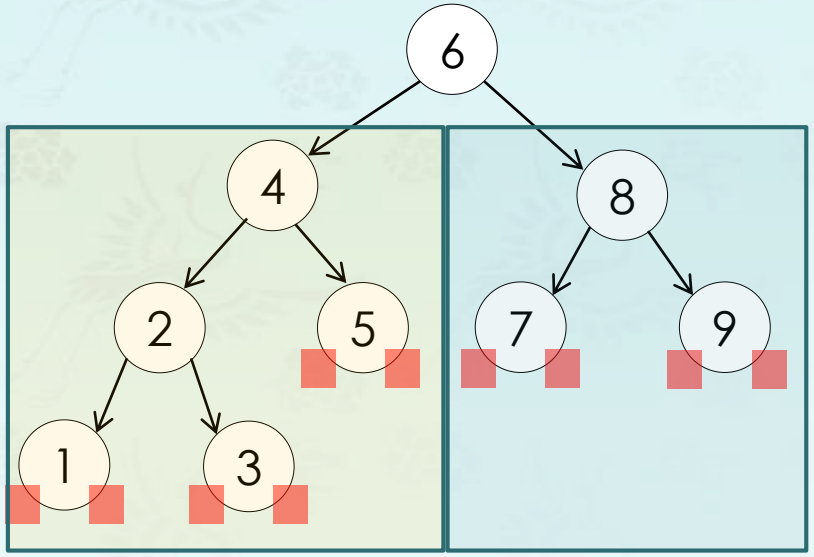
Data Structures
Chapter 5 Tree

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

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

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
```



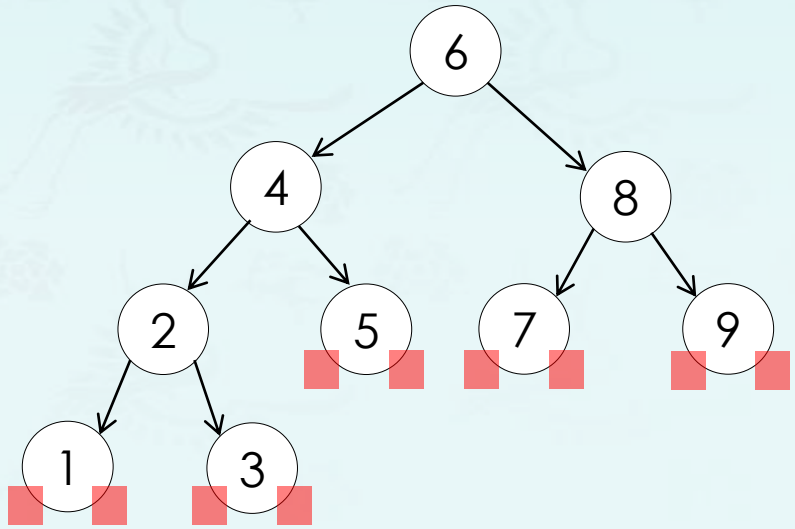
- **Hint:**
  Trace the return value of `maximumBT()` at the leaf.

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.

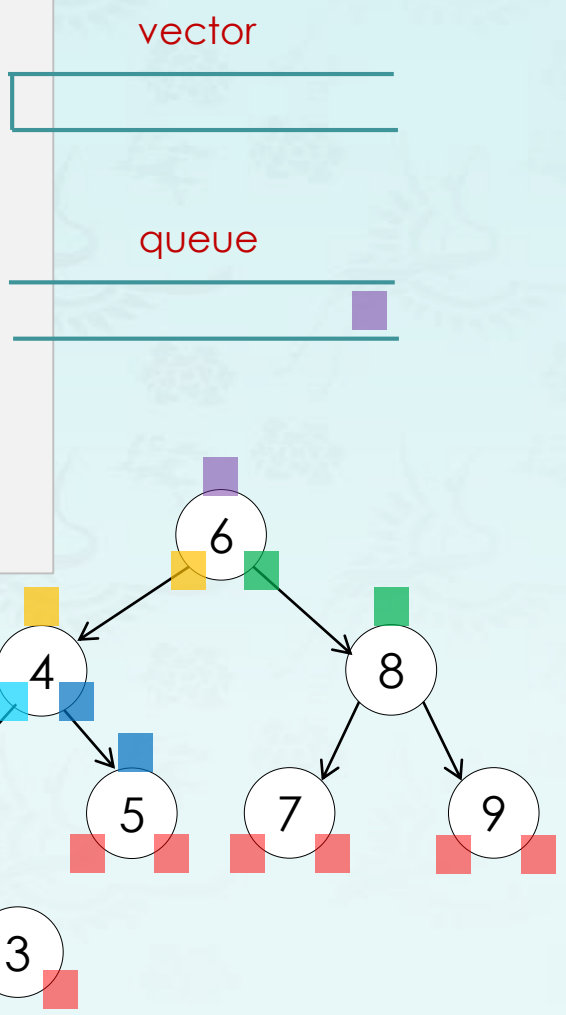
Algorithm (Iteration):

- 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.



Operations: levelorder()

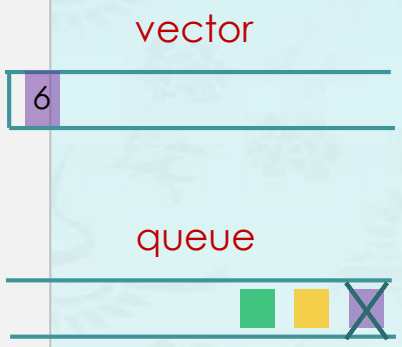
```
// level order traversal of a given binary tree using iteration.

