AVL Tree II

- Time complexity
- Reconstruct AVL tree from BST in O(n)
  - rebalanceTree()
  - Use inorder() either keys or nodes
- growN(), trimN()
  - use rebalanceTree() instead of rebalance()

Time Complexity in big O notation:

Algorithm	BST		AVL	
	Worst	Average	Worst	Average
Search	O(n)	O(log n)	O(log n)	O(log n)
Insertion	O(n)	O(log n)	O(log n)	O(log n)
Deletion	O(n)	O(log n)	O(log n)	O(log n)
grow N, trim N	O(n^2)	O(n log n)	O(n log n)	O(n log n)
rebalance()			O(log n)	O(log n)
rebalance N			O(n log n)	O(n log n)

Time Complexity in big O notation:

```
// 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

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

```
tree rebalanceTree(tree node) {
  if (node == nullptr) return nullptr;

  // your code here


  return node;
}
```

Time Complexity in big O notation:

```
// inserts N numbers of keys in the tree(AVL or BST)
// If it is empty, the key values to add ranges from 0 to N-1.
// If it is not empty, it ranges from (max+1) to (max+1 + N).
tree growN(tree root, int N, bool AVLtree) {    // recode tree.cpp
  int start = empty(root) ? 0 : value(maximum(root)) + 1;

  int* arr = new (nothrow) int[N];
  assert(arr != nullptr);
  randomN(arr, N, start);

  for (int i = 0; i < N; i++) root = grow(root, arr[i]);

  if (AVLtree) root = rebalanceTree(root);

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

Use BST grow() **instead of growAVL()**
since AVL is a BST and
we can avoid calling rebalance() N times.

