stands for function call.

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204, Handong Global University
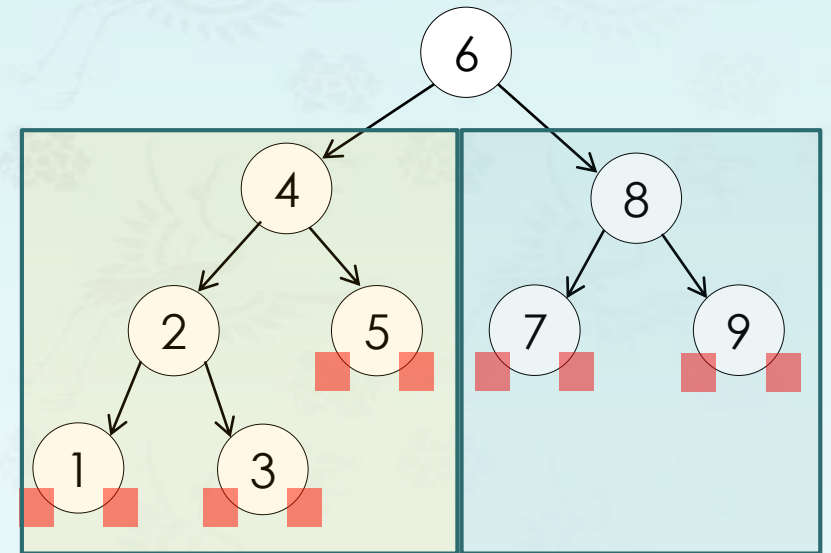
12

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 and how many returns each?

```
// 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
```
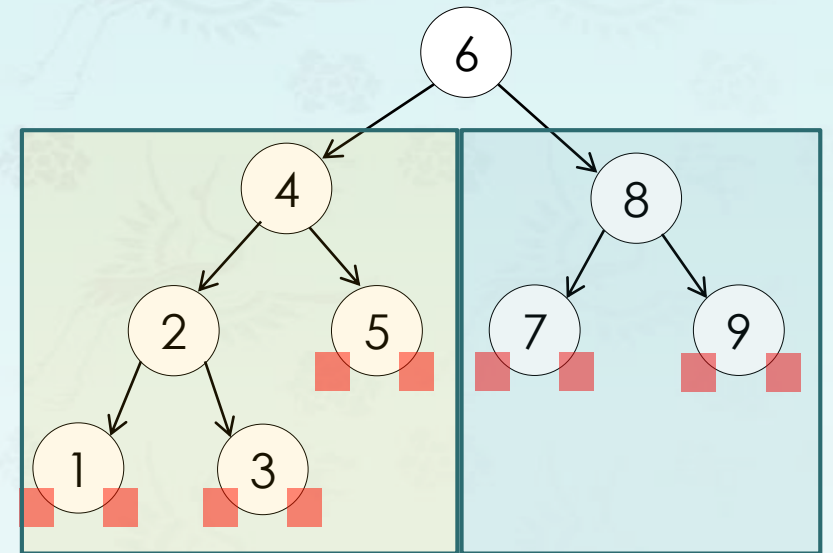
Operations: size()

```
// 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;
}
```
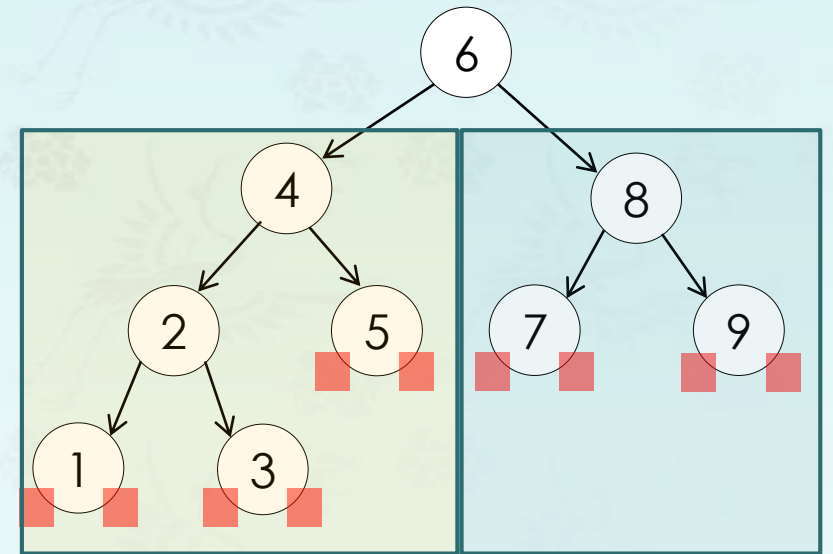
- Q1. What is the total number of the function calls to complete with the tree and how many returns each?

- 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;
}
```
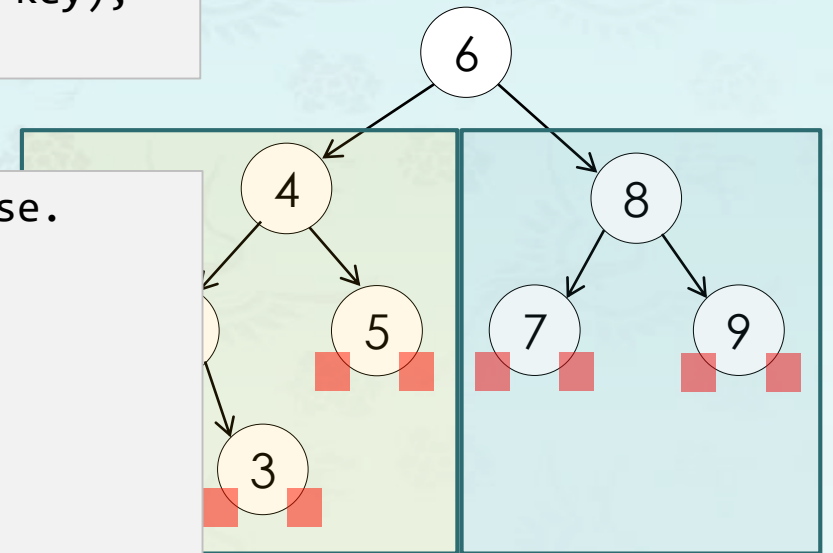
- Q1. What is the total number of the function call to complete with the tree below?

- Q2. What is the return value of the 10th and 12th function call?

- Q3. What is the return value of the node 2?

