

Data Structures

Chapter 5 Tree

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

사람아
주께서 산한 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)

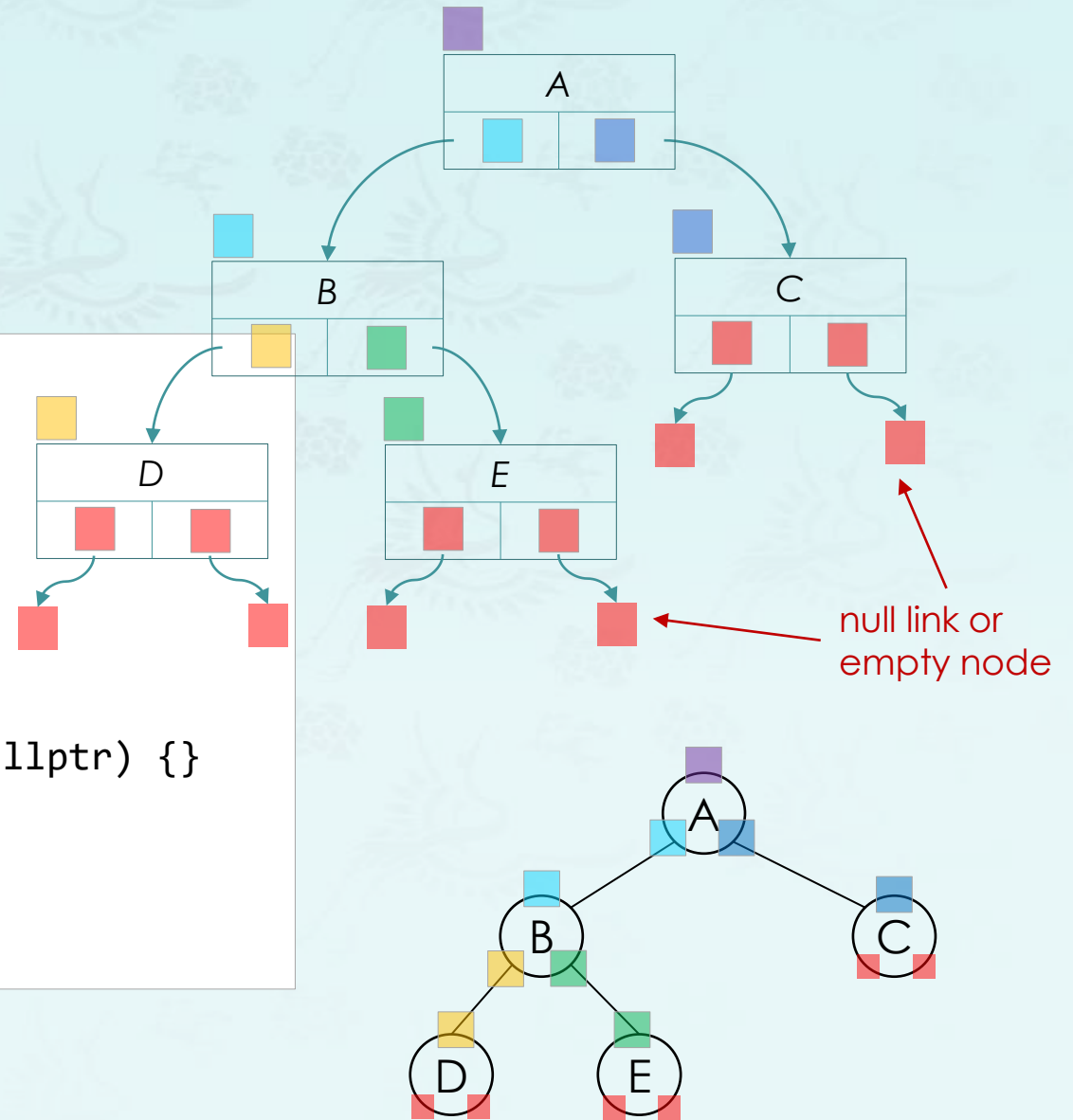
Recursion & Tree Structure

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

```
