



Data Structures

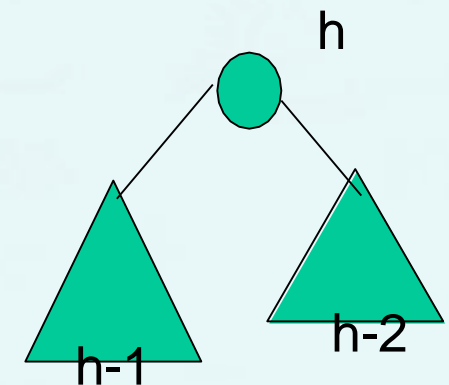
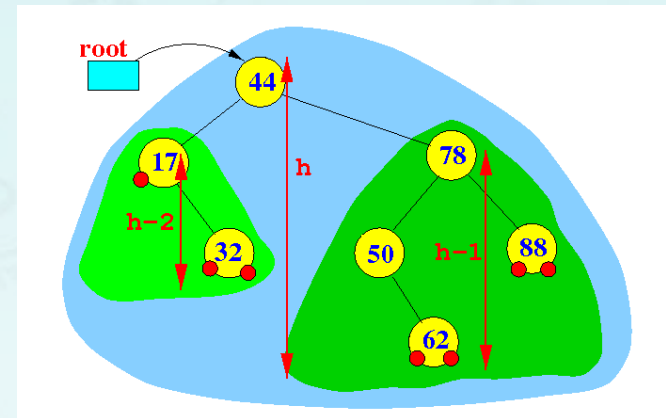
Chapter 5 Tree

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

Height of an AVL Tree

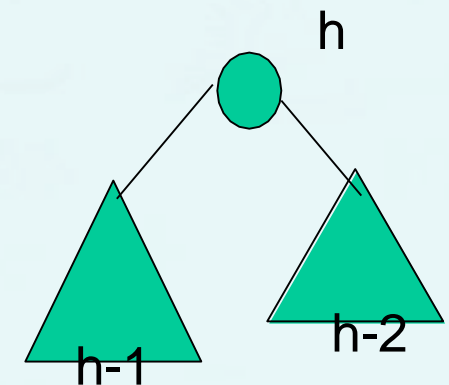
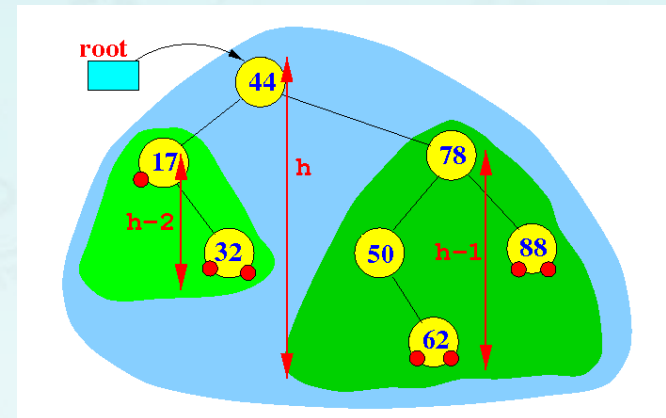
- What is the maximum height of an AVL tree having exactly n nodes?
 - To answer this question, we must ask this question first:
What is the minimum number of nodes (sparsest possible AVL tree) an AVL tree of height h ?
- Consider **the minimum number of nodes** in an AVL tree of height h :
- We can get the recurrence relationship:
$$\begin{aligned}n(0) &= 1 \\n(1) &= 2 \\n(2) &= 4 \\n(h) &= n(h-1) + n(h-2) + 1\end{aligned}$$

where $h > 1$
- This approximate solution of the recurrence is known as $n(h) \cong 1.618^h$



Height of an AVL Tree

- $n(h) = n(h - 1) + n(h - 2) + 1$
where $h > 1$
- This approximate solution of the recurrence is known as $n(h) \cong 1.618^h$
- Solve the equation above for h to get **the max height of an AVL tree** with n nodes?
 $\log_2 n \geq h * \log_2 1.62$
 $h \leq 1/\log_2 1.618 * \log_2 n$
 $h \leq 1.44 * \log_2 n$