```
// 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
```

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

17

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?
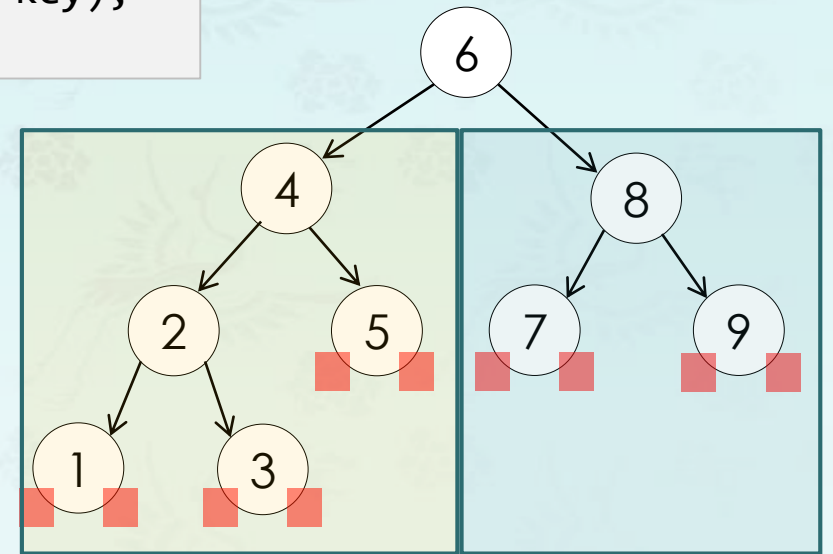
**Data Structures
Chapter 5 Tree**

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

Operations: maximumBT()

```
// Given a binary tree, return the max key in the tree.
tree maximumBT(tree node) {
  if (node == nullptr) return node;
  tree max = node;
  tree x = maximumBT(node->left);
  tree y = maximumBT(node->right);
  if (x->key > max->key) max = x;
  if (y->key > max->key) max = y;
  return max;
} // buggy on purpose
```

- **Hint:**
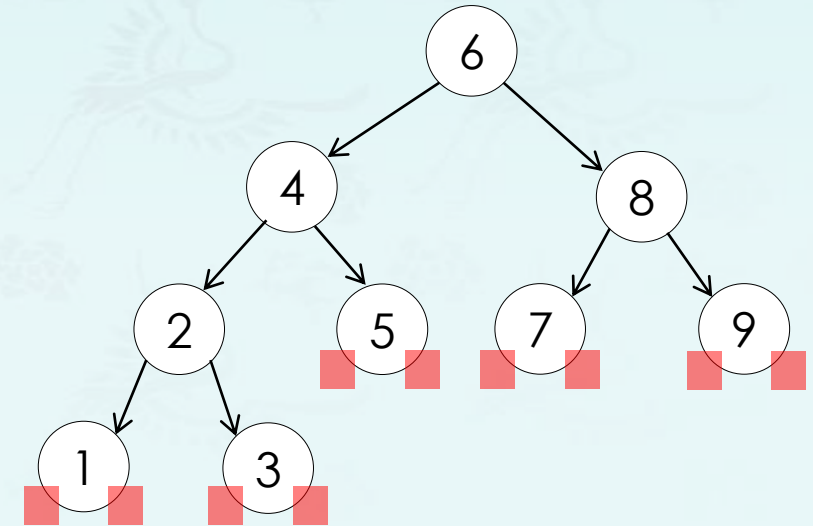 Trace the return value of `maximumBT()` at the leaf.
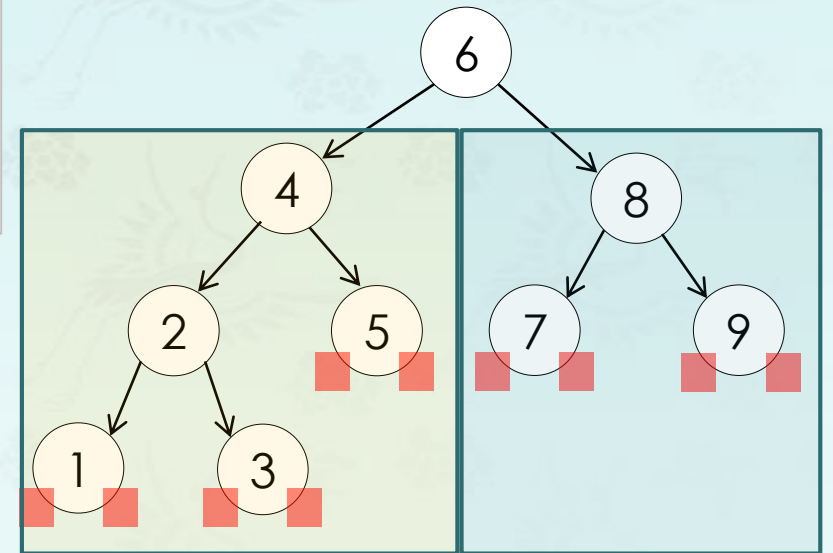
Operations: levelorder()

- This traversal visits every node on a level before going to a lower level. This search is referred to as **breadth-first search** (BFS), as the search tree is broadened as much as possible on each depth before going to the next depth.

- This will require space proportional to the maximum number of nodes at a given depth. This can be as much as the total number of nodes / 2.

Algorithm (Iteration):

- Create empty queue and push root node to it.
- Do the following while the queue is not empty.
  - Pop a node from queue and print/save it.
  - Push left child of popped node to queue if not null.
  - Push right child of popped node to queue if not null.