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
    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.
```

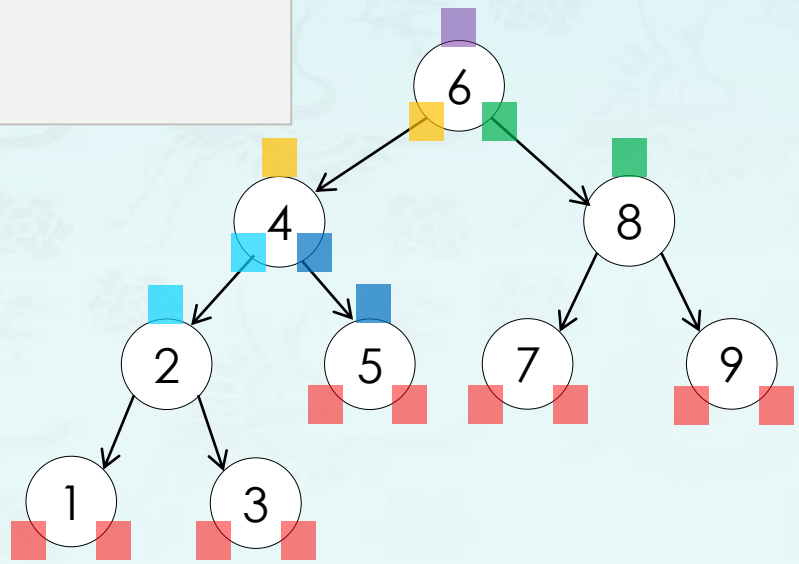


// https://en.wikipedia.org/wiki/Tree_traversal

Operations: levelorder()

```
// 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.
```

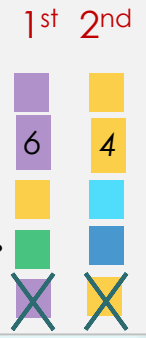


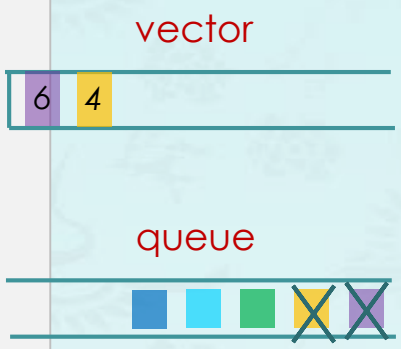


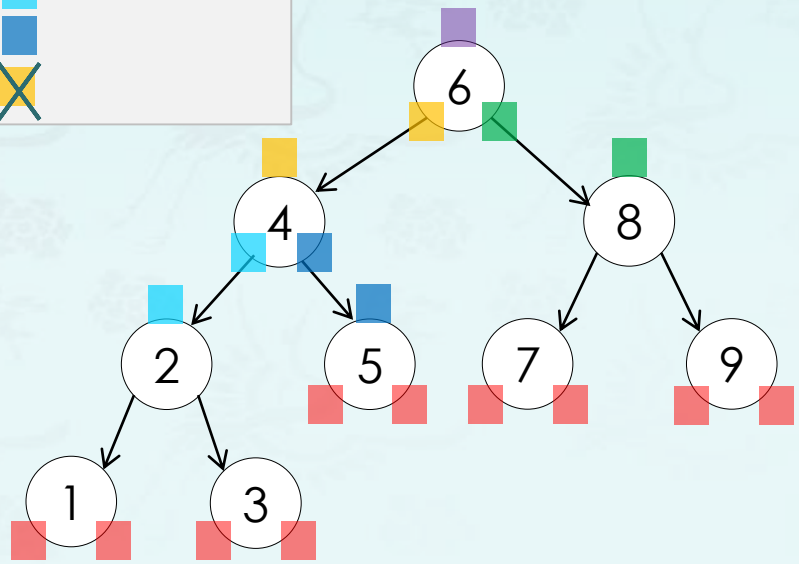
// https://en.wikipedia.org/wiki/Tree_traversal

Operations: levelorder()