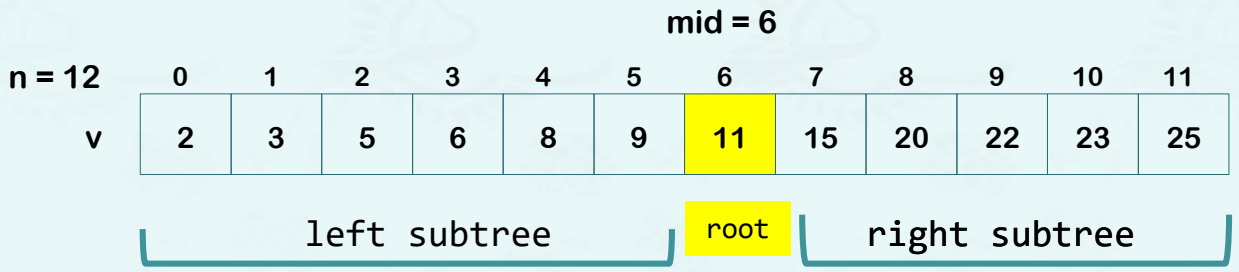
Time Complexity in big O notation:

```
// 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) {  // recode tree.cpp
  vector<int> vec;
  inorder(root, vec);
  shuffle(vec.data(), vec.size());
  int tsize = size(root);
  assert(vec.size() == tsize);   // make sure we've got them all

  int count = N > tsize ? tsize : N;
  for (int i = 0; i < N; i++) root = trim(root, arr[i]);

  if (AVLtree) root = rebalanceTree(root);    // reconstruct AVL tree

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

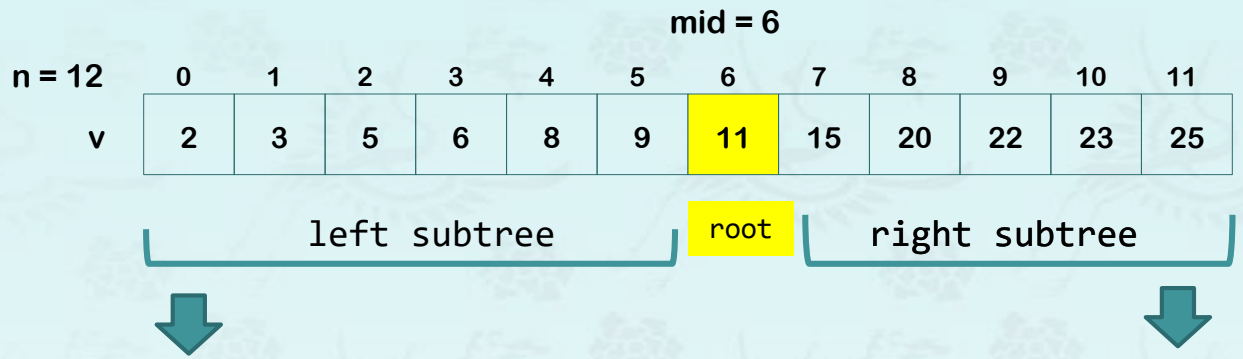
Use BST trim() instead of trimAVL()
since AVL is a BST and
we can avoid calling rebalance() N times.

RebalanceTree() reconstructs AVL from BST in O(n)

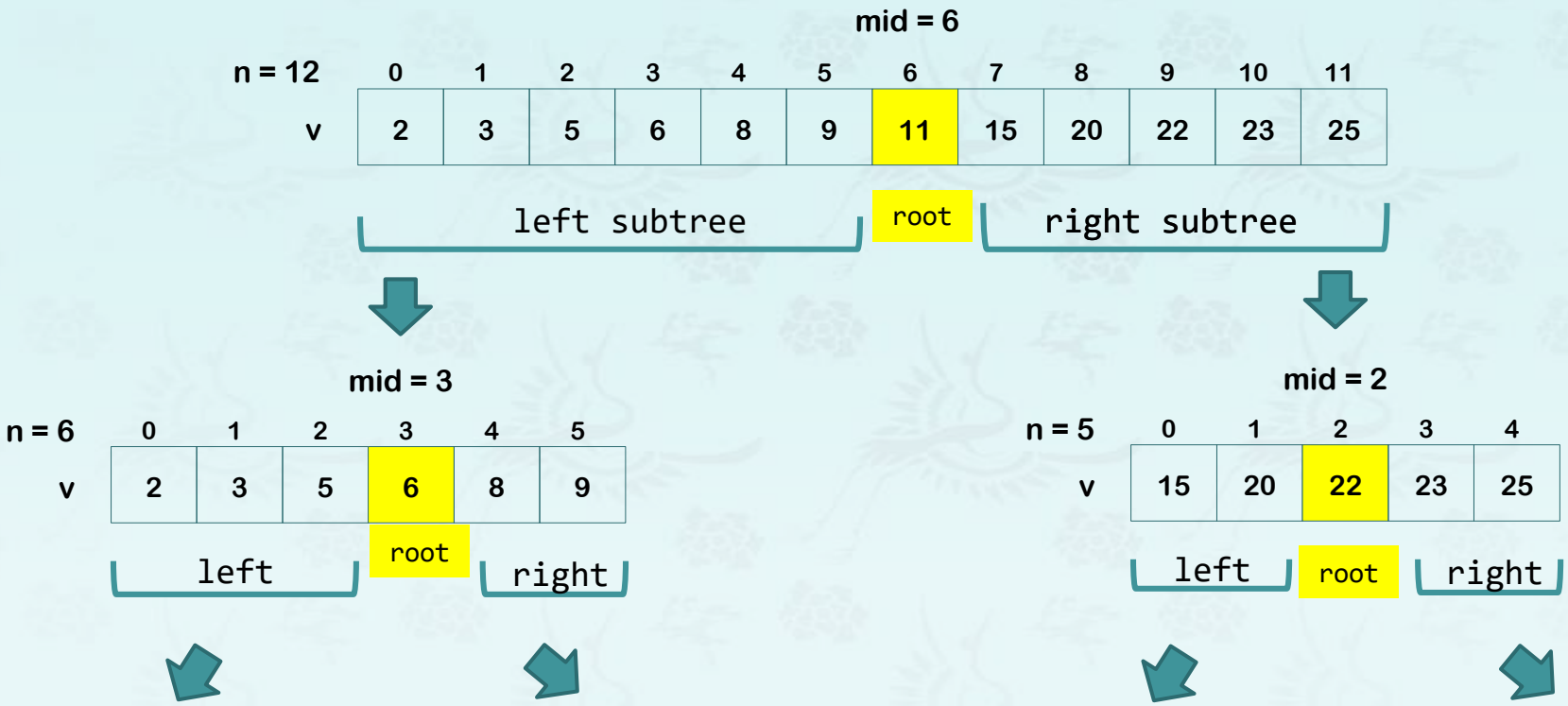
```
// reconstructs a new AVL tree from BST in O(n).
tree rebalanceTree(tree root) {
  if (root == nullptr) return nullptr;

  // Use inorder() to get an array of keys or nodes depending on recreation or recycling
  // recycling method - use an array of nodes, simply reconstructs the existing nodes.
  // recreation method - use an array of keys, clear the original tree before buildAVL()