```
// level order traversal of a given binary tree using iteration.

void levelorder(tree root, vector<int>& vec)
```
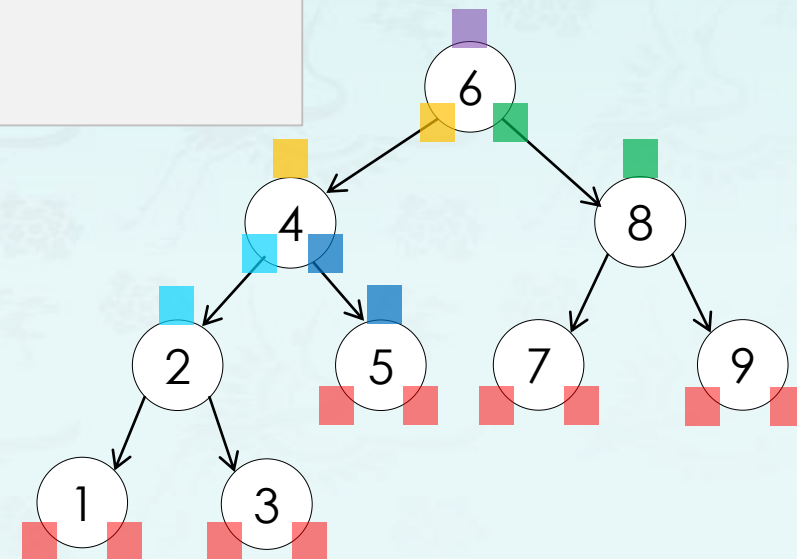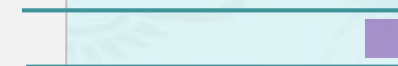- Visit the root. ("visit" means "Current node to process"
  - if it is not null, push it. ■
- while queue is not empty
  1. queue.front() – get the node from the queue
  2. visit the node (save the key in vec).
  3. if its left child is not null, push it to queue.
  4. if its right child is not null, push it to queue.
  5. queue.pop() – remove the node in the queue.

vector

queue

// https://en.wikipedia.org/wiki/Tree_traversal

Operations: levelorder()

```
// level order traversal of a given binary tree using iteration.

void levelorder(tree root, vector<int>& vec)
```
- Visit the root. ("visit" means "Current node to process"
  - if it is not null, push it. 🟪

- while queue is not empty   1st 2nd
  1. queue.front() – get the node from the queue   🟪
  2. visit the node (save the key in vec).   🟪
  3. if its left child is not null, push it to queue. 🟨
  4. if its right child is not null, push it to queue. 🟩
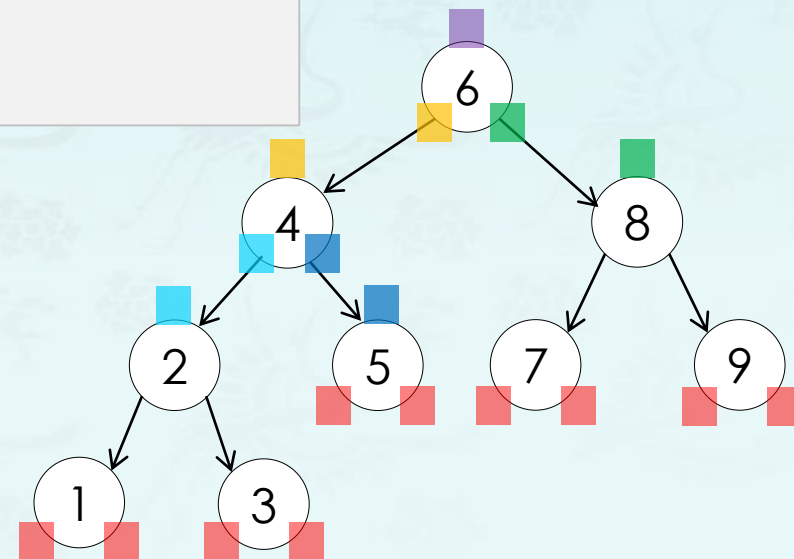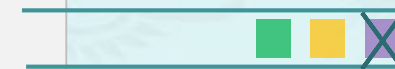 5. queue.pop() – remove the node in the queue.   ✖

vector

| 6 |

queue

// https://en.wikipedia.org/wiki/Tree_traversal

```
// level order traversal of a given binary tree using iteration.