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) {}

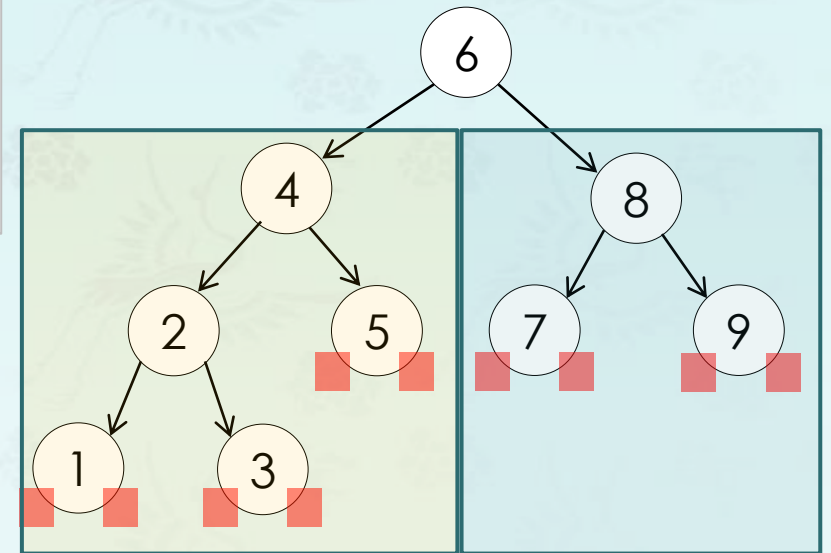
    ~TreeNode(){}
};
using tree = TreeNode*;
```



Operations: 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.

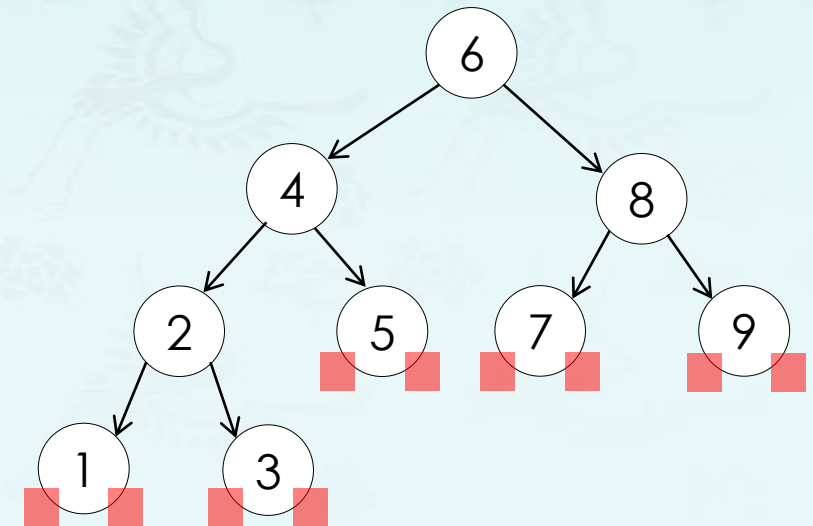


Operations: levelorder()