```
// 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.
```



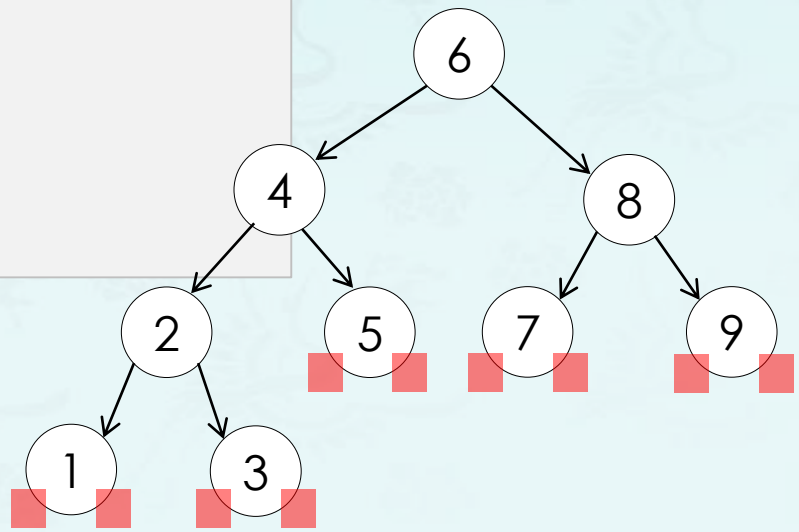




// https://en.wikipedia.org/wiki/Tree_traversal

Operations: levelorder()

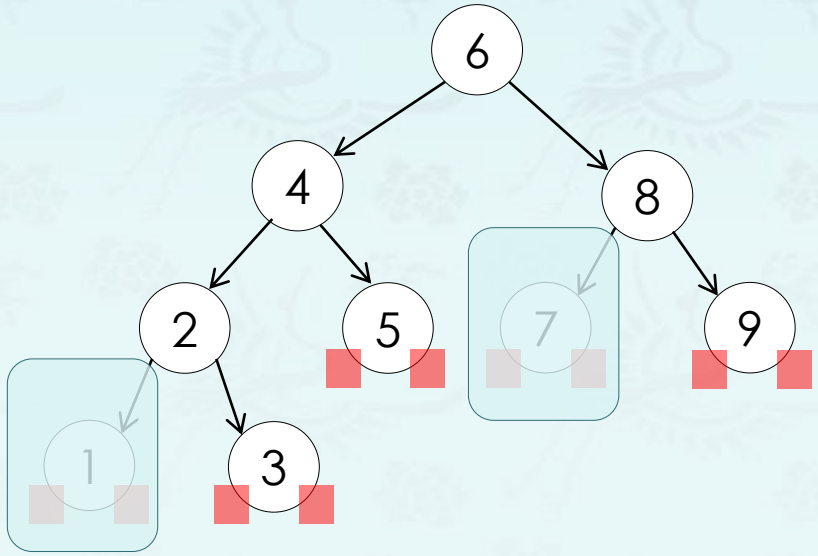
```
// level order traversal of a given binary tree using iteration.
#include <queue>
#include <vector>

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.
```

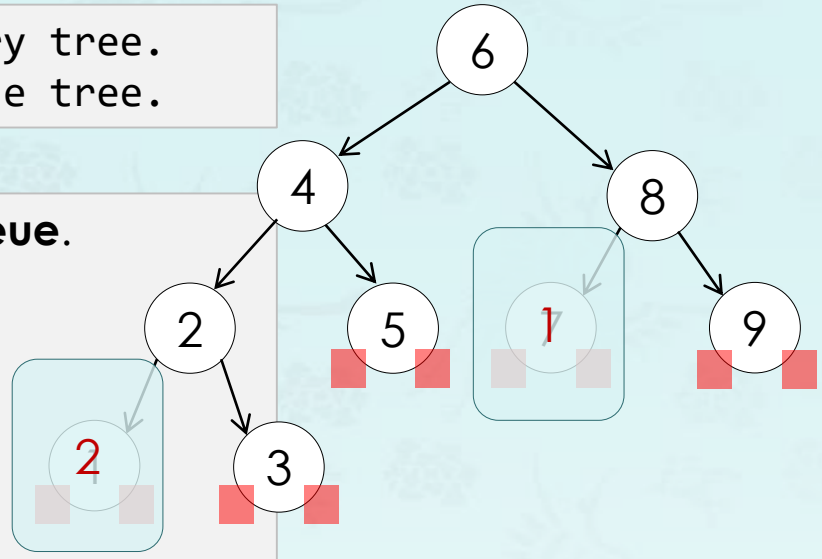


Operations: Grow a node by level order

```
// 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.