void levelorder(tree root, vector<int>& vec)
```
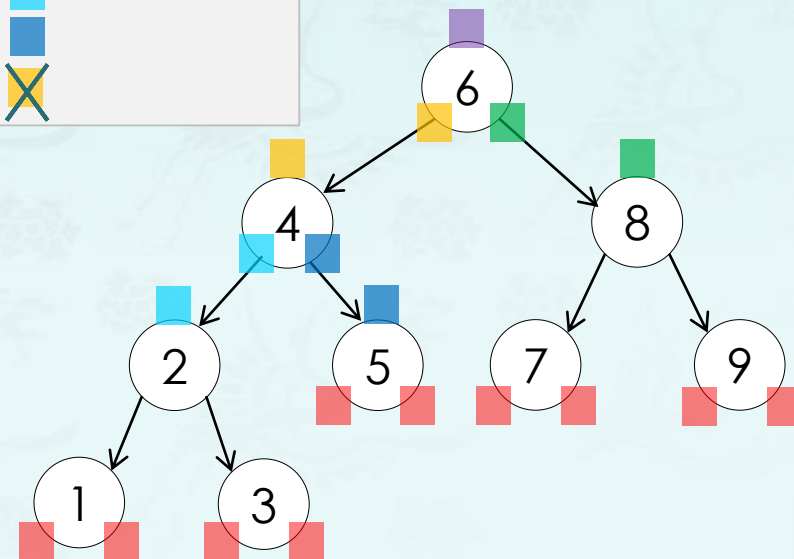- Visit the root. ("visit" means "Current node to process"
  - if it is not null, push it. 🟪
- while queue is not empty
  1. queue.front() – get the node from the queue
  2. visit the node (save the key in vec).
  3. if its left child is not null, push it to queue.
  4. if its right child is not null, push it to queue.
  5. queue.pop() – remove the node in the queue.

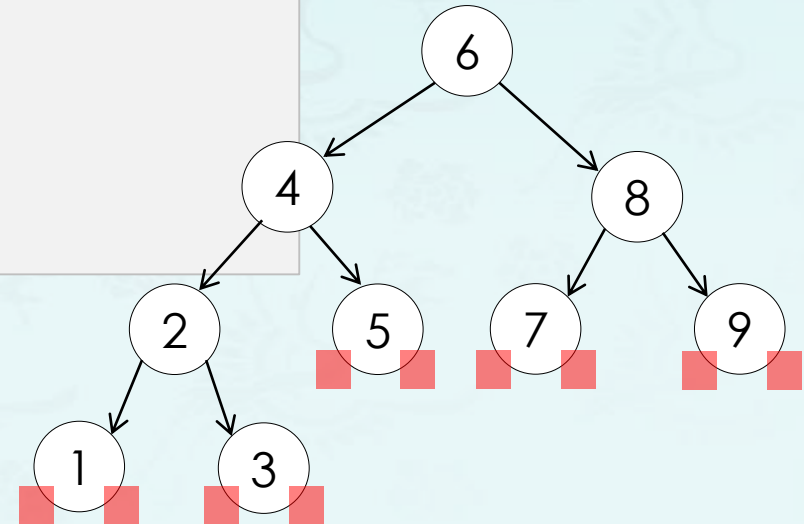// https://en.wikipedia.org/wiki/Tree_traversal

Operations: levelorder()

```cpp
// level order traversal of a given binary tree using iteration.
#include <queue>
#include <vector>

void levelorder(tree root, vector<int>& vec) {
  queue<tree> que;
  if (!root) return;
  que.push(root);
  while ...{


    cout << "your code here\n";


  }
}
```
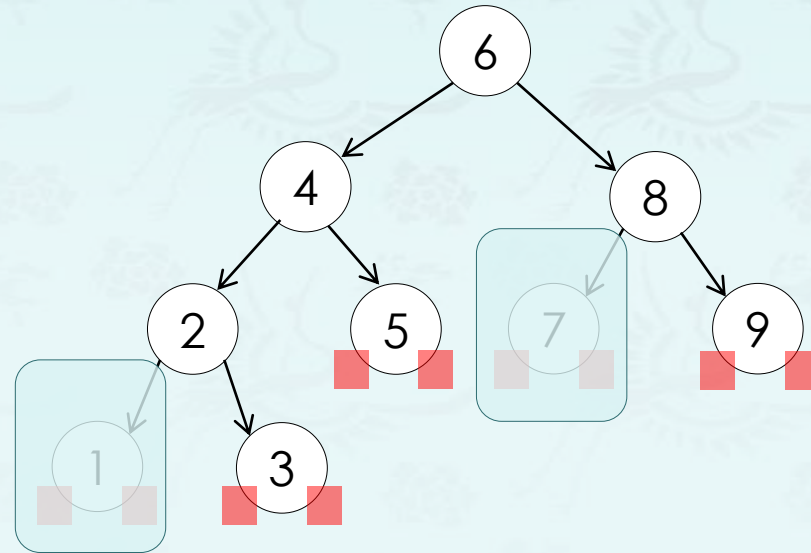

// https://en.wikipedia.org/wiki/Tree_traversal

Operations: Grow a node by level order

```
// inserts a node with the key and returns the root of the binary tree.
// Traversing it in level order, find the first empty node in the tree.
```

Operations: Grow a node by level order

```
// inserts a node with the key and returns the root of the binary tree.
// Traversing it in level order, find the first empty node in the tree.
```

The idea is to do iterative level order traversal of the given tree using **queue**.

First, push the root to the queue.
Then, while the queue is not empty,
    Get the front() node on the queue
    If the left child of the node is empty,
       make new key as left child of the node. – break and return;
    else
       add it to queue to process later since it is not nullptr.
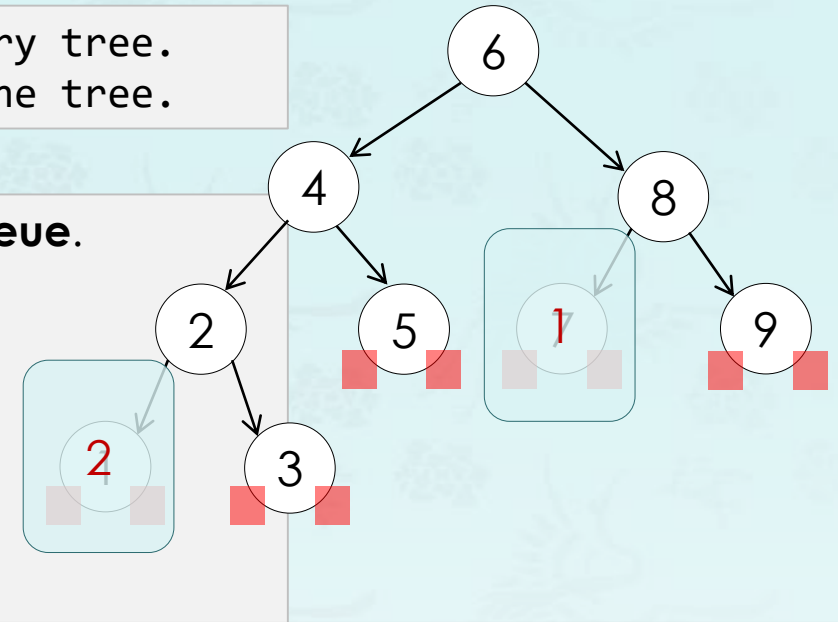   If the right child is empty,
       make new key as right child of the node. – break and return;
    else
       add it to queue to process later since it is not nullptr.
   Make sure that you pop the queue finished.
    Do this until you find a node whose either left or right is empty.