- 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.
```

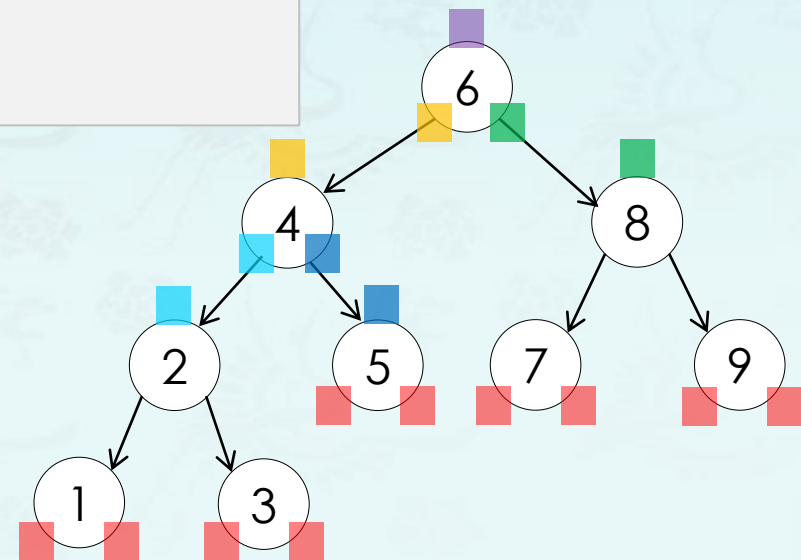
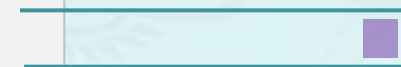
```
void levelorder(tree root, vector<int>& vec)
```

- 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.

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
 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.

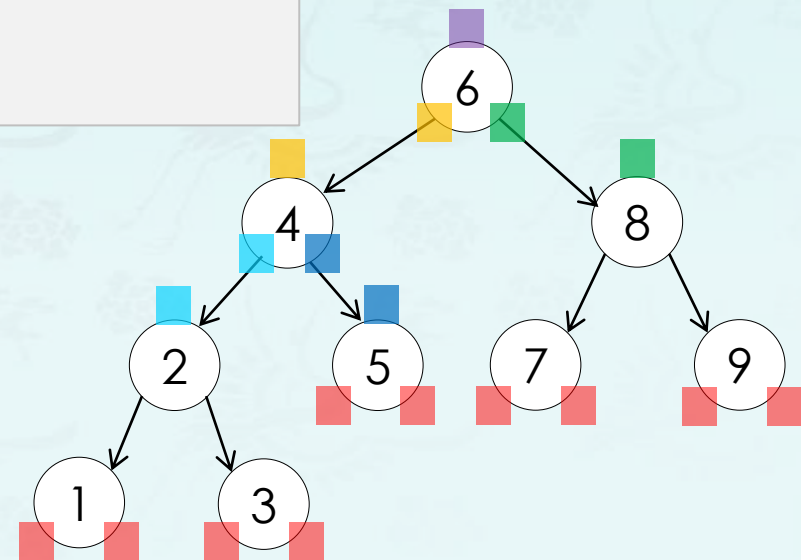
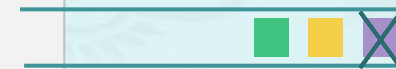
1st 2nd



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)
```

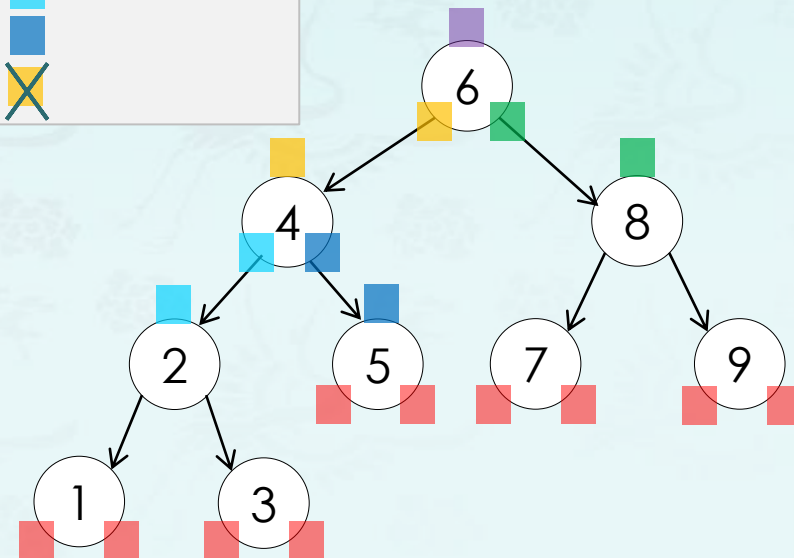
- Visit the root. ("visit" means "Current node to process")
 - if it is not null, push it. 
- while queue is not empty
 1. queue.front() - get the node from the queue
 2. visit the node (save the key in vec).
 3. if its left child is not null, push it to queue.
 4. if its right child is not null, push it to queue.
 5. queue.pop() - remove the node in the queue.