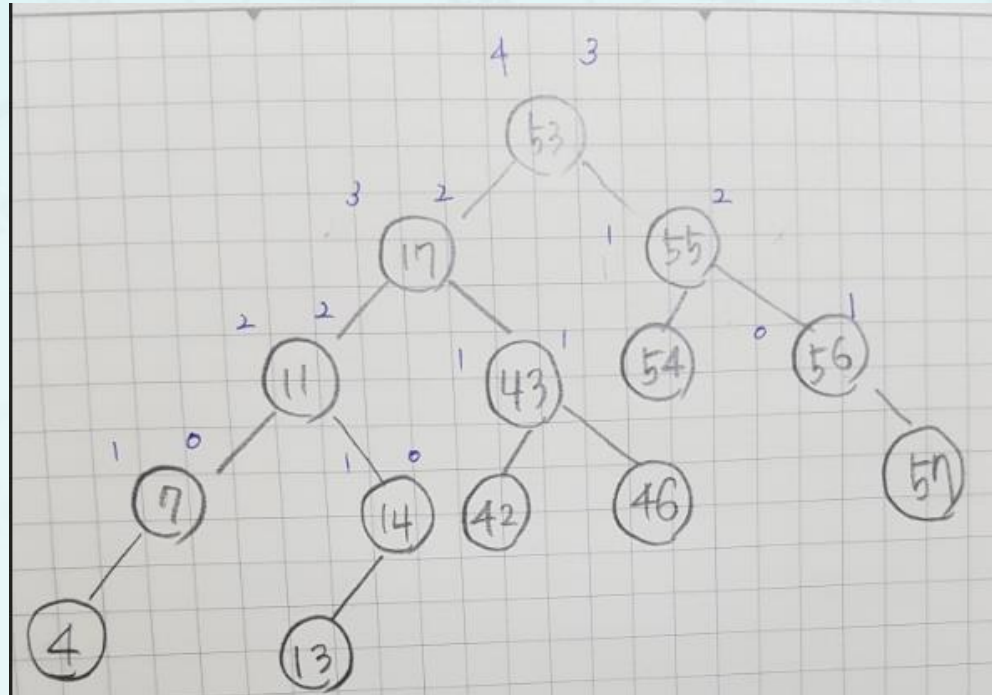


Height of an AVL Tree

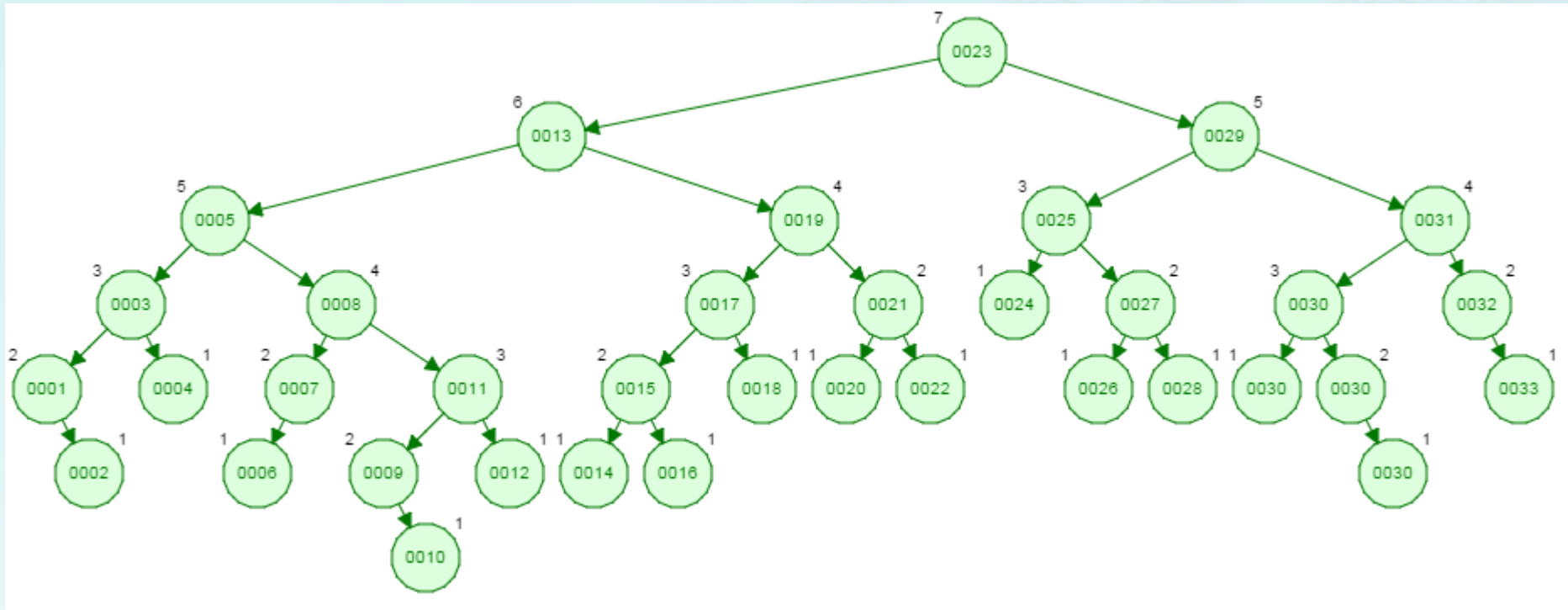
- AVL trees are binary search trees that balances itself every time an element is **inserted or deleted**. **Each node** of an AVL tree has the property that the heights of the sub-tree rooted at its children differ **by at most one**.
- If there are n nodes in AVL tree, minimum height of AVL tree is $\text{floor}(\log_2 n)$.
- If there are n nodes in AVL tree, maximum height can't exceed $1.44 * \log_2 n$.
- If height of AVL tree is h , maximum number of nodes can be $2^{h+1} - 1$.
- Minimum number of nodes in a tree with height h can be represented as:
 $N(h) = N(h-1) + N(h-2) + 1$, where $N(0) = 1$ and $N(1) = 2$.
- The complexity of searching, inserting and deletion in AVL tree is $O(\log_2 n)$.
- The cost of balancing AVL tree is $O(1)$.
What is the time complexity of adding N elements to an empty AVL tree?
Time complexity: $\log(1) + \log(2) + \dots + \log(n) \leq \log(n) + \log(n) + \dots + \log(n) = n \log(n)$

이것도 AVL tree가 될 수 있을까요?

- 저는 AVL tree가 모든 노드의 왼쪽과 오른쪽의 height의 차이가 절대값 1을 넘어서지 않는 것이라고 알고있습니다. 근데 이 트리는 모든 노드에서 왼쪽과 오른쪽의 height의 차이가 절대값 1을 넘지는 않지만 55-54가 연결되어 있는 부분의 높이가 다른 쪽에 비해 2이상 차이 나는 것을 보았습니다. 제가 아는 정의상으로는 AVL tree인거 같으면서도 저렇게 height가 2이상 차이가 나니... 결론을 내릴 수가 없어 질문 드립니다.
- 제가 AVL tree의 정의를 잘못 알고 있는건가요?



Example with leaf 24 on level 3 and leaf 10 on level 6:



- AVL maintain the maximum height difference of 1 between two children subtree, not any two leaves.
- The difference in levels of any two leaves can be any value!
The definition of AVL describes height difference only on two sub-trees from one node.