```cpp
tree growBT(tree root, int key) {
  if (root == nullptr)
    return new TreeNode(key);
  queue<tree> q;
  q.push(root);
  while (!q.empty()) {
    // your code here
  }
  return root; // returns the root node
}
```

Operations: Path from root to a node and back in BT

- Given a binary tree with unique keys, return the path from root to a given node x, and back from a node x to the root.
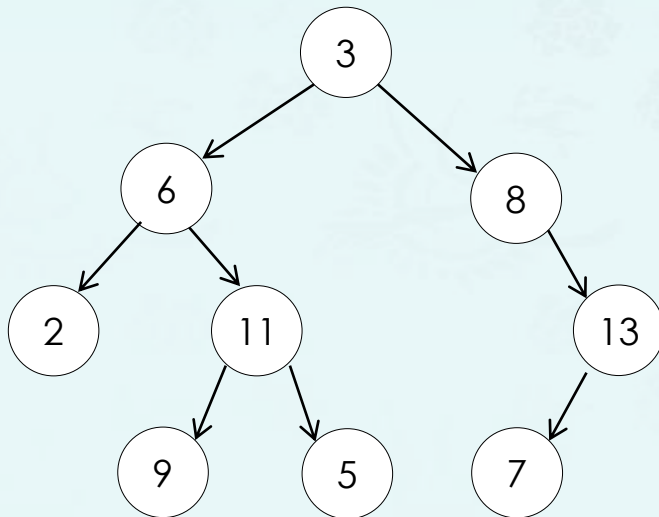
Path from root to a node

```
For example:
2 -> 3, 6, 2
9 -> 3, 6, 11, 9
13 -> 3, 8, 13
11 -> 3, 6, 11
```
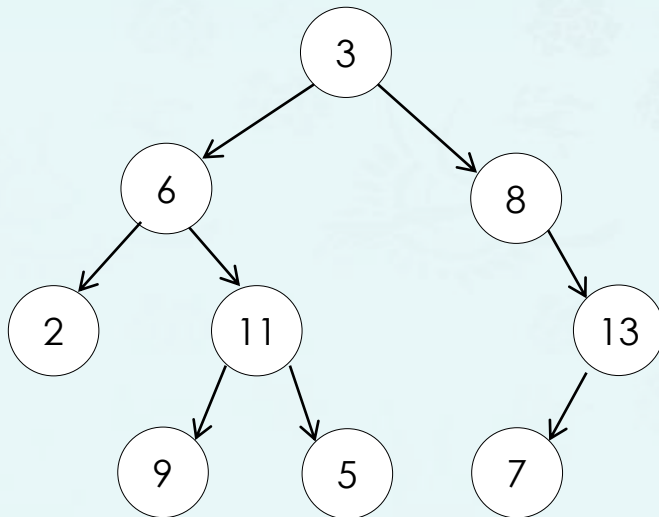
Path from a node to root

```
For example:
2 -> 2, 6, 3
9 -> 9, 11, 6, 3
13 -> 13, 8, 3
11 -> 11, 6, 3
```

- Given a binary tree with unique keys, return the path from root to a given node x.
- Algorithm:
    - If **root = nullptr**, return false. [base case]
    - Push the root's key into **vector**. ← every node goes into the vector until x is found
    - If **root's key = x**, return true. [base case]
    - Recursively, look for x in root's left or right subtree.
        - If it node **x** exists in root's left or right subtree, return true.
        - Else remove root's key from **vector** and return false.

since it is not a part of the path


```
bool findPath(tree root, tree x, vector<int>& path)
```

```
For example:
2 -> 3, 6, 2
9 -> 3, 6, 11, 9
13 -> 3, 8, 13
11 -> 3, 6, 11
```

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

29

Operations: Path from a node to root in BT (Path back)

- Given a binary tree with unique keys, return the path back to root from given node x.
- Algorithm:
  - If **root = nullptr**, return false. [base case]
  - If **root's key = x** or if it node **x** exists in root's left or right subtree during recursive search,
    - Push the root's key into **vector**. (recursive back-trace happens here)
    - Return true.
  - Else
    - return false.
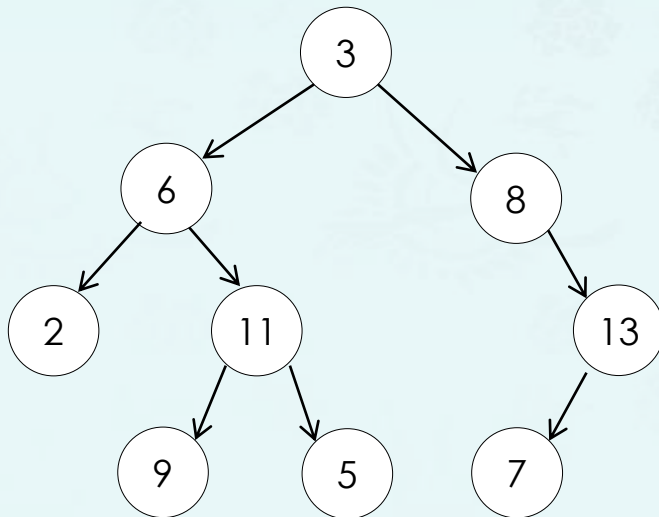