vector



queue



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

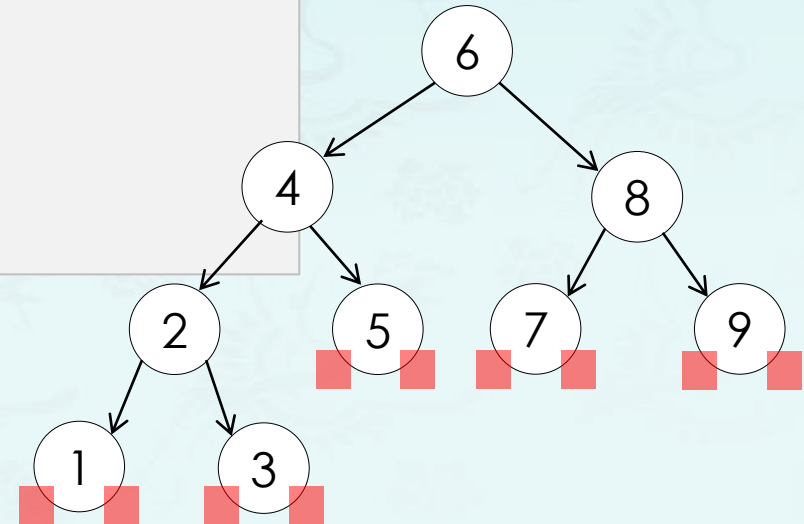
Operations: levelorder()

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

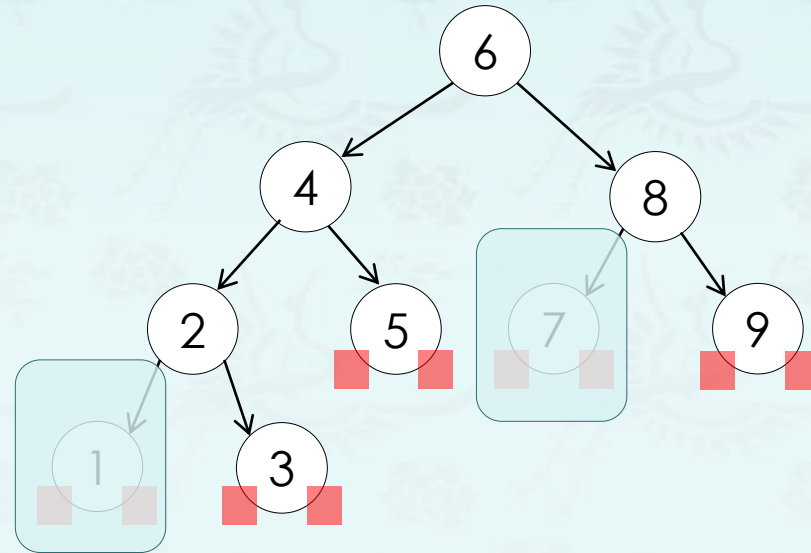
    }
}
```



// 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,

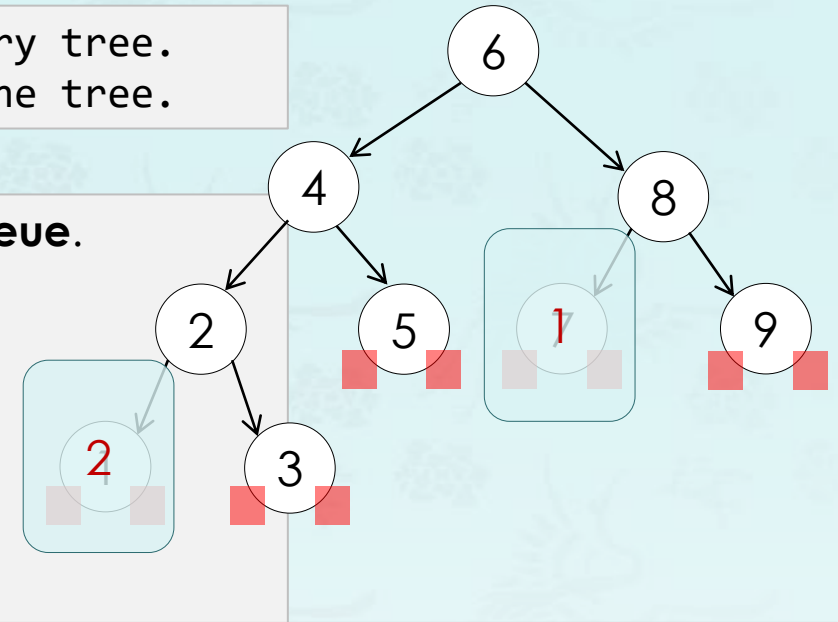
 make new key as right child of the node. – break and return;

 else

 add it to queue to process later since it is not nullptr.

Make sure that you pop the queue finished.

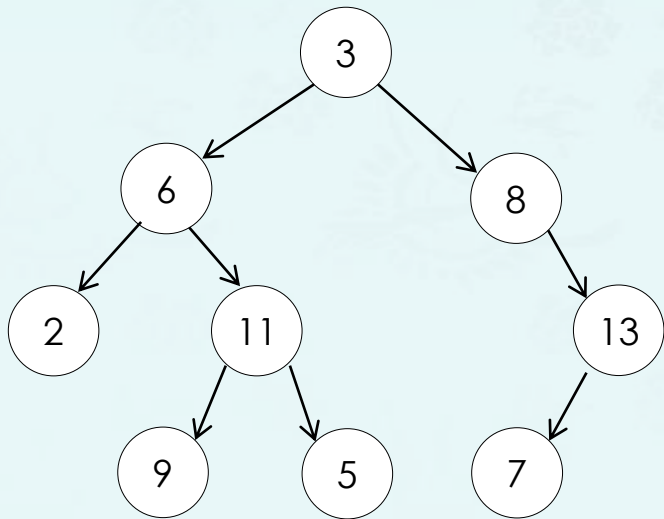
Do this until you find a node whose either left or right is empty.