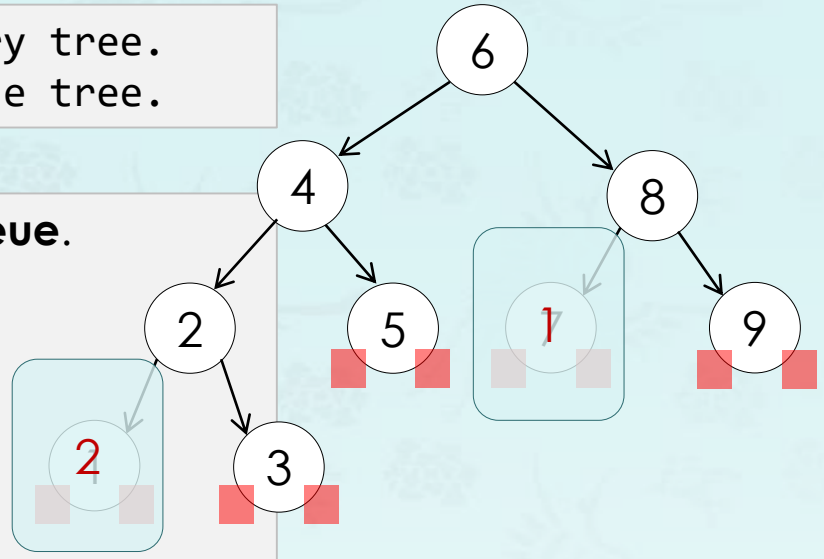


Operations: Grow a node by level order

```
// 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**.

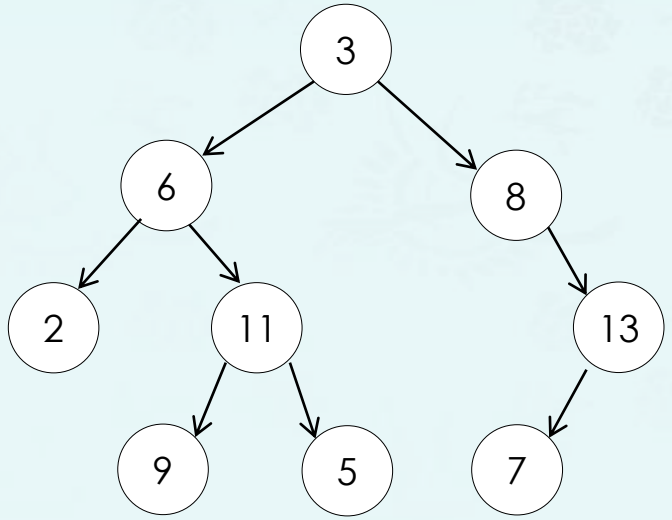
First, push the root to the queue.
Then, while the queue is not empty,
  Get the front() node on the queue
  If the left child of the node is empty,
    make new key as left child of the node. – break and return;
  else
    add it to queue to process later since it is not nullptr.
  If the right child 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.




```
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
}
```

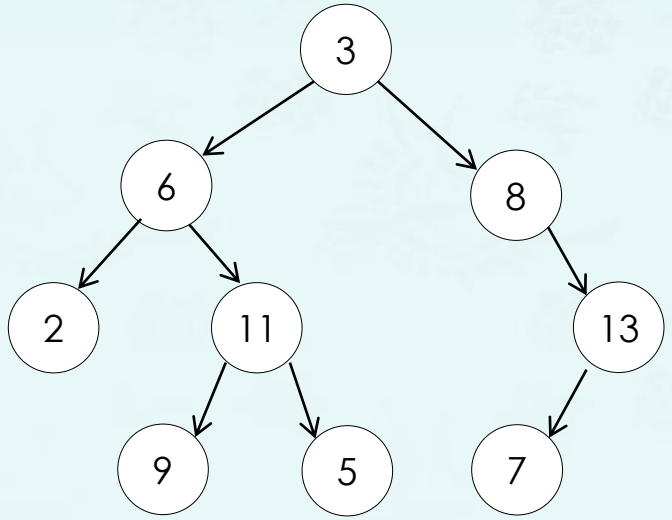
Operations: Path from root to a node in BT

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



Operations: Path from root to a node in BT

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



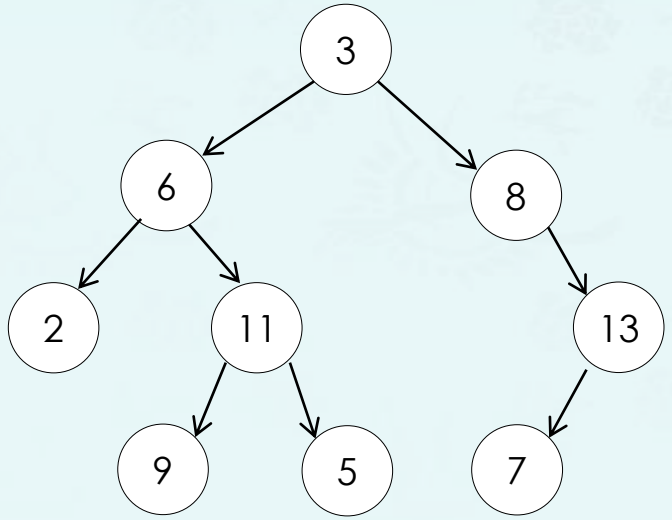
Path from root to a node

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

Operations: Path from root to a node in BT

- 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

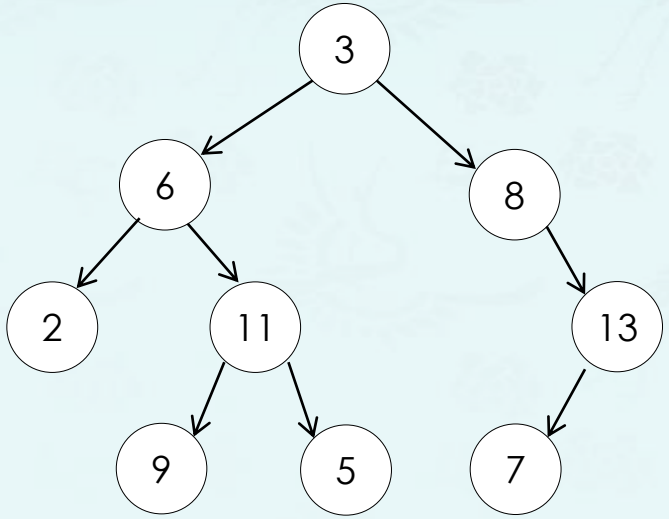