```
bool findPathBack(tree root, tree x, vector<int>& path)
```

```
For example:
2 -> 2, 6, 3
9 -> 9, 11, 6, 3
13 -> 13, 8, 3
11 -> 11, 6, 3
```
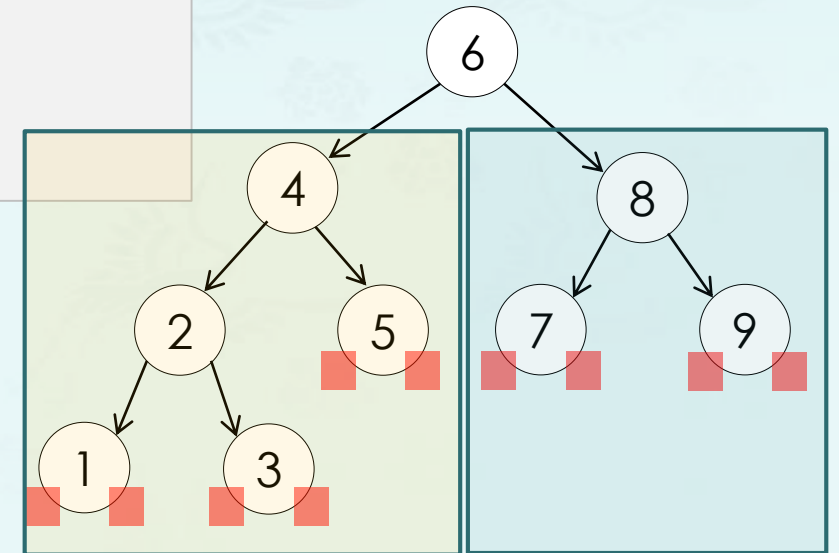
Operations: findPath() & findPathBack()

```
bool findPath(tree node, tree x, vector<int>& path) {
  if (empty(node)) return false;

  cout << "your code here

  return false;
}
```

```
bool findPathBack(tree node, tree x, vector<int>& path) {
  if (empty(node)) return false;

  cout << "your code here

  return false;
}
```

Operations: LCA (Lowest Common Ancestor in BT)

- Find the lowest common ancestor(LCA) of two given nodes, **given in a binary tree.**
  - 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)."
  - Two nodes given, p and q, are different and both values will exist in the binary tree.
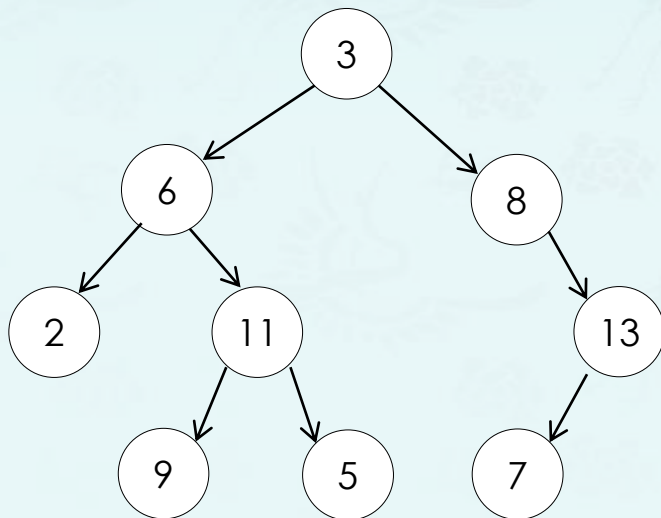

```
For example:
2, 8 -> 3
2, 5 -> 6
9, 5 -> 11
8, 7 -> 8
9, 3 -> 3
```
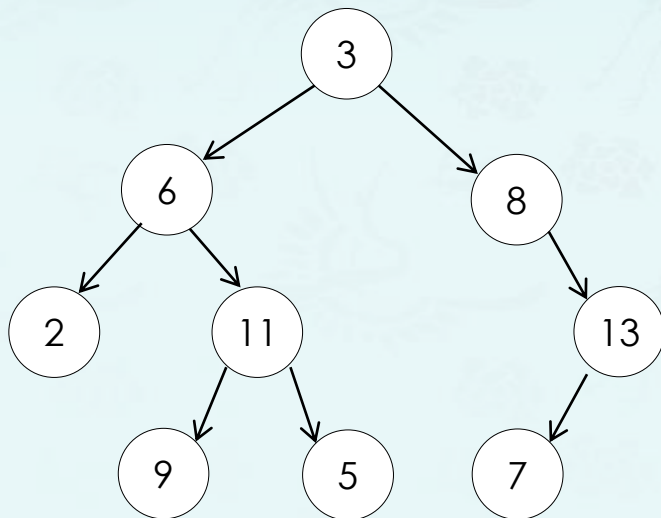
Operations: LCA (Lowest Common Ancestor in BT)

- **Intuition (Iteration):** A brute-force approach is to traverse the tree and get the path to node p and q. Compare the path and return the last match node of the path.
- **Algorithm (Iteration):**
  - Find path from root to p and store it in a vector.
  - Find path from root to q and store it in another vector.
  - Traverse both paths till the values in vector are same. Return the common element just before the mismatch.
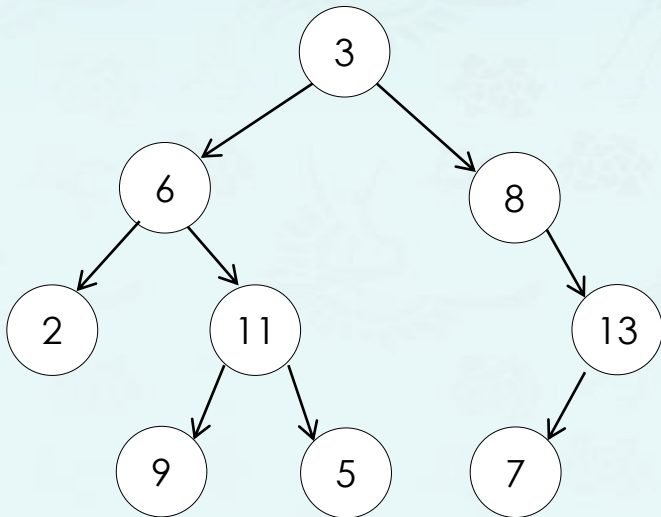