```
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  
}
```

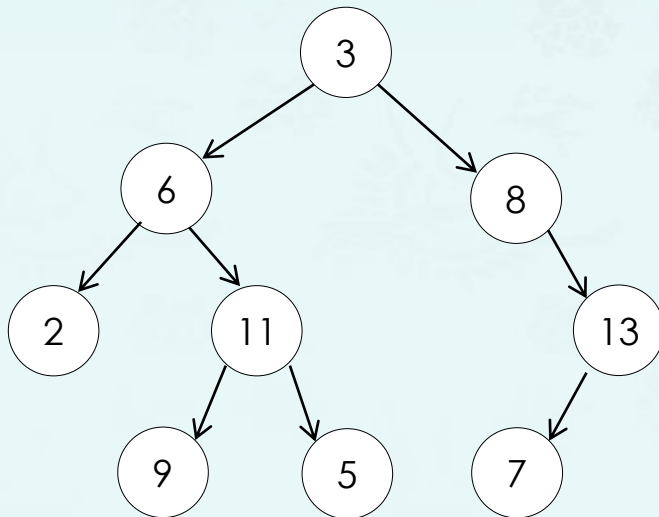
Operations: Path from root to a node in BT

- 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

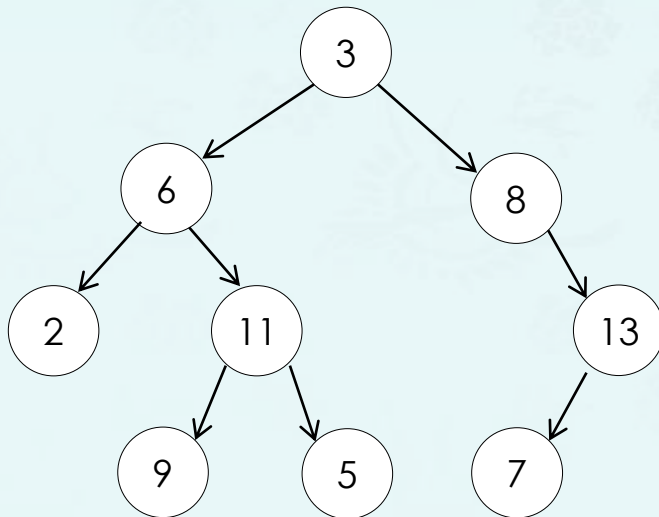
9 -> 3, 6, 11, 9

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



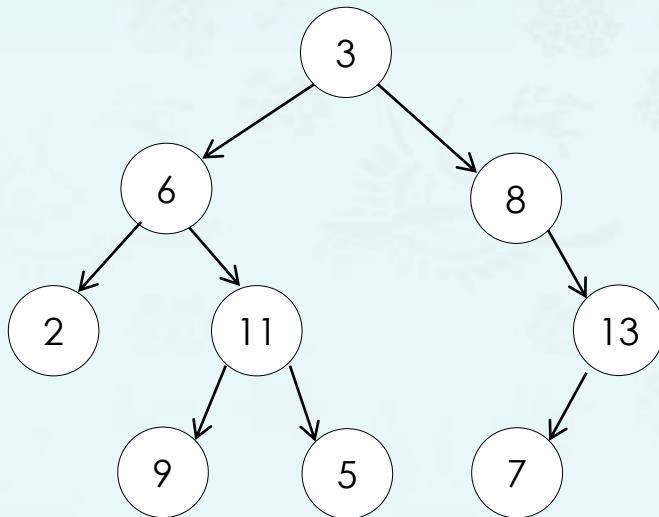
```
bool findPath(tree root, tree x, vector<int>& path)
```

For example:

```
2 -> 3, 6, 2
9 -> 3, 6, 11, 9
13 -> 3, 8, 13
11 -> 3, 6, 11
```

Operations: 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.



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

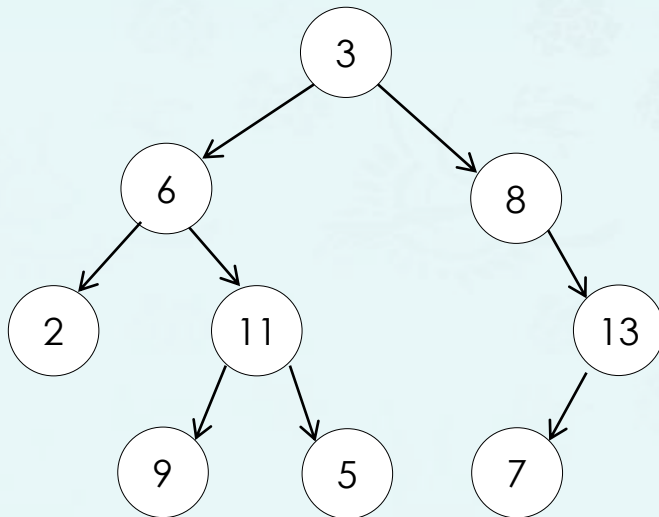
Path from a node to root

For example:

2 -> 2, 6, 3
9 -> 9, 11, 6, 3
13 -> 13, 8, 3
11 -> 11, 6, 3

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)