```
bool findPath(tree root, tree x, vector<int>& path)
```

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

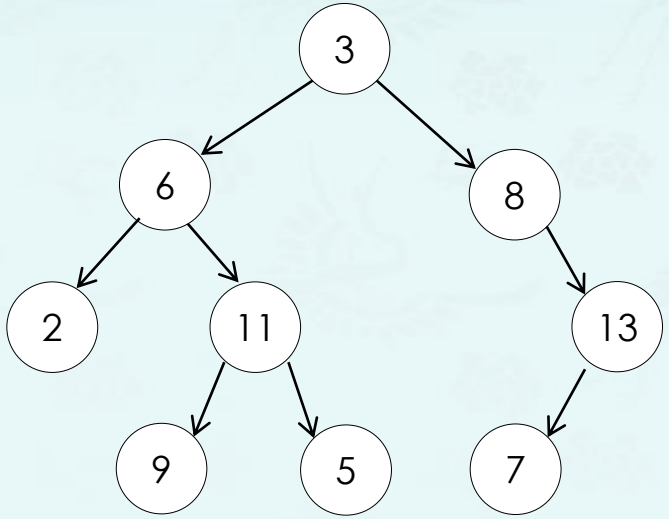
Operations: LCA (Lowest Common Ancestor in BT)

- 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.



Operations: LCA (Lowest Common Ancestor in BT)

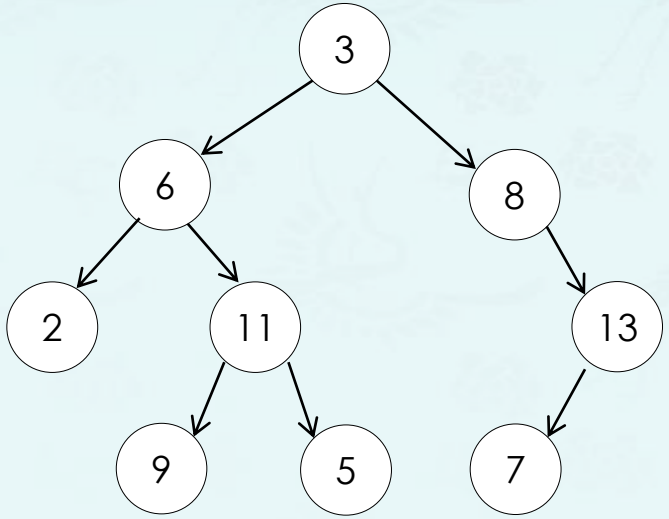
- Find the lowest common ancestor(LCA) of two given nodes, **given in a binary tree.**
  - The LCA is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself)."
  - Two nodes given, p and q, are different and both 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)

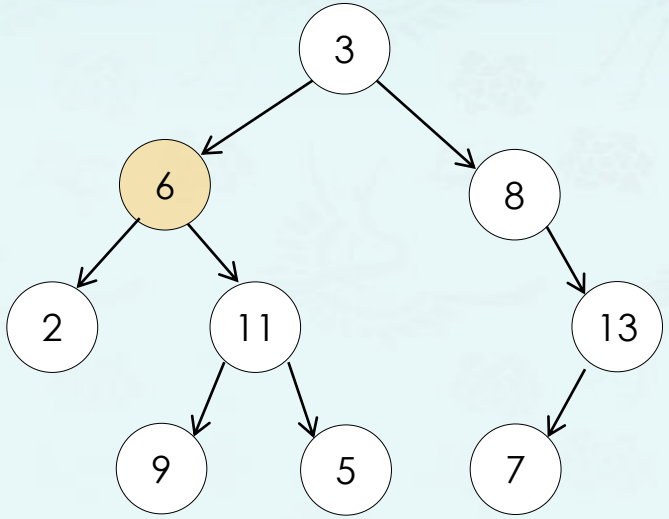
- **Intuition (Iteration):** A brute-force approach is to traverse the tree and get the path to node p and q. Compare the path and return the last match node of the path.
- **Algorithm (Iteration):**