  // your code here                          // O(n)

  return buildAVL(v.data(), v.size()); // O(n)
}
```



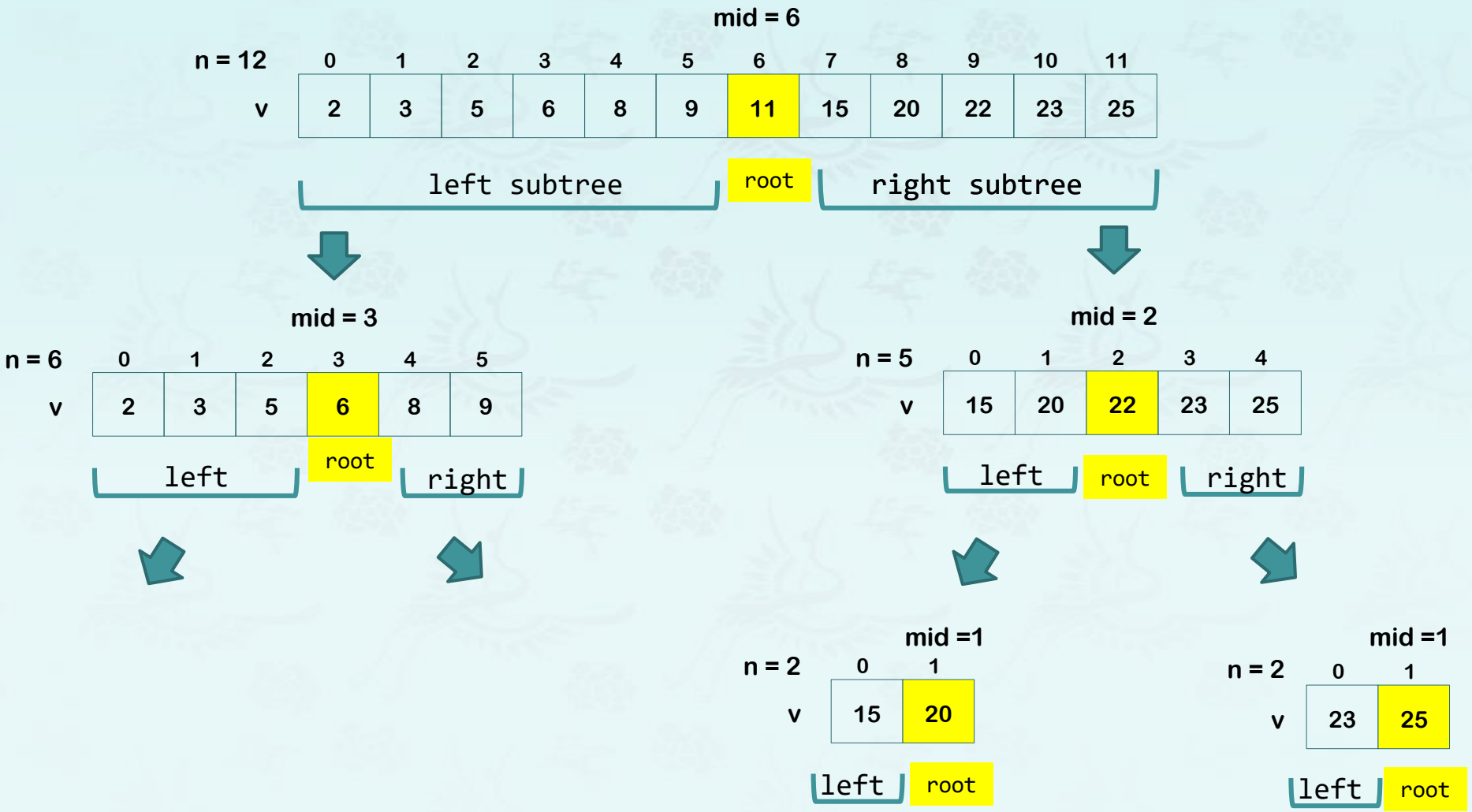
Building AVL tree from BST in O(n)



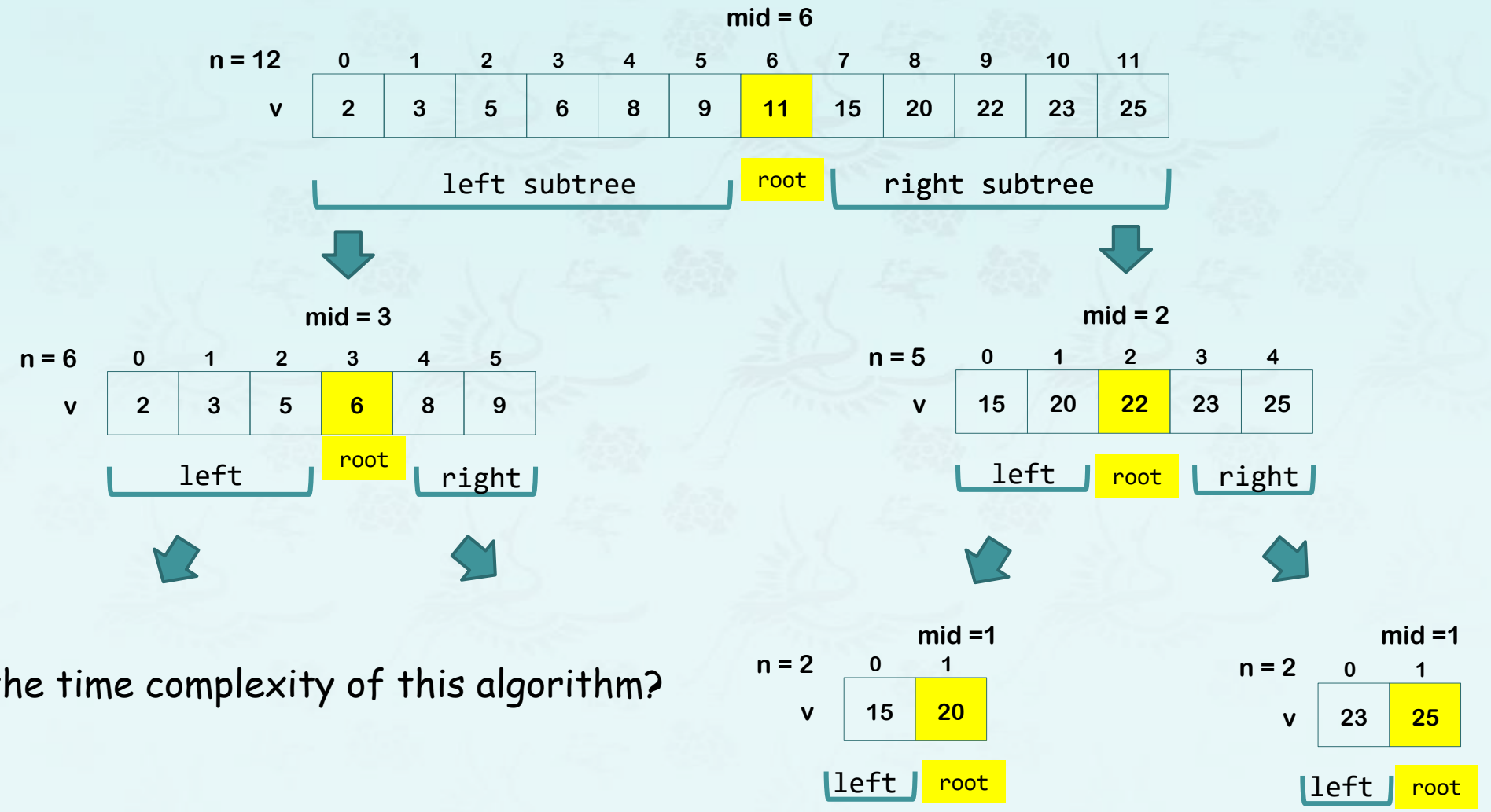
Building AVL tree from BST in O(n)



Building AVL tree from BST in O(n)

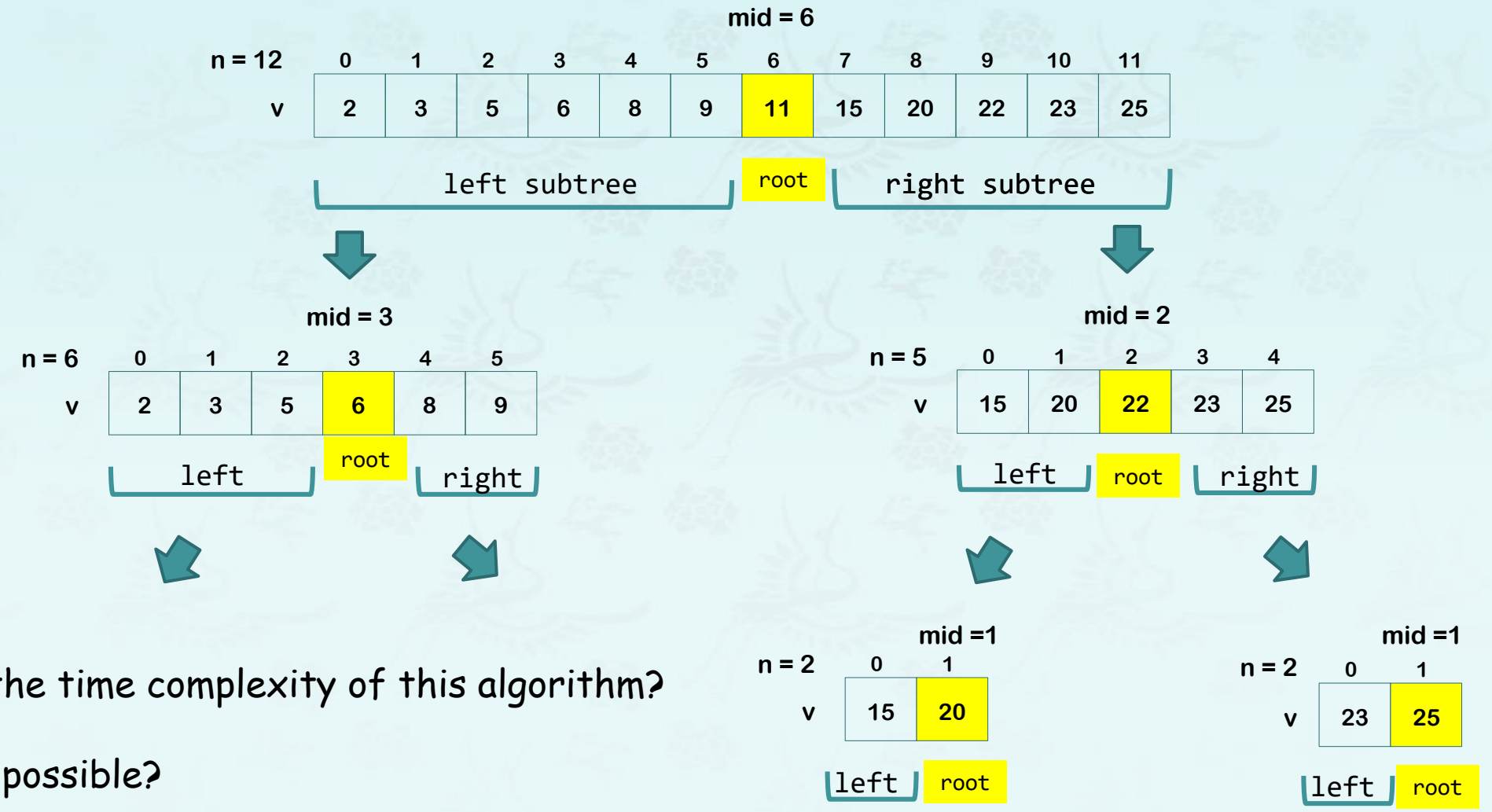


Building AVL tree from BST in O(n)