(Recall what happens after "find()" a node. The return path to the root is saved 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

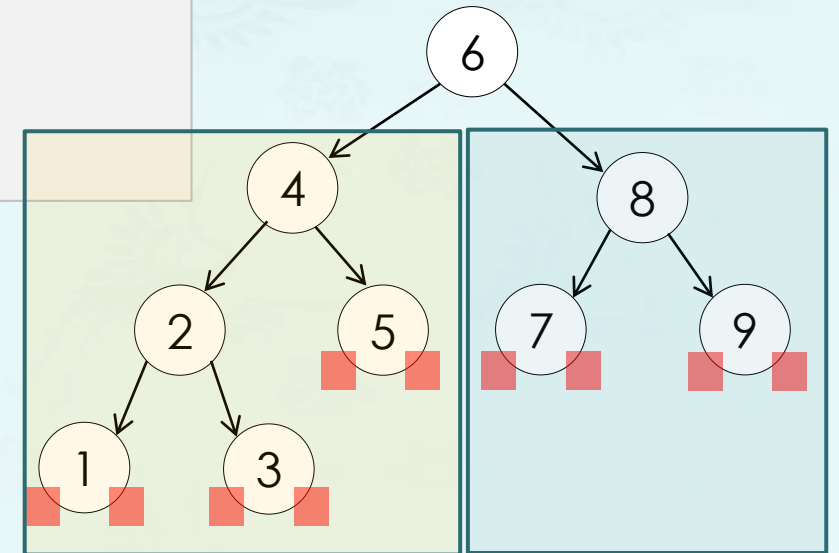
13 -> 13, 8, 3

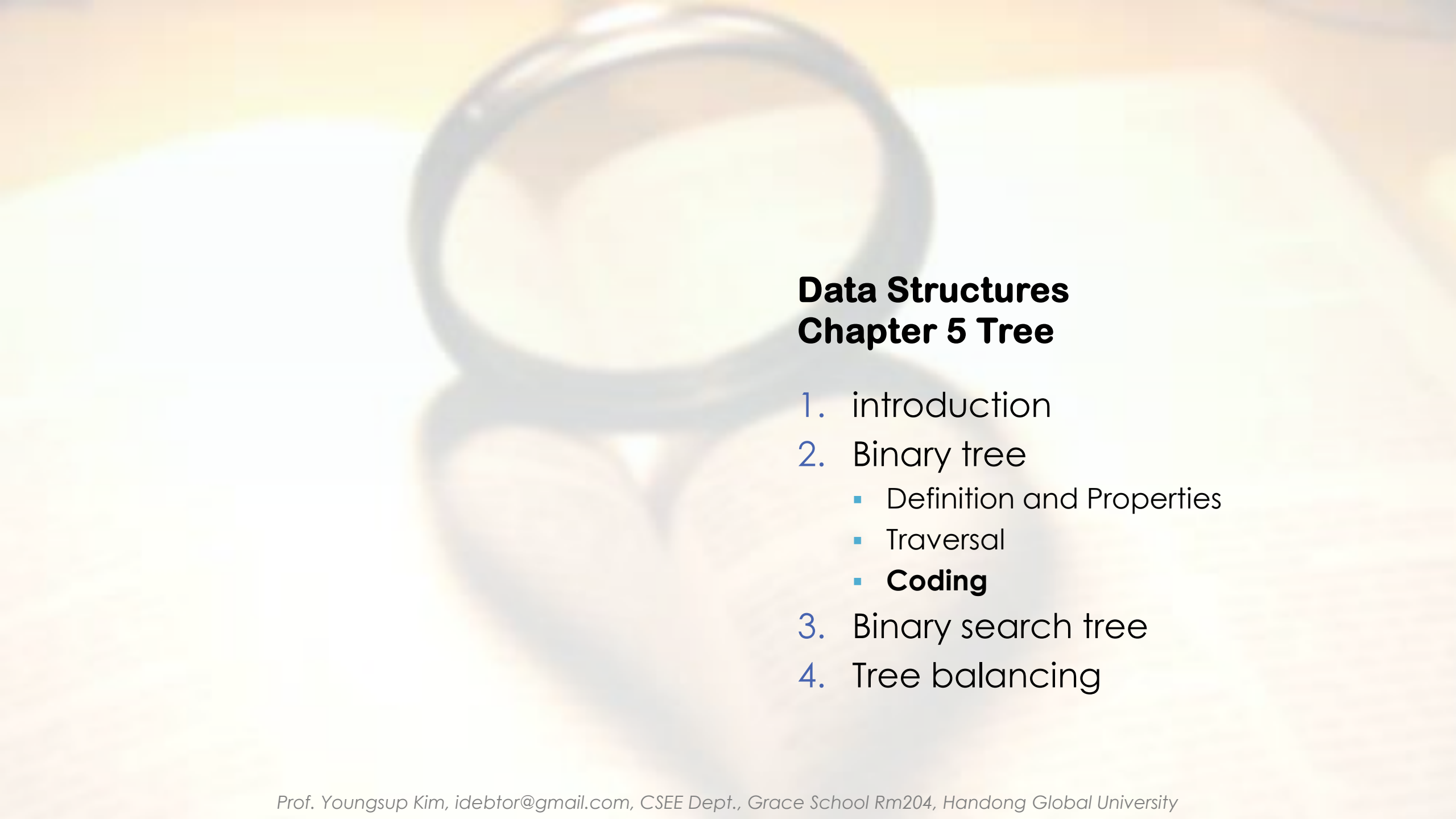
11 -> 11, 6, 3

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;  
}
```





Data Structures

Chapter 5 Tree

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