<https://stackoverflow.com/questions/28964971/height-difference-between-leaves-in-an-avl-tree>

growAVL() & trimAVL()

```
// inserts a key into the AVL tree and rebalance it.
tree growAVL(tree node, int key) {
    if (node == nullptr) return new TreeNode(key);

    // your code here ← almost same as grow()

    return rebalance(node);    // O(log n)
}
```

AVL rotation if necessary

```
// deletes a key into the AVL tree and rebalance it.
tree trimAVL(tree node, int key) {
    if (node == nullptr) return new TreeNode(key);

    // your code here ← almost same as trim()

    return rebalance(node);    // O(log n)
}
```

AVL rotation if necessary

growN() & TrimN()

```
// removes randomly N numbers of nodes in the tree(AVL or BST).  
// It gets N node keys from the tree, trim one by one randomly.  
tree trimN(tree root, int N, bool AVLtree) { // testing purpose  
    vector<int> vec;  
  
    // your code here  
  
    delete[] arr;  
    return root;  
}
```


growN() & TrimN()

```
tree growN(tree root, int N, bool AVLtree) {    // coding a faster version
    int start = empty(root) ? 0 : value(maximum(root)) + 1;
    int* arr = new (nothrow) int[N];
    assert(arr != nullptr);
    randomN(arr, N, start);

    #if 0    // use BST grow() first. then, if AVLtree, reconstruct it as AVL.
        for (int i = 0; i < N; i++) root = grow(root, arr[i]);
        if (AVLtree) root = reconstruct(root);
    #else // use its own grow() function, respectively. it is too slow
        if (AVLtree)
            for (int i = 0; i < N; i++) root = growAVL(root, arr[i]);
        else
            for (int i = 0; i < N; i++) root = grow(root, arr[i]);
    #endif

    delete[] arr;
    return root;
}
```

Reconstruct() – Building AVL tree from BST in $O(n)$

- **Goal:** Reconstruct a new AVL tree from BST in $O(n)$.
- **Intuition:** Since we can get a sorted key values from the binary search tree, we take advantage of the sorted list to form a well balanced AVL tree faster.
- **Recreation Method**
 - Use an array of keys, using an existing `inorder()` function that returns a sorted keys in vector.
 - Clear the original tree since it is not used any more.
 - Since it goes through the tree twice only, the time complexity is $O(n)$.
- **Recycling Method**
 - Use an array of nodes, simply reconstructs (or relink) the existing nodes.
 - Write a new `inorder()` that returns the sorted nodes of the tree.
 - Since it goes through the tree twice only, the time complexity is $O(n)$.
- **For pedagogical purpose**, let us use the recycling method if the number of nodes are more than 10, otherwise use the recreation method.