What is the time complexity of this algorithm?

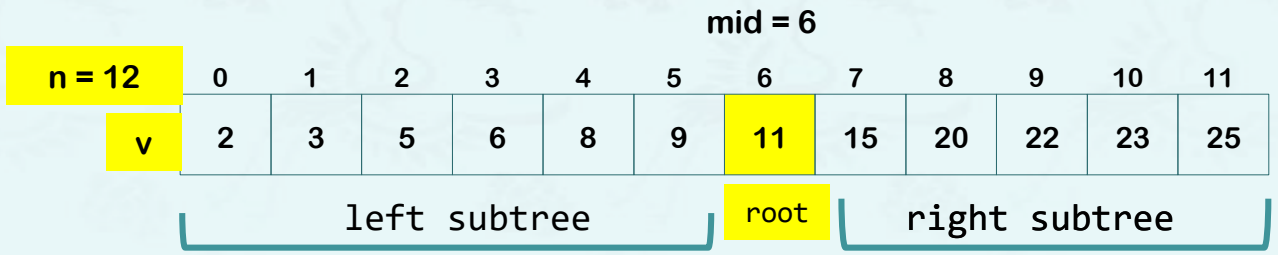
Building AVL tree from BST in O(n)



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;

  tree root = new TreeNode( v[mid] )     // create a root node
  // recursive buildAVL() calls for left & right, return it to root->left & root->right

    root -> left = buildAVL ( v, mid);
    root -> right = buildAVL ( v[mid+1], n-mid)

  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;

  tree root =                                  // mid becomes the root; don't call new TreeNode.
    root -> left = nullptr;                    // set leaf nodes to null for recycling.
    root -> right = nullptr;
  // recursive buildAVL() calls for left & right, return it to root->left & root->right



  return root;
}
```