```
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. `use findPath(root, p, path_vector)`
  - Find path from root to q and store it in another vector. `use findPath(root, q, path_vector)`
  - Traverse both paths till the values in vector are same. Return the common element just before the mismatch.



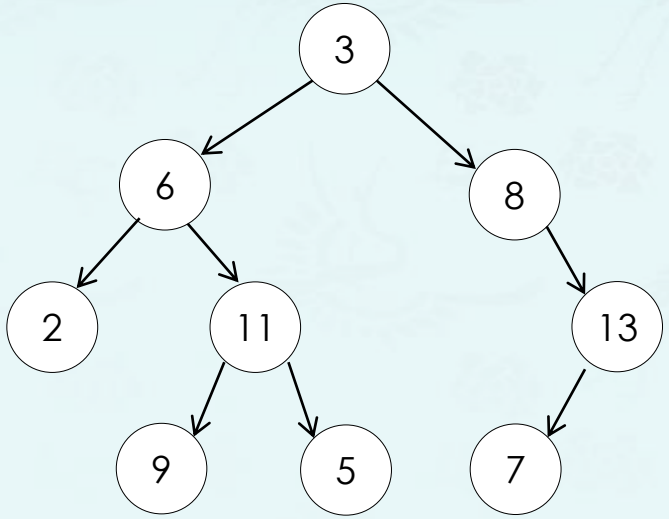

```
For example:
2, 8 -> 3
2, 5 -> 6
9, 5 -> 11
8, 7 -> 8
9, 3 -> 3
```

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

LCA(2, 5) = 6

Operations: LCA (Lowest Common Ancestor in BT)

```
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

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):**

Operations: LCA (Lowest Common Ancestor in BT) Recursion

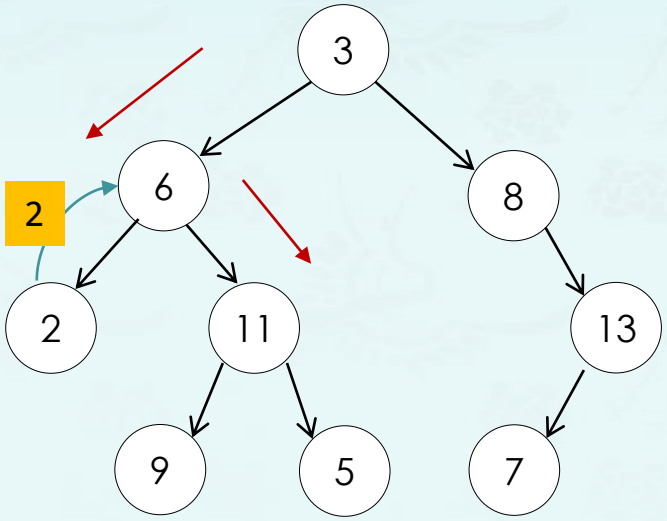
- **Intuition (Recursion):** Traverse the tree in a depth-first manner.
  - The moment you encounter either of the nodes p or q, return the node.
  - The LCA would then be the node for **which both the subtree recursions return a non-NULL node.** It can also be the node which itself is one of p or q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

Operations: LCA (Lowest Common Ancestor in BT) Recursion

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

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