Reconstruct() – Building AVL tree from BST in $O(n)$

```
// reconstructs a new AVL tree from BST in  $O(n)$ .
tree reconstruct(tree root) {
    if (root == nullptr) return nullptr;

    if (size(root) > 10) {          // recycling method

        cout << "your code here"

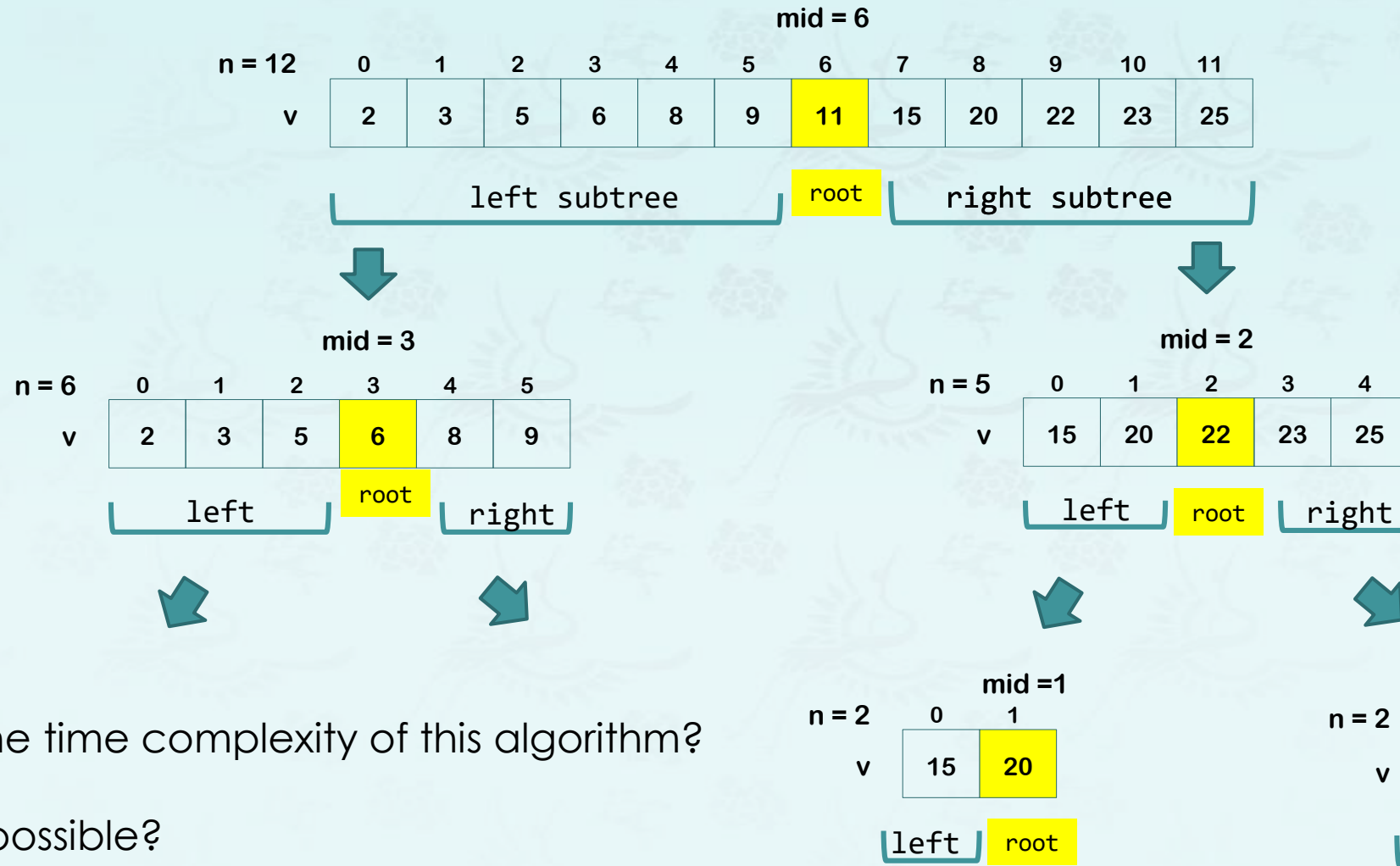
    }
    else {                          // recreation method

        cout << "your code here"

    }
    return root;
}
```



Reconstruct() – Building AVL tree from BST in $O(n)$



What is the time complexity of this algorithm?

$O(n)$

How is it possible?

Building AVL tree from BST in $O(n)$ – recreation method

```
// rebuilds an AVL tree with a list of keys sorted.  
// v - an array of keys sorted, n - the array size  
tree buildAVL(int* v, int n) {  
    if (n <= 0) return nullptr;  
    int mid = n / 2;  
  
    // create a root node  
  
    // recursive buildAVL() calls for left & right, return it to root->left & root->right  
  
    return root;  
}
```



Building AVL tree from BST in $O(n)$ – recycling method

```
// rebuilds an AVL tree using a list of nodes sorted, no memory allocations
// v - an array of nodes sorted, n - the array size
tree buildAVL(tree* v, int n) {
    if (n <= 0) return nullptr;
    int mid = n / 2;

    // v[mid] becomes the root; don't call new TreeNode.

    // recursive buildAVL() calls for left & right, return it to root->left & root->right

    return root;
}
```





Data Structures

Chapter 5 Tree

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