```
For example:
2, 8 -> 3
2, 5 -> 6
9, 5 -> 11
8, 7 -> 8
9, 3 -> 3
```

path to 2:   3 6 2
path to 5:   3 6 11 5

LCA(2, 5) = 6

```
int LCA_BTiteration(tree node, tree p, tree q) {
  if (empty(node)) return false;

  // your code here

  return 6;
}
```


For example:
2, 8 -> 3
2, 5 -> 6
9, 5 -> 11
8, 7 -> 8
9, 3 -> 3

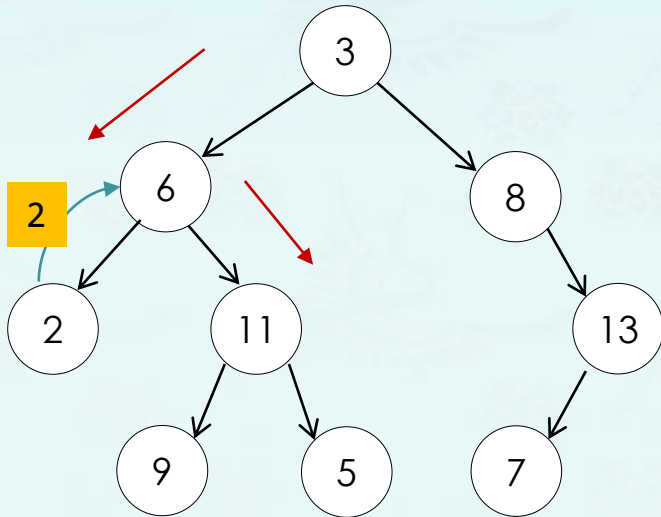path to 2:   3 6 2
path to 5:   3 6 11 5

LCA(2, 5) = 6

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

34

Operations: LCA (Lowest Common Ancestor in BT) Recursion

- **Intuition (Recursion):** Traverse the tree in a depth-first manner. The moment you encounter either of the nodes p or q, return the node. The LCA would then be the node for which both the subtree recursions return a non-NULL node. It can also be the node which itself is one of p or q and for which one of the subtree recursions returns that particular node.

- **Algorithm (Recursion):**
  - Start traversing the tree from the root node.
  - If the current node is nullptr, return nullptr. [base case]
  - If the current node itself is one of p or q, we would return that node. [base case]
  - [recursive case]
    - Search for the left side and search for the right side recursively.
    - If the left or the right subtree returns a non-NULL node, this means one of the two nodes was found below. Return the non-NULL node(s) found.
    - If at any point in the traversal, both the left and right subtree return some node, this means we have found the LCA for the nodes p and q.

- Time Complexity: O(n), Space Complexity: O(n)
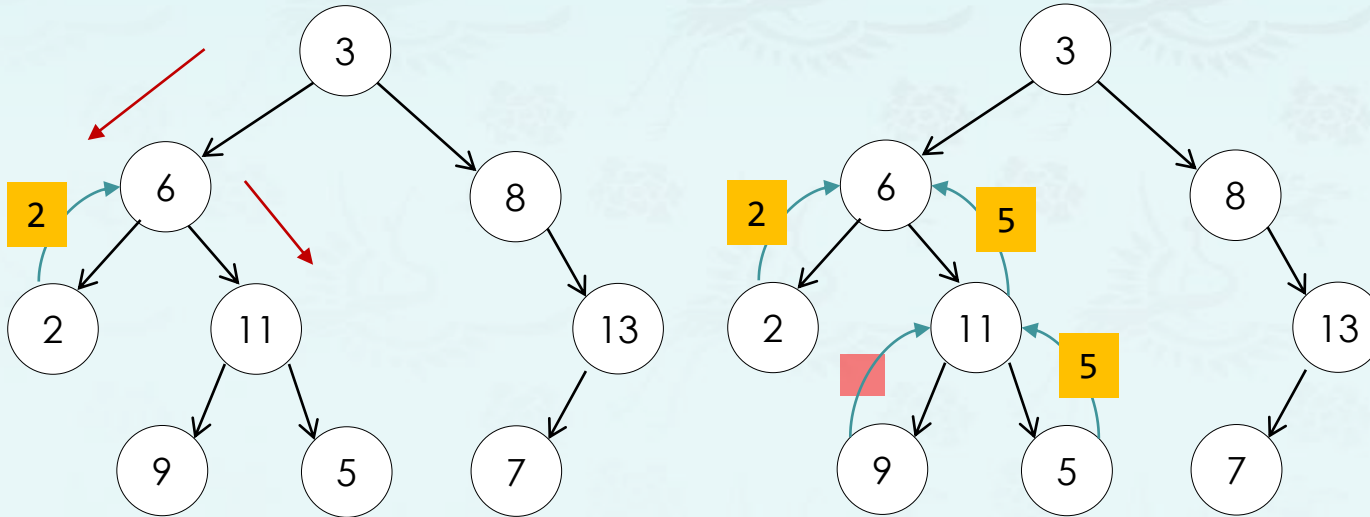
```
tree LCA_BT(tree root, tree p, tree q) {
  if (root == nullptr) return nullptr;
  if (root == p || root == q) return root;
  // recursive cases