```
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;
}
```



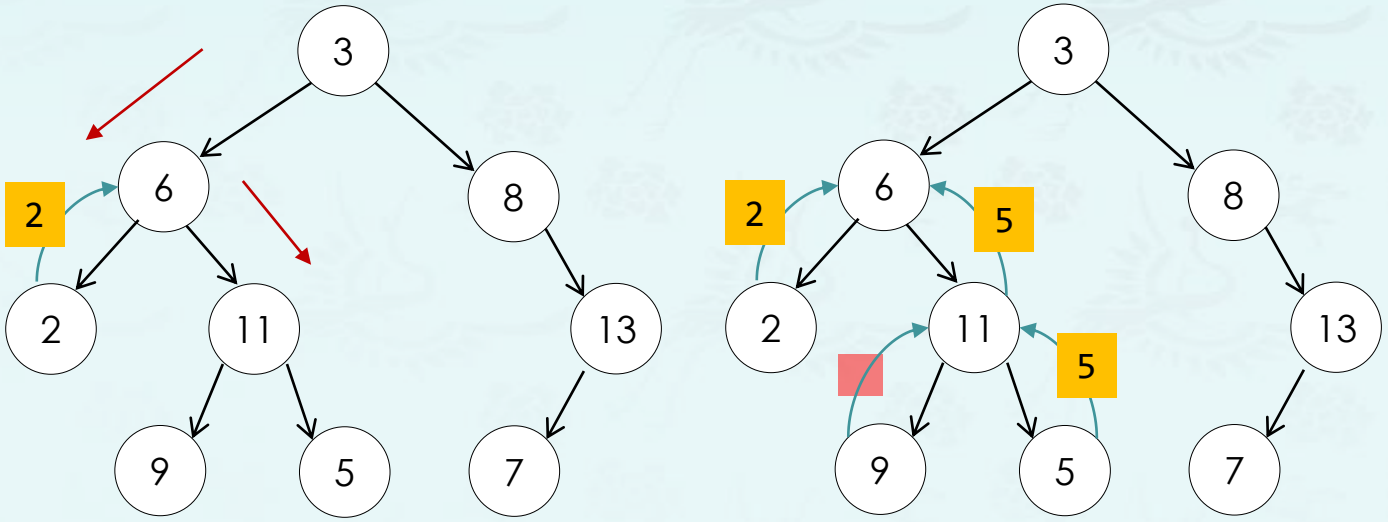
https://www.youtube.com/watch?v=13m9ZCB8gjw
https://algorithmsandme.com/lowest-common-ancestor-in-binary-tree/

Operations: LCA (Lowest Common Ancestor in BT) Recursion

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

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

```
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;
}
```

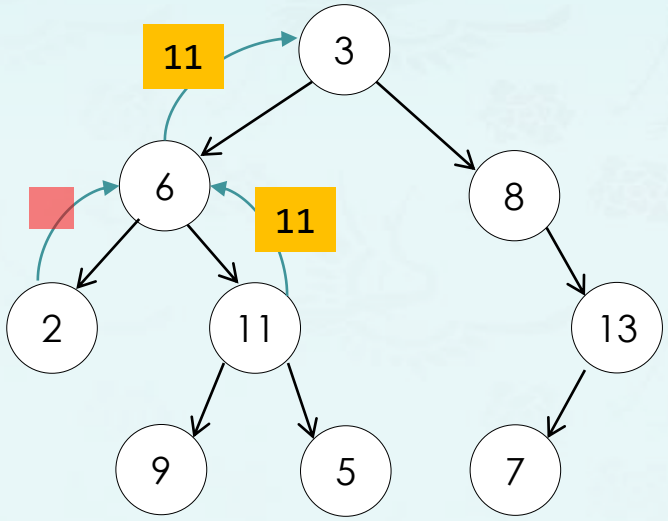


Operations: LCA (Lowest Common Ancestor in BT) Recursion

```
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;
}
```

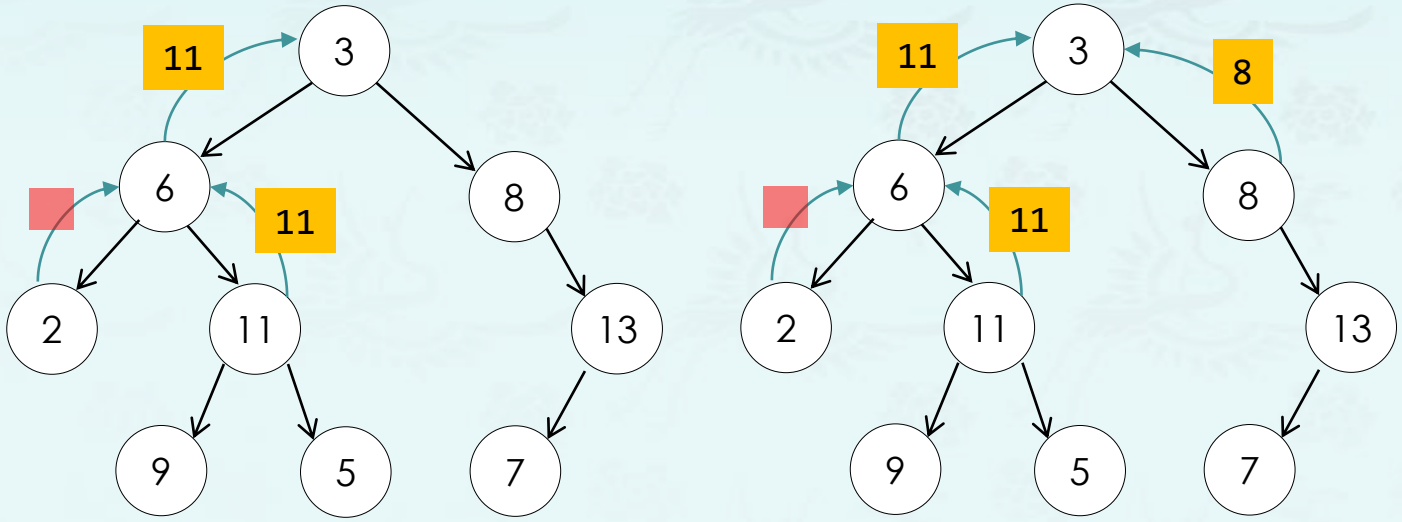


Operations: LCA (Lowest Common Ancestor in BT) Recursion

```
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;
}
```

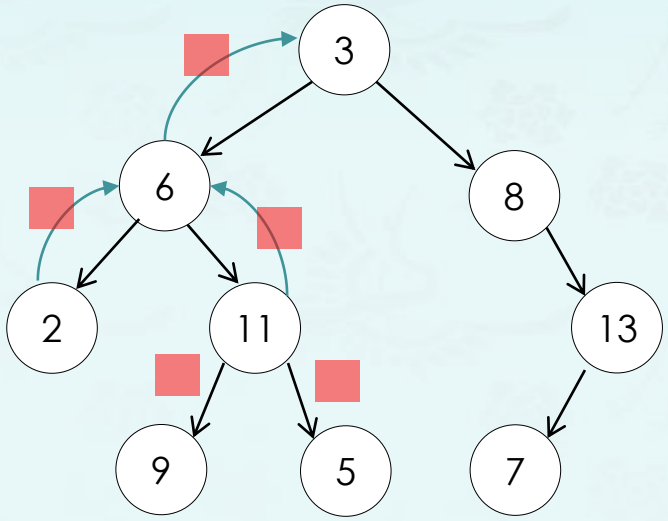


Operations: LCA (Lowest Common Ancestor in BT) Recursion

```
8, 7 -> 8
```

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

```
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;
}
```

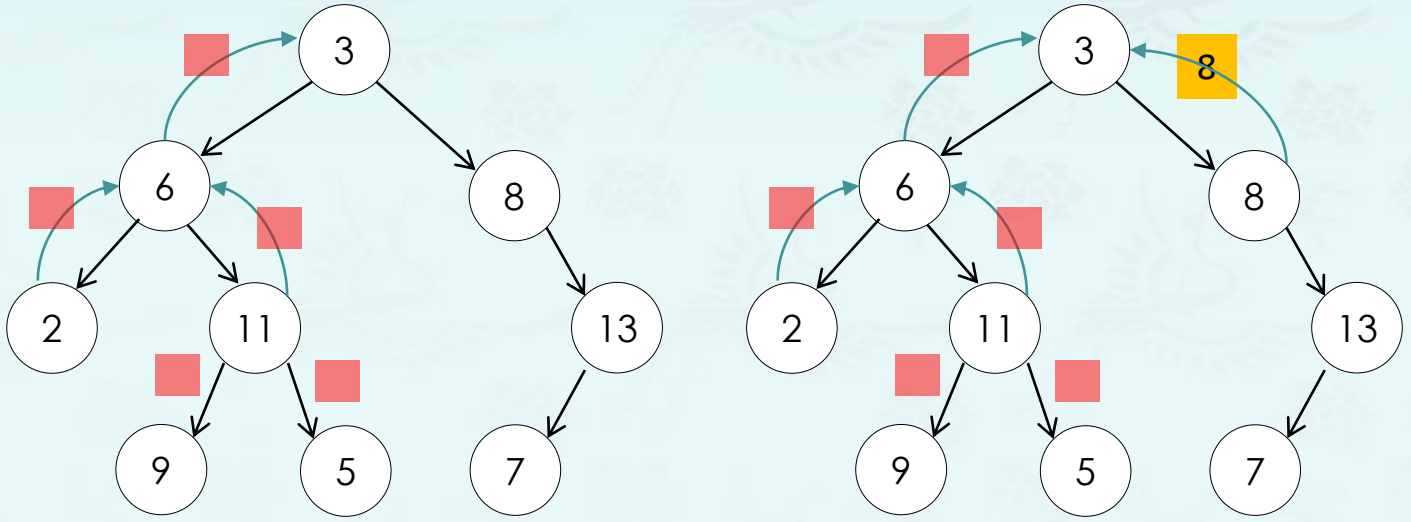


Operations: LCA (Lowest Common Ancestor in BT) Recursion

```
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, 7 -> 8`

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



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