  // trace back

  return root;
}
```

```
2, 5 -> 6
```

(2, 5) goes down left subtree of 3
Node 6 got 2 back from left,

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

36

```
tree LCA_BT(tree root, tree p, tree q) {
  if (root == nullptr) return nullptr;
  if (root == p || root == q) return root;
  // recursive cases

  // trace back

  return root;
}
```
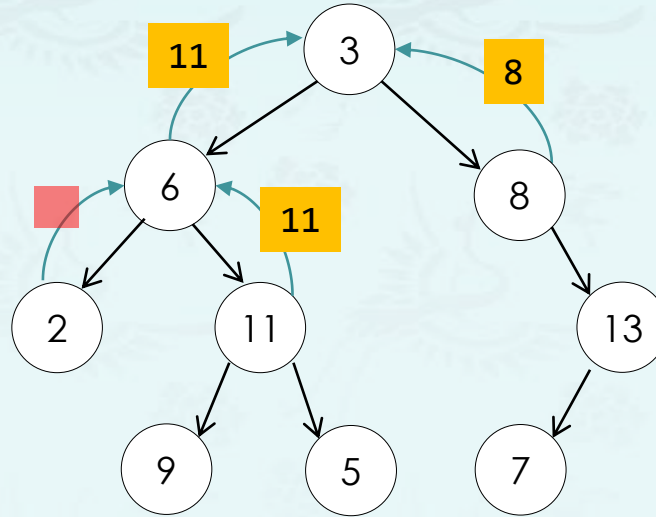
2, 5 -> 6

(2, 5) goes down left subtree of 3
Node 6 got 2 back from left,

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

37

Operations: LCA (Lowest Common Ancestor in BT) Recursion

```
tree LCA_BT(tree root, tree p, tree q) {
  if (root == nullptr) return nullptr;
  if (root == p || root == q) return root;
  // recursive cases

  // trace back

  return root;
}
```
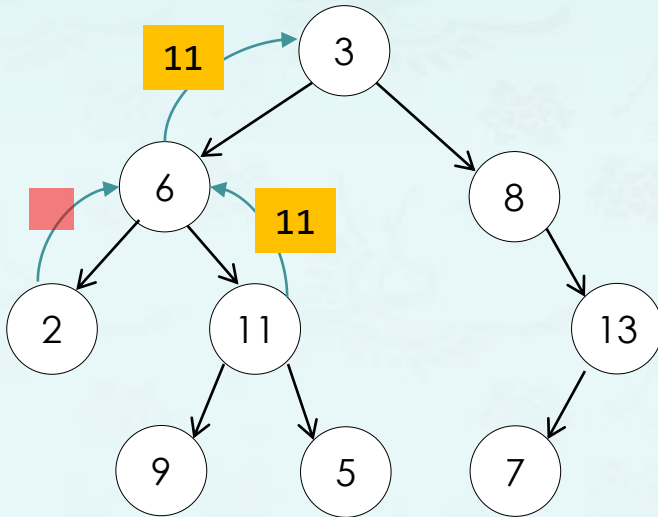
8, 11 -> 3

(8, 11) goes down left subtree of 3
Got 11 back.

```
tree LCA_BT(tree root, tree p, tree q) {
  if (root == nullptr) return nullptr;
  if (root == p || root == q) return root;
  // recursive cases

  // trace back

  return root;
}
```
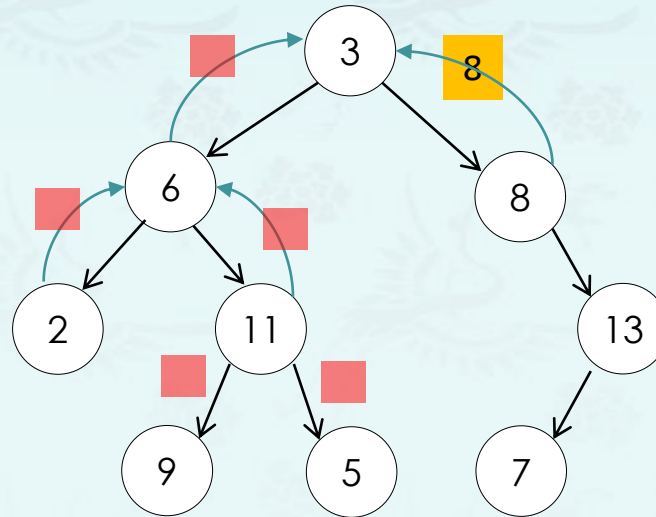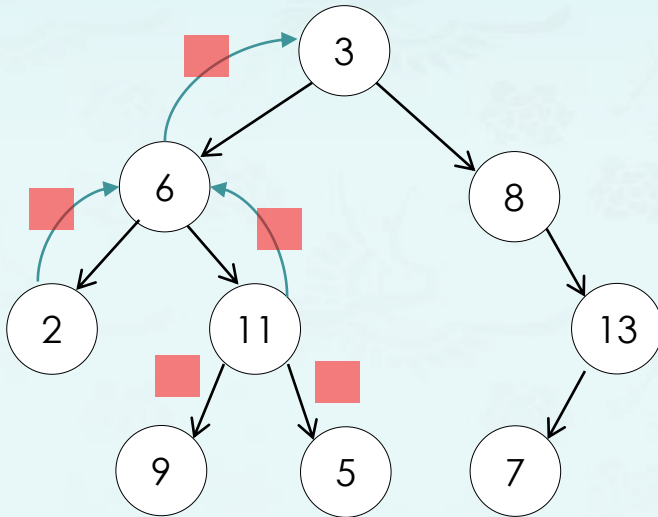
8, 7 -> 8

3 is not either (8, 11) & goes down left
6 is not & goes down left
...

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

39

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